# Extension d'ASP pour couvrir des fragments DL traitables : étude théorique et implémentation

Fabien Garreau

UNIVERSITE BRETAGNE LOIRE

université angers

# Thèse de Doctorat

Fabien GARREAU

*Mémoire présenté en vue de l'obtention du*
***grade de Docteur de l'Université d'Angers***
*sous le sceau de l'Université Bretagne Loire*

**École doctorale : Sciences et technologies de l'information, et mathématiques**

**Discipline : Informatique et applications, section CNU 27**
**Unité de recherche : Laboratoire d'étude et de recherche en informatique d'Angers (LERIA)**
**Laboratoire d'informatique, de robotique et de microélectronique de Montpellier (LIRMM)**

**Soutenue le 24 Novembre 2016**

# Extension d'ASP pour couvrir des fragments DL traitables : étude théorique et implémentation

## JURY

| | |
|---|---|
| Présidente : | **M^me^ Odile PAPINI**, Professeure, Université d'Aix Marseille |
| Rapporteurs : | **M. Andreas HERZIG**, Directeur de Recherche CNRS, Université Paul Sabatier, Toulouse |
| | **M. Philippe LAMARRE**, Professeur, Institut National des Sciences Appliquées de Lyon |
| Examinateur : | **M. Laurent GARCIA**, Maître de conférences, Université d'Angers |
| Invitées : | **M^me^ Marie-Laure MUGNIER**, Professeure, Université de Montpellier |
| | **M^me^ Claire LEFÈVRE**, Maître de conférences, Université d'Angers |
| Directeur de thèse : | **M. Igor STÉPHAN**, Maître de conférences (HDR), Université d'Angers |
| Co-directeur de thèse : | **M. Jean-François BAGET**, Chargé de Recherche, INRIA |

# Remerciements

Je tiens tout d'abord à remercier Andreas HERZIG, directeur de recherches CNRS à l'université Paul Sabatier de Toulouse, et Philippe LAMARRE, professeur à l'Institut National des Sciences Appliquées de Lyon, pour avoir accepté d'être rapporteurs de cette thèse.

La thèse est un travail dont la réussite dépend d'un bon encadrement et du soutien des proches, c'est pour cela que je tiens à remercier tout d'abord l'ensemble de mes encadrants de thèse en commençant par l'équipe d'Angers : Igor qui a contribué à ma thèse avec ses idées, ses conseils et sa logique, Laurent qui a su me soutenir, et que je dois remercier pour sa lecture, sa relecture et sa relecture... de ce manuscrit ainsi que pour les différentes tâches administratives, Claire qui m'a aidé grâce à son expertise en ASP, et les différentes aides apportées pour mes présentations et mon manuscrit. Je remercie aussi l'équipe Montpelliéraine : Jean-François pour son implication et sa créativité ainsi que Marie-Laure avec son esprit cartésien permettant un bon équilibre. Des encadrants avec des idées parfois différentes mais toujours la même envie de me faire avancer en me consacrant du temps. Une excellente équipe avec laquelle j'ai pu m'épanouir et évoluer durant ces années de thèse.

Je souhaite aussi remercier l'ensemble des membres du LERIA et particulièrement Vincent BARICHARD et David GENEST qui m'ont donné le goût de la recherche au cours de mes études supérieures, notamment pendant le projet ALLUA et mes différents stages. Merci à Christopher avec qui j'ai commencé ma thèse et avec qui je la termine pour le soutien mutuel que nous avons partagé. Je remercie aussi les autres doctorants avec qui j'ai passé de bons moments, Marc, Vincent Pierre, Arthur et Sara, l'équipe technique qui est toujours là lorsque l'on en a besoin, Catherine et Christine à qui on en demande beaucoup et qui font le maximum pour que nos déplacements se passent bien. À Montpellier je souhaite remercier l'ensemble de l'équipe GraphIk et plus particulièrement les anciens doctorants, Bruno et Michaël pour leur accueil ainsi que la découverte de la ville, les rencontres et les bons moments passés dans le sud. Je remercie aussi Annie pour la gestion de mon arrivée et de mes différents déplacements durant mon séjour.

Je tiens aussi à remercier l'Agence Nationale de la Recherche qui a financé cette thèse dans le cadre du projet ASPIQ[1] ainsi que l'ensemble des membres du projet grâce auxquels j'ai pu m'épanouir dans le monde de la recherche. Plus particulièrement, Odile pour son enthousiasme et sa motivation tout au long du projet et son soutien lors des différentes conférences, ainsi que Zied pour sa bonne humeur et les sorties dans les différents pays où l'on s'est retrouvé.

Merci enfin à mes amis et ma famille pour leur présence et leur soutien au cours de ma thèse. Xavier qui a participé à ALLUA et sur qui je peux compter. Mes parents à qui je dois mon éducation et mon envie d'apprendre et d'avancer. Pascal et Christelle pour leurs attentions et leur bienveillance au cours de ma thèse. Philippe et Michèle pour leur soutien en début de thèse et toutes les choses qu'ils ont pu m'apporter. Et bien sûr Anaïs avec qui j'avance et je construis et qui me soutient tous les jours avec sourire.

[1] Projet ASPIQ (ANR-12-BS02-0003) ASP technologIes for Querying large scale multisource heterogeneous web information http://aspiq.lsis.org/

# Introduction

Les ontologies sont utilisées pour la représentation et l'interrogation d'informations d'un domaine précis sous forme de réseau sémantique. Une ontologie regroupe un ensemble de concepts, permettant de représenter un domaine, liés entre eux par des relations taxonomiques et sémantiques. Ces liens entre les concepts permettent d'inférer de nouvelles connaissances qui ne sont pas stockées de manière explicite dans la base de connaissances. Une ontologie peut ensuite être interrogée à l'aide d'une requête pour extraire une information précise. Les données représentées peuvent être issues de plusieurs sources dont les connaissances sont plus ou moins précises, ainsi certaines données peuvent être incomplètes empêchant la déduction d'autres données, ou bien l'ensemble des données peut être incohérent, ce problème pouvant être éventuellement résolu par l'utilisation d'exceptions. Du point de vue de la représentation d'ontologies issues du web, c'est le langage OWL, basé sur RDF, qui est recommandé par le W3C, il a pour avantage d'être simple à interpréter pour une machine grâce à son format balisé, mais il est beaucoup plus difficile pour un humain d'élaborer une ontologie sous ce format à cause de sa syntaxe, et encore plus difficile de comprendre une ontologie en la lisant sous ce format. Dans cette thèse nous souhaitons raisonner sur ces ontologies en tolérant l'incohérence des données et les données incomplètes.

Les logiques de description sont un ensemble de langages visant à représenter et interroger ces ontologies. Les logiques de description sont composées de deux éléments, d'un côté un ensemble appelé $\mathcal{A}Box$ représentant les données spécifiques aux individus (par exemple : "Toto donne un cours d'informatique."), d'un autre côté un ensemble appelé $\mathcal{T}Box$ représentant les connaissances terminologiques, générales à tous les individus et liant les connaissances entre elles (par exemple : "si un individu X donne un cours Y alors X est un enseignant."). Il existe pour cela beaucoup de logiques de description plus ou moins précises, avec différentes extensions que l'on peut combiner pour obtenir un langage plus ou moins puissant en terme de représentation. Un des inconvénients est qu'en fonction de l'extension utilisée il est plus ou moins facile de représenter une ontologie avec ces langages pour un néophyte, or il existe un langage de programmation logique du premier ordre à base de règles, appelé règles existentielles [14], et qui généralise une partie de ces logiques de description dites logiques de description légères (DL-Lite [27]) et EL [10]. Les règles existentielles ont pour avantage de faciliter la représentation des informations tout en conservant toute la puissance des logiques de description légères. Ainsi la $\mathcal{A}Box$ est représentée par un ensemble de faits et la $\mathcal{T}Box$ par un ensemble de règles logiques. Ces règles existentielles permettent la déduction de l'existence d'individus inconnus, une caractéristique cruciale pour la représentation de connaissances dans un monde ouvert. Elles permettent l'interrogation mais elles ne permettent pas de représenter les exceptions. Elles se démarquent aussi par un grand nombre de travaux traitant de la décidabilité [15] d'un programme (est-ce qu'à partir d'une base de faits quelconque un programme s'arrêtera en un temps fini ou non).

D'un autre côté, le formalisme de l'Answer Set Programming (ASP [43]) est très utilisé en représentation de connaissances et en intelligence artificielle, spécialement lorsqu'il est question de représenter de l'information incomplète et pour coder des problèmes combinatoires [19, 62]. ASP est un langage de programmation logique non-monotone à base de règles dont la spécificité est de pouvoir représenter les exceptions. Une différence par rapport aux règles existentielles est qu'ASP est un langage non-monotone, de plus il peut y avoir aucun, un ou plusieurs modèles à un programme,

appelés answer set. Un avantage est qu'ASP dispose de solveurs efficaces permettant de calculer ces answer set. Malheureusement, il n'est pas possible de représenter les logiques de description légères avec ASP à cause de l'absence de variables existentielles dans les règles.

Il existe beaucoup de travaux traitant des logiques de description ou d'ASP mais il est intéressant de les réunir dans un seul formalisme pour pouvoir représenter des ontologies avec exceptions. Nous notons que l'étude de la combinaison de règles non-monotones avec des ontologies n'est pas nouvelle [64, 35, 31, 60, 39, 51, 58]. La base de connaissances est dans la plupart des cas une combinaison de deux ensembles ($\mathcal{K}$, $\mathcal{P}$) : $\mathcal{K}$ une base de connaissances décrivant les informations de l'ontologie à l'aide d'un fragment de la logique du premier ordre, par exemple avec une logique de description, et $\mathcal{P}$ décrivant les règles sous forme de programme logique.
Il existe ainsi trois classes d'intégration des deux formalismes [35, 51]. La première classe propose de gérer séparément les deux formalismes, comme proposé dans [35]. Ainsi la base de connaissances $\mathcal{K}$ est considérée comme une source de données externe qui est ensuite utilisée par le programme logique $\mathcal{P}$ avec des prédicats spéciaux permettant l'interrogation de la logique de description. Nous avons ici deux bases séparées avec deux sémantiques différentes dont le lien est fait à l'aide des prédicats spéciaux.
La seconde classe représentée dans [64, 60] correspond à un formalisme hybride qui intègre dans un même framework les logiques de description et les règles. Ainsi les prédicats provenant de $\mathcal{K}$ peuvent être utilisés avec les règles du programme $\mathcal{P}$ mais avec certaines restrictions comme l'impossibilité d'utiliser ces prédicats dans le corps négatif d'une règle.
La dernière classe intègre les logiques de description et les règles dans un seul formalisme. Par exemple, [31] utilise la quantified equilibrium logic (QEL) où il est proposé plusieurs bases de connaissances hybrides (avec *safe restriction*, *safe restriction without unique name assumption* ou avec *guarded restriction*) dont les modèles et l'expression en termes de QEL est prouvée pour chaque catégorie.
La majeure partie de ces travaux traitent de la question de la décidabilité et de la complexité. Dans ces travaux la gestion des variables existentielles est simplement autorisée dans l'ensemble $\mathcal{K}$, les têtes de règles de l'ensemble $\mathcal{P}$ restant universellement quantifiées.
Parallèlement à ces travaux [44] propose de couvrir à la fois la sémantique des modèles stables et celle de la logique du premier ordre à l'aide d'une formule logique du second ordre. Le liens entre les travaux précédents et celui-ci sont présentés dans [51].
Plus récemment d'autres approches ont été présentées comme l'extension des règles existentielles avec l'ajout de non-monotonie mais considérant une négation par défaut stratifiée [24], une approche avec la *well-founded semantic* [45] ou encore la sémantique des modèles stables [58]. Ce dernier article présente une approche où les variables existentielles sont skolémisées et s'intéresse au cas où il existe un unique modèle stable fini, de plus seule la base de connaissance autorise d'avoir de la négation par défaut et des variables existentielles dans la même règle. Cette étude traite de manière théorique et pratique le problème mais en n'utilisant qu'une partie limitée d'ASP où les programmes sont forcément stratifiés.
Pour terminer, seuls les travaux de [49] basés sur [35] et de [58] proposent une implémentation appliquée sur des informations portant sur la bio-chimie.

L'ensemble de cette thèse s'inscrit dans le cadre du projet ANR ASPIQ [2] : ASP TechnologIes for Querying large scale multisource heterogeneous web information (Techniques ASP pour l'interrogation d'informations web hétérogènes multi-sources à grande échelle). Le projet met en jeu quatre partenaires : CRIL (Artois), LERIA (Angers), LIRMM (Montpellier) et LSIS (Marseille).
L'idée globale du projet est d'utiliser l'efficacité des solveurs ASP afin de pouvoir raisonner sur des informations de grande taille issues de différentes sources (par exemple, sur le web). On souhaite pour cela étendre ASP afin d'exprimer ces informations, calculer des modèles et les interroger. La difficulté

du travail est de trouver le bon compromis entre l'expressivité du modèle et sa capacité à être traité de manière calculatoire. La thèse consiste en une partie du travail du projet en co-direction entre le LERIA et le LIRMM dont l'extension d'ASP, l'interrogation et l'implémentation d'un solveur traitant cette extension.

## Programme non-monotone existentiel

Dans cette thèse, nous proposons des programmes à base de règles non-monotones existentielles permettant de traiter des ontologies du type DL-Lite avec l'ajout d'exceptions que nous pouvons approcher de deux manières : soit en étendant ASP en tenant compte des variables existentielles ; soit en introduisant de la non-monotonie dans les règles existentielles. ASP est intéressant par son efficacité à traiter les problèmes de représentation et de raisonnement à base de connaissances et aussi par l'efficacité de ses solveurs. Les règles existentielles apportent un moyen efficace pour représenter et interroger les ontologies.

Pour définir cette extension nous avons tout d'abord établi un état de l'art à la fois d'ASP et des règles existentielles. Une première partie du travail fut de comparer les deux langages afin d'identifier les similarités et différences pour évaluer la faisabilité d'une extension. Les règles existentielles et l'Answer Set Programming (ASP) sont deux langages utilisés pour représenter des connaissances sous formes de règles, le premier est issu des graphes conceptuels et le second de la programmation logique. Ces deux langages se ressemblent syntaxiquement mais possèdent chacun une caractéristique spécifique, les règles existentielles permettent à l'aide de variables existentielles la représentation de modèles infinis comportant des individus n'apparaissant pas dans la base de faits initiale, tandis que l'ASP permet à l'aide de la négation par défaut la représentation d'exceptions et la possibilité d'avoir plusieurs modèles pour un programme. Cette comparaison a débouché sur la nécessité de déterminer la décidabilité d'un langage comportant à la fois des variables existentielles et de la négation par défaut. Ceci nous a permis d'homogénéiser les différentes propriétés d'acyclicité permettant de déterminer la terminaison d'un algorithme en marche avant pour une classe de programmes, ainsi que d'améliorer celles-ci et de déterminer les classes pour lesquelles l'extension serait décidable. Cette étude de décidabilité nous a aussi amené à comparer les différents algorithmes de marche avant, connu sous le nom de chase [38, 33, 59], et ainsi d'identifier un chase correspondant à notre extension.

À partir de cette étude nous proposons une extension du langage ASP, appelée programmes non-monotones existentiels, permettant d'interpréter les benchmarks des deux formalismes et de fournir des modèles adéquats avec l'utilisation d'un chase particulier. Cette extension a pour particularité de permettre la réécriture de ces programmes en ASP classique en utilisant la skolémisation pour traiter les variables existentielles apparaissant en tête de règle.

## Interrogation

Les ontologies, en plus de permettre la représentation de connaissances, donnent la possibilité d'interroger des connaissances afin d'extraire des informations précises. Les règles existentielles offrent cette possibilité [61]. Une deuxième partie du travail s'intéresse donc à l'interrogation de programmes non-monotones existentiels avec des requêtes dans un programme non-monotone existentiel. La recherche de réponse à une requête soulève un certain nombre de questions qu'il faut traiter. Qu'est-ce qu'une requête ? Qu'est-ce qu'une réponse à une requête ? Existe-t-il aucune, une ou plusieurs réponses ? Ces questions amènent à l'utilisation des requêtes conjonctives et requêtes booléennes conjonctives, et à la définition de deux types de réponses, la réponse sceptique (la réponse est présente pour tous les answer set) et la réponse crédule (la réponse est présente dans au moins un answer set).

Un des objectifs de l'interrogation d'un programme vise à réduire au maximum le coût algorithmique nécessaire pour obtenir une réponse à l'interrogation en n'utilisant que les données nécessaires pour la requête. L'interrogation faite sur les règles existentielles est basée sur la réécriture de la requête en utilisant une marche arrière. Le problème avec notre extension vient du caractère non-monotone qui donne la possibilité d'obtenir des programmes inconsistants, et ne permet pas l'utilisation d'une marche arrière de la même manière qu'en règles existentielles. L'inconsistance peut provenir d'une partie du programme n'ayant aucun lien avec la requête et n'étant pas détectée par la marche arrière. Peu de travaux traitent de l'interrogation en ASP, nous avons entrepris de reprendre les résultats provenant du traitement de l'interrogation à l'aide de magic sets [18] pour Datalog¬ [36, 6] (correspondant à ASP) et d'étendre ces résultats aux programmes non-monotones existentiels en travaillant notamment sur l'inconsistance des programmes.

## Implémentation

Après avoir étudié les aspects théoriques de l'extension d'ASP et des règles existentielles, il faut s'intéresser à sa mise en œuvre pratique. Pour cela, nous avons choisi de modifier un solveur ASP déjà existant afin d'ajouter de nouvelles fonctionnalités permettant le traitement du nouveau langage. Nous avons choisi de travailler sur ASPeRiX[54], un solveur ASP basé sur les règles, développé au LERIA. Ce choix se justifie par le fait que ce solveur a la particularité d'instancier les variables à la volée tandis que les autres solveurs passent par une phase d'instanciation. Ceci est intéressant du point de vue de l'extension car l'instanciation des variables existentielles peut poser des problèmes (instanciation infinie par exemple). Une première extension du solveur ASPeRiX est alors implémentée, elle transforme d'abord le programme étendu en programme ASP classique puis utilise l'algorithme déjà existant pour calculer les modèles du programme. Un module d'interrogation est aussi ajouté permettant l'utilisation de requêtes avec deux types de réponses, sceptique et crédule. Nous comparons ensuite techniquement le nouveau solveur avec les autres solveurs ASP, pour tester celui-ci sur des applications concrètes. Un des cadres prévus pour tester le solveur est une base de données académique `university` sur les universités, Lehigh University BenchMark (LUBM) [1].

## Organisation

Le plan de cette thèse est divisé en cinq chapitres. Nous commençons dans le chapitre 1 par les bases formelles utilisées dans cette thèse, constitué de quatre parties, la première présentant les bases logiques, les deuxièmes et troisièmes présentant respectivement l'Answer Set Programming et les règles existentielles et une dernière traitant des graphes de dépendance. Ce chapitre permet de poser les bases sur lesquelles la suite de cette thèse est fondée. Le chapitre 2 présente l'extension des programmes non monotones existentiels. Nous définissons dans une première partie la syntaxe et la sémantique, nous proposons ensuite une traduction de ces programmes vers des programmes ASP classiques, et pour finir nous discutons l'utilisation des chase des règles existentielles avec l'opérateur de conséquence d'ASP. Dans le chapitre 3 nous abordons le problème de la terminaison d'un algorithme en marche avant permettant le calcul de modèles pour un programme non-monotone existentiel. Nous présentons dans une première partie le principe de l'étude de la décidabilité, et les différentes classes de décidabilité regroupant des ensembles de programmes dont nous pouvons déterminer si le calcul de modèle ou l'interrogation se termine en un temps fini. Nous proposons dans une seconde partie une homogénéisation et une extension des notions d'acyclicité dans le cas de la classe d'ensemble à expansion finie, cette extension permet de considérer la négation par défaut pour améliorer la détection de programmes non-monotones existentiels décidables. Nous terminons le chapitre sur la décidabilité sur une discussion à propos de l'étude d'autres classes de décidabilité pour les programmes non-monotones existentiels. Dans le chapitre 4 nous abordons le thème de l'in-

terrogation, en présentant dans un premier temps l'interrogation dans un cadre général, puis en règles existentielles et enfin pour les programmes ASP. Nous continuons ce chapitre avec l'étude de l'inconsistance d'un programme ayant un rôle important dans l'interrogation d'un programme ASP, nous proposons ensuite d'isoler la plus petite partie d'un programme permettant de répondre à une requête précise. Nous proposons pour terminer quelques perspectives d'optimisations pour l'interrogation utilisant notamment les caractéristiques de la négation par défaut pour limiter le nombre d'instances nécessaires pour obtenir une réponse à une requête sur un programme. Enfin le chapitre 5 et dernier de cette thèse met en avant une implémentation de l'extension d'un solveur ASP pour permettre le traitement de programmes non-monotones existentiels. Ce chapitre est divisé en trois parties, la première présentant le solveur ASPeRiX et son extension permettant de traiter les programmes non-monotones existentiels et l'interrogation de ceux-ci. Dans la deuxième partie nous proposons une comparaison des différents solveurs ASP avec l'extension d'ASPeRiX sur des jeux de tests techniques portant sur des points précis de l'extension du solveur. Dans une troisième partie nous comparons les différents solveurs sur une ontologie académique qu'est `university` (Lehigh University BenchMark), dans un premier temps sur sa version originale puis sur différentes versions en ajoutant notamment de la négation par défaut. Cette thèse se terminera par une conclusion résumant l'ensemble des éléments évoqués ainsi que les perspectives de recherche.

# 1

# Bases formelles

## 1.1 Bases logique

Nous présentons dans cette première section l'ensemble des éléments permettant de définir les bases de la logique du premier ordre dont les langages ASP et règles existentielles sont issus ainsi qu'une présentation générale des logiques de description que nous souhaitons représenter dans cette thèse. Nous faisons ensuite un état de l'art d'ASP puis des règles existentielles, les deux langages principaux utilisés dans cette thèse. Enfin une courte partie sur les bases concernant les graphes de dépendance, ayant un rôle important pour les chapitres sur la décidabilité et l'interrogation.
Pour une grande partie des exemples nous utiliserons le benchmark `university` [1] qui a été utilisé et modifié au cours de cette thèse afin de correspondre au travail réalisé. Ce benchmark provient des logiques de description puis a été adapté pour les langages abordés dans cette thèse. Les termes employés en anglais sont les termes dont la traduction française n'est pas usitée ou encore est inexistante (par exemple les termes ground, safe, answer set,...).

### 1.1.1 Bases de la logique du premier ordre

**Définition 1.1.1** (Vocabulaire)**.** *Le* vocabulaire $\Phi$ *est défini de la manière suivante :*

- $\mathcal{CS}$ *l'ensemble des* symboles de constante,
- $\mathcal{PS}$ *l'ensemble des* symboles de prédicat,
- $\mathcal{FS}$ *l'ensemble des* symboles de fonction.

*Les ensembles $\mathcal{CS}$, $\mathcal{PS}$ et $\mathcal{FS}$ sont disjoints deux à deux et $\mathcal{CS}$ est non vide.*

**Définition 1.1.2** (Langage)**.** *Le* langage $\mathcal{L}$ *est défini sur $(\Phi,\mathcal{V})$ avec $\mathcal{V}$ l'ensemble des variables disjoint des ensembles composant le vocabulaire. Nous considérons la fonction $ar$ qui renvoie l'*arité *d'un élément de $\mathcal{FS}$ (resp. $\mathcal{PS}$) dans $\mathbb{N}^*$ (resp. $\mathbb{N}$) et $\mathcal{T}$ le plus petit ensemble des* termes *tel que :*

- *si* $\mathtt{c} \in \mathcal{CS}$ *alors* $\mathtt{c} \in \mathcal{T}$,
- *si* $\mathtt{v} \in \mathcal{V}$ *alors* $\mathtt{v} \in \mathcal{T}$,
- *si* $\mathtt{f} \in \mathcal{FS}$ *d'arité* $ar(\mathtt{f}) = n > 0$ *avec* $\mathtt{t_1}, \ldots, \mathtt{t_n} \in \mathcal{T}$ *alors* $\mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \in \mathcal{T}$.

*Un* terme ground *est un terme construit à partir de $\mathcal{CS}$ et $\mathcal{FS}$. L'ensemble des termes ground est noté* $\mathbf{GT}$ *et est appelé* univers de Herbrand.
*L'ensemble des* atomes $\mathcal{A}$ *est défini tel que*

- *si* $\mathtt{a} \in \mathcal{PS}$ *avec* $ar(\mathtt{a}) = 0$ *alors* $\mathtt{a} \in \mathcal{A}$,
- *si* $\mathtt{p} \in \mathcal{PS}$ *avec* $ar(\mathtt{p}) = n > 0$ *et* $\mathtt{t_1}, \ldots, \mathtt{t_n} \in \mathcal{T}$ *alors* $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \in \mathcal{A}$.

*Un atome* $\mathtt{a} = \mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ *est ground si* $\mathtt{t_i} \in \mathbf{GT}, \forall \mathtt{i}\ 1 \leq \mathtt{i} \leq n$. *La* base de Herbrand, *notée* $\boldsymbol{A}$, *est l'ensemble de tous les* atomes ground.

Par la suite les symboles de constante (resp. symboles de prédicat) sont simplifiés en *constantes* (resp. *prédicats*) lorsqu'il ne peut pas y avoir de confusion. Nous utilisons la notation suivante pour définir les différents éléments d'un atome a :

- $const(\mathtt{a})$ l'ensemble des constantes de a,
- $var(\mathtt{a})$ l'ensemble des variables de a,

Nous étendons cette notation aux ensembles d'atomes, ainsi l'ensemble des constantes (resp. variables) d'un ensemble d'atomes A est noté $const(A)$ (resp. $var(A)$).

**Définition 1.1.3** (Interprétation)**.** *Soit le langage $\mathcal{L}$ = ($\Phi$,$\mathcal{V}$) avec $\Phi$ un vocabulaire et $\mathcal{V}$ un ensemble de variables. Une* interprétation *de $\Phi$ est une paire $I = (\Delta, .^I)$ où $\Delta$ est un ensemble non vide appelé* domaine d'interprétation *et $.^I$ une* fonction d'interprétation *telle que :*

- $\forall \mathtt{c} \in \mathcal{CS}$, $\mathtt{c}^I \in \Delta$,
- $\forall \mathtt{p} \in \mathcal{PS}$ *d'arité* $k, \mathtt{p}^I \subseteq \Delta^k$,
- $\forall \mathtt{f} \in \mathcal{FS}$ *d'arité* $k, \mathtt{f}^I : \Delta^k \to \Delta$.

*Soit $A$ un ensemble d'atomes et $\sigma$ une fonction de $var(A)$ vers $\Delta$. Pour chaque terme* t *apparaissant dans $A$, nous définissons $t^I_\sigma$ par :*

- *si* $\mathtt{t} \in \mathcal{V}$, *alors* $\mathtt{t}^I_\sigma = \sigma(\mathtt{t})$ ;
- *si* $\mathtt{t} \in \mathcal{CS}$, *alors* $\mathtt{t}^I_\sigma = \mathtt{t}^I$ ;
- *sinon* $\mathtt{t} = \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_k})$ *avec* $\mathtt{f} \in \mathcal{FS}$ *d'arité* $k$, *et* $\mathtt{t}^I_\sigma = \mathtt{f}^I((\mathtt{t_1})^I_\sigma, \ldots, (\mathtt{t_k})^I_\sigma)$.

Une constante s'interprète comme un élément de $\Delta$ et un prédicat comme une relation sur $\Delta$ ayant pour arité celle du prédicat. Un symbole de fonction est interprété par une fonction d'arité $k$ avec $k$ l'arité du symbole de fonction. Soit enseignantChercheur un symbole de prédicat d'arité 1 alors $\mathtt{enseignantChercheur}^I$ est un sous ensemble de $\Delta$, soit donneCours un prédicat d'arité 2 alors $\mathtt{donneCours}^I$ est un sous ensemble de $\Delta^2$ (l'ensemble des couples possibles construits à partir de $\Delta$).

**Définition 1.1.4** (Modèle)**.** *Soit $A$ un ensemble d'atomes défini sur un langage $\mathcal{L}$ = ($\Phi$,$\mathcal{V}$) avec $\mathcal{V}$ l'ensemble des variables. Une interprétation $I = (\Delta, .^I)$ de $\Phi$ est un* modèle *de $A$ s'il existe une fonction $\sigma$ de $var(A)$ vers $\Delta$ telle que, pour chaque atome $p(t_1, \ldots, t_k) \in A$, $((t_1)^I_s, \ldots, (t_k)^I_s) \in p^I$, que nous notons $(\Delta, .^I) \vdash A$.*

**Exemple 1** (Modèle)**.** *Soit $\Phi_1$ un vocabulaire avec*

$$\mathcal{CS} = \{\mathtt{toto}, \mathtt{informatique}\}, \mathcal{PS} = \{\mathtt{enseignantChercheur}, \mathtt{donneCours}\}, \mathcal{FS} = \{\mathtt{parent}\}$$

*avec* $ar(\texttt{enseignantChercheur}) = ar(\texttt{parent}) = 1$ *et* $ar(\texttt{donneCours}) = 2$.
*Nous avons*

$$A = \{\texttt{enseignantChercheur}(\texttt{toto}), \texttt{donneCours}(\texttt{toto}, \texttt{informatique})\}$$

*un ensemble d'atomes, et deux interprétations* $I = (\Delta, .^I)$ *et* $I' = (\Delta, .^{I'})$ *avec*

$$\begin{array}{ll} \Delta = & \{toto, informatique, titi\} \\ .^I = & \{\texttt{toto}^I = toto, \\ & \texttt{informatique}^I = informatique, \\ & \texttt{parent}^I = \{toto \mapsto titi\}, \\ & \texttt{enseignantChercheur}^I = \{toto, informatique, titi\}\} \\ .^{I'} = & \{\texttt{toto}^{I'} = toto, \\ & \texttt{informatique}^{I'} = informatique, \\ & \texttt{parent}^{I'} = \{toto \mapsto titi\}, \\ & \texttt{enseignantChercheur}^{I'} = \{toto, titi\}, \\ & \texttt{donneCours}^{I'} = \{(toto, informatique)\}\} \end{array}$$

*Si nous nous plaçons dans l'univers de Herbrand nous pouvons représenter ces interprétations par les ensembles d'atomes ground suivants :*

$$\begin{array}{ll} I = & \{\texttt{enseignantChercheur}(\texttt{toto}), \texttt{enseignantChercheur}(\texttt{informatique}), \\ & \texttt{enseignantChercheur}(\texttt{parent}(\texttt{toto}))\} \\ I' = & \{\texttt{enseignantChercheur}(\texttt{toto}), \texttt{enseignantChercheur}(\texttt{parent}(\texttt{toto})), \\ & \texttt{donneCours}(\texttt{toto}, \texttt{informatique})\} \end{array}$$

*Nous avons ainsi* $I$ *n'est pas modèle de* $A$ *car* $I$ *ne contient pas* $\texttt{donneCours}(\texttt{toto}, \texttt{informatique})$. *Par contre,* $I'$ *est modèle de* $A$ *car* $I'$ *contient tous les atomes de* $A$.

**Définition 1.1.5** (Substitution)**.** *Soit* $\mathcal{X} \subseteq \mathcal{V}$ *un ensemble de variables* $\mathcal{X} = \{\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}\}$, *et* $\mathcal{T}$ *un ensemble de termes. Une* fonction de substitution $s$ *est une fonction des variables* $\mathcal{X}$ *dans les termes* $\mathcal{T}$ *notée* $[\texttt{X}_1 \leftarrow \texttt{t}_1, \ \ldots, \ \texttt{X}_\texttt{n} \leftarrow \texttt{t}_\texttt{n}]$ *(*$\texttt{X}_\texttt{i}$ *toutes distinctes entre-elles et* $\texttt{t}_\texttt{i} \neq \texttt{X}_\texttt{i}$ *pour tout* $1 \leq \texttt{i} \leq n$ *). Si* $\texttt{t} \in \mathcal{T}$, *la* substitution *de* t, *notée* $\sigma(\texttt{t})$, *est définie de la manière suivante :*

- *si* $\texttt{t} \in \mathcal{X}$, *alors* $\sigma(\texttt{t}) = s(\texttt{t})$,
- *si* $\texttt{t} \in \mathcal{V} \backslash \mathcal{X}$, *alors* $\sigma(\texttt{t}) = \texttt{t}$,
- *si* $\texttt{t} \in \mathcal{CS}$, *alors* $\sigma(\texttt{t}) = \texttt{t}$,
- *sinon,* $\texttt{t} = \texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{k})$ *avec* $\texttt{f} \in \mathcal{FS}$ *d'arité* $k$, $\texttt{t}_1, \ldots, \texttt{t}_\texttt{k}$ *des termes, et* $\sigma(\texttt{t}) = \texttt{f}(\sigma(\texttt{t}_1), \ldots, \sigma(\texttt{t}_\texttt{k}))$.

*La substitution vide, (l'*identité*), est notée* $\epsilon$. *Par extension nous avons, si* $\texttt{a} = \texttt{p}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{k})$ *un atome, l'application d'une substitution* $\sigma$ *de* a *est notée* $\sigma(\texttt{a}) = \texttt{p}(\sigma(\texttt{t}_1), \ldots, \sigma(\texttt{t}_\texttt{k}))$, *et si* $A = \{\texttt{a}_1, \ldots, \texttt{a}_\texttt{n}\}$ *est un ensemble d'atomes, l'application d'une substitution de* $A$ *est notée* $\sigma(A) = \{\sigma(\texttt{a}_1), \ldots, \sigma(\texttt{a}_\texttt{n})\}$.
*Une* substitution $\sigma$ *est dite* ground *si l'ensemble des variables* $\mathcal{X}$ *est associé à des termes ground. Soit* t *un terme (resp.* a *un atome) et* $\sigma$ *une substitution ground,* $\sigma(\texttt{t})$ *(resp.* $\sigma(\texttt{a})$*) est une* instance ground *de* t *(resp.* a*).*

**Exemple 2** (Substitution)**.** *Soit le langage* $\mathcal{L}_1$ *avec l'ensemble d'atomes*

$$A = \{\texttt{enseignantChercheur}(\texttt{toto}), \texttt{donneCours}(\texttt{toto}, \texttt{X})\}$$

*Il existe plusieurs substitutions possibles dont*

$$\sigma_1 = [\texttt{X} \leftarrow \texttt{informatique}]$$
$$\sigma_2 = [\texttt{X} \leftarrow \texttt{f}(\texttt{informatique})]$$

*pour instancier $A$ de la manière suivante*

$$\sigma_1(A) = \{\texttt{enseignant}(\texttt{toto}), \texttt{donneCours}(\texttt{toto}, \texttt{informatique})\}$$
$$\sigma_2(A) = \{\texttt{enseignant}(\texttt{toto}), \texttt{donneCours}(\texttt{toto}, \texttt{f}(\texttt{informatique}))\}$$

Nous présentons maintenant la définition des logiques de description, un formalisme permettant la représentation d'ontologies.

### 1.1.2 Logiques de description

Les logiques de description aussi appelées logiques descriptives (LD) sont une famille de langages de représentation de connaissances notamment utilisées en web sémantique pour la représentation d'ontologies. Nous nous intéressons dans cette thèse à un sous-ensemble de cette famille, les logiques de description légères souvent appelé DL-Lite qui a pour particularité d'avoir une complexité calculatoire faible comparée à d'autres logiques de description. Nous rappelons ici comment sont représentées les connaissances à l'aide des logiques de description dans le cadre d'une ontologie.

Pour représenter la connaissance en logique de description nous utilisons un couple $\Sigma$ composé des ensembles suivants : une $\mathcal{A}Box$ représentant les connaissances spécifiques aux individus (par exemple : "Toto donne un cours d'informatique.") et une $\mathcal{T}Box$ représentant les connaissances terminologiques, générales à tous les individus et liant les connaissances entre elles (par exemple : "si un individu $X$ donne un cours $Y$ alors c'est un enseignant.").

Les logiques de description légères présentent trois notions que sont les individus, les rôles, et les concepts. Les individus sont décrits à l'aide de concepts et de rôles pour représenter la base de faits et constituent la $\mathcal{A}Box$, un concept représentant une catégorisation d'un individu tandis que le rôle représente une relation entre deux individus. Par exemple le concept et le rôle suivants :

$$\texttt{Enseignant}(\texttt{toto})$$
$$\texttt{donneCours}(\texttt{toto}, \texttt{informatique})$$

$\texttt{Enseignant}(\texttt{toto})$ étant un concept définissant que Toto est un enseignant et $\texttt{donneCours}(\texttt{toto}, \texttt{informatique})$ étant un rôle définissant que Toto donne un cours d'informatique.

La $\mathcal{T}Box$ est un ensemble d'axiomes construits à partir de concepts, de rôles et de constructeurs représentant les connaissances générales de l'ontologie.

L'expressivité que permet une logique de description dépend des constructeurs autorisés. Les principaux constructeurs sont :

- Intersection de concepts : elle est notée $C1 \sqcap C2$ et représente la composition des individus appartenant à la fois au concept $C1$ et au concept $C2$.
- Union de concepts : elle est notée $C1 \sqcup C2$ et représente l'union des individus appartenant à $C1$ et à $C2$.
- Restriction universelle : elle est notée $\forall r.C$ et est composée des individus qui ne sont en relation par le rôle r qu'avec des individus du concept C.

- Restriction existentielle : elle est notée $\exists r.C$ et est composée des individus qui sont en relation par le rôle r à un individu du concept C.
- Restriction existentielle non qualifiée : elle est notée $\exists r$ et il s'agit du concept composé des individus qui sont en relation par le rôle r à un individu quelconque.
- Négation d'un concept : elle est notée $\neg C$ et est le complémentaire du concept C dans la base, c'est-à-dire l'ensemble des individus qui n'appartiennent pas au concept C.
- Inverse d'un rôle : il est noté $r^-$ et tel que si un individu $a$ est en relation avec un individu $b$ par le rôle $r$ alors $b$ est en relation avec $a$ par le rôle $r^-$.

Un exemple d'axiome avec la restriction existentielle permettant de définir qu'un individu d'un concept est mis en relation avec un autre individu par un rôle :

$$\exists \texttt{donneCours.Cours}$$

Cet exemple représente une restriction existentielle définissant qu'il existe un cours donné par un individu. Une caractéristique intéressante est la possibilité de représenter des individus appartenant à un domaine ouvert.

Les logiques de description définissent deux constructeurs spéciaux qui sont $\top$ auquel appartient tous les individus et le $\bot$ auquel n'appartient aucun individu.

Nous considérons dans cette thèse une famille de logiques de description qui est DL-Lite [28]. Elle a pour avantage d'avoir un bon rapport entre expressivité et complexité. Celle-ci permet de représenter des ontologies tout en permettant leur interrogation. DL-Lite autorise l'utilisation de l'inclusion de concepts, permettant de définir que si un individu appartient à un concept alors il appartient aussi à un autre concept. Ainsi nous avons par exemple l'axiome suivant :

$$\texttt{EnseignantChercheur} \sqsubseteq \texttt{Enseignant}$$

Cette inclusion de concepts définit que tout enseignant chercheur est aussi un enseignant.

DL-Lite permet ainsi de représenter une information n'existant pas dans le langage utilisé avec par exemple une inclusion de concepts de ce type :

$$\texttt{Enseignant} \sqsubset \exists \texttt{donneCours.Cours}$$

représentant la connaissance "si un individu est un enseignant alors il existe un cours enseigné par celui-ci". Ici nous n'avons aucune information sur ce cours mis à part qu'il existe et qu'il est donné par un enseignant. DL-Lite a été proposé pour l'expression d'ontologies simples avec une complexité basse pour faciliter l'interrogation. Elle est polynomiale en fonction de la taille de l'ontologie en complexité de données. Le but de DL-Lite est de permettre de répondre à des requêtes complexes sur l'ontologie, la requête étant reformulée en un ensemble de requêtes dans la $\mathcal{T}Box$. Il existe plusieurs membres dans la famille DL-Lite dont les plus simples sont DL-Lite$_{core}$ et DL-Lite$_{\mathcal{R}}$ permettant de constituer la base du langage du web sémantique OWL 2 QL. DL-Lite$_{core}$ représente la base de tous les membre de la famille DL-Lite et possède des restrictions fortes. La négation ne peut porter que sur un rôle ou un concept de base et ne peut apparaître qu'en partie droite d'une inclusion. Une autre restriction est que seule la restriction existentielle non qualifiée de rôles basiques est autorisée. Nous pouvons alors représenter deux types de rôles, les rôles basiques $\texttt{q} = \texttt{p}|\texttt{p}^-$ avec p un rôle atomique et $\texttt{p}^-$ l'inverse de celui-ci, et les rôles généraux $\texttt{r} = \texttt{q}|\neg\texttt{q}$. De même pour les concepts nous avons les

concepts basiques $\mathtt{B} = \mathtt{A}|\exists \mathtt{q}$ avec A un concept atomique et q un rôle basique, et les concepts généraux $\mathtt{C} = \mathtt{B}|\neg\mathtt{B}$. En DL-Lite$_{core}$ la $\mathcal{T}Box$ n'autorise que les inclusions de concepts de la forme $\mathtt{B} \sqsubseteq \mathtt{C}$ avec B un concept basique et C un concept général. Dans le cas de DL-Lite$_{\mathcal{R}}$ l'inclusion de rôles de la forme $\mathtt{q} \sqsubseteq \mathtt{r}$ est aussi autorisée dans la $\mathcal{T}Box$ avec q un rôle basique et r un rôle général.

D'un point de vue sémantique les logiques de description sont définies en termes d'interprétations (de la même manière que pour les bases logiques présentées précédemment). Ainsi une interprétation $I = (\Delta, .^I)$ est composée d'un domaine d'interprétation non vide $\Delta$ et d'une fonction d'interprétation $.^I$ qui associe un élément de $\Delta$ à chaque constante, un sous-ensemble de $\Delta$ à chaque concept et une relation binaire sur $\Delta$ à chaque rôle. Les logiques de description supposent en général l'hypothèse du nom unique (UNA) ce qui impose que deux noms différents font toujours référence à deux éléments différents du domaine.

Une interprétation $I = (\Delta, .^I)$ est un modèle d'une inclusion $C1 \sqsubseteq C2$ (resp. $R1^I \sqsubseteq R2^I$) si $C1^I \subseteq C2^I$ (resp. $R1^I \subseteq R2^I$), et c'est un modèle d'une équivalence $C1 \equiv C2$ (resp. $R1^I \equiv R2^I$) si $C1^I = C2^I$ (resp. $R1^I = R2^I$). On définit alors que $I$ est un modèle d'une $\mathcal{T}Box$ si $I$ est modèle de tous ses axiomes d'inclusion ou d'équivalence. Pour une $\mathcal{T}Box$ $\mathcal{T}$ et pour un axiome $\alpha$, $\alpha$ est une conséquence de $\mathcal{T}$ (ce que l'on notera $\mathcal{T} \models \alpha$) si tout modèle de $\mathcal{T}$ est aussi un modèle de $\alpha$.

Le test de subsumption est défini par : étant donnés une $\mathcal{T}Box$ $\mathcal{T}$ et de deux concepts $C1$ et $C2$, est-ce que $\mathcal{T} \models C1 \sqsubseteq C2$ ? Une interprétation $I$ est un modèle d'une assertion $C(a)$ (resp. $R(a, b)$) si $a^I \in C^I$ (resp. $(a^I, b^I) \in R^I$). Une interprétation est un modèle d'une $\mathcal{A}Box$ si elle est un modèle de toutes ses assertions. Une interprétation $I$ est un modèle d'une base de connaissances $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ si elle est modèle de $\mathcal{T}$ une $\mathcal{T}Box$ et de $\mathcal{A}$ une $\mathcal{A}Box$. $\mathcal{K}$ est satisfiable si elle a au moins un modèle.

Les logiques de description ne seront pas plus développées. Ces bases sont suffisantes pour saisir la suite de cette thèse, de plus l'ensemble de la famille DL-Lite peut être représenté à l'aide des règles existentielles que nous présenterons par la suite.

## 1.2 Answer Set Programming

L'answer set Programming (ASP) est un paradigme de programmation principalement utilisé pour la représentation de connaissances, notamment lorsqu'il s'agit de traiter des données incomplètes [19] et permettant aussi de traiter des problèmes combinatoires [62]. L'ASP tient ses origines de la programmation logique et de la logique des défauts, et nous pouvons noter que beaucoup de travaux découlent de l'article fondateur [43]. Sa capacité à formaliser facilement des problèmes issus de l'intelligence artificielle et l'efficacité de ses solveurs sont reconnues. ASP propose une représentation des connaissances sous forme de règles permettant l'écriture d'un programme de façon déclarative, sa principale caractéristique est d'autoriser l'utilisation d'une négation par défaut (aussi appelée négation faible), contrairement à la négation forte qui représente l'information de non appartenance d'un élément à un modèle, la négation par défaut représente l'absence d'information sur l'appartenance de cet élément à un modèle. L'ensemble des solutions d'un programme est représenté par des answer set (ensembles réponses, aussi connu sous le nom de modèles stables).

Voici un exemple d'utilisation d'ASP en représentation des connaissances construit à partir du jeu de données `university`.

```
etudiant(titi).
doctorant(toto).
etudiant(X) ← doctorant(X).
nonemploye(X) ← etudiant(X), not doctorant(X).
employe(X) ← doctorant(X).
```

Dans cet exemple nous avons les deux premières règles qui sont des faits définissant que Titi est un étudiant et Toto un doctorant. La règle suivante nous indique que tout élément `X` étant un doctorant est aussi un étudiant. Puis nous ajoutons le fait que si `X` est un étudiant et qu'il n'est pas précisé que c'est un doctorant alors `X` n'est pas employé, et dans le cas où `X` est un doctorant alors nous déduisons que `X` est employé. Nous nous intéressons ici à la déduction d'informations sur les individus Titi et Toto et plus précisément nous cherchons à savoir si Titi et Toto sont employés. Intuitivement nous voulons déduire que Titi n'est pas employé et que Toto est employé. Nous appliquons alors la troisième règle avec Toto qui est un doctorant pour déduire que Toto est aussi un étudiant. Maintenant nous souhaitons appliquer la quatrième règle qui sert à déduire si un de nos individus est employé. Nous savons que Toto est un étudiant mais aussi un doctorant, nous ne pouvons donc pas déduire que Toto n'est pas employé à cause de la négation par défaut représentée par le *not* qui empêche l'application de la règle si la condition `doctorant` est respectée, néanmoins Titi est un étudiant mais nous n'avons aucune information sur le fait que celui-ci soit un doctorant ou non, nous allons donc pouvoir déduire que Titi est employé, la négation par défaut autorisant l'application d'une règle en l'absence d'information. La dernière règle sert simplement à que si un individu est doctorant alors il est employé. L'answer set ainsi trouvé est :

$$\{\texttt{etudiant(titi)}, \texttt{doctorant(toto)}, \texttt{etudiant(toto)}, \texttt{nonemploye(titi)}, \texttt{employe(toto)}\}.$$

Nous présentons ici un ASP avec symboles de fonction [5] et sans disjonction dont le vocabulaire est celui présenté dans la section 1.1.1. La sémantique d'ASP est basée sur la sémantique des programmes logiques définis que nous présentons dans un premier temps puis nous terminerons avec les limitations de ce langage pour représenter les ontologies.

### 1.2.1 Programme logique défini

Un programme logique défini est un programme à base de règles permettant de représenter les données d'un problème.

#### Syntaxe

**Définition 1.2.1** (Règle)**.** *Une* règle *est de la forme :*

$$\texttt{h} \leftarrow \texttt{b}_1, \ldots, \texttt{b}_n.$$
$$\textit{avec } n \geq 0$$

*avec* `h` *la* tête *de la règle,* $\texttt{b}_1, \ldots, \texttt{b}_n$ *le* corps*, et* $\texttt{h}, \texttt{b}_1, \ldots, \texttt{b}_n$ *des atomes. Soit* $r$ *une règle, nous notons :*

- $\texttt{tête}(r) = \texttt{h}$ *la tête de* $r$*,*
- $\texttt{corps}(r) = \{\texttt{b}_1, \ldots, \texttt{b}_n\}$ *l'ensemble des atomes du corps de* $r$*.*

*Une règle est ground si tous ses atomes sont ground.*

**Exemple 3.** *Soit la règle :*

$$r_4 = \texttt{etudiant(X)} \leftarrow \texttt{personne(X)}, \texttt{suitCours(X,Y)}.$$

*Nous pouvons traduire* $r$ *par*

*"Si l'on peut prouver que* $X$ *est une personne qui suit un cours* $Y$*, alors on conclut que* $X$ *est un étudiant"*

*avec* $\texttt{tête}(r_4) = \texttt{etudiant(X)}$ *et* $\texttt{corps}(r_4) = \{\texttt{personne(X)}, \texttt{suitCours(X,Y)}\}$

**Définition 1.2.2** (Faits/programme)**.** *Un* fait *est une règle ground sans corps. Un programme logique défini est un ensemble de règles. Un programme est ground si toutes ses règles sont ground.*

**Définition 1.2.3** (Programme safe)**.** *Soit* $r$ *une règle,* $r$ *est* safe *si toutes les variables apparaissant dans sa tête apparaissent aussi dans son corps. Un programme est safe si toutes ses règles sont safe.*

Les règles utilisées pour les programmes définis sont toujours safe.

### Sémantique

**Définition 1.2.4** (Substitution de règle)**.** *Soit* $r$ *une règle,* $\sigma(r)$ *est l'application d'une substitution ground* $\sigma$ *à l'ensemble des atomes de* $r$*. Soit* $P$ *un programme, alors* $\mathbf{G}(P)$ *est le programme* $P$ *ground, dont toutes les règles* $r$ *ont été remplacées par toutes les applications de substitutions* $\sigma(r)$ *possibles.*

**Exemple 4** (Substitution et programme ground)**.** *Soit la règle*

$$r_4 : \texttt{etudiant(X)} \leftarrow \texttt{personne(X)}, \texttt{suitCours(X,Y)}.$$

*Nous pouvons utiliser la substitution* $\sigma_1 = [\texttt{X} \leftarrow \texttt{toto}, \texttt{Y} \leftarrow \texttt{informatique}]$ *pour instancier* $r_4$ *de la manière suivante*

$$\sigma_1(r_4) : \texttt{etudiant(toto)} \leftarrow \texttt{personne(toto)}, \texttt{suitCours(toto,informatique)}.$$

**Définition 1.2.5.** *Soit* $r$ *une règle,* $A$ *un ensemble d'atomes ground et* $\sigma$ *une substitution. La règle* $\sigma(r)$ *est* applicable *pour* $A$ *si* $\texttt{corps}(\sigma(r)) \subseteq A$*.*

**Exemple 5.** *Soit la règle ground* $\sigma_1(r_4)$ *de l'exemple 4*

$$\sigma_1(r_4) : \texttt{etudiant(toto)} \leftarrow \texttt{personne(toto)}, \texttt{suitCours(toto,informatique)}.$$

*et l'ensemble d'atomes*

$$A = \{\texttt{personne(toto)}, \texttt{suitCours(toto,informatique)}\}$$

*La règle ground* $\sigma_1(r_4)$ *est applicable pour* $A$ *car* $\texttt{corps}(\sigma_1(r_4)) \subseteq A$*.*

**Définition 1.2.6** (Application de règle)**.** *Soit* $A$ *un ensemble d'atomes ground,* $\sigma(r)$ *une règle applicable pour* $A$*. L'application de* $r$ *pour* $A$ *avec* $\sigma$ *notée* $\alpha(r, \sigma, A)$ *donne un ensemble d'atomes ground* $A' = A \cup \{\texttt{tête}(\sigma(r))\}$*.*

**Exemple 6** (Application de règle)**.** *Soit la règle ground* $\sigma_1(r_1)$ *et l'ensemble d'atomes* $A$ *de l'exemple 5. L'application* $\alpha(r_1, \sigma_1, A)$ *donne l'ensemble d'atomes*

$$A' = \{\texttt{personne(toto)}, \texttt{suitCours(toto,informatique)}, \texttt{etudiant(toto)}\}.$$

Nous définissons maintenant l'opérateur de point fixe permettant le calcul des atomes déductibles à partir des faits et des règles d'un programme ground.

**Définition 1.2.7** (Opérateur de point fixe)**.** *Soit* $P$ *un programme ground et* $A$ *un ensemble d'atomes ground. L'opérateur* $T_P$ *défini par :*

$$T_P(A) = \{\texttt{tête}(r) \mid r \in P,\ \texttt{corps}(r) \subseteq A\}$$

*calcule l'ensemble des atomes déductibles à partir de* $P$ *et de* $A$*. Nous définissons*

$$T_P^0 = \emptyset \text{ et } T_P^{k+1} = T_P(T_P^k), \forall k \geq 0$$

**Définition 1.2.8** (Modèle de Herbrand)**.** *Un* ensemble d'atomes *ground $A$ est dit* clos *par rapport à un programme ground $P$ si et seulement si pour toute règle $r \in P$, $corps(r) \subseteq A \Rightarrow tête(r) \in A$. Un ensemble d'atomes clos $A \subseteq \boldsymbol{A}$ par rapport à un programme $P$ est appelé* modèle de Herbrand. *Le plus petit modèle de Herbrand d'un programme $P$ est noté $Cn(P)$ [43].*

*$\bigcup_{k\geq 0} T_P^k$ est le plus petit point fixe de $T_P$ et $Cn(P) = \bigcup_{k\geq 0} T_P^k$. $Cn(P)$ contient tous les atomes que nous pouvons déduire à partir de $P$.*

**Exemple 7** (Modèle de Herbrand)**.** *Soit le programme suivant :*

$$P_7 = \left\{ \begin{array}{l} r_1 : \texttt{personne(toto)}. \\ r_2 : \texttt{suitCours(toto, informatique)}. \\ r_3 : \texttt{enseignantChercheur(X)} \leftarrow \texttt{personne(X)}, \texttt{donneCours(X, Y)}. \\ r_4 : \texttt{etudiant(X)} \leftarrow \texttt{personne(X)}, \texttt{suitCours(X, Y)}. \end{array} \right\}$$

*Nous avons :*

$$\begin{array}{lcl} T_{P_7}^0 & = & \emptyset \\ T_{P_7}^1 & = & T_{P_7}(\emptyset) \\ & = & \{\texttt{personne(toto)}, \texttt{suitCours(toto, informatique)}\} \\ T_{P_7}^2 & = & T_{P_7}(\{\texttt{personne(toto)}, \texttt{suitCours(toto, informatique)}\}) \\ & = & \{\texttt{personne(toto)}, \texttt{suitCours(toto, informatique)}, \texttt{etudiant(toto)}\} \\ T_{P_7}^3 & = & T_{P_7}(\{\texttt{personne(toto)}, \texttt{suitCours(toto, informatique)}, \texttt{etudiant(toto)}\}) \\ & = & \{\texttt{personne(toto)}, \texttt{suitCours(toto, informatique)}, \texttt{etudiant(toto)}\} \\ T_{P_7}^{k+1} & = & T_{P_7}^k \textit{ pour tout } k \geq 3 \end{array}$$

$$Cn(P_7) = \{\texttt{personne(toto)}, \texttt{suitCours(toto, informatique)}, \texttt{etudiant(toto)}\}$$

*Nous avons donc $Cn(P_7)$ le plus petit modèle de Herbrand de $P_7$.*

## 1.2.2 Programme logique normal

Nous définissons dans un premier temps la syntaxe et la sémantique d'un programme logique normal qui est une extension du programme logique défini avec l'ajout d'un corps négatif caractérisé par une négation par défaut.

### Syntaxe

**Définition 1.2.9** (Règle normale)**.** *Une* règle normale *est de la forme :*

$$\texttt{h} \leftarrow \texttt{b}_1^+, \ldots, \texttt{b}_n^+, \texttt{not b}_1^-, \ldots, \texttt{not b}_m^-. \quad n, m \geq 0$$

*avec* h *la tête de la règle,* $\texttt{b}_1^+, \ldots, \texttt{b}_n^+$ *le* corps positif, $\texttt{not b}_1^-, \ldots, \texttt{not b}_m^-$ *le* corps négatif *et* $\texttt{h}, \texttt{b}_1^+, \ldots, \texttt{b}_n^+, \texttt{b}_1^-, \ldots, \texttt{b}_m^-$ *des atomes. Soit $r$ une règle, on note :*

- $tête(r) = \texttt{h}$ *la tête de $r$,*
- $corps^+(r) = \{\texttt{b}_1^+, \ldots, \texttt{b}_n^+\}$ *l'ensemble des atomes du corps positif de $r$,*
- $corps^-(r) = \{\texttt{b}_1^-, \ldots, \texttt{b}_m^-\}$ *l'ensemble des atomes du corps négatif de $r$.*
- $corps(r) = corps^+(r) \cup corps^-(r)$ *l'ensemble des atomes du corps de $r$.*

Comme pour les programmes définis, une règle est ground si tous ses atomes sont ground. Nous utilisons ici la négation par défaut notée *not* qui permet de représenter des informations du type :

"Si l'on peut prouver a et si l'on ne peut pas prouver b, alors on peut conclure c"

qui se traduit de cette manière :

$$\mathtt{c} \leftarrow \mathtt{a},\ \mathtt{not\ b}.$$

**Définition 1.2.10** (Faits/programme)**.** *Un* fait *est une règle ground sans corps. Un* programme logique normal *est un ensemble de règles normales. Un programme est ground si toutes ses règles sont ground.*

Remarque : Un programme logique défini est aussi un programme logique normal.

**Définition 1.2.11** (Programme safe)**.** *Soit $r$ une règle normale, $r$ est* safe *si toutes les variables apparaissant dans sa tête ou dans son corps négatif apparaissent aussi dans son corps positif. Un programme est safe si toutes ses règles sont safe.*

**Exemple 8** (Règle safe)**.** *Soit les règles :*

$$r_1 : \mathtt{a(X,Y)} \leftarrow \mathtt{b(X)},\ \mathtt{not\ c(X)}.$$

*$r_1$ n'est pas safe car la variable $Y$ apparaît dans la tête de $r_1$ mais pas dans le corps positif.*

$$r_2 : \mathtt{a(X)} \leftarrow \mathtt{b(X)},\ \mathtt{not\ c(X,Y)}.$$

*$r_2$ n'est pas safe car la variable $Y$ apparaît dans un atome du corps négatif de $r_2$ mais pas dans le corps positif.*

$$r_3 : \mathtt{a(X,Y)} \leftarrow \mathtt{b(X,Y,Z)},\ \mathtt{not\ c(X)}.$$

*$r_3$ est safe car toutes les variables apparaissant dans sa tête et dans son corps négatif apparaissent aussi dans le corps positif.*

Un programme ASP est uniquement constitué de règles safe.

**Définition 1.2.12** (Contrainte)**.** *Une* contrainte *est une règle sans tête de cette forme :*

$$\leftarrow\ \mathtt{b}_1^+,\ \ldots,\ \mathtt{b}_n^+,\ \mathtt{not\ b}_1^-,\ \ldots,\ \mathtt{not\ b}_m^-.$$

*permettant d'empêcher l'appartenance d'atomes à un modèle du programme. Une contrainte est équivalente à une règle de ce type :*

$$\mathtt{bug} \leftarrow\ \mathtt{b}_1^+,\ \ldots,\ \mathtt{b}_n^+,\ \mathtt{not\ b}_1^-,\ \ldots,\ \mathtt{not\ b}_m^-,\ \mathtt{not\ bug}.$$

*avec* bug *un atome n'apparaissant nulle part ailleurs dans le programme.*

**Exemple 9** (Contrainte)**.** *Soit la contrainte suivante :*

$$\leftarrow\ \mathtt{donneCours(X,Y)}, \mathtt{suitCours(X,Y)}.$$

*Cette contrainte empêche d'avoir un élément $X$ qui donne et qui reçoit à la fois un cours $Y$.*

Le problème des contraintes sera abordé plus en détail dans la partie 4.4 sur l'inconsistance. Les règles normales utilisées dans cette partie sont toujours safe.

