Mémoire de stage effectué du 08/02/2021 au 07/07/2021

Dans :

L'équipe GraphIK

Centre de recherche : LIRMM & Inria

à Montpellier

Extension de la plateforme Graal pour le raisonnement numérique

Auteur : Riadh GUEMACHE

Étudiant en : 2ème année de master

Spécialité : DECOL(données,connaissances et langage naturel)

À l'université de Montpellier

Faculté Des Sciences

Département Informatique

Tuteurs

De l'entreprise : Jean-François Baget

Marie-Laure Mugnier

De l'université : : Marianne Huchard

2 juillet 2021

Remerciements

Avant tout, il me semble judicieux de commencer ce mémoire de stage de fin d'étude par des remerciements, à ceux qui m'ont beaucoup aidé durant ce stage ainsi qu'à ceux qui ont fait de mon stage un moment agréable et profitable.

Je tiens à remercier mes tuteurs de stage, Monsieur Jean-François Baget chargé de Recherches INRIA ainsi que Madame Marie-Laure Mugnier Professeur à l'Université de Montpellier, qui m'ont formé et accompagné durant toute cette période de stage avec beaucoup de patience et d'instructions.

Mes remerciements s'adressent également à l'ensemble des membres de l'équipe GRAPHIK pour m'avoir accueilli dans un environnement de travail stimulant et pour les conseils qu'ils ont pu me prodiguer au cours de ces cinq mois.

D'autre part, je remercie, toute l'équipe pédagogique du master 2 DECOL pour avoir assurée la partie théorique, en particulier ma tutrice de l'université Mme Marianne Huchard pour son accompagnement pédagogique.

Qu'ils puissent trouver dans ce travail le témoignage de mon profond respect.

Enfin, je remercie les membres du jury qui me feront le grand honneur d'évaluer ce travail.

Table des matières

1. Introduction

Ce document représente mon mémoire de stage de fin d'étude que j'effectue dans l'institut national de recherche en sciences et technologies du numérique (**INRIA**) au sein de l'équipe **GRAPHIK** au Laboratoire d'informatique, de robotique et de microélectronique de Montpellier (**LIRMM**) à Montpellier, sous la direction de Monsieur **Jean-François Baget** chargé de Recherches INRIA ainsi que Madame **Marie-Laure Mugnier** Professeur à l'Université de Montpellier.

Ce stage consiste à faire une extension du formalisme des règles existentielles et de l'implémenter dans la plateforme **GRAAL** qui a été créée par l'équipe GRAPHIK, principalement pour ajouter aux raisonnements purement symboliques la possibilité de faire des raisonnements numériques. En effet, comme nous le verrons au début de ce document, la plateforme graal ne permetait pas de faire des calculs en utilisant des fonctions créées par les utilisateurs, or cette fonctionnalité est très importante car elle permet d'utiliser le langage dans plusieurs autres domaines.

Au delà d'enrichir mes connaissances ce stage m'a permis de comprendre et mieux appréhender le métier d'ingénieur.

Le chapitre 2 présente l'équipe de recherche GraphIK au sein de laquelle j'ai réalisé mon stage, ainsi que sa thématique de recherche centrée sur les règles existentielles et la plateforme logicielle Graal implémentant divers raisonnements dans ce modèle. Le chapitre 3 présente de façon intuitive l'extension du langage que je dois décrire, puis implémenter dans la plateforme Graal. Le chapitre 4 présente de façon formelle et détaillée le langage des règles existentielles, puis l'extension que je dois réaliser. Le chapitre 5 présente de façon détaillée l'implémentation de l'extension que je dois réaliser (hiérarchie de classes, algorithmes importants ...etc). Le chapitre 6 donne un aperçu des tests réalisés sur cette extension, décrit un test comparatif d'efficacité entre l'extension réalisée et la version de départ. Enfin, le chapitre 7 présente une conclusion sur l'ensemble des tâches effectuées et présente aussi l'ensemble des perspectives d'une extension plus performante.

2. Contexte

2.1 L'équipe GraphIK

L'équipe **GraphIK** (**G**raphs for **I**nference and **K**nowledge representation) a été créée en 2010 en tant qu'équipe commune entre le LIRMM (**L**aboratoire **I**nformatique **R**obotique **M**icroélectronique **M**ontpellier) et INRIA (Institut national de recherche en sciences et technologies du numérique)), elle s'appuie sur l'ex équipe RCR (Représentation de Connaissances et Raisonnements) du LIRMM et intègre des chercheurs informaticiens du laboratoire d'agronomie Montpelliérain IATE. Elle est spécialisée dans le domaine de la représentation des connaissances et des raisonnements. Son approche est basée sur la logique et prend sa source dans les graphes conceptuels, qui utilisent des graphes/hypergraphes comme outil de représentation des connaissances. Aujourd'hui, les mêmes formules logiques sont utilisées comme abstraction commune pour de nombreux langages de KRR (Knowledge Representation and Reasoning) dans le but de développer des langages ayant des propriétés de généricité et d'efficacité algorithmique, et de les valider dans des systèmes à base de connaissances réels.

GraphIK se situe au bâtiment 5 du campus saint priest, son responsable est madame **Marie-Laure Mugnier**, elle compte huit chercheurs permanents, trois Associés, quatre doctorants et post-doctorants, deux ingénieurs ainsi que l'assistante administrative **Annie Aliaga**.

FIGURE 2.1 – Localisation des bureaux de l'équipe GraphIK

L'équipe GraphIK se focalise sur trois sujets : théorie, développement logiciel et applications. Parmi les logiciels développés par l'équipe il y a Cogui qui est un éditeur pour la construction et la vérification de bases de connaissances et la plate-forme Graal pour l'interrogation de bases de connaissance.

2.2 Le langage

Comme mon stage consiste a faire une extension de la plate-forme **Graal** basée sur le formalisme des règles existentielles alors je vais commencer par présenter ce langage.

Une base de connaissances est constituée d'une base de faits (que l'on peut voir comme les tables d'une base de données relationnelle) et d'une ontologie, qui est constituée d'un ensemble de règles existentielles. La plupart des algorithmes implémentés dans Graal servent à répondre à la question : existe-t'il une réponse à une requête conjonctive dans la base de connaissances (et quelles sont les réponses à cette requête) ? Comme ce problème est indécidable dans le cas général, un module de Graal permet d'analyser une ontologie pour savoir si, dans ce cas particulier, un des schémas d'algorithmes implémentés permettra de répondre à la requête en temps fini. Parmi ces algorithmes, citons le chaînage avant qui rajoute à la base de faits toutes ses conséquences que l'on peut déduire de l'ontologie (il n'y a alors plus qu'à interroger la base de faits saturée) et le chaînage arrière qui réécrit une requête via l'ontologie (il n'y a alors plus qu'à interroger la base de faits initiale avec cette requête réécrite).

Dans Graal, la base de faits peut être stockée indifféremment comme un graphe en mémoire, dans une base de données relationnelle, ou dans un triple store. Ces faits peuvent provenir d'un fichier DLGP (le format d'échange pour les règles existentielles) ou d'un fichier RDF. Les règles existentielles peuvent également provenir

d'un fichier DLGP, mais peuvent aussi être obtenues par traduction d'assertions en OWL2. Une bibliothèque de règles permet d'implémenter la sémantique du langage RDFS.

2.3 La plateforme Graal

La plate-forme **Graal** est un ensemble de fichiers java qui forment un ensemble de modules tels que : un module *core* qui représente le coeur de ce framework en comportant des classes fondamentales comme la classe Atom, AtomSet, ConjuctiveQuery, Rule...etc, un module *IO* pour les entrée sortie du langage, un autre module *Store* qui fournit différentes types de stockage, un module *Homomorphisme* qui implémente des algorithmes pour l'interrogation de données, un module *Forward Chaining* qui implémente les algorithmes nécessaires pour le chaînage avant, un module *Backward Chaining* qui implémente les algorithmes nécessaires pour la réécriture de requêtes, un module *Rule Set Analyzer* qui fournit des outils pour l'analyse de règles et enfin un module *KB* qui représente une couche de la base de connaissance utilisant ces différentes modules pour répondre aux requêtes.

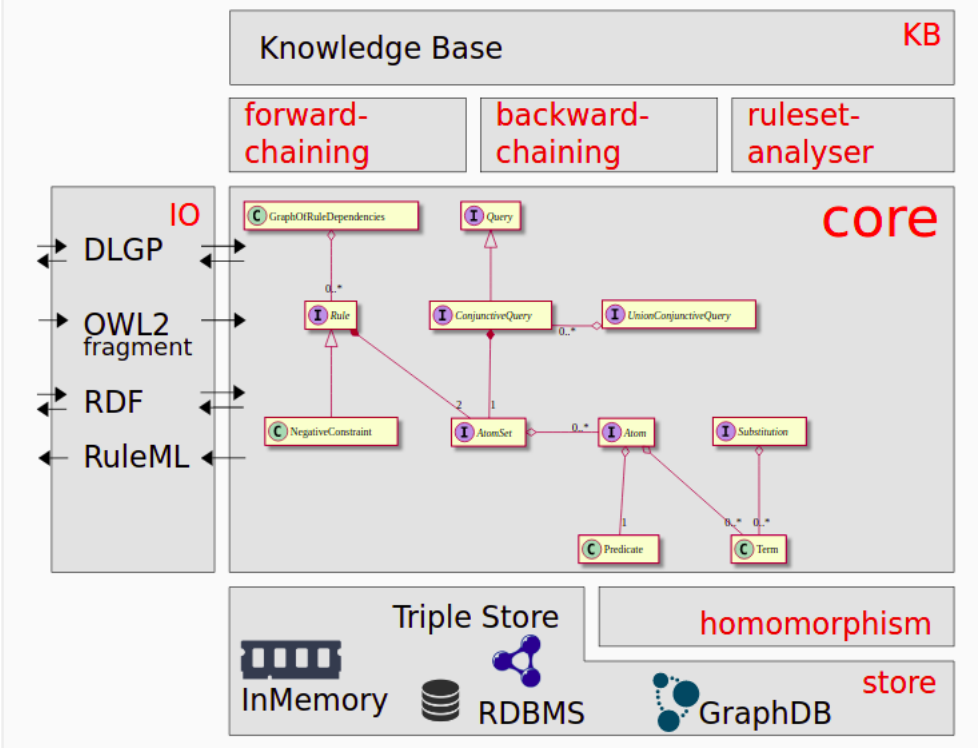


FIGURE 2.2 – Schéma de la plateforme GRAAL

3. Ma mission

3.1 Fonctions et prédicats calculés

Dans le cadre de ce stage, ma mission principale était d'enrichir Graal avec des fonctions et des prédicats calculés. En effet, les règles présentes dans Graal sont des règles purement logiques. Mais nous avons souvent besoin de filtres, c'est à dire de prédicats calculés dans les corps des règles ou dans les requêtes. Ce ne sont pas des prédicats logiques habituels puisque leur interprétation est fixée. Par exemple, dans la règle

$$note(X, Y), superieur(Y, 10) \rightarrow admis(X)$$

qui exprime que si une personne a une note superieure à 10, alors elle est admise, le filtre $superieur(X, 10)$ utilise un prédicat calculé *superieur* qui s'interprétera de la même façon dans tous les modèles. En fait, ce prédicat calculé pourra être lié à une fonction dans un langage de programmation, qui, étant donné 2 paramètres, répondra vrai ou faux. Notons déja que, pour évaluer cette fonction, toutes les variables devront avoir été instanciées (par des littéraux), et cette instanciation sera faite dans la partie "standard" de la requête.

De la même façon que nous utilisons des prédicats calculés dans le corps (hypothèse) d'une règle, nous avons besoin de fonctions calculées dans la tête (conclusion). Par exemple, la règle

$datenaissance(X, D) \rightarrow age(X, d2a(D))$

calcule l'age d'une personne X à partir de sa date de naissance D à l'aide d'une fonction $d2a$.

Notons que l'architecture logicielle devra permettre à un utilisateur d'ajouter facilement de nouveaux prédicats calculés, de nouvelles fonctions calculées, et même de nouveaux types de littéraux de façon la plus simple possible.

3.2 Objectif et limites

L'extension de Graal que je dois réaliser doit prendre en compte une extension des règles permettant l'utilisation des fonctions et prédicats calculés. Il faudra donc :

— une extension du format d'échange DLGP permettant de noter ces nouveaux objets, et son implémentation en *javaCC* ;

— mettre au point un processus de liaison dynamique avec des fonctions java pour évaluer ces fonctions et prédicats calculés. Pour ceci, nous aurons besoin de la librairie *java.lang.reflect* ;

— étendre les mécanismes d'évaluation de requêtes et d'applications de règles pour prendre en compte ces nouveaux objets dans l'algorithme de chaînage avant.

Ces nouveaux objets ne seront pris en compte ni dans le mécanisme de chaînage arrière, ni dans le module KIABORA d'analyse de règles, car ceci pose des problèmes théoriques (comment réécrire ces règles, comment évaluer le déclenchement possible d'une règle par une autre, ...) qui n'ont pas encore été résolus.

3.3 Une plateforme en évolution

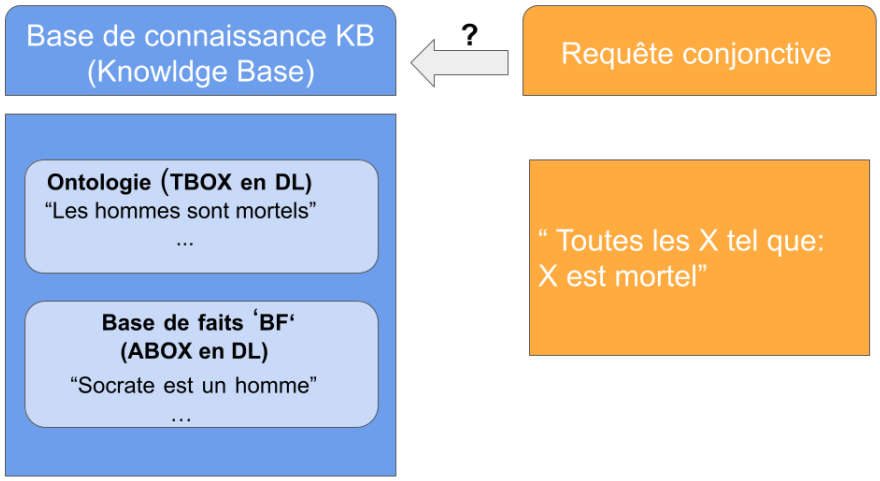
Pendant le déroulement de mon stage, la plateforme Graal était en train de subir une refonte majeure de sa version 1 vers une version 2. Cette refonte concerne la hiérarchie de classes, l'organisation en modules aussi bien que les algorithmes essentiels. Cette refonte n'était pas assez avancée pour que je puisse travailler directement sur la V2.

Aussi, mon travail de développement est destiné non pas à faire partie d'une application finale, mais à être intégré (par Florent Tornil, responsable du développement de la version 2) à celle-ci. Afin de compliquer le moins possible le travail de Florent, nous avons adopté une méthode particulière de programmation : l'idée était de modifier le moins possible les fichiers existants, de travailler uniquement par sous-classes, et de lui proposer éventuellement la hiérarchie de classes que je trouvais la meilleure pour qu'il l'intègre plus tard dans la V2.

4. Le modèle formel

4.1 Existant

Nous considérons des bases de connaissance composées d'une base de faits qui représente les données factuelles, comme des descriptions d'individus, des situations, ..., ainsi que d'une ontologie qui nous permet d'inférer de nouvelles connaissances factuelles. Nous considérons aussi des requêtes auxquelles nous allons répondre à partir de tout ce qui peut être déduit de la base de connaissance (base de faits et ontologie). La présentation du modèle formel existant est inspirée de [4].



4.1.1 Faits et requêtes

Syntaxe Nous commençons par la définition classique d'un vocabulaire en logique du premier ordre sans symbole fonctionnel. Un *vocabulaire* est une paire $V = (C, P)$ où C et P sont deux ensembles disjoints, respectivement de *constantes* et de *noms de prédicats*. Chaque prédicat est muni d'un entier positif qui est son *arité*. On considère également un ensemble infini X de *variables*, disjoint des deux premiers. Un *terme* est soit une constante, soit une variable. Par la suite, les noms de constantes commenceront par une minuscule alors

que les noms de variables commenceront par une majuscule.

Un atome défini sur un vocabulaire $V = (C, P)$ est de la forme $p(t_1, \ldots, t_k)$ où p est un prédicat d'arité k et les t_i sont des termes. Un *fait* est un ensemble d'atomes et une *requête booléenne* est un ensemble d'atomes.

Sémantique A tout ensemble d'atomes F est associé une formule logique $\Phi(F)$ qui est la fermeture existentielle de la conjonction $\phi(F)$ des atomes de F. On dit qu'une interprétation I (en logique du premier ordre) du vocabulaire V est un modèle de l'ensemble d'atomes F lorsque I est un modèle de $\Phi(F)$. On dit que Q est *conséquence sémantique* de F et on note $F \models Q$ lorsque $\Phi(F) \models \Phi(Q)$, c'est à dire lorsque tous les modèles de F sont des modèles de Q.

Exemple 1 *Soit $F = \{p(a, X), q(X, Y)\}$. Alors $\phi(F) = p(a, X) \wedge q(X, Y)$ et la formule logique associée à F est la fermeture existentielle de $\phi(F)$, c'est à dire $\Phi(F) = \exists X \, \exists Y (p(a, X) \wedge q(X, Y))$.*

Homomorphismes Une *substitution* est une application partielle de l'ensemble X des variables dans l'ensemble $X \cup C$ des termes. Si $p(t_1, \ldots, t_k)$ est un atome et σ est une substitution, alors l'application de σ sur cet atome produit un atome $\sigma(p(t_1, \ldots, t_k)) = p(t'_1, \ldots, t'_k)$ où $t'_i = \sigma(t_i)$ si t_i est dans le domaine de σ et $t'_i = t_i$ sinon. Si $F = \{a_1, \ldots, a_p\}$ est un ensemble d'atomes, alors on note $\sigma(F) = \{\sigma(a_1), \ldots, \sigma(a_p)\}$.

Exemple 2 *Soient $F = \{p(a, X), q(X, Y, Z), p(Z, T)\}$ un fait et $\sigma = \{Y : b, Z : a, T : X\}$ une substitution de domaine $\{Y, Z, T\}$. Alors $\sigma(F) = \{\sigma(p(a, X)), \sigma(q(X, Y, Z)), \sigma(p(Z, T))\} = \{p(a, X), q(X, b, a), p(a, X)\} = \{p(a, X), q(X, b, a)\}$.*

Un homomorphisme est une substitution qui permet de prouver la conséquence sémantique entre deux ensembles d'atomes.

Définition 1 (Homomorphisme) *Soient Q et F deux ensembles d'atomes. Un* homomorphisme *de Q dans F est une substitution des variables de Q dans les termes de F telle que $\sigma(Q) \subseteq F$.*

Exemple 3 *Soit $Q = \{p(a, X), q(X, Y, Z), p(Z, T)\}$ le fait de l'exemple 2 et $F = p(a, X), q(X, b, a), q(X, c, d)$ un fait. Alors la substitution $\sigma = \{Y : b, Z : a, T : X\}$ de l'exemple 2 est bien un homomorphisme de Q dans F car $\sigma(Q) = \{p(a, X), q(X, b, a)\} \subseteq F$.*

Théorème 1 *Soient Q et F deux ensembles d'atomes. Alors $F \models Q$ si et seulement si il existe un homomorphisme de Q dans F.*

Savoir s'il existe un homomomorphisme de Q dans F est un problème NP-complet. Nous ne discutons pas ici des divers algorithmes permettant d'énumérer les homomorphismes, puisque nous n'avons pas eu besoin de les modifier dans notre travail de développement.

Le cas des requêtes non booléennes Soit F un ensemble d'atomes. Une requête non booléenne sur F est de la forme $Q \to ans(X_1, \ldots, X_k)$ où Q est un ensemble d'atomes, *ans* est un nom de prédicat particulier (qui ne sert que pour l'écriture des requêtes) et les X_i sont des variables apparaissant dans Q. Un tuple $(a_1, \ldots, a_k)$ est une *réponse* à la requête si et seulement si $F \models \sigma(Q)$ où σ est la substitution qui à chaque X_i associe a_i.

4.1.2 Règles existentielles

Syntaxe Une *règle existentielle* est une paire d'ensemble d'atomes notée $B \to H$. Les ensembles H et B sont appelés respectivement *tête* (Head) et *corps* (Body) de la règle. Une *variable frontière* est une variable partagée par le corps et la tête de la règle, tandis qu'une *variable existentielle* est une variable qui apparaît uniquement dans la tête de la règle.