### Sémantique

**Définition 1.2.13.** *Soit $r$ une règle, $A$ un ensemble d'atomes* ground. *Nous définissons :*

- *soit $\sigma$ une substitution, $\sigma(r)$ est* supportée *par $A$ si $\texttt{corps}^+(\sigma(r)) \subseteq A$,*
- *soit $\sigma$ une substitution, $\sigma(r)$ est* bloquée *par $A$ si $\texttt{corps}^-(\sigma(r)) \cap A \neq \emptyset$,*
- *$r$ est* applicable *pour $A$ s'il existe une substitution $\sigma$ telle que $\sigma(r)$ est supportée et non bloquée par $A$.*

**Exemple 10** (Règle applicable)**.** *Soit la règle*

$$r_5 : \ \texttt{maitreConference(X)} \leftarrow \texttt{enseignantChercheur(X)}, \ \texttt{labo(Z)}, \ \texttt{travaillePour(X,Z)}, \ \texttt{not doctorant(X)}.$$

*la règle $r_5$ est un ajout à l'exemple 7 et $\sigma_2 = [\texttt{X} \leftarrow \texttt{toto}, \texttt{Y} \leftarrow \texttt{informatique}, \texttt{Z} \leftarrow \texttt{leria}]$ est une substitution. Soit les ensembles d'atomes suivants*

$$\begin{aligned} A &= \{\texttt{enseignantChercheur(toto)}, \ \texttt{labo(leria)}, \ \texttt{travaillePour(toto,leria)}, \\ &\quad \texttt{doctorant(toto)}\} \\ A' &= \{\texttt{enseignantChercheur(toto)}, \ \texttt{labo(leria)}\} \\ A'' &= \{\texttt{enseignantChercheur(toto)}, \ \texttt{labo(leria)}, \ \texttt{travaillePour(toto,leria)}\}. \end{aligned}$$

*Nous avons $\sigma_2(r_5)$ qui est supportée par $A$ mais non applicable pour $A$ car aussi bloquée par $A$ à cause de l'instance* $\texttt{doctorant(toto)}$. *$\sigma_2(r_5)$ n'est pas non plus applicable pour $A'$ car elle n'est pas supportée pour $A'$, il manque l'instance* $\texttt{travaillePour(toto,leria)}$. *$\sigma_2(r_5)$ est par contre applicable pour $A''$ car elle est supportée et non bloquée pour $A''$.*

**Définition 1.2.14** (Grounding)**.** *Soit $P$ un programme, le résultat du* grounding *de $P$ est le programme ground $\mathbf{G}(P)$ qui contient toutes les règles obtenues par instanciation des règles de $P$ avec les termes du domaine d'Herbrand de $P$.*

**Définition 1.2.15** (Réduit)**.** *Le* réduit *d'un programme logique normal ground $\mathbf{G}(P)$ par un ensemble d'atomes $A$ est le programme logique défini :*

$$\mathbf{G}(P)^A = \{\texttt{tête}(r) \text{:-} \texttt{corps}^+(r). | r \in \mathbf{G}(P), \texttt{corps}^-(r) \cap A = \emptyset\}$$

**Exemple 11** (Grounding)**.** *Soit le programme suivant :*

$$P_{11} = \left\{ \begin{array}{ll} r_1 : & \texttt{doctorant(titi)}. \\ r_2 : & \texttt{enseignantChercheur(titi)}. \\ r_3 : & \texttt{travaillePour(titi,leria)}. \\ r_4 : & \texttt{labo(leria)}. \\ r_5 : & \texttt{enseignantChercheur(toto)}. \\ r_6 : & \texttt{travaillePour(toto,leria)}. \\ r_7 : & \texttt{maitreConference(X)} \leftarrow \texttt{enseignantChercheur(X)}, \ \texttt{labo(Y)}, \\ & \qquad \texttt{travaillePour(X,Y)}, \ \texttt{not doctorant(X)}. \end{array} \right\}$$

*Le résultat du grounding de $P_{11}$ est le programme ground suivant :*

$$\mathbf{G}(P_{11}) = \left\{ \begin{array}{l} r_1 : \texttt{doctorant}(\texttt{titi}). \\ r_2 : \texttt{enseignantChercheur}(\texttt{titi}). \\ r_3 : \texttt{travaillePour}(\texttt{titi}, \texttt{leria}). \\ r_4 : \texttt{labo}(\texttt{leria}). \\ r_5 : \texttt{enseignantChercheur}(\texttt{toto}). \\ r_6 : \texttt{travaillePour}(\texttt{toto}, \texttt{leria}). \\ r_7^1 : \texttt{maitreConference}(\texttt{titi}) \leftarrow \texttt{enseignantChercheur}(\texttt{titi}), \\ \quad \texttt{labo}(\texttt{titi}),\ \texttt{travaillePour}(\texttt{titi}, \texttt{titi}),\ \texttt{not doctorant}(\texttt{titi}). \\ r_7^2 : \texttt{maitreConference}(\texttt{titi}) \leftarrow \texttt{enseignantChercheur}(\texttt{titi}), \\ \quad \texttt{labo}(\texttt{toto}),\ \texttt{travaillePour}(\texttt{titi}, \texttt{toto}),\ \texttt{not doctorant}(\texttt{titi}). \\ r_7^3 : \texttt{maitreConference}(\texttt{titi}) \leftarrow \texttt{enseignantChercheur}(\texttt{titi}), \\ \quad \texttt{labo}(\texttt{leria}),\ \texttt{travaillePour}(\texttt{titi}, \texttt{leria}),\ \texttt{not doctorant}(\texttt{titi}). \\ r_7^4 : \texttt{maitreConference}(\texttt{toto}) \leftarrow \texttt{enseignantChercheur}(\texttt{toto}), \\ \quad \texttt{labo}(\texttt{titi}),\ \texttt{travaillePour}(\texttt{toto}, \texttt{titi}),\ \texttt{not doctorant}(\texttt{toto}). \\ r_7^5 : \texttt{maitreConference}(\texttt{toto}) \leftarrow \texttt{enseignantChercheur}(\texttt{toto}), \\ \quad \texttt{labo}(\texttt{toto}),\ \texttt{travaillePour}(\texttt{toto}, \texttt{toto}),\ \texttt{not doctorant}(\texttt{toto}). \\ r_7^6 : \texttt{maitreConference}(\texttt{toto}) \leftarrow \texttt{enseignantChercheur}(\texttt{toto}), \\ \quad \texttt{labo}(\texttt{leria}),\ \texttt{travaillePour}(\texttt{toto}, \texttt{leria}),\ \texttt{not doctorant}(\texttt{toto}). \\ r_7^7 : \texttt{maitreConference}(\texttt{leria}) \leftarrow \texttt{enseignantChercheur}(\texttt{leria}), \\ \quad \texttt{labo}(\texttt{titi}),\ \texttt{travaillePour}(\texttt{leria}, \texttt{titi}),\ \texttt{not doctorant}(\texttt{leria}). \\ r_7^8 : \texttt{maitreConference}(\texttt{leria}) \leftarrow \texttt{enseignantChercheur}(\texttt{leria}), \\ \quad \texttt{labo}(\texttt{toto}),\ \texttt{travaillePour}(\texttt{leria}, \texttt{toto}),\ \texttt{not doctorant}(\texttt{leria}). \\ r_7^9 : \texttt{maitreConference}(\texttt{leria}) \leftarrow \texttt{enseignantChercheur}(\texttt{leria}), \\ \quad \texttt{labo}(\texttt{leria}),\ \texttt{travaillePour}(\texttt{leria}, \texttt{leria}),\ \texttt{not doctorant}(\texttt{leria}). \end{array} \right\}$$

*Soit l'ensemble d'atomes suivant :*

$$\begin{array}{rl} A = & \{\texttt{doctorant}(\texttt{titi}), \texttt{enseignantChercheur}(\texttt{titi}), \texttt{travaillePour}(\texttt{titi}, \texttt{leria}), \\ & \texttt{labo}(\texttt{leria}), \texttt{enseignantChercheur}(\texttt{toto}), \texttt{travaillePour}(\texttt{toto}, \texttt{leria})\} \end{array}$$

*Le réduit de* $\mathbf{G}(P_{11})$ *par l'ensemble d'atomes* $A$ *est le programme suivant :*

$$\mathbf{G}(P_{11})^A = \left\{ \begin{array}{ll} r_1 : & \texttt{doctorant}(\texttt{titi}). \\ r_2 : & \texttt{enseignantChercheur}(\texttt{titi}). \\ r_3 : & \texttt{travaillePour}(\texttt{titi}, \texttt{leria}). \\ r_4 : & \texttt{labo}(\texttt{leria}). \\ r_5 : & \texttt{enseignantChercheur}(\texttt{toto}). \\ r_6 : & \texttt{travaillePour}(\texttt{toto}, \texttt{leria}). \\ r_7^4 : & \texttt{maitreConference}(\texttt{toto}) \leftarrow \texttt{enseignantChercheur}(\texttt{toto}), \\ & \quad \texttt{labo}(\texttt{titi}),\ \texttt{travaillePour}(\texttt{toto}, \texttt{titi}). \\ r_7^5 : & \texttt{maitreConference}(\texttt{toto}) \leftarrow \texttt{enseignantChercheur}(\texttt{toto}), \\ & \quad \texttt{labo}(\texttt{toto}),\ \texttt{travaillePour}(\texttt{toto}, \texttt{toto}). \\ r_7^6 : & \texttt{maitreConference}(\texttt{toto}) \leftarrow \texttt{enseignantChercheur}(\texttt{toto}), \\ & \quad \texttt{labo}(\texttt{leria}),\ \texttt{travaillePour}(\texttt{toto}, \texttt{leria}). \\ r_7^7 : & \texttt{maitreConference}(\texttt{leria}) \leftarrow \texttt{enseignantChercheur}(\texttt{leria}), \\ & \quad \texttt{labo}(\texttt{titi}),\ \texttt{travaillePour}(\texttt{leria}, \texttt{titi}). \\ r_7^8 : & \texttt{maitreConference}(\texttt{leria}) \leftarrow \texttt{enseignantChercheur}(\texttt{leria}), \\ & \quad \texttt{labo}(\texttt{toto}),\ \texttt{travaillePour}(\texttt{leria}, \texttt{toto}). \\ r_7^9 : & \texttt{maitreConference}(\texttt{leria}) \leftarrow \texttt{enseignantChercheur}(\texttt{leria}), \\ & \quad \texttt{labo}(\texttt{leria}),\ \texttt{travaillePour}(\texttt{leria}, \texttt{leria}). \end{array} \right\}$$

*où nous avons supprimé les règles $r_7^1$, $r_7^2$ et $r_7^3$ car* $\texttt{doctorant}(\texttt{titi})$ *appartient à leur corps négatif, puis nous avons supprimé le corps négatif des règles $r_7^4$, $r_7^5$, $r_7^6$, $r_7^7$, $r_7^8$ et $r_7^9$ pour obtenir un programme logique défini.*

Avec le réduit nous transformons un programme logique normal ground en programme logique défini ground en ne conservant que les règles non bloquées avec un certain ensemble d'atomes. Le réduit nous permet d'appliquer l'opérateur de conséquence afin de calculer un modèle pour notre programme.

**Définition 1.2.16** (Answer set)**.** *Soit $P$ un programme, un answer set de $P$ (aussi appelé modèle stable) est un ensemble d'atomes ground $AS$ tel que $AS = Cn(\mathbf{G}(P)^{AS})$.*

**Exemple 12** (Answer set)**.** *Soit le programme $\mathbf{G}(P_{11})$ de l'exemple 11 et les ensembles d'atomes suivants :*

$$\begin{aligned}
A = \quad & \{\texttt{doctorant}(\texttt{titi}), \texttt{enseignantChercheur}(\texttt{titi}), \texttt{travaillePour}(\texttt{titi}, \texttt{leria}),\\
& \texttt{labo}(\texttt{leria}), \texttt{enseignantChercheur}(\texttt{toto}), \texttt{travaillePour}(\texttt{toto}, \texttt{leria})\}\\
A' = \quad & \{\texttt{doctorant}(\texttt{titi}), \texttt{enseignantChercheur}(\texttt{titi}), \texttt{travaillePour}(\texttt{titi}, \texttt{leria}),\\
& \texttt{labo}(\texttt{leria}), \texttt{enseignantChercheur}(\texttt{toto}), \texttt{travaillePour}(\texttt{toto}, \texttt{leria}),\\
& \texttt{maitreConference}(\texttt{toto}), \texttt{maitreConference}(\texttt{titi})\}\\
A'' = \quad & \{\texttt{doctorant}(\texttt{titi}), \texttt{enseignantChercheur}(\texttt{titi}), \texttt{travaillePour}(\texttt{titi}, \texttt{leria}),\\
& \texttt{labo}(\texttt{leria}), \texttt{enseignantChercheur}(\texttt{toto}), \texttt{travaillePour}(\texttt{toto}, \texttt{leria}),\\
& \texttt{maitreConference}(\texttt{toto})\}
\end{aligned}$$

*Nous avons $\mathbf{G}(P_{11})^A = \mathbf{G}(P_{11})^{A'} = \mathbf{G}(P_{11})^{A''}$ et*

$$\begin{aligned}
Cn(\mathbf{G}(P_{11})^A) = \quad & \{\texttt{doctorant}(\texttt{titi}), \texttt{enseignantChercheur}(\texttt{titi}),\\
& \texttt{travaillePour}(\texttt{titi}, \texttt{leria}), \texttt{labo}(\texttt{leria}), \texttt{enseignantChercheur}(\texttt{toto}),\\
& \texttt{travaillePour}(\texttt{toto}, \texttt{leria}), \texttt{maitreConference}(\texttt{toto})\}
\end{aligned}$$

*Donc l'ensemble $A$ n'est pas un answer set car l'atome* $\texttt{maitreConference}(\texttt{toto})$ *est manquant, $A'$ n'est pas un answer set car* $\texttt{maitreConference}(\texttt{titi})$ *est en trop, par contre $A''$ est un answer set car $Cn(\mathbf{G}(P_{11})^{A''}) = A''$*

Un programme ASP peut avoir aucun, un ou plusieurs answer set. L'ensemble des answer set d'un programme est noté $\mathcal{AS}$.

### 1.2.3 Limitations

Comme défini dans la syntaxe des programmes logiques normaux, ASP ne permet pas l'utilisation de règles n'étant pas safe. Le sujet principal de cette thèse est la représentation de connaissances provenant des logiques de description en absence d'informations. Le problème soulevé est que les logiques de description ne sont pas directement représentables en ASP en utilisant des règles safe. En effet, les logiques de descriptions légères, aussi appelées DL-Lite, sont multiples et ont certaines pour particularité d'autoriser la restriction existentielle que l'on peut modéliser ainsi :

$$\exists \texttt{donneCours.Cours}$$

Cette restriction n'est pas représentable en ASP à cause de la quantification existentielle. Un autre exemple est celui de l'inclusion de concepts avec une restriction universelle de ce type :

$$\texttt{enseignantChercheur} \sqsubseteq \exists \texttt{donneCours}.$$

Nous pourrions alors traduire cette inclusion en ASP de la manière suivante :

$$\texttt{donneCours}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{enseignantChercheur}(\texttt{X}).$$

Le problème de ces restrictions existentielles est qu'elles nécessitent l'utilisation de règles non safe et cette représentation n'est pas possible en ASP classique. Une représentation similaire existe cependant en ASP à l'aide de symboles de fonction. Nous pouvons alors écrire la règle précédente de la manière suivante :

$$\texttt{enseigne}(\texttt{X}, \texttt{f}(\texttt{X})) \leftarrow \texttt{enseignant}(\texttt{X}).$$

Le problème est que l'utilisation de symboles de fonction en ASP autorise l'écriture de programmes indécidables pour lesquels il peut exister une infinité d'answer set pouvant être infinis. Un exemple de programme ayant un answer set infini est la représentation de la parentalité avec le programme suivant :

$$\begin{aligned}&\texttt{parent}(\texttt{titi}, \texttt{toto}).\\ &\texttt{parent}(\texttt{Y}, \texttt{f}(\texttt{Y})) \leftarrow \texttt{parent}(\texttt{X}, \texttt{Y}).\end{aligned}$$

L'answer set de ce programme est alors infini :

$$AS = \{\texttt{parent}(\texttt{titi}, \texttt{toto}), \texttt{parent}(\texttt{toto}, \texttt{f}(\texttt{toto})), \texttt{parent}(\texttt{f}(\texttt{toto}), \texttt{f}(\texttt{f}(\texttt{toto}))), \ldots\}$$

Ainsi l'opérateur de point fixe ne s'arrête jamais car le programme crée une nouvelle instance de symbole de prédicat `parent` en imbriquant les symboles de fonction à chaque application de règle. Si nous ajoutons de la négation par défaut nous pouvons en plus créer une infinité de modèles.

$$\begin{aligned}&\texttt{humain}(\texttt{toto}).\\ &\texttt{parent}(\texttt{X}, \texttt{f}(\texttt{X})) \leftarrow \texttt{humain}(\texttt{X}).\\ &\texttt{humain}(\texttt{f}(\texttt{X})) \leftarrow \texttt{humain}(\texttt{X}).\\ &\texttt{homme}(\texttt{X}) \leftarrow \texttt{humain}(\texttt{X}), \texttt{not femme}(\texttt{X}).\\ &\texttt{femme}(\texttt{X}) \leftarrow \texttt{humain}(\texttt{X}), \texttt{not homme}(\texttt{X}).\end{aligned}$$

Ce qui nous amène à une infinité d'answer set où pour chaque humain créé nous avons le choix entre homme ou femme, étant donné qu'il existe une infinité d'humains il existe une infinité de possibilités d'alterner entre homme et femme. Si nous reprenons l'exemple avec seulement les quatres premiers answer set.

$$\begin{aligned}AS_1 = \; &\{\texttt{humain}(\texttt{toto}), \texttt{homme}(\texttt{toto}), \texttt{parent}(\texttt{toto}, \texttt{f}(\texttt{toto})),\\ &\texttt{humain}(\texttt{f}(\texttt{toto})), \texttt{homme}(\texttt{f}(\texttt{toto})), \ldots\}\\ AS_2 = \; &\{\texttt{humain}(\texttt{toto}), \texttt{homme}(\texttt{toto}), \texttt{parent}(\texttt{toto}, \texttt{f}(\texttt{toto})),\\ &\texttt{humain}(\texttt{f}(\texttt{toto})), \texttt{femme}(\texttt{f}(\texttt{toto})), \ldots\}\\ AS_3 = \; &\{\texttt{humain}(\texttt{toto}), \texttt{femme}(\texttt{toto}), \texttt{parent}(\texttt{toto}, \texttt{f}(\texttt{toto})),\\ &\texttt{humain}(\texttt{f}(\texttt{toto})), \texttt{homme}(\texttt{f}(\texttt{toto})), \ldots\}\\ AS_4 = \; &\{\texttt{humain}(\texttt{toto}), \texttt{femme}(\texttt{toto}), \texttt{parent}(\texttt{toto}, \texttt{f}(\texttt{toto})),\\ &\texttt{humain}(\texttt{f}(\texttt{toto})), \texttt{femme}(\texttt{f}(\texttt{toto})), \ldots\}\\ &\ldots\end{aligned}$$

Le nombre d'answer set augmentant, ici, exponentiellement en fonction du nombre d'imbrications de symboles de fonction.
Pour pallier ce problème de représentation d'ontologie, nous avons décidé d'étudier un langage permettant de représenter des logiques de description telles que DL-Lite et ayant des bases logiques proches d'ASP. Ce langage est celui des *règles existentielles* dont le domaine de recherche est dominé par l'étude de la décidabilité des programmes utilisant des variables existentielles. Nous espérons alors trouver des solutions pour traiter les variables existentielles en tête de règle et les programmes qui en découlent dont les modèles sont potentiellement infinis.

## 1.3 Règles existentielles

Les règles existentielles [15] forment un langage à base de règles à la croisée de la représentation des connaissances et des bases de données. Elles ont été étudiées dans le cadre des graphes conceptuels sous le nom de règles de graphes [65], et plus tôt dans le monde des bases de données sous le nom de TGDs [4] (tuple generating dependencies). Nous pouvons aussi les retrouver sous le nom de $Datalog^{\exists}$ [22], renommées par la suite $Datalog\pm$ dans [23], ou encore $\forall\exists - rules$ dans [14]. Les règles existentielles couvrent une grande partie des logiques de description et plus particulièrement les logiques de description dites légères, de type DL-Lite [9]. En effet, celles-ci sont utilisées pour l'interrogation, par des requêtes conjonctives, de bases de connaissances munies d'ontologies expressives. Ces règles ont pour particularité d'avoir une tête dont les variables peuvent être existentiellement quantifiées. Par exemple, l'inclusion :

$$\texttt{enseignantChercheur} \sqsubseteq \exists \texttt{donneCours}.$$

signifiant que pour tout enseignant il existe un cours enseigné par celui-ci sera représenté avec la règle existentielle suivante :

$$\texttt{enseignantChercheur}(\texttt{X}) \rightarrow \texttt{donneCours}(\texttt{X}, \texttt{Y}).$$

Cette particularité rend la plupart des problèmes indécidables, c'est pour cela qu'une grande partie des travaux sur les règles existentielles traitent de la décidabilité et de la définition des différentes classes assurant la décidabilité d'un problème. Notre but est d'étudier la sémantique de ce langage, permettant de représenter le fragment des logiques de description légères, afin d'identifier s'il est possible de combiner efficacement non-monotonie et utilisation de variables existentielles en tête de règle. Pour cela nous allons tout d'abord définir la sémantique des règles existentielles puis nous détaillerons les différents choix possibles pour obtenir un langage répondant à nos attentes. Nous essaierons aussi de faire le rapprochement entre les syntaxes et les sémantiques des règles existentielles et d'ASP.

### 1.3.1 Syntaxe

Les règles existentielles utilisent le vocabulaire de la logique du premier ordre défini dans la section 1.1.1 mais sans symboles de fonction.

**Définition 1.3.1** (Ensemble de faits existentiels)**.** *Un* ensemble de faits existentiels $\mathcal{F}$ *est un ensemble d'atomes.*

Un ensemble de faits existentiels peut contenir des variables qui seront alors existentiellement quantifiées. Les notions de variables et constantes d'un atome sont généralisées à un ensemble de faits existentiels. Soit $\mathcal{F}$ un ensemble de faits existentiels, nous avons $var(\mathcal{F})$ (resp. $const(\mathcal{F})$) l'ensemble des variables (resp. constantes) des atomes de $\mathcal{F}$. Un *ensemble de faits existentiels* est *ground* ou *instancié* si tous ses atomes sont ground. Un ensemble de faits existentiels ground est un ensemble de faits du point de vue ASP.

**Définition 1.3.2** (Règle)**.** *Une* règle existentielle *est une formule logique de la forme :*

$$\forall \overrightarrow{X} \forall \overrightarrow{Y} (\texttt{b}_1 \wedge \cdots \wedge \texttt{b}_n \rightarrow \exists \overrightarrow{Z} (\texttt{h}_1 \wedge \cdots \wedge \texttt{h}_m)).$$
$$n, m > 0$$

*avec* $\texttt{h}_1, \ldots, \texttt{h}_m, \texttt{b}_1, \ldots, \texttt{b}_n$ *des atomes.* $\overrightarrow{X}$ *l'ensemble de variables appartenant uniquement au corps de la règle,* $\overrightarrow{Y}$ *l'ensemble de variables appartenant à la fois au corps et à la tête que nous appellerons* frontière *et* $\overrightarrow{Z}$ *les variables n'apparaissant que dans la tête de la règle.*
*Étant donné qu'il n'y a pas de confusion possible les règles existentielles sont écrites sous forme de règle sans les quantificateurs.*

$$\mathtt{b}_1,\ \ldots,\ \mathtt{b}_n \rightarrow\ \mathtt{h}_1,\ \ldots,\ \mathtt{h}_m.$$

*avec* $\mathtt{h}_1,\ \ldots,\ \mathtt{h}_m$ *la* tête *de la règle,* $\mathtt{b}_1,\ \ldots\ ,\mathtt{b}_n$ *le* corps. *Soit* $r$ *une règle, nous notons :*

- $\mathit{tête}(r) = \{\mathtt{h}_1, \ldots, \mathtt{h}_m\}$ *l'ensemble des atomes de la tête de* $r$*,*
- $\mathit{corps}(r) = \{\mathtt{b}_1, \ldots, \mathtt{b}_n\}$ *l'ensemble des atomes du corps de* $r$*.*

*Nous notons* $\mathit{var}(r)$ *(resp.* $\mathit{const}(r)$*) l'ensemble des variables (resp. constantes) de* $r$*. Une* règle *est* ground *si tous ses atomes sont ground.*

**Exemple 13.** *Soit* $r_1$ *la règle suivante :*

$$r_1 : \mathtt{enseignant}(\mathtt{X}) \rightarrow\ \mathtt{enseigne}(\mathtt{X}, \mathtt{Y}).$$

*La variable* X *est quantifiée universellement et fait partie de la frontière car elle apparaît à la fois dans le corps et la tête de* $r_1$ *tandis que la variable* Y *est quantifiée existentiellement, n'apparaissant pas dans le corps de* $r_1$*.*

**Définition 1.3.3.** *Un programme* $P$ *est un couple* $P = (\mathcal{F}, \mathcal{R})$ *avec* $\mathcal{F}$ *un ensemble de faits existentiels et* $\mathcal{R}$ *un ensemble de règles existentielles.*

En règles existentielles les ensembles de faits existentiels et de règles existentielles sont distincts comme pour les logiques de description, ce qui permet de différencier les informations qui sont générales des informations à propos d'éléments précis. Nous verrons par la suite que certains traitements sont effectués uniquement sur l'ensemble des règles et non sur le programme complet.

**Exemple 14** (Programme de règles existentielles)**.** *Soit* $P_{14} = (\mathcal{F}_{14}, \mathcal{R}_{14})$ *un programme de règles existentielles avec :*

$$\mathcal{R}_{14} = \left\{\ r_1 : \mathtt{enseignantChercheur}(\mathtt{X}) \rightarrow\ \mathtt{donneCours}(\mathtt{X}, \mathtt{Y}).\ \right\}$$

*et*

$$\mathcal{F}_{14} = \{\mathtt{enseignantChercheur}(\mathtt{toto})\}$$

*Dans ce programme nous avons un ensemble de faits existentiels* $\mathcal{F}_{14}$ *qui nous dit que Toto est un enseignant et une règle existentielle qui définit que pour tout élément* X *qui est un enseignant alors il existe un cours* Y *enseigné par* X*.*

**Définition 1.3.4** (Règle existentielle safe)**.** *Une règle existentielle* $r$ *est dite safe si toutes les variables apparaissant dans sa tête apparaissent dans son corps soit* $\mathit{var}(\mathit{tête}(r)) \subseteq \mathit{var}(\mathit{corps}(r))$*.*

Un programme ne possédant pas de règles dont la tête est quantifiée existentiellement est un programme logique défini. Un programme défini est donc un programme de règles existentielles safe, il n'existe alors plus que des variables universellement quantifiées.

## 1.3.2 Sémantique

Les règles existentielles contrairement aux programmes logiques définis et ASP n'est pas défini sur l'univers de Herbrand. Certains éléments du domaine non nommés peuvent être capturés grâce aux variables existentielles.

**Définition 1.3.5** (Homomorphisme)**.** *Soit* $\mathcal{F}$ *et* $\mathcal{F}'$ *deux ensembles d'atomes. Un homomorphisme de* $\mathcal{F}$ *vers* $\mathcal{F}'$ *est une substitution* $\pi$ *des variables de* $\mathcal{F}$ *vers les termes de* $\mathcal{F}'$ *telle que* $\pi(\mathcal{F}) \subseteq \mathcal{F}'$*.*

**Définition 1.3.6** (Conséquence sémantique)**.** *Soit $\mathcal{F}$ et $\mathcal{F}'$ deux ensembles de faits existentiels sur un langage $\mathcal{L}$. $\mathcal{F}'$ est* conséquence sémantique *de $\mathcal{F}$ ($\mathcal{F} \models \mathcal{F}'$) si tout modèle de $\mathcal{F}$ est un modèle de $\mathcal{F}'$.*

**Théorème 1.3.1.** *Soit $\mathcal{F}$ un ensemble d'atomes, et $\mathcal{F}'$ un ensemble fini d'atomes. Alors, $\mathcal{F} \models \mathcal{F}'$ si et seulement si il existe un homomorphisme de $\mathcal{F}'$ vers $\mathcal{F}$.*

**Définition 1.3.7** (Justification)**.** *Soit $\mathcal{F}$ un ensemble de faits existentiels sur un langage $\mathcal{L}$ et $I$ une interprétation de $\mathcal{L}$. Une justification de $\mathcal{F}$ dans $I$ est un homomorphisme de $\mathcal{F}$ dans $I$.*

**Propriété 1.3.1.** *Si $\mathcal{F}$ est un ensemble de faits existentiels ground et $I$ une interprétation alors nous avons les assertions suivantes équivalentes :*

- *$I$ est modèle de $\mathcal{F}$ ;*
- *l'identité est une justification de $\mathcal{F}$ dans $I$ ;*
- *$\mathcal{F} \subseteq I$ ;*
- *il existe une unique justification de $\mathcal{F}$ dans $I$*

**Exemple 15.** *Soit $\mathcal{F} = \{\mathtt{p(a)}, \mathtt{q(b)}\}$ un ensemble de faits existentiels ground sur un langage $\mathcal{L}$ et $I = \{\mathtt{p(a)}, \mathtt{q(b)}, \mathtt{p(c)}\}$ une interprétation de $\mathcal{L}$ alors nous avons une justification de $\mathcal{F}$ dans $I$. Étant donné qu'il existe une justification de $\mathcal{F}$ dans $I$ nous avons donc $I$ modèle de $\mathcal{F}$. Si nous enlevons un des atomes $\mathtt{p(a)}$ ou $\mathtt{q(b)}$ alors nous n'avons plus de justification de $\mathcal{F}$ dans $I$ et $I$ n'est plus un modèle.*

**Définition 1.3.8** (Application de règle)**.** *Soit $\mathcal{F}$ un ensemble de faits existentiels sur un langage $\mathcal{L}$ et $r$ une règle existentielle sur $\mathcal{L}$. $r$ est dite applicable par $\mathcal{F}$ s'il existe un homomorphisme $\pi$ de $\mathtt{corps}(r)$ dans $\mathcal{F}$. Dans ce cas, une application de $r$ par $\mathcal{F}$ produit un ensemble d'atomes $\alpha(\mathcal{F}, r, \pi) = \mathcal{F} \cup fresh(\mathtt{tête}(r))$, où fresh est une substitution bijective des variables existentielles de $\mathtt{tête}(r)$ vers un ensemble de variables fraîches (i.e., des variables générées par l'application et n'apparaissant nulle part ailleurs dans le programme). On dit de $\alpha(\mathcal{F}, r, \pi)$ que c'est une dérivation immédiate de $\mathcal{F}$.*

Les variables fraîches sont des variables permettant la représentation d'une information sur un nouvel individu ne faisant pas partie du programme initial.

**Exemple 16** (Application d'une règle existentielle)**.** *Soit $P_{14} = (\mathcal{F}_{14}, \mathcal{R}_{14})$ le programme de l'exemple 14 avec :*

$$\mathcal{R}_{14} = \{\ r_1 : \mathtt{enseignantChercheur(X)} \rightarrow \mathtt{donneCours(X, Y)}.\ \}$$

*et*

$$\mathcal{F}_{14} = \{\mathtt{enseignantChercheur(toto)}\}$$

*Une application de la règle $r_1$ avec l'homomorphisme $\pi(\mathtt{X}) = \mathtt{toto}$ donne l'ensemble de faits existentiels $\alpha(\mathcal{F}_{14}, r_1, \pi) = \{\mathtt{enseignant(toto)}, \mathtt{donneCours(toto, Y_0)}\}$ avec $\mathtt{Y_0}$ une variable fraîche. Cette variable fraîche n'apparaît pas dans le programme mais peut par la suite provoquer l'application d'une autre règle s'il existe un homomorphisme la mettant en jeu.*

**Définition 1.3.9** (Dérivation-$\mathcal{R}$)**.** *Soit $\mathcal{F}$ un ensemble de faits existentiels et $\mathcal{R}$ un ensemble de règles existentielles. Une* dérivation-$\mathcal{R}$ *(de $\mathcal{F}$ vers $\mathcal{F}^k$ suivant $\mathcal{R}$) est une séquence finie ($\mathcal{F}^0 = \mathcal{F}$), $(r_0, \pi_0, \mathcal{F}^1)$,...,$(r_{k-1}, \pi_{k-1}, \mathcal{F}^k)$ telle que pour tout $0 \leq i < k$, $r_i \in \mathcal{R}$ et $\pi_i$ est un homomorphisme de $\mathtt{corps}(r_i)$ vers $\mathcal{F}^i$ tel que $\mathcal{F}^{i+1} = \alpha(\mathcal{F}^i, r_i, \pi_i)$. $\mathcal{F}^{i+1}$ est une dérivation immédiate de $\mathcal{F}^i$.*

**Définition 1.3.10** (Saturation)**.** *Soit $\mathcal{F}$ un ensemble de faits existentiels et $\mathcal{R}$ un ensemble de règles existentielles. Nous notons $\Pi(\mathcal{R},\mathcal{F})$ l'ensemble des homomorphismes d'une règle $r \in \mathcal{R}$ dans $\mathcal{F}$ :* $\Pi(\mathcal{R},\mathcal{F}) = \{(r,\pi) | r \in \mathcal{R}$ *et $\pi$ est un homomorphisme de* $\texttt{corps}(r)$ *dans $\mathcal{F}\}$.*
*La* saturation directe *de $\mathcal{F}$ par $\mathcal{R}$ est définie comme suit :*

$$\alpha(\mathcal{F},\mathcal{R}) = \mathcal{F} \cup \bigcup_{(r,\pi)\in\Pi(\mathcal{R},\mathcal{F})} \pi(\texttt{tête}(r)).$$

*La* k-saturation *de $\mathcal{F}$ par $\mathcal{R}$ est notée $\alpha_k(\mathcal{F},\mathcal{R})$ et est définie par :*

$$\alpha_0(\mathcal{F},\mathcal{R}) = \mathcal{F} \text{ et, pour } i > 0,\ \alpha_i(\mathcal{F},\mathcal{R}) = \alpha(\alpha_{i-1}(\mathcal{F},\mathcal{R}),\mathcal{R}).$$

Nous notons $\alpha_\infty(\mathcal{F},\mathcal{R}) = \bigcup_{k\in\mathbb{N}} \alpha_k(\mathcal{F},\mathcal{R})$. $\alpha_\infty$ est possiblement infinie.

**Propriété 1.3.2** (Modèle d'un programme)**.** *Soit $\mathcal{F}$, $\mathcal{F}'$ et $\mathcal{F}''$ trois ensembles de faits existentiels, et $\mathcal{R}$ un ensemble fini de règles existentielles. Alors, $(\mathcal{F},\mathcal{R}) \models \mathcal{F}''$ si et seulement s'il existe une k-saturation finie de $\mathcal{F}$ vers $\mathcal{F}'$ telle que $\mathcal{F}' \models \mathcal{F}''$.*

**Exemple 17.** *Soit $\mathcal{F} = \{\texttt{p(a)}, \texttt{q(b)}\}$ un ensemble de faits existentiels et $r_1$ : $(\texttt{p(a)} \rightarrow \texttt{p(c).})$ une règle. Soit $P$ le programme constitué de $\mathcal{F}$ et de $r_1$. Nous avons $I = \{\texttt{p(a)}, \texttt{q(b)}, \texttt{p(c)}\}$ un modèle de $P$ et $I' = \{\texttt{p(a)}, \texttt{q(b)}, \texttt{p(c)}, \texttt{p(b)}\}$ un deuxième modèle. $I$ est le plus petit modèle que nous pouvons déduire de $P$ (si l'on enlève un seul atome de $I$ ce n'est plus un modèle).*

Nous avons défini les éléments permettant de déterminer un modèle à partir d'un ensemble de règles existentielles et d'ensembles de faits existentiels, mais les variables existentielles en tête de règles rendent le calcul d'un modèle en règles existentielles indécidable [17] [72]. En effet, il existe des programmes dont les modèles sont infinis.

**Exemple 18** (Programme à modèle infini)**.** *Soit l'ensemble de règles*

$$\mathcal{R}_{18} = \{\ \texttt{humain(X)} \rightarrow \texttt{humain(Y)}, \texttt{parent(X,Y).}\ \}$$

*et l'ensemble de faits existentiels*

$$\mathcal{F}_{18} = \{\texttt{humain(toto)}\}$$

*La saturation de ce programme donne un modèle infini.*

$\alpha_1(\mathcal{F},\mathcal{R}) = \{\texttt{humain(toto)}, \texttt{humain}(\texttt{Y}_0), \texttt{parent(toto,}\texttt{Y}_0)\}$
$\alpha_2(\mathcal{F},\mathcal{R}) = \{\texttt{humain(toto)}, \texttt{humain}(\texttt{Y}_0), \texttt{parent(toto,}\texttt{Y}_0), \texttt{humain}(\texttt{Y}_1), \texttt{parent}(\texttt{Y}_0,\texttt{Y}_1)\}$
$\ldots$
$\alpha_\infty(\mathcal{F},\mathcal{R}) = \{\texttt{humain(toto)}, \texttt{humain}(\texttt{Y}_0), \texttt{parent(toto,}\texttt{Y}_0), \texttt{humain}(\texttt{Y}_1), \texttt{parent}(\texttt{Y}_0,\texttt{Y}_1), \ldots\}$

Ce problème vient de la saturation qui génère une infinité de variables fraîches impliquées dans une boucle récursive d'applications de règles. Heureusement il existe certains programmes dont le calcul de modèle ne peut s'effectuer en un temps fini pour lesquels on peut identifier de la redondance dans les informations déduites par l'application d'une règle. Il existe pour cela différents algorithmes de marche avant en règles existentielles utilisant la saturation appelés *chase* qui permettent de calculer des modèles équivalents à certains modèles infinis en s'intéressant à la redondance dans les ensembles de faits existentiels.

Par exemple, soit l'ensemble de faits existentiels $\mathcal{F}_{18} = \{\texttt{parent(toto,titi)}\}$ et la règle existentielle $r = (\texttt{parent(X,Y)} \rightarrow \texttt{parent(X,Z).})$, une saturation nous donnerait un modèle infini avec la dérivation suivante :

$$\mathcal{F}_{18} = \mathcal{F}^0_{18}, \mathcal{F}^1_{18} = \{\texttt{parent(toto,titi)}, \texttt{parent(toto,}\texttt{Y}_0)\},$$
$$\mathcal{F}^2_{18} = \{\texttt{parent(toto,titi)}, \texttt{parent(toto,}\texttt{Y}_0), \texttt{parent(toto,}\texttt{Y}_1)\}, \ldots.$$

Néanmoins il est possible de détecter une redondance dans $\mathcal{F}^2_{18}$ qui rend $\mathcal{F}^1_{18}$ et $\mathcal{F}^2_{18}$ sémantiquement équivalents, car $\texttt{parent}(\texttt{toto}, \texttt{Y}_0)$ et $\texttt{parent}(\texttt{toto}, \texttt{Y}_1)$ représentent la même information obtenue par deux applications définissant qu'il existe un individu parent de Toto. Mais les deux individus $\texttt{Y}_0$ et $\texttt{Y}_1$ représentent tous deux la même information créée par la même application et cette information est donc redondante. Nous allons donc détailler ces différents chase pour terminer sur le calcul de modèles en règles existentielles.

### 1.3.3 Chase

Nous avons présenté dans la partie précédente un algorithme de marche avant, appelé saturation, à l'aide de la dérivation correspondant à l'application d'un ensemble de règles existentielles à partir d'un ensemble de faits existentiels. Seulement l'application d'une règle avec une variable existentielle en tête peut ajouter un atome dont l'information est redondante avec les atomes déjà présent dans la base de faits. Lors de l'utilisation d'un algorithme de marche avant naïf, ces informations redondantes peuvent donner lieu à des modèles infinis. Il existe pour cela différents chase basés sur le traitement de la redondance d'information lors de l'application d'une règle. Nous définissons alors plusieurs critères pour calculer une dérivation d'un ensemble de faits existentiels. Ces différents chase sont des algorithmes corrects et complets permettant le calcul de modèles en un temps fini pour un programme de règles existentielles et donc de déterminer que la dérivation-$\mathcal{R}$ finie d'un ensemble de faits existentiels $\mathcal{F}$ à partir des règles du programme $P = (\mathcal{F}, \mathcal{R})$ est modèle de celui-ci.

#### Les différents chase

Il existe différents chase pour traiter les règles existentielles, dont une bonne partie est issue des bases de données. Ces chase sont basés sur des critères pour l'application des règles. Nous pouvons ainsi définir les chase en fonction d'un critère $\mathcal{C}r$ et nous appellerons le chase correspondant à ce critère $\mathcal{C}r - chase$. Nous présentons ici les chase dans un ordre d'efficacité, le premier étant le chase s'arrêtant le moins souvent et le dernier sera celui ayant le plus de cas d'arrêts. Le premier chase considéré est donc l'*oblivious-chase* (parfois appelé *naive-chase*). Pour ce premier chase, une règle est appliquée selon un homomorphisme $\pi$ seulement si cette règle n'a pas déjà été appliquée selon le même homomorphisme. Ce critère est impératif sinon le chase ne s'arrête jamais dès lors que nous avons une variable existentielle.

**Définition 1.3.11** (Critère oblivious)**.** *Soit $\mathcal{F}^i$ un ensemble de faits existentiels et $r$ une règle, l'ensemble de faits existentiels $\mathcal{F}^{i+1} = \alpha(\mathcal{F}^i, r, \pi)$ est le résultat de l'application de la règle $r$ selon l'homomorphisme $\pi$ dans $\mathcal{F}^i$ seulement si une application de $r$ avec $\pi$ n'a pas déjà été effectuée.*

**Exemple 19.** *Soit le programme $P_{19} = (\mathcal{F}_{19}, \mathcal{R}_{19})$ avec :*

$$\mathcal{R}_{19} = \left\{ \; r_1 : \texttt{p}(\texttt{X}, \texttt{Y}) \rightarrow \; \texttt{q}(\texttt{Z}). \; \right\}$$

*et*

$$\mathcal{F}_{19} = \{\texttt{p}(\texttt{a}, \texttt{b})\}$$

*Avec une saturation classique le modèle de ce programme serait infini, la règle $r_1$ serait appliquée infiniment car chaque application de la règle crée une nouvelle instance de l'atome $\texttt{q}(\texttt{Z}_0)$ avec $\texttt{Z}_0$ une nouvelle variable fraîche. Or, toute nouvelle instance $\texttt{q}(\texttt{Z}_\texttt{i})$ provenant de l'application de $r_1$ par le même homomorphisme n'apporte aucune nouvelle information, nous savons qu'il existe au moins un élément $\texttt{Z}_0$, il est donc inutile de l'appliquer plusieurs fois.*
*Par contre, l'application de l'oblivious chase sur ce programme se termine après une simple application de la règle $r_1$, en effet $\mathcal{F}^1_{19} = \alpha(\mathcal{F}_{19}, r_1, \pi)$ avec $\pi(\texttt{X}) = \texttt{a}$ et $\pi(\texttt{Y}) = \texttt{b}$ est la seule application*

*possible. Il n'existe pas d'autres applications avec $\mathcal{F}^1_{19}$ que $\alpha(\mathcal{F}^1_{19}, r_1, \pi)$ mais la règle a déjà été appliquée avec $\pi$, nous ne pouvons donc pas l'appliquer de nouveau. L'oblivious chase se termine donc avec $\mathcal{F}^1_{19} = \{\mathtt{p(a,b)}, \mathtt{q(Z_0)}\}$*

Il existe par contre des cas où l'oblivious chase ne se termine pas comme pour l'exemple suivant.

**Exemple 20.** *Soit le programme $P_{20} = (\mathcal{F}_{20}, \mathcal{R}_{20})$ avec :*

$$\mathcal{R}_{20} = \{\ r_1 : \mathtt{p(X,Y)} \rightarrow\ \mathtt{p(X,Z)}.\ \}$$

*et*

$$\mathcal{F}_{20} = \{\mathtt{p(a,b)}\}$$

*Ici l'application de l'oblivious chase ne se termine pas car après avoir appliqué une première fois la règle avec l'instance $\mathtt{p(a,b)}$ la règle $r_1$ crée une nouvelle instance $\mathtt{p(a,Z_0)}$ et donc un nouvel homomorphisme possible avec lequel la règle $r_1$ n'a jamais été appliquée. Seulement nous voyons que l'information n'apporte rien de nouveau car nous pouvons facilement remplacer $\mathtt{Z_0}$ par la constante $\mathtt{b}$. Il existe par contre un chase qui détecte cette redondance, c'est le skolem chase.*

Pour obtenir une réponse en un temps fini à l'exemple précédent il existe un second chase, un peu plus fort, appelé *Skolem-chase* (ou encore *frontier-chase*). Celui-ci conserve les propriétés du critère oblivious mais appliquées sur un programme dont la variable existentielle est remplacée par un terme fonctionnel. Cette transformation permet d'obtenir un programme ne contenant que des variables universelles. Nous utilisons ici $sk(r)$ pour représenter la règle $r$ skolémisée dont toutes les variables existentielles sont remplacées par terme fonctionnel $sk$ dont les arguments sont les variables de la frontière. La skolémisation sera détaillée dans la partie 2.

**Définition 1.3.12** (Critère Skolem)**.** *Soit $\mathcal{F}^i$ un ensemble de faits existentiels et $r$ une règle, l'ensemble de faits existentiels $\mathcal{F}^{i+1} = \alpha(\mathcal{F}^i, sk(r), \pi)$ est le résultat de l'application de la règle $sk(r)$ selon l'homomorphisme $\pi$ dans $\mathcal{F}^i$ seulement si une application de $sk(r)$ avec $\pi$ n'a pas déjà été effectuée.*

Reprenons l'exemple précédent avec l'utilisation du Skolem-chase.

**Exemple 21.** *Nous reprenons le programme $P_{20} = (\mathcal{F}_{20}, \mathcal{R}_{20})$.*

*Soit la règle $r_1$ skolémisée $sk(r_1) = (\mathtt{p(X,Y)} \rightarrow\ \mathtt{p(X,sk(X))}.)$, l'application du Skolem-chase sur ce programme se termine après deux applications de la règle $sk(r_1)$, en effet l'application $\alpha(\mathcal{F}_{20}, sk(r_1), \pi_1)$ avec $\pi_1(\mathtt{X}) = \mathtt{a}$ et $\pi_1(\mathtt{Y}) = \mathtt{b}$ donne l'ensemble $\mathcal{F}^1_{20} = \{\mathtt{p(a,b)}, \mathtt{p(a,sk(a))}\}$. Il est ensuite possible d'appliquer de nouveau $sk(r_1)$ avec $\pi_2(\mathtt{X}) = \mathtt{a}$ et $\pi_2(\mathtt{Y}) = \mathtt{sk(a)}$ ce qui nous donne $\mathcal{F}^2_{20} = \alpha(\mathcal{F}^1_{20}, sk(r_1), \pi_2) = \mathcal{F}^1_{20}$ et le chase se termine car il n'existe pas d'autre application possible de $sk(r_1)$ avec d'autres homomorphismes que $\pi_1$ et $\pi_2$.*

Là encore il existe des exemples avec lesquels le Skolem-chase ne se termine pas.

**Exemple 22.** *Soit le programme $P_{22} = (\mathcal{F}_{22}, \mathcal{R}_{22})$ avec :*

$$\mathcal{R}_{22} = \{\ r_1 : \mathtt{q(X)} \rightarrow\ \mathtt{p(X,Y), p(Y,Y), q(Y)}.\ \}$$

*et*

$$\mathcal{F}_{22} = \{\mathtt{q(a)}\}$$

*Soit la règle $r_1$ skolémisée $sk(r_1) = (\mathtt{q(X)} \rightarrow \mathtt{p(X, sk(X)), p(sk(X), sk(X)), q(sk(X))}.)$, l'application du Skolem-chase sur ce programme ne se termine pas car pour chaque application on ajoute une imbrication de symboles de Skolem en tête, ce qui donne une infinité d'homomorphismes. La première application crée un fait $\mathcal{F}^1_{22} = \{\mathtt{q(a)}, \mathtt{p(X, sk(a))}, \mathtt{p(sk(a), sk(a))}, \mathtt{q(sk(a))}\}$ avec l'application $\alpha(\mathcal{F}_{22}, sk(r_1), \pi_1)$ et $\pi_1(\mathtt{X}) = \mathtt{a}$, puis le chase continue avec l'application $\alpha(\mathcal{F}^1_{22}, sk(r_1), \pi_2)$ avec $\pi_2(\mathtt{X}) = \mathtt{sk(a)}$ et continue indéfiniment car il existera à chaque instance de $\mathtt{q}$ un nouvel homomorphisme possible.*

Il existe néanmoins un critère *restricted* permettant au programme précédent de se terminer. Ce *restricted-chase* détecte une redondance locale des variables existentielles fraîches créées.

**Définition 1.3.13** (Isomorphisme)**.** *Soit $\mathcal{F}$ et $\mathcal{F}'$ deux ensembles d'atomes. Un isomorphisme de $\mathcal{F}$ vers $\mathcal{F}'$ est une substitution bijective $\pi$ de $var(\mathcal{F})$ dans $var(\mathcal{F}')$ telle que $\pi(\mathcal{F}) = \mathcal{F}'$. $\pi$ est appelé renommage bijectif de variables.*

**Définition 1.3.14** (Redondance)**.** *Soit $\mathcal{F}$ et $\mathcal{F}'$ deux ensembles d'atomes. $\mathcal{F}$ est dit redondant par rapport à $\mathcal{F}'$ s'il existe un homomorphisme $\pi$ de $\mathcal{F}$ dans $\mathcal{F}'$ tel que $\pi(\mathcal{F}) \subsetneq \mathcal{F}'$.*

Remarque : une redondance est locale lorsqu'elle est issue d'une seule application de règle.

**Définition 1.3.15** (Restricted-redundant [38])**.** *Soit $r$ une règle et $\mathcal{F}$ un ensemble de faits existentiels. Un homomorphisme $\pi$ est restricted-redundant si $\exists \sigma'$ de $\sigma'(\mathtt{tête}(r))$ dans $\mathcal{F}$ tel que $\forall t \in fr(r)$, $\sigma(t) = \sigma'(t)$, avec $fr(r)$ les variables de la frontière.*

**Définition 1.3.16** (Critère restricted)**.** *Soit $\mathcal{F}^i$ un ensemble de faits existentiels et $r$ une règle, l'ensemble de faits existentiels $\mathcal{F}^{i+1} = \alpha(\mathcal{F}'^i, r, \pi)$ est le résultat de l'application de la règle $r$ selon l'homomorphisme $\pi$ dans $\mathcal{F}^i$, nous avons $\mathcal{F}'^i = \mathcal{F}^i$ si l'homomorphisme $\pi$ n'est pas restricted-redundant sinon $\mathcal{F}'^i = \mathcal{F}'^{i-1}$.*

**Exemple 23.** *Soit le programme $P_{22} = (\mathcal{F}_{22}, \mathcal{R}_{22})$ avec :*

$$\mathcal{R}_{22} = \left\{ \; r_1 : \mathtt{q(X)} \rightarrow \; \mathtt{p(X, Y), p(Y, Y), q(Y).} \; \right\}$$

*et*

$$\mathcal{F}_{22} = \{\mathtt{q(a)}\}$$

*Le restricted-chase appliqué sur ce programme se termine après une application de la règle $r_1$. L'application $\alpha(\mathcal{F}_{22}, r_1, \pi_1)$ avec $\pi_1(\mathtt{X}) = \mathtt{a}$ donne l'ensemble de faits existentiels $\mathcal{F}^1_{22} = \{\mathtt{q(a)}, \mathtt{p(a, Y_0)}, \mathtt{p(Y_0, Y_0)}, \mathtt{q(Y_0)}\}$. Une nouvelle application avec $\pi_2(\mathtt{X}) = \mathtt{Y_0}$ nous donne $\mathcal{F}^2_{22} = \{\mathtt{q(a)}, \mathtt{p(a, Y_0)}, \mathtt{p(Y_0, Y_0)}, \mathtt{q(Y_0)}, \mathtt{p(Y_0, Y_1)}, \mathtt{p(Y_1, Y_1)}, \mathtt{q(Y_1)}\}$, ce qui implique une redondance dans $\mathcal{F}^2_{22}$ car $\pi_2$ est un isomorphisme de $\{\mathtt{p(Y_1, Y_1)}, \mathtt{q(Y_1)}\}$ vers $\{\mathtt{p(Y_0, Y_0)}, \mathtt{q(Y_0)}\}$, l'application n'ajoute donc que l'atome $p(Y_0, Y_1)$ et le chase se termine. Il n'est plus possible d'ajouter de l'information en appliquant à nouveau la règle $r_1$*

Le restricted-chase ne détecte que la redondance locale ainsi un programme avec une redondance plus forte ne sera pas détectée et le restricted-chase ne se terminera pas comme dans l'exemple suivant.

**Exemple 24.** *Soit le programme $P_{24} = (\mathcal{F}_{24}, \mathcal{R}_{24})$ avec :*

$$\mathcal{R}_{24} = \left\{ \begin{array}{l} r_1 : \mathtt{s(X)} \rightarrow \; \mathtt{p(X, Y), p(X, Z), t(Z, Z).} \\ r_2 : \mathtt{p(X, Y)} \rightarrow \; \mathtt{q(Y).} \\ r_3 : \mathtt{q(X)} \rightarrow \; \mathtt{t(X, Y), q(Y).} \end{array} \right\}$$

*et*

$$\mathcal{F}_{24} = \{\mathtt{s(a)}\}$$

*Si nous développons le restricted-chase de ce programme nous avons tout d'abord l'application de* $r_1$ *qui nous donne l'ensemble* $\{\mathtt{s(a)}, \mathtt{p(a, Y_0)}, \mathtt{p(a, Z_0)}, \mathtt{t(Z_0, Z_0)}\}$ *avec* $\mathtt{Y_0}$ *et* $\mathtt{Z_0}$ *deux variables fraîches différentes. Nous appliquons ensuite la règle* $r_2$ *deux fois, une fois avec* $\mathtt{p(a, Y_0)}$ *et une seconde avec* $\mathtt{p(a, Z_0)}$ *ce qui nous donne* $\{\mathtt{s(a)}, \mathtt{p(a, Y_0)}, \mathtt{p(a, Z_0)}, \mathtt{t(Z_0, Z_0)}, \mathtt{q(Y_0)}, \mathtt{q(Z_0)}\}$*. Il n'y a ici toujours aucune redondance lors de l'application des règles. Nous appliquons enfin la règle* $r_3$ *avec* $\mathtt{q(Y_0)}$ *et* $\mathtt{q(Z_0)}$ *ce qui ajoute à l'ensemble* $\mathtt{t(Y_0, Y_1)}, \mathtt{q(Y_1)}, \mathtt{t(Z_0, Z_1)}, \mathtt{q(Z_1)}$ *puis l'algorithme recommence en appliquant infiniment les règles* $r_2$ *et* $r_3$ *car aucune redondance n'est détectée par le restricted-chase. Il est néanmoins possible de détecter une redondance lors de l'application de* $r_1$*. Notons que l'application de la règle* $r_1$ *crée une redondance avec la règle* $r_3$ *qui n'est pas locale et ces règles sont donc appliquées indéfiniment s'il n'y avait pas la règle* $r_2$*. La règle* $r_1$ *est appliquée en premier, ce qui crée deux nouvelles variables fraîches* $\mathtt{Y_0}$ *et* $\mathtt{Z_0}$*. La seule application de* $r_1$ *crée de la redondance en ajoutant* $\mathtt{p(a, Y_0)}, \mathtt{p(a, Z_0)}, \mathtt{t(Z_0, Z_0)}$ *avec* $\mathtt{p(a, Y_0)}$ *et* $\mathtt{p(a, Z_0)}$ *mais qui n'est pas détecté lors de l'application de la règle avec le restricted-chase.*

Pour pallier ce problème il existe un dernier chase appelé core-chase qui cette fois-ci prend en compte une redondance forte et non plus locale. Certains atomes d'un ensemble de faits existentiels peuvent être redondants mais plus sur une seule application de règle. Nous allons donc définir la notion de noyau faisant référence à la partie canonique d'un fait, cette notion provient de la théorie des graphes [48] mais est aussi applicable aux ensembles de faits existentiels.

**Définition 1.3.17** (Noyau d'un ensemble de faits existentiels [33])**.** *Soit* $\mathcal{F}$ *un ensemble de faits existentiels, nous appelons* $\mathcal{F}$ $noyau$ *si pour tout homomorphisme* $\pi$ *tel que* $\pi(\mathcal{F}) \subsetneq \mathcal{F}$*, nous avons* $\pi(\mathcal{F}) \not\models \mathcal{F}$

**Propriété 1.3.3.** *Soit* $Eq$ *l'ensemble des ensembles de faits existentiels équivalents à* $\mathcal{F}$*. Alors* $Eq$ *contient au moins un* $noyau$ *et les* $noyaux$ *de* $Eq$ *sont tous équivalents à un isomorphisme près. Nous pouvons choisir un représentant de ces* $noyaux$ *et le noter* $core(\mathcal{F})$*.*

Il peut exister plusieurs noyau pour un seul fait mais ils sont toujours isomorphes à un renommage de variables près, c'est pourquoi nous pouvons considérer qu'un noyau est unique.

**Définition 1.3.18** (Critère core)**.** *Soit* $\mathcal{F}^i$ *un fait et* $r$ *une règle, le fait* $\mathcal{F}^{i+1} = \alpha(\mathcal{F}^i, r, \pi)$ *est le résultat de l'application de la règle* $r$ *selon l'homomorphisme* $\pi$ *dans* $\mathcal{F}^i$ *seulement si* $core(\mathcal{F}^{i+1}) \neq \mathcal{F}^i$*. Autrement* $\alpha(\mathcal{F}^i, sk(r), \pi) = \mathcal{F}^i$*.*

**Exemple 25.** *Si nous reprenons l'exemple précédent dont la boucle n'est pas détectée par le restricted-chase.*
*Soit le programme* $P_{24} = (\mathcal{F}_{24}, \mathcal{R}_{24})$ *avec :*

$$\mathcal{R}_{24} = \left\{ \begin{array}{l} r_1 : \mathtt{s(X)} \rightarrow \mathtt{p(X, Y), p(X, Z), t(Z, Z).} \\ r_2 : \mathtt{p(X, Y)} \rightarrow \mathtt{q(Y).} \\ r_3 : \mathtt{q(X)} \rightarrow \mathtt{t(X, Y), q(Y).} \end{array} \right\}$$

*et*

$$\mathcal{F}_{24} = \{\mathtt{s(a)}\}$$

*Le core-chase s'applique directement après l'application de la règle* $r_1$*, soit* $\alpha(\mathcal{F}_{24}, r_1, \pi_1)$ *avec* $\pi_1(\mathtt{X}) = \mathtt{a}$ *donne l'ensemble* $\mathcal{F}^1_{24} = \{\mathtt{s(a)}, \mathtt{p(a, Y_0)}, \mathtt{p(a, Z_0)}, \mathtt{t(Z_0, Z_0)}\}$ *pour lequel nous pouvons calculer le core qui retire* $\mathtt{p(a, Y_0)}$ *car redondant avec* $\mathtt{p(a, Z_0)}$*, ce qui donne* $core(\mathcal{F}^1_{24}) = \{\mathtt{s(a)}, \mathtt{p(a, Z_0)}, \mathtt{t(Z_0, Z_0)}\}$ *puis se termine car il n'existe plus d'application possible telle que le core ajoute de l'information.*

Il existe bien sûr des cas pour lesquels le core-chase ne termine pas, notamment lorsqu'il n'existe pas de redondance comme dans l'exemple suivant.

**Exemple 26.** *Soit le programme* $P_{26} = (\mathcal{F}_{26}, \mathcal{R}_{26})$ *avec :*

$$\mathcal{R}_{26} = \{\ r_1 : \mathtt{h(X)} \rightarrow\ \mathtt{p(X,Y)}, \mathtt{h(Y)}.\ \}$$

*et*

$$\mathcal{F}_{26} = \{\mathtt{h(a)}\}$$

*Le core-chase ne se termine pas car il n'existe pas de redondance dans l'application de la règle* $r_1$. *Le modèle ainsi obtenu est infini* $\{\mathtt{h(a)}, \mathtt{p(a,Y_0)}, \mathtt{h(Y_0)}, \mathtt{p(Y_0,Y_1)}, \ldots\}$

Nous avons présenté les chase les plus utilisés dans le domaine des règles existentielles. En commençant par le critère le plus faible, oblivious, jusqu'au critère core la forme la plus forte. Notons que le résultat de l'application d'un chase est appelé modèle universel. Nous pouvons ainsi dire qu'un critère $\mathcal{C}r$ est plus fort qu'un critère $\mathcal{C}r'$ lorsque $\mathcal{C}r$ se termine plus souvent que $\mathcal{C}r'$ et écrire $\mathcal{C}r \geq \mathcal{C}r'$. Nous pouvons ajouter que $\mathcal{C}r$ est strictement plus fort que $\mathcal{C}r'$ lorsque $\mathcal{C}r \geq \mathcal{C}r'$ et que $\mathcal{C}r' \ngeq \mathcal{C}r$. Si nous définissons $\mathcal{C}r - finite$ l'ensemble des programmes se finissant en un temps fini avec un critère de chase $\mathcal{C}r$ alors nous pouvons conclure sur la hiérarchie suivante $core > restricted > skolem > oblivious$ et l'inclusion suivante $oblivious - finite \subset skolem - finite \subset restricted - finite \subset core - finite$ [12].

### 1.3.4 Limitations

Le langage des règles existentielles est un langage monotone, contrairement à ASP où une information déduite peut être remise en question à cause de la négation par défaut : tout atome déduit avec les règles existentielles fera partie du modèle final. De fait, il est impossible de modéliser l'exception ou d'obtenir plusieurs modèles avec les règles existentielles. Or nous souhaitons ici avoir un langage polyvalent permettant de traiter directement à la fois l'exception et les variables existentielles en tête de règle. Les règles existentielles ne permettant pas l'utilisation de la négation par défaut, il n'est pas possible comme en ASP d'avoir plusieurs solutions à un problème, les règles existentielles n'ayant qu'un unique modèle. Nous proposons alors une extension à la fois des règles existentielles et d'ASP pour couvrir l'ensemble des programmes de ces deux langages. Les règles existentielles étant une extension des programmes logiques définis avec l'ajout de variables existentielles en tête et ASP étant une extension des programmes logiques définis avec l'ajout d'une négation par défaut, nous combinons ces deux aspects pour obtenir un langage polyvalent. Mais avant de discuter de cette extension, nous souhaitons aborder le thème des graphes de dépendance permettant de représenter les dépendances entre règles ou prédicats dans un programme. Ces graphes de dépendance nous serviront par la suite pour l'étude de la décidabilité et l'interrogation.