Sémantique La formule logique $\Phi(B \to H)$ associée à une règle existentielle est $\forall \vec{X} \, \forall \vec{Y} (\phi(B) \to (\exists \vec{Z} \phi(H))$, où $\vec{X}$ est l'ensemble de variables figurant uniquement dans le corps, $\vec{Y}$ l'ensemble de variables frontières de la règle et $\vec{Z}$ l'ensemble des variables existentielles de la règle.

Exemple 4 *Soit la règle existentielle $R = p(X, V), p(Y, V) \to q(X, Z), q(Y, Z)$. Les variables n'apparaissant que dans le corps sont $\{V\}$, les variables frontières sont $\{X, Y\}$ et les variables existentielles sont $\{Z\}$. La formule logique associée à R est donc $\Phi(R) = \forall X \forall Y (p(X, V) \wedge p(Y, V)) \to \exists Z (q(X, Z) \wedge q(Y, Z))$.*

Une interprétation I (en logique du premier ordre) du vocabulaire V est un modèle d'une règle R lorsque I est un modèle de $\Phi(R)$. Soit F un fait, $\mathcal{R}$ un ensemble de règles existentielles et Q une requête booléenne. Alors on dit que Q est conséquence sémantique de F et $\mathcal{R}$, et on note $F, \mathcal{R} \models Q$ lorsque $\Phi(F), \Phi(\mathcal{R}) \models \Phi(Q)$, c'est à dire lorsque toutes les interprétations qui sont un modèle de F et de chaque règle de $\mathcal{R}$ sont aussi des modèles de Q.

Dérivation et modèle universel Soit F un fait et $R = B \to H$ une règle existentielle. On dit que R est applicable sur F si il existe un homomorphisme σ de B dans F. Dans ce cas, *l'application de R sur F suivant σ* produit un fait $\alpha(F, R, \sigma) = F \cup \sigma^{safe}(H)$, où $\sigma^{safe}(X) = \sigma(X)$ si X est une variable frontière de la règle, et $\sigma^{safe}(X)$ est une nouvelle variable (fresh variable) sinon.

Exemple 5 *Soient $R = p(X, V), p(Y, V) \to q(X, Z), q(Y, Z)$ une règle existentielle et $F = \{p(a, c), p(b, c)\}$ un fait. Alors R est applicable sur F car $\sigma = \{X : a, Y : b, V : c\}$ est un homomorphisme du corps de la règle dans F. L'application de R sur F suivant σ produit le fait $\alpha(F, R, \sigma = F \cup \{q(a, Z_0), q(b, Z_0)\}$. Notons que la variable existentielle Z de R a été substituée par une nouvelle variable Z_0 lors de l'application de la règle.*

Soit F un fait et $\mathcal{R}$ un ensemble de règles. Une $\mathcal{R}$-dérivation de F est une séquence (possiblement infinie) $F = F_0, F_1, F_2 \ldots$ telle que pour tout $i \geq 1$, F_i est obtenu par l'application d'une règle de $\mathcal{R}$ sur F_{i-1}. Une

dérivation est *complète* si toute application possible d'une règle est réalisée au cours de la dérivation, c'est à dire, pour tout F_i, pour toute règle R applicable à F_i suivant σ, il existe un F_j tel que $F_j = \alpha(F_{j-1}, R, \sigma)$. Le *résultat* d'une dérivation est l'union de tous les faits apparaissant dans la dérivation. Remarquons que puisque une dérivation peut-être infinie, F^* peut lui aussi être infini. On note F^* le résultat d'une dérivation complète.

Théorème 2 *Soient F un fait et $\mathcal{R}$ un ensemble de règles existentielles. Alors F^* est un* modèle universel *de $(F, \mathcal{R})$, c'est à dire que pour toute requête booléenne Q, $F, \mathcal{R} \models Q$ si et seulement si $F^* \models Q$.*

Exemple 6 *Soient $F = \{p(a,b), p(b,c), q(a,d)\}$, et $\mathcal{R} = \{R\}$ avec $R = p(X,Y), q(X,Z) \rightarrow q(Y,T)$. La règle R est applicable sur F suivant $\sigma = \{X : a, Y : b, Z : d\}$, et son application suivant σ produit $F' = F \cup \{q(b, T_0)\}$. Notons que σ est le seul homomorphisme du corps de R dans F. Maintenant, la règle R est applicable sur F' suivant $\sigma' = \{X : b, Y : c, Z : T_0\}$, et son application suivant σ' produit $F'' = F' \cup \{q(c, T_1)\}$. La dérivation (F, F', F'') est complète car toutes les applications de R ont été effectuées, et le résultat de cette dérivation est $F^* = F \cup F' \cup F'' = F''$. Ce résultat, qui est un modèle universel, est fini.*

Il peut arriver que le modèle universel ne soit pas fini, comme le montre l'exemple suivant.

Exemple 7 *Soient $F = \{q(a)\}$, et $\mathcal{R} = \{R\}$ avec $R = q(X) \rightarrow p(X,Y), q(Y)$. La règle R est applicable sur F suivant $\sigma = \{X : a\}$, et son application suivant σ produit $F_1 = F \cup \{p(a, Y_0), q(Y_0\}$. Maintenant, la règle R est applicable sur F_1 suivant $\sigma = \{X : Y_0\}$, et son application suivant σ produit $F_2 = F1 \cup \{p(Y_0, Y_1), q(Y_1)\}$. La règle R est applicable sur F_2 et comme à chaque application dans une étape i, elle produit un atome $q(Y_i)$, alors la dérivation $(F, F_1, F2, \ldots)$ est infinie, et le résultat de cette dérivation est $F^* = \{q(a), p(a, Y_0), q(Y_0), p(Y_0, Y_1), q(Y_1), p(Y_1, Y_2), q(Y_2), \ldots\}$. Ce modèle universel est infini.*

Ceci est inévitable, car le problème de savoir si une requête booléenne se déduit d'un fait et d'un ensemble de règles est indécidable. De nombreux travaux de recherche ont été menés pour identifier des classes de règles pour lesquelles il existe une dérivation complète finie, ou pour identifier des méthodes menant à des dérivations plus courtes mais dont le résultat est quand même un modèle universel. Ces méthodes sont appelées *chases* en bases de données, et nous pouvons citer :

— l'*oblivious chase*, qui n'applique pas deux fois la même règle dans une dérivation suivant le même homomorphisme ;

— le *semi-oblivious chase*, qui n'applique pas deux fois la même règle dans une dérivation suivant le même homomorphisme *restreint à la frontière de la règle* ;

— le *restricted chase* qui n'applique pas une règle $R = B \rightarrow H$ sur F suivant σ si le résultat de l'application se replie sur σ, c'est à dire si il existe un homomorphisme de $\sigma'(H)$ dans F, où σ' est la restriction de σ à la frontière de R.

— enfin, le *core chase* remplace le fait produit par son plus petit fait équivalent après chaque application de règle.

Ces différents types de chase sont de plus en plus puissants (oblivious $\preceq$ semi-oblivious $\preceq$ restricted $\preceq$ core) et on peut montrer que :

— si XY et le résultat du Y chase est fini, alors le résultat du X chase est fini ;

— si $X \preceq Y$, alors il existe une instance pour laquelle le résultat du X chase est infinie mais le résultat du Y chase est fini.

Ce mécanisme qui développe un modèle universel à partir d'un fait et d'un ensemble de règles est aussi appelé *chaînage avant*. Un autre mécanisme consiste à *réécrire* une requête à partir de l'ensemble de règles. Il suffit alors d'interroger le fait initial avec la requête réécrite. Ce mécanisme de *chaînage arrière*, qui lui aussi peut ne pas s'arrêter, ne sera pas développé dans ce mémoire.

4.2 Evolution souhaitée

Dans cette partie, nous exposons les fondements théoriques pour les extensions du langage que nous devons implémenter. Les fonctions et prédicats calculés sont inspirés de [1], tandis que la négation par l'échec dans les règles existentielles a été étudiée dans [2].

4.2.1 Faits et requêtes

Syntaxe Nous ajoutons au vocabulaire des *noms de prédicats calculés*, des *noms de fonctions calculées*, des *types de littéraux* et des *valeurs littérales*. Les noms de prédicats calculés ainsi que les noms de fonctions calculées sont munis d'une arité et d'une *signature* (la signature d'un nom de prédicat ou d'un nom de fonction calculée d'arité k est un k-tuple de types de littéraux, et la signature d'un nom de fonction calculée contient également le type de littéral pour sa valeur de retour). Un type de littéral a un ensemble de valeurs littérales possibles.

Exemple 8 *Le type de littéral* `entier` *a pour valeurs littérales possibles* $\{0, 1, 2, 3, \ldots\}$. *Le type de littéral* `chaîne` *a pour valeurs littérales possibles* $\{$`"foo"`, `"bar"`, $\ldots\}$. *La fonction calculée* `longueur` *est d'arité 1, a pour signature (*`chaîne`, `entier`*), elle prend comme paramètre une chaîne et retourne sa longueur, qui est un entier. La fonction calculée* `somme` *est d'arité 2, sa signature est (*`entier`, , `entier`*), elle prend comme paramètres 2 entiers et retourne leur somme, qui est un entier. Le prédicat calculé* `inferieur` *est d'arité 2, sa signature est (*`entier`, `entier`*), il prend comme paramètre 2 entiers et, comme tout prédicat calculé, retourne un booléen (vrai si et seulement si le premier entier est inférieur au deuxième).*

Les termes des atomes d'un fait peuvent maintenant être des constantes, des variables, ou des valeurs littérales. *Attention, les termes des atomes d'un faits ne peuvent pas être des termes fonctionnels (calculés), et leur prédicat ne peut pas être un prédicat calculé.*

Un *terme calculé* est de la forme $f(t_1, \ldots, t_k)$, où f est un nom de fonction calculée d'arité k, et chaque t_i est soit une variable, soit une valeur littérale dont le type est le i^{me} type de la signature de f, soit un terme calculé dont le type de retour est le i^{me} type de la signature de f.

Exemple 9 *L'expression* $\texttt{somme}(3, \texttt{longueur}(X))$ *est un terme calculé qui rajoute 3 à la longueur de X (qui doit être une chaîne). Nous utiliserons souvent la notation simplifiée* $3 + \texttt{longueur}(X)$ *qui veut dire exactement la même chose.*

L'expression $\texttt{longueur}(\texttt{somme}(X, Y))$ *n'est pas un terme calculé car le type de retour de* `somme` *n'est pas celui requis par* `longueur`.

Dans ce qui suit, un *filtre* sera une sorte de requête booléenne qui ne peut s'évaluer que dans le contexte d'une substitution. Nous considérons deux sortes de filtres :

— le *filtre calculé* qui ressemble à un atome mais utilise un nom de prédicat calculé.

— la *négation* qui est la négation par l'échec d'une requête booléenne (attention, cette requête sera plus complexe que celle qu'on a vue précédemment).

Un *filtre calculé* est de la forme $p(t_1, \ldots, t_k)$ où p est un nom de prédicat calculé d'arité k et chaque t_i est soit une variable, soit une valeur littérale dont le type est le i^{me} type de la signature de p, soit un terme calculé dont le type de retour est le i^{me} type de la signature de p.

Une *F-requête booléenne* est de la forme $a_1, \ldots, a_p, f_1, \ldots, f_k, not\ Q_1, \ldots, not\ Q_r$ où les a_i sont des atomes au sens habituel, les f_j sont des filtres, et les Q_s sont des F-requêtes booléennes sans négation. Toutes les variables apparaissant dans un filtre calculé doivent apparaître dans un des atomes a_i.

Exemple 10 *L'expression* $\texttt{inferieur}(X + 2, Y + 3)$ *est un filtre calculé. Nous utiliserons par la suite la notation simplifiée* $X + 2 < Y + 3$. *L'expression* $p(X, Y), X + 2 < Y + 3$ *est une F-requête booléenne, mais* $p(X, X), X + 2 < Y + 3$ *ne l'est pas car la variable* Y *du filtre n'apparaît pas dans l'atome* $p(X, X)$. *L'expression* $p(X, Y), X + 2 < Y + 3, not\, p(X, Z), Z < 3$ *est également une F-requête booléenne.*

Sémantique La sémantique des fonctions et prédicats calculés est la même dans toutes les interprétations. Ainsi, $X < Y$ ne sera vrai que dans le cas où l'évaluation de X est inférieure à celle de Y. C'est pour cela que nous n'évaluerons un prédicat ou une fonction calculée que dans le contexte d'une substitution. Ainsi, $X < Y$ sera évalué à vrai dans le contexte de la substitution $\{X : 1, Y : 3\}$, mais sera évalué à faux dans le contexte de la substitution $\{X : 3, Y : 1\}$. Pour évaluer un prédicat ou une fonction calculée, il faudra que

toutes les variables apparaissant à l'intérieur soient dans le domaine de la substitution : on pourra alors calculer récursivement cette évaluation.

Exemple 11 *Èvaluons le filtre* $f = Y * (X + 2) < Y + 3$ *dans le contexte de la substitution* $\sigma = \{X : 1, Y : 5\}$. *Comme toutes les variables de* f *appartiennent au domaine de* σ *alors on pourra évaluer* f *en évaluant tout d'abord récursivement chaque terme du prédicat* $<$.

Pour le terme $Y * (X + 2)$, *il faut d'abord évaluer* Y *(dans le contexte de* σ*) puis* $X + 2$. *Puisque* Y *est une variable, son évaluation est* $\sigma(Y) = 5$, *qui est un littéral de type entier. Le terme* $X + 2$ *est fonctionnel, il faut d'abord évaluer* $\sigma(X) = 1$ *(encore un entier), puis le littéral* 2 *évalué par lui-même. L'évaluation de la somme de ces deux entiers est l'entier* 3. *Le produit* $Y * (X + 2)$ *est maintenant évalué à* 15.

De la même façon, le terme $Y + 3$ *est évalué à* 8, *et l'évaluation du prédicat calculé* $Y * (X + 2) < Y + 3$ *retourne faux car 15 n'est pas inférieur à 8.*

Une négation est interprétée par la négation par l'échec : le filtre *not* Q est évalué à vrai dans le contexte d'une substitution σ et dans un fait F si et seulement si $\sigma(Q)$ n'est pas conséquence sémantique de F.

Définition 2 (Modèle d'une F-requête booléenne) *Une interprétation* I *(où toutes les valeurs littérales sont interprétées par elles-mêmes) est un modèle d'une F-requête booléenne* $a_1, \ldots, a_p, f_1, \ldots, f_k, \text{not } Q_1, \ldots, \text{not } Q_r$ *si et seulement si il existe une assignation* λ *des variables de* $\{a_1, \ldots, a_p\}$ *dans le domaine de* I *qui satisfait tous les atomes et tous les filtres (ils sont évalués à vrai dans le contexte de cette assignation) de la F-requête.*

La traduction logique d'une F-requête booléenne $Q = a_1, \ldots, a_p, f_1, \ldots, f_k, not\ Q_1, \ldots, not\ Q_r$ est calculée de la façon suivante : On note $\vec{X}$ les variables qui apparaissent dans les atomes standard a_i (on rappelle que toutes les variables des filtres f_j sont dans $\vec{X}$) et on note $\vec{Y_q}$ les variables de la négation Q_q qui ne sont pas dans $\vec{X}$. Alors la traduction d'un filtre f_r est $\phi(f_r) = f_r$, la traduction d'une négation Q_q (dans le contexte d'une F-requête F) est $\phi_F(Q_q) = \exists \vec{Y_q} \phi(Q_q)$, et la la traduction de la F-requête est $\Phi(Q) = \exists \vec{X} \phi(F)$ où $\phi(F) = \phi(\{a_i\}) \wedge \phi(f_1) \wedge \cdots \wedge \phi(f_k) \wedge \phi_F(Q_1) \wedge \cdots \wedge \phi_F(Q_r)$.

Exemple 12 *La traduction de la F-requête* $F = p(X, Y), X < Y, not\, r(X, Z), not\, s(Y, Z)$ *est* $\Phi(F) = \exists X \exists Y (p(X, Y) \wedge X < Y \wedge not\, (\exists Z r(X, Z)) \wedge not\, (\exists Z s(Y, Z)))$. *On remarque dans cet exemple que la variable* Z *apparaît 2 fois, dans la portée de deux quantificateurs différents, et ces 2 occurrences ne représentent pas le même individu. On pourrait les renommer en* $\Phi(F) = \exists X \exists Y (p(X, Y) \wedge X < Y \wedge not\, (\exists Z_1 r(X, Z_1)) \wedge not\, (\exists Z_2 s(Y, Z_2)))$ *pour plus de clarté.*

On peut vérifier qu'une interprétation est un modèle d'une F-requête booléenne F (au sens de la définition 2) si et seulement si c'est un modèle de la formule $\Phi(F)$ (au sens d'une extension de la logique du premier ordre avec négation par l'échec et fonctions et prédicats dont l'interprétation est fixée). Définir formellement cette logique sort du cadre de ce travail.

F-homomorphismes L'homomorphisme sert à prouver qu'une requête conjonctive booléenne est la conséquence sémantique d'un fait. La généralisation aux F-requêtes conjonctives booléennes va nécessiter la généralisation de l'homomorphisme : un F-homomorphisme sera donc un homomorphisme qui respecte tous les filtres.

Définition 3 *Soit Q une F-requête composée d'un ensemble Q^+ d'atomes, d'un ensemble Q^f de filtres calculés, et d'un ensemble Q^- de négations. Soit F un fait. Un* F-homomorphisme *de Q dans F est un homomorphisme σ de Q^+ dans F qui satisfait tous les filtres, c'est à dire :*

- *si f est un filtre calculé dans Q^f, alors l'évaluation de f dans le contexte de σ retourne vrai ;*
- *si q est une négation de Q^-, alors il n'existe pas de F-homomorphisme σ' de q dans F compatible avec σ, c'est à dire tel que si X est une variable qui apparaît à la fois dans Q^+ et q, alors $\sigma(t) = \sigma'(t)$.*

On remarque que si F ne contient pas de variable, alors la satisfaction d'une négation q peut s'écrire "si il n'existe pas d'homomorphisme de $\sigma(q)$ dans F", mais ce n'est pas vrai dans le cas général, comme le montre l'exemple suivant.

Exemple 13 *Soit $Q = p(X,Y), not\, q(Y)$ une requête et $F = p(a,U), q(a)$ un fait. Le seul homomorphisme de $Q^+ = p(X,Y)$ dans F est $\sigma = \{X : a, Y : U\}$. Comme il n'y a pas de F-homomorphisme σ' de la négation $q(Y)$ dans F compatible avec σ, c'est à dire tel que $\sigma'(Y) = U$, alors l'homomorphisme σ satisfait la négation et est donc un F-homomorphisme.*

Remarquons que si on avait adopté la formulation alternative de la remarque précédente, on aurait bien un homomorphisme de $\sigma(q^-) = q(U)$ dans F, avec $\{U : a\}$, et donc σ ne serait pas un homomorphisme.

C'est bien le premier comportement qui est souhaité pour respecter la sémantique logique.

Le théorème suivant n'est pas démontré, mais adapte à nos objets les résultats de [2] et [1].

Théorème 3 *Soient Q une F-requête booléenne et F un ensemble d'atomes. Alors $F \models Q$ si et seulement si il existe un F-homomorphisme de Q dans F.*

Exemple 14 *Soient $F = \{p(1,2), p(3,4), q(4,6)\}$ un fait, et $Q = p(X,Y), X < Y, not\ q(Y,Z), not\ q(X,Z)$ une F-requête booléenne. Il y a deux homomorphismes de $Q^+ = p(X,Y)$ dans F : $\sigma_1 = \{X : 1, Y : 2\}$ et $\sigma_2 = \{X : 3, Y : 4\}$. Ces deux homomorphismes satisfont le filtre calculé $X < Y$. Le seul homomorphisme $\sigma' = \{Y : 4, Z : 6\}$ de la première négation de Q dans F est compatible avec σ_2, donc σ_2 ne respecte pas tous les filtres et n'est donc pas un F-homomorphisme de Q dans F, mais comme il n'est pas compatible avec σ_1, il n'y a pas d'homomorphisme compatible et σ_1 satisfait la première négation. Pour la deuxième négation, le seul homomorphisme est $\{X : 4, Y : 6\}$ qui n'est pas compatible avec σ_1, qui satisfait donc tous les filtres. Cet homomorphisme σ_1 est donc un F-homomorphisme de Q dans F, et, d'après le théorème, on a $F \models Q$.*

Théorème 4 *En supposant que toutes les fonctions utilisées (fonctions calculées et prédicats calculés) se calculent en temps polynomial, savoir si il existe un homomorphisme de Q dans F est un problème DP-complet.*

Preuve Pour montrer qu'il existe un F-homomorphisme de $Q = a_1, \ldots, a_p, f_1, \ldots, f_k, not\ Q_1, \ldots, not\ Q_r$ dans F il faut :

- montrer qu'il existe un homomorphisme σ de $a_1, \ldots, a_p$ dans F (ce problème est NP-complet)
- montrer que σ respecte tous les filtres calculés f_i (ceci se fait par hypothèse en temps polynomial)
- montrer que pour chaque Q_i, il n'existe pas d'homomorphisme de $\sigma(Q_i)$ dans F (ce problème est co-NP-complet)

Notre problème est donc constitué d'un problème NP-complet et d'un problème co-NP complet, et comme ces deux sous problèmes peuvent être indépendants (par exemple avec $Q = Q^+, not\, Q^-$ lorsque Q^+ et Q^- n'ont aucune variable en commun, alors ce problème est DP-complet. □

Le cas des F-requêtes non booléennes Celles-ci se définissent de la même manière que dans le cas standard. Soit F un ensemble d'atomes. Une F-requête non booléenne sur F est de la forme $Q \rightarrow ans(X_1, \ldots, X_k)$ où Q est un ensemble d'atomes, *ans* est un nom de prédicat particulier (qui ne sert que pour l'écriture des requêtes) et les X_i sont des variables apparaissant dans la partie positive Q^+ de Q. Un tuple $(a_1, \ldots, a_k)$ est une *réponse* à la F-requête si et seulement si $F \models \sigma(Q)$ où σ est la substitution qui à chaque X_i associe a_i.