## 1.4 Graphes de dépendance

Afin de représenter graphiquement les connaissances d'un programme à base de règles, il est commun d'utiliser une représentation sous forme de graphe. Il existe différents graphes permettant de représenter les dépendances entre les éléments d'un programme et ainsi analyser le comportement d'un programme. Les graphes nous donnent la possibilité de détecter facilement si une règle est susceptible d'en déclencher une autre et donc est susceptible de déduire une information à partir des faits initiaux. Il existe pour cela différents graphes, ici nous nous intéressons principalement à deux graphes utilisés à la fois en règles existentielles et ASP. Ce sont les graphes de dépendance des règles et les

graphes de dépendance des prédicats. Comme leur nom l'indique ils permettent de représenter les dépendances entre les règles et entre les prédicats pour un programme. Ces graphes nous permettent entre autre de déterminer si certains programmes vont se terminer en un temps fini ou encore dans le cas d'ASP si un programme possède au moins un modèle, ces thèmes seront abordés respectivement dans les parties 3 décidabilité et 4 interrogation.

### 1.4.1 Les graphes

Nous allons définir dans cette partie ce qu'est un graphe puis les deux graphes de dépendance les plus utilisés en les généralisant pour qu'ils s'appliquent à la fois aux règles existentielles et à ASP.

**Définition 1.4.1** (Graphe orienté)**.** *Un* graphe orienté $\mathcal{G}$ *est un couple (V,A) avec V un ensemble de sommets et A un ensemble d'arcs. Un* arc *est un couple de sommets* $(V_1, V_2)$ *orienté de* $V_1$ *vers* $V_2$*.*

**Définition 1.4.2** (Chemin)**.** *Un* chemin *dans un graphe orienté est une suite finie d'arcs* $(A_1, \ldots, A_n)$ *avec* $A_i = (S_i, S_{i+1})$, $\forall i \leq n$*.*

**Définition 1.4.3** (Circuit)**.** *Un* circuit *(ou* cycle*) est un chemin* $(A_1, \ldots, A_n)$ *avec* $A_1 = (S_1, S'_1)$ *et* $A_n = (S_n, S'_n)$ *tel que* $S'_n = S_1$*.*

### 1.4.2 Graphe de dépendance des règles

Le *graphe de dépendance des règles* d'un programme noté GRD (graph of rule dependencies) est un graphe orienté permettant de visualiser les dépendances existantes entre les règles d'un programme. Il sert notamment à déterminer s'il existe une terminaison à l'application de règles dans un programme en analysant s'il existe un déclenchement cyclique des règles du programme. Il n'est pas possible de déterminer tous les programmes se terminant mais nous pouvons isoler certaines propriétés permettant de déterminer si le calcul de modèle de certains programmes ASP ou de règles existentielles se termine à coup sûr et que nous décrirons par la suite. Avant de s'intéresser au graphe nous définissons ce qu'est la dépendance de règles dans le cas général.

**Définition 1.4.4** (Dépendance de règles)**.** *Soit* $r_1$ *et* $r_2$ *deux règles d'un programme. On dit que* $r_2$ *dépend de* $r_1$ *s'il existe un ensemble d'atomes* $A$ *tel que* $r_1$ *est applicable par* $A$ *suivant un homomorphisme* $\pi$ *et* $r_2$ *est applicable par* $\alpha(A, r_1, \pi)$ *suivant un nouvel homomorphisme* $\pi'$*.*

Pour les règles existentielles nous devons prendre en considération les variables existentielles pour minimiser le nombre de dépendances. Avec la définition précédente une règle $r_1$ peut être considérée dépendante d'une règle $r_2$ avec des variables existentielles en tête alors que celle-ci ne permettra jamais l'application de $r_2$, ceci vient du fait que l'application d'une règle avec des variables existentielles en tête crée des instances uniques et il est donc parfois impossible qu'une variable apparaissant plusieurs fois dans le corps d'une règle soit substituée par ce même individu provenant de l'application. Nous utilisons les unificateurs par pièce [15] pour vérifier si une règle est bien dépendante d'une autre règle. Les unificateurs par pièce sont initialement utilisés pour la réécriture de requête, ils permettent une unification entre la tête d'une règle $r_1$ et le corps d'une règle $r_2$ et de remplacer le corps de $r_2$ par le corps de la règle $r_1$ unifiée avec $r_2$. Dans notre cas, les unificateurs vont permettre d'ajouter un arc ou non dans un graphe de dépendance de règle.

**Définition 1.4.5** (Unificateurs par pièce [15])**.** *Soit* $r_1 : \mathtt{B}_1 \rightarrow \mathtt{H}_1$*. et* $r_2 : \mathtt{B}_2 \rightarrow \mathtt{H}_2$*. deux règles avec* $H_1, H_2, B_1, B_2$ *des ensembles d'atomes. Un unificateur par pièce de* $B_2$ *avec* $H_1$ *est une substitution* $\mu$ *de* $var(B'_2) \cup var(H'_1)$*, où* $B'_2 \subseteq B_2$ *et* $H'_1 \subseteq H_1$*, telle que :*

- $\mu(B'_2) = \mu(H'_1)$ ;

- *les variables existentielles dans $H'_1$ ne sont pas unifiées à des variables apparaissant à la fois dans $B'_2$ et $B_2 \backslash B'_2$.*

Plus simplement, si une variable $X$ apparaissant dans $B'_2$ est unifiée à une variable existentielle $Y$ apparaissant dans $H'_1$, alors tous les atomes dans lesquels apparaît $X$ doivent aussi appartenir à $B'_2$.

**Exemple 27.** *Soit les règles existentielles :*

$$\left\{ \begin{array}{l} r_1 : \ \mathtt{t(X,Y)} \rightarrow \ \mathtt{p(Z,Y)}, \mathtt{q(Y)}. \\ r_2 : \ \mathtt{p(U,V)}, \mathtt{q(U)} \rightarrow \ \mathtt{t(V,W)}. \end{array} \right\}$$

*Le seul unificateur par pièce de $\mathtt{corps}(r_2)$ avec $\mathtt{tête}(r_1)$ unifie* U *et* Y*. L'application de la règle $r_1$ ne peut donc pas provoquer l'application de la règle $r_2$. Si nous avons l'atome $\mathtt{t(a,b)}$ nous pouvons appliquer $r_1$ et nous obtenons les atomes $\mathtt{p(Z_0,b)}, \mathtt{q(b)}$ avec $\mathtt{Z_0}$ une variable fraîche. Malheureusement cette application ne peut pas déclencher une application de $r_2$ car $\mathtt{Z_0}$ est une variable fraîche (donc unique), il est donc impossible d'avoir une substitution de la variable* U *de la règle $r_2$ correspondant à la variable fraîche $\mathtt{Z_0}$ étant donné que l'application de la règle $r_1$ nous donne $\mathtt{p(Z_0,b)}, \mathtt{q(b)}$.*

**Propriété 1.4.1** (Dépendance par unificateurs [14])**.** *Soit $r_1$ et $r_2$ deux règles d'un programme. $r_2$ dépend de $r_1$ si et seulement s'il existe un unificateur par pièce de $\mathtt{corps}(r_2)$ avec $\mathtt{tête}(r_1)$.*

**Définition 1.4.6** (Graphe de dépendance des règles)**.** *Un graphe $\mathcal{G}_\mathcal{R}$ de dépendance des règles d'un programme est un graphe dont les sommets sont étiquetés par les règles du programme et dont les arcs représentent les dépendances entre les règles. Soit $r_1$ et $r_2$ deux règles d'un programme, il existe un arc allant de $r_1$ à $r_2$ si $r_2$ dépend de $r_1$.*

**Exemple 28.** *Soit l'ensemble de règles ASP (non-safe) suivant :*

$$\mathcal{R} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X,Y)} \leftarrow \ \mathtt{h(X)}. \\ r_2 : \mathtt{q(X,Y)} \leftarrow \ \mathtt{p(X,Y)}, \mathtt{not\ h(Y)}. \\ r_3 : \mathtt{h(X)} \leftarrow \ \mathtt{s(X)}. \end{array} \right\}$$

*Remarque : nous utilisons ici la notation ASP à cause de la négation par défaut. Ce programme n'est pas correct en ASP car non-safe, cet exemple sert à illustrer un moyen de représenter un graphe avec à la fois des variables existentielles et de la négation par défaut. Ce type de programme sera autorisé dans le langage défini dans la prochaine partie.*
*Ici nous avons la règle $r_1$ qui peut déclencher la règle $r_2$ et la règle $r_3$ qui peut déclencher $r_1$. Par contre la règle $r_2$ ne peut rien déclencher. Le graphe correspondant est :*

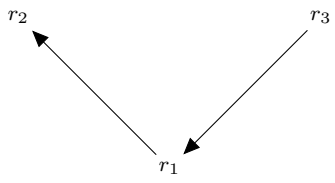



FIGURE 1.1 – Graphe de dépendance des règles

### 1.4.3 Graphe de dépendance des symboles de prédicat

Le *graphe de dépendance des symboles de prédicat* d'un programme est un graphe orienté permettant de visualiser les dépendances existantes entre les prédicats d'un programme. Il sert comme le graphe de dépendance des règles à déterminer s'il existe une terminaison à l'application de règles dans un

programme mais cette fois-ci en analysant s'il existe une dépendance cyclique entre les prédicats du programme. Comme pour le graphe de dépendance des règles la détection de la terminaison d'un programme n'est possible que sur certaines classes de programmes que nous détaillerons dans la partie 3 décidabilité. Alors que le graphe de dépendance des règles va permettre de détecter si une règle peut être appliquée par une autre le graphe de dépendance des symboles de prédicat va permettre de détecter comment se transmettent les variables du corps d'une règle vers la tête. Avant de s'intéresser au graphe nous définissons ce qu'est la dépendance de prédicats dans le cas général.

**Définition 1.4.7** (Dépendance d'un symbole de prédicat)**.** *Soit $r$ une règle, un symbole de prédicat* p *dépend d'un symbole de prédicat* q *si et seulement si* p *apparaît dans un atome* $\mathtt{a} \in \mathit{tête}(r)$ *et* q *apparaît dans un atome* $\mathtt{b} \in \mathit{corps}(r)$.

**Définition 1.4.8** (Graphe de dépendance des symboles de prédicat)**.** *Un* graphe de dépendance des symboles de prédicat *$\mathcal{G}$ est un graphe orienté avec ses sommets étiquetés par les symboles de prédicat du programme. Il existe un arc allant d'un prédicat* p *à un prédicat* q *si* q *dépend de* p.

En plus des définitions précédentes nous devons prendre en compte des dépendances spéciales pour le cas de la négation par défaut en ASP et des variables existentielles pour les règles existentielles. Ces nouvelles dépendances vont ajouter de l'expressivité au graphe précédent.

**Définition 1.4.9** (Dépendance positive et négative d'un symbole de prédicat)**.** *Soit $r$ une règle, un symbole de prédicat* p dépend positivement *(resp.* négativement*) d'un symbole de prédicat* q *si* p *apparaît dans un atome* $\mathtt{a} \in \mathit{tête}(r)$ *et* q *apparaît dans un atome* $\mathtt{b} \in \mathit{corps}^+(r)$ *(resp* $\mathit{corps}^-(r)$*).*

**Définition 1.4.10** (Dépendance existentielle d'un symbole de prédicat)**.** *Soit $r$ une règle et* Y *une variable existentielle de $r$, un symbole de prédicat* p dépend existentiellement *d'un symbole de prédicat* q *si* p *apparaît dans un atome* $\mathtt{a} \in \mathit{tête}(r)$ *et* q *apparaît dans un atome* $\mathtt{b} \in \mathit{corps}^+(r)$ *et* $\mathtt{Y} \in \mathit{var}(a)$.

**Définition 1.4.11** (Graphe de dépendance des symboles de prédicat (complet))**.** *Un* graphe de dépendance des symboles de prédicat *$\mathcal{G}$ est un graphe orienté avec ses sommets étiquetés par les symboles de prédicat du programme. Il existe un arc positif (resp. négatif, existentiel) allant d'un symbole de prédicat $p$ à un symbole de prédicat $q$ si $q$ dépend positivement (resp. négativement, existentiellement) de $p$.*

**Exemple 29.** *Le graphe 1.2 :*

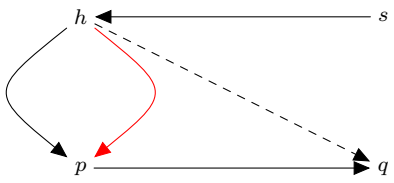



FIGURE 1.2 – Graphe de dépendance des symboles de prédicat

*dont les arcs existentiels sont représentés en rouge et les arcs négatifs en pointillés, est pour l'ensemble de règles suivant :*

$$\mathcal{R} = \left\{ \begin{array}{l} r_1 = \mathtt{p(X,Y)} \leftarrow \mathtt{h(X)}. \\ r_2 = \mathtt{q(X,Y)} \leftarrow \mathtt{p(X,Y)}, \mathtt{not\ h(Y)}. \\ r_3 = \mathtt{h(X)} \leftarrow \mathtt{s(X)}. \end{array} \right\}$$

*Ici nous avons le prédicat $p$ qui dépend de $h$ positivement et existentiellement, $q$ dépend de $p$ positivement et de $h$ négativement et enfin $h$ dépend de $r$ positivement.*

# 2

# Programme non monotone existentiel

Nous avons présenté dans l'état de l'art deux langages permettant la représentation de connaissance qui sont les règles existentielles et ASP. Malheureusement aucun de ces langages ne permet de représenter efficacement une ontologie avec exception, ASP n'autorisant pas l'utilisation de variables existentielles et les règles existentielles ne gérant pas la négation par défaut comme le permet ASP. Il est donc impossible de représenter une règle de ce type :

Si X est une personne et qu'il n'est pas spécifié que X est un enseignant chercheur qui donne un cours Y alors X est un doctorant et il existe un directeur de thèse D qui dirige X

Pour traiter des programmes issus de la fusion d'ontologies nous avons montré précédemment que les langages que forment les règles existentielles et ASP ne sont pas suffisants, nous avons besoin d'un langage proposant à la fois l'utilisation de variables existentielles et de négation par défaut. Nous proposons donc un formalisme permettant de représenter à la fois des programmes issus des règles existentielles et d'ASP. Celui-ci rend possible la représentation de programmes à base de règles existentielles non-monotones, en relaxant la propriété safe d'ASP (interdisant l'utilisation de variables existentielles) et en autorisant l'utilisation de négation par défaut pour les règles existentielles. L'objectif étant de combiner ces deux aspects dans un même langage tout en gardant une sémantique cohérente et permettant de représenter l'ensemble des programmes des deux langages. Nous définissons dans une première partie la syntaxe et la sémantique de ce formalisme, puis proposons une traduction vers un ASP classique permettant la représentation des modèles de ce formalisme en ASP. Nous discutons enfin en dernière partie une comparaison des chase des règles existentielles avec l'opérateur de conséquence d'ASP.

## 2.1 Syntaxe et Sémantique

Dans cette partie nous proposons une syntaxe et une sémantique permettant la représentation de programmes à base de règles non-monotones existentielles sans disjonction. Nous avons vu dans la partie 1.2 qu'ASP ne permet pas l'utilisation d'un programme si celui-ci n'est pas safe et que les règles existentielles n'utilisent pas de corps négatif. L'utilisation de variables existentielles repose sur le relâchement du caractère safe des règles pour accepter l'utilisation de variables dans la tête et dans le corps négatif d'une règle sans qu'elles n'apparaissent dans le corps positif de celle-ci. De plus nous souhaitons autoriser un ensemble d'atomes en tête de règle ainsi qu'un ensemble d'atomes pour

chaque négation par défaut permettant de conserver le lien entre les variables existentielles d'une règle, nous verrons par la suite que cela permet de représenter plus facilement certains programmes et permet de faire plus facilement le lien entre un corps négatif et une tête multi-atomique. La syntaxe des règles ASP de la forme :

$$\mathtt{h} \leftarrow \ \mathtt{b_1}, \ldots, \mathtt{b_n}, \mathtt{not}\ \mathtt{n_1}, \ldots, \mathtt{not}\ \mathtt{n_s}.$$

avec $\mathtt{h}, \mathtt{b_1}, \ldots, \mathtt{b_n}, \mathtt{n_1}, \ldots, \mathtt{n_s}$ des atomes, n'autorise pas l'utilisation de variables existentielles, l'ensemble des variables de la règle sont universellement quantifiées. Nous souhaitons étendre ces règles par des règles que nous appelons *règles NME* pour règles non monotones existentielles autorisant entre autre l'utilisation des variables existentielles. L'écriture d'un programme sous la forme $P = (\mathcal{F}, \mathcal{R})$ est préférée pour traiter par la suite les problèmes issus des règles existentielles et de l'interrogation faisant appel aux graphes de dépendance dans lesquels les faits ne sont pas considérés.

**Définition 2.1.1** (règles NME)**.** *Une règle NME est une règle de la forme :*

$$\mathtt{h_1}, \ldots, \mathtt{h_v} \leftarrow \ \mathtt{b_1}, \ldots, \mathtt{b_m}, \mathtt{not}\ (\mathtt{n}^1_1, \ldots, \mathtt{n}^1_{\mathtt{u}_1}), \ldots, \mathtt{not}\ (\mathtt{n}^s_1, \ldots, \mathtt{n}^s_{\mathtt{u}_s}).$$

*avec* $m + s, v, u_1, ..., u_s \geq 1$*, et* $\mathtt{h_1}, \ldots, \mathtt{h_v}, \mathtt{b_1}, \ldots, \mathtt{b_m}, \mathtt{n}^1_1, \ldots, \mathtt{n}^1_{\mathtt{u}_1}, \ldots, \mathtt{n}^s_1, \ldots, \mathtt{n}^s_{\mathtt{u}_s} \in \mathcal{A}$*.*

*Un* programme NME *est un couple* $P = (\mathcal{F}, \mathcal{R})$ *avec* $\mathcal{F}$ *un ensemble d'atomes ground appelé* ensemble de faits *et* $\mathcal{R}$ *un ensemble de règles NME.*

Remarque : l'ensemble $\{\mathtt{h_1}, \ldots, \mathtt{h_v}\}$ représentant la tête de la règle est une conjonction d'atomes et non pas une disjonction comme dans la définition d'un ASP disjonctif. Pour notre langage, la disjonction d'atomes n'est pas autorisée. De même dans une règle NME, plusieurs atomes sont autorisés dans un corps négatif et sont une conjonction d'atomes.
Nous avons choisi d'utiliser une écriture des règles correspondant à ASP, avec l'implication vers la gauche $\leftarrow$, pour des questions de simplicité de compréhension par rapport au corps négatif présent uniquement en ASP.

Nous considérons alors la notation suivante :

- $\mathtt{tête}(r) = \{\mathtt{h_1}, \ldots, \mathtt{h_v}\}$, l'ensemble des atomes appartenant à la tête de $r$,
- $\mathtt{corps}^+(r) = \{\mathtt{b_1}, \ldots, \mathtt{b_m}\}$ l'ensemble des atomes appartenant au corps positif de $r$,
- $\mathtt{corps}^-(r) = \{\{\mathtt{n}^1_1, \ldots, \mathtt{n}^1_{\mathtt{u}_1}\}, \ldots, \{\mathtt{n}^s_1, \ldots, \mathtt{n}^s_{\mathtt{u}_s}\}\}$ l'ensemble des atomes appartenant aux corps négatifs de $r$,
- $\mathcal{V}(r)$ l'ensemble des variables de $r$,
- $\mathcal{V}_{H\exists}(r)$ l'ensemble des variables qui apparaissent dans $\mathtt{h_1}, \ldots, \mathtt{h_v}$ mais pas dans $\mathtt{b_1}, \ldots, \mathtt{b_m}$ (l'ensemble des variables existentielles de la tête de $r$),
- $\mathcal{V}_{\exists}(r)(\mathtt{n}^i_1, \ldots, \mathtt{n}^i_{\mathtt{u}_i})$ l'ensemble des variables qui apparaissent dans $\mathtt{n}^i_1, \ldots, \mathtt{n}^i_{\mathtt{u}_i}$ mais pas dans $\mathtt{b_1}, \ldots, \mathtt{b_m}$, $1 \leq i \leq s$ (l'ensemble des variables existentielles du corps négatif $\mathtt{n}^i_1, \ldots, \mathtt{n}^i_{\mathtt{u}_i}$),
- $\mathcal{V}_{N\exists}(r) = \bigcup_{1 \leq i \leq s} \mathcal{V}_{\exists}(r)(\mathtt{n}^i_1, \ldots, \mathtt{n}^i_{\mathtt{u}_i})$ l'ensemble des variables existentielles des corps négatifs,
- $\overline{\mathcal{V}_{N\exists}}(r) = \mathcal{V}(r) \setminus \mathcal{V}_{N\exists}(r)$ l'ensemble des variables universelles des corps négatifs, appelé *frontière négative*,
- $\mathcal{V}_{\exists}(r) = \mathcal{V}_{H\exists}(r) \bigcup \mathcal{V}_{N\exists}(r)$ l'ensemble des variables existentielles de $r$, tête et corps négatifs confondus,

- $\mathcal{V}_{H\forall}(r)$ l'ensemble des variables qui apparaissent au moins dans $\mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{v}$ et dans $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}$ (l'ensemble des variables universelles de la tête de $r$, appelé *frontière*).

- $\mathcal{V}_\forall(r)(\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}})$ l'ensemble des variables qui apparaissent au moins dans $\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}}$ et dans $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}$ (l'ensemble des variables universelles du corps négatif $\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}}$).

Une des contraintes imposée est que les ensembles $\mathcal{V}_\exists(r)(\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}})$ pour tout $1 \leq i \leq s$ doivent être disjoints ainsi que les ensembles $\mathcal{V}_{H\exists}(r)$ et $\mathcal{V}_{N\exists}(r)$ (Si une variable apparaît dans plusieurs des $\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}}$ ou si elle apparaît dans $\mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{v}$ et dans un des $\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}}$, $1 \leq i \leq s$, alors elle doit apparaître dans $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}$ et est universellement quantifiée).

Cette contrainte est nécessaire pour empêcher l'utilisation de règle de ce type :

$$\exists \mathtt{Z}(\mathtt{p}(\mathtt{Z})) \leftarrow \ \mathtt{not}\ \exists \mathtt{Z}(\mathtt{q}(\mathtt{Z})).$$

Ici il y a deux variables quantifiées existentiellement dont la portée est locale, chacune des variables correspond donc à un élément différent. Il est donc préférable d'utiliser deux noms de variable différents pour éviter la confusion, surtout que nous ne représentons pas les quantificateurs dans la définition des règles NME.

Pour toute règle $r_1$ et $r_2$ d'un ensemble de règles $\mathcal{R}$, $\mathcal{V}_\exists(r_1)$ et $\mathcal{V}_\exists(r_2)$ doivent être disjointes (tous les noms des variables existentielles du programme sont différents).

Concernant les variables intervenant dans la règle, elles peuvent être quantifiées universellement ou existentiellement. Les quantificateurs ne sont pas exprimés explicitement dans la règle mais ils dépendent de leur position dans la règle : les variables apparaissant dans $\mathtt{corps}^+(r)$ sont quantifiées universellement tandis que celles n'apparaissant pas dans $\mathtt{corps}^+(r)$ le sont existentiellement.

Notons que les variables existentielles, dans la tête ou dans chaque partie du corps négatif, sont quantifiées localement.

Pour chaque programme $P$, nous considérons que son langage $\mathcal{L}_P = (\mathcal{CS}, \mathcal{FS}, \mathcal{PS})$ est exactement composé des symboles de constante, symboles de fonction et symboles de prédicat apparaissant dans $P$.

**Exemple 30.** *Soit* $P_{30} = (\mathcal{F}_{30}, \mathcal{R}_{30})$ *un programme NME de langage* $\mathcal{L}_{30} = (\{\mathtt{toto}\}, \emptyset, \{\mathtt{p}, \mathtt{dT}, \mathtt{d}, \mathtt{eC}, \mathtt{dC}\})$ *avec* $ar(\mathtt{p}) = ar(\mathtt{d}) = ar(\mathtt{eC}) = 1$ *et* $ar(\mathtt{dT}) = ar(\mathtt{dC}) = 2$. `p` *pour* `personne`, `dT` *pour* `dirigeThese`, `d` *pour* `directeur`, `eC` *pour* `enseignantChercheur` *et* `dC` *pour* `donneCours`. *Nous avons* $P_{30}$ *composé de l'ensemble de règles NME :*

$$\mathcal{R}_{30} = \{\ r_1 :\ \mathtt{dT}(\mathtt{X}, \mathtt{D}), \mathtt{d}(\mathtt{D}) \leftarrow\ \mathtt{p}(\mathtt{X}), \mathtt{not}(\mathtt{eC}(\mathtt{X}), \mathtt{dC}(\mathtt{X}, \mathtt{Y}))\ .\ \}$$

*et de l'ensemble de faits :*

$$\mathcal{F}_{30} = \ \{\mathtt{p}(\mathtt{toto}), \mathtt{eC}(\mathtt{toto})\}$$

*La règle* $r_1$ *signifie que pour une personne* `X` *il existe un directeur* `D` *qui dirige la thèse de* `X` *un doctorant de* `D`*, tant qu'il n'est pas signifié que* `X` *est un enseignant chercheur et qu'il existe un cours* `Y` *donné par* `X`.

*Nous avons*

$$\mathcal{V}_{H\forall}(r) = \{\mathtt{X}\},\ \mathcal{V}_{H\exists}(r) = \{\mathtt{D}\},\ \mathcal{V}_\exists(r)(\mathtt{eC}(\mathtt{X}), \mathtt{dC}(\mathtt{X}, \mathtt{Y})) = \{\mathtt{Y}\},\ \overline{\mathcal{V}_{N\exists}}(r) = \{\mathtt{X}, \mathtt{D}\}.$$

**Proposition 2.1.1.** *Tout programme ASP (classique du premier ordre) et tout ensemble de règles existentielles sont des programmes NME.*

**Preuve 2.1.1.** *C'est une conséquence directe de la définition 2.1.1.*

Le langage proposé permet ainsi de représenter syntaxiquement l'ensemble des programmes ASP et des règles existentielles, et va même plus loin en proposant la gestion d'ensembles d'atomes dans les parties du corps négatif permettant de simplifier certains programmes. Par exemple, les règles suivantes :

$$\left\{ \begin{array}{l} r_1 : \texttt{dT(X,D)}, \texttt{d(D)} \leftarrow \texttt{p(X)}, \texttt{not(eC(X)}, \texttt{dC(X,Y))}. \\ r_2 : \texttt{eC(X)}, \texttt{dC(X,Y)} \leftarrow \texttt{maitreConference(X,Z)}, \texttt{universite(Z)}. \end{array} \right\}$$

nécessiteront plus de deux règles pour être représentées en ASP classique, et il est aussi plus simple de voir le lien entre $r_1$ et $r_2$ étant donné que le corps négatif de $r_1$ est identique à la tête de $r_2$.

### 2.1.1 Skolémisation et grounding partiel

Nous avons vu dans la section 1.3.3 qu'il existe plusieurs algorithmes de point fixe en marche avant, appelés chase, permettant d'appliquer des règles avec variables existentielles en traitant la redondance. Sur ces différents chase nous avons parlé du skolem-chase qui repose sur une skolémisation des variables existentielles transformant celles-ci en termes fonctionnels de Skolem (les symboles de fonction de Skolem sont équivalents à des symboles de fonction classiques mais n'apparaissant pas autre part dans le programme). Les symboles de fonctions étant déjà utilisés en ASP nous savons qu'il est possible d'utiliser une skolémisation de variables existentielles, une fois les variables skolémisées elles deviennent de simples termes fonctionnels. La sémantique d'un programme NME utilise donc la skolémisation des variables existentielles apparaissant dans les têtes de règle.

Nous définissons la skolémisation comme suit.

**Définition 2.1.2** (Symbole de Skolem)**.** *Soit $r$ une règle NME, $n$ la cardinalité de $\mathcal{V}_{H\forall}(r)$ et $\texttt{Y} \in \mathcal{V}_{H\exists}(r)$ une variable existentielle de $r$ alors $\texttt{sk}_\texttt{r}^\texttt{Y}$ est un symbole de fonction de Skolem d'arité $n$ (si $n = 0$ alors $\texttt{sk}_\texttt{r}^\texttt{Y}$ est un symbole de constante).*

**Exemple 31** (Suite de l'exemple 30)**.** *Le symbole $\texttt{sk}_{\texttt{r}_1}^\texttt{D}$ est un symbole de fonction de Skolem d'arité 1 pour la variable existentielle $\texttt{D}$ de la tête de la règle $r_1$.*

**Définition 2.1.3** (Règle et programme skolémisé)**.** *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme NME de langage $\mathcal{L}_P$. Soit $s$ une séquence ordonnée de l'ensemble de variables $\mathcal{V}_{H\forall}(r)$ d'une règle NME $r \in \mathcal{R}$. sk(r) représente une* règle skolémisée *obtenue à partir de $r$ comme suit : chaque variable existentielle $\texttt{Y} \in \mathcal{V}_{H\exists}(r)$ est substituée par le terme $\texttt{sk}_\texttt{r}^\texttt{Y}(\texttt{s})$ avec $\texttt{sk}_\texttt{r}^\texttt{Y}$ le symbole de fonction (resp. la constante) de Skolem associé à $\texttt{Y}$ et $n = ar(\texttt{sk}_\texttt{r}^\texttt{Y})$ la taille de $s$ (zéro si $\mathcal{V}_{H\forall}(r) = \emptyset$). Le* programme skolémisé *sk(P) d'un programme NME $P$ est défini par $sk(P) = (\mathcal{F}, sk(\mathcal{R}))$ avec $sk(\mathcal{R}) = \{sk(r) | r \in \mathcal{R}\}$.*

**Exemple 32** (Suite de l'exemple 30)**.** *Le résultat de la skolémisation de la règle $r_1$ est la règle skolémisée :*

$$sk(r_1) : \texttt{dT(X,sk}_{\texttt{r}_1}^\texttt{D}\texttt{(X))}, \texttt{d(sk}_{\texttt{r}_1}^\texttt{D}\texttt{(X))} \leftarrow \texttt{p(X)}, \texttt{not(eC(X)}, \texttt{dC(X,Y))}.$$

*Par conséquent*

$$sk(P_{30}) = (\mathcal{F}_{30}, \{sk(r_1)\})$$

*et*

$$\mathcal{L}_{sk(P_{30})} = (\{\texttt{toto}\}, \{\texttt{sk}_{\texttt{r}_1}^\texttt{D}\}, \{\texttt{p}, \texttt{phdS}, \texttt{d}, \texttt{eC}, \texttt{dC}\})$$

Les règles skolémisées ne sont pas encore safe car des variables existentielles subsistent dans les corps négatifs. Nous souhaitons instancier le programme, mais il n'existe pas de grounding pour les variables existentielles dans le corps négatif. Nous proposons donc un *grounding partiel* d'une règle restreint aux variables universelles de celle-ci, les variables existentielles restant non instanciées. Si

nous prenons une règle non-ground ($\mathtt{p(X)} \leftarrow \mathtt{q(X)},\ \mathtt{not\ r(X,Z)}.$), elle pourrait être déclenchée par une constante $\mathtt{a}$ si $\mathtt{q(a)}$ est vrai et, pour tout $\mathtt{z}, \mathtt{r(a,z)}$ n'est pas vrai. Seule la substitution de $\mathtt{X}$ est importante car s'il existe une substitution possible pour $\mathtt{X}$ alors la règle est bloquée quelque soit la substitution de $\mathtt{Z}$. Supposons deux constantes $\mathtt{a}$ et $\mathtt{b}$. Alors ($\mathtt{p(a)} \leftarrow \mathtt{q(a)},\ \mathtt{not\ r(a,a)}.$) et ($\mathtt{p(a)} \leftarrow \mathtt{q(a)},\ \mathtt{not\ r(a,b)}.$) ne sont pas équivalentes à la règle non-ground pour $\mathtt{X} = \mathtt{a}$ car la première instance pourrait être déclenchée si $\mathtt{r(a,b)}$ est vrai (mais pas $\mathtt{r(a,a)}$) et la seconde pourrait être déclenchée si $\mathtt{r(a,a)}$ est vrai (mais pas $\mathtt{r(a,b)}$). Aucune des instances $\mathtt{r(a,b)}$ ou $\mathtt{r(a,a)}$ ne devrait être vraie pour que la règle initiale soit déclenchée. Nous définissons un grounding partiel, concernant uniquement les variables universelles. Par exemple, le grounding partiel d'une instance de la règle non-ground précédente serait : ($\mathtt{p(a)} \leftarrow \mathtt{q(a)}, \mathtt{not\ r(a,Z)}.$). Nous utilisons l'expression *partiellement instancié* pour tous les éléments sur lesquels un grounding partiel est appliqué.

**Définition 2.1.4** (Programme partiellement instancié)**.** *Soit $r \in \mathcal{R}$ une règle d'un programme NME $P = (\mathcal{F}, \mathcal{R})$ de langage $\mathcal{L}_P$.* $\mathbf{PG}(r) = \{\sigma(r)$ *— pour tout $\sigma$ une substitution des variables de $\overline{\mathcal{V}_{N\exists}}(r)$ sur $\mathcal{L}_P\}$. Le programme partiellement instancié $\mathbf{PG}(P) = (\mathcal{F}, \mathbf{PG}(\mathcal{R}))$ d'un programme NME $P$ est défini par $\mathbf{PG}(\mathcal{R}) = \bigcup_{r\in\mathcal{R}} \mathbf{PG}(r)$.*

**Exemple 33** (Suite de l'exemple 30)**.** *Le langage du programme skolémisé $sk(P_{30})$ contient uniquement une constante, toto, et seulement un symbole de fonction, $sk^D_{r_1}$. L'ensemble des termes ground est infini et le grounding partiel mène alors au programme infini $\mathbf{PG}(sk(P_{30})) = (\mathcal{F}_{30}, \mathbf{PG}(sk(\mathcal{R}_{30})))$ composé de l'ensemble de règles NME :*

$$\mathbf{PG}(sk(\mathcal{R}_{30})) = \left\{ \begin{array}{l} \mathtt{dT(toto, sk^D_{r_1}(toto)), d(sk^D_{r_1}(toto))} \leftarrow \\ \qquad\qquad \mathtt{p(toto), not\ (eC(toto), dC(toto, Y))} \\ \mathtt{dT(sk^D_{r_1}(toto), sk^D_{r_1}(sk^D_{r_1}(toto))), d(sk^D_{r_1}(sk^D_{r_1}(toto)))} \leftarrow \\ \qquad\qquad \mathtt{p(sk^D_{r_1}(toto)), not\ (eC(sk^D_{r_1}(toto)), dC(sk^D_{r_1}(toto), Y)).} \\ \ldots \end{array} \right\}$$

*et de l'ensemble de faits :*

$$\mathcal{F}_{30} = \{\mathtt{p(toto)}, \mathtt{eC(toto)}\}$$

**Proposition 2.1.2.** *Un programme NME partiellement instancié sans tête multiple, sans ensemble d'atomes liés à une négation par défaut et sans variable existentielle est un programme ASP (classique) ground.*

**Preuve 2.1.2.** *C'est la conséquence directe des définitions 2.1.1 et 2.1.4.*

## 2.1.2 Réduit et modèles

Nous souhaitons maintenant calculer des modèles à partir d'un programme skolémisé et partiellement instancié. Nous allons pour cela redéfinir la notion de réduit et l'opérateur de conséquence afin de calculer un modèle à partir d'un programme partiellement instancié (en tenant compte des variables existentielles dans le corps négatif). Nous adaptons aussi ces définitions à l'écriture choisie pour un programme séparant faits et règles. Le nouveau réduit va nous permettre dans un premier temps de supprimer les corps négatifs d'un programme contenant les variables existentielles restantes à l'aide d'une instanciation à la volée.

**Définition 2.1.5** (Réduit)**.** *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme NME de langage $\mathcal{L}_P$ et $A \subseteq \mathbf{GA}(\mathcal{L}_{sk(P)})$. Le réduit d'un programme partiellement instancié $\mathbf{PG}(sk(P))$ par rapport à $A$ est le programme défini partiellement instancié*

$$\mathbf{PG}(sk(P))^A = \begin{array}{l} (\mathcal{F}, \{\mathtt{tête}(r) \leftarrow \mathtt{corps}^+(r).|r \in \mathbf{PG}(sk(\mathcal{R})), \textit{pour tout } \mathtt{N} \in \mathtt{corps}^-(r) \\ \textit{et pour toute substitution ground } \sigma \textit{ sur } \mathcal{L}_{sk(P)}, \sigma(\mathtt{N}) \not\subseteq A\}) \end{array}$$

**Exemple 34** (Suite de l'exemple 30)**.** *Soit l'ensemble d'atomes*

$$A_1 = \{\mathtt{p(toto)}, \mathtt{eC(toto)}, \mathtt{dT(toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}, \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}\}.$$

*Alors, pour la règle*

$$\mathtt{dT(toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}, \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))} \leftarrow \mathtt{p(toto)}, \mathtt{not}\ (\mathtt{eC(toto)}, \mathtt{dC(toto, Y)}).$$

*il n'y a pas d'instance ground de* $\mathtt{eC(toto)}$,$\mathtt{dC(toto, Y)}$ *incluse dans* $A_1$ *(étant donné que* $A_1$ *ne contient aucune instance de* $\mathtt{dC}$*) et la partie positive de la règle est gardée. Les autres règles sont gardées pour la même raison. Le programme obtenu est alors* $\mathbf{PG}(sk(P_{30}))^{A_1} = (\mathcal{F}_{30}, \mathbf{PG}(sk(\mathcal{R}_{30}))^{A_1})$ *avec l'ensemble de règles NME :*

$$\mathbf{PG}(sk(\mathcal{R}_{30}))^{A_1} = \left\{ \begin{array}{l} \mathtt{dT(toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}, \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))} \leftarrow \mathtt{p(toto)}., \\ \mathtt{dT}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto)}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}), \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}) \leftarrow \mathtt{p}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto)}). \\ \ldots \end{array} \right\}$$

*et l'ensemble de faits :*

$$\mathcal{F}_{30} = \{\mathtt{p(toto)}, \mathtt{eC(toto)}\}$$

*Considérons maintenant*

$$A_2 = A_1 \cup \{\mathtt{dC(toto, info)}\}$$

*et*

$$\mathcal{F}'_{30} = \mathcal{F}_{30} \cup \{\mathtt{dC(toto, info).}\}$$

*Ici,* $\mathtt{eC(toto)}, \mathtt{dC(toto, info)}$ *est une instance ground du corps négatif de la règle*

$$\mathtt{dT(toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}, \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))} \leftarrow \mathtt{p(toto)}, \mathtt{not}\ (\mathtt{eC(toto)}, \mathtt{dC(toto, Y)}).$$

*qui est incluse dans* $A_2$*. La règle est alors exclue du réduit. Les parties positives des autres règles sont conservées. Le programme obtenu est donc* $\mathbf{PG}(sk(P'_{30}))^{A_2} = (\mathcal{F}'_{30}, \mathbf{PG}(sk(\mathcal{R}_{30}))^{A_2})$ *composé de l'ensemble de règles NME :*

$$\mathbf{PG}(sk(\mathcal{R}_{30}))^{A_2} = \left\{ \begin{array}{l} \mathtt{dT}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto)}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}), \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto))}) \leftarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{p}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}\mathtt{(toto)}). \\ \ldots \end{array} \right\}$$

*et de l'ensemble de faits :*

$$\mathcal{F}_{30'} = \{\mathtt{p(toto)}, \mathtt{eC(toto)}, \mathtt{dC(toto, info)}\}$$

**Proposition 2.1.3.** *Le réduit d'un programme NME partiellement instancié est un ensemble de règles existentielles.*

**Preuve 2.1.3.** *C'est la conséquence directe des définitions 2.1.1 et 2.1.4.*

Notons que le réduit d'un programme qui est skolémisé et partiellement instancié est un programme défini ground : il ne contient plus de variables. L'opérateur de conséquence peut alors être défini de façon classique, la seule différence étant que les règles peuvent avoir une conjonction d'atomes en tête.

**Définition 2.1.6** (L'opérateur de conséquence $T_P$ et $Cn$ sa fermeture)**.** *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme défini issu du réduit d'un programme NME de langage $\mathcal{L}_P$. L'opérateur $T_P : 2^{\mathbf{GA}(\mathcal{L}_P)} \rightarrow 2^{\mathbf{GA}(\mathcal{L}_P)}$ est défini par*

$$T_P(A) = \mathcal{F} \cup \{\mathtt{a} | r \in \mathcal{R}, \mathtt{a} \in \mathtt{tête}(r), \mathtt{corps}^+(r) \subseteq A\}.$$

*$Cn(P) = \bigcup_{k \geq 0} T_P^k$ est le plus petit point fixe de l'opérateur de conséquence $T_P$.*

**Exemple 35** (Suite de l'exemple 30)**.** *$Cn(\mathbf{PG}(sk(P_{30}))^{A_1}) = A_1$ mais $Cn(\mathbf{PG}(sk(P'_{30}))^{A_2}) = \{\mathtt{p(toto)}, \mathtt{eC(toto)}, \mathtt{dC(toto, info)}\}$.*

**Définition 2.1.7** (∃-answer set)**.** *Soit $P$ un programme NME de langage $\mathcal{L}_P$ et $A \subseteq \mathbf{GA}(\mathcal{L}_{sk(P)})$. $A$ est un ∃-answer set de $P$ si et seulement si $A = Cn(\mathbf{PG}(sk(P))^A)$.*

**Exemple 36** (Suite de l'exemple 30)**.** *$A_1$ est un ∃-answer set de $P_{30}$ et $\{\mathtt{p(toto)}, \mathtt{eC(toto)}, \mathtt{dC(toto, info)}\}$ est un ∃-answer set de $P'_{30}$.*

**Proposition 2.1.4.** *Soit $P$ un programme ASP (classique) de langage $\mathcal{L}_P$ et $A \subseteq \mathbf{GA}(\mathcal{L}_P)$. $A$ est un answer set de $P$ si et seulement si $A$ est un ∃-answer set de $P$ considéré comme un programme NME.*

**Preuve 2.1.4.** *Étant donné $P$ un programme ASP classique, $sk(P) = P$ et son programme ASP (classique) instancié correspondent exactement à $\mathbf{PG}(P) = \mathbf{PG}(sk(P))$. D'où $X \in \mathbf{GA}(\mathcal{L}_P) = \mathbf{GA}(\mathcal{L}_{sk(P)})$ est un answer set de $\mathbf{G}(P)$, par la définition 2.1.7 et de la définition 1.2.16 d'un answer set classique, si et seulement si c'est un ∃-answer set de $P$ considéré comme un programme NME.*

## 2.2 Traduction d'un programme NME vers ASP

Afin d'utiliser l'efficacité des solveurs ASP pour calculer les modèles de nos programmes NME, nous définissons dans cette section une traduction d'un programme NME vers un programme ASP classique et nous montrons que les ∃-answer set du programme initial correspondent aux answer set du nouveau programme. Pour cela, nous passons par trois phases : la première consistant à remplacer les conjonctions d'atomes avec variables existentielles des corps négatifs par des corps négatifs mono-atomiques sans variables existentielles, la seconde consistant à la skolémisation des variables existentielles en tête de règle, et la dernière à remplacer les règles avec tête multi-atomiques par des règles avec une tête mono-atomique.

### 2.2.1 Normalisation

La première étape de la traduction est la normalisation, qui a deux objectifs : supprimer les conjonctions d'atomes des parties négatives des règles et supprimer les variables existentielles des parties négatives. Le programme obtenu possède des answer set équivalents au programme original.

**Définition 2.2.1.** *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme NME de langage $\mathcal{L}_P$ et $r$ une règle NME de $\mathcal{R}$ ($m + s > 0$, $n, u_1, \ldots, u_s \geq 1$) :*

$$\mathtt{h}_1, \ldots, \mathtt{h}_v \leftarrow \mathtt{b}_1, \ldots, \mathtt{b}_m, \mathtt{not}\ (\mathtt{n}_1^1, \ldots, \mathtt{n}_{u_1}^1), \ldots, \mathtt{not}\ (\mathtt{n}_1^s, \ldots, \mathtt{n}_{u_s}^s).$$

*avec $\mathtt{h}_1, \ldots, \mathtt{h}_v, \mathtt{b}_1, \ldots, \mathtt{b}_m, \mathtt{n}_1^1, \ldots, \mathtt{n}_{u_1}^1, \ldots, \mathtt{n}_1^s, \ldots, \mathtt{n}_{u_s}^s \in \mathbf{A}(\mathcal{L}_P)$. Soit $\mathcal{N}$ un ensemble de nouveaux symboles de prédicat ($\mathcal{N} \cap \mathcal{PS} = \emptyset$).*

*La* normalisation *d'une telle règle NME est l'ensemble des règles NME*

$$\mathbf{N}(r) = \left\{ \begin{array}{l} \mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{v} \leftarrow \mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}, \mathtt{not}\ \mathtt{n}_1, \ldots, \mathtt{not}\ \mathtt{n}_\mathtt{s}. \\ \mathtt{n}_1 \leftarrow \mathtt{n}^1_1, \ldots, \mathtt{n}^1_{\mathtt{u}_1}. \\ \ldots \\ \mathtt{n}_\mathtt{s} \leftarrow \mathtt{n}^\mathtt{s}_1, \ldots, \mathtt{n}^\mathtt{s}_{\mathtt{u}_\mathtt{s}}. \end{array} \right\}$$

*avec* $\mathtt{n}_\mathtt{i}$ *le nouvel atome* $\mathtt{p}^{\mathtt{n}_\mathtt{i}}(\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{w})$, $\mathtt{p}^{\mathtt{n}_\mathtt{i}} \in \mathcal{N}$ *un nouveau symbole de prédicat pour chaque* $\mathtt{n}_\mathtt{i}$ *et* $\{\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{w}\} = \mathcal{V}_\forall(r)(\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}})$, *l'ensemble des variables universelles apparaissant dans le corps négatif* $\mathtt{n}^\mathtt{i}_1, \ldots, \mathtt{n}^\mathtt{i}_{\mathtt{u}_\mathtt{i}}$.

*La normalisation de* $P$ *est définie de la façon suivante :* $\mathbf{N}(P) = (\mathcal{F}, \mathbf{N}(\mathcal{R}))$ *avec* $\mathbf{N}(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} \mathbf{N}(r)$.

**Exemple 37** (Suite de l'exemple 30)**.** *Soit la règle*

$$r_1 : \ \mathtt{dT(X, D)}, \mathtt{d(D)} \leftarrow \ \mathtt{p(X)}, \mathtt{not(eC(X), dC(X, Y))}.$$

*et* $\mathtt{p}^\mathtt{n}$ *un nouveau symbole de prédicat. La partie négative de la règle* $r_1$ *:* $\mathtt{not(eC(X), dC(X, Y))}$ *a une unique variable universelle,* X*. L'ensemble d'atomes lié au* not *est remplacé par* $\mathtt{not}\ \mathtt{p}^\mathtt{n}\mathtt{(X)}$*. Et une nouvelle règle* $r_1^\ddagger$ *est ajoutée où* Y *qui était une variable existentielle dans* $r_1$ *devient une variable universelle dans* $r_1^\ddagger$.

$$\mathbf{N}(r_1) = \left\{ \begin{array}{l} r_1^\dagger : \mathtt{dT(X, D)}, \mathtt{d(D)} \leftarrow \ \mathtt{p(X)}, \mathtt{not}\ \mathtt{p}^\mathtt{n}\mathtt{(X)}. \\ r_1^\ddagger : \mathtt{p}^\mathtt{n}\mathtt{(X)} \leftarrow \ \mathtt{eC(X)}, \mathtt{dC(X, Y)}. \end{array} \right\}$$

$\mathbf{N}(P_{30}) = (\mathcal{F}_{30}, \mathbf{N}(\mathcal{R}_{30}))$ *avec* $\mathbf{N}(\mathcal{R}_{30}) = \{r_1^\dagger, r_1^\ddagger\}$.

**Définition 2.2.2.** *Soit* $\mathbf{GAN}(\mathcal{L}_{sk(P)})$ *l'ensemble d'atomes skolémisé et instancié pour les nouveaux symboles de prédicat* $\mathcal{N}$ *défini comme suit :*

- *si* $\mathtt{a} \in \mathcal{N}$ *avec* $ar(\mathtt{a}) = 0$ *alors* $\mathtt{a} \in \mathbf{GAN}(\mathcal{L}_{sk(P)})$,
- *si* $\mathtt{p} \in \mathcal{N}$ *avec* $ar(\mathtt{p}) > 0$ *et* $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n} \in \mathbf{GT}(\mathcal{L}_{sk(P)})$ *alors* $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n}) \in \mathbf{GAN}(\mathcal{L}_{sk(P)})$.

La proposition suivante montre que la normalisation conserve les answer set d'un programme NME : la normalisation ajoute seulement des atomes formés à partir des nouveaux symboles de prédicats venant de $\mathcal{N}$.

**Proposition 2.2.1.** *Soit* $P$ *un programme NME de langage* $\mathcal{L}_P$ *et* $A \subseteq \mathbf{GA}(\mathcal{L}_{sk(P)})$*. Si* $A$ *est un* $\exists$*-answer set de* $P$ *alors il existe* $B \subseteq \mathbf{GAN}(\mathcal{L}_{sk(P)})$ *tel que* $A \cup B$ *est un* $\exists$*-answer set de* $\mathbf{N}(P)$*. Si* $A$ *est un* $\exists$*-answer set de* $\mathbf{N}(P)$ *alors* $A \setminus \mathbf{GAN}(\mathcal{L}_{sk(P)})$ *est un* $\exists$*-answer set de* $P$.

Le lemme utilisé dans la preuve suivante montre que si la normalisation est appliquée sur seulement une règle $r$ et seulement une partie du corps négatif de la règle, alors les answer set du programme original sont préservés à l'ajout près des nouveaux atomes. Si $r$ a la forme suivante :

$$\mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{v} \leftarrow \ \mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}, \mathtt{not}\ (\mathtt{n}^1_1, \ldots, \mathtt{n}^1_{\mathtt{u}_1}), \ldots, \mathtt{not}\ (\mathtt{n}^\mathtt{s}_1, \ldots, \mathtt{n}^\mathtt{s}_{\mathtt{u}_\mathtt{s}}).$$

alors la normalisation partielle de $r$ pour $(n^s_1, \ldots, n^s_{u_s})$ génère les règles

$$\mathtt{r}^\dagger = \mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{v} \leftarrow \ \mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}, \mathtt{not}\ (\mathtt{n}^1_1, \ldots, \mathtt{n}^1_{\mathtt{u}_1}), \ldots, \mathtt{not}\ (\mathtt{n}^{\mathtt{s}-1}_1, \ldots, \mathtt{n}^{\mathtt{s}-1}_{\mathtt{u}_{\mathtt{s}-1}}), \mathtt{not}\ \mathtt{n}_\mathtt{s}.$$

et

$$r^\ddagger = \mathtt{n}_\mathtt{s} \leftarrow \ \mathtt{n}^\mathtt{s}_1, \ldots, \mathtt{n}^\mathtt{s}_{\mathtt{u}_\mathtt{s}}.$$

Le programme $P$ et le programme $\mathbf{N}(P)$, où la règle $r$ est remplacée par les règles $r^\dagger$ et $r^\ddagger$, ont les mêmes answer set à l'exception de $\mathtt{n}_\mathtt{s}$. La preuve peut-être construite par induction en appliquant le lemme à chaque partie du corps négatif de $r$ puis à chaque règle du programme.

**Preuve 2.2.1.** *La preuve se fait par induction du lemme suivant : Soit $P$ un programme NME de langage $\mathcal{L}_P$, $r = (\mathtt{h} \leftarrow \mathtt{b}, \mathtt{not}\ (\mathtt{n}_1, \ldots, \mathtt{n}_u).) \in \mathbf{PG}(sk(P))$, $P' = \mathbf{PG}(sk(P)) \setminus \{r\}$, $r^\ddagger = \mathbf{PG}(\mathtt{n} \leftarrow \mathtt{n}_1, \ldots, \mathtt{n}_u.) \subseteq \mathbf{PG}(sk(\mathbf{N}(P)))$, $r^\dagger = (\mathtt{h} \leftarrow \mathtt{b}, \mathtt{not}\ \mathtt{n}.)$ et $A \subseteq \mathbf{GA}(\mathcal{L}_{sk(P)})$. S'il existe une substitution $\theta$ telle que $\{\theta(\mathtt{n}_1), \ldots, \theta(\mathtt{n}_u)\} \subseteq A$ alors $Cn((P' \cup \{r\})^A) = A$ si et seulement si $Cn((P' \cup \{r^\dagger\} \cup r^\ddagger)^{A \cup \{\mathtt{n}\}}) = A \cup \{\mathtt{n}\}$. Si pour toutes les substitutions $\theta$, $\{\theta(\mathtt{n}_1), \ldots, \theta(\mathtt{n}_u)\} \not\subseteq A$ alors $Cn((P' \cup \{r\})^A) = A$ si et seulement si $Cn((P' \cup \{r^\dagger\} \cup r^\ddagger)^A) = A$.*

*Remarquons que $\mathtt{n} \notin Cn(P'^A) \cup A$.*

- *S'il existe une substitution $\theta$ telle que $\{\theta(\mathtt{n}_1), \ldots, \theta(\mathtt{n}_u)\} \subseteq A$ alors $(P' \cup \{r\})^A = P'^A = (P' \cup \{r^\dagger\})^{A \cup \{\mathtt{n}\}}$ alors $Cn((P' \cup \{r\})^A) = Cn(P'^A)$ et $Cn((P' \cup \{r^\dagger\} \cup r^\ddagger)^{A \cup \{\mathtt{n}\}}) = Cn(P'^A) \cup \{\mathtt{n}\}$. Alors $Cn((P' \cup \{r\})^A) = A$ ssi $Cn(P'^A) = A$ ssi $Cn(P'^A) \cup \{\mathtt{n}\} = A \cup \{\mathtt{n}\}$ ssi $Cn((P' \cup \{r^\dagger\} \cup r^\ddagger)^{A \cup \{\mathtt{n}\}}) = A \cup \{\mathtt{n}\}$.*
- *Si pour toutes les substitutions $\theta$, $\{\theta(\mathtt{n}_1), \ldots, \theta(\mathtt{n}_u)\} \not\subseteq A$ alors $(P' \cup \{r\})^A = (P' \cup \{\mathtt{h} \leftarrow \mathtt{b}.\})^A$ et $(P' \cup \{r^\dagger\} \cup r^\ddagger)^A = (P' \cup \{\mathtt{h} \leftarrow \mathtt{b}.\})^A \cup r^\ddagger$. Alors $Cn((P' \cup \{r\})^A) = Cn((P' \cup \{\mathtt{h} \leftarrow \mathtt{b}.\})^A) = Cn((P' \cup \{\mathtt{h} \leftarrow \mathtt{b}.\})^A \cup r^\ddagger) = Cn((P' \cup \{r^\dagger\} \cup r^\ddagger)^A)$. Alors $Cn((P' \cup \{r\})^A) = A$ si et seulement si $Cn((P' \cup \{r^\dagger\} \cup r^\ddagger)^A) = A$.*

Un programme NME après normalisation ne possède plus de corps négatifs multi-atomiques ni de variables existentielles dans le corps négatif. Il reste cependant les variables existentielles en tête de règles, pour cela nous appliquons la skolémisation définie précédemment pour supprimer ces variables existentielles.

### 2.2.2 Skolémisation

Comme présenté dans la partie précédente, la skolémisation permet de remplacer les variables existentielles en tête par des termes fonctionnels. Nous obtenons alors des têtes uniquement universellement quantifiées. La skolémisation est donc appliquée de la même manière que dans la définition 2.1.3.

**Exemple 38** (Suite de l'exemple 30)**.** *Le programme $P_{30}$, après normalisation, est skolémisé. Nous obtenons alors l'ensemble de règles :*

$$sk(\mathbf{N}(\mathcal{R}_{30})) = \left\{ \begin{array}{l} \mathtt{dT}(\mathtt{X}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{X})), \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{X})) \leftarrow \mathtt{p}(\mathtt{X}), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{X}). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{X}) \leftarrow \mathtt{eC}(\mathtt{X}), \mathtt{dC}(\mathtt{X}, \mathtt{Y}). \end{array} \right\}$$

*avec l'ensemble de faits :*

$$\mathcal{F}_{30} = \{\mathtt{p}(\mathtt{toto}), \mathtt{eC}(\mathtt{toto})\}$$

Après normalisation et skolémisation, le programme NME ne possède plus de variables existentielles nous pouvons alors effectuer un grounding classique sur le programme normalisé puis skolémisé.

**Exemple 39** (Exemple 30 suite)**.** *Nous appliquons le grounding sur le programme $P_{30}$, après normalisation, et skolémisation. Nous obtenons alors l'ensemble de règles :*

$$\mathbf{PG}(sk(\mathbf{N}(\mathcal{R}_{30}))) = \left\{ \begin{array}{l} \mathtt{dT}(\mathtt{toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \leftarrow \mathtt{p}(\mathtt{toto}), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}) \leftarrow \mathtt{eC}(\mathtt{toto}), \mathtt{dC}(\mathtt{toto}, \mathtt{toto}). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}) \leftarrow \mathtt{eC}(\mathtt{toto}), \mathtt{dC}(\mathtt{toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})). \\ \ldots \\ \mathtt{dT}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}))), \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}))) \leftarrow \\ \qquad\qquad \mathtt{p}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \leftarrow \mathtt{eC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{dC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{toto}). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \leftarrow \mathtt{eC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{dC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})). \\ \ldots \end{array} \right\}$$

*avec l'ensemble de faits :*

$$\mathcal{F}_{30} = \{\mathtt{p(toto)}, \mathtt{eC(toto)}\}$$

Après grounding, le programme ne contient plus de variables. La proposition suivante montre que la skolémisation et le grounding préservent les answer set d'un programme NME normalisé.

**Proposition 2.2.2.** *Soit $P$ un programme NME normalisé de langage $\mathcal{L}_P$ et $A \subseteq \mathbf{GA}(\mathcal{L}_{sk(P)})$. $A$ est un $\exists$-answer set de $P$ si et seulement si $A$ est un $\exists$-answer set de $\mathbf{PG}(sk(P))$.*

**Preuve 2.2.2.** *Étant donné que pour tout $r \in \mathbf{PG}(sk(P)), \mathcal{V}_{N\exists}(r) = \emptyset$ (puisque $r$ est normalisée), $\overline{\mathcal{V}_{N\exists}}(r) = \mathcal{V}(r)$ et $\mathcal{V}_{H\exists}(r) = \emptyset$ (puisque $r$ est skolémisée) alors $\mathbf{PG}(sk(P))$ = sk($\mathbf{PG}(sk(P))$) = $\mathbf{PG}(sk(\mathbf{PG}(sk(P))))$.*

*Par la définition 2.1.7, $A$ est un $\exists$-answer set de $P$ si et seulement si $A = Cn(\mathbf{PG}(sk(P))^A)$ si et seulement si $A = Cn(\mathbf{PG}(sk(\mathbf{PG}(sk(P))))^A)$ si et seulement si $A$ est un $\exists$-answer set de $\mathbf{PG}(sk(P))$.*

Une fois un programme NME normalisé et skolémisé, les seules parties restantes dans le programme ne provenant pas d'un ASP classique sont les conjonctions d'atomes dans les têtes des règles.

### 2.2.3 Expansion

La dernière étape de la traduction consiste en l'expansion où l'on supprime les ensembles d'atomes dans chaque tête tout en gardant le lien entre les variables existentielles. Il suffit simplement de dupliquer une règle autant de fois que la règle possède d'atomes dans sa tête, chaque nouvelle règle ayant un seul de ces atomes en tête. Procéder à la skolémisation avant l'expansion permet de préserver le lien entre les variables existentielles en tête. Le programme obtenu est équivalent en terme d'answer set.

**Définition 2.2.3.** *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme skolémisé puis normalisé ground et $r \in \mathcal{R}$ ($m+s > 0$, $v > 0$) :*

$$\mathtt{h_1}, \ldots, \mathtt{h_v} \leftarrow \mathtt{b_1}, \ldots, \mathtt{b_m}, \mathtt{not\ n_1}, \ldots, \mathtt{not\ n_s}.$$

*L'*expansion *d'une telle règle est l'ensemble défini par :*

$$\mathbf{Exp}(r) = \left\{ \begin{array}{l} \mathtt{h_1} \leftarrow \mathtt{b_1}, \ldots, \mathtt{b_m}, \mathtt{not\ n_1}, \ldots, \mathtt{not\ n_s}. \\ \ldots \\ \mathtt{h_v} \leftarrow \mathtt{b_1}, \ldots, \mathtt{b_m}, \mathtt{not\ n_1}, \ldots, \mathtt{not\ n_s}. \end{array} \right\}$$

*L'expansion de $P$ est définie par $\mathbf{Exp}(P) = (\mathcal{F}, \mathbf{Exp}(\mathcal{R}))$ avec $\mathbf{Exp}(\mathcal{R}) = \bigcup_{r\in\mathcal{R}} \mathbf{Exp}(r)$.*

**Exemple 40** (Suite de l'exemple 30)**.** *La règle suivante du programme de l'exemple 39 :*

$$\mathtt{dT(toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r_1}}\mathtt{(toto))}, \mathtt{d(sk}^{\mathtt{D}}_{\mathtt{r_1}}\mathtt{(toto))} \leftarrow \mathtt{p(toto)}, \mathtt{not\ p^n(toto)}.$$

*est séparée en deux règles :*

$$\left\{ \begin{array}{l} \mathtt{dT(toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r_1}}\mathtt{(toto))} \leftarrow \mathtt{p(toto)}, \mathtt{not\ p^n(toto)}. \\ \mathtt{d(sk}^{\mathtt{D}}_{\mathtt{r_1}}\mathtt{(toto))} \leftarrow \mathtt{p(toto)}, \mathtt{not\ p^n(toto)}. \end{array} \right\}$$

*Le même traitement est appliqué aux autres règles avec les prédicats* dT *et* d *en tête.*
*Nous obtenons le programme composé de l'ensemble de règles :*

$$\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(\mathcal{R}_{30})))) = \left\{ \begin{array}{l} \mathtt{dT}(\mathtt{toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \leftarrow \mathtt{p}(\mathtt{toto}), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}). \\ \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \leftarrow \mathtt{p}(\mathtt{toto}), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}) \leftarrow \mathtt{p}(\mathtt{toto}), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{toto}) \leftarrow \mathtt{eC}(\mathtt{toto}), \mathtt{dC}(\mathtt{toto}, \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})). \\ \ldots \\ \mathtt{dT}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}))) \leftarrow \\ \qquad\qquad \mathtt{p}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \\ \mathtt{d}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}))) \leftarrow \mathtt{p}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{not}\ \mathtt{p}^{\mathtt{n}}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \leftarrow \mathtt{eC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{dC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{toto}). \\ \mathtt{p}^{\mathtt{n}}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})) \leftarrow \mathtt{eC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto})), \mathtt{dC}(\mathtt{sk}^{\mathtt{D}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{sk}(\mathtt{toto})). \\ \ldots \end{array} \right\}$$

*et de l'ensemble de faits :*

$$\mathcal{F}_{30} = \{\mathtt{p}(\mathtt{toto}), \mathtt{eC}(\mathtt{toto})\}$$

**Proposition 2.2.3.** *Soit $P$ un programme NME skolémisé puis normalisé ground de langage $\mathcal{L}_P$ et $A \subseteq \mathbf{GA}(\mathcal{L}_P)$. $A$ est un $\exists$-answer set de $P$ si et seulement si $A$ est un $\exists$-answer set de $\mathbf{Exp}(P)$.*

**Preuve 2.2.3.** *La seule différence est sur le calcul du point fixe de l'opérateur de conséquence (classique) $T_P$ et le nouvel opérateur $T_P$ défini dans la définition 2.1.6 dont le point fixe est clairement identique étant donné que $P$ est ground.*

**Proposition 2.2.4.** *Soit $P$ un programme NME. Le programme $P$ normalisé, puis skolémisé ground, et expansé $\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(P))))$, est un programme ASP (classique ground).*

**Preuve 2.2.4.** *Cette proposition est une conséquence directe des définitions 2.1.3, 2.1.4, 2.2.1, 2.2.3 et de la proposition 2.1.2.*

La proposition suivante établit l'équivalence, aux nouveaux atomes introduits par la normalisation près, entre les $\exists$-answer set d'un programme NME et les answer set classiques du programme après normalisation, skolémisation et expansion. Nous montrons qu'un $\exists$-answer set calculé à partir d'un programme NME est équivalent à un answer set calculé sur sa version ASP après transformation.

**Proposition 2.2.5.** *Soit $P$ un programme NME de langage $\mathcal{L}_P$ et $A \subseteq \mathbf{GA}(\mathcal{L}_{sk(P)})$. Si $A$ est un $\exists$-answer set de $P$ alors il existe $B \subseteq \mathbf{GAN}(\mathcal{L}_{sk(P)})$ tel que $A \cup B$ est un answer set (classique) de $\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(P))))$. Si $A$ est un answer set (classique) de $\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(P))))$, alors $A \setminus \mathbf{GAN}(\mathcal{L}_{sk(P)})$ est un $\exists$-answer set de $P$.*

**Preuve 2.2.5.** *Soit $P$ un programme NME et $A \subseteq \mathbf{GA}(\mathcal{L}_{sk(P)})$.*

- *si $A$ est un $\exists$-answer set de $P$ alors, d'après la proposition 2.2.1, il existe $B \subseteq \mathbf{GAN}(\mathcal{L}_{sk(P)})$ tel que $A \cup B$ est un $\exists$-answer set de $\mathbf{N}(P)$. D'après la proposition 2.2.2, $A \cup B$ est un $\exists$-answer set de $\mathbf{PG}(sk(\mathbf{N}(P)))$. D'après la proposition 2.2.3, $A \cup B$ est un $\exists$-answer set de $\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(P))))$. D'après les proposition 2.1.4 et 2.2.4, $A \cup B$ est un answer set de $\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(P))))$.*
- *Si $A$ est un answer set (classique) de $\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(P))))$ alors, d'après les propositions 2.1.4 et 2.2.4, $A$ est un $\exists$-answer set de $\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(P))))$. D'après la proposition 2.2.3, $A$ est un $\exists$-answer set de $\mathbf{PG}(sk(\mathbf{N}(P)))$. D'après la proposition 2.2.2, $A$ est un $\exists$-answer set de $\mathbf{N}(P)$. D'après la proposition 2.2.1, $A \setminus \mathbf{GAN}(\mathcal{L}_{sk(P)})$ est un $\exists$-answer set de $P$.*

## 2.3 Comparaison des chase avec l'opérateur de conséquence

Pour un programme logique défini l'algorithme de marche avant utilisé pour calculer un modèle est l'opérateur de conséquence tandis qu'en règles existentielles nous considérons les différents chase. Nous souhaitons mettre en avant une comparaison des chase utilisés en règles existentielles avec cet opérateur de point fixe, le but étant de trouver le chase optimal se rapprochant le plus de celui-ci parmi ceux présentés précédemment. Pour cela nous allons représenter les variables existentielles à l'aide des symboles de fonction pour les programmes logiques définis. Les symboles de fonctions permettant de représenter une infinité de nouvelles constantes dans un programme de façon similaire aux variables existentielles. Reprenons la définition 1.2.7 de l'opérateur de point fixe. Soit $P = (\mathcal{F}, \mathcal{R})$ un programme défini instancié et $A$ un ensemble d'atomes. L'opérateur $T_P$ défini par :

$$T_P(A) = \mathcal{F} \cup \{\texttt{tête}(r) \mid r \in \mathcal{R},\ \texttt{corps}(r) \subseteq A\}$$

calcule l'ensemble des atomes déductibles à partir de $P$ et de $A$. À partir de cet opérateur on définit la suite :

$$T_P^0 = \emptyset \; et \; T_P^{k+1} = T_P(T_P^k), \forall k \geq 0$$

$\cup_{k\geq 0} T_P^k$ est le plus petit point fixe de $T_P$ et $Cn(P) = \cup_{k\geq 0} T_P^k$. $Cn(P)$ contient tous les atomes que l'on peut déduire à partir de $P$.
Pour comparer les chase à l'opérateur de point fixe nous considérons uniquement les programmes définis (la vérification qu'un ensemble d'atomes est un modèle d'un programme se faisant sur les programmes définis en ASP suite à l'application du réduit).

**Conjecture 1.** *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme de règles existentielles, nous avons $Cn(\mathbf{Exp}(\mathbf{PG}(sk(P)))) = \alpha_\infty(\mathcal{F}, \mathcal{R})$ avec l'utilisation du skolem-chase.*

**Exemple 41.** *Soit le programme $P_{41} = (\mathcal{F}_{41}, \mathcal{R}_{41})$ avec :*

$$\mathcal{R}_{41} = \{\ r_1 : \texttt{p(X, Y)} \rightarrow\ \texttt{p(X, Z)}).\ \}$$

*et*

$$\mathcal{F}_{41} = \{\texttt{p(a, b)}\}$$

*Nous appliquons le skolem-chase sur le programme $P_{41}$, soit $\mathcal{F}_{41}$ l'ensemble de faits initial nous avons $\mathcal{F}_{41}^1 = \alpha(\mathcal{F}_{41}, \mathcal{R}_{41}, \pi^1) = \{\texttt{p(a, b)}, \texttt{p(a, sk(a))}\}$ avec $\pi^1(\texttt{X}) = \texttt{a}, \pi^1(\texttt{Y}) = \texttt{b}$ puis $\mathcal{F}_{41}^2 = \alpha(\mathcal{F}_{41}^1, \mathcal{R}_{41}, \pi^2) = \{\texttt{p(a, b)}, \texttt{p(a, sk(a))}\}$ avec $\pi^2(\texttt{X}) = \texttt{a}, \pi^2(\texttt{Y}) = \texttt{sk(a)}$ et le skolem-chase s'arrête car il n'existe plus d'homomorphisme différent de $\pi^1$ et $\pi^2$ pour appliquer la règle $r_1$ avec $\mathcal{F}41^2$.*

*Maintenant nous calculons le modèle de $P_{41}$ avec l'opérateur de point fixe d'ASP, pour cela nous skolémisons tout d'abord le programme $P_{41}$ avec :*

$$sk(r_1) : \texttt{p(X, sk(X))} \leftarrow\ \texttt{p(X, Y)}.$$

*puis nous appliquons l'opérateur de point fixe sur le programme skolémisé et instancié. $T_{P_{41}}^0 = \emptyset$, $T_{P_{41}}^1 = \mathcal{F}_{41} = \{\texttt{p(a, b)}\}$, $T_{P_{41}}^2 = \mathcal{F}_{41} \cup \{\texttt{p(a, sk(a))}\}$, $T_{P_{41}}^3 = T_{P_{41}}^2$, nous avons alors atteint un point fixe et donc $Cn(P_{41}) = \{\texttt{p(a, b)}, \texttt{p(a, sk(a))}\}$ ce qui est identique au modèle issu du skolem-chase.*

### 2.3.1 Limitation de représentation

Nous avons vu dans la section 2.1 que le formalisme des règles non-monotones existentielles permet la représentation syntaxique de l'ensemble des programmes ASP et de règles existentielles. En terme de sémantique, ce formalisme permet aussi de représenter l'ensemble des programmes ASP et

les modèles associés, néanmoins nous sommes limités pour la sémantique des règles existentielles. Comme discuté dans la section 1.3.3 les règles existentielles comprennent plusieurs chase définissant les propriétés nécessaires à l'application d'une règle et permettant ainsi de déclencher la terminaison de certains programmes plus ou moins tôt. La sémantique proposée pour le formalisme des programmes NME peut facilement être associée au skolem-chase grâce au processus de skolémisation proposé. Malheureusement, le chase permettant de représenter le plus de modèles finis avec les règles existentielles est le core-chase, et on sait que $skolem-finite < restricted-finite < core-finite$ (le skolem-chase se termine moins souvent que le restricted-chase et le core-chase).

**Exemple 42.** *Soit le programme* $P_{22} = (\mathcal{F}_{22}, \mathcal{R}_{22})$ *avec :*

$$\mathcal{R}_{22} = \{\ r_1 : \mathtt{p(X,Y)}, \mathtt{p(Y,Y)}, \mathtt{q(Y)} \leftarrow\ \mathtt{q(X)}.\ \}$$

*et*

$$\mathcal{F}_{22} = \{\mathtt{q(a)}\}$$

*si nous interprétons ce programme comme un programme de règles existentielles en utilisant le restricted-chase pour le calcul du modèle, nous obtenons le modèle suivant :*

$$\{\mathtt{q(a)}, \mathtt{p(a,Y_0)}, \mathtt{p(Y_0,Y_0)}, \mathtt{q(Y_0)}\}$$

*tandis qu'avec l'exécution du skolem-chase ne s'arrête pas et nous obtenons un modèle infini :*

$$\{\mathtt{q(a)}, \mathtt{p(a}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}, \mathtt{p(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}, \mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}, ...\}$$

*Si nous considérons maintenant le programme* $P_{22} = (\mathcal{F}_{22}, \mathcal{R}_{22})$ *comme un programme NME et que nous le traduisons en ASP classique nous obtenons tout d'abord :*

$$\mathbf{Exp}(\mathbf{PG}(sk(\mathbf{N}(\mathcal{R}_{22})))) = \left\{ \begin{array}{l} \mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))} \leftarrow\ \mathtt{q(a)}. \\ \mathtt{p(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)), sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)))} \leftarrow\ \mathtt{q(a)}. \\ \mathtt{p(a, sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)))} \leftarrow\ \mathtt{q(a)}. \\ \mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)))} \leftarrow\ \mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}. \\ \mathtt{p(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)))}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))} \leftarrow\ \mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}. \\ \mathtt{p(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)))} \leftarrow\ \mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}. \\ \ldots \end{array} \right\}$$

*et*

$$\mathcal{F}_{22} = \{\mathtt{q(a)}\}$$

*Si nous appliquons l'opérateur de point fixe l'unique answer set obtenu est :*

$$\{\mathtt{q(a)}, \mathtt{p(a}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}, \mathtt{p(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a)}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}, \mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(a))}, ...\}$$

*qui est identique au modèle obtenu par le skolem-chase en règles existentielles.*

Nous considérons maintenant un algorithme de marche avant reprenant le principe du restricted-chase, où l'on applique une règle uniquement si elle n'ajoute pas d'information redondante. Pour le calcul d'answer set avec notre formalisme de programme NME nous obtenons un answer set équivalent au modèle obtenu en règles existentielles pour l'exemple précédent. Mais le problème survient lors de l'ajout d'une négation par défaut.

**Exemple 43.** *Soit le programme* $P_{43} = (\mathcal{F}_{43}, \mathcal{R}_{43})$

$$\mathcal{R}_{43} = \left\{ \begin{array}{l} \texttt{parent}(\texttt{Y},\texttt{X}), \texttt{parent}(\texttt{Z},\texttt{X}) \leftarrow \texttt{humain}(\texttt{X}). \\ \texttt{femme}(\texttt{X}) \leftarrow \texttt{parent}(\texttt{X},\texttt{Y}),\ \texttt{not}\ \texttt{homme}(\texttt{X}). \\ \texttt{homme}(\texttt{X}) \leftarrow \texttt{parent}(\texttt{X},\texttt{Y}),\ \texttt{not}\ \texttt{femme}(\texttt{X}). \\ \texttt{aPere}(\texttt{Y}) \leftarrow \texttt{parent}(\texttt{X},\texttt{Y}), \texttt{homme}(\texttt{X}). \\ \texttt{aMere}(\texttt{Y}) \leftarrow \texttt{parent}(\texttt{X},\texttt{Y}), \texttt{femme}(\texttt{X}). \\ \texttt{ok} \leftarrow \texttt{humain}(\texttt{X}), \texttt{aPere}(\texttt{X}), \texttt{aMere}(\texttt{X}). \end{array} \right\}$$

*et*

$$\mathcal{F}_{43} = \{\texttt{humain}(\texttt{a})\}$$

*Ce programme représente une famille d'un enfant et teste si l'enfant possède deux parents dont une mère et un père. Ici l'exécution du skolem-chase calcule quatre answer set grâce à l'application de la première règle qui nous permet de déduire* $\texttt{parent}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a}), \texttt{a})$ *et* $\texttt{parent}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a}), \texttt{a})$ *puis deux applications possibles, dans un cas par manque d'information le parent est une femme et dans l'autre cas toujours par manque d'information c'est un homme. Ce qui nous donne les answer set suivants :*

$\{\texttt{humain}(\texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{femme}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a})), \texttt{femme}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a})), \texttt{aMere}(\texttt{a})\}$
$\{\texttt{humain}(\texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{femme}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a})), \texttt{homme}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a})), \texttt{aMere}(\texttt{a}),$
$\texttt{aPere}(\texttt{a}), \texttt{ok}\}$
$\{\texttt{humain}(\texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{homme}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a})), \texttt{femme}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a})), \texttt{aMere}(\texttt{a}),$
$\texttt{aPere}(\texttt{a}), \texttt{ok}\}$
$\{\texttt{humain}(\texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{parent}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a}), \texttt{a}), \texttt{homme}(\texttt{sk}^{\texttt{Y}}_{\texttt{r}_1}(\texttt{a})), \texttt{homme}(\texttt{sk}^{\texttt{Z}}_{\texttt{r}_1}(\texttt{a})), \texttt{aPere}(\texttt{a})\}$

*Définissant les cas où si l'on n'a aucune information sur une personne alors nos pouvons considérer par défaut que c'est soit un homme soit une femme, ce qui implique que sans information sur les parents d'une personne celui-ci peut avoir un père et une mère mais aussi deux pères ou deux mères. Ici nous déduisons dans deux answer set qu'il manque un parent du sexe opposé et dans les deux autres nous déduisons* ok*. Tandis que si nous effectuons le restricted-chase il ne reste que deux answer set car lors de l'application de la première règle nous avons déjà une information redondante. Nous ne gardons alors qu'un seul* $\texttt{parent}(\texttt{Y}_0, \texttt{a})$ *ce qui nous amène à avoir les answer set suivants :*

$\{\texttt{humain}(\texttt{a}), \texttt{parent}(\texttt{Y}_0, \texttt{a}), \texttt{femme}(\texttt{Y}_0), \texttt{aMere}(\texttt{a})\}$
$\{\texttt{humain}(\texttt{a}), \texttt{parent}(\texttt{Y}_0, \texttt{a}), \texttt{homme}(\texttt{Y}_0), \texttt{aPere}(\texttt{a})\}$

*Nous pouvons ainsi obtenir deux ensembles d'answer set différents si nous considérons notre programme avec le skolem-chase ou le restricted-chase. Le problème est que le restricted-chase est sensé détecter plus de redondance et ainsi permettre le calcul d'un modèle plus général que celui issu du skolem-chase, en supprimant les redondances possibles dans le modèle, or ici l'atome* ok *est déduit dans deux des answer set du skolem-chase alors qu'il n'est pas déduit avec le restricted-chase. Pour pouvoir utiliser un chase plus fort il faudrait alors définir une nouvelle sémantique adaptée aux programmes NME, mais pour le moment le sens de cette nouvelle sémantique n'apporte pas une solution satisfaisante à la représentation que nous souhaitons faire. C'est pourquoi sans résultat plus satisfaisant il est préférable de continuer sur un skolem-chase même si le nombre de programmes pour lesquels le chase se termine est moins important.*

## 2.4 Conclusion

Nous proposons dans un premier temps un langage basé sur des règles non-monotones existentielles étendant à la fois ASP et les règles existentielles. Ce langage nous permet de représenter de manière

formelle l'ensemble des programmes provenant d'ASP et des règles existentielles, tout en conservant une sémantique proche d'ASP, notamment pour le calcul de modèle. Nous prouvons également l'équivalence en terme de modèle d'un programme ASP lorsque considéré comme un programme NME. Afin de permettre le traitement de ce langage à l'aide de solveurs ASP existants nous proposons une traduction vers ASP se déroulant en trois phases, la normalisation permettant de ramener les règles à un corps négatif mono-atomique sans variable existentielle, la skolémisation supprimant les variables existentielles en tête de règle et l'expansion ramenant les règles à une tête mono-atomique tout en conservant le lien entre les anciennes variables existentielles. Ce formalisme permet donc la représentation d'ontologies avec exception en étendant à la fois ASP et les règles existentielles, mais comme ces deux langages la terminaison d'un programme n'est pas assurée. Le calcul des modèles avec un programme NME reste indécidable, tout comme en ASP avec symboles de fonction et règles existentielles. En effet, les symboles de fonction nous empêchent d'assurer la terminaison d'un programme, nous proposons alors dans la prochaine section une étude sur la décidabilité dans le cadre des règles non-monotones existentielles permettant de conclure sur le calcul de modèle de ce formalisme. Nous aborderons par la suite le thème de l'interrogation d'un programme NME et de l'implémentation d'un solveur pour celui-ci.

# 3

# Décidabilité

## 3.1 Introduction

Dès lors que l'on raisonne sur un langage proposant des modèles possiblement infinis, la question de la décidabilité pour le calcul d'un modèle se pose. En effet, la terminaison d'un programme en un temps fini n'est pas toujours assurée : pour les règles existentielles à cause de l'utilisation de variables existentielles en tête de règle d'où l'étude de sa décidabilité [20] [29] ; pour ASP à cause de ses symboles de fonction [21] [69] permettant même d'obtenir une infinité d'answer set pour un programme. La question qui se pose est de savoir si le raisonnement sur un programme NME se termine en un temps fini. Par exemple, la règle existentielle :

$$r_1 : \texttt{humain}(\texttt{X}) \rightarrow \texttt{humain}(\texttt{Y}),\ \texttt{parent}(\texttt{X},\texttt{Y}).$$

signifiant que "pour tout humain X il existe un humain Y qui est le père de X auquel on ajoute le fait"

$$\mathcal{F} = \{\texttt{humain}(\texttt{toto})\}$$

le calcul d'un modèle ne se termine pas car pour l'humain toto on déduit qu'il existe un nouvel humain $\texttt{Y}_0$ parent de celui-ci, puis on déduit qu'il existe un humain $\texttt{Y}_1$ parent de $\texttt{Y}_0$ et ainsi de suite. De même si l'on skolémise le programme et que l'on remplace la variable Y par un symbole de fonction sk, on a une infinité d'imbrications de symboles de fonction sk. Contrairement au calcul d'un modèle pour le programme suivant :

$$\mathcal{R} = \left\{ \begin{array}{l} r_1 : \texttt{parent}(\texttt{X},\texttt{Y}) \rightarrow \texttt{humain}(\texttt{X}), \texttt{estParent}(\texttt{X}). \\ r_2 : \texttt{humain}(\texttt{Y}) \rightarrow \texttt{parent}(\texttt{X},\texttt{Y}). \end{array} \right\}$$

avec le fait

$$\mathcal{F} = \{\texttt{humain}(\texttt{toto}), \texttt{estParent}(\texttt{toto})\}$$

dont la terminaison est assurée, car une fois les règles $r_1$ puis $r_2$ appliquées une première fois, il nous manque l'information que le nouvel humain est parent pour appliquer de nouveau $r_1$. L'information estParent pour le nouvel humain ne peut ici jamais être créée sous forme de faits. La terminaison de ce programme peut facilement être identifiée par une propriété cherchant l'absence d'un cycle incluant une variable existentielle dans le graphe de dépendance des prédicats.

Il existe ainsi plusieurs classes de programmes décidables en règles existentielles et en ASP. Ces classes, dites de décidabilité, peuvent être abstraites ou concrètes. Les classes abstraites étant des

classes dont l'appartenance d'un programme est indécidable regroupant plusieurs classes concrètes dont l'appartenance d'un programme est décidable. Nous allons dans un premier temps présenter dans la section 3.2 une première classe abstraite commune à ASP et aux règles existentielles, appelée *Finite Expansion Set*. Nous présentons cette classe tout d'abord comme définie dans la littérature, puis nous proposons une homogénéisation des classes concrètes la composant à l'aide d'un nouveau graphe que nous étendons par la suite, nous finirons enfin par la prise en compte de la négation par défaut pour améliorer les classes présentées. Nous discuterons ensuite sur les autres classes de décidabilité issues uniquement des règles existentielles que sont *Finite Unification Set* et *Bounded Tree-Width Set* et leur compatibilité avec les programmes NME dans la section 3.3. Enfin nous conclurons dans la dernière section sur les classes de décidabilité pour les programmes NME.

Remarque : pour la suite de ce chapitre nous utiliserons la syntaxe des programmes NME pour tous les exemples.

## 3.2 Finite Expansion Set

Parmi l'ensemble des classes de décidabilité existantes, on retrouve la classe d'ensemble à expansion finie, notée FES (Finite Expansion Set) commune à plusieurs domaines, nous la retrouvons notamment dans la littérature ASP et règles existentielles. Cette classe regroupe l'ensemble des programmes dont la terminaison en marche avant en un temps fini est assurée. Comme discuté dans la section 1.3.3 traitant des différents chase, nous savons que $oblivious - finite \subset skolem - finite \subset restricted - finite \subset core - finite$, nous pouvons ajouter que $core - finite = FES$ comme démontré dans [17]. La classe FES correspond donc à l'ensemble des programmes se terminant en un temps fini lors de l'utilisation du core chase. Malheureusement, les programmes FES ne sont pas reconnaissables, mais cette classe dite abstraite peut être décomposée en plusieurs classes décidables dites concrètes. Cette partie fait suite aux articles [46, 47] présentant l'ensemble des classes concrètes connues incluses dans FES. Nous allons donc présenter dans un premier temps ces classes et leurs propriétés, pour ensuite introduire notre contribution avec une homogénéisation de ces classes à l'aide d'un graphe puis une extension de celles-ci. Nous terminerons sur la classe FES en discutant de l'usage de la négation par défaut pour la terminaison de programmes.

### 3.2.1 Présentation

La classe FES correspond donc à l'ensemble des programmes qui se terminent en un temps fini lors de l'application d'un algorithme en marche avant. Nous définissons la classe FES de la manière suivante :

**Définition 3.2.1.** *Soit $\mathcal{R}$ un ensemble de règles, $\mathcal{R}$ est un ensemble à expansion finie si pour tout ensemble de faits $\mathcal{F}$, il existe un entier $k$ tel que $\alpha_k(\mathcal{F}, \mathcal{R}) = \alpha_\infty(\mathcal{F}, \mathcal{R})$.*

Cette classe représente une classe de décidabilité abstraite, c'est un problème indécidable de savoir si un ensemble de règles est FES, les ensembles de règles FES ne sont pas reconnaissables. On peut néanmoins définir des sous-classes décidables, appelées classes concrètes, de la classe FES. Parmi ces classes, nous pouvons identifier deux grands ensembles distincts : un premier qui s'intéresse aux dépendances entre les positions au sein des atomes des règles du programme (une variable existentielle est-elle transmise d'une position à une autre) ; et le second qui s'intéresse à la dépendance entre règles (l'application d'une règle permet-elle l'application d'une autre). Le premier ensemble est représenté principalement par les classes Weakly Acyclic, Finite Domain, Argument Restricted, Jointly Acyclic, Super Weakly Acyclic et Model-Summarizing Acyclicity tandis que le second est simplement représenté par la classe acyclicity Graph of Rule Dependencies. Pour ces deux ensembles

le but est de chercher l'absence de cycle dans la transmission des variables existentielles pour le premier ensemble et dans l'application des règles pour le second. Nous allons présenter tout d'abord le premier ensemble, dont les classes seront présentées par ordre d'efficacité, la première classe étant la moins efficace (détecte le moins de terminaison de programmes) et la dernière la plus efficace, puis nous aborderons la classe aGRD. Dans les exemples utilisés pour chaque classe nous utilisons la skolémisation comme définie dans la définition 2.1.3 pour traiter les variables existentielles. Les programmes NME utilisant la skolémisation et les cas de terminaison du skolem chase étant inclus dans le core chase.

### Weakly Acyclic

La première classe proposée est Weakly Acyclic que l'on notera WA. Cette classe propose initialement une détection de cycles dans le graphe de dépendance des prédicats assurant un risque que le chase sur le programme ne se termine pas. Pour cela [37] [38] proposent de détecter les cycles possédant au moins un arc existentiel dans le graphe des prédicats. En effet, dès lors qu'il existe un arc existentiel dans un graphe de dépendance des prédicats on sait qu'un nouvel élément existentiel va être créé et que le chase sur le programme peut ne pas se terminer.

**Définition 3.2.2** (Weakly Acyclic)**.** *Un ensemble de règles $\mathcal{R}$ est WA s'il n'existe pas de cycle dans le graphe des prédicats de $\mathcal{R}$ possédant un arc existentiel.*

**Exemple 44.** *Soit le programme $P_{44} = (\mathcal{F}_{44}, \mathcal{R}_{44})$ avec :*

$$\mathcal{R}_{44} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X)}, \mathtt{q(Y)} \leftarrow \mathtt{h(X)}. \\ r_2 : \mathtt{h(U)} \leftarrow \mathtt{p(U)}. \end{array} \right\}$$

*Nous devinons facilement que ce programme est Weakly Acyclic car l'unique variable existentielle est* Y *et elle ne fait pas partie d'un cycle dans le graphe de la figure 3.1. Cette variable ne peut donc pas se transmettre et créer une boucle infinie d'existentielles. Soit un fait $\mathcal{F}_{44} = \{\mathtt{h(titi)}\}$, l'application des règles $r_1$ et $r_2$ ne peut pas boucler infiniment car si on applique $r_1$ on obtient $\mathtt{p(titi)}$ et $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}(\mathtt{titi}))$, avec $\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}$ un symbole de skolem créé à partir de* Y*, puis $r_2$ donne $\mathtt{h(titi)}$ et le programme s'arrête car une nouvelle application de $r_1$ donnerait un symbole de skolem identique à la première application.*

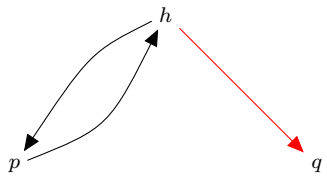



FIGURE 3.1 – Graphe de dépendance des prédicats du programme Weakly Acyclic $P_{44}$

A contrario

**Exemple 45.** *Soit le programme $P_{45} = (\mathcal{F}_{45}, \mathcal{R}_{45})$ avec :*

$$\mathcal{R}_{45} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X,Y)} \leftarrow \mathtt{h(X)}, \mathtt{q(X)}. \\ r_2 : \mathtt{h(V)} \leftarrow \mathtt{p(U,V)}. \end{array} \right\}$$

*Ce programme n'est pas Weakly Acyclic car il existe un cycle avec un arc existentiel dans le graphe des prédicats de la figure 3.2. Nous pouvons ici aussi comprendre facilement que la variable existentielle* $\mathtt{Y}$ *dans* $r_1$ *va se transmettre par l'intermédiaire de* $\mathtt{V}$ *dans* $r_2$*. Soit le fait* $\mathcal{F}_{45} = \{\mathtt{h(toto)}, \mathtt{q(toto)}\}$*,* $r_1$ *est déclenchée et ajoute* $\mathtt{p}(\mathtt{toto}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ *puis on déclenche* $r_2$ *avec* $\mathtt{p}(\mathtt{toto}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ *qui donne* $\mathtt{h}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$*, qui se transmet à la position de* $\mathtt{h}$*, et s'arrête car on ne peut pas déclencher* $r_1$ *étant donné qu'on ne peut pas avoir* $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$*.*

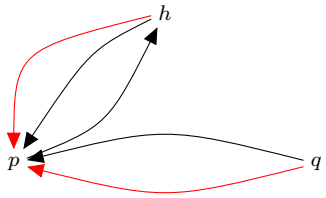



FIGURE 3.2 – Graphe de dépendance des prédicats du programme non Weakly Acyclic $P_{45}$

Le problème est que le programme $P_{45}$ sera considéré comme ne se terminant pas par la propriété Weakly Acyclic alors qu'il se termine et montre donc les limites de cette classe qui ne s'intéresse qu'à la dépendance entre prédicats sans analyser le lien entre les positions des variables au sein des atomes. WA est donc la première classe permettant de déterminer si un programme avec variables existentielles va se terminer mais c'est aussi la plus faible. Nous l'avons remarqué notamment ici en ne pouvant pas déterminer que le programme $P_{45}$ se termine.

### Finite Domain

Le programme $P_{45}$ met en avant l'intérêt de s'intéresser aux liens entre les positions des variables, ce que propose la classe Finite Domain (FD). La classe FD est initialement proposée pour ASP dans [25], elle permet de déterminer si un cycle contenant des variables existentielles va s'arrêter en regardant s'il existe une variable de la frontière qui ne fait pas partie de ce cycle et permet de limiter l'instanciation d'une variable existentielle localement. De ce fait on restreint le nombre d'existentielles pouvant être créés car elles seront dépendantes des instanciations de la frontière. Nous définissons pour cela la notion de position de la manière suivante.

Notation : On note $fr(r)$ l'ensemble des variables de la frontière de la règle $r$.

**Définition 3.2.3** (Position)**.** *Soit une règle* $r$ *d'un programme* $P$*, une position dans* $r$ *d'un prédicat* $\mathtt{a}$ *est notée* $\mathtt{r}[\mathtt{a}_\mathtt{i}, \mathtt{X}_\mathtt{j}]$ *avec* $i$ *la place occupée par le prédicat* $\mathtt{a}$ *dans* $r$ *et* $\mathtt{X}$ *un argument de* $\mathtt{a}$ *à la place* $j$*. On appelle position existentielle (resp. frontière) la position d'une variable existentielle (resp. frontière).*

Remarque : $i$ n'est utilisée que s'il existe une ambigüité entre deux prédicats.

**Exemple 46.** *Soit une règle*

$$r : \ \mathtt{p(X,Y)} \leftarrow \ \mathtt{p(X,Z)}, \mathtt{q(Z,Y)}.$$

*nous avons alors les positions suivantes :* $\mathtt{r}[\mathtt{p}_1, \mathtt{X}_1]$*,* $\mathtt{r}[\mathtt{p}_1, \mathtt{Z}_2], \mathtt{r}[\mathtt{q}, \mathtt{Z}_1], \mathtt{r}[\mathtt{q}, \mathtt{Y}_2], \mathtt{r}[\mathtt{p}_2, \mathtt{X}_1], \mathtt{r}[\mathtt{p}_2, \mathtt{Y}_2]$*. La numérotation commence par les prédicats du corps puis ceux de la tête.*

**Définition 3.2.4** (Finite Domain)**.** *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme, l'ensemble des positions Finite Domain de $\mathcal{R}$ est l'ensemble maximal $FD(\mathcal{R})$ de positions de $\mathcal{R}$ tel que, pour chaque position* pos*, chaque règle $r$ dont* pos $\in$ *tête*$(r)$ *satisfait la condition suivante. Soit* t *le terme correspondant à la position* pos *apparaissant dans* *tête*$(r)$ *alors :*

- *soit $vars(\mathtt{t}) = \emptyset$, ou*
- t *est un sous-terme d'une position Finite Domain d'un prédicat appartenant au corps positif d'une règle ou*
- *chaque variable apparaissant dans $t$ apparaît aussi dans une position Finite Domain d'un prédicat d'un corps positif n'étant pas récursif avec* pos*.*

*Si toutes les positions de $\mathcal{R}$ sont Finite Domain, alors $P$ est dit Finite Domain.*

**Exemple 47.** *Soit le programme $P_{45}$ de l'exemple 45. $P_{45}$ est FD car la position $\mathtt{r}_1[\mathtt{q}, \mathtt{X}_1]$ satisfait la troisième condition en n'étant pas récursif avec la position $\mathtt{r}_1[\mathtt{p}, \mathtt{Y}_2]$ et ne peut donc pas être instanciée par la même variable existentielle que* $\mathtt{h}(\mathtt{V})$*.*

Nous pouvons facilement en déduire que tous les cas détectés par WA seront détectés par FD et donc $WA \subset FD$. Malheureusement FD reste une classe de décidabilité faible qui ne détecte que peu de cas de terminaison, comme on peut le voir sur cet exemple :

**Exemple 48.** *Soit le programme $P_{48} = (\mathcal{F}_{48}, \mathcal{R}_{48})$ avec :*

$$\mathcal{R}_{48} = \left\{ \begin{array}{l} r_1 : \mathtt{p}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{h}(\mathtt{X}). \\ r_2 : \mathtt{h}(\mathtt{V}) \leftarrow \mathtt{p}(\mathtt{U}, \mathtt{V}), \mathtt{q}(\mathtt{V}). \end{array} \right\}$$

*$P_{48}$ n'est pas FD car il existe un cycle dans le graphe des prédicats du programme et que l'instanciation de l'atome $\mathtt{p}(\mathtt{X}, \mathtt{Y})$ n'est pas restreinte par une position autre que celles du symbole de prédicat* h*. Par contre le programme se termine tout de même car si on applique les règles avec le fait $\mathcal{F}_{48} = \{\mathtt{h}(\mathtt{titi})\}$ l'application de la règle $r_1$ donne $\mathtt{p}(\mathtt{titi}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{titi}))$ mais la règle $r_2$ ne sera jamais appliquée car il n'existe pas d'instance de $\mathtt{q}(\mathtt{V})$ avec le symbole de skolem* sk *qui ne peut-être obtenu que par l'application de $r_1$.*

*Ce programme n'est pas détecté par FD car la position $\mathtt{r}_1[\mathtt{h}, \mathtt{X}_1]$ est bien récursive. Et la propriété FD ne permet pas de détecter que c'est la position $\mathtt{r}_2[\mathtt{q}, \mathtt{V}_1]$ qui bloque l'instanciation car n'étant pas récursive.*

### Argument Restricted

Comme vu précédemment la classe FD a ses limites en s'intéressant que localement au lien entre les positions des variables avec un domaine fini. La classe Argument Restricted (AR) est proposée l'année suivante dans [56]. Elle s'intéresse au lien entre les variables dans tout le programme. Ainsi une variable dont l'existentielle a été transmise peut être stoppée par une autre position si celle-ci ne fait pas partie du cycle. AR propose ainsi un classement des positions pour détecter les cycles. Si nous reprenons le programme $P_{48}$ n'étant pas FD on remarque que celui-ci est AR, en effet la position $\mathtt{r}_2[\mathtt{q}, \mathtt{V}_1]$ ne fait pas partie du cycle et est liée à la position $\mathtt{r}_2[\mathtt{p}, \mathtt{V}_2]$ partageant la même variable. Ceci engendre la terminaison du cycle car la variable en position $\mathtt{r}_2[\mathtt{q}, \mathtt{V}_1]$ ne peut pas être instanciée avec la même existentielle que la variable en position $\mathtt{r}_2[\mathtt{p}, \mathtt{V}_2]$.

**Définition 3.2.5** (Argument Restricted)**.** *Un programme est Argument Restricted (AR) s'il existe une fonction $\rho$ des positions vers des entiers, telle que pour toute position $\mathtt{r}[\mathtt{p}, \mathtt{X_i}]$ en tête il existe une position $\mathtt{r}[\mathtt{q}, \mathtt{X'_i}]$ du corps positif satisfaisant la condition suivante :*

$$\rho(\mathtt{r}[\mathtt{p}, \mathtt{X_i}]) - \rho(\mathtt{r}[\mathtt{q}, \mathtt{X'_i}]) \geq d(\mathtt{r}[\mathtt{p}, \mathtt{X_i}]) - d(\mathtt{r}[\mathtt{q}, \mathtt{X'_i}]).$$

*avec $d(\mathtt{r}[\mathtt{p}, \mathtt{X_i}])$ le niveau de profondeur maximum du symbole de fonction en position $\mathtt{r}[\mathtt{p}, \mathtt{X_i}]$.*

Le ranking est défini de manière à ce que l'incrémentation du rang des positions existentielles soit finie. Dans ce cas le programme est AR.

**Exemple 49.** *Soit le programme de l'exemple 48, nous pouvons établir l'argument-ranking suivant :*

$$\begin{array}{l}\rho(\mathtt{r_1}[\mathtt{h}, \mathtt{X_1}]) = 0\\ \rho(\mathtt{r_1}[\mathtt{p}, \mathtt{X_1}]) = 0\\ \rho(\mathtt{r_1}[\mathtt{p}, \mathtt{Y_2}]) = 1\\ \rho(\mathtt{r_2}[\mathtt{q}, \mathtt{V_1}]) = 0\\ \rho(\mathtt{r_2}[\mathtt{p}, \mathtt{U_1}]) = 0\\ \rho(\mathtt{r_2}[\mathtt{p}, \mathtt{V_2}]) = 1\\ \rho(\mathtt{r_2}[\mathtt{h}, \mathtt{V_1}]) = 0\end{array}$$

*Nous avons donc un argument-ranking valide car :*

$$\begin{array}{l}\rho(\mathtt{r_1}[\mathtt{p}, \mathtt{X_1}]) - \rho(\mathtt{r_1}[\mathtt{h}, \mathtt{X_1}]) \geq d(\mathtt{r_1}[\mathtt{p}, \mathtt{X_1}]) - d(\mathtt{r_1}[\mathtt{h}, \mathtt{X_1}])\\ \rho(\mathtt{r_1}[\mathtt{p}, \mathtt{Y_2}]) - \rho(\mathtt{r_1}[\mathtt{h}, \mathtt{X_1}]) \geq d(\mathtt{r_1}[\mathtt{p}, \mathtt{Y_2}]) - d(\mathtt{r_1}[\mathtt{h}, \mathtt{X_1}])\\ \rho(\mathtt{r_2}[\mathtt{h}, \mathtt{V_1}]) - \rho(\mathtt{r_2}[\mathtt{q}, \mathtt{V_1}]) \geq d(\mathtt{r_2}[\mathtt{h}, \mathtt{V_1}]) - d(\mathtt{r_2}[\mathtt{q}, \mathtt{V_1}])\end{array}$$

*nous voyons que le programme $P_{48}$ est AR grâce à la position $\mathtt{r_2}[\mathtt{q}, \mathtt{V_1}]$ qui restreint l'application de la règle $r_2$ grâce à la variable $\mathtt{V}$ partagée par la position $\mathtt{r_2}[\mathtt{p}, \mathtt{V_2}]$. Pour appliquer la règle il faudrait alors que la variable existentielle soit transmise aux deux prédicats ce qui n'est pas le cas.*

L'exemple suivant est décidable mais n'est par contre pas détecté par AR.

**Exemple 50.** *Soit le programme $P_{50} = (\mathcal{F}_{50}, \mathcal{R}_{50})$ avec :*

$$\mathcal{R}_{50} = \left\{ \begin{array}{l} r_1 : \mathtt{p}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{q}(\mathtt{Z}, \mathtt{X}).\\ r_2 : \mathtt{p}(\mathtt{Y}, \mathtt{X}) \leftarrow \mathtt{q}(\mathtt{Z}, \mathtt{X}).\\ r_3 : \mathtt{h}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{p}(\mathtt{X}, \mathtt{Y}).\\ r_4 : \mathtt{q}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{h}(\mathtt{X}, \mathtt{Y}), \mathtt{h}(\mathtt{Y}, \mathtt{X}).\end{array} \right\}$$

*$P_{50}$ n'est pas AR car il n'existe pas d'argument-ranking valide, mais cela ne l'empêche pas de se terminer. Si nous instancions le programme avec $\mathcal{F}_{50} = \{\mathtt{q}(\mathtt{titi}, \mathtt{toto})\}$, une fois les règles $r_1$ et $r_2$ appliquées on obtient $\mathtt{p}(\mathtt{toto}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}(\mathtt{toto}))$ et $\mathtt{p}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_2}}(\mathtt{toto}), \mathtt{toto})$ on peut ensuite appliquer deux fois $r_3$ ce qui donne $\mathtt{h}(\mathtt{toto}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_1}}(\mathtt{toto}))$ et $\mathtt{h}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r_2}}(\mathtt{toto}), \mathtt{toto})$ puis le programme s'arrête car les deux symboles de skolem créés ne sont pas les mêmes et donc le corps $\mathtt{h}(\mathtt{X}, \mathtt{Y}), \mathtt{h}(\mathtt{Y}, \mathtt{X})$ ne peut pas être instancié.*

Cet exemple n'est pas détecté par la classe AR car elle considère chaque cycle indépendamment, ce qui n'est pas suffisant pour déterminer que ce programme va se terminer.

### Jointly Acyclic

La classe Jointly Acyclic (JA), proposée en 2011 dans [50], est une amélioration de AR prenant en compte tous les cycles en conservant les mêmes critères. Si tous les cycles d'un programme possèdent une variable existentielle qui est restreinte alors le programme se terminera en un temps fini. Ainsi l'exemple 50 est détecté par JA.

**Définition 3.2.6** (Jointly Acyclic)**.** *Soit un ensemble de règles $\mathcal{R}$. Pour chaque règle $r$ et pour toute variable* $\mathtt{X}$*, nous avons $\Pi_X^B$ (resp. $\Pi_X^H$) l'ensemble des positions où* $\mathtt{X}$ *apparaît dans le corps (resp. la tête) d'une règle $r \in \mathcal{R}$. De plus pour chaque variable existentielle* $\mathtt{Y}$*, nous avons $\Omega_Y$ le plus petit ensemble de positions tel que :*

- $\Pi_Y^H \subseteq \Omega_Y$,
- $\Pi_X^H \subseteq \Omega_Y$ *pour toute variable* $\mathtt{X} \in fr(r)$.

*Le graphe de dépendance existentielle a pour sommets les variables existentielles tel qu'il existe un arc allant d'une variable* $\mathtt{X}$ *vers une variable* $\mathtt{Y}$ *si la règle où* $\mathtt{Y}$ *apparaît possède une variable universelle* $\mathtt{Z} \in fr(r)$ *telle que $\Pi_Z^B \subseteq \Omega_X$. L'ensemble $\mathcal{R}$ est Jointly Acyclic (JA) si son graphe de dépendance existentielle est acyclic.*

**Exemple 51.** *Soit le programme $P_{50}$ composé de l'ensemble de règles :*

$$\mathcal{R}_{50} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X,Y)} \leftarrow \mathtt{q(Z,X)}. \\ r_2 : \mathtt{p(Y,X)} \leftarrow \mathtt{q(Z,X)}. \\ r_3 : \mathtt{h(X,Y)} \leftarrow \mathtt{p(X,Y)}. \\ r_4 : \mathtt{q(X,Y)} \leftarrow \mathtt{h(X,Y)}, \mathtt{h(Y,X)}. \end{array} \right\}$$

*nous avons montré précédemment que ce programme n'était pas AR car il n'existe pas d'argument-ranking. Néanmoins on peut remarquer que les variables existentielles* $\mathtt{Y}$ *en tête des règles $r_1$ et $r_2$ font parties de deux cycles différents. Ces cycles ne sont pas liés entre eux car ils dépendent de deux variables existentielles différentes. En effet, les deux variables existentielles se transmettent mais avec deux valeurs différentes et à des places différentes. La variable existentielle de $r_1$ se transmet aux variables $X$ et $Y$ du prédicat en tête de la règle $r_4$ en même temps que la variable existentielle de la règle $r_2$, cependant ces deux variables existentielles ne sont pas les mêmes et ne permettront pas de déclencher la règle $r_4$ car elles dépendent de deux cycles indépendants. JA considère les deux cycles indépendamment tandis que AR ne fait pas la différence entre ces deux cycles.*

Ce n'est malheureusement pas suffisant pour détecter l'ensemble des cas décidables comme le montre le prochain exemple.

**Exemple 52.** *Soit le programme $P_{52} = (\mathcal{F}_{52}, \mathcal{R}_{52})$ avec :*

$$\mathcal{R}_{52} = \left\{ \begin{array}{l} r_1 : \mathtt{q(X,Y)}, \mathtt{q(Y,X)}, \mathtt{q(X,X)} \leftarrow \mathtt{p(X)}. \\ r_2 : \mathtt{h(X)} \leftarrow \mathtt{q(X,X)}. \\ r_3 : \mathtt{p(X)} \leftarrow \mathtt{h(X)}. \end{array} \right\}$$

*Ce programme n'est pas JA pourtant il s'arrête en un temps fini. Si on applique la première règle avec $\mathcal{F}_{52} = \{\mathtt{p(toto)}\}$ on crée alors trois instances de* $\mathtt{q} : \mathtt{q(toto,toto)}, \mathtt{q(toto,sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(toto))}$ *et* $\mathtt{q(sk}^{\mathtt{Y}}_{\mathtt{r_1}}\mathtt{(toto),toto)}$*. On va alors déclencher une seule fois la règle $r_2$ et instancier* $\mathtt{h(toto)}$ *pour enfin ne pas déclencher la règle $r_3$ car* $\mathtt{p(toto)}$ *existe déjà.*

Le programme $P_{52}$ paraît simple à détecter mais n'est pourtant pas détecté par JA. En effet JA ne différencie pas la provenance de chacune des variables et ne détecte pas que la règle $r_2$ ne sera appliquée que par l'instance $\mathtt{q(toto,toto)}$ et que les existentielles transmises ne permettront pas l'application de $r_2$.

### Super Weakly Acyclic

La classe Super Weakly Acyclic (SWA) est introduite dans [59] comme une version améliorée de Weak Acycilic et permet de détecter s'il existe une substitution du corps d'une règle avec les existentielles transmises en appliquant la même méthode que Jointly Acyclic mais sur un programme déjà skolémisé. Ainsi pour qu'une variable existentielle soit transmise il faut qu'il existe une substitution possible du corps avec la tête de celle qui la précède après que le programme ait été skolémisé.

**Définition 3.2.7** (Super Weakly Acyclic)**.** *Soit $\mathcal{R}$ un ensemble de règles dont aucune variable n'apparaît dans plus d'une règle. Un ensemble de positions $\Pi'$ couvre un ensemble de positions $\Pi$, si pour chaque position $\mathtt{r}[\mathtt{p}, \mathtt{X_i}] \in \Pi$ il existe une position $\mathtt{r}[\mathtt{p}', \mathtt{X}'_{\mathtt{i}'}] \in \Pi'$ avec les substitutions $\sigma$ et $\sigma'$ telles que $\sigma(\mathtt{X}) = \sigma(\mathtt{X}')$ et $i = i'$. Étant donné une règle $r$ et une variable $\mathtt{Y} \in vars(r)$, il existe trois ensembles de positions $In(\mathtt{Y})$, $Out(\mathtt{Y})$, et $Move(\mathtt{Y})$ définis comme suit :*

- *l'ensemble $In(\mathtt{Y})$ contient toutes les positions $\mathtt{r}[\mathtt{p}, \mathtt{Y_i}] \in corps(r)$ ;*
- *l'ensemble $Out(\mathtt{Y})$ contient toutes les positions $\mathtt{sk(r)}[\mathtt{p}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}}(\mathtt{t}_1, \ldots, \mathtt{t}_k)] \in tête(sk(r))$ avec $\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}}$ un symbole de skolem d'arité $k$ et $\mathtt{t}_1, \ldots, \mathtt{t}_k$ les variables de la frontière ;*
- *l'ensemble $Move(\mathtt{Y})$ est le plus petit ensemble de positions tel que :*
  - *$Out(\mathtt{Y}) \subseteq Move(\mathtt{Y})$ ;*
  - *$Out(\mathtt{Y}') \subseteq Move(\mathtt{Y})$ pour toute variable universelle $\mathtt{Y}'$ telle que $Move(\mathtt{Y})$ couvre $In(\mathtt{Y}')$.*

*Le graphe de dépendance SWA contient un sommet pour chaque règle de $\mathcal{R}$, et un arc allant d'une règle $r$ vers une règle $r'$ s'il existe une variable $\mathtt{X}' \in fr(r')$ et une variable existentielle $\mathtt{Y} \in tête(r)$ telles que $Move(\mathtt{Y})$ couvre $In(\mathtt{X}')$. L'ensemble $\mathcal{R}$ est Super Weakly Acyclic (SWA) si le graphe de dépendance SWA est acyclique.*

**Exemple 53.** *Sur l'exemple 52, la règle $r_1$ skolémisée est la règle*

$$skr_1 : \mathtt{q}(\mathtt{X}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{X})), \mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{X}), \mathtt{X}), \mathtt{q}(\mathtt{X}, \mathtt{X}) \leftarrow \mathtt{p}(\mathtt{X}).$$

*Nous pouvons voir que les atomes $\mathtt{q}(\mathtt{X}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{X}))$ et $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{X}), \mathtt{X})$ n'ont pas de substitutions possible avec le corps de $r_2$ $\mathtt{q}(\mathtt{X}, \mathtt{X})$. La règle $r_1$ ne pourra donc jamais déclencher $r_2$ en transmettant ses variables existentielles.*

Nous identifions ainsi tous les programmes se terminant après un cycle d'application des règles (i.e. chaque règle est appliquée au plus une fois pour vérifier s'il existe une variable existentielle qui se transmet via un cycle). En terme de complexité l'ensemble des classes présentées jusqu'ici s'intéressent à la détection de cycle dans un graphe de positions après avoir parcouru chaque sommet du graphe au maximum une fois ce qui peut être fait en temps polynomial (PTIME). Il existe néanmoins des programmes qui s'arrêteront après un nombre fini de tours de cycle. Ces programmes ne sont donc pas détectés par la propriété SWA. Comme dans l'exemple suivant.

**Exemple 54.** *Soit le programme $P_{54} = (\mathcal{F}_{54}, \mathcal{R}_{54})$ avec :*

$$\mathcal{R}_{54} = \left\{ \begin{array}{l} r_1 : \mathtt{p}(\mathtt{X}, \mathtt{Y}), \mathtt{q}(\mathtt{Y}) \leftarrow \mathtt{h}(\mathtt{X}). \\ r_2 : \mathtt{h}(\mathtt{X}) \leftarrow \mathtt{p}(\mathtt{X}, \mathtt{Y}), \mathtt{q}(\mathtt{Y}). \\ r_3 : \mathtt{p}(\mathtt{X}, \mathtt{Y}), \mathtt{s}(\mathtt{Y}) \leftarrow \mathtt{q}(\mathtt{X}). \\ r_4 : \mathtt{t}(\mathtt{X}) \leftarrow \mathtt{s}(\mathtt{X}). \\ r_5 : \mathtt{q}(\mathtt{X}) \leftarrow \mathtt{p}(\mathtt{X}, \mathtt{Y}), \mathtt{t}(\mathtt{Y}). \end{array} \right\}$$

*Soit un fait $\mathcal{F}_{54} = \{\mathtt{h}(\mathtt{toto})\}$, nous choisissons d'appliquer les règles de haut en bas. L'application de $r_1$ donne $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ et $\mathtt{p}(\mathtt{toto}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$. Nous appliquons ensuite $r_2$ qui*

*génère* $\mathtt{h}(\mathtt{toto})$ *mais est déjà présent dans la base de faits donc celle-ci reste inchangée. Nous déclenchons* $r_3$ *avec* $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ *et* $\mathtt{p}(\mathtt{toto}, \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ *en ajoutant* $\mathtt{s}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_3}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto})))$ *et* $\mathtt{p}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_3}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto})))$ *qui déclenche* $r_4$ *et ajoute* $\mathtt{t}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_3}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto})))$ *puis* $r_5$ *termine en ajoutant* $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ *qui n'est pas ajouté à la base de faits car déjà présent suite à l'application de* $r_3$*. Le premier cycle est terminé sans avoir pu détecter que ce programme est SWA. Pourtant celui-ci va se terminer car la prochaine application de la règle* $r_2$ *ne produira pas de nouvelle instance de* $\mathtt{h}(\mathtt{X})$ *car les instances* $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ *et* $\mathtt{p}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_3}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto})))$ *ne peuvent pas être utilisés ensemble pour appliquer la règle* $r_2$*.* $\mathtt{q}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}))$ *et* $\mathtt{p}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto}), \mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_3}(\mathtt{sk}^{\mathtt{Y}}_{\mathtt{r}_1}(\mathtt{toto})))$ *possèdent deux symboles de fonction de skolem différents, pour substituer la variable* $Y$*, non détectés par SWA car apparaissant sur le deuxième cycle d'applications.*

Ainsi il nous faut donc parcourir les règles sur plusieurs cycles d'application pour détecter la terminaison. C'est ce que propose la prochaine classe.

### Model-Summarizing Acyclicity

Model-Summarizing Acylicity (MSA) [47] est la dernière classe proposée basée sur le chaînage avant. Celle-ci propose la détection de la terminaison sur un nombre fini de tours de cycle. Le but est donc de lancer un programme et voir si une variable existentielle continue de se propager dans un cycle après un nombre fini de tours. Si tel est le cas le programme ne se terminera pas, c'est le cas pour l'exemple 54. L'inconvénient de cette classe est que la vérification qu'un programme lui appartient est $\mathtt{EXP} - \mathtt{TIME}$-membership alors que la vérification pour les classes précédentes est faite en temps polynomial.

### Acyclic Graph of Rule Dependencies

Nous nous sommes auparavant intéressé aux dépendances entre les positions des atomes dans un programme. Mais ce n'est pas le seul paramètre qui peut impacter la terminaison d'un programme en marche avant, on peut également s'intéresser aux dépendances entre les règles pour déterminer si une règle peut en déclencher une autre (et non plus un prédicat qui transmet un argument à un autre). Pour cela on peut utiliser un graphe de dépendance de règles, comme défini dans la définition 1.4.11, et rechercher un cycle dans ce graphe. En effet, un cycle dans le graphe de dépendance des règles implique que le programme peut ne pas se terminer car si l'application d'une règle engendre une chaîne d'application qui revient à cette même règle alors la chaîne va se déclencher de nouveau et ainsi de suite... Cette classe de programme, ne possédant pas de cycle dans son graphe de dépendance de règles, s'appelle acyclic Graph of Rule Dependencies [11] (aGRD).

**Exemple 55.** *Reprenons le programme de l'exemple 48. Soit le programme* $P_{48} = (\mathcal{F}_{48}, \mathcal{R}_{48})$ *avec :*

$$\mathcal{R}_{48} = \left\{ \begin{array}{l} r_1 : \mathtt{p}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{h}(\mathtt{X}). \\ r_2 : \mathtt{h}(\mathtt{V}) \leftarrow \mathtt{p}(\mathtt{U}, \mathtt{V}), \mathtt{q}(\mathtt{V}). \end{array} \right\}$$

*celui-ci n'est pas FD et pourtant il est aGRD car la règle* $r_1$ *une fois appliquée ne permet pas l'application de* $r_2$*.* $r_2$ *ne dépend donc pas de* $r_1$ *et il n'existe pas d'arc allant de* $r_1$ *vers* $r_2$ *dans le graphe de dépendance des règles de* $P_{48}$*.*

Cette classe contrairement aux classes précédentes n'est pas incluse et n'inclut aucune autre classe, ses caractéristiques étant différentes, on peut alors trouver des programmes MSA qui ne sont pas aGRD et vice-versa.

**Exemple 56.** *Soit le programme* $P_{56} = (\mathcal{F}_{56}, \mathcal{R}_{56})$ *avec :*

$$\mathcal{R}_{56} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X,Y)} \leftarrow \mathtt{p(X,X)}, \mathtt{q(X,Z)}, \mathtt{q(Y,Z)}. \\ r_2 : \mathtt{s(X,Y)} \leftarrow \mathtt{p(Z,X)}. \\ r_3 : \mathtt{q(X,Y)} \leftarrow \mathtt{s(Z,X)}. \end{array} \right\}$$

*ce programme n'est pas MSA et pourtant il est aGRD car il n'existe pas de cycle dans son graphe de dépendance des règles. En effet, il n'existe pas de dépendance de* $r_1$ *avec elle même ni de dépendance de* $r_1$ *avec* $r_3$*.*

Nous pouvons donc utiliser le graphe de dépendance des règles en parallèle de la propriété MSA pour déterminer si un programme se termine ou non pour augmenter nos chances de détection. Un des intérêts pour la classe aGRD par rapport à la classe MSA est sa complexité dont nous savons que tester si un programme appartient à la classe aGRD est *CO-NP-complet* et donc plus simple que MSA qui est $\mathtt{EXP} - \mathtt{TIME}$. En effet, pour la construction du graphe de dépendance des règles il nous faut tester les dépendances de toutes les règles deux à deux, puis nous devons simplement parcourir le graphe ainsi construit pour vérifier s'il existe un cycle ou non. Il existe cependant une propriété plus forte que MSA qui inclut aussi aGRD et permet donc de détecter la terminaison de tous les programmes présentés précédemment.

### Model-Faithful Acyclicity

La classe Model-Faithful Acyclicity (MFA) est une version améliorée de MSA prenant en compte la profondeur des symboles de fonction imbriqués. Cela a pour impact d'inclure à la fois la classe MSA (et celles qu'elle inclut) ainsi que la classe aGRD. Elle est à ce jour la classe concrète comprenant le plus de programme au sein de la classe FES. Malheureusement sa complexité est encore plus grande que celle de MSA devenant $2 - \mathtt{EXP} - \mathtt{time}$-complet à cause de son analyse en profondeur des symboles de fonction.

Nous avons donc vu toutes les classes de décidabilité connues permettant de déterminer si un pro-

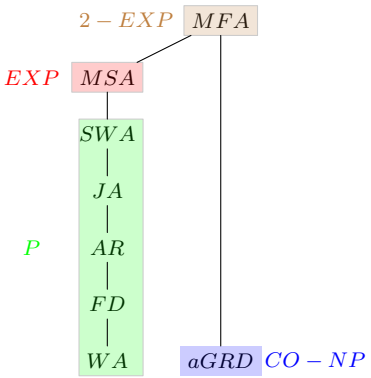



FIGURE 3.3 – Hiérarchie des classes de décidabilité FES

gramme va se terminer en un temps fini en suivant une simple marche en avant. On peut ainsi représenter l'ensemble de ces classes hiérarchiquement, comme prouvé dans [47]. On a ainsi la hiérarchie de la figure 3.3 où l'on peut voir la complexité de chaque classe allant de $P$ pour les classes de WA à SWA, jusqu'à $2 - \mathtt{EXP} - \mathtt{time}$ pour la classe MFA. Les liens représentent l'inclusion des classes, les classes possédant un lien avec une autre classe se situant en dessous incluent strictement celle-ci. On note que aGRD est incluse dans MFA mais incomparable avec toutes les autres classes.