4.2.2 Règles existentielles

Nous étendons maintenant les règles existentielles afin que leur corps soit une F-requête quelconque, et afin de pouvoir utiliser des fonctions calculées dans leur tête.

Syntaxe Une règle existentielle est maintenant de la forme $B \rightarrow H$ où B est une F-requête et H un *P-fait*.

Un *P-atome* est de la forme $p(t_1, \ldots, t_k)$ où p est un prédicat (standard) d'arité k et chaque t_i peut être soit un terme ou bien un terme calculé. Un P-atome a peut s'évaluer dans le contexte d'une substitution σ dont le domaine contient toutes les variables des termes calculés de a. Dans ce cas, l'évaluation de a dans le contexte de σ produit un atome noté $\sigma(a)$. Remarquons que cet atome produit ne contient plus de fonction calculée.

Exemple 15 *Le P-atome $p(X, Y, Y + 3 + Z, T)$ ne peut s'évaluer que dans le contexte d'une substitution dont le domaine contient Y et Z. Dans le contexte de la substitution $\{Y : 1, Z : 2, T : a\}$, son évaluation produira $p(X, 1, 6, a)$.*

Un P-fait F est un ensemble de P-atomes. Il peut s'évaluer dans le contexte d'une substitution σ quand tous ses atomes peuvent s'évaluer dans le contexte de σ. Dans ce cas, l'évaluation de F dans le contexte de σ produit un fait $\sigma(F) = \{\sigma(a) | a \in F\}$.

Une règle existentielle est maintenant de la forme $B^+, B^- \to H$ où H est un P-fait et B^+, B^- est une F-requête booléenne (on note B^+ l'ensemble de ses atomes standard et B^- l'ensemble de ses filtres). La *frontière* de la règle est maintenant l'ensemble des variables apparaissant à la fois dans B^+ et dans H. Les autres variables de H sont des *variables existentielles*. Toutes les variables apparaissant dans une fonction calculée de H doivent aussi apparaître dans B^+ (c'est à dire être des variables frontière).

Exemple 16 *La règle $q(X), not\, r(X,Y) \to p(X+Y)$ n'est syntaxiquement pas valide car la variable Y apparaît dans un terme calculé mais n'est pas une variable frontière (elle n'apparaît que dans une négation).*

La règle $p(X), not\, r(X,Y), not\, s(X,Y) \to r(Y,X)$ est une règle existentielle. Nous avons déjà vu que les variables Y des négations étaient dans la portée d'un quantificateur local à cette négation, on pourrait donc écrire la règle $p(X), not\, r(X,Y_1), not\, s(X,Y_2) \to r(Y,X)$ sans changer sa sémantique. On voit alors que la variable Y de la tête n'est plus dans le corps de la règle : c'est bien une variable existentielle.

Sémantique Nous ne donnons pas ici de sémantique logique pour les règles car l'introduction de la négation pose de graves problèmes théoriques qui ne peuvent pas être résolus dans le cadre de la logique du premier ordre. Nous étendons donc dans le paragraphe suivant la "sémantique opérationnelle" des règles existentielles au sens habituel. Nous discuterons également des problèmes posés par la négation. Le lecteur pourra se référer à [2] pour les problèmes liés à une sémantique formelle basée sur une extension des modèles stables aux règles existentielles.

Dérivations : une sémantique opérationnelle Soit F un fait et $R = B \to H$ une règle existentielle. On dit que R est applicable sur F si il existe un F-homomorphisme σ de B dans F. Dans ce cas, l'application de R sur F suivant σ produit un fait $\alpha(F, R, \sigma) = F \cup \sigma^{\text{safe}(H)}$, où $\sigma^{\text{safe}(H)}$ est obtenu de la façon suivante :

— remplacer toutes les variables existentielles de H par des "fresh variables" (comme dans le cas standard)

— évaluer H dans le contexte de la substitution σ (c'est à dire appliquer la substitution, comme dans le cas standard), puis évaluer le résultat des fonctions calculées, comme expliqué ci-dessus).

Exemple 17 *Soient $F = \{p(1,2), p(3,2)\}$ et $R = p(X,Y), X < Y \to q(X+Y, Z)$. Alors R est applicable sur F suivant le F-homomorphisme $\sigma = \{X : 1, Y : 2\}$ et produit le fait : $\alpha(F, R, \sigma) = F \cup \{q(3, Z_0)\}$ (notons le remplacement de Z par Z_0 dans la "safe substitution").*

On peut donc définir maintenant un mécanisme de dérivation identique au cas standard en utilisant ceette nouvelle méthode d'application de règles. Mais la négation va poser un grave problème.

Négation et stratification Comme le montre l'exemple suivant, la négation nous fait perdre la propriété d'un modèle universel unique (voir cours M2) :

Exemple 18 *Soit $F = p(a)$ et les règles $R_1 = p(X), not\, q(X) \rightarrow s(X)$ et $R_2 = p(X), not\, s(X) \rightarrow q(X)$.*

Considérons la dérivation suivante à partir de F : on applique R_1 sur F pour obtenir $F_1 = p(a), s(a)$. La règle R_2 n'est pas applicable sur F_1, donc F_1 est le résultat d'une dérivation complète.

Maintenant considérons la dérivation complète dans laquelle on applique R_2 en premier pour obtenir le résultat $F_2 = p(a), q(a)$ d'une dérivation complète. On voit bien que ces deux résultats ne sont pas équivalents, ce qui est une propriété essentielle des règles existentielles standard.

On voit dans cet exemple que ce qui ne marche pas est dû à une absence de monotonie. Dans les règles existentielles standard, lorsqu'on peut appliquer une règle à une certaine étape de la dérivation, elle reste applicable plus tard dans la dérivation. Essayons de formaliser cette notion :

Une F-requête Q est monotone si pour tout fait F, pour tout F-homomorphisme σ de Q dans F, si $F \subseteq F'$ alors σ est un F-homomorphisme de Q dans F'.

Une règle existentielle sera monotone si son corps est monotone.

On peut prouver (voir [2]) que si le chase est monotone (si $i < j$ alors $F_i \subseteq F_j$), alors toute dérivation utilisant des règles monotones produira un résultat équivalent. On a vu dans l'exemple précédent que les règles avec négation pouvaient ne pas être monotones. On peut aussi prouver que si une règle existentielle ne contient pas de négation (les seuls filtres sont des filtres calculés), alors elle est monotone.

Proposition 1 *Soit Q une F-requête. Si la partie négative de Q est vide (Q est une requête sans négation), alors Q est monotone.*

Preuve : soient F un fait et $Q = (Q^+, Q^f)$ une F-requête où Q^+ est la partie normale de la requête et Q^f l'ensemble de ses filtres calculés. Supposons que σ est un F-homomorphisme de Q dans F. Soit maintenant un fait F' tel que FF'. Alors σ est toujours un homomorphisme de Q^+ dans F et comme il respecte encore tous les filtres (ça ne dépend pas des faits), il est un F-homomorphisme de Q dans F. Par définition, Q est donc monotone. □

En conclusion, si nos règles ne comportent pas de négation, alors le résultat de la dérivation sera unique. Dans le cas contraire, on peut assurer l'unicité si les règles sont *stratifiables*. Intuitivement, un ensemble de règles est stratifiable si on peut le partitionner en ensembles $\mathcal{R}_1, \ldots, \mathcal{R}_k$ tels que :

— si p est un prédicat qui apparaît dans une négation d'une règle du paquet $\mathcal{R}_i$, alors il ne peut pas apparaître dans la tête d'une règle d'un paquet $\mathcal{R}_j$ tel que $j \geq i$ (ceci empêche de remettre en cause une application si on procède dans l'ordre des paquets de règles)

— aucune application d'une règle d'un paquet i ne peut créer une nouvelle application d'une règle d'un paquet $j < i$ (ceci assure que la saturation dans l'ordre des paquets n'oublie aucune application de règle)

Si un ensemble de règles est stratifiable, alors la saturation dans l'ordre des paquets donne un résultat unique.

Enfin, si un ensemble de règles comporte des négations et n'est pas stratifiable, le modèle du chase n'est plus adapté : il faut alors considérer une "arborescence de chases" comme réalisé dans [2].

Typage statique et dynamique Au cours de ce travail, la vérification de types n'a été ni étudiée ni implémentée (ceci sortait du cadre de ce travail). Pourtant, ceci peut poser des problèmes comme le montre l'exemple suivant :

Exemple 19 *Soit la règle* $p(X) \rightarrow q(X+1)$*. Si on l'applique sur* $p(3)$*, ceci va produire* $q(4)$*. Mais si on l'applique sur* $p(a)$*, l'appel à la fonction* `somme` *va produire une erreur.*

Lorsqu'on analyse cette fonction, il faudra s'apercevoir que l'arument de $q(X+1)$ est produit par la fonction `somme`, donc est un `entier`, et que comme cet argument est obtenu par la variable X de $p(X)$, alors cet argument de p doit aussi être un `entier`. Il faudrait donc pouvoir imposer à tout atome dont le prédicat est p d'avoir un type `entier` pour son argument. Même si je n'ai pas eu le temps d'étudier ce problème (et encore moins de l'implémenter), il semble qu'on pourrait travailler de la façon suivante (inspirée de [1]) :

- tout prédicat (standard), en plus de son arité k, possède une signature, c'est à dire un k-tuple de types qu'il doit respecter ;
- cette signature se "devine" au fur et à mesure de la lecture des données, et il y a une erreur de typage lors de la construction de la base de connaissances lorsque des données ne permettent pas de construire une signature (par exemple on lit l'atome $p(X,1)$ qui nous fait deviner que la signature de p est $(any, entier)$ puis l'atome $p(\text{"}a\text{"}, Y)$ nous fait préciser la signature de p qui sera $(chaine, entier)$ mais si on lit maintenant $p(1,3)$ on ne pourra plus faire préciser la signature si il n'y a pas de sous-type commun à chaine et entier.
- comme vu dans l'exemple précédent, la lecture d'une règle permettra aussi de préciser la signature.

Il nous semble que si le nom d'une fonction suffit à la sélectionner (il ne peut pas y avoir 2 fonctions de même nom, même avec des arguments différents), une analyse de la base de connaissance en 2 passes (règles puis faits) permettra de s'assurer d'un typage correct, et évitera toute erreur dans les raisonnements. Dans le cas de la sélection multiple (des fonctions différentes de même nom sont sélectionnées suivant le type de leurs arguments), cela semble plus compliqué. Pourtant, il serait intéressant que + puisse se comprendre comme une somme si ses arguments sont des entiers et comme une concaténation si ce sont des chaînes...

5. Implémentation

Nous présentons ici les extensions de la plateforme Graal [3] que nous avons réalisées. Nous présentons tout d'abord le format d'échange DLGP utilisé dans cette plateforme [5] ainsi que les modifications apportées à sa grammaire pour pouvoir considérer nos nouveaux objets. Les sections suivantes sont consacrées à l'évaluation des F-requêtes et à l'application des règles. Pour cela, nous avons souhaité fournir un mécanisme permettant à l'utilisateur de pouvoir facilement lier des noms de fonctions, prédicats et types à du code java permettant de les évaluer. Ceci nous a imposé de travailler avec le package `java.lang.reflect`.

A chaque fois, nous avons insisté sur des détails techniques qui ne sont peut-être pas indispensables à la lecture de ce mémoire mais seront je pense utiles lors de l'intégration de ces extensions à la prochaine version de Graal.

5.1 Extension du langage DLGP

Le format *Dlgp* (Datalog plus) étend le format *Datalog* standard, il permet de coder des règles existentielles, des faits et des requêtes conjonctives. Les règles existentielles sont étendues pour considérer d'autres types d'objets comme *les contraintes négatives* (une règle existentielle avec une tête contenant le symbole absurde, interprété toujours à faux), les règles avec égalité (qui codent des dépendances fonctionnelles, prévues dans le format mais qui ne peuvent être utilisées qu'en chaînage avant). Les termes et les prédicats sont codés par une chaîne de caractères construite avec les lettres de l'alphabet $[a \dots zA \dots Z]$, les chiffres $[1 \dots 9]$ et le caractère underscore $'\_'$. Et comme en Datalog, les variables commencent par une majuscule et les constantes par une minuscule. Pour que le format Dlgp soit compatible avec des données du web sémantique, les notions d'*IRI* et de *Literal* du format *Turtle* sont introduites. En plus de la possibilité de coder les prédicats ou les constantes par des chaînes de caractères, les prédicats peuvent être codés par des IRIs et les constantes peuvent être codés par des littéraux. Une *IRI* peut s'écrire sous trois formes : sous forme d'une *IRI relative*, d'une IRI *absolue* ou d'un *nom préfixé*. Les IRIs relatives et absolues sont de la forme $< iri >$ (où iri est une *IRIREF* dans la grammaire turtle), et un nom préfixé (*PrefixedName* dans la grammaire turtle) comme son nom l'indique est une chaîne de caractères préfixée par un espace de nom. Un *Literal* est soit une chaîne de caractères, un entier, un double, un décimal ou un booléen. Plus d'informations sur l'IRI (relative ou absolue), les noms préfixés et les littérales

sont consultables dans la documentation turtle accessible via le lien suivant : `http://www.w3.org/TR/turtle/`

Exemple 20 *Soient :*

- *personne(annie) est un atome où personne est le nom du prédicat et annie un terme(ici : constante).*
- $< iri\_relative >(X)$, $< iri\_absolue >(X)$ *et* $foaf : parent(X, Y)$ *avec* $@prefix\ foaf :< http : //xmlns.com/foaf/0.1/ >$ *comme entête qui doit être déclarée, sont des atomes.*
- $2, 'chat', \ldots,$ *sont des littéraux.*
- $X = Y$ *est une égalité entre deux termes (les deux variable* X *et* Y*).*

5.1.1 La grammaire

Syntaxe : Les symboles non terminaux sont entre deux chevrons (<>), et les symboles terminaux sont en gras, deux expressions séparées par une barre verticale ($E_1|E_2$) représente le fait de pouvoir choisir entre E_1 et E_2, les crochets ([,]) pour un choix optionnel et les accolades suivies du symbole étoile ($\{X\}*$) pour permettre la répétition de leur contenu ici X un nombre indéterminé de fois éventuellement zéro fois, pour les accolades suivies du symbole plus ($\{X\}+$) c'est pour permettre la répétition de X un nombre indéterminé de fois mais au moins une fois. Le symbole % sert à faire des commentaires. Le $< I - ident >$ représente une IRI relative et l'IRI absolue est obtenu en ajoutant la base comme préfixe du $(I - ident)$. **Notez** que tout ce qui est en **rouge** représente l'extension de la grammaire réalisée dans le cadre de ce stage.

Éléments de base :

< *uppercase-letter* > : := **A** . . . **Z**
< *lowercase-letter* > : := **a** . . . **z**
< *digit* > : := **0** . . . **9**
< *underscore* > : := **_**
< *letter* > : := < *uppercase-letter* > | < *lowercase-letter* >
< *simple-char* > : := < *letter* > | < *digit* > | < *underscore* >
< *PN_CHARS* > (*voir la grammaire turtle*)
< *space* > : := #**x20**(*où* #**x20** *représente le code Hex du caractère espace*)

Tokens :

< *u-ident* > : := < *uppercase-letter* > {< *simple-char* >}*
< *l-ident* > : := < *lowercase-letter* > {< *simple-char* >}*
< *label* > : := {< *PN_CHARS* > | < *space* >}*

Grammaire globale :

< *document* > : := < *header* >< *body* >

< *header* > : := {< ***import*** >} ∗ {< *base* > | < *prefix* > | < ***computed*** > | < *top* > | < *una* >}∗

< ***import*** > : := **@import** < *IRIREF* >

< *base* > : := **@base** < *IRIREF* >

< *prefix* > : := **@prefix** < *PNAME_NS* >< *IRIREF* > (*pour PNAME_NS* voir la grammaire turtle)

< ***computed*** > : := **@computed** < *PNAME_NS* >< *IRIREF* >

< *top* > : := **@top** < *l-ident* > |**@top** < *IRIREF* >

< *una* > : := **@una**

< *body* > : := {< *statement* >} ∗ |{< *section* >}∗

< *section* > : := **@facts** {< *fact* >} ∗ |**@rules** {< *rule* >} ∗ |**@constraints** {< *constraint* >} ∗ |**@queries** {< *query* >}∗

< *statement* > : := < *fact* > | < *rule* > | < *constraint* > | < *query* >

< *fact* > : := [[< *label* >]] < *not-empty-conjunction* > .

< *rule* > : := [[< *label* >]] < *not-empty-conjunction* > :- < ***rule-body*** > .

< *constraint* > : := [[< *label* >]] ! :- < ***rule-body*** > .

< *query* > : := [[< *label* >]] ?[(< *term-list* >)] :- < ***rule-body*** > .

< ***rule-body*** > : := < *conjunction* > {,**not** < *not-empty-conjunction* >} ∗ | **not** < *not-empty-conjunction* > {,**not** < *not-empty-conjunction* >}∗

< *conjunction* > : := [< *not-empty-conjunction* >]

< *not-empty-conjunction* > : := < *atom* > {, < *atom* >}∗

< *atom* > : := < *std-atom* > | < ***special-atom*** >

< *std-atom* > : := < *predicate* > (< *not-empty-term-list* >)

< ***special-atom*** > : := < *term* >< ***special-predicate*** >< *term* >

< ***special-predicate*** > : := > | < | >= | <= |! = | =

< *term-list* > : := [*not-empty-term-list*]

< *not-empty-term-list* > : := < *term* > {, < *term* >}∗

< *term* > : := < *variable* > | < *constant* > | < ***functional-term*** >

< *predicate* > : := < *l-ident* > | < *IRIREF* > | < *PrefixedName* >

< *variable* > : := < *u-ident* >

< *constant* > : := < *l-ident* > | < *IRIREF* > | < *PrefixedName* > | < *literal* > | < ***special-literal*** >

< ***special-literal*** > : := < *integer* > | < *decimal* > | < *double* > | < *string* > | < *date* > | < ***user-literal*** >

< ***user-literal*** > : := < *string* > ^ ^ < *PrefixedName* >< *string* >

< ***functional-term*** > : := < ***std-functional-term*** > | < ***special-functional-term*** >

< ***std-functional-term*** > : := < *PrefixedName* > (< *not-empty-term-list* >)

< ***special-functional-term*** > : := < ***AdditiveExpression*** >

< ***AdditiveExpression*** > : := < ***MultiplicativeExpression*** > {(+|−) < ***MultiplicativeExpression*** >}∗

$< \textbf{\textit{MultiplicativeExpression}} > ::= < \textbf{\textit{Factor}} > \{(*|/|\%) < \textbf{\textit{Factor}} >\}*$

$< \textbf{\textit{Factor}} > ::= < \textbf{\textit{Unary-functional-term}} > \{\hat{} < \textbf{\textit{Unary-functional-term}} >\}?$

$< \textbf{\textit{Unary-functional-term}} > ::= (< \textbf{\textit{special-functional-term}} >) | < term > | - < term >$

Existant : Nous allons s'arrêter sur quelques éléments :

- Entête :
  - *@base* est une directive Turtle qui sert à compléter les IRIs relatives, elle n'est pas obligatoire, si elle n'est pas déclarée une base par défaut a été définie : $< http://www.lirmm.fr/dlgp/ >$.
  - *@prefix* est aussi une directive Turtle qui permet de déclarer une paire (préfixe, espace de nom) où chaque préfixe rencontré sera remplacé par son espace de nom. Tous les préfixes utilisés doivent être déclarés dans l'entête par cette directive. Il était interdit de déclarer deux fois le même préfixe mais pour mettre en place le mécanisme du *import* nous étions obligés de permettre la déclaration du même préfixe plusieurs fois, ce point est détaillé dans la section import 5.1.2.1.
  - *@top* permet de définir des prédicats qui ont comme sémantique $\top$.
  - *@una* déclare que cette base de connaissance fait l'hypothèse du nom unique(Unique Name Assumption) où si deux constantes sont différentes alors il font référence à deux entités différentes dans le monde.
- Corps :
  - *@facts, @rules, @constraints et @queries* sont des directives qui permettent de déclarer le type des éléments qui viennent juste après(fait, rule, constraint ou query), ces directives servent à prévenir le parser le plus tôt possible des types d'éléments, elles ne sont pas obligatoires mais si on ne les mets pas, le parser aura aucun moyen de savoir si il est entrain de parser un fait ou bien une règle jusqu'à ce qu'il fini d'analyser tout le fait ou toute la tête de la règle(jusqu'à ce qu'il rencontre la fin du fait '.' ou bien l'implication ' :-').
  - Chaque élément peut éventuellement être procédé par un label.
  - Un fait est un ensemble d'atomes, une règle est un ensemble d'atomes impliquant un ensemble d'atomes et les requêtes ainsi que les contraintes négatives sont aussi des ensembles d'atomes.
  - Un atome standard a un nom de prédicat et un ensemble de termes, les termes sont soit des constantes, variables ou littéraux.