### 3.2.2 Homogénéisation des notions d'acyclicité

Nous avons vu précédemment que les classes concrètes issues de la classe FES possèdent un rapport d'efficacité et d'inclusion $WA \subset FD \subset AR \subset JA \subset SWA \subset MSA$. Nous proposons dans cette partie de généraliser chacune des classes présentées à l'aide d'un même graphe dit de positions. Ces résultats ont été publiés dans [13]. Nous allons d'abord présenter ce graphe de positions qui permet de représenter plus d'information que les simples graphes des prédicats et de règles présentés dans l'état de l'art. Pour généraliser l'ensemble des classes nous utilisons une notion de propriété d'acyclicité plus ou moins forte en fonction de la classe représentée. L'idée est donc d'associer à chaque propriété d'acyclicité une fonction qui pour chaque position attache un sous-ensemble de positions atteignables à partir de celle-ci, selon des contraintes de propagation. Une propriété est satisfaite si aucune position existentielle ne peut être atteinte à partir d'elle même suivant une propagation.

#### Graphes de positions

Que ce soit le graphe de dépendance des règles ou le graphe de dépendance des prédicats ils connaissent tous deux des limites de représentation. Pour pallier ces limites, il a été proposé le graphe de positions qui permet de lier les dépendances entre les positions des prédicats et les dépendances entre règles. Pour cela on va chercher à représenter un graphe avec le plus de détails possible en s'intéressant à la position de chaque variable de chaque prédicat pour toutes les règles du programme. Ce graphe a néanmoins une contrainte, il est beaucoup plus imposant que les précédents mais plus précis en terme de dépendances. Il demande donc plus de temps pour être parcouru. Nous présentons ici trois variantes de ce graphe. La première est une version classique utilisée par exemple dans [47]. Nous présenterons ensuite trois extensions de ce graphe dans la prochaine section permettant d'améliorer la détection de cycles.

À l'aide des positions de la définition 3.2.3 nous définissons dans un premier temps un graphe de positions basique représentant les positions et leurs dépendances au sein d'une règle de la manière suivante :

**Définition 3.2.8** (Graphe de positions basique)**.** *Un* graphe de positions basique *noté PG (*position graph*) est un graphe orienté avec ses sommets étiquetés par les positions appartenant aux règles du programme. Soit une règle $r$, on note le graphe de positions basique associé à la règle $r$, PG($r$), qui est constitué de trois types d'arcs :*

- *Un arc positif allant d'une position* $\mathtt{r}[\mathtt{p}, \mathtt{X_i}]$ *du corps positif vers une position* $\mathtt{r}[\mathtt{q}, \mathtt{X_j}]$ *de la tête ;*
- *Un arc négatif allant d'une position* $\mathtt{r}[\mathtt{p}, \mathtt{X_i}]$ *du corps négatif vers une position* $\mathtt{r}[\mathtt{q}, \mathtt{X_j}]$ *de la tête ;*
- *Un arc existentiel allant de toutes les positions frontières du corps positif de $r$ vers la position* $\mathtt{r}[\mathtt{q}, \mathtt{Y_j}]$ *de la tête si* Y *est une variable existentielle ;*

*Soit $\mathcal{R}$ un ensemble de règles, le graphe de positions basique de $\mathcal{R}$, noté PG($\mathcal{R}$) est l'union disjointe des PG($r_i$), pour toute $r_i \in \mathcal{R}$.*

Remarque : pour le graphe de positions le corps de la règle se trouve à gauche et la tête à droite pour une meilleure lisibilité. Les variables soulignées correspondent à la position concernée dans le graphe de positions, pour un atome $\mathtt{p}(\mathtt{X}, \mathtt{Y})$ d'une règle r, la position $\mathtt{r}[\mathtt{p}, \mathtt{Y_2}]$ sera représentée sur le graphe par un sommet $\mathtt{p}(\mathtt{X}, \underline{\mathtt{Y}})$.

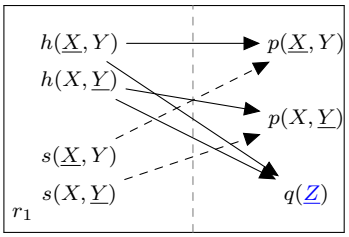
Une position existentielle $\mathtt{r}[\mathtt{a}, \mathtt{X_j}]$ d'une règle $r$ (représentée en bleue) est dite infinie s'il existe un ensemble d'atomes $F$ tel que l'application successive d'un ensemble de règles $\mathcal{R}$ dont $r$ fait partie, produit un nombre infini d'instances pour la variable X. Pour détecter les positions pouvant être

à l'origine d'un modèle infini, nous représentons la façon dont peut se « propager » une variable au sein de $\mathcal{R}$ en ajoutant des arcs à PG($\mathcal{R}$), appelés arcs de transition, qui partent des positions en tête des règles de $\mathcal{R}$ vers les positions du corps des règles du même ensemble. Ainsi si une position existentielle $\mathtt{r}[\mathtt{a}, \mathtt{X_j}]$ est infinie, il doit exister un cycle passant par $\mathtt{r}[\mathtt{a}, \mathtt{X_j}]$ dans PG($\mathcal{R}$).

**Exemple 57.** *Soit la règle :*

$$r_1 : \mathtt{p}(\mathtt{X}, \mathtt{Y}), \mathtt{q}(\mathtt{Z}) \leftarrow \mathtt{h}(\mathtt{X}, \mathtt{Y}), \mathtt{not\ s}(\mathtt{X}, \mathtt{Y}).$$

*le graphe de positions basique associé est*




Il peut exister un arc de transition selon différents critères de dépendance entre règles, nous proposons alors quatre types de graphe répondant à des critères différents. Le graphe le plus simple à considérer est le graphe de positions complet, noté $PG^F$ (pour *Full Position Graph*) dont les arcs de transition correspondent au cas où toutes les règles sont dépendantes entre elles.

**Définition 3.2.9** (Graphe de positions complet ($PG^F$))**.** *Soit $\mathcal{R}$ un ensemble de règles. $PG^F(\mathcal{R})$ est obtenu en ajoutant un arc de transition pour chaque position $\mathtt{r_1}[\mathtt{p_i}, \mathtt{X_j}]$ en tête d'une règle $r_1 \in \mathcal{R}$ vers une position $\mathtt{r_2}[\mathtt{p_{i'}}, \mathtt{X'_j}]$ du corps d'une règle $r_2$ du graphe de positions basique $PG(\mathcal{R})$.*

Les cycles de ce graphe sont en bijection avec les cycles du graphe de dépendance des prédicats défini pour Weakly Acyclic. Nous définirons par la suite trois autres types de graphes $PG^D$, $PG^U$ et $PG^{U^+}$ dont l'existence d'un arc de transition répond à d'autres critères. Nous généralisons ces graphes avec la notation $PG^{\mathcal{X}}$, que nous appelons simplement graphe de positions, pour toutes les propriétés communes aux différents graphes.

Nous allons dans un premier temps redéfinir les classes WA à SWA sur un graphe de positions à l'aide de propriétés d'acyclicité. Le $PG^F$ va nous permettre de représenter toutes les classes allant de WA jusqu'à SWA sur un unique graphe. Nous définissons pour cela des propriétés associées à des fonctions de marquage permettant la détection de cycles dans le graphe de positions d'un ensemble de règles, correspondant aux classes WA, FD, AR, JA et SWA.

**Définition 3.2.10** (Fonction de marquage)**.** *Une fonction de marquage associe à un sommet dans un graphe de positions, un sous-ensemble de ses successeurs (directs ou indirects), appelé marquage.*

**Définition 3.2.11** (Cycle marqué)**.** *Un cycle marqué pour un sommet $s$ (sachant $\mathtt{M}$ une fonction de marquage) est un cycle $\mathcal{C} \in PG^{\mathcal{X}}$ tel que $s \in \mathcal{C}$ et pour tout $s' \in \mathcal{C}$, $s'$ appartient au marquage de $s$ selon $\mathtt{M}$.*

Remarque : Évidemment, moins il existe de situations où un marquage peut se propager vers une position d'un graphe, plus la propriété d'acyclicité est forte.

**Définition 3.2.12** (Propriété d'acyclicité)**.** *Soit* $\mathtt{M}$ *une fonction de marquage et* $PG^{\mathcal{X}}$ *un type de graphe de positions. La propriété d'acyclicité* $\mathtt{PA}$ *associée à* $\mathtt{M}$ *dans* $PG^{\mathcal{X}}$*, notée* $\mathtt{PA}^{\mathcal{X}}$*, est satisfaite s'il n'existe pas de cycle marqué pour une position existentielle dans* $PG^{\mathcal{X}}$*. Soit* $\mathcal{R}$ *un ensemble de règles et* $PG^{\mathcal{X}}(\mathcal{R})$ *le graphe de positions associé, si* $\mathtt{PA}^{\mathcal{X}}$ *est satisfaite, nous pouvons aussi dire que* $PG^{\mathcal{X}}(\mathcal{R})$ *satisfait la propriété* $\mathtt{PA}$*.*

*Notation :* $\mathtt{PA}^{\mathcal{X}} \subset \mathtt{PA}^{\mathcal{X}'}$ *signifie que l'ensemble des programmes détectés par la propriété* $\mathtt{PA}^{\mathcal{X}}$ *sont aussi détectés par la propriété* $\mathtt{PA}^{\mathcal{X}'}$*.*

**Définition 3.2.13** (Propriétés de marquage)**.** *Soit* $pos$ *une position de* $PG^{F}$ *un graphe de positions, on note* $\Gamma^{+}(pos)$ *l'ensemble des successeurs (directs) de* $pos$*. Soit* $\mathtt{r}[\mathtt{p}, \mathtt{X_j}]$ *une position, on note* $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$ *le marquage issu de la position* $\mathtt{r}[\mathtt{p}, \mathtt{X_j}]$*. Nous pouvons définir trois conditions permettant de faciliter la définition des propriétés d'acyclicité qui sont les suivantes :*

- *P1 :* $\Gamma^{+}(\mathtt{r}[\mathtt{p}, \mathtt{X_j}]) \subseteq M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$*.*
- *P2 : Pour toute position* $\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}] \in M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$ *telle que* $\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}]$ *apparaît en tête de* $r'$ *alors* $\Gamma^{+}(\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}]) \subseteq M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$*.*
- *P3 : Pour toute variable* $\mathtt{X}'$ *appartenant au corps d'une règle, telle que pour toute position* $\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}]$*, il existe* $\mathtt{r}''[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}] \in M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$ *telle que* $\Gamma^{+}(\mathtt{X}') \subseteq M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$*, où* $\Gamma^{+}(\mathtt{X}')$ *est l'union de tous les* $\Gamma^{+}(pos)$*, où* $pos$ *est une position d'atome dans laquelle apparaît* $\mathtt{X}$*.*

*P1 assure que le marquage d'un sommet inclut tous ses successeurs. P2 assure que le marquage inclut tous les successeurs de tous les sommets marqués en tête de règle. P3 assure que pour chaque variable de la frontière d'une règle telle que toutes les positions des prédicats où elle apparaît sont marquées, le marquage inclut tous ses successeurs*

Nous allons maintenant redéfinir les classes concrètes à l'aide du $PG^{F}$ et des propriétés de marquage associées.

### Weakly acyclic

Nous commençons par la classe la plus faible WA. Cette classe peut être détectée facilement à l'aide d'un $PG^{\mathcal{X}}$ si on utilise le marquage suivant :

**Définition 3.2.14.** *Un marquage* $\mathtt{M}$ *sur un graphe de positions* $PG^{\mathcal{X}}$ *est un marquage WA si pour une position* $\mathtt{r}[\mathtt{p}, \mathtt{X_j}] \in PG^{\mathcal{X}}$*,* $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$ *est le plus petit ensemble tel que :*

- *P1 est satisfait et*
- *pour toute position* $\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}] \in M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$*,* $\Gamma^{+}(\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}]) \subseteq M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$*.*

Ce qui correspond à ce que pour chaque position marquée le marquage se transmet à tous les successeurs directs ou indirects.

**Proposition 3.2.1.** *Un ensemble de règles* $\mathcal{R}$ *est WA si et seulement si* $PG^{F}(\mathcal{R})$ *satisfait la propriété associée au marquage WA.*

**Preuve 3.2.1.** *Soit* $\mathcal{R}$ *un ensemble de règles. D'après la définition 3.2.2, si* $\mathcal{R}$ *n'est pas WA, alors il existe un cycle dans le graphe de positions passant par un arc existentiel. Soit la position* $\mathtt{r}[\mathtt{p}, \mathtt{Y_i}]$ *par laquelle l'arc se termine, et* $Y$ *une variable existentielle. Soit* $\mathtt{M}$ *le marquage WA d'une position existentielle* $\mathtt{r}'[\mathtt{p}, \mathtt{Y_j}]$*. P1 assure que les successeurs de* $\mathtt{r}'[\mathtt{p}, \mathtt{Y_j}]$ *sont marqués. La fonction de marquage va marquer l'ensemble des positions présentes sur son chemin. D'après les définitions 3.2.8 et 3.2.9*

*du graphe de positions chaque cycle du graphe des symboles de prédicat correspond à un ensemble de cycles dans le $PG^F$. Étant donné que la position $\mathtt{r}[\mathtt{p}, \mathtt{Y_i}]$ appartient à un cycle alors cette position sera aussi marquée par la fonction de marquage dans le $PG^F$. Donc $PG^F(\mathcal{R})$ ne satisfait pas la propriété d'acyclicité du marquage WA.*
*Considérons maintenant le cas où $\mathcal{R}$ est WA, alors il n'existe pas de cycle passant par un arc existentiel dans le graphe de positions de $\mathcal{R}$. Toujours d'après les définitions 3.2.8 et 3.2.9 du graphe de positions nous savons qu'à chaque cycle du $PG^F(\mathcal{R})$ correspond un cycle dans le graphe des symboles de prédicat, si $\mathcal{R}$ est WA alors il n'y a pas de cycle dans le $PG^F$ passant par une position existentielle.*

**Exemple 58.** *Soit le programme de l'exemple 44 constitué de l'ensemble de règles :*

$$\mathcal{R}_{44} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X), q(Y)} \leftarrow \mathtt{h(X)}. \\ r_2 : \mathtt{h(U)} \leftarrow \mathtt{p(U)}. \end{array} \right\}$$

*et le $PG^F$ associé en figure 3.4 : On voit clairement que la variable existentielle de la position*

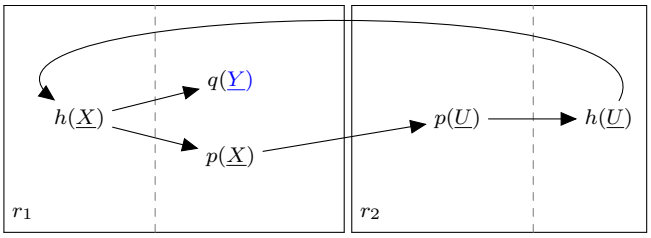



FIGURE 3.4 – Marquage WA sur le $PG^F$ du programme $P_{44}$

$\mathtt{r_1}[\mathtt{q}, \mathtt{Y_1}]$ *en bleu sur le $PG^F$ ne peut pas être transmise à une position de la règle $r_2$ et qu'il n'existe donc pas de cycle marqué dans le $PG^F$. $P_{44}$ est donc Weakly Acyclic.*

### Finite Domain

On peut aussi détecter la propriété FD en représentant le marquage associé sur notre graphe de positions $PG^{\mathcal{X}}$ à l'aide de la propriété suivante :

**Définition 3.2.15.** *Un marquage $\mathtt{M}$ sur un graphe de positions $PG^{\mathcal{X}}$ est un marquage FD si pour une position $\mathtt{r}[\mathtt{p}, \mathtt{X_j}] \in PG^{\mathcal{X}}$, $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$ est le plus petit ensemble tel que :*

- *P1 et P3 sont satisfaites, et*
- *pour toute position $\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}] \in M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$, $\Gamma^+(\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}]) \backslash \{\mathtt{r}[\mathtt{p}, \mathtt{X_j}]\} \subseteq M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$.*

Ce qui correspond à ce que pour chaque position marquée par un marquage $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$, le marquage se transmet à tous les successeurs directs ou indirects sauf si le successeur est la position $\mathtt{r}[\mathtt{p}, \mathtt{X_j}]$ elle même, dans ce cas il faut que toutes les positions dont $\mathtt{r}[\mathtt{p}, \mathtt{X_j}]$ est le successeur soient marquées par $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$.

**Proposition 3.2.2.** *Un ensemble de règles $\mathcal{R}$ est FD si et seulement si $PG^F$ satisfait la propriété associée au marquage FD.*

**Preuve 3.2.2.** *Soit $\mathcal{R}$ un ensemble de règles FD, alors pour chaque position existentielle $\mathtt{r}[\mathtt{p}, \mathtt{Y_i}]$ il existe une position $\mathtt{r}[\mathtt{p}, \mathtt{X_i}]$ pour chaque variable de la frontière dans le graphe de positions telle que $\mathtt{r}[\mathtt{p}, \mathtt{X_i}]$ n'appartient pas à un cycle. D'après la définition 3.2.4 si une position existentielle de la tête de la règle dépend d'une position avec un domaine fini alors le programme est FD, cette position*

*correspond tout simplement à une position* $\mathtt{r}[\mathtt{p}, \mathtt{X_i}]$ *n'appartenant à aucun cycle dans notre* $PG^F$. *Si une position n'appartient à aucun cycle alors son domaine est forcément fini et peut restreindre l'instanciation d'une variable existentielle. Étant donné le* $PG^F(\mathcal{R})$, *la condition P3 assure que* $\mathcal{R}$ *est FD.*

**Exemple 59.** *Représentons maintenant le marquage FD sur notre* $PG^F$. *Soit le programme de l'exemple 45 constitué de l'ensemble de règles :*

$$\mathcal{R}_{45} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X, Y)} \leftarrow \mathtt{h(X), q(X).} \\ r_2 : \mathtt{h(V)} \leftarrow \mathtt{p(U, V).} \end{array} \right\}$$

*et le* $PG^F$ *associé avec le marquage WA en figure 3.5 et le marquage FD en figure 3.6. En bleu*

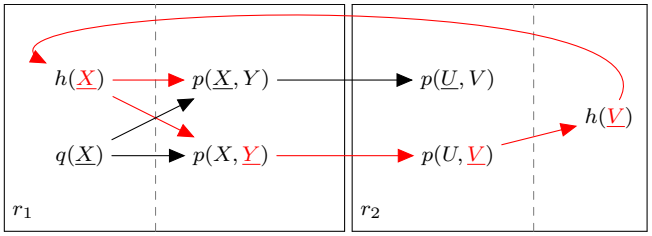



FIGURE 3.5 – Marquage WA sur le $PG^F$ du programme $P_{45}$

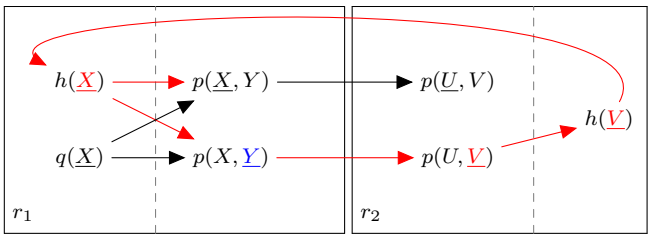



FIGURE 3.6 – Marquage FD sur le $PG^F$ du programme $P_{45}$

*sont représentées les variables existentielles non-marquées et en rouge les variables marquées par la fonction de marquage. On peut observer sur le graphe que* $P_{45}$ *est FD et non WA car le marquage WA crée un cycle (*$\mathtt{r_1}[\mathtt{p}, \mathtt{Y_2}], \mathtt{r_2}[\mathtt{p}, \mathtt{V_2}], \mathtt{r_2}[\mathtt{h}, \mathtt{V_1}], \mathtt{r_1}[\mathtt{h}, \mathtt{X_1}], \mathtt{r_1}[\mathtt{p}, \mathtt{Y_2}]$*) sur la position* $\mathtt{r_1}[\mathtt{p}, \mathtt{Y_2}]$ *par contre le marquage FD s'arrête à la position* $\mathtt{r_1}[\mathtt{h}, \mathtt{X_1}]$ *et n'est pas transmise car la position* $\mathtt{r_1}[\mathtt{q}, \mathtt{X_1}]$ *n'est pas marquée donc il existe un prédécesseur de la position* $\mathtt{r_1}[\mathtt{p}, \mathtt{Y_2}]$ *qui n'est pas marqué.*

### Argument Restricted

On peut aussi détecter la propriété AR en représentant le marquage associé sur notre graphe de positions $PG^{\mathcal{X}}$ à l'aide de la propriété suivante :

**Définition 3.2.16.** *Un marquage* $\mathtt{M}$ *sur un graphe de positions* $PG^{\mathcal{X}}$ *est un marquage AR si pour une position* $\mathtt{r}[\mathtt{p}, \mathtt{X_j}] \in PG^{\mathcal{X}}$, $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$ *est le plus petit ensemble tel que :*

- *P1, P2 et P3 sont satisfaites, et*
- *pour toute position existentielle* $\mathtt{r'}[\mathtt{p'}, \mathtt{X'_{j'}}]$, $\Gamma^+(\mathtt{r'}[\mathtt{p'}, \mathtt{X'_{j'}}]) \subseteq M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$.

Ce qui correspond à ce que pour chaque position marquée par un marquage $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$, le marquage se transmet à tous les successeurs directs ou indirects sauf pour le successeur d'une position $\mathtt{r'}[\mathtt{p'}, \mathtt{X'_{j'}}]$ en tête, dans ce cas il faut que toutes les positions dont $\mathtt{r'}[\mathtt{p'}, \mathtt{X'_{j'}}]$ est le successeur soient marquées par $M(\mathtt{r}[\mathtt{p}, \mathtt{X_j}])$.

**Proposition 3.2.3.** *Un ensemble de règles $\mathcal{R}$ est AR si et seulement si $PG^F$ satisfait la propriété associée au marquage AR.*

**Preuve 3.2.3.** *Soit $\mathcal{R}$ un ensemble de règles AR, alors il existe un ranking sur les arguments tel que pour chaque règle le rang d'une variable existentielle est strictement supérieur à celui des rangs de chaque variable de la frontière du corps et le rang d'une variable de la frontière en tête doit être supérieur ou égal à celui de cette variable dans le corps. D'après la définition* **??***, le processus de marquage est équivalent au ranking, chaque fois qu'un sommet est marqué le rang du terme est incrémenté. S'il existe un cycle AR dans le $PG^F(\mathcal{R})$ cela signifie qu'il existe au moins un rang ne respectant pas les conditions précitées. Le processus de marquage peut être une façon de calculer l'argument ranking.*

**Exemple 60.** *Soit le programme de l'exemple 48 constitué de l'ensemble de règles :*

$$\mathcal{R}_{48} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X,Y)} \leftarrow \mathtt{h(X)}. \\ r_2 : \mathtt{h(V)} \leftarrow \mathtt{p(U,V)}, \mathtt{q(V)}. \end{array} \right\}$$

*et le $PG^F$ associé avec le marquage FD en figure 3.7 et le marquage AR en figure 3.8 : Nous*

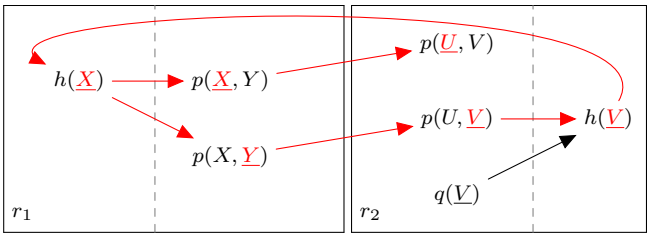



FIGURE 3.7 – Marquage FD sur le $PG^F$ du programme $P_{48}$

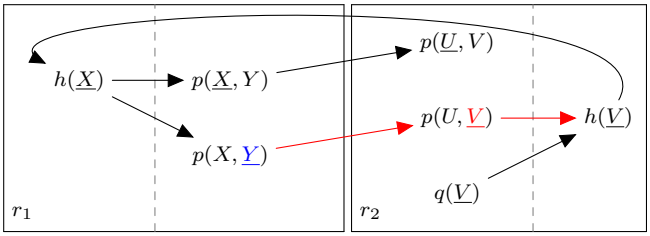



FIGURE 3.8 – Marquage AR sur le $PG^F$ du programme $P_{48}$

*pouvons observer sur le $PG^F$ que $P_{48}$ est AR mais non FD car le marquage FD crée un cycle ($\mathtt{r_1[p,Y_2]}$,$\mathtt{r_2[p,V_2]}$,$\mathtt{r_2[h,V_1]}$, $\mathtt{r_1[h,X_1]}$, $\mathtt{r_1[p,Y_2]}$) sur la position $\mathtt{r_1[p,Y_2]}$, par contre le marquage AR s'arrête à la position $\mathtt{r_2[p,V_2]}$ et n'est pas transmise car la position $\mathtt{r_2[q,V_1]}$ n'est pas marquée donc il existe un prédécesseur de la position $\mathtt{r_2[h,V_1]}$ qui n'est pas marqué.*

### Jointly Acyclic

Nous pouvons détecter la propriété JA en représentant le marquage associé sur notre graphe de positions $PG^{\mathcal{X}}$ à l'aide de la propriété suivante :

**Définition 3.2.17.** *Un marquage $\mathtt{M}$ sur un graphe de positions $PG^{\mathcal{X}}$ est un marquage JA si pour une position $\mathtt{r[p,X_j]} \in PG^{\mathcal{X}}$, $M(\mathtt{r[p,X_j]})$ est le plus petit ensemble tel que P1, P2 et P3 sont satisfaites.*

Ce qui correspond au même marquage que AR mais en prenant chaque marquage indépendamment. Ainsi il faut que tous les prédécesseurs soient marqués par la même variable pour qu'une position soit marquée. Si deux positions sont marquées par deux variables existentielles différentes alors le successeur ne sera pas marqué.

**Proposition 3.2.4.** *Un ensemble de règles $\mathcal{R}$ est JA si et seulement si $PG^F$ satisfait la propriété associée au marquage JA.*

**Preuve 3.2.4.** *La définition de la fonction de marquage JA est la même que la définition 3.2.6 initiale de la propriété JA. L'ensemble $Move$ des positions est le même que l'ensemble des positions marquées.*

**Exemple 61.** *Soit le programme de l'exemple 50 constitué de l'ensemble de règles :*

$$\mathcal{R}_{50} = \left\{ \begin{array}{ll} r_1 : \mathtt{p(X,Y)} \leftarrow & \mathtt{q(Z,X)}. \\ r_2 : \mathtt{p(Y,X)} \leftarrow & \mathtt{q(Z,X)}. \\ r_3 : \mathtt{h(X,Y)} \leftarrow & \mathtt{p(X,Y)}. \\ r_4 : \mathtt{q(X,Y)} \leftarrow & \mathtt{h(X,Y), h(Y,X)}. \end{array} \right\}$$

*et le $PG^F$ associé avec le marquage JA en figure 3.9, en rouge le marquage pour la position $\mathtt{r_2[p,Y_1]}$ et en orange pour la position $\mathtt{r_1[p,Y_2]}$ :*

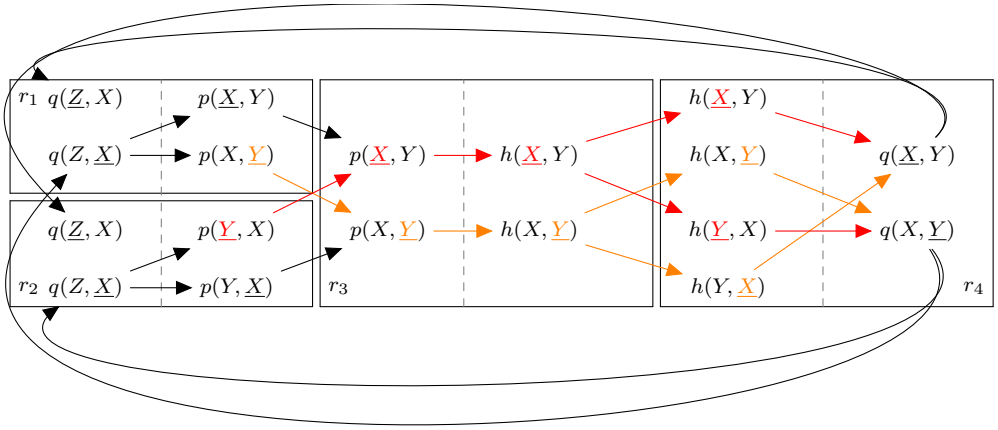



FIGURE 3.9 – Marquage JA sur le $PG^F$ du programme $P_{50}$

*Nous pouvons voir que les prédécesseurs des positions $\mathtt{r_4[q,X_1]}$ et $\mathtt{r_4[q,Y_2]}$ sont marqués par deux variables existentielles différentes, ici représentées en rouge et orange, elles ne peuvent donc pas être marquées. Le marquage AR aurait par contre créé deux cycles $C_1$ et $C_2$ en continuant la propagation sur les positions $\mathtt{r_4[q,X_1]}$ et $\mathtt{r_4[q,Y_2]}$.*
*$C_1$ = ($\mathtt{r_1[p,Y_2]}, \mathtt{r_3[p,Y_2]}, \mathtt{r_3[h,Y_2]}, \mathtt{r_4[h_2,X_2]}, \mathtt{r_4[q,X_1]}, \mathtt{r_1[q,X_2]}, \mathtt{r_1[p,Y_2]}$), correspondant au chemin en orange, et*
*$C_2$ = ($\mathtt{r_2[p,Y_1]}, \mathtt{r_3[p,X_1]}, \mathtt{r_3[h,X_1]}, \mathtt{r_4[h_2,Y_1]}, \mathtt{r_4[q,Y_2]}, \mathtt{r_2[q,X_2]}, \mathtt{r_2[p,Y_1]}$), correspondant au chemin en rouge.*

### Super Weakly Acyclic

Nous pouvons détecter la propriété SWA en représentant le marquage associé sur notre graphe de positions $PG^{\mathcal{X}}$ à l'aide de la propriété suivante :

**Définition 3.2.18.** *Un marquage $\mathtt{M}$ sur un graphe de positions $PG^{\mathcal{X}}$ est un marquage SWA si pour une position $\mathtt{r[p,X_j]} \in PG^{\mathcal{X}}$, $M(\mathtt{r[p,X_j]})$ est le plus petit ensemble tel que :*

- *P1 et P3 sont satisfaites,*
- *pour toute position existentielle* $\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}] \in M(\mathtt{r}[\mathtt{p}, \mathtt{X}_\mathtt{j}])$ *apparaissant dans une tête de règle,* $\{\mathtt{r}''[\mathtt{p}'', \mathtt{X}''_{\mathtt{j}''}] \in \Gamma^+(\mathtt{r}'[\mathtt{p}', \mathtt{X}'_{\mathtt{j}'}])\} \subseteq M(\mathtt{r}[\mathtt{p}, \mathtt{X}_\mathtt{j}])$.

Ce qui correspond à ce que pour chaque position marquée par un marquage $M(\mathtt{r}[\mathtt{p}, \mathtt{X}_\mathtt{j}])$, le marquage se transmet de la même manière que le marquage JA en ajoutant le fait que pour une position $\mathtt{r}[\mathtt{p}, \mathtt{X}_\mathtt{j}]$ d'un atome $\mathtt{a}$ en tête, le marquage se transmet uniquement si une instance de $\mathtt{a}$ est une instance de $\mathtt{a}'$ son successeur.

**Proposition 3.2.5.** *Un ensemble de règles* $\mathcal{R}$ *est SWA si et seulement si* $PG^F$ *satisfait la propriété associée au marquage SWA.*

**Preuve 3.2.5.** *La définition de la propriété SWA est légèrement différente dans le papier original, mais il est possible de la représenter de la même manière en utilisant les* $Move$ *comme dans JA. D'après la définition 3.2.1, il est possible de définir le marquage de la même manière que celle utilisée par les* $Move$ *pour JA en ajoutant le fait que pour une position* $\mathtt{r}[\mathtt{p}, \mathtt{X}_\mathtt{j}]$ *en tête d'un atome* $\mathtt{a}$*, le marquage se transmet uniquement si une instance de* $\mathtt{a}$ *est une instance de* $\mathtt{a}'$ *son successeur. Ainsi la définition de SWA est identique au marquage de la définition 3.2.18.*

**Exemple 62.** *Soit le programme de l'exemple 52 constitué de l'ensemble de règles :*

$$\mathcal{R}_{52} = \left\{ \begin{array}{l} r_1 : \mathtt{q(X,Y)}, \mathtt{q(Y,X)}, \mathtt{q(X,X)} \leftarrow \mathtt{p(X)}. \\ r_2 : \mathtt{h(X)} \leftarrow \mathtt{q(X,X)}. \\ r_3 : \mathtt{p(X)} \leftarrow \mathtt{h(X)}. \end{array} \right\}$$

*et le* $PG^F$ *associé avec le marquage SWA représenté en figure 3.10. Nous observons que les mar-*

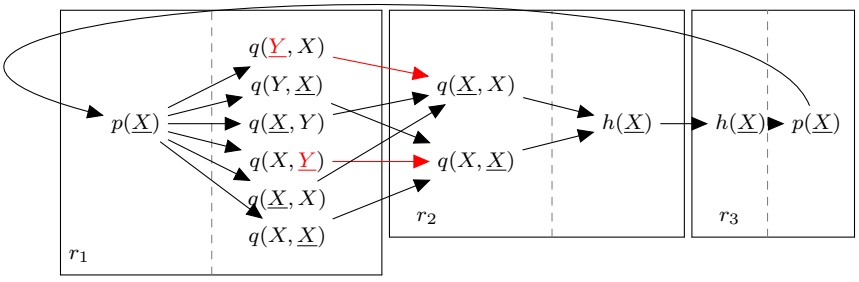



FIGURE 3.10 – Marquage SWA sur le $PG^F$ du programme $P_{52}$

*quages effectués par les deux variables existentielles ne se font pas sur la même position et que les atomes* $\mathtt{q(Y,X)}$ *et* $\mathtt{q(X,Y)}$ *de* $r_1$ *ne peuvent pas s'unifier avec* $\mathtt{q(X,X)}$ *de* $r_2$ *(les instances du symbole de prédicat* $\mathtt{q}$ *crées par la règle* $r_1$ *ne sont pas compatibles avec le corps de la règle* $r_2$*). Nous avons* $\mathtt{q(Y,X)}$ *et* $\mathtt{q(X,Y)}$ *en tête de* $r_1$ *et* $\mathtt{q(X,X)}$ *dans le corps de* $r_2$*, une fois* $r_1$ *skolémisée les variables existentielles ne pourront plus se transmettre.*

La prochaine proposition atteste que nous pouvons associer une propriété sur le $PG^F$ pour l'ensemble des classes allant de Weakly Acyclic à Super Weakly Acyclic.

**Proposition 3.2.6.** *Un ensemble de règles* $\mathcal{R}$ *est SWA (resp. FD, AR, JA, SWA) si et seulement si* $PG^F(\mathcal{R})$ *satisfait la propriété d'acyclicité associée au marquage WA (resp. FD, AR, JA, SWA).*

**Preuve 3.2.6.** *La preuve de la proposition 3.2.6 découle des preuves 3.2.1,3.2.2, 3.2.3,3.2.4,3.2.5.*

#### Model-Summarizing Acyclicity

La propriété MSA ne peut pas être représentée par une propriété d'acyclicité sur notre graphe de positions. Néanmoins elle inclut toutes les propriétés présentées précédemment. Nous verrons par la suite que cette propriété peut tout de même être étendue.

### 3.2.3 Extension des notions d'acyclicité

Nous avons donc présenté une première hiérarchie, celle des classes FES, dont les classes peuvent-être représentées sur le $PG^F$. Celui-ci ne permet pour le moment que de représenter les dépendances entre les positions du programme. Nous avons vu dans un premier temps une homogénéisation des classes connues et nous souhaitons maintenant étendre ces classes pour prendre en compte à la fois la dépendance des prédicats mais aussi celle des règles. Nous avons donc un lien manquant avec la classe aGRD qui est incomparable avec les autres classes FES. Nous proposons maintenant d'étendre ces classes à la classe aGRD en définissant un nouveau graphe de positions prenant en compte la dépendance des règles. Puis nous proposerons d'aller encore plus loin en proposant une extension utilisant les unificateurs. Nous pourrons ainsi définir les propriétés d'acyclicité sur ces nouveaux graphes et définir une nouvelle hiérarchie étendue.

#### Graphe de positions avec dépendance des règles

Nous avons vu précédemment que les classes WA, FD, AR, JA, SWA et MSA ne détectent pas les programmes décidables de la classe aGRD. Néanmoins toutes ces premières sont représentables sur le $PG^F$ à l'aide de propriétés de marquage. Mais la classe aGRD est elle aussi représentable sur un graphe de positions et nous savons qu'il est possible de combiner une propriété d'acyclicité avec la classe aGRD. Nous allons donc présenter ce nouveau graphe de positions permettant de représenter aGRD ainsi que proposer une extension des propriétés d'acyclicité sur ce nouveau graphe. Ces résultats proviennent toujours de [13].

**Définition 3.2.19** (Graphe de positions avec dépendance des règles $(PG^D)$)**.** *Soit $\mathcal{R}$ un ensemble de règles. $PG^D(\mathcal{R})$ est obtenu à partir de $PG^F(\mathcal{R})$ en ne conservant que les arcs de transition d'une position $\mathtt{r_i[p_i, X_j]}$ en tête d'une règle $r_1 \in \mathcal{R}$ vers une position $\mathtt{r_2[p_{i'}, X_j]}$ du corps d'une règle $r_2$ tels qu'il existe un arc allant de $r_1$ vers $r_2$ dans le graphe de dépendance des règles de l'ensemble $\mathcal{R}$ .*

**Exemple 63.** *Soit le programme de l'exemple 48 constitué de l'ensemble de règles :*

$$\mathcal{R}_{48} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X,Y)} \leftarrow \mathtt{h(X)}. \\ r_2 : \mathtt{h(V)} \leftarrow \mathtt{p(U,V)}, \mathtt{q(V)}. \end{array} \right\}$$

*Nous construisons un $PG^D$ associé à cet ensemble de règle en figure 3.11. Les arcs appartenant au $PG^F$ et n'appartenant pas au $PG^D$ sont représentés en pointillés. Nous avons vu précédemment que ce programme est AR mais n'est pas FD, il existe donc un cycle marqué dans le $PG^F$ en utilisant le marquage FD. Il est pourtant possible de déterminer que ce programme est décidable sans avoir à utiliser la propriété AR étant donné qu'il est aGRD. En effet, nous remarquons sur la figure 3.11 que le cycle correspondant à la propriété FD sur le $PG^F$ n'en est pas un sur le $PG^D$ car les arcs allant de $r_1$ vers $r_2$ n'existent pas dans le $PG^D$.*

Nous proposons alors une extension des classes WA, FD, AR, JA, SWA et MSA à l'aide du graphe $PG^D$.

**Proposition 3.2.7.** *Soit $\mathtt{PA}$ une propriété d'acyclicité définie sur le $PG^F$, il existe alors une propriété étendue $\mathtt{PA}^D$ qui correspond à la propriété $\mathtt{PA}$ définie sur le $PG^D$. On a alors $\mathtt{PA} \subset \mathtt{PA}^D$.*

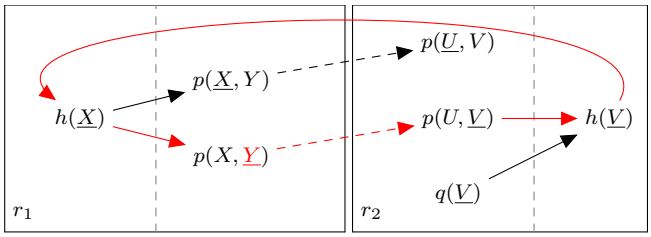



FIGURE 3.11 – Graphe de positions avec dépendance des règles AR sur le $PG^F$ du programme $P_{48}$

**Preuve 3.2.7.** *Soit* PA *une propriété d'acyclicité définie sur le $PG^F$, d'après la définition 3.2.19 le $PG^D$ correspond au $PG^F$ auquel nous avons supprimé les arcs de transition pour les règles n'ayant pas de dépendance dans le graphe de dépendance des règles. Ainsi un cycle peut être interrompu par la suppression d'un arc de transition en supprimant une propagation du marquage de la propriété* PA*. Le $PG^D$ n'ajoute aucun arc et ne peut donc créer aucun cycle supplémentaire, ce qui signifie qu'un programme* PA *est aussi* $\texttt{PA}^D$*. Nous avons donc* $\texttt{PA} \subset \texttt{PA}^D$*.*

**Exemple 64.** *Soit le programme $P_{56} = (\mathcal{F}_{56}, \mathcal{R}_{56})$ avec l'ensemble de règles suivant :*

$$\mathcal{R}_{56} = \left\{ \begin{array}{l} r_1 : \mathtt{p(X,Y)} \leftarrow \mathtt{p(X,X)}, \mathtt{q(X,Z)}, \mathtt{q(Y,Z)}. \\ r_2 : \mathtt{s(X,Y)} \leftarrow \mathtt{p(Z,X)}. \\ r_3 : \mathtt{q(X,Y)} \leftarrow \mathtt{s(Z,X)}. \end{array} \right\}$$

*Cet ensemble de règles n'est pas MSA mais il est aGRD. Nous allons donc représenter le $PG^F$ correspondant en figure 3.12 et le transformer en $PG^D$, en supprimant des arcs, afin que l'on puisse déterminer sur un seul graphe la propriété SWA et aGRD. Les flèches pointillées représentent les arcs*

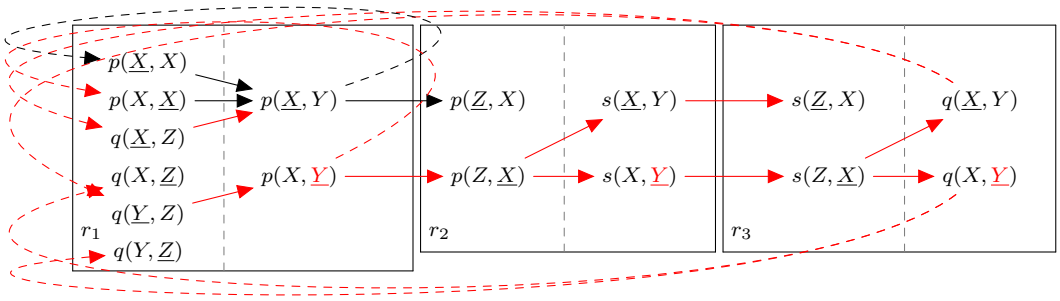



FIGURE 3.12 – Graphe de positions avec dépendance des règles SWA sur le $PG^F$ du programme $P_{56}$

*présents dans le $PG^F$ mais absents dans le $PG^D$. Ce programme qui n'était pas SWA avec le $PG^F$ l'est sur le $PG^D$. On peut donc dire que ce programme est $SWA^D$.*

On peut alors définir la hiérarchie étendue au $PG^D$ en figure 3.13. Cette hiérarchie présente les différentes inclusions entre propriétés d'acyclicités, avec les propriétés en haut à droite les propriétés qui détectent le plus de programmes décidables et à l'inverse en bas à gauche les propriétés qui détectent le moins de programmes décidables. Notons que la classe aGRD est incomparable avec la classe WA. Cette hiérarchie indique aussi les temps nécessaires à la vérification qu'un programme appartient à l'une de ces classes. Ainsi les classes originales allant de WA à SWA sont vérifiables en temps polynomial tandis que les classes étendues au graphe de dépendance des règles appartiennent à la classe de complexité CO-NP. Plus haut dans la hiérarchie nous avons MSA qui reste $EXP{-}TIME$

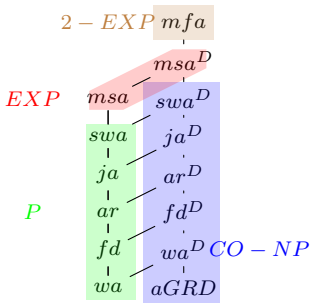



FIGURE 3.13 – Hiérarchie des classes FES étendues avec dépendance des règles

en étant étendue au graphe de dépendance des règles, et MFA qui est $2 - EXP - TIME$ et qui ne bénéficie pas d'amélioration grâce au graphe de dépendance des règles.

Nous savons maintenant qu'il est possible de combiner les classes de décidabilité basées sur la dépendance des positions avec la dépendance des règles en supprimant dans le graphe de positions les arcs entre les règles n'ayant pas de dépendance. Mais il est possible d'aller encore plus loin en s'intéressant aux unificateurs.

**Graphe de positions avec unificateur** Nous pouvons encore améliorer cette hiérarchie en analysant les unificateurs par pièce existant entre le corps d'une règle avec la tête d'une autre. Cette propriété d'unificateur par pièce renforce celle de la dépendance entre règles, ainsi il est possible de supprimer des arcs supplémentaires dans le graphe de positions d'un programme. On définit pour cela la notion de règle agglomérée, ainsi une règle agglomérée associée avec $(r_i, r_j)$ rassemble les informations de tous les unificateurs par pièce sélectionnés sur un chemin allant de $r_i$ à un prédécesseur de $r_j$. Le principe de règle agglomérée peut être retrouvé dans [16], c'est une redéfinition de la réécriture de séquence proposée dans [14]. Ainsi une règle agglomérée est une réécriture d'une séquence de règles formant un chemin dans le graphe de dépendance des règles. De manière non formelle, si nous avons deux règles $r_i$ et $r_j$ de l'ensemble $\mathcal{R}$ et $C$ un ensemble non-vide de chemins allant de $r_i$ aux prédécesseurs directs de $r_j$ dans le graphe de dépendance des règles de $\mathcal{R}$. Pour chaque chemin $c \in C$ nous pouvons associer une règle $r^c$ correspondant à la règle $r_i$ réécrite en conservant les informations de ses prédécesseurs suivant les unificateurs utilisés. Ainsi $r^c$ contient toute les informations permettant l'application des règles de la séquence $(r_i, \ldots, r_j)$ à partir des unificateurs utilisés entre chaque règle de la séquence.

**Exemple 65** (Règle agglomérée)**.** *Soit le programme $P_{65} = (\mathcal{F}_{65}, \mathcal{R}_{65})$ avec l'ensemble de règles suivant :*

$$\mathcal{R}_{65} = \left\{ \begin{array}{l} r_1 : \mathtt{q}(\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{p}(\mathtt{X},\mathtt{Y}). \\ r_2 : \mathtt{t}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{q}(\mathtt{X},\mathtt{Y}). \\ r_3 : \mathtt{p}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{t}(\mathtt{X},\mathtt{Y}). \end{array} \right\}$$

*Soit le chemin $(r_1, r_2, r_3)$, nous avons la règle agglomérée suivante :*

$$r_1^3 : \mathtt{p}(\mathtt{Y},\mathtt{Z}), \mathtt{t}(\mathtt{Y},\mathtt{Z}), \mathtt{q}(\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{p}(\mathtt{X},\mathtt{Y}).$$

*qui récupère l'ensemble des informations des unificateurs $\mu_1$ qui unifie $r_1$ avec $r_2$ et de l'unificateur $\mu_2$ qui unifie $r_2$ avec $r_3$. L'application de la règle $r_1^3$ est équivalente à l'application de la séquence de règle $(r_1,\ r_2,\ r_3)$.*

**Définition 3.2.20** (Graphe de positions avec unificateur par pièce $PG^U$)**.** *Soit $\mathcal{R}$ un ensemble de règles. $PG^U(\mathcal{R})$ est obtenu en ne conservant que les arcs de transition d'une position $\mathtt{r_1}[\mathtt{p_i}, \mathtt{X_j}]$ en*

*tête d'une règle $r_1 \in \mathcal{R}$ vers une position $\mathtt{r_2[p_{i'}, X_j]}$ du corps d'une règle $r_2$ du graphe de positions basique $PG(\mathcal{R})$ s'il existe un unificateur par pièces $\mu$ de $\mathtt{corps}(r_j)$ avec la tête d'une règle agglomérée $r_i^k$.*

**Exemple 66.** *Soit le programme $P_{66} = (\mathcal{F}_{66}, \mathcal{R}_{66})$ avec l'ensemble de règles suivant :*

$$\mathcal{R}_{66} = \left\{ \begin{array}{l} r_1 : \mathtt{p(Z,Y), q(Y)} \leftarrow \mathtt{t(X,Y)}. \\ r_2 : \mathtt{t(V,W)} \leftarrow \mathtt{p(U,V), q(U)}. \end{array} \right\}$$

*Nous construisons le $PG^U$ associé en figure 3.14. Étant donné que le seul unificateur par pièce de $\mathtt{corps}(r_2)$ avec $\mathtt{tête}(r_1)$ unifie $\mathtt{U}$ et $\mathtt{Y}$ il n'est pas possible d'appliquer la règle $r_2$ à partir d'une application de la règle $r_1$. Ici les arcs appartenant au $PG^D$ et n'appartenant pas au $PG^U$ sont*

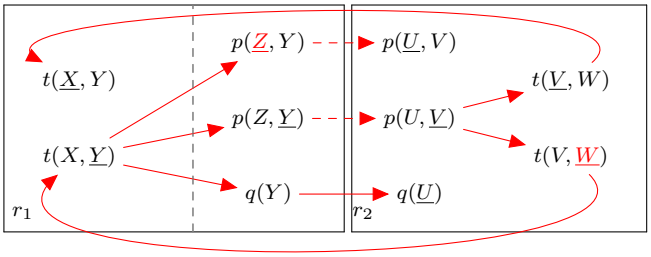



FIGURE 3.14 – Graphe de positions avec unificateurs du programme $P_{66}$

*représentés en pointillés. On voit alors que le cycle correspondant à la propriété WA sur le $PG^D$ n'en est pas un sur le $PG^U$ car les arcs allant du symbole de prédicat $\mathtt{p}$ de $r_1$ vers $r_2$ n'existent pas dans le $PG^U$.*

Nous proposons alors une extension des classes $WA^D$, $FD^D$, $AR^D$, $JA^D$, $SWA^D$ et $MSA^D$ à l'aide du graphe $PG^U$.

**Proposition 3.2.8.** *Soit $\mathtt{PA}$ une propriété d'acyclicité définie sur le $PG^D$, il existe alors une propriété étendue $\mathtt{PA}^U$ qui correspond à la propriété $\mathtt{PA}$ définie sur le $PG^U$. On a alors $\mathtt{PA}^D \subset \mathtt{PA}^U$.*

**Preuve 3.2.8.** *Soit $\mathtt{PA}^D$ une propriété d'acyclicité définie sur le $PG^D$, d'après la définition 3.2.20 le $PG^U$ correspond au $PG^F$ auquel nous avons supprimé les arcs de transitions entre les positions pour lesquelles il n'existe pas d'unificateur par pièces d'une position de la tête d'une règle et une position du corps d'une autre règle. D'après [15], la notion d'unificateur par pièce est plus forte que celle de dépendance entre règles car il n'existe pas d'unificateur par pièce entre une position de la tête d'une règle et la position du corps d'une autre règle s'il n'existe pas de dépendance entre ces deux règles. Ainsi on peut en déduire que tout programme pour lequel la propriété $\mathtt{PA}^D$ est vérifiée alors la propriété $\mathtt{PA}^U$ est aussi vérifiée. Nous avons alors $\mathtt{PA}^D \subset \mathtt{PA}^U$.*

On peut cependant encore étendre cette propriété d'unificateur en analysant plus finement les cycles marqués et les unificateurs. Cette extension n'augmente en aucun cas la complexité. Nous définissons la notion de séquence d'unificateurs incompatibles, qui assure qu'une séquence donnée d'applications de règle est impossible. En résumé, un cycle marqué pour lequel toutes les séquences d'unificateurs sont incompatibles peut être ignoré. Cet ajout, en plus d'avoir un impact sur les règles positives, prend en compte la négation par défaut pour améliorer encore plus le nombre de programmes détectés.

**Définition 3.2.21** (Unificateurs compatibles [15]). *Soit $r_1$ et $r_2$ deux règles appartenant à l'ensemble de règles $\mathcal{R}$. Un unificateur $\mu$ de $\texttt{corps}(r_2)$ avec $\texttt{tête}(r_1)$ est compatible si, pour chaque position $\texttt{r}_2[\texttt{p}, \texttt{X}_\texttt{i}]$ du corps, telle que $\texttt{r}_1[\texttt{p}, \texttt{Y}_\texttt{i}]$ est une position existentielle, le $PG^U(\mathcal{R})$ contient un chemin, d'une position dans laquelle $\texttt{Y}$ apparaît, vers $\texttt{r}_2[\texttt{p}, \texttt{X}_\texttt{i}]$, qui ne passe pas par une autre position existentielle. Autrement, $\mu$ est incompatible.*

Avec cette définition un unificateur par pièce est aussi un unificateur compatible. Nous pouvons maintenant définir la notion de règle unifiée issue d'une composition de $r_1$ et $r_2$ à partir d'un unificateur compatible, puis nous définirons la notion de séquence compatible d'unificateurs à partir de celle-ci.

**Définition 3.2.22** (Règle unifiée, séquence compatible d'unificateurs). *Les définitions de règle unifiée et de séquence compatible d'unificateurs sont les suivantes :*

- *Soit $r_1$ et $r_2$ des règles telles qu'il existe un unificateur compatible $\mu$ de $\texttt{corps}(r_2)$ avec $\texttt{tête}(r_1)$. La règle unifiée associée $r_\mu = r_1 \diamond_\mu r_2$ est définie par $\texttt{tête}(r_\mu) = \mu(\texttt{tête}(r_1)) \cup \mu(\texttt{tête}(r_2))$, et $\texttt{corps}(r_\mu) = \mu(\texttt{corps}(r_1)) \cup (\mu(\texttt{corps}(r_2) \backslash \mu(\texttt{tête}(r_1))))$ ;*
- *Soit $s = (r_1, \ldots, r_{k+1})$ une séquence de règles. Soit la règle $r_1^k = (r_1 \diamond_{\mu_1} r_2 \cdots \diamond_{\mu_k} r_{k+1})$, où, pour $1 \leq i \leq k$, $(\mu_1, \ldots, \mu_k)$ est une séquence compatible d'unificateurs de $\texttt{corps}(r_{i+1})$ avec $\texttt{tête}(r_i)$ si : (1) $\mu_1$ est un unificateur compatible de $\texttt{corps}(r_2)$ avec $\texttt{tête}(r_1)$, et (2) si $k > 0$, la séquence obtenue à partir de $s$ est une séquence compatible d'unificateurs.*

**Exemple 67** (Unificateur compatible). *Soit le programme $P_{67} = (\mathcal{F}_{67}, \mathcal{R}_{67})$ avec l'ensemble de règles suivant :*

$$\mathcal{R}_{67} = \left\{ \begin{array}{ll} r_1 : \texttt{q}(\texttt{Y}, \texttt{Z}) \leftarrow & \texttt{p}(\texttt{X}, \texttt{Y}). \\ r_2 : \texttt{t}(\texttt{X}, \texttt{Y}) \leftarrow & \texttt{q}(\texttt{X}, \texttt{Y}). \\ r_3 : \texttt{p}(\texttt{X}, \texttt{Y}) \leftarrow & \texttt{s}(\texttt{X}, \texttt{Y}), \texttt{t}(\texttt{X}, \texttt{Y}). \\ r_4 : \texttt{s}(\texttt{X}, \texttt{Y}) \leftarrow & \texttt{q}(\texttt{X}, \texttt{Y}). \end{array} \right\}$$

*Il existe un cycle de dépendances $(r_1, r_2, r_3, r_1)$ et un cycle correspondant dans le $PG^U$. Nous voulons savoir si cette séquence d'applications de règles est possible. Nous construisons alors la nouvelle règle suivante, qui est une composition faite à partir de $r_1$ et $r_2$ (définie dans la définition 3.2.21) :*

$$r_1 \diamond_\mu r_2 : \texttt{q}(\texttt{Y}, \texttt{Z}), \texttt{t}(\texttt{Y}, \texttt{Z}) \leftarrow \texttt{p}(\texttt{X}, \texttt{Y}).$$
*avec $\mu$ l'unificateur qui unifie les atomes $\texttt{q}(\texttt{Y}, \texttt{Z})$ et $\texttt{q}(\texttt{X}, \texttt{Y})$.*

*Nous pouvons voir qu'il n'existe pas d'unificateur par pièce de $r_3$ avec $r_1 \diamond_\mu r_2$, étant donné que la variable $\texttt{Y}$ de $r_3$ serait une variable différente rattachée à $Z$. En résumé $r_3$ n'est pas applicable à la suite le l'application de $r_1 \diamond_\mu r_2$. Cependant, l'atome nécessaire à l'application de $\texttt{s}(\texttt{X}, \texttt{Y})$ peut être amené par une séquence d'applications de règles $(r_1, r_4)$. Nous relaxons alors la notion d'unificateur par pièce pour prendre en compte des séquences d'applications de règles plus ou moins longues.*

**Proposition** (Unificateur compatible [13]). *Soit $r_1$ et $r_2$ deux règles et $\mu$ un unificateur de $\texttt{corps}(r_2)$ avec $\texttt{tête}(r_1)$. Si $\mu$ est incompatible, alors aucune application de $r_2$ ne peut utiliser un atome de $\mu(\texttt{tête}(r_1))$.*

Si nous reprenons l'exemple 67, nous avons la séquence $(r_1 \diamond_{\mu_1} r_2 \diamond_{\mu_2} r_3 \diamond_{\mu_3} r_1)$, avec $\mu_i$ les unificateurs compatibles correspondant. Nous pouvons ainsi étendre les propriétés d'acyclicité précédentes avec les unificateurs compatibles.

**Définition 3.2.23** (Cycles compatibles). *Soit $\texttt{PA}$ une propriété d'acyclicité, et $PG^U$ le graphe de positions avec unificateurs associé. Les cycles compatibles pour la position $\texttt{r}_1[\texttt{p}, \texttt{X}_1]$ dans le $PG^U$ sont tous les cycles $\mathcal{C}$ marqués pour la position $\texttt{r}_1[\texttt{p}, \texttt{X}_1]$ pour la propriété $\texttt{PA}$, telle qu'il existe une séquence d'unificateurs compatibles induite par $\mathcal{C}$. La propriété $\texttt{PA}^{U^+}$ est satisfaite si, pour chaque position existentielle $\texttt{r}_1[\texttt{p}, \texttt{X}_1]$, il n'existe pas de cycle compatible dans le $PG^U$ pour $\texttt{r}_1[\texttt{p}, \texttt{X}_1]$.*

**Proposition 3.2.9.** *Soit* PA *une propriété d'acyclicité définie sur le* $PG^U$*, il existe alors une propriété étendue* $\texttt{PA}^{U^+}$ *qui correspond à la propriété* PA *définie sur le* $PG^{U^+}$*. On a alors* $\texttt{PA}^U \subset \texttt{PA}^{U^+}$*.*

**Preuve 3.2.9.** *Soit* $\mathcal{R}$ *un ensemble de règles satisfaisant la propriété* $\texttt{PA}^{U^+}$*. D'après la définition 3.2.23, le* $PG^{U^+}$ *correspond au* $PG^U$ *auquel on ne conserve que les arcs de transition tels que les cycles sont compatibles. Comme pour la preuve des propositions 3.2.8 et 3.2.7, les arcs de transition sont seulement supprimés et donc les cas d'acyclicité détectés par le* $PG^U$ *sont aussi détectés par le* $PG^{U^+}$*.*

On peut alors définir la hiérarchie étendue complète en figure 3.15 :

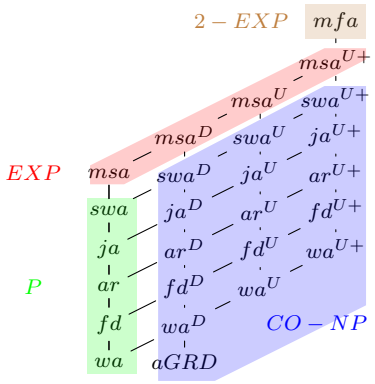



FIGURE 3.15 – Hiérarchie des classes FES étendues avec dépendance des règles

### 3.2.4 Acyclicité et non-monotonie

Il est encore possible d'améliorer la détection de cycle lorsqu'on utilise la négation par défaut comme nous l'avons proposé dans [16]. Nous savons que la négation par défaut permet de bloquer certaines règles et donc d'empêcher l'application de celles-ci. À partir de ce constat il est naturel de se demander si la négation par défaut permet la terminaison d'un programme. Pour répondre à cette question nous nous sommes intéressé à la classe FES. Nous souhaitons ainsi améliorer encore le graphe de positions pour qu'il puisse détecter des cycles qui s'arrêteraient grâce à la négation par défaut.

Tout d'abord, nous définissons la notion de *règle auto-bloquante*.

**Définition 3.2.24** (Règle auto-bloquante)**.** *Une règle*

$$r : \mathtt{h}_1, \ldots, \mathtt{h}_k \leftarrow \mathtt{b}_1^+, \ldots, \mathtt{b}_n^+, \mathtt{not}\ \mathtt{b}_1^-, \ldots, \mathtt{not}\ \mathtt{b}_m^-.$$

*est dite auto-bloquante s'il existe un atome du corps négatif* $\mathtt{b}_i^-$*,* $1 \leq i \leq m$*, tel que* $\mathtt{b}_i^- \in \{\mathtt{h}_1, \ldots, \mathtt{h}_k\}$*.*

Une telle règle ne pourra jamais être appliquée, et ne produira donc aucune instance d'atome. Nous reprenons maintenant la définition des règles unifiées et de la dépendance des règles en ajoutant le corps négatif.