Nouveaux éléments : Maintenant nous allons présenter tous les éléments ajoutés à la grammaire :

- Entête :

— *@import* cette directive sert à importer un fichier dlgp existant, cette directive n'est pas obligatoire et nous avons choisi de la mettre toujours en premier, c'est à dire qu'on ne peut pas mettre une autre directive avant elle parce-que nous avons permis le déclaration du même préfixe plusieurs fois. Voir la section 5.1.2.1 pour les détails.

— *@computed* est une directive qui permet de déclarer un couple (préfixe, espace de nom) où l'espace de nom n'est pas une IRI mais une URL vers un fichier de configuration *json*, où il y aura toutes les informations nécessaires pour utiliser ce préfixe. Détails du fonctionnement dans la section computed 5.1.2.2

- Corps :
  — Pour ajouter la négation on introduit *rule-body* qui représente dans notre cas un ensemble d'atomes ou une conjonction de clauses négatives (la négation).

  — Les faits sont toujours considérés comme des ensembles d'atomes, tandis que le corps d'une règle, d'une requête ou d'une contrainte négative est considéré comme un *rule-body*.

  — Concernant les atomes calculés, on distingue deux catégories : la première regroupe les atomes calculés standard, pour cela nous avons gardé la même notation que celles des atomes normaux introduits par un préfixe, ce qui fait qu'au moment de l'analyse syntaxique on ne peut pas savoir si on a à faire à un atome calculé ou à un atome normal, cette ambiguïté est vite levée, en effet il suffit de voir dans l'entête la déclaration du préfixe de cet atome, donc si ce préfixe est déclaré par la directive *@prefix* alors l'atome en question est normal sinon (déclaré par la directive *@computed*) l'atome en question est calculé. La deuxième catégorie regroupe les atomes calculés spéciaux, ce type permet de faciliter à l'utilisateur l'écriture du code, pour utiliser le prédicat calculé inférieur, il faudra déclarer un préfixe et spécifier dans le fichier de config l'emplacement de la fonction qui représente ce prédicat, ensuite il faut écrire le prédicat sous la forme $prefixe : inferieur(terme1, terme2)$. Or grâce à cette distinction, l'utilisateur aura simplement à écrire $terme1 < terme2$ et le parser se chargera de faire l'interprétation. Nous avons pris en compte les prédicats spéciaux inférieur ($<$), inférieur ou égal ($<=$), supérieur ($>$), supérieur ou égal ($>=$), différent ($!=$) et égal ($=$). Voir la section 5.2 pour les détails.

  — Pareil pour les termes calculés, on distingue deux catégories : la première regroupe les termes calculés standards, pour cela nous avons utilisé la même notation que pour les prédicats standards. Par contre dans ce cas l'ambiguïté se manifestera entre une constante introduite par un préfixe et un terme. Cette ambiguïté sera levée de la même manière, c'est-à-dire en vérifiant dans l'entête la déclaration du préfixe. La deuxième catégorie regroupe les termes calculés spéciaux, qui sont là eux aussi pour faciliter l'écriture à l'utilisateur et rendre la syntaxe plus flexible, par exemple pour faire la somme de deux nombres il faudra déclarer un préfixe tel que dans son fichier de config associé, on trouvera les informations nécessaires de la fonction 'somme', ensuite il faut écrire un terme sous la forme

$p(\ldots, prefixe : somme(X, Y))$. Or grâce à cette distinction, l'utilisateur aura à écrire simplement $X + Y$ et le parser se chargera de l'interprétation. Nous avons considéré les fonction usuelles tels que : plus (+), moins (−), mult (∗), division (/), modulo (*mod*) ainsi que la puissance (^). Voir la section 5.2 pour les détails.

— Les constantes peuvent être des littéraux ce qui est a été introduit par $special - literal$, nous avons fixé les types primitifs XSD (int, float, double, string et date) et nous avons donné la possibilité à un utilisateur de déclarer ses propres types.

Exemple 21 *Voici un exemple d'une suite d'instructions dlgp acceptée par la grammaire ci-dessus :*

```
1  %ceci est un commentaire
2  %entete
3  @import </chemin/vers/exemple1.dlgp>
4  @base <http://www.example.org
5  @prefix graphik: <https://graphik.fr/>
6  @prefix xsd: <http://www.w3.org/2001/XMLSchema#>
7  @computed pred:</chemin/vers/fichier/configpred.json>
8  @computed fct:</chemin/vers/fichier/configfct.json>
9
10
11 @facts
12 p(4,1),p(5,"3"^^xsd:integer).
13 graphik:equipe(X,Y).
14
15 @rules
16 [R1] q(X):-p(X,Y).
17 [R1] r(X+1,X+2):-p(X,Y), X<Y, not X=5 .
18
19 @constraint
20 [C1] !:- r(X,X).
21 [C2] !:- r(X,X), pred:f(X,Y).
22
23 @queries
24 [q1]?(X):-q(X).
25 [q2]?(X,Y) :- r(X,Y),X=fct:f(Y),not p(X,Z).
```

5.1.2 Les nouvelles Instructions

Nous avons introduit dans la section précédente les nouveaux éléments ajoutés au langage DLGP, concernant les nouveaux objets c'est à dire la négation, prédicats calculés, et termes calculés ils seront détaillés dans la

section 5.2, et ici nous allons voir les deux nouvelles instructions de déclaration l' import ainsi que le computed.

5.1.2.1 Import

Cette instruction permet comme son nom l'indique d'importer tous les éléments d'un fichier dlgp c'est à dire si on importe un fichier qui contient une entête, un ensemble d'atomes, règles, requêtes et contraintes négatives alors tous ses éléments seront importés et mis dans la base de connaissance. Comme précisé précédemment, pour mettre au point ce mécanisme d'import nous étions obligés de permettre la déclaration du même préfixe plusieurs fois, en effet un fichier 'A' qui déclare un préfixe 'pref' peut importer un fichier 'B' qui déclare à son tour le même préfixe 'pref', alors tout simplement le dernier préfixe analysé sera gardé i.e à chaque fois qu'on rencontre un préfixe déjà déclaré au lieu d'arrêter l'analyse et de renvoyer une erreur, on écrase le préfixe existant. À cause du fait qu'on garde le dernier préfixe rencontré nous avons choisi d'obliger que l'import soit toujours au début du fichier et jamais après une autre déclaration, en effet, si une fichier 'A' déclare un préfixe 'pref' et que ensuite il importe un fichier 'B' qui déclare à son tours le même préfixe 'pref' alors celui qui sera gardé est celui du fichier 'B' or logiquement celui de 'A' devrait l'être.

Exemple 22 *Soient les deux fichier A et B suivant :*

```
1 @import </chemin/vers/B.dlgp>
2 @prefix ex: <https://préfixe/fichier/A>
3
4 ex:p(X).
```

Fichier A

```
1 @prefix ex: <https://préfixe/fichier/B>
```

Fichier B

Avec le mécanisme d'import mis au point, dans l'atome $ex : p(X)$ *le préfixe ex sera remplacé par l'espace de nom* $\ldots/A$ *ce qui donnera l'atome* $< https : /prfixe/fichier/A > (X_0)$*. Supposons que nous n'avons pas permis la déclaration du même préfixe plusieurs fois, dans ce cas cet import ne peut pas fonctionner et retourne une erreur du genre "prefix already declared". Aussi si nous avons donné la possibilité de déclarer des import après d'autres déclarations par exemple on inverse la ligne 1 et 2 du fichier A, dans ce cas dans l'atome* $ex : p(X)$ *le préfixe ex sera remplacé par* $< \ldots/B >$ *et donnera donc l'atome* $< https : /prfixe/fichier/B > (X_0)$ *or le préfixe le plus proche de l'atome et celui se trouvant dans le même fichier à savoir* $\ldots/A$*.*

Principe Pour mettre au point le import nous avons fixé une base de connaissance ainsi qu'un buffer par défaut, pour qu'au moment de l'analyse syntaxique lorsque on a une instruction d'import on récupère le fichier importé, on l'analyse tel que : son entête sera ajoutée dans le buffer et tous ses éléments (atomes, règles,

requêtes et contraintes) dans la base de connaissance. Alors, lorsque l'analyse du fichier importé sera fini on revient pour finir l'analyse du fichier qui l'a importé en sachant que maintenant dans la base de connaissance nous avons tous les éléments du fichiers importé et dans le buffer son entête.

Exemple 23 *Soient les deux fichier A et B suivant :*

```
1  @import </chemin/vers/B.dlgp>
2  @prefix ex: <https:/préfixe/fichier/A>
3
4  ex:p(X).
5  [R1] q(X):-ex:p(X).
6  [q1] ?(X) :- q(X).
7  [C1] !:-r(X).
```

Analyse du fichier A

```
1  @prefix ex: <https:/préfixe/fichier/B>
2  @prefix pref: <https:/deuxième/préfixe/fichier/B>
3  ex:s(X).
4  [R2] q(X):-ex:s(X).
5  [q2] ?(X) :- q(X).
6  [C2] !:-r(X).
```

Analyse du fichier B

Comme le fichier 'A' importe le fichier 'B' alors lorsqu'on analyse le 'A' et qu'on rencontre l'instruction d'import (ligne 1) on vas chercher le 'B' pour l'analyser et ajouter tous ses éléments dans le base de connaissance ainsi que son entête dans le buffer pour ensuite revenir et continuer l'analyse du fichier 'A' (revenir à la ligne 2). Finalement dans la base de connaissance on trouvera les deux atomes $ex : p(X), ex : s(X)$*, les deux règles* R_1, R_2*, les deux requêtes* q_1, q_2 *et les deux contraintes* C_1, C_2*, et dans le buffer on trouvera le préfixe* $pref$ *de 'B' ainsi que ex de 'A' sachant que le préfixe ex de 'B' sera écrasé.*

Nous avons détecté deux problèmes qui se présentent dans certains cas de figure, le premier est les imports cycliques et le deuxième se présente lorsqu'on importe le même fichier plusieurs fois. Pour résoudre le premier problème nous l'avons abordé du point de vue des graphes, et donc nous avons gardé la trace des imports et à chaque fois qu'on détecte un cycle on arrête l'analyse en renvoyant une erreur (sinon l'analyse serait infinie). Concernant le deuxième problème, avant sa résolution et lorsque on importe plusieurs fois un même fichier, ce qui se passe c'est que l'analyse de ce fichier se fait autant de fois qu'il est importé ce qui est inefficace dans le cas où le fichier contient un nombre important d'éléments(atomes, règles, requêtes et contraintes) de plus s'il contient un atome comportant une variable ($p(\ldots, X, \ldots)$), cet atome sera ajouter autant de fois que le fichier est analysé (comme à chaque fois il y a un renommage de variables alors le même atome sera considéré

comme différent à chaque renommage ($p(X_0) \not\Rightarrow p(X_1)$)). De ce fait, lorsque le même fichier est rencontré pour la deuxième fois uniquement son entête sera analysée, ce qui permet de garder le bon ordre concernant les déclarations et de ne pas analyser à nouveau l'ensemble de ses éléments.

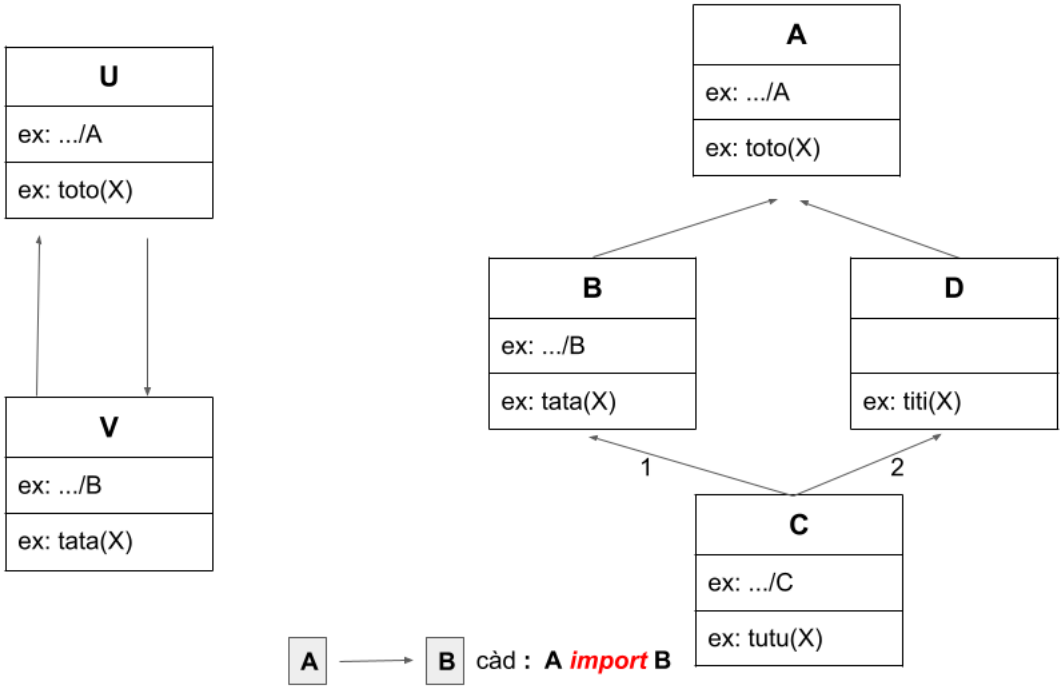


FIGURE 5.1 – Exemple d'import cyclique à droite et d'import multiple à gauche

Exemple 24 *Cette figure montre les deux problèmes qui peuvent se produire, en effet 'U' et 'V' s'importe mutuellement ce qui forme une boucle infini d'import, tandis que le fichier 'C' importe les deux fichiers 'B' et 'D' tel que tout les deux importent le même fichier 'A', dans ce cas, uniquement l'entête du fichier 'A' sera ré-analysée sinon si on choisi de refaire l'analyse de tout le fichier l'atome* $ex : toto(X)$ *sera ajouter deux fois dans la base de connaissance, et si on choisi de ne pas refaire du tout l'analyse alors dans l'atome* $ex : titi(X)$ *le préfixe ex sera remplacer par* $\ldots/B$ *qui est le dernier préfixe analysé ce qui est incorrect car le plus proche est* $\ldots/A$.

5.1.2.2 Computed

Cette instruction permet de déclarer un préfixe qui sera utilisé pour introduire les atomes et termes calculés, elle indique l'emplacement du fichier de configuration json où se trouveront toutes les informations liées à ce préfixe et donc aux atomes et termes introduits par ce préfixe. Chaque atome ou terme calculé est lié à une fonction qui décrit la manière pour pouvoir l'utiliser, et le fait qu'il doit être introduit par un préfixe permet

via ce préfixe d'aller chercher la bonne fonction représentant cet atome ou terme calculé. La différence par rapport à un préfixe normal est que ce type de préfixe a comme espace de noms une URL vers un fichier de configuration json ce qui fait que le traitement effectué lors de l'analyse de cette instruction sera différent. En effet, à chaque fois qu'on analyse une instruction computed on récupère le fichier json on le lit et on récupère toutes les information qui s'y trouvent. On sauvegarde toutes ces informations dans des tables de hachage, pour que, lorsque ce préfixe est utilisé, on puisse chercher dans ses informations l'utilisation adéquate.

Fichier de configuration JSON Ce fichier de config contient toutes les informations concernant l'emplacement des méthodes et classes java liés à un préfixe donné. Il se compose de deux parties, la première partie que nous avons appelée 'elements', elle contient des objets json de la forme :

```
"name":{
        "type":"(predicate or function or literal)",
        "location":{
            "path":"/paths/to/package/",
            "package":"package",
            "class":"class"
                }
        }
```

fichier1.json

tel que : "name" représente le nom du prédicat, fonction ou type que l'utilisateur voudra utiliser à travers le préfixe qui a mené à ce fichier json. Les champs "type", "location" représentent respectivement le type de l'élément ("predicate" pour un prédicat calculé, "function" pour un terme calculé et "literal" pour un type utilisateur), son emplacement ("path" pour le chemin, "package" pour le nom du package et "class" pour le nom de la classe). C'est à dire si nous avons un préfixe "$pref :< .../fichier1.json >$" alors, $pref : name(\ldots)$ désigne un prédicat calculé dont le nom est "name" et le type "predicate" et l'emplacement de la fonction java qui permettra de l'évaluer est précisé dans le champ "location", $p(\ldots, pref : name(\ldots), \ldots)$ désigne un atome qui contient un terme calculé dont le nom est "name", le type est "function" et $p(\ldots, "Z"\hat{}\hat{}pref : name, \ldots)$ désigne un atome qui contient un littéral dont le type est "name" et dont la classe représentant ce type se trouve dans "location". À partir de chaque objet de la partie "elements", on récupère une instance de la classe qui se trouve dans "location" ensuite si le champ "type" est "predicate" ou "function" alors on sauvgarde le couple ($< .../fichier1.json : name,$ *cette instance récupérée* $>$) dans une table de hachage sinon ("type" est "literal") on sauvegarde le même couple mais dans une autre table dédiée aux types. Ces deux tables de hachage seront utilisées par la suite pour invoquer(java reflect) la bonne méthode représentant le prédicat ou terme calculé ou bien créer le bon type. La deuxième partie, que nous avons appelée 'default', contient l'emplacement des classes java(chemin vers la classe, le package où se trouve la classe et le nom de la classe) où il faudra regarder si on veut utiliser un prédicat ou un terme calculé mais que nous ne l'avions pas trouvé

dans la partie "elements". Cette partie est de la forme :

```
"default":{
        "types":{}
        "predicates":{
            "path":"/paths/to/package/",
            "package":"package",
            "class":"class"
                }
        "functions":{
            "path":"/paths/to/package/",
            "package":"package",
            "class":"class"
                }
        }
```

fichier2.json

tel que : dans le champ "predicates" on trouve l'emplacement des prédicats par défaut liés au préfixe 'pref :' avec $pref :< .../fichier2.json >$, dans le champ "fuctions" on trouve l'emplacement des fonctions liés au préfixe 'pref :'. À partir de ces deux champs(objets json) nous allons ajouter à la table de hachage les deux couples $< .../fichier2.json$, *une instance de la classe à partir du champ "predicates"* $>$
$< .../fichier2.json$, *une instance de la classe à partir du champ "functions"* $>$. Le champ "types" est vide parce-que on ne peut pas définir un type par défaut pour tous les littérales (aucun sens). Si l'on rencontre un atome $pref : name(...)$ et que "name" ne figure pas parmis les objets de la partie "elements" alors "name" sera sûrement une méthode de la classe qui se trouve dans l'emplacement fourni par le champ "predictes" dans la partie "default".

5.1.3 Détails d'implémentation

Cette grammaire a été implémentée en utilisant javaCC (java compiler compiler) qui est un compilateur de compilateur, il permet de faire des programmes en langage java, il comporte un générateur de parseur qui permet de créer un analyseur syntaxique. JavaCC prend en paramètre un fichier .jj qui contient entre autres la description de la grammaire et permet de générer une classe java qui représente le parser correspondant.

Après avoir fait l'extension de la grammaire, l'étape suivante est de modifier le fichier .jj décrivant cette grammaire, pour prendre en compte l'ensemble des extensions définies.

Refonte du parser À cause de la complexité des règles de grammaire dans le fichier .jj (qui comportait énormément de code), une première étape avant d'entamer la modification était de réorganiser ce fichier pour rendre la grammaire lisible et facilement modifiable. En effet, dans ce fichier nous avons trouvé des classes

et méthodes qui gèrent les événements qui servent à la création d'objets reconnus, d'autres qui gèrent les IRIs ...etc. Ces classes et méthodes ont rendu le fichier .jj illisible et donc sa modification très coûteuse. Pour remédier à ce problème, nous avons allégé le parser(.jj) de toutes les classes et méthodes qui ne sont pas nécessaires à l'analyse syntaxique. Comme déjà expliqué dans la mission, la difficulté c'est qu'il faut changer le minimum dans le fonctionnement interne du parser pour une intégration rapide dans la version 2 de GRAAL. Pour respecter cette contrainte nous avons simplement déplacé toutes les classes et méthodes qui n'ont rien à voir avec la grammaire dans une classe abstraite (ADlgpItemFactory.java), où nous avons défini toutes les méthodes nécessaires à la création d'objets, c'est à dire que désormais, lorsque le parser reconnaît un objet, il fait appel à une méthode externe qui fera ensuite le même raisonnement qu'auparavant. Cette classe abstraite est spécialisée par une classe (GraalFacory.java) où sont implémentées toutes ces méthodes.

Exemple 25 *Lorsque le parser analyse une instruction de déclaration d'un préfixe :*

```
1  void parsePrefixDecl() :
2  {
3    Token t;
4    String prefix = "";
5    Object ns;
6  }
7  {
8    < PREFIX > t = < PNAME_NS >
9    {
10     prefix = t.image;
11     ns = parseIRIPrefix();
12
13     if (prefixManager.containsPrefix(prefix)){
14       throw this.generateParseException("Prefix already declared: " + prefix);}
15
16     String s = ns.toString();
17     if (s.startsWith("<"))
18     s = s.substring(1, s.length() - 1);
19
20     prefixManager.setPrefix(prefix, s);
21     fireParseEvent(new DeclarePrefixEvent(prefix, s));
22     }
23 }
```

Listing1 : Méthode qui permet de parser un préfixe (.jj AVANT la réorganisation)

```
1  void parsePrefixDecl():
2  {
3    Token t;
```

```
}
{
  < PREFIX > t = < PNAME_NS >
  {
    try {
        factory.buildPrefixDecl(t.image,parseIRIPrefix());}
    catch(BuildException ex) {
        throw this.generateParseException(ex.getMessage());
        }
    }
}
```

Listing2 : Méthode qui permet de parser un préfixe (.jj APRÈS la réorganisation)

Comme vous pouvez le voir dans le listing 1, la méthode fait appel à d'autres méthodes et classes pour gérer l'objet qu'elle est entrain d'analyser (le préfixe ici), sachant que ces méthodes et classes était aussi dans le même fichier (le .jj), elle fait un traitement aussi sur l'IRI associé au préfixe, tandis que dans le listing 2 il y a uniquement un seul appel à la méthode (buildPrefixeDecl) qui permet d'encapsuler le traitement fait dans le listing 1, ce qui a rendu le parser beaucoup plus lisible et généralisable.

Import Voici la méthode invoquée lorsqu'on analyse une instruction d'import : elle est dans le package "fr.lirmm.graphik.graal.io.dlp".

```
public void buildImportDecl(String url) {
        int indexOf=url.lastIndexOf("/");
        try {
        File file=new File(url.substring(0, indexOf+1),url.substring(indexOf+1, url.length()))
    ;
        IMPORT_FILE_STATUS status=this.checkImportsStatus(file);
        switch(status){
        case CLOSE:
            DLGP2Parser parserHead = new DLGP2Parser(this, new FileReader(file));
            try {
                parserHead.header();
            }catch(fr.lirmm.graphik.dlgp3.parser.ParseException e) {e.printStackTrace();}
            return;
        case OPEN:
            throw new ParseException("A cycle was detected when parsing: "+file.getName());
        case NEW:
            imports.put(file, 0);
            DlgpParser parser=new DlgpParser(file);
            kbb.addAll(parser);
            imports.put(file, 1);
            break;
```

```
21          }
```

La variable *status* de type *IMPORT_FILE_STATUS* est une énumération java (OPEN,CLOSE ou NEW). Elle est initialisée par la méthode *checkImportsStatus* qui pour un fichier passé en paramètre renvoi son état, ce qui permet lors de l'import de connaître l'étape du fichier qu'on importe tel que : si le valeur retour de cette méthode est NEW alors le fichier est importé pour la premier fois, CLOSE c'est qu'il est importé au moins une fois et OPEN c'est à dire que le fait d'importer ce fichier va créer un cycle. Dans ce dernier cas on jette une exception pour arrêter l'analyse, dans le cas où on rencontre le fichier pour la première fois (NEW) après l'avoir analysé on rajoute tous ses éléments dans la base de connaissance (ligne 17 et 18), finalement, si le fichier a déjà été rencontré et qu'il ne forme pas de cycle alors on analyse seulement son entête (ligne 10).

Computed Notez que la classe qui gère les instructions computed est *InvokeManager* et celle qui permet de lire et extraire les informations du fichier json est *ConfigFileReader*, elles sont toutes les deux dans le package "fr.lirmm.graphik.dlgp3.parser", et qu'un attribut de type *InvokeManager* est déclaré et il est commun à toutes les déclarations computed.

Lorsque on analyse une instruction @*computed pref* :< *URL* >, la méthode la plus importante invoquée est *getObjectToInvoke* de la classe *InvokeManager* :

```
1  public void getObjectToInvoke(String prefix,String prefixFilePath){
2          JSONObject jsonObject=this.confReader.readJesonFile(prefixFilePath);
3          JSONObject defaultJsonObject=this.confReader.getDefault(jsonObject);
4          JSONObject elementsJsonObject=this.confReader.getElements(jsonObject);
5          if(!elementsJsonObject.isEmpty()) {
6              for(Object key :elementsJsonObject.keySet()) {
7                  JSONObject currentJsonObject=(JSONObject)elementsJsonObject.get(key);
8                  String type=this.confReader.getType(currentJsonObject);
9                  try {
10                     JSONObject locationJsonObject=this.confReader.getLocation(
       currentJsonObject);
11                     switch(type) {
12                     case "literal":
13                         typesTable.put(prefix+((String) key), this.confReader.
       getClassFromJsonObject(locationJsonObject));
14                         break;
15                     case "predicate":
16                         prefixTable.put(prefix+((String) key), this.confReader.
       getClassFromJsonObject(locationJsonObject));
17                         break;
18                     case "function":
```

```
19                          prefixTable.put(prefix+((String) key), this.confReader.
      getClassFromJsonObject(locationJsonObject));
20                          break;
21                      default:
22                          throw new ConfigFileException(type+"not in <literal,predicate,function
      >");
23                          }
24          }
```

On commence par lire le fichier json accessible via l'URL, c'est la méthode *readJsonFile* de la classe *configFileReader* qui s'en charge. Ensuite on récupère les deux parties "elements" et "default" du fichier sous forme d'objets json. Après on parcourt l'objet correspondant à la partie "elements" tel que pour chaque élément et en fonction du champ "type" soit on rajoute dans la table hachage dédiée pour les préfixes couple (clé,valeur) si c'est une fonction ou un prédicat, soit on rajoute dans la table de hachage dédiée aux types ce même couple. Le couple clé est obtenu en concaténant l'URL à laquelle on rajoute # comme suffixe avec le champ "name", et la valeur représente soit l'instance de la classe où se trouve la méthode liée au prédicat ou à la fonction soit simplement l'instance de la classe qui représente le nouveau type utilisateur(information de l'emplacement de cette classe dans le champ "location" et l'instance est créée par la méthode *getClassFromJsonObject* de la classe *configFileReader* qui permet de charger une classe(loadClass) à partir d'un chemin, nom du package et le nom de la classe). Concernant la partie "default" on rajoute seulement deux couples (clé,valeur) avec comme clé l'URL à laquelle on rajoute # comme suffixe pour récupérer facilement l'URL et éventuellement le nom du prédicat, fonction ou type à partir de cette clé, et comme valeur l'instance de la classe où se trouvent les prédicats et fonctions.

5.2 Évaluation d'une requête

Avant de commencer l'évaluation de requête nous allons voir en détail les nouveaux éléments ajoutés au langage DLGP.

Prédicats calculés Nous avons des prédicats qu'on nomme standards qui sont de la forme ($pred : p(\ldots)$) et pour avoir une écriture lisible et flexible, nous avons choisi quelques prédicats fréquents ($>, >=, <=, != et =$) et dont on a permit l'écriture infixée classique ($X < Y$ au lieu de $pred : inferieur(X, Y)$) qu'on a nommé les prédicats spéciaux.

Les standards Lorsqu'on analyse un atome de la forme $pref : p(\ldots)$ la première chose qui est faite est de s'assurer que le préfixe a bien été déclaré pour récupérer l'URL associé à ce dernier. Si $pref :< URL >$ est la

déclaration du préfixe alors notre atome est maintenant de la forme $< URL\#p > (\ldots)$. Ensuite, premièrement, comme nous ne pouvons pas savoir par avance si le prédicat est calculé ou normal, on teste si son préfixe a été déclaré en tant que @*computed* ou @$prefix$ et deuxièmement, comme on ne peut pas savoir si c'est un prédicat calculé ou un terme calculé, on regarde la suite tel que si ce n'est pas un prédicat spéciale$(>, >, \ldots)$ alors on a bien reconnu un prédicat calculé sinon c'est que c'est le terme gauche d'un prédicat spécial.

Exemple 26 *Soit l'exemple suivant où on déclare un préfixe @prefix et un autre @computed ainsi que quelques atomes :*

```
1  @prefix pref: <http://pref/de/>
2  @computed comp:<http://comp/config.json>
3
4  ....p(X,Y),comp:f(X),pref:g(Y).
5  ....p(X,Y),comp:pred(X)...suite
```

Ligne 4 : les deux prédicats $comp : f(X)$ *et* $pref : g(Y)$ *sont différentiables grâce à la déclaration, en effet comp est déclaré en tant que computed ce qui fait que l'atome* $comp : f(X)$ *est considéré comme un atome calculé et* $pref$ *est déclaré en tant que* $prefix$ *donc l'atome* $pref : g(Y)$ *est considéré comme un atome standard.*

Ligne 5 : selon la suite $comp : pred(X)$ *sera un terme calculé si la suite est l'un des prédicats spéciaux* $(<, >, \ldots)$ *sinon ça serait un atome calculé.*

Dès le moment où on est sûr que le prédicat est calculé on le crée. Un prédicat calculé (classe *ComputedAtom*) contient un attribut de type *InvokeManager* qui permet de récupérer dans la table de hachage dédiée aux préfixes les informations nécessaires concernant la méthode java liée à ce prédicat, ainsi que toutes les informations indispensables à savoir le nom du prédicat, ses termes. Et il contient en plus par rapport aux atomes normaux, une méthode *eval* qui permet de l'évaluer dans le contexte d'une substitution. En effet, cette méthode prend en paramètre une substitution et crée une image des termes de cet atome. Ensuite elle invoque la méthode java liée au prédicat de cet atome avec comme paramètres cette image, c'est à dire l'ensemble des littéraux de cette image, en respectant le nombre et les types de paramètres. Le nombre sera l'arité du prédicat et les types seront l'ensemble des types des images de ses termes c'est dire l'ensemble des types des littéraux image de ces termes là, tel que une table de hachage a été définie et gérée dans la classe *InvokeManager* qui permet de matcher les types GRAAl avec des types java, par exemple si l'atome est de la forme $pref : p(X, Y)$ et que le type GRAAL de l'image de X dans une substitution σ est t_i alors dans cette table de hachage il y a un couple (clé,valeur) tel que : la clé sera le type t_i et la valeur est la classe java correspondante. L'évaluation d'un tel atome comme tout prédicat permet de renvoyer un résultat booléen.

Notez que les prédicats calculés sont interdits dans les faits ou dans la tête d'une règle. En effet, un prédicat calculé se résume au final à un booléen et le booléen est un littéral et comme on ne peut pas avoir un littéral

tout seul comme fait (un fait est un ensemble d'atomes), aussi on ne peut pas déduire un littéral tout seul (on déduit des faits).

Exemple 27 *Soit le prédicat calculé suivant :*

```
1 @computed pred:<chemin/vers/fichier/A.json>
2
3 ... pred:appartient(X,Y).
```

et soit la méthode appartient *lié à ce prédicat et dont le chemin est connu grâce au préfixe "pred :".*

```
1 public classe MesPredicats{
2     public boolean appartient(String X, String Y){
3         return (Y.indexOf(X)!=-1)
4             return true;
5         else
6             return false
7     }
8 }
```

et soit la substitution $\sigma = \{X :' a', Y :' France'\}$*, L'évaluation de l'atome de ligne 3 se fait par l'invocation de la méthode* eval *qui vas créer l'image de cet atome dans la substitution* σ *à savoir* $pred : appartient("a","France")$ *ensuite elle utilisera l'attribut de type* InvokeManager *pour rechercher l'instance de la classe (ici* MesPredicts*) où se trouve la méthode* appartient *liée à ce prédicat. Donc après l'invocation de la méthode avec les bons paramètres ainsi que les bons types de paramètres(ici le type des deux littéraux images de ces termes est* $xsd :: string$ *or dans la table de hachage dédier il y a l'information qui dit que le type* $xsd :: string$ *représente le type String de java) elle retournera un booléen (ici true).*

Les spéciaux Avant toutes analyses syntaxique il y a une suite de traitements automatiques où on trouve entre autres l'ajout dans la tables dédiée aux préfixes un ensemble de couple (clé,valeur) où les clés représentent l'ensemble des prédicats standards (leurs nom complet) et les valeurs représentent l'ensemble des instance de classe où on trouve les méthodes qui décrivent comment évaluer ces prédicats (dans notre cas toutes ses méthodes se trouvent dans la même classe), et ça grâce à la lecture d'un fichier de config json par défaut qui contient toutes ses informations, et dont le traitement se fait de la même manière qu'un fichier de config utilisateur. Lorsqu'on analyse un atome de la forme $X < Y$, la première étape consiste à récupérer le nom complet du prédicat pour se retrouver avec une notation préfixée $nomPred(X, Y)$. Le nom complet du prédicat est la concaténation de la base par défaut (http ://www.lirmm.fr/dlgp/) avec le nom du prédicat correspondant

préfixé par #, sachant que (< inf), (> sup), ..., (! = diff) sont les noms des prédicats choisis. Comme, dans la phase d'avant l'analyse, l'ensemble d'informations concernant les prédicats spéciaux a été ajouté dans la table de hachage dédiée aux préfixes, alors aux moment de l'évaluation d'un atome spéciale dans le contexte d'une substitution le même raisonnement sera repris car on se retrouve avec un prédicat standard dont le nom complet existe forcément dans la table de hachage.

Exemple 28 *Soit le prédicat calculé suivant :*

```
@computed pred:<chemin/vers/fichier/A.json>

... X<Y.
```

et soit la méthode inf *liée à ce prédicat qui est prédéfinie et dont le chemin est connu grâce au fichier de config par défaut dédié au prédicats spéciaux.*

```
public class MesPredicatsSpeciaux{
    public boolean inf(Integer X, Integer Y){
        return X<Y;
    }
}
```

et soit la substitution $\sigma = \{X : 3, Y : 2\}$. *Lors de l'analyse, le symbole* < *sera remplacer par son complet et donc l'atome par une écriture préfixé, dès qu'on obtient le nom complet de ce prédicat spéciale ici http ://www.lirmm.fr/dlgp/#inf, et donc un atome standard http ://www.lirmm.fr/dlgp/#inf(X,Y), l'évaluation suivra le même principe que pour les atomes calculés standard et renverra un booléen (ici false).*

Termes calculés De même, pour les termes calculés nous avons les standards qui sont de la forme $fct : f(\ldots)$ et pour avoir une écriture plus simple nous avons défini quelques termes qu'on a nommé spéciales tels que, $+, -, *, /, mod, \hat{}$ et dont on a permis l'écriture infixée par exemple on écrit $X + Y$ au lieu d'avoir à écrire $fct : somme(X, Y)$.

Les standards Lorsqu'on analyse un terme de la forme $fct : f(\ldots)$ la première chose qui est faite est de lever l'ambiguïté parce-que on peut avoir une constante sous la forme $fct : a$, ensuite le raisonnement est exactement le même que celui adopté pour les prédicats calculés standards mis à part deux points importants : le premier point réside sur le fait qu'un terme calculé renvoi un littérale et non pas un booléen. Cette différence est prise en compte par la création d'un littéral à partir d'une valeur de retour d'une méthode java(la méthode liée au terme calculé), et ça grâce à une table de hachage inverse à la table dédiée aux types qui permet de

traduire une bijection entre les types java et les types GRAAL, c'est à dire si on a un terme calculé de la fome $fct : f(X, Y)$ et que la valeur retour de la méthode f liée à ce terme est de type t_i sachant que t_i sera un type java alors il y a forcément dans cette table inverse un couple(clé,valeur) où la clé sera le type t_i et la valeur sera le types GRAAL correspondant. En effet, comme tous les types primitifs ont été prédéfinis à savoir ($xsd :: integer$, $xsd :: string$, $xsd :: decimal$, $xsd :: double$, $xsd :: boolean$ et $xsd :: date$), ils sont dans la table dédiée aux types. Par exemple on trouvera le couple ($< xsd :: integer, java.lang.Integer >$) dans cette table. A chaque fois qu'un utilisateur défini son propre type, un couple sera ajouté à cette table alors dans la table inverse on trouvera forcément les types retours des méthodes liées aux termes calculés, à moins que l'utilisateur n'utilise un type non primitif et qu'il n'aurait pas déclaré au préalable, auquel cas ce serait une erreur de sa part. Cette table inverse est faite sur le principe qu'il y a une bijection entre ses deux ensembles de types(java et GRAAL) car c'était l'hypothèse de départ la plus facile à mettre en place. Cependant, un utilisateur pourrait vouloir définir un type propre à lui qui serait le type d'une valeur retour d'une méthode qu'il aura définie. Par exemple si un utilisateur crée son propre type *compoundInteger* qui représente des entiers composés, et donc il va vouloir par exemple les représenter sous forme de *String* java ce qui enlèvera la bijection, car dans ce cas le type *String* de java n'est plus que *xsd : :string* mais aussi *compoundInteger*. Le deuxième point, est le fait que contrairement aux prédicats calculés un terme calculé peut avoir un autre terme calculé dans son ensemble de termes, c'est à dire qu'il peut être sous la forme $fct : f(\ldots, fct : g(\ldots), \ldots)$ avec $(fct : f)$ et $(fct : g)$ comme des termes calculés. Cette imbrication est gérée par une méthode récursive qui permet dans le contexte d'une substitution d'évaluer les termes en commençant du plus profond.

Exemple 29 *Soit le terme calculé suivant :*

```
@computed fct:<chemin/vers/fichier/B.json>

... p(fct:somme(fct:longueur(X),fct:longueur(Y))).
```

et soit la méthode longueur *qui calcule la longueur d'une chaîne de caractères qui est liée au terme* fct : *longueur ainsi que la méthode* somme *qui calcule la somme entre deux entiers et qui sont accessibles grâce aux informations du fichier de config B.json :*

```
public Integer longueur(String s){
    return s.length();
}
public Integer somme(Integer X, Integer Y){
    return X+Y;
}
```

et soit la substitution $\sigma = \{X : "Montpellier", Y : "Comdie"\}$*. Alors l'évaluation du terme* $fct : somme(fct : longueur(X), fct : longueur(Y))$ *se fera dès lors que l'évaluation des deux termes* $fct : longueur(X)$ *et* $fct : longueur(Y)$ *sera faite. Dans notre cas, après la création de l'image des termes les plus profonds dans la substitution* σ *on aura* $fct : longueur("Montpellier")$ *et* $fct : longueur("Comdie")$*, ensuite leur évaluation donnera respectivement 11 et 7, donc on aura* $fct : somme(11, 7)$*, ce qui donnera 18.*

Les spéciaux De même que pour les prédicats spéciaux, dans la phase d'avant analyse on trouve entre autre l'ajout dans la table des préfixes l'ensemble de couple (clé, valeur) où les clés représentent l'ensemble des noms complets de termes calculés prédéfinis, et l'ensemble des valeurs représentent l'ensemble des instances de classes où on trouve les méthodes qui décrivent comment évalue ses termes(dans notre cas toutes ses méthodes se trouvent dans la même classe). Et ça grâce au même fichier de config json que celui utilisé pour les prédicats spéciaux (Comme ce sont des prédicats et termes prédéfinis et qu'ils suivent globalement le même principe alors on a choisi de mettre un seul fichier de config pour les deux). Lorsqu'on analyse un terme de la forme $X + Y$, au départ, le même raisonnement que pour les prédicats spéciaux est mise au point avec $\{(+\ add), (-\ sub), (*\ mult), (/\ div), (mod\ mod)$ et $(\hat{}\ power)\}$ comme étant les noms choisis, et dès qu'on obtient un terme calculé standard le raisonnement sera alors le même que pour les termes calculés standards.