**Définition 3.2.25** (Séquence compatible d'unificateurs avec négation par défaut)**.** *Soit* $r_1$ *et* $r_2$ *deux règles existentielles avec* $r_2$ *qui dépend de* $r_1$*, il existe alors* $\mu$ *un unificateur tel que nous pouvons construire une règle* $r_\mu = r_1 \diamond_\mu r_2$ *représentant la séquence d'application de règles capturée par l'unificateur* $\mu$*. Soit* $b^-$ *le corps négatif de la règle* $r_1$ *ou la règle* $r_2$*, alors* $\mu(b^-)$ *est le corps négatif de* $r_\mu$*.*

Nous pouvons ainsi étendre la notion d'auto-blocage à une séquence compatible d'unificateurs avec négation par défaut.

**Définition 3.2.26** (Séquence compatible d'unificateurs auto-bloquante)**.** *Soit un unificateur $\mu$ de $corps(r_2)$ avec $tête(r_1)$, $\mu$ est auto-bloquant si $r_1 \diamond_\mu r_2$ est auto-bloquant, et $r_2$ dépend de $r_1$ quand il existe un unificateur de $corps(r_2)$ avec $tête(r_1)$ qui n'est pas auto-bloquant.*

Les définitions 3.2.25 et 3.2.26 correspondent à la dépendance positive définie dans [58]. Dans cet article il est aussi défini qu'un ensemble de règles existentielles non-monotones est $R - acyclic$ s'il n'existe pas de cycle de dépendance positive incluant une variable existentielle. Si nous reprenons notre $PG^D$ alors nous pouvons représenter la notion de $R - acylicity$ qui est plus forte que la propriété aGRD mais moins forte que $WA^D$ étant donné que l'on renforce la propriété des cycles sur les dépendances entre règles mais que les composantes fortement connexes sont déjà prises en compte par WA.

En considérant la dépendance étendue au corps négatif, nous pouvons étendre les précédents résultats obtenus pour le $PG^D$ et le $PG^U$ ainsi que les classes $\mathtt{PA}^{U+}$ en considérant les séquences compatibles d'unificateurs auto-bloquants. Soit $\mathcal{C}$ un cycle compatible pour une position $\mathtt{r}[\mathtt{p}, \mathtt{X_j}]$ dans le $PG^{U+}$, et $\mathcal{C}_\mu$ l'ensemble des séquences compatibles d'unificateurs provenant de $\mathcal{C}$. Une séquence $\mu_1 \dots \mu_k \in \mathcal{C}_\mu$ est dite auto-bloquante si la règle $r_1 \diamond_{\mu_1} r_2 \dots r_k \diamond_{\mu_k} r_1$ est auto-bloquante. Si toutes les séquences dans $\mathcal{C}_\mu$ sont auto-bloquantes alors $\mathcal{C}$ est auto-bloquant.

**Exemple 68.** *Soit le programme $P_{68} = (\mathcal{F}_{68}, \mathcal{R}_{68})$ avec l'ensemble de règles suivant :*

$$\mathcal{R}_{68} = \left\{ \begin{array}{l} r_1 : \mathtt{h(X,Y)} \leftarrow \mathtt{q(X)}, \mathtt{not\ p(X)}. \\ r_2 : \mathtt{s(X,Y)} \leftarrow \mathtt{h(X,Y)}. \\ r_3 : \mathtt{p(X)}, \mathtt{q(Y)} \leftarrow \mathtt{s(X,Y)}. \end{array} \right\}$$

*Le $PG^{U+}$ possède un unique cycle, avec une unique séquence compatible d'unificateurs. La règle $r_1 \diamond r_2 \diamond r_3 = (\mathtt{h(X,Y)}, \mathtt{s(X,Y)}, \mathtt{p(X)}, \mathtt{q(Y)} \leftarrow \mathtt{q(X)}, \mathtt{not\ p(X)}.)$ est auto-bloquante, et donc la séquence $r_1 \diamond r_2 \diamond r_3 \diamond r_1$ aussi. Il n'existe donc pas de cycle existentiel pour ce programme.*

**Proposition 3.2.10.** *Si, pour toute position existentielle $\mathtt{r}[\mathtt{p}, \mathtt{X_j}]$, tous les cycles compatibles pour $\mathtt{r}[\mathtt{p}, \mathtt{X_j}]$ dans le $PG^{U+}$ sont auto-bloquant, alors le calcul d'answer set se termine.*

**Preuve 3.2.10.** *Si un cycle est non-compatible ou auto-bloquant, alors aucune séquence d'applications de règles ne peut l'utiliser. En effet, si tous les cycles compatibles sont auto-bloquants, alors toutes les dérivations obtenues avec les applications des règles skolémisées sont finies. Donc, le calcul de modèle dans le cadre du skolem-chase s'arrête.*

Le dernier point important est que la complexité dans le pire des cas pour le test d'acyclicité n'augmente pas. Pour conclure sur la décidabilité en marche avant d'un programme non monotone existentiel, les cas de d'arrêt présents sur les programmes de règles existentiels restent des cas d'arrêts pour les programmes non-monotones existentiels auxquels nous pouvons ajouter les cas d'arrêt correspondants aux cycles auto-bloquants.

## 3.3 Autres classes de décidabilité et compatibilité avec les programmes NME

Nous avons pour le moment présenté les classes de décidabilité connues pour ASP et les règles existentielles. Nous avons vu précédemment qu'il est possible de réécrire facilement un programme NME

en programme ASP classique tout en conservant les answer set du programme original, un answer set pouvant être calculé en un temps fini dans un programme NME reste calculable en un temps fini pour sa traduction en ASP classique. Nous pouvons en déduire que les classes compatibles en ASP le sont avec les programmes NME. Les classes présentées précédemment traitent de la terminaison en marche avant mais il existe aussi deux autres types de classes abstraites en règles existentielles, une traitant de la terminaison en temps fini d'un programme en marche arrière à partir d'une requête, et une seconde traitant d'une représentation du noyau sous forme d'arbre à largeur arborescente bornée. Nous présentons donc ici ces deux classes et discuterons leur compatibilité avec les programmes NME. En effet, ces classes n'ont pas été étudiées en détail. Plus complexes à adapter à ASP, nous expliquerons pourquoi dans les discussions sur chacune des classes.

### 3.3.1 Finite Unification Set

Nous allons donc nous intéresser dans un premier temps aux ensembles de règles à unification finie, notés FUS (Finite Unification Set). Un programme est FUS si on peut réécrire en un temps fini une requête à partir de son ensemble de règles telle que celle-ci corresponde aux faits initiaux du programme. Cette classe est uniquement issue des règles existentielles. Nous verrons par la suite sa compatibilité avec ASP.

**Définition 3.3.1** (Ensemble à unification finie [70])**.** *Soit $\mathcal{R}$ un ensemble de règles, $\mathcal{R}$ est appelé ensemble à unification finie si pour toute requête $Q$, il existe $\mathcal{Q}$, une union de requêtes conjonctives, telle que pour tout ensemble de faits $\mathcal{F}$ on a $(\mathcal{F}, \mathcal{R}) \models Q$ si et seulement s'il existe $Q' \in \mathcal{Q}$ telle que $\mathcal{F} \models Q'$. On dit que la réécriture $\mathcal{Q}$ est adéquate et complète en fonction de $\mathcal{R}$.*

La réécriture d'une requête comme définie dans [70] utilise la notion de réécriture de séquence correspondant aux règles agglomérées. Ainsi pour chaque règle $r \in \mathcal{R}$ nous réécrivons la requête $Q$ en une requête $Q'$ s'il existe un unificateur de $Q$ avec $r$. Un programme $P = (\mathcal{F}, \mathcal{R})$ est alors FUS s'il existe une séquence de réécriture finie $Q' = Q \diamond_{\mu_1} r_1 \ldots r_{k-1} \diamond_{\mu_k} r_k$ telle que $\mathcal{F} \models Q'$. Une telle réécriture est appelée réécriture-$\mathcal{R}$.

**Théorème 3.3.1** (Complétude [65])**.** *Soit $\mathcal{F}$ un fait, $\mathcal{R}$ un ensemble de règles existentielles, et $Q$ une requête booléenne. Alors $\mathcal{F}, \mathcal{R} \models Q$ si et seulement s'il existe une réécriture-$\mathcal{R}$ $Q'$ de $Q$ telle que $\mathcal{F} \models Q'$.*

**Exemple 69.** *Soit le programme $P_{69} = (\mathcal{F}_{69}, \mathcal{R}_{69})$ avec l'ensemble de règles suivant :*

$$\mathcal{R}_{69} = \left\{ \begin{array}{l} r_1 : \texttt{humain(X)} \leftarrow \texttt{homme(X).} \\ r_2 : \texttt{humain(X)} \leftarrow \texttt{femme(X).} \end{array} \right\}$$

*et la requête suivante :*

$$Q_{69} : \texttt{humain(X)}$$

*Ce programme est un programme logique défini dont la requête peut être réécrite simplement. Ici $Q_{69}$ peut être réécrite en deux requêtes :*

$$Q'_{69} : \texttt{homme(X)} = Q_{69} \diamond_{\mu_{69}} r_1$$
$$Q''_{69} : \texttt{femme(X)} = Q_{69} \diamond_{\mu'_{69}} r_2$$

*Ainsi si le programme possède un fait composé de soit* `homme(t)` *ou* `femme(t)` *avec* `t` *un terme, alors il existe une réponse à la requête. La requête $Q_{69}$ peut donc être réécrite à partir de ce programme en un temps fini. Ce programme est donc FUS. Si nous appliquons maintenant la réécriture sur un programme logique normal nous avons deux solutions : réécrire la requête à partir des corps de règles entiers, ou réécrire à partir des corps positifs uniquement.*

De même que pour la classe FES, FUS est une classe abstraite et indécidable, on ne peut donc pas reconnaître les programmes FUS. Il est par contre possible de définir des sous-classes concrètes de FUS reconnaissables uniquement à partir de la syntaxe. Nous ne détaillerons pas ces classes concrètes dans cette thèse mais nous allons plutôt nous intéresser à la compatibilité de cette classe abstraite avec ASP.

### 3.3.2 ASP et FUS

Dans le cas de la classe FUS, le problème qui se pose est la réécriture de règles avec un corps négatif. La réécriture étant définie pour des règles positives il faudrait redéfinir cette réécriture pour des règles ASP.

**Exemple 70.** *Soit le programme $P_{70} = (\mathcal{F}_{70}, \mathcal{R}_{70})$ avec l'ensemble de règles suivant :*

$$\mathcal{R}_{70} = \left\{ \begin{array}{l} r_1 : \texttt{homme(X)} \leftarrow \texttt{humain(X), not femme(X).} \\ r_2 : \texttt{femme(X)} \leftarrow \texttt{humain(X), not homme(X).} \end{array} \right\}$$

*et la requête suivante :*

$$Q_{70} : \texttt{homme(X)}$$

*La réécriture à partir du corps positif donne :*

$$Q'_{70} : \texttt{humain(X)}$$

*Le problème est que* $\texttt{humain(X)}$ *n'est pas une réponse à la requête étant donné que nous n'avons pas d'information sur le sexe de la personne, elle peut donc être au choix femme ou homme. La réécriture est donc inexacte. Si nous réécrivons maintenant la requête à l'aide du corps complet nous pouvons réécrire dans un premier temps la requête à l'aide de $r_1$ ce qui nous donne :*

$$Q'_{70} : \texttt{humain(X), not femme(X)}$$

*Le problème est maintenant de traiter la partie négative, étant donné que l'on pourrait réécrire femme(X) à partir de la règle $r_2$.*

$$Q''_{70} : \texttt{humain(X), not (humain(X), not homme(X))}$$

*mais la réécriture ne se terminerait pas.*

Sans un nouvel algorithme prenant en considération la négation par défaut et permettant une réécriture correcte de la requête il n'est pas possible de réécrire en un temps fini une requête avec un programme logique normal. La définition de la classe FUS pour ASP mérite à elle seule un travail considérable, bien que l'approche sur l'interrogation dans la prochaine partie amène quelques éléments intéressants pouvant servir de piste, la décidabilité n'étant qu'une partie de cette thèse nous n'avons abordé que la classe FES représentant déjà un grand nombre de programmes décidables. FUS n'est pas la dernière classe abstraite de décidabilité étudiée en règles existentielles, une autre classe existe s'intéressant cette fois-ci à l'arborescence d'un programme.

### 3.3.3 (Greedy) Bounded Treewidth Set

Après avoir étudié la terminaison d'un programme en marche avant puis aborder brièvement la marche arrière dans la section précédente, nous allons nous intéresser à sa représentation sous forme d'arbre.

**Définition 3.3.2** (Bounded Tree-width Set [71])**.** *Un ensemble de règles $\mathcal{R}$ est appelé ensemble à largeur arborescente bornée (BTS pour "Bounded Tree-width Set") si pour tout fait $\mathcal{F}$, il existe un entier $b$ (dépendant de $\mathcal{F}$ et $\mathcal{R}$) tel que pour toute dérivation-$\mathcal{R}$ $\mathcal{F}'$ de $\mathcal{F}$, la largeur arborescente de noyau ($\mathcal{F}'$) est inférieure ou égale à $b$.*

En considérant la définition 1.3.9 de la dérivation-$\mathcal{R}$. Nous définissons une dérivation-$\mathcal{R}$ gloutonne si pour toute application de règle dans cette dérivation, il existe une substitution des variables de la frontière dans des termes ajoutés par l'application de la règle précédente. Il est ainsi possible de construire un arbre décomposant la dérivation d'un ensemble de faits avec une méthode gloutonne.

**Définition 3.3.3** (Greedy Bounded Tree-width Set of rules (gBTS) [71])**.** *Un ensemble de règles $\mathcal{R}$ est dit à largeur arborescente gloutonne bornée si pour tout fait $\mathcal{F}$ toute dérivation-$\mathcal{R}$ de $\mathcal{F}$ est gloutonne.*

Pour les mêmes raisons que pour la classe FUS nous ne détaillerons pas dans cette thèse les classes concrètes composant BTS. Il est préférable de discuter de la compatibilité de celle-ci avec ASP.

### 3.3.4 ASP et BTS

Le problème dans l'utilisation de la classe BTS avec ASP est dû au fait qu'il est impossible de savoir si la construction d'un modèle va mener à un answer set. On cherche dans BTS à déterminer qu'un arbre de dérivation est à profondeur finie. Le problème est qu'il existe pour un programme ASP non-stratifié (ne possédant pas un unique modèle) plusieurs dérivations-$\mathcal{R}$ ce qui rend la propriété gBTS obsolète. De plus il est impossible de déterminer à l'avance si un arbre va continuer de grandir ou bien être coupé par un corps négatif contrairement aux règles existentielles où lorsque l'on a déduit une connaissance elle n'est jamais remise en cause.

**Exemple 71.** *Soit l'ensemble de règles suivant :*

$$\mathcal{R}_{71} = \left\{ \begin{array}{l} r_1 : \texttt{mere}(\texttt{Y},\texttt{X}), \texttt{humain}(\texttt{Y}) \leftarrow \texttt{humain}(\texttt{X}), \texttt{not homme}(\texttt{X}). \\ r_2 : \texttt{pere}(\texttt{Y},\texttt{X}), \texttt{humain}(\texttt{Y}) \leftarrow \texttt{humain}(\texttt{X}), \texttt{not femme}(\texttt{X}). \end{array} \right\}$$

*Il n'est pas possible de déterminer si cet ensemble de règle est $gBTS$ car il existe une infinité de dérivation-$\mathcal{R}$, étant donné que pour chaque humain il existe un parent, soit homme soit femme ayant aussi un parent. Il n'est ainsi pas possible de déterminer si toutes les dérivations-$\mathcal{R}$ sont bornées et s'il n'existe pas pour chaque dérivation un atome du corps négatif qui peut arrêter la dérivation gloutonne à un certain point.*

La classe BTS mérite néanmoins d'être approfondie pour les programmes NME, elle permettrait d'élargir considérablement le nombre de programmes dont nous sommes capables d'assurer qu'ils se terminent.

## 3.4 Conclusion

Pour conclure, nous avons défini un ensemble de classes de décidabilité pour l'extension des programmes NME correspondant à la classe abstraite FES et ses sous classes concrètes. Cette étude nous a permis d'homogénéiser les classes la constituant en utilisant une représentation sous forme de graphe de positions et même d'étendre ces notions à l'aide des dépendances de règles et des unificateurs. Même si la classe FES permet de détecter une grande partie des programmes DL-Lite décidables, nous savons aussi qu'ils ne sont pas tous détectables et donc que nous ne sommes pas en mesure de déterminer si un programme DL-Lite modélisé par un programme NME se terminera

ou non. Pour pouvoir déterminer la décidabilité de l'ensemble des programmes qui nous intéresse il faudrait que l'on puisse déterminer si un programme NME fait partie d'une classe gBTS ce qui n'est malheureusement pas encore le cas tout comme les classes composant FUS qui restent indécidables pour un programme NME. Ceci fait partie des axes de recherche laissés ouverts permettant de parfaire la détection de la décidabilité des programmes NME.

# 4

# Interrogation

## 4.1 Introduction

Un aspect fondamental dans le raisonnement sur des ontologies est l'interrogation. Celle-ci repose sur la déduction d'une réponse à une question sur un programme, formulée par une requête. L'interrogation permet ainsi d'extraire les informations d'une ontologie répondant à certaines conditions tout en limitant au maximum le nombre de calculs nécessaires. Elle est très étudiée en règles existentielles et en bases de données mais peu en ASP. L'interrogation nous permet d'isoler des connaissances répondant à une requête sur un programme sans forcément avoir besoin de calculer le modèle en entier. Le but étant de calculer les connaissances strictement nécessaires à la réponse d'une requête. Nous pouvons, par exemple, dans l'ontologie `university`, poser les interrogations suivantes :

"Toto enseigne-t-il l'informatique ?",
"Existe-t-il une matière enseignée par Toto ?" ou encore
"Quelles sont les matières enseignées par Toto ?".

Pour les deux premières interrogations nous attendons une réponse par oui ou non et pour la dernière l'ensemble des matières qui sont enseignées par Toto.
Le principe de l'interrogation d'ontologie est de se servir des informations présentes dans la requête pour calculer uniquement les informations nécessaires pour y répondre.
Dans cette thèse, nous souhaitons étudier l'interrogation sur un programme NME, ce qui, après traduction en ASP, peut être vu comme un approfondissement de l'étude de l'interrogation en ASP. Le problème est que l'interrogation sur une ontologie est indécidable en règles existentielles [15]. Il existe néanmoins des classes de programme connues pour lesquelles l'interrogation est décidable. Un des intérêts de l'interrogation est qu'il n'est pas nécessaire de calculer l'ensemble d'un modèle pour pouvoir répondre à une requête, et cela devient un gros atout pour l'interrogation d'ontologies qui sont constituées d'un grand nombre de données mais ne possèdent que très peu de dépendances entre elles, contrairement aux problèmes combinatoires qui possèdent souvent peu de données mais beaucoup de dépendances entre eux. Par exemple si nous interrogeons l'ontologie `university` en demandant si Toto enseigne un cours d'informatique, nous n'avons pas besoin de savoir que Titi suit un cours de mathématique, et donc une partie du programme n'est pas nécessaire. Nous pouvons alors utiliser des méthodes permettant de répondre à une requête rapidement à l'aide d'un algorithme réécrivant le programme à partir de celle-ci. Le programme sera alors réduit aux seules règles nécessaires pour répondre à la requête. Malgré cela, la plus grande difficulté reste qu'en ASP un

programme peut n'avoir aucun answer set. Ainsi une réécriture classique, du point de vue des règles existentielles, ne fonctionne pas en ASP, car nous pouvons nous retrouver avec une réponse sur le programme réécrit alors que le programme initial ne possède pas de modèle. Dans ce cas la validité d'une réponse à une interrogation peut-être remise en cause car s'il n'existe pas de modèle cela implique qu'il existe une incohérence dans les données. Certaines méthodes comme les *magic sets* [18][36] ont été proposés pour des programmes avec variables existentielles sans négation par défaut [8] ou des programmes ASP possédant au moins un modèle [6] (quels que soient les faits initiaux) appelés super-consistants[7]. Le problème est que cette méthode laisse une grande partie des programmes de côté.
Dans cette thèse, nous offrons une réponse efficace à l'interrogation d'un programme ASP d'une part en isolant les règles nécessaires à l'obtention d'une réponse valide et d'autre part en optimisant le nombre d'instanciations nécessaires pour cette réponse tout en gardant complétude et correction de la réponse avec le programme original.
Dans un premier temps nous définissons ce que sont les requêtes et les réponses en nous intéressant à l'interrogation en règles existentielles puis à l'existence d'une réponse dans un ensemble d'answer set en ASP. Nous traitons ensuite le cas spécifique de l'inconsistance et l'utilité d'utiliser un prétraitement sur un programme pour tester la consistance de celui-ci. Nous nous intéressons ensuite aux techniques permettant d'isoler les règles nécessaires pour répondre à une requête spécifique, puis aux optimisations possibles grâce à la réécriture du programme en fonction de la requête. Pour terminer nous proposons une méthode permettant d'utiliser les propriétés de la négation par défaut pour restreindre encore plus l'instanciation nécessaire à l'obtention d'une réponse.

## 4.2 Interrogation en règles existentielles

Dans le domaine des bases de données et des systèmes à base de connaissances, les requêtes conjonctives sont utilisées pour interroger un programme, soit pour savoir si un élément existe, soit pour récupérer l'ensemble des éléments vérifiant certaines conditions. Nous pouvons ainsi poser des interrogations sous la forme d'ensembles de faits existentiels :

$$Q_1 : \{\texttt{donneCours}(\texttt{toto}, \texttt{informatique})\}$$
$$Q_2 : \exists \texttt{X}\{\texttt{donneCours}(\texttt{toto}, \texttt{X})\}$$
$$Q_3 : \{\texttt{donneCours}(\texttt{toto}, \texttt{X}), \texttt{cours}(\texttt{X})\}$$

Pour la requête $Q_1$ nous attendons une réponse booléenne : soit la réponse à la requête est vraie, dans ce cas nous avons réussi à déduire dans notre programme que Toto enseigne l'informatique, soit elle est fausse et dans ce cas il n'est pas possible de déduire cette affirmation à l'aide des faits et des règles de notre programme. Dans le cas de la requête $Q_2$ nous voulons savoir s'il existe au moins une matière enseignée par Toto, dans ce cas s'il existe une instance avec une matière enseignée par Toto la réponse est vraie sinon elle est fausse. Dans le dernier cas, nous souhaitons obtenir toutes les matières enseignées par Toto, la réponse sera alors toutes les instances pouvant être déduites à partir de notre programme vérifiant cette condition.

**Définition 4.2.1** (Requête conjonctive)**.** *Une* requête conjonctive *sur un programme est une conjonction de la forme :*

$$\exists \texttt{X}_1, \ldots \exists \texttt{X}_\texttt{k}\{\texttt{a}_1, \ \ldots, \ \texttt{a}_\texttt{n}\}$$

*avec* $n > 0$, $k \geq 0$, $\texttt{a}_1, \ \ldots, \ \texttt{a}_\texttt{n}$ *des atomes, et* $\texttt{X}_1, \ldots, \texttt{X}_\texttt{k}$ *des variables quantifiées existentiellement apparaissant dans au moins un atome de* $a_1, \ \ldots, \ a_n$*. Une* requête conjonctive *est* booléenne *si toutes les variables de l'ensemble* $\{\texttt{a}_1, \ \ldots, \ \texttt{a}_\texttt{n}\}$ *sont existentiellement quantifiées.*

Nous pouvons reformuler le problème d'interrogation en règles existentielles de cette manière : Soit un programme $P = (\mathcal{F}, \mathcal{R})$ et une requête conjonctive $Q$, est-ce que $(\mathcal{F}, \mathcal{R}) \models Q$ ?

**Définition 4.2.2** (Réponse à une requête conjonctive)**.** *Une* réponse *à une requête conjonctive sur un programme $P = (\mathcal{F}, \mathcal{R})$ est l'ensemble des substitutions $\sigma(\texttt{ans}) \subseteq M$, avec $M$ le modèle issu de la saturation de $P$ et* ans *l'atome réponse. Pour une requête conjonctive booléenne la réponse est* vrai *si* $\texttt{ans} \in M$ *et* faux *sinon.*

**Exemple 72.** *Soit $P_{72} = (\mathcal{F}_{72}, \mathcal{R}_{72})$ le programme composé de l'ensemble de règles existentielles suivant :*

$$\mathcal{R}_{72} = \left\{ \begin{array}{l} r_1 : \texttt{enseignantChercheur(X)}, \texttt{domaine(Y)} \rightarrow \texttt{donneCours(X,Y)}, \texttt{cours(Y)}. \\ r_2 : \texttt{maitreConference(X)} \rightarrow \texttt{enseignantChercheur(X)}. \\ r_3 : \texttt{enseignant(X)} \rightarrow \texttt{donneCours(X,Y)}, \texttt{cours(Y)}. \end{array} \right\}$$

*et de l'ensemble de faits existentiels suivant :*

$$\mathcal{F}_{72} = \{\texttt{maitreConference(titi)}, \texttt{domaine(informatique)}, \texttt{enseignant(titi)}\}$$

*Si nous appliquons le $skolem - chase$ le modèle obtenu est :*

$$\begin{array}{rl} M = & \{\texttt{maitreConference(titi)}, \texttt{domaine(informatique)}, \texttt{enseignant(titi)}, \\ & \texttt{enseignantChercheur(titi)}, \texttt{donneCours(titi,informatique)}, \\ & \texttt{cours(informatique)}, \texttt{donneCours(titi,sk(titi))}, \texttt{cours(sk(titi))}\} \end{array}$$

*Considérons la requête $Q^1_{72} = \{\texttt{donneCours(titi,informatique)}\}$ la réponse à cette requête est* vrai *car $(\mathcal{F}_{72}, \mathcal{R}_{72}) \models Q^1_{72}$. Considérons maintenant la requête $Q^2_{72} = \exists \texttt{X}\{\texttt{donneCours(titi,X)}\}$ la réponse est encore* vrai *car $(\mathcal{F}_{72}, \mathcal{R}_{72}) \models Q^2_{72}$. Pour finir considérons la requête $Q^3_{72} = \{\texttt{donneCours(titi,X)}\}$ dont la réponse est l'ensemble des substitutions $\sigma$ telles que $(\mathcal{F}_{72}, \mathcal{R}_{72}) \models Q^3_{72}$ (soit encore l'ensemble constitué des substitutions $\sigma = [\texttt{X} \leftarrow \texttt{informatique}]$ et $\sigma' = [\texttt{X} \leftarrow \texttt{sk(titi)}]$).*

## 4.3 Interrogation en ASP

En ASP, l'interrogation a le même but qu'en règles existentielles, questionner sur l'existence d'un élément ou récupérer l'ensemble des éléments vérifiant certaines conditions, mais doit prendre en compte qu'un programme ASP a la possibilité d'avoir aucun, un ou plusieurs answer set. Il est donc nécessaire de définir une sémantique d'une réponse à une requête propre à ASP en traitant les cas où il n'y a pas d'answer set ou bien plusieurs answer set. Ainsi nous définissons plusieurs réponses possibles pour une requête qui peuvent être crédules ou sceptiques : dans le premier cas la réponse doit être valide pour au moins un answer set et dans le second la réponse doit être valide pour tous les answer set. La représentation utilisée pour les requêtes ASP est la même que celle utilisée pour DATALOG. Une requête est donc représentée sous forme de règle afin de l'intégrer à l'ensemble des règles du programme et permettre par la suite de faire abstraction des faits pour certains traitements.

**Définition 4.3.1** (Requête conjonctive)**.** *Une* requête conjonctive *sur un programme est une règle de la forme :*

$$\texttt{ans}(\texttt{X}_1, \ldots, \texttt{X}_k) \leftarrow \texttt{a}_1, \ldots, \texttt{a}_n.$$

*avec $k \geq 0$, $n > 0$, $\{\texttt{a}_1, \ldots, \texttt{a}_n\}$ un ensemble d'atomes,* ans *un symbole de prédicat n'apparaissant pas dans le programme ni dans le corps de la requête, appelé* prédicat réponse*, et $\texttt{X}_1, \ldots, \texttt{X}_k$ des variables apparaissant dans au moins un atome $\texttt{a}_i$. Une* requête conjonctive *est* booléenne *si $k = 0$.*

Nous simplifions par *requête* lorsque nous parlons des requêtes conjonctives et conjonctives booléennes indifféremment.
Nous pouvons ainsi intégrer la requête $Q$ à l'ensemble des règles $\mathcal{R}$ d'un programme $P$.

Soit $P = (\mathcal{F}, \mathcal{R})$ un programme ASP, nous avons $\mathbf{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ le programme $P$ avec $\mathcal{F}$ la base de faits, $\mathcal{R}$ l'ensemble des règles et $Q$ la requête sur le programme $P$.

**Théorème 4.3.1** (Programme avec requête)**.** *Soit $\mathcal{AS} = \{AS_1, \ldots, AS_n\}$ l'ensemble des answer set de $P$. Si $\mathcal{AS}^Q = \{AS_1^Q, \ldots, AS_n^Q\}$ est l'ensemble des answer set d'un programme **P** alors*

$$AS_i^Q = AS_i \cup \{\texttt{tête}(\sigma(Q)) | \texttt{corps}(\sigma(Q)) \subseteq AS_i\}$$

*pour tout $i$, $1 \leq i \leq n$, et pour toute substitution $\sigma$.*

Remarque : le corps d'une requête ne possède pas de corps négatif donc $\texttt{corps}^+(Q) = \texttt{corps}(Q)$.

**Preuve 4.3.1.** *Soit $P$ un programme et $Q$ une requête sur $P$. D'après la définition 4.3.1, $\forall r \in \mathcal{R}, \texttt{tête}(Q) \notin r$. Une application de $Q$ ne peut pas permettre l'application d'une règle de $P$. Si la règle $Q$ est appliquée lors de l'application de l'opérateur de conséquence $T_P$ alors les instances de* ans *sont ajoutées à l'answer set (seul l'atome* ans *est ajouté pour une requête conjonctive booléenne). Soit $AS_i$ un answer set de $P$ alors nous avons $AS_i^Q = AS_i$ auquel on ajoute les instances de* ans *obtenues par l'application de $Q$ par $AS_i$. D'après le théorème 1.2.7 $AS_i$ est un point fixe provenant de l'application de l'opérateur de conséquence $T_P$. Étant donné que* ans *ne permet pas d'appliquer d'autres règles alors $AS_i^Q$ est le même point fixe sur **P** à l'ajout des applications de $Q$ près.*

Pour traiter le problème de l'interrogation nous allons utiliser dans les exemples un programme basé sur l'ontologie university mais simplifié avec seulement des symboles de prédicat d'arité 1 afin de mettre en avant des problèmes spécifiques à l'utilisation de la négation par défaut. Les règles de l'ontologie originale ne permettant pas de mettre en valeur les problèmes que nous souhaitons souligner, et ne comportant pas d'exception, nous avons créer des exemples spécifiques à ASP à partir de university. Ici nous cherchons aussi à avoir un programme nous permettant d'obtenir plusieurs answer set. L'utilisation de programmes NME ouvre de nouvelles perspectives de programmes à traiter qu'il est rare de rencontrer dans des ontologies classiques.

**Exemple 73.** *Soit $P_{73} = (\mathcal{F}_{73}, \mathcal{R}_{73})$ le programme suivant*

$$\mathcal{R}_{73} = \left\{ \begin{array}{l} r_1 : \texttt{administration(X)} \leftarrow \texttt{personnelUniversite(X)}, \texttt{not enseignantChercheur(X)}. \\ r_2 : \texttt{enseignantChercheur(X)} \leftarrow \texttt{personnelUniversite(X)}, \texttt{not administration(X)}. \\ r_3 : \texttt{conseilAdministration(X)} \leftarrow \texttt{administration(X)}, \texttt{direction(X)}. \\ r_4 : \texttt{maitreConference(X)} \leftarrow \texttt{enseignantChercheur(X)}, \texttt{not doctorant(X)}. \\ r_5 : \texttt{doctorant(X)} \leftarrow \texttt{enseignantChercheur(X)}, \texttt{etudiant(X)}. \\ r_6 : \texttt{chercheur(X)} \leftarrow \texttt{enseignantChercheur(X)}. \\ r_7 : \texttt{contrainte1} \leftarrow \texttt{maitreConference(X)}, \texttt{doctorant(X)}, \texttt{not contrainte1}. \\ r_8 : \texttt{contrainte2} \leftarrow \texttt{etudiant(X)}, \texttt{chomeur(X)}, \texttt{not contrainte2}. \end{array} \right\}$$

$$\mathcal{F}_{73} = \{\texttt{administration(toto)}, \texttt{direction(toto)}, \texttt{personnelUniversite(titi)}, \texttt{direction(titi)}, \texttt{etudiant(titi)}\}$$

*et*

$$Q_{73} : \texttt{ans(X)} \leftarrow \texttt{conseilAdministration(X)}.$$

*Le programme $P_{73}$ possède deux answer set*

$$\begin{aligned} AS_1 = \ & \{\texttt{administration(toto)}, \texttt{direction(toto)}, \texttt{personnelUniversite(titi)}, \\ & \texttt{direction(titi)}, \texttt{etudiant(titi)}, \texttt{conseilAdministration(toto)}, \\ & \texttt{administration(titi)}, \texttt{conseilAdministration(titi)}\} \\ AS_2 = \ & \{\texttt{administration(toto)}, \texttt{direction(toto)}, \texttt{personnelUniversite(titi)}, \\ & \texttt{direction(titi)}, \texttt{etudiant(titi)}, \texttt{conseilAdministration(toto)}, \\ & \texttt{enseignantChercheur(titi)}, \texttt{doctorant(titi)}, \texttt{chercheur(titi)}\} \end{aligned}$$

*si nous ajoutons $Q_{73}$ à l'ensemble des règles $\mathcal{R}_{73}$ de $P_{73}$ alors la règle $Q_{73}$ est appliquée et l'atome* ans *est ajouté à l'answer set. Donc le programme $\boldsymbol{P}_{73}$ possède deux answer set*

$$\begin{aligned} AS_1^Q = \ & \{\texttt{administration(toto)}, \texttt{direction(toto)}, \texttt{personnelUniversite(titi)}, \\ & \texttt{direction(titi)}, \texttt{etudiant(titi)}, \texttt{conseilAdministration(toto)}, \\ & \texttt{administration(titi)}, \texttt{conseilAdministration(titi)}, \texttt{ans(toto)}, \texttt{ans(titi)}\} \\ AS_2^Q = \ & \{\texttt{administration(toto)}, \texttt{direction(toto)}, \texttt{personnelUniversite(titi)}, \\ & \texttt{direction(titi)}, \texttt{etudiant(titi)}, \texttt{conseilAdministration(toto)}, \\ & \texttt{enseignantChercheur(titi)}, \texttt{doctorant(titi)}, \texttt{chercheur(titi)}, \texttt{ans(toto)}\} \end{aligned}$$

Par la suite pour simplifier le programme nous utilisons les écritures suivantes pour les symboles de prédicat : pU pour personnelUniversite, ad pour administration, d pour direction, cA pour conseilAdministration, eC pour enseignantChercheur, mC pour maitreConference, et pour etudiant, chr pour chercheur, phdS pour doctorant, ch pour chomeur. Tous les symboles de prédicat étant d'arité 1.

Comme discuté dans l'introduction de cette partie, nous définissons une réponse à un programme ASP de deux façons, en considérant soit une réponse crédule dont la réponse est valide pour au moins un answer set, soit une réponse sceptique où la réponse est valide pour tous les answer set. La réponse crédule correspond à une réponse naïve à la requête, si la réponse est faux pour tous les answer set sauf un, la réponse crédule est tout de même vrai. Et dans le cas de la réponse sceptique si la réponse est vrai pour tous les answer set sauf un, la réponse est tout de même faux.

Le problème de l'interrogation en ASP réside dans le fait qu'un programme peut ne pas avoir d'answer set, dans ce cas nous définissons le programme comme inconsistant.

**Définition 4.3.2** (Consistance)**.** *Soit $\mathcal{AS}$ l'ensemble des answer set d'un programme ASP. Un programme est* consistant *si $\mathcal{AS} \neq \emptyset$,* inconsistant *sinon.*

Notons que la non existence d'un answer set $\mathcal{AS} = \emptyset$, est différente de l'existence d'un answer set vide $\mathcal{AS} = \{\emptyset\}$. Le premier correspond à un programme n'ayant pas d'answer set tandis que le second correspond à un programme dont l'unique answer set ne contient pas d'atome.

Nous proposons ici de considérer la réponse à un programme inconsistant comme absurde. La crédibilité d'une réponse à un programme inconsistant étant discutable car l'inconsistance provient soit d'un conflit dans les faits ou dans les règles du programme. Une interrogation étant posée sur un modèle s'il n'existe pas de modèle alors une réponse est absurde car le programme ne peut pas donner d'answer set valide.

Nous distinguons deux types de requêtes, nous définissons dans un premier temps les requêtes conjonctives classiques (non-booléennes) dont la réponse est un ensemble d'atomes, puis les requêtes conjonctives booléennes dont la réponse est vrai ou faux.

**Définition 4.3.3** (Réponse à une requête conjonctive)**.** *Soit $\boldsymbol{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ le programme $P = (\mathcal{F}, \mathcal{R})$ requêté par*

$$Q = \texttt{ans}(\texttt{X}_1,\ \ldots,\ \texttt{X}_k) \leftarrow\ \texttt{a}_1,\ \ldots,\ \texttt{a}_n.$$

*avec $k > 0$ et $n > 0$, une requête conjonctive. Soit $\mathcal{AS}^Q$ l'ensemble des answer set de **P**, la* réponse *à $Q$ dans $P$ est* `absurde` *si $\mathcal{AS}^Q = \emptyset$ sinon la réponse est* valide *et est l'ensemble des substitutions $\sigma$ tel que :*

- $\forall AS^Q \in \mathcal{AS}^Q,\ \sigma(\texttt{ans}) \subseteq AS^Q$ *pour la* réponse sceptique
- $\exists AS^Q \in \mathcal{AS}^Q,\ \sigma(\texttt{ans}) \subseteq AS^Q$ *pour la* réponse crédule

**Exemple 74.** *Soit le programme $P_{73}$ avec la requête conjonctive suivante :*

$$Q_{73} = \texttt{ans}(\texttt{X}) \leftarrow\ \texttt{cA}(\texttt{X}).$$

*les deux answer set de $\boldsymbol{P}_{73}$ sont les suivants :*

$$\begin{aligned} AS_1^Q = \ & \{\texttt{ad}(\texttt{toto}), \texttt{d}(\texttt{toto}), \texttt{pU}(\texttt{titi}), \texttt{d}(\texttt{titi}), \texttt{et}(\texttt{titi}), \texttt{cA}(\texttt{toto}), \texttt{a}(\texttt{titi}), \texttt{cA}(\texttt{titi}), \\ & \texttt{ans}(\texttt{toto}), \texttt{ans}(\texttt{titi})\} \\ AS_2^Q = \ & \{\texttt{ad}(\texttt{toto}), \texttt{d}(\texttt{toto}), \texttt{pU}(\texttt{titi}), \texttt{d}(\texttt{titi}), \texttt{et}(\texttt{titi}), \texttt{cA}(\texttt{toto}), \texttt{eC}(\texttt{titi}), \\ & \texttt{phdS}(\texttt{titi}), \texttt{chr}(\texttt{titi}), \texttt{ans}(\texttt{toto})\} \end{aligned}$$

*La réponse sceptique à $Q_{73}$ est $\{[\texttt{X} \leftarrow \texttt{toto}]\}$ car* $\texttt{ans}(\texttt{toto})$ *appartient aux deux answer set. La réponse crédule, quant à elle, est $\{[\texttt{X} \leftarrow \texttt{toto}], [\texttt{X} \leftarrow \texttt{titi}]\}$ (nous avons donc deux substitutions possibles pour* `X`*).*

**Définition 4.3.4** (Réponse à une requête booléenne conjonctive)**.** *Soit $\boldsymbol{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ le programme $P = (\mathcal{F}, \mathcal{R})$ requêté par*

$$Q = \texttt{ans} \leftarrow\ \texttt{a}_1,\ \ldots,\ \texttt{a}_n.$$

*avec $n > 0$, une requête booléenne conjonctive et $\mathcal{AS}^Q$ l'ensemble des answer set de **P**.*
*Si $\mathcal{AS}^Q = \emptyset$ alors la réponse à $Q$ dans $P$ est* `absurde`*,*
*sinon la réponse est* valide *et :*

- *pour la réponse sceptique :*
  - *si $\forall AS^Q \in \mathcal{AS}^Q,\ \texttt{ans} \in AS^Q$ alors la réponse est* `vrai`
  - *sinon la réponse est* `faux`
- *pour la réponse crédule :*
  - *si $\exists AS^Q \in \mathcal{AS}^Q,\ \texttt{ans} \in AS^Q$ alors la réponse est* `vrai`
  - *sinon la réponse est* `faux`

**Exemple 75.** *Soit le programme $P_{73}$ avec la requête conjonctive suivante :*

$$Q_{75} = \texttt{ans} \leftarrow\ \texttt{cA}(\texttt{titi}).$$

*les deux answer set de $\boldsymbol{P}_{75} = (\mathcal{F}_{73}, \mathcal{R}_{73} \cup \{Q_{75}\})$ sont les suivants :*

$$\begin{aligned} AS_1^Q = \ & \{\texttt{ad}(\texttt{toto}), \texttt{d}(\texttt{toto}), \texttt{pU}(\texttt{titi}), \texttt{d}(\texttt{titi}), \texttt{et}(\texttt{titi}), \texttt{cA}(\texttt{toto}), \texttt{a}(\texttt{titi}), \texttt{cA}(\texttt{titi}), \texttt{ans}\} \\ AS_2^Q = \ & \{\texttt{ad}(\texttt{toto}), \texttt{d}(\texttt{toto}), \texttt{pU}(\texttt{titi}), \texttt{d}(\texttt{titi}), \texttt{et}(\texttt{titi}), \texttt{cA}(\texttt{toto}), \texttt{eC}(\texttt{titi}), \texttt{chr}(\texttt{titi}), \\ & \texttt{phdS}(\texttt{titi})\} \end{aligned}$$

*La réponse sceptique à $Q_{75}$ est* `faux` *car* ans *n'appartient pas à $AS_2^Q$. La réponse crédule est* `vrai` *car* ans *apparaît dans l'answer set $AS_1^Q$.*

D'autres approches de l'interrogation en ASP ont été effectuées dans [6]. En comparaison, dans la documentation du solveur ASP DLV [3] il est proposé de considérer une réponse à une requête comme vide ($\emptyset$) lorsqu'un programme est inconsistant. Ceci correspond dans notre cas à la réponse absurde, le programme contenant soit des faits ou des règles erronées. Il ne faut pas confondre $\emptyset$ avec $\{\emptyset\}$ : le premier correspond à la réponse lorsqu'il n'existe pas d'answer set et le deuxième correspond à une réponse vide lorsqu'il est impossible de déduire une instance de `ans` dans les answer set existants. De même, il est proposé de considérer une réponse sceptique à une requête booléenne comme `vrai` lorsqu'un programme est inconsistant. En effet, nous cherchons l'appartenance de `ans` à tous les modèles, étant donné qu'il n'existe pas de modèle, la réponse est donc `vrai` car `ans` appartient bien à tous les modèles existants. A contrario la réponse crédule est `faux` en l'absence de modèle car il n'existe pas d'answer set, et donc pas d'answer set contenant `ans`. En considérant la réponse absurde dans le cas où le programme est inconsistant il est facile de différencier la réponse vide de la réponse absurde.

En ASP, nous distinguons donc différentes réponses en fonction de l'appartenance d'une réponse à aucun, un ou plusieurs modèles à l'instar de l'appartenance d'une formule dans une ou toutes les extensions en logique des défauts [66] et contrairement à d'autres langages, comme Datalog, qui ne possèdent qu'un unique modèle. Nous souhaitons interroger ces programmes et la méthode la plus simple pour répondre à une requête est de calculer l'ensemble des answer set puis de chercher les réponses à notre requête directement dans les answer set trouvés. Il existe néanmoins un inconvénient à l'interrogation de programmes ASP, c'est que le temps de calcul sera au minimum aussi long que la recherche de l'ensemble des answer set et dans le cas d'answer set infinis le calcul de la réponse ne se terminera jamais. Nous allons donc étudier différentes méthodes permettant de trouver les réponses à notre requête sans avoir à calculer entièrement les answer set et de réduire au maximum le nombre de règles et d'instances nécessaires pour obtenir une réponse. Mais avant cela il faudra traiter le problème de l'inconsistance en ASP.

## 4.4 Inconsistance en ASP

Une des différences notables de l'ASP par rapport aux logiques de description et aux règles existentielles est la possibilité d'avoir des programmes inconsistants. Le problème est que lors de l'interrogation d'un programme il faut s'assurer que celui-ci possède au moins un answer set, dans le cas contraire la réponse serait `absurde`. Dans un premier temps nous souhaitons nous assurer qu'un programme n'est pas inconsistant en vérifiant qu'il existe au moins un answer set à notre programme et en effectuant le moins de calcul possible, pour ensuite pouvoir traiter la requête si nécessaire. Cette étape de test de la consistance d'un programme peut être vue comme optionnelle si l'on considère que les données du programme sont valides. Il est donc souvent proposé de considérer des programmes ayant au moins un answer set ou bien de ne tester la consistance que sur les données en lien avec la requête, étant donné que s'il y a une inconsistance sur des données déconnectées de la requête celle-ci n'a pas d'impact sur la réponse. Dans notre cas, nous souhaitons nous assurer de la consistance des données pour fournir une réponse en concordance avec celles-ci, en considérant que si le programme est inconsistant alors il y a un problème soit sur les faits ou bien les règles du programme et que la réponse n'aurait pas de sens dans ce cas.

Nous proposons alors une méthode efficace afin de s'assurer qu'il existe une réponse valide à une requête sans avoir à calculer l'ensemble des answer set. Cette méthode consiste à isoler les parties d'un programme pouvant rendre un programme inconsistant puis vérifier si ces parties rendent effectivement le programme inconsistant. Nous allons alors tester l'existence d'un answer set sur chacune de ces parties et montrer que cela assure l'existence d'un answer set sur le programme original.

Un des avantages est que notre méthode peut-être utilisée en pré-traitement d'une requête et donc ne nécessite pas d'être exécutée une nouvelle fois pour une requête différente sur le même programme,

il est donc même possible de passer cette étape si nous ne souhaitons pas nous assurer de la consistance du programme. Pour ce faire nous reprenons la notion de règles dangereuses, utilisée dans le cadre de Datalog¬ [18] avec l'aide des *magic sets* [36] pour répondre à une requête sur un programme consistant, que nous étendons à tout programme ASP.

La première étape pour vérifier si un programme est inconsistant en limitant le nombre de calcul est d'isoler les parties du programme pouvant être à l'origine d'une inconsistance. Il existe dans la littérature [34] une méthode utilisant le graphe de dépendance des symboles de prédicat pour isoler ces parties dans un programme. Pour cela nous cherchons à identifier des cycles impairs d'arcs négatifs dans le graphe des symboles de prédicat responsables de l'inconsistance d'un programme.

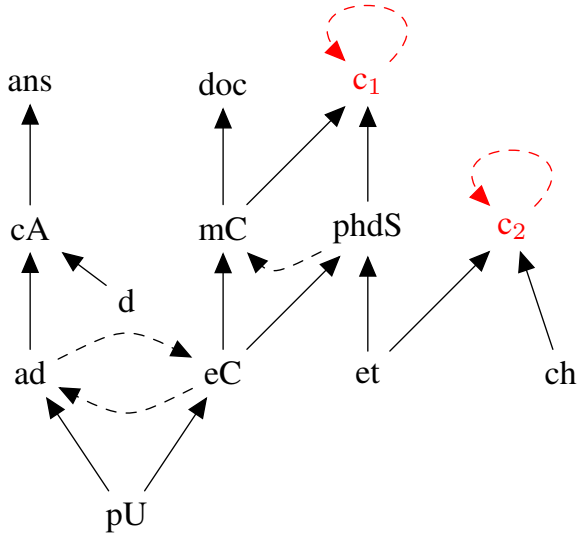
**Définition 4.4.1** (Cycle d'inconsistance)**.** *Soit $\mathcal{G}$ le graphe de dépendance des symboles de prédicat d'un programme ASP. Un* cycle d'inconsistance *est un cycle dans $\mathcal{G}$ comprenant un nombre impair d'arcs négatifs.*

**Théorème** (Inconsistance)**.** *[34] Un programme peut être inconsistant seulement s'il possède un cycle d'inconsistance.*

**Exemple 76.** *Soit le programme $\boldsymbol{P}_{73} = (\mathcal{F}_{73}, \mathcal{R}_{73} \cup \{Q_{73}\})$. Nous rappelons l'ensemble de règles $\mathcal{R}_{73} \cup \{Q_{73}\}$ et définissons le graphe de dépendance des symboles de prédicat associé en figure 4.1 :*

FIGURE 4.1 – Cycles d'inconsistance dans le graphe de dépendance des symboles de prédicat de l'ensemble $\mathcal{R}_{73} \cup \{Q_{73}\}$.

$r_1$ : ad(X) ← pU(X), not eC(X).
$r_2$ : eC(X) ← pU(X), not ad(X).
$r_3$ : cA(X) ← ad(X), d(X).
$r_4$ : mC(X) ← eC(X), not phdS(X).
$r_5$ : phdS(X) ← eC(X), et(X).
$r_6$ : chr(X) ← eC(X).
$r_7$ : c1 ← mC(X), phdS(X), not c1.
$r_8$ : c2 ← et(X), ch(X), not c2.
$Q$ : ans(X) ← cA(X).




*Cet exemple possède deux cycles impairs d'arcs négatifs sur le symbole de prédicat* c1 *et le symbole de prédicat* c2 *représentés en pointillés rouges sur le graphe associé. Dans cet exemple les deux cycles d'inconsistance de l'ensemble de règles $\mathcal{R}_{73} \cup \{Q_{73}\}$ sont issus des règles $r_6$ et $r_7$ qui sont deux contraintes pouvant rendre un programme inconsistant. Le déclenchement d'une contrainte empêche l'appartenance d'un ensemble d'atomes aux answer set du programmes. Elles sont identifiables sur le graphe des prédicats en créant un cycle d'inconsistance.*

**Définition 4.4.2** (Dépendance d'un cycle)**.** *Un cycle d'inconsistance $C$ dépend d'un symbole de prédicat* p *s'il existe un chemin dans le graphe des symboles de prédicat allant de* p *à un symbole de prédicat de $C$. Un cycle $C$ dépend d'une règle $r$ si $C$ dépend du symbole de prédicat de l'atome $a \in \texttt{tête}(r)$.*

**Théorème 4.4.1** (Consistance d'un programme avec requête)**.** *Si un programme $P = (\mathcal{F}, \mathcal{R})$ est consistant (resp. inconsistant) alors le programme $\boldsymbol{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ est aussi consistant (resp. inconsistant).*

**Preuve 4.4.1.** *La preuve est déduite directement à partir du théorème 4.3.1*

**Exemple 77.** *Reprenons le programme $P_{73} = (\mathcal{F}_{73}, \mathcal{R}_{73})$ de l'exemple 73 avec la requête*

$$Q'_{73} = \texttt{ans} \leftarrow \texttt{c1}.$$

*Nous avons alors les answer set :*

$$AS_1^Q = \{\texttt{ad(toto)}, \texttt{d(toto)}, \texttt{pU(titi)}, \texttt{d(titi)}, \texttt{et(titi)}, \texttt{cA(toto)}, \texttt{a(titi)}, \texttt{cA(titi)}\}$$
$$AS_2^Q = \{\texttt{ad(toto)}, \texttt{d(toto)}, \texttt{pU(titi)}, \texttt{d(titi)}, \texttt{et(titi)}, \texttt{cA(toto)}, \texttt{eC(titi)}, \texttt{phdS(titi)}\}$$

*Nous pouvons observer que le seul moyen d'obtenir* ans *dans un answer set est d'appliquer $Q'_{73}$ qui a besoin d'une instance de* c1*. Nous pouvons voir aussi que* ans *ne peut déclencher aucune autre règle.* c1 *ne peut pas appartenir à un answer set car le programme serait inconsistant. Donc si $P_{73}$ est consistant alors $\boldsymbol{P}_{73}$ est aussi consistant. Si nous ajoutons* eC(titi) *à notre base de faits alors nous avons $P_{73}$ et $\boldsymbol{P}_{73}$ inconsistants car nous déduisons toujours* phdS(titi) *ce qui déclenche la règle $r_7$ :* (c1 ← mC(X), phdS(X), not c1.) *et* ans *ne peut pas rétablir la consistance car il n'apparaît dans aucun corps de règle.*

**Théorème 4.4.2** (Existence d'une réponse)**.** *Il existe une réponse non* absurde *à une requête dans un programme ASP si et seulement si le programme est consistant.*

**Preuve 4.4.2.** *Soit $P$ un programme et $Q$ une requête sur $P$. D'après le théorème 4.4.1, si $P$ est inconsistant alors $\boldsymbol{P}$ est inconsistant, donc d'après les définitions 4.3.3 et 4.3.4, la réponse dans $\boldsymbol{P}$ est* absurde*.*
*À l'inverse si P est consistant alors $\boldsymbol{P}$ aussi et la réponse à $Q$ ne peut pas être* absurde*.*

**Exemple 78.** *Considérons de nouveau le programme $P_{73}$ de l'exemple 73. Ce programme est consistant car il possède deux answer set. Quelle que soit $Q$ une requête sur $P_{73}$ il existe une réponse dans $\boldsymbol{P}_{73} = (\mathcal{F}_{73}, \mathcal{R}_{73} \cup \{Q\})$. Maintenant si nous ajoutons à la base de faits $\mathcal{F}_{73}$ l'atome* eC(titi) *alors il n'existe plus d'answer set à $P_{73}$ qui est donc inconsistant. Nous ne pouvons donc pas trouver de réponse à $Q$ dans $\boldsymbol{P}_{73}$ car il n'existe pas d'answer set, la réponse est donc* absurde*.*

Nous savons que l'inconsistance d'un programme est liée aux cycles d'inconsistance mais ce que nous souhaitons c'est isoler les règles d'un programme pouvant être responsables de l'inconsistance. Pour cela nous allons nous intéresser aux *règles dangereuses* de [8]. Les règles identifiées comme dangereuses sont toutes les règles d'un programme dont l'application est susceptible de le rendre inconsistant. Ce sont toutes les règles dont un cycle d'inconsistance dépend. Nous associons alors un ensemble de règles dangereuses à chaque cycle d'inconsistance. Une fois les règles dangereuses identifiées sous forme d'ensembles nous verrons ensuite qu'il suffit de tester leur consistance pour déterminer si un programme est consistant ou non. Une différence par rapport à [8] est que les règles dangereuses sont signées en positives et négatives afin de différencier les règles dangereuses positives, qui peuvent rendre un programme inconsistant, des règles dangereuses négatives, qui (à l'inverse) si elles sont appliquées pourront bloquer l'application d'une règle dangereuse positive qui aurait rendu le programme inconsistant ou bien l'application d'une règle dangereuse négative pouvant déjà bloquer une autre règle.

**Définition 4.4.3** (Règle dangereuse)**.** *Soit $\mathcal{G}$ le graphe de dépendance des symboles de prédicat d'un programme ASP. Un symbole de* prédicat *est dit* dangereux positif *si :*

- *celui-ci apparaît dans un* cycle d'inconsistance *de $\mathcal{G}$ ;*
- *celui-ci apparaît dans le corps positif d'une règle avec un symbole de prédicat dangereux positif en tête.*

*Un symbole de* prédicat *est dit* dangereux négatif *si celui-ci n'est pas dangereux positif et si :*

- *celui-ci apparaît dans le corps négatif d'une règle avec un symbole de prédicat dangereux positif en tête ;*
- *celui-ci apparaît dans le corps (positif ou négatif) d'une règle avec un symbole de prédicat dangereux négatif en tête.*

*Une* règle *est dite* dangereuse positive *(resp.* négative*) si celle-ci possède un symbole de prédicat dangereux positif (resp. négatif) en tête.*
*Pour un cycle d'inconsistance de $\mathcal{G}$, on appelle ensemble de règles dangereuses l'ensemble des règles dangereuses positives et négatives dont ce cycle dépend.*

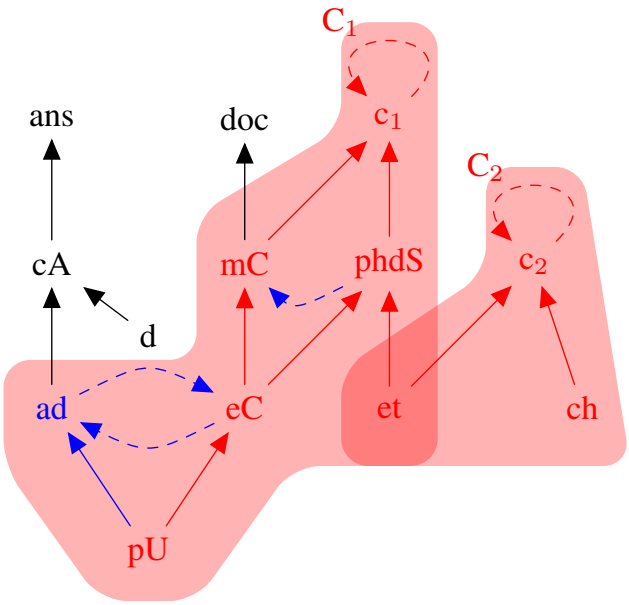
Il existe autant d'ensembles de règles dangereuses associées à un graphe qu'il existe de cycles d'inconsistance.

Pour déterminer si un programme est inconsistant nous avons besoin des deux types de règles dangereuses, chaque type de règle ayant son rôle. Une instance d'une règle dangereuse positive peut-être directement à l'origine d'une inconsistance et nécessite que nous calculions toutes les instances de la règle pour déterminer si le programme est inconsistant. Une règle dangereuse négative peut bloquer une autre règle dangereuse pouvant empêcher une inconsistance, si la règle bloquée est dangereuse positive, ou la rétablir, si celle-ci est dangereuse négative, et n'a besoin que des instances pouvant bloquer les instances d'une règle dangereuse. Les règles dangereuses dans leur ensemble servent à traiter l'inconsistance mais nous verrons dans la partie 4.6.2 que les règles dangereuses négatives peuvent servir à optimiser l'instanciation. Une règle dangereuse ne peut pas être positive et négative à la fois, une règle dangereuse positive nécessitant toutes ses instances prévaudra sur une règle dangereuse négative ne nécessitant que les instances pouvant bloquer une autre règle dangereuse.

**Exemple 79.** *Nous considérons le programme de l'exemple 73 et le graphe de dépendance des symboles de prédicat associé en figure 4.2 dans lequel sont identifiés les deux cycles d'inconsistance de $P_{73}$ (entourés en rouge) notés $C_1$ et $C_2$, ainsi que les symboles de prédicat dangereux positifs (resp. négatifs) en rouge (resp. bleu).*
*Nous pouvons identifier deux ensembles de règles dangereuses dans $P_{73}$, représentés par les règles dont le symbole de prédicat en tête fait partie des symboles de prédicat entourés par une forme rouge : les règles dangereuses dont dépend le cycle $C_1$ et celles dont dépend le cycle $C_2$. Pour $C_1$, nous avons un cycle d'inconsistance sur* `c1` *qui dépend de lui même négativement donc* `c1` *est dangereux positif et* `mC` *et* `phdS` *dont dépend* `c1` *sont aussi dangereux positif de même pour* `et`, `eC` *et* `pU`*. Les règles $r_2, r_4, r_5, r_7$ ayant un de ces symboles de prédicat en tête sont donc des règles dangereuses positives. Le symbole de prédicat* `ad` *dont dépend négativement* `eC` *est dangereux négatif. La règle $r_1$ est donc dangereuse négative.* `mC` *dépend aussi négativement du symbole de prédicat* `phdS` *mais étant donné que celui-ci est déjà dangereux positif il reste dangereux positif. Pour le cycle d'inconsistance $C_2$ nous avons un cycle sur $c2$ avec* `et` *et* `ch` *qui sont dangereux positifs donc $r_8$ est dangereuse positive. Pour finir aucun cycle ne dépend de* `chr`, `cA`, `d` *et* `ans` *donc les règles $r_3, r_6$ et $Q$ ne sont pas dangereuses.*

FIGURE 4.2 – Symbole de prédicat dangereux dans le graphe de dépendance des symboles de prédicat de l'ensemble $\mathcal{R}_{73} \cup \{Q_{73}\}$.

$r_1$ : ad(X) ← pU(X), not eC(X).
$r_2$ : eC(X) ← pU(X), not ad(X).
$r_3$ : cA(X) ← ad(X), d(X).
$r_4$ : mC(X) ← eC(X), not phdS(X).
$r_5$ : phdS(X) ← eC(X), et(X).
$r_6$ : chr(X) ← eC(X).
$r_7$ : c1 ← mC(X), phdS(X), not c1.
$r_8$ : c2 ← et(X), ch(X), not c2.
$Q$ : ans(X) ← cA(X).