Exemple 30 *Soit le terme calculé suivant :*

```
1  ... p(X,X+Y).
```

et soit la méthode java add *liée à ce terme qui prend en paramètre deux réels et renvoi leurs somme :*

```
1  public Double add(Double X, Double Y){
2      return X+Y;
3  }
```

et soit la substitution $\sigma = \{X : 2.5, Y : 1.3\}$

Lors de l'analyse, le symbole + sera remplacer par son nom complet et donc le terme par une écriture préfixé, dès qu'on obtient le nom complet de ce terme spéciale ici http ://www.lirmm.fr/dlgp/#add, et donc un terme standard http ://www.lirmm.fr/dlgp/#add(X,Y), l'évaluation suivra le même principe que pour les termes calculés standard et renverra une valeur de type java Double qui sera traduite a un littérale de type GRAAL $xsd :: double$ (ici 3.8).

La négation La négation est de la forme $not\ a_1, not\ a_2, not\ a_k$ où les a_i sont des ensembles d'atomes, lorsque on analyse une négation, et comme la création d'ensembles d'atomes se fait par des évènements, dès qu'on

rencontre la chaîne ",not" ou "not" car par exemple on peut avoir une requête avec un corps comportant des atomes positifs et d'autres négatifs comme on peut l'avoir avec seulement des atomes négatifs, on stocke ses évènements liés à la créations des atomes de chaque ensemble a_i dans une liste de liste (la négation est un ensemble de clause négative) pour qu'au moment de la création on garde bien l'ordre des atomes dans les clauses et des clauses dans l'ensemble de clauses, c'est à dire on vas avoir une liste de liste de la forme $[[e_1, e_2, \ldots, e_k], [e_1, e_2, \ldots, e_p], [e_1, e_2, \ldots, e_s]]$ où les e_j sont les évènements déclenchés pour toutes atomes de a_i. Pour résumer, la création d'un atome positif se fait via un évènement qui dit aux écouteurs d'évènements de créer un atome qui a comme prédicat p et comme ensemble de termes $(t_1, \ldots, t_i)$, même principe adapté pour la création d'un atome négatif tel que : on dit aux écouteurs d'évènement qui créent les atomes, voici un atome qui a comme prédicat p et comme termes$(t_1, \ldots, t_i)$ par contre voici sa position dans la conjonction négative, ce qui fait que l'atome est négatif et qu'il sera mit dans la bonne place.

Exemple 31 *Soit la négation suivante not* $p(X, Y), X < Y, not\ q(Z)$*, l'analyse de cette négation se fera d'abord par la création d'une liste de liste* $[[e_1, e_2], [e_3]]$ *avec* e_1 *l'évènement qui permet de créer l'atome* $p(X, Y)$*,* e_2 *pour* $X > Y$ *et* e_3 *pour* $q(Z)$*, à chaque fois qu'un événement est déclenché l'écouteur d'évènement est prévenu de la position de l'atome dans la conjonction négative, donc lorsque* e_1 *sera déclenché une paire* $(i = 1, j = 1)$ *sera envoyé au écouteurs d'évènements où* i *représente la ligne et* j *la colonne de l'atome, c'est à dire que lorsque l'atome* $p(X, Y)$ *sera créé on saura que c'est un atome négatif et on connaîtra sa position, ce qui permet d'avoir à la fin une liste de liste* $[[a_1, a_2], [a_3]]$ *où les* a_i *sont les atomes créés, dans notre exemple ça serait* $[[p(X, Y), X < Y], [q(Z)]]$*.*

Notez que lors de la création d'un atome il y a un simple test qui est fait qui consiste à regarder si on a reçu une paire (i, j) dans ce cas on crée un atome négatifs qu'on met à la bonne place sinon on crée un atome positif.

Détails techniques Dans GRAAL, au moment de l'analyse et lorsque on reconnaît un atome, un évènement est créé contenant toutes les information nécessaires pour la création de cet atome, quand cet évènement sera déclenché il vas envoyer toutes ses information à un écouteur d'évènement qui vas créer l'atome correspondant.

```
protected class FindsAtomEvent implements ParseEvent
      {
        Object predicate = null;
        Object [] terms = null;
        public FindsAtomEvent(Object predicate, Object [] terms)
        {
          this.predicate = predicate;
          this.terms = terms;
        }

```

```
11          public void fire(ParserListener listener)
12          {
13            listener.createsAtom(predicate, terms);
14          }
15        }
```

```
1  public void createsAtom(Object predicate, Object[] terms) {
2          List<Term> list = new LinkedList<Term>();
3          for (Object t : terms) {
4              list.add(createTerm(t));
5          }
6
7          atom = new DefaultAtom(createPredicate(predicate, terms.length),
8                  list);
9
10         this.addAtomInRightList(atom);
11     }
```

La classe *FindsAtomEvent* représente l'évènement qui se crée lorsqu'un atome est rencontré, il comporte un prédicat et une liste de termes (ce qu'un atome contient), lorsque il se déclenche, la méthode *createsAtomes* de la classe *AbstracteDlgpListener* sera invoquée qui permet de créer un atome (classe *DefaultAtom*) avec les information nécessaires(prédicat et liste de termes).

Les atomes calculés Lorsque on reconnaît un atome calculé le même raisonnement est adapté, par contre ce n'est ni le même évènement qui sera créé ni la même méthode qui sera invoquée.

```
1  protected class FindsComputedAtomEvent implements ParseEvent
2        {
3          Object predicate = null;
4          Object [] terms = null;
5          public FindsComputedAtomEvent(Object predicate, Object [] terms)
6          {
7            this.predicate = predicate;
8            this.terms = terms;
9          }
10
11         public void fire(ParserListener listener)
12         {
13             String prefix=(String) predicate;
14             prefix=prefix.substring(0,prefix.indexOf(":")+1);
```

```
15              listener.createsComputedAtom(predicate, terms,invokeManager);
16          }
17      }
```

```
1  public void createsComputedAtom(Object predicate, Object[] terms,Object invoker) {
2          List<Term> list = new LinkedList<Term>();
3          for (Object t : terms) {
4              list.add(createTerm(t));
5          }
6
7          cpAtom = new ComputedAtom(createPredicate(predicate, terms.length),
8                  list,invoker);
9          this.addAtomInRightList(cpAtom);
10     }
```

La classe *FindsComputedAtomEvent* représente l'évènement qui sera créé lorsqu'on analyse un atome calculé, et il comporte en plus du prédicat et de la liste de termes un objet de type *InvokeMnagaer* qui sert pour son évaluation. Quand cet évènement sera déclenché la méthode *createsComputedAtom* de la classe *AbstracteDlgpListener* sera invoquée ce qui permet de créer un atome calculé (la classe *ComputedAtom*) avec les informations reçues à savoir le prédicat, la liste de termes et l'objet qui sert à son évaluation. Concernant les atomes spéciaux, un autre évènement sera créé (*FindsSpecialAtomEvent*) qui aura la particularité d'avoir à la place de la liste de termes seulement deux termes (terme de droite et terme de gauche), et lorsque cet évènement se déclenche une autre méthode sera invoquée (*createsSpecialAtom*) qui met les deux terme reçu dans une liste pour ensuite crée un atome calculé standard (*ComputedAtom*).

La méthode *eval* de la classe *ComputedAtom* se charge de l'évaluation d'un atome calculé dans le contexte d'une substitution.

```
1  public boolean eval(Substitution s) {
2          boolean result=false;
3          this.substituteComputedAtomTerms(s);
4          Class<?> tempType;
5          String predicate=this.getPredicate().toString();
6          String fullPredicate=predicate.substring(0,predicate.indexOf("\\"));
7          try {
8              Lookup lookup = MethodHandles.lookup();
9              MethodType mt = MethodType.methodType(boolean.class,this.paramsType);
10             tempType=(Class<?>)this.invoker.getInvokerObject(fullPredicate);
11             MethodHandle method = lookup.findVirtual(tempType,this.predicateName, mt);
12             MethodHandle methodHandle = method.asSpreader(Object[].class,this.paramsType.
```

```
        length);
13              result=(boolean)methodHandle.invoke(tempType.newInstance(),this.listParams);
14              }
15              catch ...
16  }
```

D'abord on invoque la méthode *substituteComputedAtomTerms* de la même classe qui permet de créer l'image des termes et leurs types à partir de la substitution passée en paramètre ainsi que du table de types. Ensuite une fois l'image créée et les types aussi, on récupère le nom du prédicat et on demande via l'objet de type *InvokeManager* à la table des préfixe l'instance de la classe ou se trouve le méthode dont le nom est le nom du prédicat et qui accepte les paramètres de type : les types qu'on a créés, si une tel méthode avec les bons types existe alors elle sera invoqué avec comme paramètre l'image des termes qu'on a créé sinon une exception sera levée ("non such function...").

Les termes calculés Notez que la création de termes ne passe pas par des évènements comme le cas des atomes.

Lorsque on reconnaît un terme calculé standard une méthode *buildFunctionalOrIriTerm* de la classe *ADlgpItemFactory* est invoquée :

```
1  public Object buildFunctionalOrIriTerm(Object nameFunction,ArrayList<Object> listTermFunction,
       String prefix) {
2          if(isComputed(prefix)) {
3              return new DefaultFunctionalTerm(nameFunction.toString(),invokeManager,
       listTermFunction.toArray(new Object [ listTermFunction.size() ]));
4          }else {
5              return nameFunction;
6          }
7      }
```

Cette méthode permet d'enlever l'ambiguïté entre un terme calculé et une constante tel que : si c'est une constante (sans paramètre) alors elle renvoi cette constante sinon elle crée un terme calculé(la classe *DefaultFunctionalTerm*) avec comme arguments le nom du terme et la liste de ses termes. Ce terme sera un élément dans la liste des termes d'un atome, si cet atome est calculé alors l'évaluation de ce terme se fera lorsque on voudra évaluer cet atome, sinon (cet atome est normale) alors ce terme sera évalué lorsque on cherchera l'image de cette atome par rapport à une substitution. Dans les deux cas, l'évaluation est faite par l'invocation de la méthode *eval* de la classe *DefaultFunctionalTerm* :

```
1  public Literal eval(Substitution s) {
2
3      Literal result=null;
4      this.substituteFunctionalTerm(s);
5      Class<?> tempType;
6      try {
7          Lookup lookup = MethodHandles.lookup();
8          MethodType mt = MethodType.methodType(Object.class,this.paramsType);
9          tempType=(Class<?>)this.invokerObject.getInvokerObject(this.fullFunctionName);
10         MethodHandle method = lookup.findVirtual(tempType,this.functionName, mt);
11         MethodHandle methodHandle = method.asSpreader(Object[].class,this.paramsType.length);
12         Object objectResult=methodHandle.invoke(tempType.newInstance(),this.listParams);
13         result=new DefaultLiteral(new DefaultURI(this.invokerObject.getGraalType(objectResult.
       getClass()))),
14                 objectResult);
15     catch ...
```

Premièrement, cette méthode fait appel à la méthode *substituteFunctionalTerm* de la même classe qui permet de calculer l'image des termes de ce terme ainsi que leurs type dans le contexte d'un substitution, par contre un terme calculé peut dans sa liste de termes d'autre termes calculé ce qui fait que le calcule d'images n'est plus le même que pour les prédicats calculés, c'est pour ça qu'il y a une classe *FctTermList* qui contient a son tours une méthode *eval* qui permet de gérer cette récursive tel que : on lui passant une substitution et sachant qu'elle connaît la liste des terme du terme en question elle parcourt cette liste où pour chaque terme si c'est un littérale alors elle le renvoie, sinon si c'est une variable elle calcule son image dans la substitution sinon si c'est un terme calculé dans ce cas elle évalue ce terme la dans le contexte de cette substitution et donc le résultat de ce terme sera son image dans cette substitution. Deuxièmement, elle récupère l'instance de la classe où se trouve la méthode liée à son terme et elle vérifie qu'une méthode ayant comme nom le nom du terme et comme types de paramètres les types retrouvés, ensuite elle l'invoque en lui passant comme paramètre l'image calculée. Dernièrement, comme le résultat serait un littérale, alors elle crée un littérale (*DefaultLiteral*) à partir du résultat obtenu et en se servant de la table inverse des types (la méthode *getGraalType* de la classe *InvokeManager* prend en entrée un type java et renvoi son type GRAAL correspondant).

Lorsque on reconnaît un terme calculé spéciale une méthode *buildSpecialFunctionalTerm* de la classe *ADlgpItemFactory* est invoqué qui permet de chercher le nom complet du terme spéciale ensuite elle crée un terme calculé standard à partir des argument passé en paramètre à savoir, l'opérande gauche, une liste d'opérateurs et une liste d'opérandes tel que : si la liste d'opérandes est vide alors l'opérateur est unaire, sinon pour chaque opérande dans la liste des opérandes on prend l'opérateur qui se situe au même niveau dans liste des opérateurs et on crée le terme calculé correspondant.

Exemple 32 *Soit le terme calculé spéciale $X + Y * Z$, comme dans la grammaire nous avons prit en en compte la priorité entre les opérateurs, alors les arguments envoyés a la méthode serait : (opérande gauche : Y) (liste des opérateurs :[*,+]) et (la liste des opérandes : [Z,X]) alors le premier terme calcumé qui sera créé est //www.lirmm.fr/dlgp/mult(Y,Z) car Y et l'opérateur * se situe au même niveau ensuite le dernier terme calulé qui sera créé est //www.lirmm.fr/dlgp/add(X,//www.lirmm.fr/dlgp/mult(Y,Z))*

Une fois le terme standard est créé le raisonnement pour les termes standards calculés est appliqué.

La négation Lorsque on reconnaît une négation, l'événement *FindsNegativeConjucntion* est créé, qui contient la liste de listes des événements crée pour chaque atome de chaque clause négative, lorsque cette événement est déclenché une méthode *warnDlgpListener* de la classe *AbstractDlgpListener* est invoquée pour chaque atome avec comme paramètre la paire (i,j) représentant l'indice de l'atome dans la négation, cette méthode permet simplement de mettre à jour la position de l'atome qui vas être créé. Une fois que l'atome sera créé, un test sera effectué tel que : si les indices ne sont pas mis à jour (initialisé à -1) alors l'atome qu'on a créé est un atome positif et rien de tout ça ne s'est passé sinon c'est que l'atome créé est négatif il sera donc mis dans une liste de liste (par respect aux indices) d'atomes représentant la négation qu'on a reconnu.

5.2.1 Principe

Une requête n'est plus vue comme un ensemble d'atomes uniquement mais comme un ensemble d'atomes et un ensemble de filtres, les filtres sont soit des atomes calculés soit une négation. Donc une requête est de la forme $?(X_1, \ldots, X_k) : -(A_1, \ldots, A_l), (F_1, \ldots, F_n), not\ C_1, \ldots, not\ C_p$ où les X_i représentent les variables réponses de la requête qui doivent obligatoirement figurer dans la partie normale de la requête, les A_i représente l'ensemble des atomes normaux de la requête, ils forment la partie normale de la requête (un atome normal c'est un atome qui n'est pas un filtre). Si on a un atome normal et qu'il contient un terme calculé alors cet atome se réécrit en un atome normal sans termes calculés et un filtre, par exemple si $p(X, fct : f(Y))$ est un atome normale et que $fct : f(Y)$ un terme calculé alors on peut le réécrire sous forme $p(X, Z), Z = fct : f(Y)$ et on voit bien que $p(X, Z)$ est un atome normal et $Z = fct : f(Y)$ est un atomes calculé. Maintenant on interdit les termes calculés dans les atomes de la partie normale de la requête, et ce mécanisme de réécriture a été fait uniquement pour les têtes de règles lors du chaînage avant(voir le chapitre 5.3.3) et il sera fait aux requêtes prochainement, par contre il faudra prendre en compte le fait que si on veut réécrire la requête $p(X, fct(Y)), q(V, fct : f(Y))$ la réécriture donnera $p(X, Z_0), Z_0 = fct(Y), q(V, fct : f(Y)), Z_1 = fct : f(Y)$ alors que, idéalement la réécriture devrait donner $p(X, Z_0), q(V, Z_1), Z_0 = Z_1 = fct : f(Y)$ donc prendre en compte le fait que $fct : f(Y)$ a été substitué par la même variable. Les F_i représentent l'ensemble des prédicats calculés de la requête, la partie calculé de la requête, évidemment on peut avoir des termes calculés dans cette partie et toutes les variables apparaissant dans la partie calculé de la requête doivent apparaître dans au moins un atome de la partie

normale de la requête. Les C_i représente les clauses négatifs de la requête qui sont vues comme des filtres qui s'évaluent dans le contexte d'une substitution, chaque clause peut être vue comme une requête booléen tels que a chaque fois qu'une de ces requête a une réponse dans le contexte d'un substitution alors cette substitution est éliminé de l'ensemble de réponses de la requête principale.

Lorsqu'on reconnaît une requête un événement *StartsObjectEvent* sera créé avec comme information le type de l'objet reconnu parmi (FACT,QUERY, RULE, NEG_CONSTRAINT et UNKNOWN) donc le type sera QUERY et éventuellement le nom de la requête, le déclenchement de cet évènement permet d'initialiser les attributs où nous allons mettre l'ensemble des parties de la requête avant sa création à savoir un attribut *normalAtomSet* où on mettra la partie normale de la requête, un autre attribut *computedAtomSet* où on mettra la partie calculée de la requête, un autre *atomSet* où on mettra la partie normale avec la partie calculée, un autre *answerVars* pour les variables réponses et un dernier *negConjunction* pour la partie négative. Ensuite à chaque fois qu'on reconnaît un élément(la manière de créer cet élément est détaillé dans la section précédente), en fonction de sa nature il sera ajouté dans l'un de ses attributs tels que : si on reconnaît des variables réponses alors celles ci seront ajoute dans l'attribut *answerVars*, si on reconnaît un atome normale alors celui-ci sera ajouté dans l'attribut *normaleAtomSet* ainsi que dans *atomSet* , si on reconnaît un atome calculé il sera ajouté dans l'attrubut *computedAtomSet* ainsi que *atomSet*, en effet l'attribut *atomSet* servira pour la création de la requête tel que après la création il y aura une séparation entre ses deux partie(normale et calculé) et ce mécanisme qui n'est pas très optimiste est fait pour ne pas modifier le constructeur de la requête conjonctive, on aurait pu spécialiser la classe qui représente les requêtes conjonctives pour avoir une requête conjonctive avec atome calculés par exemple mais comme les atomes calculés sont considérer comme la requête conjonctive est un élément centrale il est utilisé différentes endroit dans le logiciel et ça s'inscrit dans le cadre de la modification minimale des choses excitantes pour une intégration plus facile dans la V2 de GRAAL. Enfin, si on reconnaît une négation, elle serait ajoutée dans l'attribut *negConjunction* par respect aux positions des atomes dans les clauses négatives.

Après avoir reconnu la fin de la requête un évènement *ConjunctionEndsEvent* sera créé qui aura comme information le type de l'objet qui vient d'être reconnu à savoir une requête(QUERY). Le déclenchement de cet évènement permet d'invoquer la méthode *endConjunction* de la classe *AbstracteDlgpListener* qui en fonction du type de l'objet reconnu fera des choses avec les trois attributs qui ont été remplis. Dans le cas qui nous intéresse c'est à dire le cas où on a reconnu une requête, la première chose c'est de tester que toutes les variables réponses de la requête appariassent bien dans la partie normale de la requête, la deuxième chose c'est de tester que toutes les variables qui apparaissent dans la partie calculé de la requête apparaissent bien dans la partie normale de la requête, ses deux testes seront fait grâce aux deux attributs *computedAtomSet* et *normaleAtomSet*. Enfin si la partie négative est vide alors elle crée une requête conjonctive (la classe *DefaultConjunctiveQuery*) en lui passant comme arguments la liste de variables réponses, les atomes normaux et calculés (*atomSet*) et éventuellement un nom de la requête sinon elle crée une requête conjonctive avec négation (la classe *Default-*

ConjunctiveQueryWithNegation) cette classe est une spécialisation de la classe *DefaultConjunctiveQuery*, cette fois nous avons fait une spécialisation à la place du mécanisme précédent parce-que si on avait mit les atomes négatifs avec les positifs alors au moment de séparation on ne sera plus le positif et le négatifs et on perdra leurs ordre aussi, ce qui fait qu'on passe par cette classe mais au finale il vas simplement servir d'intermédiaire pour passer la partie négative de la requête pour qu'à la fin on aura une requête conjonctive avec une partie négative.
Notez qu'une requête peut être vide, et que sa partie normale peut l'être aussi mais dans ce la partie calculée doit être aussi vide, et que la partie négative peut être vide aussi.