Dans l'algorithme 1, nous traitons chaque cycle d'inconsistance afin de récupérer toutes les règles dangereuses qui leur sont associées. Nous utilisons pour cela un ensemble de couples règle/marquage pour stocker les règles dangereuses en différenciant les règles dangereuses positives des négatives. Pour chaque cycle, nous détectons tout d'abord l'ensemble des règles dangereuses positives associées aux symboles de prédicat du cycle. Nous appelons dans un premier temps la fonction $\texttt{CycleInconsistant}(\mathcal{G}, \mathcal{R}, +)$ qui à partir d'un graphe de dépendance des symboles de prédicat $\mathcal{G}$ renvoie à chaque appel l'ensemble des règles de $\mathcal{R}$ avec un symbole de prédicat en tête appartenant à un cycle d'inconsistance non parcouru et les marque positivement. Si tous les cycles ont été parcourus la fonction renvoie un ensemble vide. Soit $D$ un ensemble de couples règle/marquage, $\texttt{dRule}(D)$ (resp. $\texttt{dRuleP}(D)$, $\texttt{dRuleN}(D)$) renvoie les règles dangereuses (resp. positives, négatives) de $D$ sans le marquage. Une fois les règles d'un cycle calculées nous ajoutons à cet ensemble toutes les règles du programme dont le cycle dépend. Dans un premier temps nous marquons les règles dangereuses positives puis les négatives. Soit $pred(A)$ l'ensemble des symboles de prédicat de l'ensemble d'atomes $A$. Nous parcourons donc toutes les règles dangereuses positives dont un cycle dépend et nous utilisons la fonction $\texttt{dangerous}(p, \mathcal{R}, +)$ sur les symboles de prédicat du corps positif, qui à partir d'un symbole de prédicat p et de l'ensemble de règles du programme $\mathcal{R}$ renvoie l'ensemble des règles qui possèdent ce symbole de prédicat en tête et les marque positivement. Une fois l'ensemble des règles dangereuses positives dépendant du cycle calculées on va appliquer la fonction $\texttt{dangerous}(p, \mathcal{R}, -)$ sur les corps négatifs de cet ensemble qui à partir d'un symbole de prédicat p et de $\mathcal{R}$ renvoie l'ensemble des règles qui possèdent ce symbole de prédicat en tête et les marque négativement à moins qu'elles ne soient déjà marquées positivement. Nous terminons en utilisant $\texttt{dangerous}(p, \mathcal{R}, -)$ sur le corps (positif et négatif) des règles dangereuses négatives. L'ensemble $\mathcal{D}$ contient tous les ensembles de règles dangereuses marquées détectés.

**Théorème 4.4.3** (Complétude et correction de l'algorithme 1)**.** *L'algorithme 1 calcule l'ensemble des règles dangereuses, comme défini dans la définition 4.4.3, dans l'ensemble $\mathcal{D}$.*

**Preuve 4.4.3** (Détection des règles dangereuses)**.** *Soit l'ensemble $\mathcal{R}$ des règles d'un programme $P$ et*

**Algorithme 1 :** Détection règles dangereuses

**Data** : L'ensemble $\mathcal{R}$ des règles du programme, le graphe $\mathcal{G}$ de dépendance des symboles de prédicat
**Result** : L'ensemble $\mathcal{D}$ des ensembles de règles dangereuses marquées
**while** $D := CycleInconsistant(\mathcal{G}, \mathcal{R}) \neq \emptyset$ **do**
  **foreach** $r \in dRuleP(D)$ **do**
    **foreach** $p \in pred(body^+(r))$ **do**
      $D := D \cup dangerous(p, \mathcal{R}, +)$;
    **end**
  **end**
  **foreach** $r \in dRuleP(D)$ **do**
    **foreach** $p \in pred(body^-(r))$ **do**
      $D := D \cup dangerous(p, \mathcal{R}, -)$;
    **end**
  **end**
  **foreach** $r \in dRuleN(D)$ **do**
    **foreach** $p \in pred(body(r))$ **do**
      $D := D \cup dangerous(p, \mathcal{R}, -)$;
    **end**
  **end**
  $\mathcal{D} := \mathcal{D} \cup \{D\}$;
**end**
**return** $\mathcal{D}$

*le graphe $\mathcal{G}$ de dépendance des symboles de prédicat de $P$. Nous avons $\mathcal{D}$ l'ensemble des ensembles de règles dangereuses marquées de $P$ à l'issue de l'algorithme 1. Supposons $d_r^+$ une règle dangereuse positive telle que $d_r^+ \notin D$ avec $D \in \mathcal{D}$. D'après la définition 4.4.3 une règle est dangereuse positive si son symbole de prédicat en tête appartient à un cycle d'inconsistance ou au corps positif d'une règle dangereuse positive. Si $d_r^+$ appartient à un cycle d'inconsistance alors $(d_r^+, +) \in D$ avec $D \in \mathcal{D}$ suite à l'application de la fonction $\texttt{CycleInconsistant}(\mathcal{G}, \mathcal{R})$ qui calcule toutes les règles ayant un symbole de prédicat du cycle d'inconsistance en tête. $d_r^+$ ne peut donc pas appartenir à un cycle auquel cas il appartiendrait à un ensemble de $\mathcal{D}$. Dans ce cas il existe un symbole de prédicat $\texttt{p} \in pred(\texttt{tête}(d_r^+))$ tel que $\texttt{p} \in pred(\texttt{corps}^+(r^+))$ avec $(r^+, +) \in D$ une règle dangereuse positive. L'algorithme 1 calcule, pour chaque cycle d'inconsistance $C$, toutes les règles $r \in \mathcal{R}$ avec un symbole de prédicat $\texttt{p} \in pred(\texttt{tête}(r))$ tel que $\texttt{p} \in pred(\texttt{corps}^+(r^+))$ avec $(r^+, +) \in D$ et $D$ l'ensemble des règles dépendantes du cycle d'inconsistance $C$, ce qui revient à parcourir toutes les règles dont le cycle $C$ dépend. Si $d_r^+$ n'est pas détectée alors $\texttt{tête}(d_r^+) \notin \texttt{corps}^+(r^+)$, et $d_r^+$ ne peut donc pas être dangereuse positive sans appartenir à un ensemble de $\mathcal{D}$.*
*Supposons maintenant $d_r^-$ une règle dangereuse négative non détectée par l'algorithme 1. D'après la définition 4.4.3, une règle dangereuse est négative si elle n'est pas positive et que son symbole de prédicat en tête apparaît dans le corps négatif d'une règle dangereuse positive ou dans le corps d'une règle dangereuse négative. Si $(d_r^-, +) \in D$ avec $D \in \mathcal{D}$ alors $(d_r^-, -) \notin D$ d'après la définition de la fonction $\texttt{dangerous}$ qui marque négativement une règle seulement si elle n'est pas déjà marquée positivement, donc $(d_r^-, +) \notin D$. Soit un symbole de prédicat $\texttt{p} \in pred(\texttt{tête}(d_r^-))$ tel que $\texttt{p} \in pred(\texttt{corps}^-(r^+))$ avec $(r^+, +) \in D$ alors $d_r^- \in D$ car toutes les règles dangereuses positives sont déjà détectées par l'algorithme et la seconde boucle ajoute toutes les règles dangereuses négatives respectant cette condition. Si $\texttt{p} \in pred(\texttt{tête}(d_r^-))$ tel que $\texttt{p} \in pred(\texttt{corps}(r^-))$ avec $(r^-, -) \in D$ alors $d_r^- \in D$ car l'ensemble des règles dangereuses négatives pouvant provenir d'une règle*

*dangereuse positive sont détectées lors de la deuxième boucle de l'algorithme, la troisième boucle détecte ainsi toutes les règles dangereuses négatives issues du corps des règles dangereuses négatives détectées précédemment.* $d_r^-$ *ne peut donc pas être dangereuse négative sans appartenir à un ensemble de* $\mathcal{D}$*. Cet algorithme est donc complet car toutes les règles dangereuses sont stockées.*
*Supposons maintenant qu'il existe une règle* $r$ *non dangereuse appartenant à un ensemble de* $\mathcal{D}$*, alors le symbole de prédicat* $\texttt{p} \in pred(tête(r))$ *n'appartient ni à un cycle d'inconsistance de* $\mathcal{G}$ *ni au corps d'une règle dangereuse. L'algorithme stocke dans un premier temps les règles dont le symbole de prédicat en tête appartient à un cycle d'inconsistance,* $r$ *n'est donc pas stockée à ce moment là. Par la suite nous parcourons les règles dangereuses de* $\mathcal{D}$ *afin de stocker les règles ayant un symbole de prédicat en tête appartenant au corps (positif ou négatif) des règles parcourues. Les règles stockées dans* $\mathcal{D}$ *à ce moment sont des règles dangereuses (positives ou négatives) car elles appartiennent au corps d'une règle dangereuse. Nous avons donc parcouru toutes les règles stockées dans* $\mathcal{D}$ *sans pouvoir stocker* $r$*. L'algorithme est donc correct car il est impossible de stocker une règle non dangereuse dans* $\mathcal{D}$*.*

**Exemple 80.** *Reprenons l'ensemble de règles de l'exemple 73*

$$\mathcal{R}_{73} = \left\{ \begin{array}{l} r_1 : \texttt{ad(X)} \leftarrow \texttt{pU(X)}, \texttt{ not eC(X)}. \\ r_2 : \texttt{eC(X)} \leftarrow \texttt{pU(X)}, \texttt{ not ad(X)}. \\ r_3 : \texttt{cA(X)} \leftarrow \texttt{ad(X)}, \texttt{ d(X)}. \\ r_4 : \texttt{mC(X)} \leftarrow \texttt{eC(X)}, \texttt{ not phdS(X)}. \\ r_5 : \texttt{phdS(X)} \leftarrow \texttt{eC(X)}, \texttt{et(X)}. \\ r_6 : \texttt{chr(X)} \leftarrow \texttt{eC(X)}. \\ r_7 : \texttt{c1} \leftarrow \texttt{mC(X)}, \texttt{ phdS(X)}, \texttt{ not c1}. \\ r_8 : \texttt{c2} \leftarrow \texttt{et(X)}, \texttt{ch(X)}, \texttt{ not c2}. \end{array} \right\}$$

*Dans cet exemple nous appliquons l'algorithme 1 pour construire l'ensemble d'ensembles de règles dangereuses* $\mathcal{D}_{73}$ *de* $\mathcal{R}_{73}$*. Soit* $\mathcal{G}_{73}$ *le graphe des symboles de prédicat de* $P_{73}$*, il existe deux cycles d'inconsistance dans* $\mathcal{G}_{73}$ *le premier sur le symbole de prédicat* $c1$ *et le second sur le symbole de prédicat* c2*. Le premier ensemble de règles dangereuses* $D_1$ *sera donc constitué d'abord de la règle* $r_7$ *marquée positivement car* c1 *est le seul symbole de prédicat composant le cycle d'inconsistance* $C_1$ *et* $r_7$ *la seule règle ayant* c1 *en tête, donc* $D_1 = \{(r_7, +)\}$*. Nous continuons en regardant l'ensemble des symboles de prédicat dont dépend* c1*, nous avons* mC *et* phdS *qui sont des symbole de prédicat dangereux positifs ce qui ajoute* $r_4$ *et* $r_5$ *à* $D_1$*, étant donné que ces symboles de prédicat apparaissent en tête de ces règles ce qui donne* $D_1 = \{(r_7, +), (r_4, +), (r_5, +)\}$*. De la même manière nous avons* eC *dont* mC *et* et *dépendent positivement ce qui fait que* $r_2$ *est dangereuse positive et* pU *est un fait donc n'engendre pas de règle dangereuse. Nous avons alors* $D_1 = \{(r_7, +), (r_4, +), (r_5, +), (r_2, +)\}$*, si nous regardons les dépendances de* eC *nous remarquons que celui-ci dépend négativement de* ad *ce qui induit que* $r_1$ *est une règle dangereuse négative, de même avec la dépendance de* mC *avec* phdS *et* ad *qui dépend négativement de* eC *mais cela n'ajoute pas de règle dangereuse négative car celles-ci sont déjà dangereuses positives. Nous avons alors* $D_1 = \{(r_7, +), (r_4, +), (r_5, +), (r_2, +), (r_1, -)\}$*, et il n'existe pas d'autres dépendances au cycle d'inconsistance* $C_1$*.*
*Le second ensemble de règles dangereuses* $D_2$ *est issu du cycle d'inconsistance* $C_2$ *et est composé uniquement du symbole de prédicat* c2*. Nous avons alors* $r_8$ *règle dangereuse positive, donc* $D_2 = \{(r_8, +)\}$ *car les seuls symboles de prédicat dont dépend le symbole de prédicat* c2 *sont* et *et* ch *qui n'apparaissent dans aucune tête de règle. L'algorithme s'arrête ainsi car toutes les dépendances de tous les cycles d'inconsistances ont été parcouru, nous avons alors l'ensemble des ensembles de règles dangereuses de* $\mathcal{R}_{73}$*,* $\mathcal{D}_{73} = \{D_1, D_2\}$*.*

Afin de différencier les règles dangereuses positives et les règles dangereuses négatives permettant ensemble la détection de l'inconsistance, nous montrons dans l'exemple 81 l'influence de chacune d'entre elles sur le calcul d'un answer set.

**Exemple 81.** *Prenons le programme* $P'_{73} = (\mathcal{F}'_{73}, \mathcal{R}_{73})$ *l'ensemble de règles de l'exemple 80 avec la base de faits* $\mathcal{F}'_{73} = \{\texttt{eC(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}\}$. *Nous avons alors* $P'_{73}$ *qui est inconsistant à cause de la règle* $r_7$ : $(\texttt{c1} \leftarrow \texttt{mC(X)}, \texttt{phdS(X)}\ \texttt{not c1}.)$ *car il n'existe pas d'ensemble d'atomes* $A = Cn(\mathbf{G}(P'_{73})^A)$. *L'ensemble d'atomes le plus proche serait* $A = \{\texttt{eC(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}, \texttt{phdS(titi)}, \texttt{chr(titi)}, \texttt{c1}\}$ *mais le modèle de Herbrand serait* $Cn(\mathbf{G}(P'_{73})^A) = \{\texttt{eC(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}, \texttt{phdS(titi)}, \texttt{chr(titi)}\}$ *car la règle* $r_7$ *disparaît avec le réduit. Nous voyons donc que les règles* $r_5$ *et* $r_7$ *seules permettent d'avoir un programme inconsistant et que c'est le déclenchement de* $r_7$ *qui est à l'origine de l'inconsistance. Si nous prenons maintenant* $P''_{73} = (\mathcal{F}''_{73}, \mathcal{R}_{73})$ *avec* $\mathcal{F}''_{73} = \{\texttt{pU(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}, \texttt{ad(titi)}\}$, *nous obtenons cette fois-ci un answer set* $\{\texttt{pU(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}, \texttt{ad(titi)}, \texttt{chr(titi)}\}$. *Ici nous avons* ad *qui est un symbole de prédicat dangereux négatif qui permet d'empêcher l'inconsistance provoquée par le symbole de prédicat* eC *vu précédemment. De même si nous avions* $\{\texttt{eC(titi)}, \texttt{phdS(titi)}\}$ *l'inconsistance serait empêchée car la règle* $r_4$ *ne peut pas être déclenchée et empêche donc le déclenchement de* $r_7$ *responsable de l'inconsistance. Contrairement aux symboles de prédicat et règles dangereuses positives qui sont la cause de l'inconsistance, les symboles de prédicat et les règles dangereuses négatives peuvent permettre d'empêcher une inconsistance. Mais une règle dangereuse négative peut aussi rendre son inconsistance à un programme si elle empêche une autre règle dangereuse négative de s'appliquer. Soit le programme* $P_{81} = (\mathcal{F}_{81}, \mathcal{R}_{81})$ *avec*

$$\mathcal{R}_{81} = \left\{ \begin{array}{l} r'_1 : \texttt{ad(X)} \leftarrow \texttt{pU(X)}, \texttt{biatss(X)}. \\ r'_2 : \texttt{eC(X)} \leftarrow \texttt{pU(X)}, \texttt{not prag(X)}. \\ r_3 : \texttt{cA(X)} \leftarrow \texttt{ad(X)}, \texttt{d(X)}. \\ r_4 : \texttt{mC(X)} \leftarrow \texttt{eC(X)}, \texttt{not phdS(X)}. \\ r_5 : \texttt{phdS(X)} \leftarrow \texttt{eC(X)}, \texttt{et(X)}. \\ r_6 : \texttt{chr(X)} \leftarrow \texttt{eC(X)}. \\ r_7 : \texttt{c1} \leftarrow \texttt{mC(X)}, \texttt{phdS(X)}, \texttt{not c1}. \\ r_8 : \texttt{c2} \leftarrow \texttt{et(X)}, \texttt{ch(X)}, \texttt{not c2}. \\ r_9 : \texttt{prag(X)} \leftarrow \texttt{pU(X)}, \texttt{nonchercheur(X)}, \texttt{not ad(X)}. \end{array} \right\}$$

*et* $\mathcal{F}_{81} = \{\texttt{pU(titi)}, \texttt{nonchercheur(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}\}$. *Sur ce programme nous savons que Titi est un personnel universitaire, administratif et non chercheur. Si la règle dangereuse positive* $r'_2$ *est appliquée alors le programme est inconsistant, seulement elle est bloquée par l'application de la règle dangereuse négative* $r_9$ *donc le programme reste consistant et nous avons un answer set. Si nous ajoutons maintenant* $\texttt{biatss(titi)}$ *le programme devient inconsistant car la règle dangereuse négative* $r'_1$ *empêche l'application de la règle dangereuse négative* $r'_2$ *qui empêchait l'inconsistance. Ainsi il faut faire attention car nous avons une règle dangereuse négative qui rétablit l'inconsistance de notre programme. Donc les règles dangereuses positives peuvent créer une inconsistance tandis que les règles dangereuses négatives peuvent soit empêcher une inconsistance ou alors rétablir une inconsistance empêchée par une autre règle dangereuse négative.*

Les ensembles de règles dangereuses maintenant isolées dans l'ensemble $\mathcal{D}$ nous souhaitons calculer uniquement à partir de l'ensemble $\mathcal{D}$ si un programme est consistant. Jusqu'ici il est possible d'avoir une redondance de règles dangereuses entre deux ensembles de $\mathcal{D}$. Nous souhaitons supprimer ces redondances pour que par la suite un traitement ne soit pas effectué deux fois sur une même règle. De plus, deux ensembles de règles dangereuses peuvent être consistants un à un avec une base de faits alors que l'union de ces deux ensembles est inconsistant.

**Exemple 82.** *Soit* $P_{82}$ *un programme constitué de l'ensemble de règles*

$$\mathcal{R}_{82} = \left\{ \begin{array}{l} r_1 : \texttt{t(X)} \leftarrow \texttt{b(X)}, \texttt{not t(X)}. \\ r_2 : \texttt{s(X)} \leftarrow \texttt{d(X)}, \texttt{not s(X)}. \\ r_3 : \texttt{b(X)} \leftarrow \texttt{a(X)}, \texttt{not d(X)}. \\ r_4 : \texttt{d(X)} \leftarrow \texttt{c(X)}, \texttt{not b(X)}. \end{array} \right\}$$

*et la base de faits* $\mathcal{F}_{82} = \{\texttt{a(1)}, \texttt{c(1)}\}$. *Nous avons ici deux ensembles de règles dangereuses* $D_1 = \{(r_1, +), (r_3, +), (r_4, -)\}$ *et* $D_2 = \{(r_2, +), (r_4, +), (r_3, -)\}$. *Si nous calculons les answer set pour chacun des ensembles séparément nous obtenons un answer set* $AS_1 = \{\texttt{a(1)}, \texttt{c(1)}, \texttt{d(1)}\}$ *pour* $D_1$ *et un answer set* $AS_2 = \{\texttt{a(1)}, \texttt{c(1)}, \texttt{b(1)}\}$ *pour* $D_2$. *Le problème est que si nous calculons les answer set de* $P_{82}$, *qui correspond à l'union des règles dangereuses non marquées des ensembles* $D_1$ *et* $D_2$, *le programme ne possède pas d'answer set. En effet, les règles* $r_3$ *et* $r_4$ *appartiennent toutes deux à la fois à* $D_1$ *et* $D_2$, *l'application de* $r_3$ *permet d'obtenir* $AS_2$ *mais empêche* $AS_2$ *d'être un answer set de* $D_1$ *tandis que l'application de* $r_4$ *permet d'obtenir* $AS_1$ *mais empêche* $AS_1$ *d'être un answer set de* $D_2$.

L'algorithme 2 calcule l'ensemble $\Delta$ qui correspond aux unions des ensembles de règles dangereuses de $\mathcal{D}$ qui possèdent au moins une règle en commun. Nous parcourons tous les ensembles de règles dangereuses en cherchant les ensembles ayant une règle en commun, nous faisons l'union de ces ensembles et nous retirons de $\mathcal{D}$ un des deux ensembles. Pour cela nous utilisons l'opérateur `unionRule` qui calcule l'union de deux ensembles de règles dangereuses, avec la particularité de ne garder que les règles dangereuses marquées positivement lorsqu'une règle apparaît positive dans un ensemble et négative dans le second.

**Algorithme 2 :** Ensemble des ensembles de règles dangereuses unifiées

**Data** : L'ensemble $\mathcal{D}$ des ensembles de règles dangereuses
**Result** : L'ensemble $\Delta$ des ensembles de règles dangereuses unifiées
$\Delta = \mathcal{D}$;
**foreach** $D_i \in \Delta$ **do**
  **foreach** $D_j \in \Delta \backslash D_i$ **do**
    **if** $dRule(D_i) \cap dRule(D_j) \neq \emptyset$ **then**
      $D_i := D_i \; unionRule \; D_j$;
      $\Delta := \Delta \backslash D_j$;
    **end**
  **end**
**end**
**return** $\Delta$

**Théorème 4.4.4** (Inconsistance et règles dangereuses)**.** *Un programme* $P = (\mathcal{F}, \mathcal{R})$ *est inconsistant si et seulement si* $P' = (\mathcal{F}, \bigcup_{D \in \Delta} D)$ *est inconsistant.*

**Preuve 4.4.4.** *Supposons qu'il existe un programme* $P = (\mathcal{F}, \mathcal{R})$ *inconsistant, avec* $\mathcal{G}$ *son graphe des symboles de prédicat, tel que* $P' = (\mathcal{F}, \bigcup_{D \in \Delta} D)$ *possède un answer set. Si* $P$ *est inconsistant alors d'après le théorème 4.4* inconsistance *(sans numérotation) il existe un cycle d'inconsistance dans* $\mathcal{G}$ *le graphe des symboles de prédicat de* $P'$. *D'après la définition 4.4.3, il existe un ensemble de règles dangereuses pour chaque cycle d'inconsistance de* $\mathcal{G}$, *les règles dangereuses d'un ensemble sont toutes les règles dont dépend un cycle d'inconsistance. Soit* $D \in \Delta$ *un ensemble de règles dangereuses responsable de l'inconsistance dans* $P$, *d'après l'algorithme 1,* $D$ *contient alors toutes les règles dépendantes d'au moins un cycle d'inconsistance de* $\mathcal{G}$. *L'algorithme 2 calcule l'union des règles dangereuses de chaque ensemble de* $\mathcal{D}$ *contenant l'ensemble des règles rendant* $P$ *inconsistant. Les seules règles écartées sont les règles n'ayant aucune dépendance avec les cycles d'inconsistance*

*et ne pouvant pas rendre le programme inconsistant. L'ensemble $D$ appartient donc à $\Delta$. S'il existe un answer set pour $P'$, $P$ ne peut pas être inconsistant étant donné que les cycles d'inconsistance de $P$ ne dépendent que des règles de $D$. Nous avons donc, si $P$ est inconsistant alors $P'$ est aussi inconsistant.*
*Supposons maintenant que $P'$ est inconsistant et que $P$ possède un answer set. Si $P'$ est inconsistant alors d'après le théorème 4.4 il existe un cycle d'inconsistance dans $\mathcal{G}'$ et donc $\Delta' \neq \emptyset$ avec $\Delta'$ l'ensemble des ensembles de règles dangereuses unifiées du programme $P'$. D'après les algorithmes 1 et 2, $\bigcup_{D\in\Delta} D$ contient toutes les règles dangereuses de l'ensemble de règles $\mathcal{R}$. Le programme $P'$ contient donc toutes les règles dangereuses de $P$, l'ensemble $\Delta$ étant construit à partir des dépendances des règles dangereuses dont le symbole de prédicat en tête appartient à un cycle d'inconsistance, ces règles dangereuses appartiennent à $\Delta$ et donc appartiennent aussi à $P$, ainsi que toutes les règles dont elles dépendent (c'est le résultat de l'algorithme 1). Étant donné que $\Delta$ contient toutes les règles pouvant rendre le programme $P$ inconsistant, nous pouvons en déduire qu'en calculant $\Delta'$ nous calculons les cycles d'inconsistance de l'ensemble de règles $\mathcal{R}$. Nous avons donc $\Delta' = \Delta$. De plus nous avons $\mathcal{R}\backslash(\bigcup_{D\in\Delta} D)$ un ensemble de règles n'ayant aucune dépendance avec les règles appartenant aux ensembles de $\Delta$. Donc si $P'$ est inconsistant il n'existe aucune règle appartenant à $\mathcal{R}$ pouvant modifier la consistance du programme $P$. Donc si $P'$ est inconsistant alors $P$ ne peut pas être consistant.*

Afin d'assurer la consistance nous réutilisons l'ensemble retourné par l'algorithme 2 et nous calculons pour chaque ensemble de règles dangereuses un premier answer set. D'après le théorème 4.4.4, si l'algorithme trouve un premier answer set pour l'ensemble $\Delta$ alors le programme est consistant. Si la consistance est vérifiée alors nous pouvons nous intéresser aux règles dont la requête dépend, sinon la réponse est `absurde`. Nous notons que la phase de détection d'inconsistance est détachée de l'interrogation étant donné que nous pouvons effectuer celle-ci en pré-traitement sans influencer le temps de réponse pour l'interrogation. Nous insistons sur le fait que la détection d'inconsistance sur des ontologies avec exceptions est peu complexe étant donné que les données des ontologies sont très souvent déconnectées. Les ensembles de règles dangereuses seront donc des ensembles de règles de petite taille, correspondant à une petite partie du programme original, simples à traiter. A contrario, les problèmes combinatoires sont souvent constitués de règles très dépendantes les unes des autres, ce qui implique que les ensembles de règles dangereuses correspondent à un sous-ensemble contenant une grande partie du programme original.

## 4.5 Dépendance de la requête

Le but étant de limiter le nombre de règles nécessaires à l'obtention d'une réponse à notre requête, nous cherchons à isoler l'ensemble des règles suffisantes pour répondre à celle-ci. Nous cherchons dans un premier temps à isoler l'ensemble des règles dont dépend directement la requête et permettant de générer une instance de l'atome réponse lorsqu'elles sont appliquées. Mais nous verrons que ces règles ne sont pas suffisantes pour obtenir une réponse sur le programme complet, et que nous devons étendre cet ensemble aux ensembles de règles dangereuses possédant une règle en commun entre elles, car elles peuvent empêcher l'appartenance d'une instance de l'atome réponse à un answer set du programme complet. Une fois ces ensembles de règles isolés, nous avons réduit considérablement le nombre de règles de notre programme nécessaires pour répondre à notre requête. Nous rappelons qu'avec les ontologies il est fréquent d'avoir des données très peu connectées entre elles, il est donc particulièrement intéressant d'écarter les règles qui n'ont pas d'impact sur la réponse à notre requête.

**Définition 4.5.1** (Dépendance de la requête)**.** *Une règle $r_b$* dépend *d'une règle $r_a$ si :*

- *soit le symbole de prédicat $p \in pred(\texttt{tête}(r_a))$ appartient à $pred(\texttt{corps}(r_b))$ ;*

- *soit $r_b$ dépend d'une règle qui dépend de $r_a$.*

Notons que la définition 4.5.1 est récursive et qu'elle définie la plus petite relation satisfaisant les deux propriétés.

Soit $\mathbf{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ le programme ASP $P = (\mathcal{F}, \mathcal{R})$ requêté par $Q$, nous notons $\mathcal{Q}(\mathcal{R})$ l'ensemble des règles de **P** dont dépend $Q$ auquel nous ajoutons $Q$. Nous utilisons ici une notation abusive incluant la requête à l'ensemble de règles du programme, ceci permet de simplifier l'écriture du manuscrit par la suite.
L'algorithme 3 construit l'ensemble de règles $\mathcal{Q}(\mathcal{R})$ à partir de $\mathcal{R}$ et $Q$. L'algorithme initialise $\mathcal{Q}(\mathcal{R})$ à $\{Q\}$ puis parcourt l'ensemble $\mathcal{Q}(\mathcal{R})$ en y ajoutant toutes les règles qui possèdent un symbole de prédicat en tête appartenant au corps d'une règle de $\mathcal{Q}(\mathcal{R})$. Nous utilisons pour cela la fonction `headRule`$(\mathtt{p}, \mathcal{R})$ qui renvoie l'ensemble des règles de $\mathcal{R}$ ayant le symbole de prédicat p en tête.

**Algorithme 3 :** Détection des règles de la requête

**Data** : L'ensemble des règles du programme $\mathcal{R}$, la requête Q
**Result** : L'ensemble des règles $\mathcal{Q}(\mathcal{R})$ dont dépend la requête
$\mathcal{Q}(\mathcal{R})$ := {Q};
**foreach** $r \in \mathcal{Q}(\mathcal{R})$ **do**
  **foreach** $p \in body(r)$ **do**
    $\mathcal{Q}(\mathcal{R}) := \mathcal{Q}(\mathcal{R}) \cup headRule(p, \mathcal{R})$;
  **end**
**end**
**return** $\mathcal{Q}(\mathcal{R})$;

**Théorème 4.5.1** (Correction et complétude de l'algorithme 3)**.** *L'algorithme 3 calcule l'ensemble des règles dont la requête dépend comme défini dans la définition 4.5.1.*

**Preuve 4.5.1.** *Soit $\mathbf{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ un programme ASP. Supposons $r$ une règle dont la requête $Q$ dépend mais n'appartenant pas à $\mathcal{Q}(\mathcal{R})$. D'après la définition 4.5.1 et l'algorithme 3, l'ensemble de règles $\mathcal{Q}(\mathcal{R})$ contient l'ensemble des règles $r'$ telles que son symbole de prédicat en tête appartient au corps d'une règle $r'' \in \mathcal{Q}(\mathcal{R})$ de plus une règle $r'$ dépend d'une règle $r''$ si le symbole de prédicat en tête de $r''$ apparaît dans le corps de $r'$. Donc une règle $r$ ne peut pas dépendre de $Q$ sans appartenir à $\mathcal{Q}(\mathcal{R})$.*
*Supposons maintenant que $r$ ne dépend pas de $Q$ et appartient à $\mathcal{Q}(\mathcal{R})$. Si $r$ appartient à $\mathcal{Q}(\mathcal{R})$ alors le symbole de prédicat en tête de $r$ appartient au corps d'une règle $r' \in \mathcal{Q}(\mathcal{R})$. Si une règle appartient à $\mathcal{Q}(\mathcal{R})$ alors $Q$ dépend de cette règle, étant donné que $r'$ dépend de $r$ et que $Q$ dépend de $r'$ alors $Q$ dépend de $r$ et donc $r$ ne peut pas appartenir à $\mathcal{Q}(\mathcal{R})$ sans que $Q$ dépende de $r$.*

**Exemple 83.** *Soit l'ensemble de règles de l'exemple 73 :*

$$\mathcal{R}_{73} = \left\{ \begin{array}{l} r_1 : \mathtt{ad(X)} \leftarrow \mathtt{pU(X)},\ \mathtt{not\ eC(X)}. \\ r_2 : \mathtt{eC(X)} \leftarrow \mathtt{pU(X)},\ \mathtt{not\ ad(X)}. \\ r_3 : \mathtt{cA(X)} \leftarrow \mathtt{ad(X)},\ \mathtt{d(X)}. \\ r_4 : \mathtt{mC(X)} \leftarrow \mathtt{eC(X)},\ \mathtt{not\ phdS(X)}. \\ r_5 : \mathtt{phdS(X)} \leftarrow \mathtt{eC(X)}, \mathtt{et(X)}. \\ r_6 : \mathtt{chr(X)} \leftarrow \mathtt{eC(X)}. \\ r_7 : \mathtt{c1} \leftarrow \mathtt{mC(X)},\ \mathtt{phdS(X)},\ \mathtt{not\ c1}. \\ r_8 : \mathtt{c2} \leftarrow \mathtt{et(X)}, \mathtt{ch(X)},\ \mathtt{not\ c2}. \end{array} \right\}$$

*et la requête*

$$Q_{73} = \texttt{ans(X)} \leftarrow \texttt{cA(X)}.$$

*Nous avons* $\mathcal{Q}(\mathcal{R}_{73}) = \{Q_{73}, r_3, r_2, r_1\}$,

Maintenant que nous avons isolé l'ensemble des règles dont $Q$ dépend, nous montrons que la réponse à une requête dans $\mathbf{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ est aussi une réponse dans $\mathbf{P}_\mathcal{Q} = (\mathcal{F}, \mathcal{Q}(\mathcal{R}))$ ainsi que sa réciproque lorsque nous considérons un programme ayant toujours au moins un answer set quelque soit la base de faits.

**Définition 4.5.2** (Programme super-consistant)**.** *[7] Un programme* $P = (\mathcal{F}, \mathcal{R})$ *est dit super-consistant si pour toute base de faits* $\mathcal{F}'$*,* $(\mathcal{F}', \mathcal{R})$ *est consistant.*

D'après [7] lorsque nous cherchons une réponse sur un programme super-consistant il n'est pas nécessaire de vérifier l'existence d'un answer set sur les ensembles de règles dangereuses car le programme est défini pour avoir au moins un answer set quels que soient les faits initiaux.

**Théorème 4.5.2** (Réponse à un programme super-consistant)**.** *Soit* $\mathbf{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ *un programme ASP super-consistant. Si* $R$ *est une réponse à* $Q$ *dans* $\mathbf{P}$ *alors* $R$ *est une réponse à* $Q$ *dans* $\mathbf{P}_\mathcal{Q} = (\mathcal{F}, \mathcal{Q}(\mathcal{R}))$.

**Preuve 4.5.2.** *Soit* $\mathbf{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ *un programme ASP super-consistant avec* $Q$ *une requête sur* $\mathbf{P}$*, nous avons* $\mathbf{P}_\mathcal{Q} = (\mathcal{F}, \mathcal{Q}(\mathcal{R}))$ *avec* $\mathcal{Q}(\mathcal{R})$ *l'ensemble des règles dépendantes de la requête* $Q$*. D'après la définition 4.5.2,* $\mathbf{P}$ *possède au moins un answer set quelle que soit sa base de faits. D'après le théorème 4.4.1,* $\mathbf{P}$ *et* $\mathbf{P}_\mathcal{Q}$ *sont donc consistants si nous ne leur ajoutons pas de règles. Supposons qu'il existe une réponse dans* $\mathbf{P}_\mathcal{Q}$ *n'étant pas une réponse dans* $\mathbf{P}$*. Alors il existe une instance de* ans *dans un answer set de* $\mathbf{P}_\mathcal{Q}$ *n'appartenant pas à un answer set de* $\mathbf{P}$*. D'après la définition 4.5.1,* $\mathcal{Q}(\mathcal{R})$ *est l'ensemble des règles de* $\mathcal{R}$ *dont la requête dépend contenant toutes les règles permettant d'obtenir des instances de* ans *donc s'il existe une réponse à* $Q$ *dans* $\mathbf{P}_\mathcal{Q}$ *elle sera aussi une réponse dans* $\mathbf{P}$*.*
*Supposons maintenant qu'il existe une réponse dans* $\mathbf{P}$ *n'apparaissant pas dans* $\mathbf{P}_\mathcal{Q}$*. Étant donné que* $\mathcal{Q}(\mathcal{R})$ *contient toutes les règles permettant d'obtenir des instances de* ans *il ne peut pas exister de réponse sur* $\mathbf{P}$ *n'appartenant pas à* $\mathbf{P}_\mathcal{Q}$*.*

Nous illustrons dans l'exemple 84 que seules les règles de $\mathcal{Q}(\mathcal{R})$ sont nécessaires pour répondre à une requête $Q$ sur un programme super-consistant $P$.

**Exemple 84.** *Soit le programme ASP super-consistant* $\mathbf{P}_{84}$ *composé de l'ensemble de règles* $\mathcal{R}_{84}$ *(*$\mathcal{R}_{73}$ *sans les contraintes) :*

$$\mathcal{R}_{84} = \left\{ \begin{array}{l} r_1 : \texttt{ad(X)} \leftarrow \texttt{pU(X), not eC(X).} \\ r_2 : \texttt{eC(X)} \leftarrow \texttt{pU(X), not ad(X).} \\ r_3 : \texttt{cA(X)} \leftarrow \texttt{ad(X), d(X).} \\ r_4 : \texttt{mC(X)} \leftarrow \texttt{eC(X), not phdS(X).} \\ r_5 : \texttt{phdS(X)} \leftarrow \texttt{eC(X), et(X).} \\ r_6 : \texttt{chr(X)} \leftarrow \texttt{eC(X).} \end{array} \right\}$$

*et de la requête* $Q_{84} = (\texttt{ans(X)} \leftarrow \texttt{cA(X).})$*. Nous commençons par ajouter* $Q_{84}$ *à* $\mathcal{Q}(\mathcal{R}_{84}) = \{Q_{84}\}$*, nous calculons ensuite toutes les règles avec* cA *en tête, donc* $\mathcal{Q}(\mathcal{R}_{84}) = \{Q_4, r_3\}$*. Nous prenons ensuite les symboles de prédicat apparaissant dans le corps de* $r_3$*, soit uniquement* ad *et* d*, seul* ad *apparaît en tête d'une règle donc nous ajoutons la règle* $r_1$ *dans laquelle il apparaît en tête. Nous*

*continuons avec les symboles de prédicat du corps de $r_1$ nous ajoutons ainsi $r_2$. Nous avons donc $\mathcal{Q}(\mathcal{R}_{84}) = \{Q_{84}, r_3, r_1, r_2\}$. Nous avons calculé ainsi toutes les règles dont la requête dépend. Nous pouvons voir qu'il n'est pas possible de créer une instance de* `ans` *avec les règles $r_4, r_5$ et $r_6$ les symboles de prédicat en tête de ces règles n'apparaissant pas dans le corps des règles de l'ensemble $\mathcal{Q}(\mathcal{R}_{84})$. Les règles $r_4, r_5$ et $r_6$ ne sont donc pas nécessaires pour trouver une réponse à $\boldsymbol{P}_{84}$.*

Nous savons maintenant que lorsque notre programme est super-consistant seul l'ensemble de règles $\mathcal{Q}(\mathcal{R})$ dont la requête dépend est suffisant pour répondre à une requête sur **P**. Nous élargissons maintenant cette propriété à tous les programmes consistants en prenant en compte les ensembles de règles dangereuses du programme.

Nous cherchons maintenant à déterminer quelles sont les règles dangereuses dont l'application a un impact sur la réponse à une requête. Pour cela nous nous intéressons aux règles dangereuses appartenant à l'ensemble $\mathcal{Q}(\mathcal{R})$.

**Définition 4.5.3.** *Nous appelons $D\mathcal{Q}(\mathcal{R})$ l'ensemble des règles dangereuses à l'intersection de $\mathcal{Q}(\mathcal{R})$ :* $D\mathcal{Q}(\mathcal{R}) = \{d \in D_i | D_i \in \Delta, dRule(D_i) \cap \mathcal{Q}(\mathcal{R}) \neq \emptyset\}$

L'algorithme 4 permet la détection de l'ensemble des règles dangereuses à l'intersection des règles dont la requête dépend. Pour chaque ensemble de $\Delta$ nous cherchons s'il existe une règle appartenant à $\mathcal{Q}(\mathcal{R})$. Nous ajoutons ainsi tous les ensembles de règles de $\Delta$ à l'intersection de $\mathcal{Q}(\mathcal{R})$ à l'ensemble $D\mathcal{Q}(\mathcal{R})$.

**Algorithme 4 :** Détection des règles dangereuses à l'intersection de la requête

**Data** : L'ensemble $\mathcal{Q}(\mathcal{R})$ des règles dont la requête dépend,
l'ensemble $\Delta$ des ensembles de règles dangereuses marquées
**Result** : L'ensemble des règles dangereuses $D\mathcal{Q}(\mathcal{R})$ à l'intersection de $\mathcal{Q}(\mathcal{R})$
$D\mathcal{Q}(\mathcal{R}) := \emptyset$ ; **foreach** $D \in \Delta$ **do**
  **if** $dRule(D) \cap \mathcal{Q}(\mathcal{R}) \neq \emptyset$ **then**
    $D\mathcal{Q}(\mathcal{R}) := D\mathcal{Q}(\mathcal{R})\ unionRule\ D$
  **end**
**end**

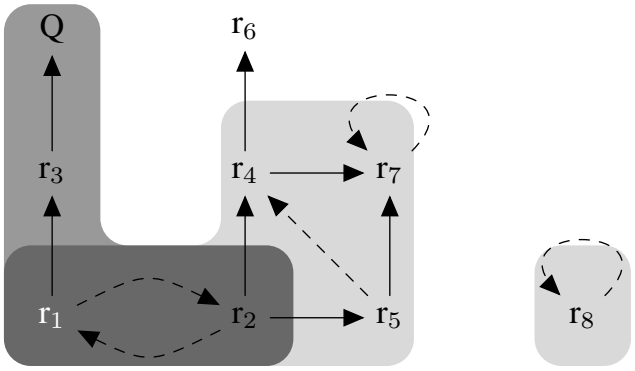
**Théorème 4.5.3** (Correction et complétude de l'algorithme 4)**.** *L'algorithme 4 calcule l'ensemble des règles dangereuses à l'intersection des règles dont la requête dépend comme défini dans la définition 4.5.3.*

**Preuve 4.5.3.** *Soit un programme $P = (\mathcal{F}, \mathcal{R})$ et un ensemble de règles dangereuses $D \in \Delta$ tel que $dRule(D) \cap \mathcal{Q}(\mathcal{R}) \neq \emptyset$. Supposons qu'il existe $D' \in D\mathcal{Q}(\mathcal{R})$ tel que $D \not\subseteq D'$. D'après la définition 4.5.3 et l'algorithme 4 $D\mathcal{Q}(\mathcal{R})$ est l'union (en conservant uniquement le marquage positif si une règle est marquée positivement dans un des ensembles) des ensembles de règles dangereuses $D \in \Delta$ tels que $dRule(D) \cap \mathcal{Q}(\mathcal{R}) \neq \emptyset$. Il existe donc forcément un ensemble $D' \in D\mathcal{Q}(\mathcal{R})$ tel que $D \subseteq D'$.*
*Soit un ensemble de règles $\mathcal{R}$ et un ensemble de règles dangereuses $D \in \Delta$ tel que $dRule(D) \cap \mathcal{Q}(\mathcal{R}) = \emptyset$ et $D \subseteq D'$ avec $D' \in D\mathcal{Q}(\mathcal{R})$. De même d'après la définition 4.5.3 et l'algorithme 4 seuls les ensembles de règles dangereuses tels que $dRule(D) \cap \mathcal{Q}(\mathcal{R}) \neq \emptyset$ sont unifiés. Il est donc impossible que $D \subseteq D'$.*

**Exemple 85.** *Soit l'ensemble de règles $\mathcal{R}_{73}$ et la requête $Q_{73}$ de l'exemple 83. Nous avons $\mathcal{Q}(\mathcal{R}_{73}) = \{Q_{73}, r_3, r_1, r_2\}$, et $D\mathcal{Q}(\mathcal{R}_{73}) = \{(r_7, +), (r_5, +), (r_4, +), (r_2, +), (r_1, -)\}$. Nous pouvons en plus considérer $D_2 = \{(r_8, +)\}$ un ensemble de règles dangereuses n'étant pas à l'intersection de $\mathcal{Q}(\mathcal{R}_{73})$. Le graphe de dépendance des règles est représenté en figure 4.3 avec les ensembles de règles $\mathcal{Q}(\mathcal{R}_{73}), D_1$ et $D_2$.*

FIGURE 4.3 – Graphe de dépendance des règles avec requête et règles dangereuses.

$r_1$ : ad(X) ← pU(X), not eC(X).
$r_2$ : eC(X) ← pU(X), not ad(X).
$r_3$ : cA(X) ← ad(X), d(X).
$r_4$ : mC(X) ← eC(X), not phdS(X).
$r_5$ : phdS(X) ← eC(X), et(X).
$r_6$ : chr(X) ← eC(X).
$r_7$ : c1 ← mC(X), phdS(X), not c1.
$r_8$ : c2 ← et(X), ch(X), not c2.
$Q$ : ans(X) ← cA(X).




*Les règles de l'ensemble $\mathcal{R}_{73}$ sont représentées par les $r_i$ en noir pour les règles dangereuses positives et en blanc pour les règles dangereuses négatives. On représente une règle $r_b$ dépendant d'une règle $r_a$ à l'aide d'une flèche allant de $r_a$ vers $r_b$. Les ensembles $\mathcal{Q}(\mathcal{R}_{73})$, $dRule(D\mathcal{Q}(\mathcal{R}_{73}))$, $\mathcal{Q}(\mathcal{R}_{73}) \cap dRule(D\mathcal{Q}(\mathcal{R}_{73}))$ et $D_2$ sont représentés respectivement par les formes grise, gris-clair, gris foncé et gris-clair. L'ensemble $\mathcal{Q}(\mathcal{R}_{73}) \cap dRule(D\mathcal{Q}(\mathcal{R}_{73}))$ correspond à l'intersection des formes correspondant à $\mathcal{Q}(\mathcal{R}_{73})$ et $dRule(D\mathcal{Q}(\mathcal{R}_{73}))$ (incluant $\mathcal{Q}(\mathcal{R}_{73}) \cap dRule(D\mathcal{Q}(\mathcal{R}_{73}))$).*

Nous isolons pour le moment deux ensembles de règles d'un programme $\mathbf{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ dont $Q$ dépend, $\mathcal{Q}(\mathcal{R})$ et $D\mathcal{Q}(\mathcal{R})$. Nous montrons maintenant qu'une réponse à $Q$ dans $\mathbf{P}$ est aussi une réponse dans $\mathbf{P}_{\mathcal{Q} \cup D\mathcal{Q}} = (\mathcal{F}, \mathcal{Q}(\mathcal{R}) \cup \mathtt{dRule}(D\mathcal{Q}(\mathcal{R})))$ et réciproquement une réponse à $Q$ dans $\mathbf{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ est aussi une réponse dans $\mathbf{P}$. Autrement dit, une réponse sur le programme original est aussi une réponse sur le programme composé seulement des règles dont la requête dépend et des règles dangereuses à l'intersection de celles-ci et réciproquement.

**Exemple 86.** *Soit le programme $\boldsymbol{P}_{86} = (\mathcal{F}_{86}, \mathcal{R}_{86})$ composé de :*

$$\mathcal{R}_{86} = \left\{ \begin{array}{l} r_1 : \mathtt{ad(X)} \leftarrow \mathtt{pU(X)},\ \mathtt{not\ eC(X)}. \\ r_2 : \mathtt{eC(X)} \leftarrow \mathtt{pU(X)},\ \mathtt{not\ ad(X)}. \\ r_3 : \mathtt{cA(X)} \leftarrow \mathtt{ad(X)},\ \mathtt{d(X)}. \\ r_4 : \mathtt{mC(X)} \leftarrow \mathtt{eC(X)},\ \mathtt{not\ phdS(X)}. \\ r_5 : \mathtt{phdS(X)} \leftarrow \mathtt{eC(X)}, \mathtt{et(X)}. \\ r_6 : \mathtt{these(X)} \leftarrow \mathtt{phdS(X)}. \\ r_7 : \mathtt{c1} \leftarrow \mathtt{mC(X)},\ \mathtt{phdS(X)},\ \mathtt{not\ c1}. \\ r_8 : \mathtt{c2} \leftarrow \mathtt{et(X)}, \mathtt{ch(X)},\ \mathtt{not\ c2}. \end{array} \right\}$$

*et $\mathcal{F}_{86} = \{\mathtt{pU(titi)}, \mathtt{mC(titi)}, \mathtt{et(titi)}\}$, une base de faits. Ce programme est composé de l'ensemble de règle $\mathcal{R}_{86}$ auquel nous avons ajouté la requête $\{Q_{86}\}$ : $(\mathtt{ans} \leftarrow \mathtt{these(X)}.)$ et une base de faits $\mathcal{F}_{86}$. Nous avons $\mathcal{Q}(\mathcal{R}_{86}) = \{Q_{86}, r_6, r_5, r_2, r_1\}$, $D\mathcal{Q}(\mathcal{R}_{86}) = \{(r_7, +), (r_5, +), (r_4, +), (r_2, +), (r_1, -)\}$ et $D_2 = \{(r_8, +)\}$ un ensemble de règles dangereuses n'étant pas à l'intersection de la requête. $\boldsymbol{P}_{86}$ est consistant car tous les ensembles de règles*

*dangereuses possèdent un answer set, $\mathcal{F}_{86}$ pour $(\mathcal{F}_{86}, \texttt{dRule}(D_2))$ et un unique answer set pour $(\mathcal{F}_{86}, \texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{86})))$ qui est :*

$$AS = \{\texttt{pU(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}, \texttt{ad(titi)}\}$$

*La réponse (sceptique et crédule) à $Q_{86}$ est* `faux` *pour le programme $(\mathcal{F}_{86}, \mathcal{Q}(\mathcal{R}_{86}) \cup \texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{86})))$. L'application de la règle $r_2$ : $(\texttt{eC(X)} \leftarrow \texttt{pU(X)}, \texttt{not ad(X)}.)$ avec $\texttt{pU(titi)}$ ne permettant par d'obtenir un answer set car cela déclenche la règle $r_5$ puis la contrainte $r_7$, le seul answer set vient de l'application de $r_1$ : $(\texttt{ad(X)} \leftarrow \texttt{pU(X)}, \texttt{not eC(X)}.)$ avec $\texttt{pU(titi)}$ qui bloque ainsi la règle $r_2$. Si nous calculons les answer set de $(\mathcal{F}_{86}, \mathcal{Q}(\mathcal{R}_{86}))$ nous avons*

$$AS'_1 = \{\texttt{pU(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}, \texttt{ad(titi)}\}$$
$$AS'_2 = \{\texttt{pU(titi)}, \texttt{mC(titi)}, \texttt{et(titi)}, \texttt{eC(titi)}, \texttt{phdS(titi)}, \texttt{these(titi)}, \texttt{ans}\}$$

*la réponse crédule à la requête $Q_{86}$ pour le programme $\mathcal{Q}(\mathcal{R}_{86})$ est par contre* `vrai` *car* `ans` *appartient à $AS'_2$. Nous avons donc deux réponses différentes pour les ensembles $\mathcal{Q}(\mathcal{R}_{86})$ et $\mathcal{Q}(\mathcal{R}_{86}) \cup D\mathcal{Q}(\mathcal{R}_{86})$ sachant que la réponse sur le programme correspond à la réponse de $\mathcal{Q}(\mathcal{R}_{86}) \cup \texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{86}))$. L'ensemble $\mathcal{Q}(\mathcal{R}_{86})$ possède un answer set de trop qui provient du fait que les règles contenant le cycle d'inconsistance ne font pas partie de cet ensemble. La contrainte $r_7$ n'est donc pas appliquée, le modèle $AS'_2$ ne mène donc pas à un answer set dans le programme complet et ne devrait pas être pris en compte pour calculer la réponse au programme. L'ensemble de règles $\mathcal{Q}(\mathcal{R}_{86})$ n'est donc pas suffisant pour trouver une réponse à $Q$ dans $\boldsymbol{P}_{86}$ il faut l'étendre à $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{86}))$ pour obtenir les mêmes réponses.*

**Théorème 4.5.4** (Réponse à un programme consistant)**.** *Soit $\boldsymbol{P}$ un programme ASP consistant. $R$ est une réponse à $Q$ dans $\boldsymbol{P}$ si et seulement si $R$ est une réponse dans $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$.*

**Preuve 4.5.4.** *Soit $\boldsymbol{P} = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ un programme consistant. Supposons qu'il existe une réponse dans $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ n'étant pas une réponse dans $\boldsymbol{P}$. D'après l'algorithme 4, $D\mathcal{Q}(\mathcal{R})$ contient l'ensemble des règles dangereuses à l'intersection de la requête ainsi que les règles dépendantes de celles-ci. Étant donné que $\boldsymbol{P}$ est consistant et que les autres ensembles de règles dangereuses n'ont pas de dépendance avec $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}))$ alors $(\mathcal{F}, \texttt{dRule}(D\mathcal{Q}(\mathcal{R})))$ est consistant. Étant donné que $D\mathcal{Q}(\mathcal{R})$ contient toutes les règles dangereuses à l'intersection de $\mathcal{Q}(\mathcal{R})$, il n'existe pas d'autres règles dangereuses dont l'ensemble $\mathcal{Q}(\mathcal{R})$ est dépendant. Comme $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}))$ est consistant, nous pouvons en déduire que $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ est consistant. Il existe donc au moins une réponse dans $\boldsymbol{P}$ et dans $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ non* `absurde`*. D'après le théorème 4.5.2, nous savons que si le programme est consistant quelque soit $\mathcal{F}$ alors une réponse dans $\boldsymbol{P}_{\mathcal{Q}}$ est une réponse dans $\boldsymbol{P}$. D'après l'algorithme 3, $\mathcal{Q}(\mathcal{R})$ contient toutes les règles dont la requête dépend et $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}))$ contient toutes les règles pouvant empêcher un ensemble d'atomes sur $\mathcal{Q}(\mathcal{R})$ d'être un answer set. Étant donné que $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ contient l'ensemble des règles dont $Q$ dépend ainsi que les règles dangereuses à l'intersection de $\mathcal{Q}(\mathcal{R})$, toute application de règles dont $Q$ dépend va déclencher les règles dangereuses à l'intersection. Donc s'il existe un answer set $AS$ de $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ alors nous pouvons étendre celui-ci avec les applications des règles de $\boldsymbol{P}$ n'apparaissant pas dans $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ car $\boldsymbol{P}$ est consistant et il n'existe pas de règle dans $\boldsymbol{P}$ pouvant modifier les atomes issus de l'application des règles de $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ (il n'existe pas de dépendance). Nous en déduisons que toute réponse dans $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ est aussi une réponse dans $\boldsymbol{P}$.*
*Supposons maintenant qu'une réponse dans $\boldsymbol{P}$ n'est pas une réponse dans $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$. Alors il existe un answer set $AS$ de $\boldsymbol{P}$ qui donne une instance de* `ans` *n'apparaissant pas dans $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$. D'après le théorème 4.5.1, nous savons que $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ possède toutes règles dont $Q$ dépend ($\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ contient $\boldsymbol{P}_{\mathcal{Q}}$) et que toutes les instances de* `ans` *possibles seront calculées. Nous déduisons qu'il existe un ensemble d'atomes équivalent à $AS$ sur $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$, qui est un answer set à l'ajout des instances provenant des règles appartenant seulement à $\boldsymbol{P}$ près. Étant donné que $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ possède toutes les règles dangereuses pouvant empêcher un ensemble d'atomes d'être un answer set les réponses obtenues sur $\boldsymbol{P}_{\mathcal{Q} \cup D\mathcal{Q}}$ sont équivalentes à celles obtenues sur $P$..*

Les règles nécessaires pour répondre à une requête sur un programme sont désormais isolées. Nous avons mis en avant que pour répondre à une requête sur un programme les règles dont la requête dépend ne sont pas suffisantes et qu'il faut ajouter à celles-ci les ensembles de règles dangereuses ayant une règle en commun. Il n'est par contre souvent pas nécessaire de calculer des answer set complets sur cet ensemble, nous présentons dans la prochaine partie plusieurs perspectives permettant de réduire le nombre d'instanciations nécessaire pour répondre à une requête. Dans un premier temps en s'intéressant aux instances nécessaires pour l'ensemble des règles dont la requête dépend à l'aide des constantes présentes dans celle-ci, et dans un deuxième temps en s'intéressant aux ensembles de règles dangereuses dont l'instanciation est nécessaire pour valider une réponse. Le problème est que l'instanciation d'un programme peut ne pas se terminer et même dans le cas où celle-ci se termine elle demande un temps de calcul assez conséquent. Nous souhaitons donc maintenant nous intéresser à l'instanciation des règles et plus particulièrement chercher à la minimiser lors d'une interrogation.

## 4.6 Perspectives d'optimisation

Pour améliorer l'efficacité de l'interrogation nous proposons des méthodes encore à l'étude afin de minimiser l'instanciation des règles nécessaires pour répondre à une requête. Nous avons dans un premier temps isolé les parties du programme suffisantes pour répondre à une requête. Maintenant, nous souhaitons instancier le moins d'atomes possible parmi les règles isolées. Pour cela il y a deux grands axes. Le premier axe se porte sur l'ensemble $\mathcal{Q}(\mathcal{R})$ : pour les requêtes contenant au moins une constante, nous allons prendre en compte cette information en répercutant celle-ci sur les règles du programme afin de réduire le nombre d'instanciations de certains atomes. Dans le deuxième axe nous souhaitons réduire les instanciations sur l'ensemble $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}))$. Cet ensemble ne peut pas utiliser les constantes présentes dans la requête pour instancier les règles qui le compose ce qui implique d'utiliser d'autres éléments comme la négation par défaut et les règles dangereuses négatives pour réduire le nombre d'instanciations.

### 4.6.1 Instanciation de la requête

Lors d'une interrogation, les arguments des atomes peuvent être des variables ou des constantes. Lorsqu'il y a une variable dans une requête, les réponses fournies seront des ensembles de substitutions possibles pour ces variables tandis que les constantes agiront comme une restriction en imposant une substitution à l'argument. Nous allons chercher dans un premier temps à utiliser les constantes présentes dans la requête pour limiter le nombre d'instanciations effectuées.

**Exemple 87.** *Soit le programme $P_{87} = (\mathcal{F}_{87}, \mathcal{R}_{87})$ composé de :*

$$\mathcal{R}_{87} = \left\{ \begin{array}{l} r1 : \texttt{p(X,Y)} \leftarrow \texttt{a(X,Y)}, \texttt{ not q(X,Y)}. \\ r2 : \texttt{q(X,Y)} \leftarrow \texttt{b(X,Y)}, \texttt{ not p(X,Y)}. \\ r3 : \texttt{s(X,Y)} \leftarrow \texttt{p(X,Y)}. \\ r4 : \texttt{t(X,Y)} \leftarrow \texttt{q(X,Y)}, \texttt{ not t(X,Y)}. \end{array} \right\}$$

*et de la base de faits $\mathcal{F}_{87} = \{\texttt{a(1,2)}, \texttt{a(3,3)}, \texttt{a(2,1)}, \texttt{b(2,1)}\}$ et la requête $Q_{87} = (\texttt{ans} \leftarrow \texttt{ s(1,2)}.)$. Nous avons $\mathcal{Q}(\mathcal{R}_{87}) = \{r_1, r_2, r_3\}$, $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{87})) = \{r_1, r_2, r_4\}$. Nous réécrivons les règles de $\mathcal{Q}(\mathcal{R}_{87})$ en instanciant partiellement celles-ci en utilisant les constantes de la requête. ce qui donne :*

$$\left\{ \begin{array}{l} r1' : \texttt{p(1,2)} \leftarrow \texttt{a(1,2)}, \texttt{ not q(1,2)}. \\ r2' : \texttt{q(1,2)} \leftarrow \texttt{b(1,2)}, \texttt{ not p(1,2)}. \\ r3' : \texttt{s(1,2)} \leftarrow \texttt{p(1,2)}. \end{array} \right\}$$

*Nous appliquons donc simplement $r'_1$ puis $r'_3$ pour avoir la réponse à notre requête puis nous étendons à $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{87}))$ pour vérifier que la réponse n'est pas absurde. On remarque que le nombre d'instanciations nécessaires est diminué car les variables dans les règles $r_1$, $r_2$ et $r_3$ ont été remplacées par des constantes, il n'y a donc aucune autre instanciation à faire que celles déjà effectuées. Le problème est de s'assurer que toutes les instanciations nécessaires sont bien effectuées avec la marche arrière pour pouvoir obtenir les bonnes réponses avec la marche avant.*

### 4.6.2 Instanciation des règles dangereuses négatives

Un inconvénient majeur avec les règles dangereuses reste le fait de devoir instancier l'ensemble des variables contrairement à l'ensemble des règles de la requête où l'on peut réduire le nombre d'instanciation grâce aux constantes présentes dans la requête. Nous nous sommes intéressés aux dépendances négatives dans le graphe des symboles de prédicat afin de réduire les instanciations des règles dangereuses au minimum. Nous remarquons alors que certaines instanciations ne sont pas nécessaires pour prouver l'existence d'un réponse non absurde tant que la consistance du programme est vérifiée.