5.2.2 Algorithme Simple

Maintenant qu'on a créé une requête avec un ensemble de variables réponses, une partie normale, une autre calculée et une partie négative nous allons voire comment répondre à ce type de requêtes dans une base de faits :

Algorithme Nous avons mis en place un algorithme simple pour répondre à une requête de la forme $Q = ?(\mathcal{X}) : -\mathcal{N}, \mathcal{F}, \mathcal{C}$ dans une base de faits F où $\mathcal{X}$ est l'ensemble des variables réponses, $\mathcal{N}$ la partie normale de la requête, $\mathcal{F}$ la partie calculée et $\mathcal{C}$ la partie négative, tel que :

Premièrement, si la partie calculée de la requête ainsi que la partie négative sont vides, dans ce cas on a à faire à une requête normale c'est à dire sans aucun nouveau objet parmi les objets qui ont été rajoutés, donc il suffit de faire appelle a l'algorithme de calcul d'homomorphismes qui a été mis en place et qui permet de renvoyer un ensemble de substitutions de la requête dans une base de fait. Deuxièmement, on a aux moins une des deux parties calculées ou négatives qui n'est pas vide, en d'autres termes on a aux moins un filtre, dans ce cas un algorithme simple mis en place qui consiste à créer une requête conjonctive à partir de la partie normale de la requête, ensuite à l'aide de l'algorithme de calcul d'homomorphismes existant, on calcule toutes les substitutions réponses à cette requête dans une base de faits. Une fois qu'on a l'ensemble des substitution de la partie normale on commence à les filtrer tel que : pour chaque substitution de cet ensemble : pour chaque élément de la partie calculée c'est à dire pour chaque atome calculé on l'évalue dans le contexte de la substitution en cours et donc on garde cette substitution si et seulement si cette évaluation est positive, ensuite, on crée une requête conjonctive à partir de chaque clause de la partie négative, après on cherche l'existence d'un homomorphisme de cette requête dans la base de faits qui respecte la substitution en cours sachant que si un tel homomorphisme existe alors la substitution en cours est filtrée (**Notez** que lorsque la partie calculée de la requête créée à partir de la clause négative n'est pas vide alors on cherche l'existence d'un homomorphisme qui satisfait cette partie calculée). Cet algorithme permet à la fin de trouver tous les F-homomorĥismes de la requête Q dans l'ensemble d'atomes F.

Algorithm 1 findsQueryFhomomorphisme : calculé Σ = L'ensemble des réponses de Q dans F

```
if (F = ∅) and (C = ∅) then
  Σ ← calculateHomomorphisme(Q, F)
else
  Q' ← createQuery(N)
  Σ' ← calculateHomomorphisme(Q', F)
  for all σ ∈ Σ' do
    filter ← true
    while filter and (F ≠ ∅) do
      filter ← evaluate(f ∈ F, σ)
      F ← F/f
    end while
    while filter and (C ≠ ∅) do
      Q'' ← createQuery(C ∈ C)
      F' ← getCalculateParts(Q'')
      if (F' = ∅) then
        filter ← existHomomorphisme(Q'', F, wrpt.σ)
      else
        Σ'' ← calculateHomomorphisme(Q'', F, wrpt.σ)
        for all σ' ∈ Σ'' do
          while filter and F' ≠ ∅ do
            filter ← evaluate(f ∈ F', σ ∪ σ')
            F' ← F'/f
          end while
        end if
      end while
    end for
    if filter then
      Σ ← Σ ∪ {σ}
    end if
  end for
end if
return Σ
```

Cet algorithme est dit simple parce-que il existe plusieurs optimisations possibles et la plus importante serait de filtrer le plus tôt possible c'est à dire dans l'algorithme de backtrack dès qu'un filtre est évaluable on l'évalue, lorsque les variables d'un filtre appartiennent à la substitution en cours (la substitution partielle trouvée à un moment i du backtrack), alors l'évaluation du filtre est possible parce-que toutes ses variables ont une image dans cette substitution et donc cela permettra de filtrer des substitutions même avant de les avoir complètement calculées.

5.2.3 Détails d'implémentation

La méthode *addAtomInRightList* de la classe *AbstractDlgpListener* suivante :

```
1  private void addAtomInRightList(Atom atom) {
2          if(this.NegAtomIndexJ==0) {
3              this.negConjunction.add(new LinkedListAtomSet());
4              this.negConjunction.get(NegAtomIndexI).add(atom);
5          }else {
6              if(this.NegAtomIndexJ!=-1)
7                  this.negConjunction.get(NegAtomIndexI).add(atom);
8              else {
9                  this.atomSet.add(atom);
10                 if(atom instanceof ComputedAtom)
11                     this.computedAtomSet.add(atom);
12                 else
13                     this.normaleAtomSet.add(atom);
14             }
15
16         }
17     }
```

permet lorsque un atome est créé de l'ajouter dans le bon attribut tel que : si sa position (i,j) a été modifiée c'est qu'il appartient à une clause négative, donc on le met dans l'attribut *negConjunction* dédié à cette partie et le fait de modifier sa position permet donc de le mettre dans la bonne clause (i) et au bon indice (j), sinon c'est que sa position est restée comme elle a été initialisée à savoir (-1,-1), ce qui fait que l'atome est positif (calculé ou standard), alors on le met dans l'attribut dédié aux deux parties normale et calculée, ensuite en fonction de la nature de l'atome on le met encore soit dans l'attribut de la partie calculée soit dans celui de la partie normale.

Lorsque on reconnaît une requête on crée l'ensemble de ses parties en les mettant dans leurs attributs dédiés, ensuite on crée la requête à partir de des ces attributs, après sa création, les méthodes d'accès suivante de la classe *DefaultConjunctiveQuery* seront utilisées dans l'algorithme de calcul du F-homomorphisme : une

méthode *getNormalAtomSet* qui permet de renvoyer la partie normale de la requête, une autre méthode *getComputedParts* qui permet de renvoyer la partie calculée de la requête une dernière méthode *getNegatedParts* qui permet de renvoyer la partie négative de la requête. Cet algorithme est implémenté principalement dans la méthode *filterSub* de la classe *SmartHomomorphism* :

```
1   private CloseableIterator<Substitution> filterSub(CloseableIterator<Substitution>
    resultIterator,AtomSet atomSet) throws HomomorphismException{
2
3       List<Substitution> resultSub=new ArrayList<Substitution>();
4       CloseableIterator<Substitution> negationSub;
5       InMemoryAtomSet computedAtomSet=this.computedAtomSet;
6       List<InMemoryAtomSet> negatedParts=this.negatedParts;
7       InMemoryAtomSet negComputedAtom;
8
9       if(this.computedAtomSet.isEmpty() && this.negatedParts.isEmpty())
10          return resultIterator;
11      try {
12          while(resultIterator.hasNext()) {
13              Substitution s=resultIterator.next();
14              boolean filter=true;
15              CloseableIterator<Atom> it=computedAtomSet.iterator();
16              while(filter && it.hasNext())
17                  filter=((ComputedAtom)it.next()).eval(s);
18              Iterator<InMemoryAtomSet> itNeg=negatedParts.iterator();
19              while (filter && itNeg.hasNext()) {
20                  ConjunctiveQuery q=new DefaultConjunctiveQuery(itNeg.next());
21                  negComputedAtom=q.getComputedAtomSet();
22                  if(negComputedAtom.isEmpty()) {
23                      filter=!(this.exist(q,atomSet,s));
24                  }
25                  else {
26                      negationSub=this.execute(q,atomSet,s);
27                      CloseableIterator<Atom> itAtom=negComputedAtom.iterator();
28                      while(filter && itAtom.hasNext()) {
29                          if(negationSub.hasNext())
30                              filter=!((ComputedAtom)itAtom.next()).eval(negationSub.next().
    compose(s));
31                          else {
32                              if(q.getNormalAtomSet().isEmpty())
33                                  filter=!((ComputedAtom)itAtom.next()).eval(s);
34                              else
35                                  itAtom.next();
36                          }
37                      }
```

```
38                      }
39                  }
40                  if(filter) {
41                      resultSub.add(s);
42                      }
43                  }
44          }catch(IteratorException e) {
45              throw new HomomorphismException(e);
46          }
47
48          return new CloseableIteratorAdapter<Substitution>(resultSub.iterator());
49          }
```

Lors de la recherche de réponses à une requête dans une base de faits la méthode *execute(query,AtomSet)* de la classe *SmartHomomorphism* sera invoquée, elle fait appel à la méthode *getNormalQuery* de la même classe en lui passant en argument la requête en question pour récupérer la partie normale et la partie négative de la requête et les stocker respectivement dans l'attribut *computedAtomSet* et *negatedParts* de cette même classe et récupérer aussi une requête conjonctive créée à partir de la partie normale de la requête. Une fois la requête réduite à la partie normale est créée, on fait appel à la méthode *filterSub* en lui passant comme paramètres l'ensemble des substitutions de cette requête sur l'ensemble d'atomes. À l'aide de ces deux attributs *computedAtomSet* ainsi que *negatedParts* qui ont été initialisés, cette dernière méthode invoquée permet en utilisant tous les ingrédients qu'ils lui ont été préparés, d'implémenter l'algorithme vu précédemment5.2.2.

5.3 Application d'une règle

5.3.1 Principe

Une règle (corps $\rightarrow$ tête) n'est plus vue comme un ensemble d'atomes impliquant un ensemble d'atomes mais une requête conjonctive impliquant un P-fait, donc une règle est de la forme $Q \rightarrow P$ où Q est une requête conjonctive et le P-fait P est un ensemble d'atomes (avec prédicat standard) dont les termes peuvent être des termes calculés.

Comme dans la version de base de Graal, il y a une ambiguïté au moment de l'analyse, en effet on ne peut pas savoir en avance si on est entrain de reconnaître un fait ou bien la tête d'une règle, cette ambiguïté est seulement levée lorsque on reconnaît la fin du fait('.') ou bien la fin de la tête d'une règle (' :-'), Ce qui fait qu'au moment de l'analyse tant que l'ambiguïté n'est pas levée les évènements créés lorsqu'on reconnaît des atomes sont stockés dans une liste qu'on appelle liste d'évènement en attente, et comme son nom l'indique les évènements de cette liste ne seront pas déclenchés jusqu'à ce qu'on sache ce qu'on est en train d'analyser. Dès

qu'on sait que c'est une règle qu'on analyse (c'est le cas qui nous intéresse ici), tous les évènements mis en attente seront déclenchés (**Notez** que comme c'est le cas pour les requêtes, avant le déclenchement de ces évènements qui concernent les atomes reconnus, un évènement *StartObjectEvent* est créé pour faire toutes les initialisations nécessaires), ensuite un évènement *conjunctionEndsEvent* sera créé et déclenché et son déclenchement permet d'invoquer la méthode *endsConjuncions* de la classe *AbstractDlgpListener* qui simplement mettra tous les atomes créés dans l'attribut consacré pour la tête d'une règle. Après cette étape, on commencera à reconnaître des atomes du corps de la règle, et la création du corps de la règle suit exactement le même raisonnement que pour la création de la requête, ce qui fait que la règle est composée de trois parties : une tête qui est un *AtomSet*, un *AtomSet* pour les atomes normaux et pour les atomes calculé du corps de la règle, et une liste d'*AtomSet* éventuellement vide pour la partie négative qui représente la partie négative du corps de la règle. Nous avons pensé à dire simplement que le corps d'une règle est directement une requête mais le problème c'est que ça implique de modifier le constructeur de la règle et comme la règle est aussi un élément centrale dans GRAAL alors on a utilisé le même mécanisme utilisé dans les requêtes pour traiter la partie négative d'une règle.(**Notez** que le mécanisme de séparation utilisé dans les requêtes qui consiste à séparer en un ensemble d'atomes calculé et un autre ensemble d'atomes normaux n'est pas utilisé lors de la création de règle (la cause est expliquée dans l'application de la règle)).

Maintenant qu'on a créé une règle de la forme $R = A_2 : -A_1, C$ où A_2 est ensemble d'atomes qui ne contient pas de prédicats calculés, A_1 est un ensemble d'atomes contenant et des prédicats standards et des prédicats calculés et respectant la contrainte qui dit que si une variable apparaît dans un atome calculé c'est qu'elle apparaît aux mois dans un des atomes normaux, cette contrainte est respectée en suivant le même mécanisme que pour les requêtes, et C est la partie négative du corps de la règle, **Notez** que si une variable apparaît dans une clause négative $c \in C$ ainsi que dans la tête de la règle et qu'elle n'apparaît pas dans la partie normale de A_1 alors cette variable sera traitée comme une variable existentielle dans la tête de la règle. Nous pouvons voir comment cette règle est appliquée, en effet une telle règle est applicable dans une base de faits si et seulement si on trouve un F-homomorphisme du corps de la règle dans la base de fait et son application donne un fait qui est l'image du p-fait tête de la règle. Pour cela un algorithme simple qui se base principalement sur l'algorithme de réponses à une requête qu'on a vu précédemment a été mis en place tel que :

Algorithm 2 : applicateRule : calculé S = l'ensemble des faits résultat de l'application d'une règle R dans une base de fait F

$Q \leftarrow createNegativeQuery(A_1, C)$
$\Sigma \leftarrow findsQueryFhomomorphisme(Q, F)$
for all $\sigma \in \Sigma$ **do**
  $S \leftarrow S \cup imageOf(A_2, \sigma)$
end for
return S

Premièrement, on crée une requête conjonctive à partir du corps de la règle, une séparation de A_1 en un

ensemble d'atomes calculé et un autre ensemble d'atomes normaux sera faite lors de la création de cette requête, et c'est pour cela que nous n'avons pas fait la séparation au moment de la création de le règle sinon on y serait obligé de mettre la partie calculée et normale dans un seul ensemble pour pouvoir créer cette requête et donc refaire la séparation apèrs sa création (il n'y a pas de constructeur de requête qui prend en argument les parties calculées et normales). Une fois que la requête qui représente le corps de la règle est créée on utilise l'algorithme de recherche d'homomorphisme vu précédemment pour trouver toutes les substitution de cette requête dans la base de fait, enfin pour chaque substitution on calcule l'image de la tête de règle tel que : si la tête de la règle contient un terme calculé alors l'image de ce terme sera le résultat de son évaluation dans le contexte de la substitution en cours, si la tête contient une variable existentielle celle-ci sera renommée par une variable fraîche.

Exemple 33 *Soient* $R = r(X, Z) : -p(X, Y), not\ q(Y, Z)$ *une règle, et* $F = \{p(1, 2)\}$*, alors la variable* Z *de la tête de la règle est considérée comme une variable existentielle et lors de l'application de la règle elle sera renommée. Une requête* Q *sera créée à partir de la tête de la règle, ensuite en utilisant l'algorithme vu précédemment on calcule l'ensemble* Σ *de tous les F-homomorphisme de cette requête dans* F*, ici* $\Sigma = \{\{X : 1, Y : 2\}\}$ *et comme* Σ *n'est pas vide alors la règle* R *est applicable dans* F *et son application donne le fait* $r(1, Z_0)$*.*

Si nous avons une base de connaissance $K = (\mathcal{R}, F)$ où F est un ensemble d'atomes et $\mathcal{R}$ un ensemble de règles, et qu'on a une requête Q. Lors de l'analyse syntaxique, une base de connaissance sera créée (classe *DefaultKnowledgeBase*) où tous les objets rencontrés seront mis, une fois que la base de connaissance est créée et que l'ensemble d'atomes et de règles créés est ajouté à cette base ainsi que la requête Q, répondre à la requête Q dans la base K se fait en saturant la base de fait F par l'ensemble de règles $\mathcal{R}$ et ça par une suite d'applications successives des règles de $\mathcal{R}$ jusqu'à ce qu'il n' y ait plus de règles applicables(le chase), ensuite on interroge la base résultante F^*. Il existe plusieurs chase(oblivious, semi-oblivious, restricted et le core-chase), Concernant l' oblivious et le semi-oblivious chase leurs implementation n'as pas beaucoup changé par rapport la version de Graal sans extension, mis à part le changement de l'application d'une règle comme on l'a vu précédemment. Pour le restricted chase, une règle est applicable si en plus de l'existence d'un F-homomorphisme du corps de règle dans F il n'existe pas de F-homomorphisme de la tête de la règle qui respecte cette substitution dans F, donc l'application d'une règle dans ce chase est vu de la manière suivant, une règle de la forme $B \rightarrow H$ est applicable si et seulement s'il existe un F-homomorphisme de $B, notH$ dans F ce qui fait qu'au moment de l'application d'une règle lorsque on crée une requête à partir du corps de la règle la partie négative de cette requête comportera aussi la tête de la règle, par exemple si $R = A_2 : -A_1, C$ est une règle, lorsque on voudra l'appliquer, la requête créée à partir du corps de cette règle sera $Q = A_2, (C \cup \{not\ A_2\})$. Ensuite il suffira de chercher les F-homomorphismes de cette requête dans F.

5.3.2 Le problème de la négation

La saturation avec des règles contenant un corps avec négation se fait sans problème mis à part le fait que lorsque les règles ne sont pas monotones le résultat de la saturation dépendra généralement de l'ordre d'application de règles, comme nous l'avons vu dans le chapitre Modèle formel, il existe une solution qui consiste à calculer une stratification de l'ensemble de règles lorsque celle ci existe ce qui permet d'avoir un ordre d'application pour les règles permettant d'avoir une base de fait saturée unique. Un module qui permet de faire la stratification a été implémenté dans le cadre d'un autre stage, l'étape suivante serait d'intégrer ce module dans la plateforme pour pouvoir stratifier l'ensemble de règle avant d'appliquer les chases.

5.3.3 Détails d'implémentation

Création La création de la règle se fait essentiellement dans cette partie de la fonction *endConjunction* de la classe *AbtractDlgpListener* :

```
1  case RULE:
2              if (this.atomSet2 == null) {
3                  this.atomSet2 = this.atomSet;
4                  this.atomSet = new LinkedListAtomSet();
5              } else {
6                  bodyVars = this.normaleAtomSet.getVariables();
7                  computedBodyVars = this.computedAtomSet.getVariables();
8                  for(Term t:computedBodyVars) {
9                      if(!bodyVars.contains(t))
10                         throw new ParseError("The variable ["+ t +"] of the computed atome
       does not appear in the normal part of the rule body.");
11                 }
12
13                 if(this.negConjunction.isEmpty())
14                     this.createRule(DefaultRuleFactory.instance().create(this.label, this.
       atomSet,this.atomSet2));
15                 else
16                     this.createRule(new DefaultRuleWithNegation(this.label, this.atomSet,this.
       atomSet2,this.negConjunction));
17             }
```

Dans cette classe un attribut *atomSet2* a été prévue pour la tête de la règle tel que, après avoir créer tout les atomes de la tête de règle (la création d'atomes a été détallée précédemment) ils seront ajoutés dans l'attribut *atomSet* comme vu précédemment, et comme le même évènement *endConjunctionEvent* se déclenche pour signaler et la fin de la tête et la fin du corps de la règle alors, un test a été mit en place pour savoir si

on a reconnu la tête ou le corps de la règle, qui consiste à tester si l'attribut *atomSet2* est vide ce qui fait qu'on n'a pas encore reconnu la tête donc tous les atomes qui ont été reconnus et crée seront mit dans cette attribut(*atomSet2*), sinon c'est que la tête a déjà été reconnu et donc on a reconnu le corps aussi. Maintenant le corps et la tête de la règle sont respectivement mit dans les attributs *atomSet et atomSet2* alors on passe à la création de la règle tel que : comme le cas des requêtes si la partie négative du corps ici(*negConjucntion*) est vide alors on crée une règle normale(*DefaultRule*) sinon on crée une règle avec négation (*DefaultRuleWithNegation*) suivant le même mécanisme que celui des requêtes.

Application d'une règle et les chases Pour l'application d'une règle dans l'oblivious et le semi-oblivious chase la seule chose qui a changé c'est qu'au moment de la création de requête à partir du corps de règle un test est mis en place pour savoir si il faut créer une requête conjonctive ou une requête conjonctive avec négation, et ça simplement en fonction du corps de la règle tel que si sa partie négative est vide alors la requête créée sera normale sinon ça serait une requête avec négation. Dans le cas du restricted chase la méthode *delegatedApply* suivante est importante tel que :

```
1 public  CloseableIterator<Atom> delegatedApply(Rule rule, T atomSet) throws
      RuleApplicationException {
2         try {
3             ConjunctiveQuery query;
4             rule.getNegatedParts().add(rule.getRewriteHead());
5             query=new DefaultConjunctiveQueryWithNegation(rule.getBody(),rule.getNegatedParts
      ());
6             CloseableIterator<Substitution> results=SmartHomomorphism.instance().execute(query
      , atomSet);
7             return new RuleApplierIterator(results, rule, atomSet);
8         }
```

Avant la création de la requête ici(*query*) à partir du corps de la règle la tête éventuellement réécrite sera rajoutée à la partie négative du corps de règle ce qui fait que la requête créée aura la négation de tête de règle dans sa partie négative ce qui permet d'appliquer le principe du restricted chase.

Réécriture de la tête d'une règle Avant la création de la requête ici(*query*) à partir du corps de la règle la tête éventuellement réécrite sera rajoutée à la partie négative du corps de règle ce qui fait que la requête créée aura la négation de tête de règle dans sa partie négative ce qui permet d'appliquer le principe du restricted chase. Ceci nous pose un problème, car notre syntaxe autorise des fonctions calculées dans les prédicats standards de la tête, mais pas dans ceux d'une requête, la réécriture de $B \rightarrow H$ en $B, not\, H$ nécessaire au restricted chase va donc créer une erreur. Il est donc nécessaire de réécrire la tête (sans changer sa sémantique) afin d'éviter

cette erreur.

Nous avons vu précédemment que dans le corps d'une requête il est interdit d'avoir des termes calculé dans la partie sa normale ce qui fait qu' au moment de la saturation de la base de faits en utilisant le restricted-chase, lorsque on crée une requête à partir de la règle à appliquer on rajoute la tête de cette règle dans la partie négative de cette requête créée, donc il faut s'assurer que tous les atomes (normaux) de la tête à ajouté ne contiennent pas de termes calculés, c'est pour ça que nous avons introduit un système de réécriture qui permet de réécrire tous les atomes de la tête de règle qui contiennent au moins un terme calculé.

La réécriture de la tête de la règle est faite à l'aide la méthode *getRewiteHead* de la classe *DefaultRule* suivante :

```
1  public InMemoryAtomSet getRewriteHead() {
2          InMemoryAtomSet rewritedHead=new LinkedListAtomSet();
3          CloseableIterator<Atom> it=this.head.iterator();
4          try {
5              while(it.hasNext()) {
6                  Atom atom=it.next();
7                  if(atom.getFunctionalTerm().isEmpty())
8                      rewritedHead.add(atom);
9                  else {
10                     int indexOfTerm=0;
11                     List<Term> list = new LinkedList<Term>();
12                     for(Term t:atom.getTerms()) {
13                         if(t.isFunctionalTerm()) {
14                             list.add(t);
15                             Term v= DefaultTermFactory.instance().createVariable(this.head.
       getFreshSymbolGenerator().getFreshSymbol());
16                             list.add(v);
17                             atom.setTerm(indexOfTerm,v);
18                             rewritedHead.add(atom);
19                             rewritedHead.add(new ComputedAtom(new Predicate("http://www.lirmm.
       fr/dlgp/#eq",2),list,((FunctionalTerm) t).getInvoker()));
20                             }
21                         indexOfTerm++;
22                         }
23                     list.clear();
24                     indexOfTerm=0;
25                     }
26                 }
```

Pour chaque atome de la tête de la règle on teste s'il ne contient pas de termes calculés pour ne pas le réécrire sinon on vas remplacer chaque terme calculé par une variable et on vas rajouter un prédicat d'égalité

entre cette variable et ce terme calculé (même système de réécriture pour les requêtes).

6. Évaluation

6.1 Tests

Comme nous l'avons précisé dans le chapitre Ma mission, la plateforme GRAAL subit une refonte générale de la version 1 vers la version 2, que ce soit dans la hiérarchie des classes, l'organisation en modules ou les algorithmes essentiels, c'est pour cela que nous n'avons pas mis en place des tests unitaires car l'extension réalisée dans ce stage est destinée à être intégré dans la version 2, par contre un ensemble considérable de tests ont été fait, en effet à chaque fois qu'une fonctionnalité est ajoutée plusieurs tests ont été effectués, leur code dlgp a été archivé et servira pour réaliser les tests unitaires dans les prochaines versions de Graal.

Exemples de tests Soient les deux fichier dlgp suivants qui regroupent quelques exemples de tests qui ont été effectués tel que :

```
@import </chemin/vers/fichier2.dlgp>
%@prefix ex:<exemple/test/>
p(2,3).
ex:toto(X).
```

fichier1.dlg

```
@import </chemin/vers/fichier1.dlgp>
@prefix ex:<exemple/test/>
@computed fct:<chemin/vers/config.json>

@facts
p(1,2),p(3,4),p(4,5).
ex:tata(X).
@rules
[R1] r(Z,X+1) :- p(X,Y),not p(Y,Z),not X=5
[R2] q(Y*2) :- r(X,Y), fct:pred(Y), not r(fct:f(Y),Z).
@queries
[q1] ?(X,Y) :- r(X,Y).
[q2] ?(X) :- q(X), X<Y.
```

fichier2.dlg

En passant le deuxième fichier au parser qui importe le premier fichier, nous allons avoir une erreur "a cycle was detected ..." parce-que le fichier1 l'importe à son tour, on commente la ligne 1 du premier fichier, on aura une erreur "prefix not declared ..." car le préfixe 'ex' n'a pas été déclaré dans le fichier 2, on décommente la ligne 2 de ce fichier l'erreur disparaîtra, ensuite on aura l'erreur "the variable [Y] was not declared ..." car la variable Y du corps de la requête $q1$ n'apparaît pas dans sa partie normale($q(X)$) on enlève cette requête pour enlever l'erreur, enfin il n'y aura plus d'erreurs.

6.2 Efficacité par rapport à l'existant

Nous avons rajouté des fonctionnalités à Graal, mais nous ne voulons pas que ces nouvelles fonctionnalités impactent négativement l'efficacité de Graal quand aucun de nos nouveaux objets n'est utilisé.

Ces tests sont effectués sur une machine qui a un processeur Intel® Core™ i5-4210U CPU @ 1.70GHz × 4 et une RAM 3.8 Gio @ 2400 MHz. Nous avons utilisé pour ces tests un jeu de données de 100 règles, 3935 faits ainsi que 20 requêtes, qu'on a passé plusieurs fois à GRAAL sans extension (la version à partir de laquelle j'ai travaillé) ensuite à GRAAL avec l'extension (la version que j'ai réalisée), nous avons compté le temps de parsing, de la création et le chargement des règles et des faits dans la base de connaissance ainsi que le temps de réponses aux requêtes et nous avons obtenu les résultat suivant(L'unité de mesure est les millisecondes (ms)) :

	Chargement de données	Réponses aux requêtes	Totale
GRAAL sans extension	1014	79444	80525
GRAAL avec l'extension	1127	98524	99705

Ce que nous avons remarqué c'est que le temps de parsing, création et chargement de données dans la base de connaissance a légèrement augmenté dans la version que j'ai réalisée et ça est dû au fait qu'il y a une phase d'initialisation, par exemple le chargement de toutes les instances de classes où se trouvent les méthodes java prédéfinies qui représentent les prédicats et termes spéciaux, le temps de réponse aux requêtes a augmenté de presque 20%, cette inefficacité est engendrée par la saturation de la base de faits, en effet dans la version de GRAAL sans extension le restricted-chase est utilisé pour la saturation, on rappelle que dans ce chase une règle $B \rightarrow H$ est applicable sur un ensemble d'atomes F s'il existe un homomorphisme de $B, notH$ dans F, le mécanisme d'application de règles qui prend en compte cette manière d'application existe déjà dans cette version, et il consiste à créer la requête $Q = B, notH$ et chercher tous les homomorphismes de cette requête dans F sachant que lors de la recherche c'est au moment du backtrack qu'il intervient pour filtrer les homomorphismes, tandis que, dans la version que j'ai réalisée où le restricted-chase est aussi utilisé pour la saturation, un autre

mécanisme est mis en place qui consiste un créer la même requête Q mais la recherche d'homomorphismes se fait en appliquant l'algorithme de recherche de F-homomorphismes vu précédemment 5.2.2 qui intervient à la fin du backtrack. Dès que l'optimisation de l'algorithme de recherche de F-homomorhisme (voir la section 7.2) sera implémentée ce problème d'inefficacité sera résolu.

Pour faire des tests sur les nouveaux objets ajoutés à la plateforme nous avons modifié l'ensemble des 20 requêtes en rajoutant des filtres et des négations(des nouveaux objets) par exemple :

Exemple 34 *Soit la troisième requête de l'ensemble des 20 requête suivante :*

```
[q03] ?(VAR_m4004_c1) :- m4004(VAR_m4004_c0, VAR_m4004_c1, VAR_m4004_c2, VAR_m4004_c3),
    m297004(VAR_m297004_c0, VAR_m4004_c1, VAR_m297004_c2, VAR_m297004_c3).
```

L'ensemble des réponses à cette requête était le suivant :
$\{VAR\_m4004\_c1 \rightarrow "X2"\}$
$\{VAR\_m4004\_c1 \rightarrow "X0"\}$
$\{VAR\_m4004\_c1 \rightarrow "X4"\}$
$\{VAR\_m4004\_c1 \rightarrow "X1"\}$
$\{VAR\_m4004\_c1 \rightarrow "X3"\}$
nous avons rajouté deux filtres(un atome calculé (<) qui contient un terme calculé(comp :plus(String,String) qui renvoie la somme des deux derniers chiffres des chaînes de caractères passées en paramètre par exemple plus$("X1","X3")$ *renvoi* $1+3=4$*) et une négation) à cette requête tel que :*

```
[q03] ?(VAR_m4004_c1) :- m4004(VAR_m4004_c0, VAR_m4004_c1, VAR_m4004_c2, VAR_m4004_c3),
    m297004(VAR_m297004_c0, VAR_m4004_c1, VAR_m297004_c2, VAR_m297004_c3),comp:plus(
    VAR_m4004_c1,"X5")<7 ,not VAR_m4004_c1="X1".
```

ce qui fait que le résultat maintenant est réduit uniquement à la substitution $\{VAR\_m4004\_c1 \rightarrow "X0"\}$.

Après avoir modifier l'ensemble de ces requête nous avons interrogé la base de connaissance sur cet nouvel ensemble de requêtes et nous avons obtenu les résultats suivant : **Notez** que cette fois ci nous avons utilisé uniquement la plateforme qui contient les extension par ce que les requêtes contiennent les nouveaux objets rajoutés.

	Chargement de données	Réponses aux requêtes	Totale
GRAAL avec l'extension	1250	103035	104250

On remarque que le temps de chargement de données a légèrement augmenté par rapport au requêtes sans nouveaux objets dans la plateforme GRAAL avec l'extension, et ça est dû au fait qu'on a rajouté des déclarations

pour pouvoir inclure de nouveaux prédicats et termes calculés, par exemple la fonction $plus(String, String)$, et ces déclaration sont traitées au moment de l'analyse syntaxique. Le temps de réponses aux requêtes a augmenté à cause du temps de calcul qu'est fait lors du filtrages (termes calculés, atomes calculés et négation). Notons que l'optimisation algorithmique dont nous avons déjà parlé pourrait également dans ce cas améliorer le résultat.

7. Conclusion et perspectives

7.1 Conclusion

Ce stage a été enrichissant pour moi, c'était également une expérience particulière car il m'a permis de découvrir un milieu professionnel riche et très vaste, ainsi qu'un secteur d'activité passionnant.

L'objectif de ce stage était de réaliser une extension d'une plateforme dédiée au raisonnement sur des bases de connaissances nommé GRAAL, pour ajouter la possibilité de faire des raisonnement numérique.

Les raisonnements numériques sont désormais possibles, on peut en tant qu'utilisateur implémenter une méthode java et l'utiliser dans nos programmes, implémenter une classe java et la considérer comme un type, mettre de la négation que ce soit dans la tête d'une règle ou dans le corps d'une requête, importer un autre fichier et pouvoir utiliser toutes les déclarations ainsi que tous les objets de ce fichier, ce qui permet de factoriser du code.

Le cahier des charges a donc bien été rempli malgré les nombreuses difficultés rencontrées. L'une parmi les principales difficultés est le fait que la plateforme est en évolution, donc j'étais obligé de modifier l'existant au minimum, la grammaire est implémentée en javaCC un langage que j'ai découvert durant ce stage, le fichier où elle est décrite était presque illisible, donc une première étape de réorganisation s'imposait.

Ce stage fut une occasion de mettre à l'épreuve mes capacités de gestion, il m'a permis d'évoluer dans plusieurs domaines principalement dans le plan humain. Cette expérience m'a permis d'accroître sensiblement mon sens du relationnel et me permet aujourd'hui de me sentir prêt sur le plan humain pour intégrer une entreprise en tant que professionnel dans des conditions optimales. J'ai pu notamment améliorer mon écoute, en effet les conseils fournis par mon responsable ont nécessité une grande attention de ma part pour travailler de manière optimale.

7.2 Perspectives

Il reste quelques petits problèmes au moment de l'analyse syntaxique tels que :

— Lorsqu'on analyse un terme calculé de la forme $X + 1$ ou $X - 1$, au lieu qu'il soit interprété comme une

addition ou une soustraction entre la variable X et l'entier 1 il est interprété comme la variable X suivi de l'entier $+1$ ou $-$ ce qui génère une erreur de parsing, pour le moment il suffit de mettre un espace entre le symbole $+$ ou $-$ et l'entier 1.

— Lorsqu'on analyse un terme calculé de la forme $X-1$, au lieu qu'il soit interprété comme une soustraction entre la variable X est l'entier 1, il est interprété comme la variable $X-$ suivi de l'entier 1, de la même manière, il suffit de mettre un espace entre X et $-$. Pour résoudre ce problème, interdire le symbole $-$ dans les noms de variables serait une solution envisageable.

— Le dernier problème existe lorsque on parse une terme calculé ou un prédicat calculé qui serait le dernier dans l'expression et qui est de la forme (X *pred ou fct* 1.), au lieu qu'il soit interprété comme un terme ou un prédicat calculé entre la variable X et l'entier 1 il est interprété comme une terme ou prédicat calculé entre la variable X et le décimale ou bien le double (1.) et dont il manque la suite. Pour le moment il suffit de mettre un espace entre l'entier 1 et le symbole de fin d'expression (.) .

Tous ces problèmes sont dû à une mauvaise gestion des priorités dans la grammaire et par manque de temps je n'ai pas pu revenir dessus pour les régler, pour le moment on a pu les détourner sans les résoudre en mettant l'espace là où il le fallait comme précisé précédemment.

Si la plateforme GRAAL n'était pas entrain de subir une refonte générale et qu'on pouvait modifier au plus profond dans son architecture, on pourrait définir une hiérarchie de classes plus sophistiquée pour représenter les nouveaux objets, par exemple au lieu de créer des requêtes qui contiennent une partie négative éventuellement vide on allait avoir un vrai type de requête avec négation ou bien au lieu d'avoir une règle avec un corps qui contient une partie négatives éventuellement vide on allait avoir un nouveau type de règle où leurs corps serait une requête directement ce qui faciliterait beaucoup leurs manipulation.

Exemple 35 *On allait avoir la hiérarchie suivante pour les règles où :* DefaultRule *est la classe qui représente la règle dans GRAAL et* NewDefaultRule *serait celle qui représenterait les règles si on pouvait changer la structure des classes.*

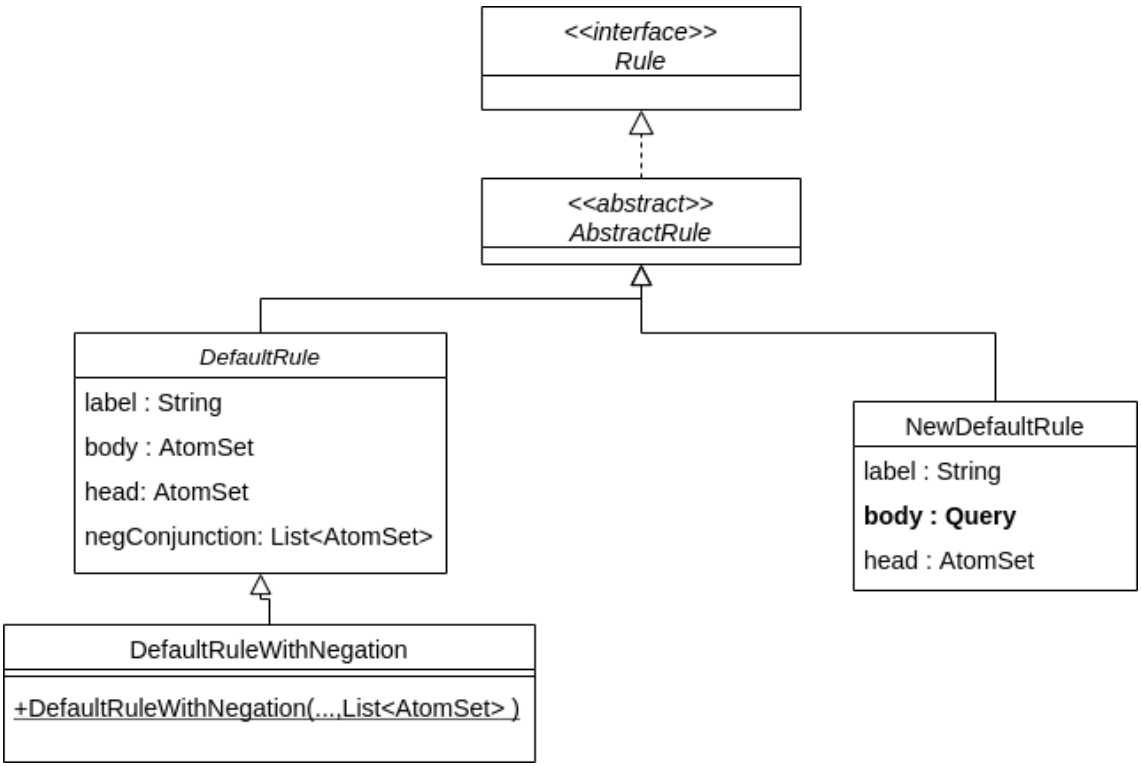


FIGURE 7.1 – Hiérarchie pour les règles

Un point très important serait de modifier les algorithmes simples définis pour répondre aux requêtes ainsi que pour faire le chaînage avant, pour faire des algorithmes optimisés où les filtres seront appliqués dès que leur application est possible(toutes leurs variables ont une image dans la substitution en cours 5.2.2). Concernant la stratification, c'était la dernière étape qui m'a été confiée dans ce stage, une tâche qui n'était pas dans le cahier de charge et que je n'ai malheureusement pas pu faire, qui permet de résoudre lorsque cela est possible le problème de la négation(5.3.2).

Un deuxième point serait de prendre en compte le typage des méthodes java qui représentent les prédicats et termes calculés, par exemple en rajoutant un champ dans le fichier de configuration dédié aux types des paramètres de ces méthodes et un champ supplémentaire pour les méthodes qui représentent les termes calculés dédié aux types de retours(le type de retour d'un prédicat est toujours booléen), ce qui fait qu'un prédicat ou un terme calculé aura l'ensemble des types de ses arguments et un terme calculé aura aussi un type de retour. Donc, si un prédicat ou un terme calculé a été reconnu (les types de ses paramètres sont connus et le type de retour pour les termes calculés aussi) et qu'on l'utilise dans une requête, règle...etc alors on pourra savoir au fur et à mesure de l'analyse s'il y a une erreur de typage.

Vu qu'un terme peut être un littéral spécial et qu'un littéral spécial possède un type alors on pourrait prendre en compte le typage des prédicats normaux aussi (voir le paragraphe 4.2.2) tel que la première fois

qu'on analyse un prédicat on prend en compte le type de ses arguments (termes), et donc comme pour les prédicats calculés il auront l'ensemble des types de leurs arguments, alors si un prédicat a été reconnu et qu'ensuite il est utilisé dans une requête, règle...etc alors on pourra savoir au fur et à mesure de l'analyse s'il y a une erreur de typage.

Ce stage de fin d'étude m'aura donc amené énormément de points positifs soit sur le plan technique, professionnel ou humain et je tiens donc à remercier une dernière fois l'équipe GraphIK ainsi que mon établissement l'université de Montpellier 2, sans eux rien de tout ça n'aurait été possible.

Bibliographie

[1] Jean-François Baget. A Datatype Extension for Simple Conceptual Graphs and Conceptual Graphs Rules. In Simon Povolina, Richard Hill, and Uta Priss, editors, *ICCS : International Conference on Conceptual Structures*, volume LNCS of *Conceptual Structures : Knowledge Architectures for Smart Applications*, pages 83–96, Sheffield, United Kingdom, July 2007. Springer.

[2] Jean-François Baget, Fabien Garreau, Marie-Laure Mugnier, and Swan Rocher. Revisiting Chase Termination for Existential Rules and their Extension to Nonmonotonic Negation. In Sébastien Konieczny and Hans Tompits, editors, *NMR : Non-Monotonic Reasoning*, volume INFSYS Research Report Series, Vienna, Austria, July 2014.

[3] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. Graal : A Toolkit for Query Answering with Existential Rules. In *RuleML : Web Rule Symposium*, volume LNCS of *Rule Technologies : Foundations, Tools, and Applications*, pages 328–344, Berlin, Germany, August 2015.

[4] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On Rules with Existential Variables : Walking the Decidability Line. *Artificial Intelligence*, 175(9-10) :1620–1654, March 2011.

[5] GraphIK team. Dlgp : An extended datalog syntax (version 2.1), 2021. `http://graphik-team.github.io/graal/doc/dlgp`.

[6] GraphIK team. Graal documentation, 2021. `http://graphik-team.github.io/graal/doc/index`.