**Exemple 88.** *Nous reprenons le programme $P_{87} = (\mathcal{F}_{87}, \mathcal{R}_{87})$ de l'exemple précédent avec la requête $Q_{87} = (\texttt{ans} \leftarrow \texttt{s}(1,2).)$. Nous avons $\mathcal{Q}(\mathcal{R}_{87}) = \{Q_{87}, r_1, r_2, r_3\}$, $D\mathcal{Q}(\mathcal{R}_{87}) = \{(Q_{87}, +), (r_1, -), (r_2, +), (r_4, +)\}$. En réécrivant les règles de $\mathcal{Q}(\mathcal{R}_{87})$ nous obtenons la réponse vrai à notre requête sur $\mathcal{Q}(\mathcal{R}_{87})$ en ayant déduit $\texttt{p}(1,2)$ et $\texttt{s}(1,2)$. Nous souhaitons vérifier que la réponse n'est pas absurde en étendant celle-ci à $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{87}))$. Notre base de faits est donc $\{\texttt{a}(1,2), \texttt{a}(3,3), \texttt{a}(2,1), \texttt{b}(2,1), \texttt{p}(1,2), \texttt{s}(1,2)\}$. En essayant d'appliquer les règles de $\texttt{dRule}(D\mathcal{Q}(\mathcal{R}_{87}))$ nous remarquons que les instances des règles dangereuses négatives (ici $r_1$) ne peuvent pas rendre absurde la réponse trouvée. En effet, en appliquant $r_1$ avec $\texttt{a}(3,3)$ il est impossible d'obtenir une instance de $\texttt{t}(\texttt{X},\texttt{Y})$ pouvant par la suite rendre la réponse absurde. Par contre si nous déclenchons la règle $r_2$ avec $\texttt{b}(2,1)$ nous créons une instance de $\texttt{t}(\texttt{X},\texttt{Y})$ si la règle $r_1$ n'a pas déjà été déclenchée avec $\texttt{a}(2,1)$ qui bloque l'application de $r_2$. L'application de $r_1$ ne peut donc pas rendre la réponse absurde mais peut rendre une réponse non absurde si elle bloque l'application d'une règle dangereuse positive. Dans notre cas nous économisons une instanciation de règle avec $\texttt{a}(3,3)$ car il n'existe pas de règle qui sera bloquée par celle-ci et les instances de $\texttt{a}(\texttt{X},\texttt{Y})$ ne peuvent pas rendre une réponse absurde.*

## 4.7 Conclusion

L'interrogation permet d'extraire des informations d'une ontologie répondant à certaines conditions, nous avons montré que nous étions capable d'isoler une partie suffisante d'un programme à partir d'une requête pour répondre à celle-ci afin de réduire le temps de calcul nécessaire pour y répondre. Un des aspects étudiés est l'inconsistance d'un programme ASP, nous avons mis en exergue qu'il était nécessaire d'intégrer les règles dangereuses (responsable de l'inconsistance) aux règles dont la requête dépend afin de donner une réponse correcte à une interrogation. Nous mettons aussi en avant que dans les ontologies, les données sont souvent isolées ayant peu de dépendances entre elles ce qui est un avantage pour l'interrogation étant donné qu'il n'est alors souvent pas nécessaire de traiter tout le programme pour obtenir une réponse à notre requête ou bien prouver la consistance de celui-ci.
Dans la dernière partie nous proposons quelques possibles améliorations pour réduire le nombre d'instanciations nécessaire pour l'obtention d'une réponse à une requête. Deux optimisations sont proposées : la première consiste à utiliser les constantes présentes dans la requête pour réécrire les règles du programme initial avec celles-ci. Ainsi nous diminuons le nombre d'instanciations en n'utilisant que celles nécessaire à la requête. La seconde proposition consiste à utiliser les règles dangereuses négatives permettant de bloquer d'autres règles dangereuses afin de réduire l'instanciation de

ces règles dangereuses. Ces deux parties n'en sont qu'à leur prémices mais ouvrent des perspectives d'optimisation de l'interrogation en ASP dans le cadre de l'implémentation que nous allons d'ailleurs aborder dans la prochaine partie.

# 5

# Implémentation

Nous avons proposé dans les parties précédentes une extension commune aux règles existentielles et à ASP nommée programme NME, nous souhaitons maintenant présenter un solveur permettant de traiter des programmes issus de ce langage et l'interrogation de ces programmes. Pour cela, il existe déjà des solveurs ASP efficaces, nous avons par ailleurs vu qu'il était possible de traduire un programme NME en programme ASP classique simplement. Nous avons donc choisi d'étendre un solveur ASP existant avec un module de traduction ainsi qu'un module d'interrogation. Pour terminer nous avons effectué une série de tests à partir du benchmark `university` servant à tester l'efficacité de l'extension du solveur proposé et comparer les résultats après traduction en ASP classique avec les autres solveurs.

## 5.1 ∃-ASPeRiX

Un solveur ASP est un programme permettant de calculer les answer set d'un programme ASP. ∃-ASPeRiX est une extension du solveur ASPeRiX permettant de traiter la syntaxe et la sémantique NME mise en avant dans cette thèse. Cette extension est décomposée en deux modules intégrés au solveur, un premier module permettant la traduction d'un programme NME vers un programme ASP avec quelques optimisations comparé à la transformation présentée dans la partie 2.2, et un second permettant l'interrogation d'un programme. Cette extension utilise après traduction l'algorithme du solveur ASPeRiX pour calculer les answer set, nous allons donc dans un premier temps présenter ASPeRiX et ses spécificités. Dans un deuxième temps nous présenterons la syntaxe puis la grammaire acceptées par le solveur, nous nous intéresserons ensuite à l'implémentation de la traduction et pour finir, celle de l'interrogation.

### 5.1.1 ASPeRiX

Afin de calculer les answer set, la majorité des solveurs nécessite au préalable d'avoir un programme logique propositionnel en entrée, pour cela le programme logique du premier ordre passe d'abord par une phase de grounding effectuée par un outil appelé grounder. Une fois instancié le programme peut-être très grand, voire infini. La figure 5.1 illustre le processus du calcul d'answer set pour ces solveurs.

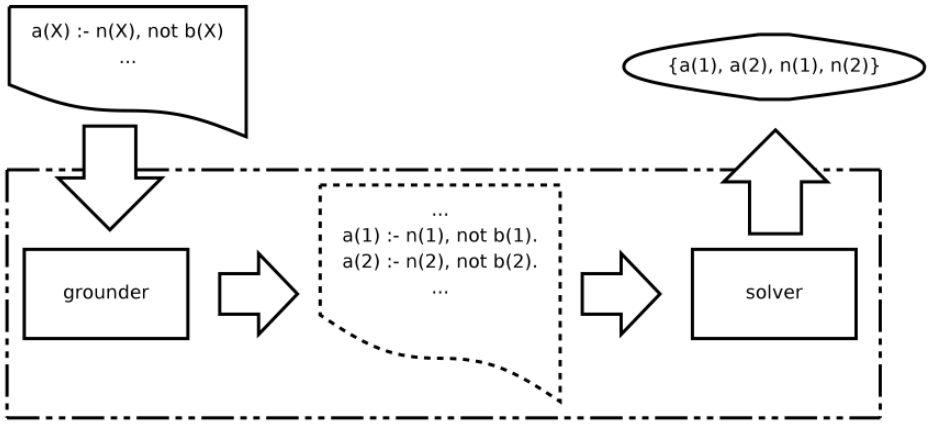



FIGURE 5.1 – Architecture du calcul d'answer set

Les grounders cherchent à instancier le minimum d'atomes servant au calcul des answer set d'un programme. Les grounders utilisent ainsi des techniques pour optimiser cette phase d'instanciation et limiter au mieux le nombre d'instanciations nécessaires, le programme logique du premier ordre passant obligatoirement par cette phase avant de passer à la phase de calcul des answer set. Le programme permettant de calculer les answer set d'un programme logique du premier ordre est souvent divisé en deux outils distincts comme lparse [68] et gringo [41, 32] pour la partie grounder, et respectivement smodels [67] et clasp [42] pour la partie calcul d'answer set. DLV [55] propose un outil tout-en-un rassemblant le grounder et le solveur mais en conservant deux phases distinctes. D'autres solveurs proposent après grounding une transformation vers un programme SAT comme Assat et Cmodels.

Le problème de cette méthode est que le travail d'instanciation réalisé par le grounder peut calculer un grand nombre d'instanciations inutiles pour un programme comme dans l'exemple suivant :

**Exemple 89.** *Soit le programme ASP :*

$$P = \left\{ \begin{array}{l} \mathtt{a} \leftarrow \mathtt{not\ b}. \\ \mathtt{b} \leftarrow \mathtt{not\ a}. \\ \leftarrow \mathtt{a}. \\ \mathtt{p(0)}. \\ \mathtt{p(X+1)} \leftarrow \mathtt{a, p(X)}. \end{array} \right\}$$

*L'instanciation de $P$ est infinie (tant qu'il n'existe pas une borne maximale définie pour les entiers) alors que le programme possède un unique answer set fini $AS = \{\mathtt{b}, \mathtt{p(0)}\}$ à cause de la contrainte $(\leftarrow \mathtt{a}.)$ qui empêche l'application de la règle $(\mathtt{p(X+1)} \leftarrow \mathtt{a, p(X)}.)$.*

ASPeRiX, choisi pour l'extension aux programmes NME, se démarque en proposant de calculer les answer set directement à partir du programme logique du premier ordre [52, 53] sans passer par une phase d'instanciation complète, mais en instanciant les règles à la volée seulement lorsqu'elles sont applicables avec les atomes déjà déduits [57]. Étant donné qu'ASPeRiX réalise le grounding pendant la phase de calcul d'answer set, nous parlons maintenant de solveur pour désigner l'ensemble grounder avec le solveur. Ce solveur, développé au LERIA à l'université d'Angers, propose une approche basée sur les règles. Ce qui signifie que les choix effectués dans l'arbre de recherche se font sur l'application ou non d'une règle et non sur les valeurs de vérité des atomes comme pour la plupart des solveurs. ASPeRiX a été choisi pour son instanciation à la volée qui peut être efficace pour les programmes utilisant des symboles de fonctions (dans notre cas issus principalement de la skolémisation). Une instanciation à la volée permet, par exemple, de ne pas systématiquement calculer toutes les instances associées aux termes fonctionnels, ce qui peut dans certain cas éviter une

instanciation infinie d'une règle qui ne sera finalement jamais appliquée. D'autres solveurs proposent une instanciation à la volée comme GASP ou plus récemment Omiga mais ASPeRiX a l'avantage d'être maintenu, d'avoir des sources accessibles avec les principaux développeurs à l'université d'Angers.
Afin de pouvoir calculer les answer set d'un programme NME, la syntaxe et la grammaire acceptées par ASPeRiX ont été modifiées pour dans un premier temps traduire le programme en ASP classique puis effectuer le calcul. Nous présentons d'abord la syntaxe acceptée par l'extension d'ASPeRiX que nous appellerons ∃−ASPeRiX.

### 5.1.2 Syntaxe

La syntaxe acceptée par ∃−ASPeRiX est basée sur la syntaxe ASP−Core [26] qui est une normalisation proposée pour les solveurs ASP. Nous utilisons le vocabulaire proposé dans la partie 1.1.1.

**Termes.** Un terme peut être une constante, une variable, un terme fonctionnel ou une *opération arithmétique*.

- Une constante peut être une *constante symbolique* (une chaîne de caractère commençant par une lettre minuscule), une *constante numérique* (un entier) ou une *chaîne de caractère constante* (chaîne de caractère entre guillemets).
- Une variable est une chaîne de caractères commençant par une lettre majuscule.
- Un *terme fonctionnel* est de la forme $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ avec f un symbole fonctionnel d'arité $n$ et $\mathtt{t}_i$ des termes. Un terme fonctionnel $\mathtt{f}()$ d'arité 0 est équivalent à une constante symbolique f ;
- Une *opération arithmétique* est de la forme $-\mathtt{t}_1$ ou $\mathtt{t}_1 \circ \mathtt{t}_2$ avec $\mathtt{t}_1$ et $\mathtt{t}_2$ deux termes, et $\circ \in \{+, -, *, /\}$.

**Atomes.** Un atome peut être un *atome classique*, un *atome relationnel* ou un *atome intervalle*.

- Un atome classique est de la forme $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ avec $\mathtt{t}_1, \ldots, \mathtt{t}_n$ des termes et p un symbole de prédicat (une chaîne de caractère commençant par une lettre minuscule ou une chaîne de caractères entre guillemets). Un atome classique $\mathtt{p}()$ d'arité 0 est représenté par son nom de prédicat p sans les parenthèses. Étant donné un atome classique p, un littéral peut prendre la valeur de p ou $\neg$p (qui est noté $-$p). Les atomes classiques peuvent être liés à une négation par défaut *not* pour former le corps négatif d'une règle (à ne pas confondre avec la négation forte $\neg$).
- Un atome relationnel est de la forme $\mathtt{t}_1 \diamond \mathtt{t}_2$ avec $\diamond \in \{<, <=, =, != , >, >=\}$ et $t_1, t_2$ deux termes.
- Un atome intervalle noté $p(t_1..t_2)$ avec $p$ un symbole de prédicat et $t_1, t_2$ des termes pouvant être des constantes numériques ou des variables. Cet atome intervalle définit un ensemble d'atomes qui vont de $p(t_1)$ à $p(t_2)$.

**Règle.** Une règle est de la forme $\mathtt{H}$ : - $\mathtt{B}^+$, not $\mathtt{N}_1, \ldots,$ not $\mathtt{N}_k$. avec H un ensemble d'atomes intervalles et d'atomes classiques (nous rappelons que pour un programme NME l'ensemble d'atomes en tête est une conjonction d'atomes et non pas une disjonction), $\mathtt{B}^+$ un ensemble d'atomes classiques et d'atomes relationnels, $\mathtt{N}_1, \ldots, \mathtt{N}_k$ des ensembles d'atomes classiques liés par une négation par défaut. Une particularité des programmes NME est la possibilité d'avoir des règles non-safe, ce qui signifie

qu'une règle peut avoir des variables apparaissant seulement dans la tête d'une règle ou seulement dans le corps négatif d'une règle sans devoir apparaître dans le corps positif.

**Règles spéciales.** Un fait est une règle seulement composée d'une tête de la forme `H.` avec `H` un ensemble d'atomes intervalles et d'atomes classiques. Une contrainte est une règle seulement composée d'un corps et la conclusion de cette règle est $\bot$ (absurde) notée (`:- B`$^+$`, not N`$_1$`, ..., not N`$_k$`.`). Une règle positive est une règle sans corps négatif et est équivalente à une règle existentielle.

Une restriction syntaxique pour les programmes est qu'une variable existentielle ne peut pas apparaître à la fois dans la tête et le corps négatif d'une règle comme précisé dans la partie 2.1.

### 5.1.3 Grammaire

Nous présentons maintenant la grammaire acceptée par l'analyseur syntaxique d'∃−ASPeRiX.

#### Éléments de syntaxe

| | | |
|---|---|---|
| $< number >$ | $::=$ | $0..9 \mid 0..9 < number >$ |
| $DOT$ | $::=$ | $'.'$ |
| $PAREN\_OPEN$ | $::=$ | $'('$ |
| $PAREN\_CLOSE$ | $::=$ | $')'$ |
| $MINUS$ | $::=$ | $'-'$ |
| $NOT$ | $::=$ | "$not$" |

#### Tokens

| | | |
|---|---|---|
| $< symbolic\ constant >$ | $::=$ | une chaîne composée de lettres, chiffres et de caractères de soulignement commençant par une lettre minuscule |
| $< variable >$ | $::=$ | une chaîne de lettres et chiffres commençant par une lettre majuscule |
| $< string >$ | $::=$ | une séquence de caractères entre guillemets |
| $< arithmetic\ constant >$ | $::=$ | $[MINUS] < number >$ |
| $< range >$ | $::=$ | $< arithmetic\ constant > .. < arithmetic\ constant >$ |
| $< arithmetic\ expression >$ | $::=$ | expression arithmétique construite à partir de constantes, variables d'opérateurs binaires $+, -, *, /$ et mod, d'opérateur unaires abs, et de parenthèses. |
| $< term >$ | $::=$ | $< variable > \mid < arithmetic\ expression > \mid < string >$ |
| $< terms >$ | $::=$ | $[< terms >,] < term >$ |
| $< extended\ terms >$ | $::=$ | $[< extended\ terms >,](< term > \mid < predicate\ atom >)$ |
| $< relational\ atom >$ | $::=$ | $< term > (< \mid > \mid = \mid == \mid != \mid >= \mid <=) < term >$ |
| $< equality >$ | $::=$ | $< term > (= \mid ==) < term >$ |
| $< predicate\ atom >$ | $::=$ | $< symbolic\ constant > [PAREN\_OPEN < extended\ terms > PAREN\_CLOSE]$ |
| $< literal >$ | $::=$ | $([MINUS] < predicate\ atom >) \mid < relational\ atom >$ |
| $< literals >$ | $::=$ | $[< literals >,] < literal >$ |
| $< range\ literal >$ | $::=$ | $[MINUS] < symbolic\ constant > [PAREN\_OPEN(< extended\ terms > \mid < range >)PAREN\_CLOSE]$ |
| $< range\ literals >$ | $::=$ | $[< range\ literals >,](< range\ literal > \mid < equality >)$ |
| $< positive\ body >$ | $::=$ | $< literals >$ |
| $< negative\ bodies >$ | $::=$ | $[< negative\ bodies >,]NOT(< literal > \mid (PAREN\_OPEN < literals > PAREN\_CLOSE))$ |
| $< body >$ | $::=$ | $[< body >,](< positive\ body > \mid < negative\ bodies > \mid < relational >)$ |
| $< head >$ | $::=$ | $< range\ literals >$ |

**Program**

$$
\begin{array}{lcl}
< fact > & ::= & < head > DOT \\
< normal\ rule > & ::= & < head > :- < body > DOT \\
< positive\ rule > & ::= & < head > :- < positive\ body > DOT \\
< constraint > & ::= & :- < positive\ body > DOT \\
 & | & :- < positive\ body >, < negative\ bodies > DOT \\
 & | & :- < negative\ bodies > DOT \\
< rule > & ::= & < fact > \\
 & | & < normal\ rule > \\
 & | & < positive\ rule > \\
 & | & < constraint > \\
< query > & ::= & ans[PAREN\_OPEN < variables > PAREN\_CLOSE] :- < body > DOT \\
 & | & ? < symbolic\ constant > [PAREN\_OPEN < variables > PAREN\_CLOSE] \\
 & & :- < body > DOT \\
< special\ command > & ::= & \#hide < predicate > / < number > DOT \\
 & | & \#show < predicate > / < number > DOT \\
< program > & ::= & [program\ \backslash n](< rule > | < special\ command > | < query >)
\end{array}
$$

### 5.1.4 Module de traduction NME vers ASP

Le solveur ∃−ASPeRiX est donc une extension d'ASPeRiX traitant le langage de règles non-monotones existentielles proposé dans cette thèse. Comme expliqué dans la partie 2.2 il est possible de traduire un programme NME en un programme ASP classique, c'est le rôle du module de traduction qui a été intégré à ASPeRiX. La majeure partie de la traduction s'exécute lors de l'analyse syntaxique du programme NME proposé au solveur. Le module reprend les phases proposées pour la traduction d'un programme NME en programme ASP de la partie 2.2, soit, la skolémisation effectuée lorsque l'analyseur détecte une variable existentielle, l'expansion gérée directement dans la structure de données d'ASPeRiX sous forme d'ensemble d'atomes, et enfin la normalisation avec la création de nouvelles règles normalisées lors de l'analyse syntaxique. Le système suit la procédure illustrée en figure 5.2.
Nous allons détailler ces trois phases et la façon dont elles sont traitées dans ∃−ASPeRiX.

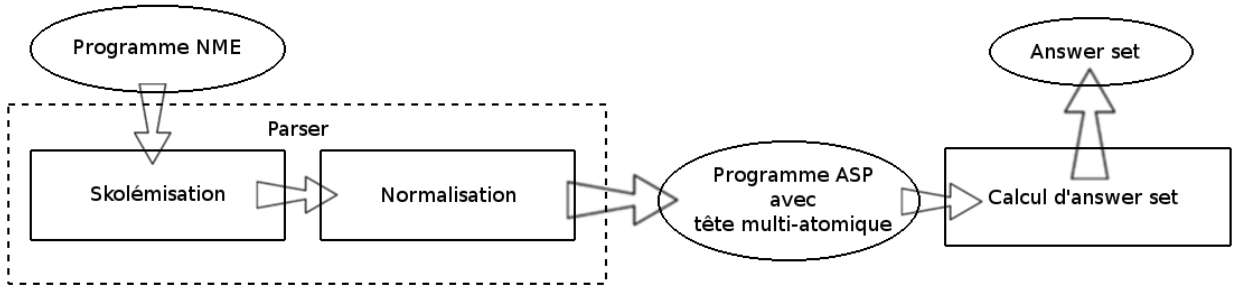



FIGURE 5.2 – Architecture du calcul d'answer set pour un programme NME

**Gestion des variables existentielles** Comme pour la traduction proposée précédemment, ∃−ASPeRiX utilise une skolémisation. Les symboles de fonction étant autorisés dans le solveur ASPeRiX il n'y a pas de soucis pour transformer les règles du programme original, contenant des variables existentielles en tête, en un programme contenant des symboles de fonction. Pour cela nous utilisons la même skolémisation que celle présentée dans la partie 2.1.1. Ainsi le traitement de la skolémisation s'exécute de la manière suivante :

L'algorithme remplace toutes les variables existentielles par un nouveau symbole de fonction de skolem $\texttt{skRiX}(\texttt{Fr})$ unique avec $i$ l'identifiant de la règle (les règles étant conservées à l'aide d'un identifiant entier), $X$ le nom de la variable skolémisée et $Fr$ l'ensemble des variables de la frontière de la règle.

**Exemple 90.** *La règle $R1$ suivante :*

$$\texttt{dT(X,D)}, \texttt{d(D)} \texttt{ :- } \texttt{p(X)}, \texttt{not(eC(X)}, \texttt{dC(X,Y))}.$$

*est skolémisée en :*

$$\texttt{dT(X, skR1D(X))}, \texttt{d(skR1D(X))} \texttt{ :- } \texttt{p(X)}, \texttt{not(eC(X)}, \texttt{dC(X,Y))}.$$

*Par la suite* $\texttt{skR1D(X)}$ *est géré comme terme fonctionnel classique par* `ASPeRiX`.

**Gestion des têtes multi-atomiques** Une différence par rapport à ASP classique est la gestion d'ensembles d'atomes en tête de règle. Alors que l'ASP disjonctif considère un ensemble d'atomes en tête comme une disjonction, dans un programme NME nous proposons une conjonction d'atomes en tête de règle. Une façon de traiter les ensembles d'atomes en tête est d'utiliser l'expansion comme définie dans la partie 2.2.3, seulement cela augmenterait le nombre de règles à traiter. Pour cela, nous avons décidé de stocker directement l'ensemble d'atomes en tête dans la structure de données d'`ASPeRiX`. Cette action a pour but d'optimiser le temps de calcul, ainsi là où l'algorithme du solveur devrait tester le support de $n$ règles dont le corps est identique, $\exists$−`ASPeRiX` teste une seule règle et ajoute l'ensemble d'atomes en tête dans la structure de données.

**Gestion du corps négatif multi-atomique avec variables existentielles** Pour l'utilisation de corps négatifs multi-atomiques avec variables existentielles, nous réécrivons directement la règle à partir de l'analyse syntaxique. Nous appliquons la réécriture proposée dans la partie 2.2.1. Pour chaque ensemble d'atomes attaché à un $not$ nous ajoutons une nouvelle règle au programme avec en corps l'ensemble d'atomes qui était attaché au $not$ et en tête un atome unique $RiNj$ qui remplacera l'ensemble d'atomes attaché au $not$. Cet atome $RiNj$ a pour argument l'ensemble des variables de la frontière avec $i$ l'identifiant de la règle et $j$ la position de l'ensemble d'atomes au sein du corps négatif de la règle. Ainsi, s'il existe plusieurs ensembles d'atomes dans les corps négatifs, ils posséderont tous un atome de remplacement unique. Pour que les nouvelles règles ajoutées soient transparentes aux yeux de l'utilisateur, les nouveaux atomes sont cachés (donc n'apparaissent pas dans les answer set).

**Exemple 91.** *La règle suivante :*

$$R1 : \texttt{dT(X, skR1D(X))}, \texttt{d(skR1D(X))} \texttt{ :- } \texttt{p(X)}, \texttt{not(eC(X)}, \texttt{dC(X,Y))}.$$

*est normalisée en :*

$$\begin{array}{ll} R1 : & \texttt{dT(X, skR1D(X))}, \texttt{d(skR1D(X))} \texttt{ :- } \texttt{p(X)}, \texttt{not R1N1(X)}. \\ R1' : & \texttt{R1N1(X)} \texttt{ :- } \texttt{eC(X)}, \texttt{dC(X,Y)}. \end{array}$$

*Le symbole de prédicat* `R1N1` *reste caché et n'est utilisé que pour l'application de la règle $R_1$ sans apparaître dans les answer set.*

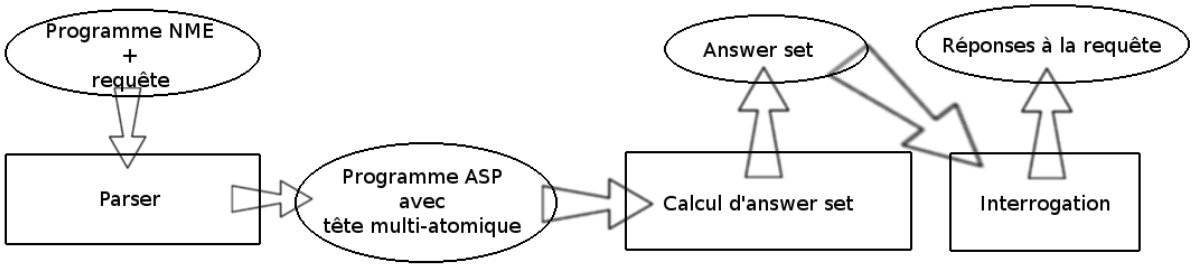



FIGURE 5.3 – Architecture de l'interrogation d'un programme NME

### 5.1.5 Module d'interrogation

Un premier module d'interrogation a été implémenté dans ASPeRiX, celui-ci permet l'utilisation d'une requête le plus simplement possible. Ainsi nous calculons l'ensemble des answer set (ou une partie si l'on peut déjà avoir une réponse) puis nous vérifions si la réponse à notre requête appartient à au moins un answer set pour une réponse crédule et à tous les answer set pour la réponse sceptique comme illustré dans la figure 5.3.
Il est donc possible d'interroger un programme avec une requête conjonctive classique de la forme :

$$\texttt{ans(X)} \texttt{ :- } \texttt{cA(X)}, \texttt{dT(X,Y)}, \texttt{phdS(Y)}.$$

ou bien avec une requête conjonctive booléenne de la forme :

$$\texttt{ans} \texttt{ :- } \texttt{cA(X)}, \texttt{dT(X,Y)}, \texttt{phdS(Y)}.$$

L'analyseur syntaxique stocke dans un premier temps l'atome réponse puis teste son appartenance aux answer set. Nous avons alors deux réponses possibles, la réponse crédule et la réponse sceptique pour lesquelles nous pourrons utiliser deux types de requête, les requêtes conjonctives et les requêtes conjonctives booléennes dont les réponses seront respectivement un ensemble d'ensembles de substitutions et un booléen ($true$ ou $false$). Dans les exemples présentés au dessus, la première requête interroge le programme pour connaître tous les individus qui font partie du conseil d'administration dirigeant au moins une thèse, la seconde demande simplement s'il existe un individu faisant partie du conseil d'administration et qui dirige une thèse.

**Réponse à une requête conjonctive** Pour la réponse crédule d'une requête conjonctive, nous parcourons tous les answer set en ajoutant toutes les instances possibles de l'atome réponse à un ensemble de réponse. Une fois l'ensemble des answer set parcouru la réponse à la requête est l'ensemble des substitutions possibles pour l'atome réponse (il suffit pour cela de rattacher directement les constantes en argument de l'ensemble des réponses, aux variables en argument de la requête).

**Exemple 92.** *Soit le programme ASP* $\boldsymbol{P}_{73}$ *suivant :*

$$
\begin{array}{l}
\texttt{ad(X)} \texttt{ :- pU(X), not eC(X).} \\
\texttt{eC(X)} \texttt{ :- pU(X), not ad(X).} \\
\texttt{cA(X)} \texttt{ :- ad(X), d(X).} \\
\texttt{mC(X)} \texttt{ :- eC(X), not phdS(X).} \\
\texttt{phdS(X)} \texttt{ :- eC(X), et(X).} \\
\texttt{chr(X)} \texttt{ :- eC(X).} \\
\texttt{c1} \texttt{ :- mC(X), phdS(X), not c1.} \\
\texttt{c2} \texttt{ :- et(X), ch(X), not c2.} \\
\texttt{ad(toto).} \\
\texttt{d(toto).} \\
\texttt{pU(titi).} \\
\texttt{d(titi).} \\
\texttt{et(titi).} \\
\texttt{ans(X)} \texttt{ :- cA(X).}
\end{array}
$$

*les answer set correspondants sont :*

$$
\begin{array}{rl}
AS_1^Q = & \{\texttt{ad(toto)}, \texttt{d(toto)}, \texttt{pU(titi)}, \texttt{d(titi)}, \texttt{et(titi)}, \\
& \texttt{cA(toto)}, \texttt{ad(titi)}, \texttt{cA(titi)}, \texttt{ans(toto)}, \texttt{ans(titi)}\} \\
AS_2^Q = & \{\texttt{ad(toto)}, \texttt{d(toto)}, \texttt{pU(titi)}, \texttt{d(titi)}, \texttt{et(titi)}, \\
& \texttt{chr(titi)}, \texttt{cA(toto)}, \texttt{eC(titi)}, \texttt{phdS(titi)}, \texttt{ans(toto)}\}
\end{array}
$$

*l'algorithme calcule donc l'union de toutes les instances de l'atome réponse dans un ensemble.*

$$\{\texttt{ans(toto)}, \texttt{ans(titi)}\}$$

*la réponse retournée par le solveur est donc l'ensemble des substitutions possibles*

$$\{[\texttt{X} < -\texttt{toto}], [\texttt{X} < -\texttt{titi}]\}$$

Pour la réponse sceptique d'une requête conjonctive, nous parcourons un premier answer set que nous stockons, puis nous parcourons les autres answer set en calculant à chaque fois l'intersection de l'answer set stocké avec l'answer set suivant. Ce qui nous permet de supprimer toutes les instances n'appartenant pas à tous les answer set.

Dans le cas de la réponse sceptique, il est possible d'arrêter l'interrogation plus tôt dans le cas où l'intersection des answer set est vide ($\emptyset$). Au cours du calcul de la réponse dès lors que l'intersection des answer set est vide nous pouvons arrêter la recherche de réponse car l'intersection sera maintenant toujours vide, l'algorithme s'arrête alors avant la fin du parcours et retourne la réponse $\{\ \}$. Cela ne fonctionne que dans le cas d'une réponse sceptique à une requête conjonctive non booléenne, il n'est pas possible de stopper l'interrogation de la même manière pour la réponse crédule à une requête conjonctive non booléenne.

**Exemple 93.** *Soit le programme ASP* $\boldsymbol{P}_{73}$ *précédent. L'algorithme récupère donc l'intersection de toutes les instances de l'atome réponse dans un ensemble. Supposons que le solveur calcule* $AS_1^Q$ *en premier alors le solveur construit l'ensemble des réponses :*

$$\{\texttt{ans(toto)}, \texttt{ans(titi)}\}$$

*puis le solveur calcule le second et dernier answer set* $AS_2^Q$ *dont l'ensemble des réponses est*

$$\{\texttt{ans(toto)}\}$$

*et procède à l'intersection des deux ensemble de réponses. Nous obtenons alors :*

$$\{\texttt{ans(toto)}\}$$

*la réponse retournée par le solveur est donc l'ensemble des substitutions possibles*

$$\{[\texttt{X} < -\texttt{toto}]\}$$

**Réponse à une requête conjonctive booléenne** Pour la réponse crédule d'une requête booléenne conjonctive, nous parcourons les answer set jusqu'à trouver une première occurrence de l'atome réponse, dès que la première occurrence est trouvée alors l'algorithme s'arrête et renvoie $true$ sans avoir besoin de parcourir les autres answer set. Si aucune occurrence de l'atome réponse est trouvée alors la réponse est $false$.

**Exemple 94.** *Soit le programme ASP* $\boldsymbol{P}_{75}$ *suivant :*

```
ad(X) :- pU(X), not eC(X).
eC(X) :- pU(X), not ad(X).
cA(X) :- ad(X), d(X).
mC(X) :- eC(X), not phdS(X).
phdS(X) :- eC(X), et(X).
chr(X) :- eC(X).
c1 :- mC(X), phdS(X), not c1.
c2 :- et(X), ch(X), not c2.
ad(toto).
d(toto).
pU(titi).
d(titi).
et(titi).
ans :- cA(titi).
```

*les answer set correspondants sont :*

$$\begin{aligned} AS_1^Q = \; & \{\texttt{ad(toto)}, \texttt{d(toto)}, \texttt{pU(titi)}, \texttt{d(titi)}, \texttt{et(titi)}, \\ & \texttt{cA(toto)}, \texttt{ad(titi)}, \texttt{cA(titi)}, \texttt{ans}\} \\ AS_2^Q = \; & \{\texttt{ad(toto)}, \texttt{d(toto)}, \texttt{pU(titi)}, \texttt{d(titi)}, \texttt{et(titi)}, \\ & \texttt{chr(titi)cA(toto)}, \texttt{eC(titi)}, \texttt{phdS(titi)}\} \end{aligned}$$

*l'algorithme vérifie s'il existe un answer set avec l'atome réponse, il détecte* $AS_1^Q$ *et n'a pas besoin de parcourir le second pour répondre* $true$*.*

Pour la réponse sceptique d'une requête booléenne conjonctive, nous parcourons les answer set jusqu'à trouver un answer set ne possédant pas l'atome réponse, dans ce cas la réponse est $false$ et l'algorithme s'arrête sans parcourir le reste des answer set. S'il existe une occurrence de l'atome réponse dans tous les answer set alors la réponse est $true$ après les avoir tous parcourus.

**Exemple 95.** *Soit le programme ASP* $\boldsymbol{P}_{75}$ *précédent. L'algorithme parcourt chaque answer set et vérifie si l'atome réponse est présent dans chacun d'entre eux. S'il y a un seul answer set qui ne possède pas l'atome réponse alors l'algorithme s'arrête sinon il continue avec tous les answer set. Ici supposons que l'algorithme parcourt en premier* $AS_2^Q$*, l'atome réponse n'est pas présent donc il n'y a pas besoin de parcourir* $AS_1^Q$*, le solveur renvoie directement* $false$*.*

## 5.2 Comparaison des solveurs ASP

Pour tester l'efficacité du solveur ∃−ASPeRiX, nous avons décidé dans un premier temps de le comparer aux solveurs ASP classiques, ainsi qu'au solveur DLV∃ permettant de traiter des programmes ASP avec variables existentielles en tête, sur des tests techniques mettant en valeur les nouveautés qu'apporte ∃−ASPeRiX sur des opérations de base. Le but étant de comparer dans un premier temps les fonctionnalités disponibles permettant de traiter les programmes NME puis de comparer leur efficacité sur ces programmes après avoir été traduits en ASP classique. Nous testons ici la réaction du solveur lors de l'utilisation de têtes multi-atomiques, de variables existentielles en tête, de négations par défaut multi-atomiques et en présence d'answer set infinis limités par un nombre maximal d'imbrication de symboles de fonction.

### 5.2.1 Principe

Tout d'abord seule la version étendue d'ASPeRiX est compatible avec le format de règles non-monotones existentielles proposé dans cette thèse. Nous avons donc utilisé une série de tests techniques dont une partie est basée sur l'ontologie university qui a dans un premier temps été traduite dans un format adapté aux règles existentielles puis à laquelle nous avons ajouté de la négation par défaut dans certains cas, absente de l'ontologie originale. Pour l'ensemble des exemples proposés nous avons utilisé le module de traduction afin de tester ces exemples sur d'autres solveurs ASP (Remarque : le grounder gringo ne gère pas les guillemets pour les noms de prédicats ce qui pose soucis lorsque l'on souhaite calculer des modèles à partir d'ontologies utilisant des URI). Nous vous proposons le tableau de compatibilité 5.1 avec l'ensemble des fonctionnalités des solveurs présentés. Nous avons testé l'ensemble des solveurs sur trois versions de l'ontologie, l'originale (sans négation par défaut), une version skolémisée toujours sans négation par défaut puis une version avec atomes multiples en tête. Dans cette dernière version toutes les règles ayant le même corps sont combinées en une unique règle avec en tête un ensemble d'atomes correspondant à l'union des têtes de chacune de ces règles. Nous avons ensuite ajouté de la négation par défaut tout d'abord pour tester l'efficacité de l'utilisation des corps négatifs multi-atomiques, puis un exemple avec une infinité d'answer set. L'ensemble des tests ont été effectués sur un processeur Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz avec 4Gb de RAM hormis les tests sur l'ontologie university complète qui a nécessité plus de RAM.

TABLE 5.1 – Tableau de compatibilité

| | ASPeRiX | Clasp | DLV | DLV$^{\exists}$ | ∃−ASPeRiX |
|---|---|---|---|---|---|
| Symboles de fonction (Skolem) | ✓ | ✓ | ✓ | ✓ | ✓ |
| Prédicats entre guillemets | ✓ | × | ✓ | ✓ | ✓ |
| Tête multi-atomique | × | × | × | × | ✓ |
| Variables existentielles en tête | × | × | × | ✓(avec #exists{X}) | ✓ |
| Variables existentielles dans les corps négatifs | × | × | × | × | ✓ |
| Corps négatifs avec ensemble d'atomes | × | × | × | × | ✓ |

✓= fonctionnalité supportée
x = fonctionnalité non supportée

Les versions des solveurs testées sont les suivantes :

- ASPeRiX 0.2.5

- Clasp [42] 3.1.0 avec Gringo [41, 32] 4.4.0
- DLV [55](December 17th, 2012)
- DLV$^{\exists}$ [8] (version provenant de `https://www.mat.unical.it/kr2012/` juin 2016)
- ∃−ASPeRiX 0.2.5 (extension de la branche d'ASPeRiX 0.2.5)

### 5.2.2 Jeux de test

Nous présentons ici l'ensemble des tests effectués pour comparer ∃−ASPeRiX aux autres solveurs ASP ainsi que les temps obtenus pour chaque test. Ces tests sont des opérations de base permettant d'illustrer le tableau 5.1 précédent, ils mettent en valeur les qualités et les défauts de l'extension ∃−ASPeRiX et l'intérêt des fonctionnalités ajoutées.

**Premier test** Génération d'une instanciation avec et sans tête multi-atomique. Nous testons ici l'efficacité du solveur en présence d'une tête multi-atomique plutôt que l'utilisation de plusieurs règles possédant le même corps.

**TMA.** Le jeu de test TMA (tête multi atomique) teste l'application d'une règle avec 3 atomes en tête avec 1 000 000 d'instances.

**TMAe.** Le jeu de test TMAe (tête multi-atomique expansée) représente la version expansée du jeu TMA avec 3 règles avec tête mono-atomique permettant d'obtenir le même résultat.

| TMA | TMAe |
|---|---|
| personne1(1..1000). | personne1(1..1000). |
| personne2(1001..2000). | personne2(1001..2000). |
| pere(X),fils(Y),memeFamille(X,Y) :-parent(X,Y). | pere(X) :-parent(X,Y). |
| parente(X,Y) :- pere(X),fils(Y),memeFamille(X,Y). | fils(Y) :-parent(X,Y). |
| | memeFamille(X,Y) :-parent(X,Y). |
| | parente(X,Y) :-pere(X),fils(Y),memeFamille(X,Y). |

| solveur | TMA | TMAe |
|---|---|---|
| Clasp | × | 5s |
| DLV | × | 8.5s |
| ∃DLV | × | 8.5s |
| ASPeRiX | × | 7.5s |
| ∃−ASPeRiX | 7.5s | 7.5s |

TABLE 5.2 – Tableau de comparaison tête multi-atomique

En théorie la version multi-atomique en tête devrait être plus rapide que la version classique car nous testons une seule fois si la règle avec tête multi-atomique est applicable, au lieu de trois dans le cas du programme expansé, en pratique le temps reste identique car l'extension du solveur ne prend pas encore en compte efficacement les têtes multi-atomiques. Il y a autant d'instanciation dans les deux cas alors qu'il y a plus de règles dans la version expansée. L'amélioration de l'implémentation permettrait de réduire le temps d'instanciation lors du traitement d'une règle multi-atomique.

**TCMA.** Le jeu de test TCMA (tête et corps multi-atomiques) teste l'application d'une règle avec 3 atomes en tête et deux atomes dans le corps avec 1 000 000 d'instances.

**TCMAe.** Le jeu de test TCMAe(tête et corps multi-atomiques expansée) représente la version expansée du jeu TCMA avec 3 règles avec tête mono-atomique permettant d'obtenir le même résultat.

| TCMA | TCMAe |
|---|---|
| p(1..50000). | p(1..50000). |
| q(1..50000). | q(1..50000). |
| t1(X),t2(Y),t3(X,Y) :-p(X), q(X). | t1(X) :-p(X), q(X). |
| | t2(X) :-p(X), q(X). |
| | t3(X) :-p(X), q(X). |

| solveur | TCMA | TCMAe |
|---|---|---|
| Clasp | × | 0.2s |
| DLV | × | 0.5s |
| ∃DLV | × | 0.5s |
| ASPeRiX | × | 7.5s |
| ∃-ASPeRiX | 23.8s | 7.5s |

TABLE 5.3 – Tableau de comparaison tête multi-atomique

Dans ce test, le temps est amélioré pour la version avec tête multi-atomique pour ∃-ASPeRiX même si celui-ci est beaucoup moins performant pour l'instanciation du corps comparé aux solveurs Clasp et DLV.

**Second test.** Génération d'un answer set infini s'arrêtant à 12 000 imbrications de symboles de fonction. Nous testons ici l'efficacité du solveur à traiter les variables existentielles en tête, et le comparons au solveur DLV$^{\exists}$. Pour cela nous testons avec un programme dont l'answer set est infini en ne calculant que 12 000 imbrications de symboles de fonction pour obtenir un answer set. Pour cela nous testons quatre versions différentes :

**RE.** Le jeu RE (règles existentielles) est un programme NME calculant l'existence d'un parent homme pour tout homme. Ce programme n'est composé que de règles existentielles et est un programme classique pour illustrer les programmes avec modèle infini. Le but étant de calculer les 12 000 premiers parents hommes d'un individu titi.

**RESk.** Le jeu RESk (règles existentielles skolémisées) est la version skolémisée et expansée du jeux RE. La skolémisation et l'expansion sont directement effectuées par ∃-ASPeRiX. Ce jeu est une version compatible avec l'ensemble des solveurs ASP et les answer set de ce programme sont identiques aux answer set du jeu RE.

**REdlve.** Le jeu REdlve (règles existentielles DLV$^{\exists}$), DLV$^{\exists}$ n'autorisant pas les têtes multi-atomiques, nous avons créé une version expansée conservant le lien avec la variable existentielle en tête de règle et utilisant la syntaxe proposée pour l'utilisation de variables existentielles dans le manuel d'utilisation de DLV$^{\exists}$. Les answer set de ce programme sont identiques aux answer set du jeu RE à l'ajout de l'atome `supportHomme` près, bien que le programme soit différent.

**REdlvSk.** Le jeu REdlvSk (règles existentielles DLV$^{\exists}$ skolémisées) est une version skolémisée et expansée du jeu REdlv permettant de tester les autres solveurs.

| RE | RESk |
|---|---|
| parent(Y,X),homme(Y) :-homme(X).<br>homme(titi). | homme(skR1Y(X)) :- homme(X).<br>parent(skR1Y(X),X) :- homme(X).<br>homme(titi). |

| REdlve | REdlveSk |
|---|---|
| #exists{Y} supportHomme(Y,X) :-homme(X).<br>parent(Y,X) :-supportHomme(Y,X).<br>homme(Y) :-supportHomme(Y,X).<br>homme(titi). | supportHomme(skR1Y(X),X) :-homme(X).<br>parent(Y,X) :-supportHomme(Y,X).<br>homme(Y) :-supportHomme(Y,X).<br>homme(titi). |

| solveur | RE | RESk | REdlve | REdlveSk |
|---|---|---|---|---|
| Clasp | × | × | × | × |
| DLV | × | 3.5s | × | 8.7s |
| ∃DLV | × | 1.8s | 0.02s | 5.2s |
| ASPeRiX | × | 4.7s | × | 2.2s |
| ∃−ASPeRiX | 4.7s | 4.7s | × | 2.4s |

TABLE 5.4 – Tableau de comparaison règles existentielles

Ici pour ∃−ASPeRiX le temps est plus long pour la version classique comparée à la version proposée par DLV avec l'atome supportHomme. Nous remarquons que DLV est plus efficace sur la version classique que ∃−ASPeRiX mais que cela s'inverse pour la version proposée par DLV. Ceci est sûrement lié à la façon de gérer les applications de règles entre les deux solveurs. Pour le cas du jeu REdlv, le calcul du modèle n'est pas fait entièrement, dans le modèle calculé les variables existentielles ne sont pas instanciées, probablement pour éviter le calcul d'un modèle infini. Le modèle ainsi obtenu est :

$$\texttt{parent}(\texttt{fparent1}(\#1,\texttt{titi}),\texttt{titi}),\texttt{homme}(\texttt{titi})$$

Il n'est pas correct d'un point de vue sémantique en ASP, le modèle calculé n'est pas un answer set. Ce modèle permet simplement de répondre à une interrogation dans DLV$^{\exists}$. Le temps obtenu ne peut donc pas être comparé avec les temps des autres solveurs.
Pour le cas de Clasp, il ne permet pas de limiter le nombre d'imbrications de symboles de fonction, il n'est donc pas possible de tester cet exemple.

**Troisième test.** Ce test provient de l'ontologie university DL-Lite, dont la $\mathcal{T}BOX$ a été traduite en programme NME sans négation par défaut, puis testée avec plusieurs versions (expansée, skolémisée,...). La traduction de cette ontologie est décrite dans la section 5.3.2, elle ne contient pas de négation par défaut mais des règles existentielles avec tête multi-atomiques. Nous testons ici avec les faits suivants :

$$\texttt{student}(1..500000).$$
$$\texttt{professor}(1..500000).$$

dont l'unique answer set est le suivant :

$$AS = \{\texttt{student}(1..500000),\texttt{professor}(1..500000),\texttt{facultystaff}(1..500000),$$
$$\texttt{employee}(1..500000),\texttt{worksfor}(1..500000,\texttt{skr57X2}(1..500000)),$$
$$\texttt{memberof}(1..500000,\texttt{skr57X2}(1..500000)),\texttt{member}(\texttt{skr57X2}(1..500000),1..500000),$$
$$\texttt{organization}(\texttt{skr57X2}(1..500000)),\texttt{person}(1..500000),\texttt{course}(\texttt{skr73X2}(1..500000)),$$
$$\texttt{work}(\texttt{skr73X2}(1..500000)),\texttt{takescourse}(1..500000,\texttt{skr73X2}(1..500000))\}$$

TABLE 5.5 – Tableau de comparaison `university` DL-Lite

| solveur | university | uC | uMA | udlve | uMASk |
|---|---|---|---|---|---|
| Clasp | × | 9.1s | × | × | × |
| DLV | × | 22s | × | × | × |
| DLV$^{\exists}$ | × | 20.8s | × | 30.5s | × |
| ASPeRiX | × | 6.7s | × | × | × |
| ∃−ASPeRiX | 7s | 6.4s | 6.8s | 6.9s | 6.4s |

**university.** Le benchmark « university » est l'ontologie `university` en version DL-Lite transformée en programme Datalog, avec cette syntaxe seul ∃−ASPeRiX est capable de le gérer grâce aux mutli-atomes en tête et aux variables existentielles.

**uC.** Le benchmark uC (university Clasp) est une version skolémisée et expansée de university afin que le programme soit compatible avec tous les solveurs, pour cela les règles existentielles du type :

$$r1 : \text{"takesCourse"}(\texttt{X1}, \texttt{X2}), \text{"Course"}(\texttt{X2}) :\text{-}\ \text{"Student"}(\texttt{X1}).$$

sont réécrites en :

$$\begin{aligned} r1 :&\ \text{"takesCourse"}(\texttt{X1}, \texttt{skr1X2}(\texttt{X1})) :\text{-}\ \text{"Student"}(\texttt{X1}).\\ r2 :&\ \text{"Course"}(\texttt{skr1X2}(\texttt{X1})) :\text{-}\ \text{"Student"}(\texttt{X1}).\end{aligned}$$

De plus chaque prédicat entre guillemets est transformé en prédicat équivalent sans guillemets et en minuscule pour être compatible avec Clasp.

**uMA.** Le benchmark uMA (university multi-atomes) est une version optimisée de university remplaçant les règles ayant un corps identique avec une unique règle dont la tête est l'union des têtes de ces règles.

**Exemple 96.** *Soit la règle :*

$$\text{"Person"}(\texttt{X}) :\text{-}\ \text{"advisor"}(\texttt{X}, \texttt{Y}).\text{"Professor"}(\texttt{Y}) :\text{-}\ \text{"advisor"}(\texttt{X}, \texttt{Y}).$$

*devient :*

$$r1 : \text{"Person"}(\texttt{X}), \text{"Professor"}(\texttt{Y}) :\text{-}\ \text{"advisor"}(\texttt{X}, \texttt{Y}).$$

Ce benchmark est compatible uniquement avec ∃−ASPeRiX

**udlve.** Le benchmark (university DLV∃) reprend la méthode proposée par DLV$^{\exists}$ dans le manuel d'utilisation pour traduire les règles existentielles. Ainsi la traduction de tête multi-atomique est faite de cette manière :

$$r1 : \text{"takesCourse"}(\texttt{X1}, \texttt{X2}), \text{"Course"}(\texttt{X2}) :\text{-}\ \text{"Student"}(\texttt{X1}).$$

en :

$$\begin{aligned} r1 :&\ \#exists\{\texttt{X2}\}\text{"SupportStudent"}(\texttt{X1}, \texttt{X2}) :\text{-}\ \text{"Student"}(\texttt{X1}).\\ r2 :&\ \text{"takesCourse"}(\texttt{X1}, \texttt{X2}) :\text{-}\ \text{"SupportStudent"}(\texttt{X1}, \texttt{X2}).\\ r3 :&\ \text{"Course"}(\texttt{X2}) :\text{-}\ \text{"SupportStudent"}(\texttt{X1}, \texttt{X2}).\end{aligned}$$

**uMASk.** Le benchmark uMASk (university multi-atomes skolémisée) est la version skolémisée de uMA. À cause des têtes multi-atomiques ce benchmark n'est compatible qu'avec ∃−ASPeRiX. Cette version montre que le traitement de variables existentielles prend un peu plus de temps que la version skolémisée mais que la version multi-atomique permet d'en gagner.

L'ensemble de ces tests permettent de voir que le temps d'exécution reste stable pour ∃−ASPeRiX quelle que soit la manière d'écrire l'ontologie university, skolémisée, expansée, etc... Il met aussi en avant que sur ce type d'exemple ∃−ASPeRiX reste en concurrence avec les autres solveurs.

**Quatrième test.** Ce benchmark est une version d'university DL-Lite avec l'ajout d'une négation par défaut multi-atomiques qui permet l'obtention de plusieurs answer set. Ce benchmark crée toutes les différentes possibilités avec 12 étudiants qui peuvent être chacun soit étudiant classique soit doctorant.

```
phdStudent(X) :- facultystaff(X), not (student(X), takesCourse(X, Y)).
takesCourse(X, Y), normalStudent(X) :- student(X), not phdStudent(X).
student(1..12).
```

puis le même benchmark skolémisé, normalisé puis expansé :

```
r1N1(X) :- student(X), takesCourse(X, Y).
phdStudent(X) :- facultystaff(X), not r1N1(X).
normalStudent(X) :- student(X), not phdStudent(X).
takesCourse(X, skr4Y(X)) :- student(X), not phdStudent(X).
student(1..12).
```

Ces programmes donnent 4096 answer set.

**uN.** Le benchmark uN (university avec négation par défaut) est la version classique du problème utilisant tous les ajouts d'∃−ASPeRiX comme les variables existentielles les têtes multi-atomiques et les corps négatifs multi-atomiques. Cette version n'affiche pas les answer set pour ne pas prendre en compte le temps d'affichage long pour 4096 answer set.

**uND.** Le benchmark uND (university avec négation par défaut et display) est la version du benchmark uN avec affichage des answer set. L'affichage étant assez long par rapport au calcul des answer set, ce test permet d'identifier le temps nécessaire pour l'affichage.

**uCN.** Le benchmark uCN (university avec négation par défaut Clasp) est une version skolémisée, normalisée puis expansée de uN. Ce benchmark est testé sans affichage.

**uCND.** Le benchmark uCND est la version du benchmark uCN avec affichage des answer set.

Les résultats des solveurs sont les suivants : Ce benchmark permet de mettre en avant la réaction d'∃−ASPeRiX avec un exemple de programme avec plusieurs answer set et l'ensemble des fonctionnalités ajoutées. Pour ∃−ASPeRiX nous observons une amélioration du temps liée à la tête multi-atomique (avec ou sans affichage). En comparaison avec les autres solveurs, ∃−ASPeRiX n'est pas assez efficace et nécessiterait des améliorations au niveau de l'algorithme de résolution du solveur original ASPeRiX. On note par contre un temps quasi-instantané lorsque l'affichage n'est pas demandé. Nous n'avons pas identifié si l'answer set était bel et bien calculé mais la différence de temps entre l'affichage et le calcul uniquement est beaucoup plus grande que sur ASPeRiX, il est donc très

TABLE 5.6 – Tableau de comparaison `university` DL-Lite non stratifié

| solveur | uN | uND | uCN | uCND |
|---|---|---|---|---|
| Clasp | × | × | 0.2s | 9.9s |
| DLV | × | × | 0.095s | 9.9s |
| ∃DLV | × | × | 0.095 | 9.9 |
| ASPeRiX | × | × | 8.5s | 11.9s |
| ∃−ASPeRiX | 7.5s | 11.4s | 9.4s | 12.6s |

probable qu'il ne soit pas calculé.

L'ensemble des tests techniques présentés jusqu'à maintenant servent principalement à tester le bon fonctionnement du solveur, et des nouvelles fonctionnalités. Le solveur ASPeRiX n'étant déjà pas aussi efficace que DLV et Clasp, ce problème persiste sur l'extension d'ASPeRiX même si, sur quelques exemples, il est possible de voir quelques améliorations liées aux têtes multi-atomiques. Il est par contre intéressant de noter que l'expressivité du nouveau langage permet de représenter de la connaissance plus simplement et que seul ∃−ASPeRiX permet de traiter ces programmes sans transformation préalable.

## 5.3 Comparaisons sur l'ontologie `university` complète

Dans cette série de tests nous avons comparé le solveur ∃−ASPeRiX avec le plugin NoHR de Protégé permettant d'ajouter des règles avec négation par défaut à une ontologie puis d'interroger le programme résultant. Nous avons dans un premier temps traité l'ontologie `university` en la traduisant en programme NME puis nous l'avons interrogée avec les requêtes fournies avec l'ontologie. Dans un second temps nous avons ajouté des règles non-monotones provenant d'abord de l'article [30] puis les deux autres règles NME proposées dans la partie précédente. Nous allons d'abord décrire l'ontologie puis la traduction utilisée et enfin comparer les résultats obtenus et les concessions faites pour obtenir un résultat. L'ontologie est divisée en plusieurs parties avec d'un côté :

- une TBOX représentant l'ensemble des connaissances terminologiques sur l'organisation d'une université ;
- des ABOX que l'on peut générer avec un générateur représentant l'ensemble des universités et leurs départements. Ainsi pour une université générée, il y a entre 15 et 25 départements générés aléatoirement à partir d'une graine aléatoire [1] ;
- un ensemble de 14 requêtes SPARQL permettant d'interroger l'ontologie sur différents aspects. Les requêtes correspondent à des requêtes conjonctives non-booléennes.

### 5.3.1 Description de l'ontologie

L'ontologie `university` est donc une ontologie composée d'une TBOX, d'un générateur de ABOX et d'un jeu de 14 requêtes conjonctives non-booléennes. Cette ontologie est définie en OWL-Lite ($\mathcal{SHIF}$) ce qui nécessite une traduction pour la traiter avec le solveur ∃−ASPeRiX. `university`

[1] Une graine aléatoire correspond à un nombre permettant d'initialiser un générateur de nombre pseudo-aléatoires. Dans le cas d'`university` ce nombre est utilisé pour générer des universités différentes, avec un nombre de départements et de personnels dépendant de la graine utilisée.

est une ontologie académique servant de jeu de test pour comparer les différents outils traitant les ontologies et l'interrogation de données.

### Structure de la TBOX

La TBOX du benchmark `university` contient l'ensemble des règles définissant l'ontologie `university`. Elle peut être récupérée à l'adresse suivante :

http://swat.cse.lehigh.edu/onto/univ-bench.owl

Cette ontologie définit 43 classes et 32 propriétés (dont 25 `object properties` et 7 `datatype properties`). Elle utilise les fonctions du langage OWL-Lite suivantes

- `inverseOf`;
- `TransitiveProperty`;
- `someValuesFrom restrictions`;
- `intersectionOf`.

Cette ontologie comporte un petit nombre de classes mais beaucoup de restrictions et de propriétés par classe.

### Génération des instances avec UBA

L'outil UBA est un outil permettant de générer des données, définissant la base de faits, pour le benchmark `university`. Il est disponible à l'adresse suivante :

http://swat.cse.lehigh.edu/projects/lubm/uba1.7.zip

Les paramètres du générateur sont les suivants :

- `-univ <univ_num>` : pour définir le nombre d'universités à générer ; par défaut 1
- `-index <starting_index>` : pour définir l'index auquel démarre la première université ; par défaut 0
- `-seed <seed>` : changer la graine et donc générer des données différentes que celles par défaut ; par défaut 0
- `-daml` : génère des données DAML+OIL ; par défaut OWL
- `-onto <ontology_url>` : définit l'url de l'ontologie contenant les règles (dans notre cas la TBOX)

Les données générées dans l'article sur NoHR sont générées avec UBA pour 1,5,9,10,15, et 20 universités. Nous générons alors les ABOX avec la commande :

```
java edu.lehigh.swat.bench.uba.Generator -univ i
              -onto http://swat.cse.lehigh.edu/onto/univ-bench.owl
```

avec $i$ le nombre d'universités souhaité. UBA génère alors aléatoirement (selon une graine) entre 15 et 25 ABOX par université correspondant aux départements de l'université avec pour nom $Univesityi\_j.owl$ avec $i$ le numéro de l'université et $j$ le numéro de la ABOX correspondant au numéro du département allant de 0 à 25 en fonction du nombre de ABOX générées. Pour une université générée avec les paramètres par défaut, il y a 15 départements donc 15 ABOX générées allant de $University0\_0.owl$ à $University0\_14.owl$.

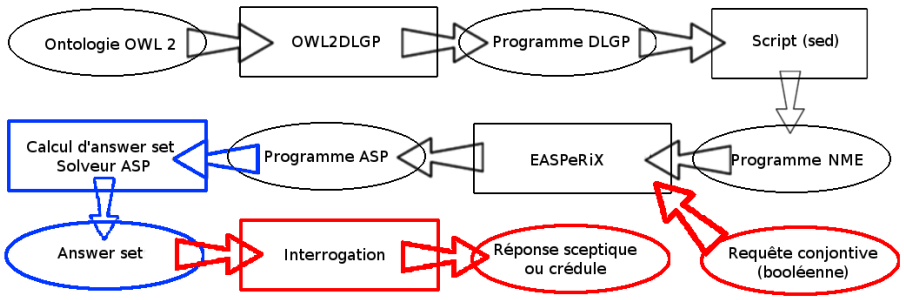



FIGURE 5.4 – Architecture de la traduction

**Les requêtes**

Il y a 14 requêtes conjonctives non booléennes dans le benchmark `university` sous un format SPARQL. Elles sont disponibles à cette adresse :

http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt

Les requêtes sont traduites à la main du format original vers ASP. Exemple avec la requête 9 :

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
 {?X rdf:type ub:Student .
  ?Y rdf:type ub:Faculty .
  ?Z rdf:type ub:Course .
  ?X ub:advisor ?Y .
  ?Y ub:teacherOf ?Z .
  ?X ub:takesCourse ?Z}
```

traduite en :

```
ans(X,Y,Z) :- "Student"(X), "Faculty"(Y), "Course"(Z), "advisor"(X,Y),
                    "takesCourse"(X,Z), "teacherOf"(Y,Z).
```

Des outils de traduction de requêtes SPARQL vers des requêtes compatibles avec le format présenté ici existent mais n'ont pas été utilisés pour ces exemples. Les requêtes ayant été traduites avant d'avoir eu connaissance de ces outils. Dans cette thèse seules les 14 requêtes de `university` étaient nécessaires et donc rapidement traduisibles à la main, il serait par contre intéressant d'utiliser un outil pour un benchmark avec un nombre de requêtes plus important.

### 5.3.2 Traduction

Afin de pouvoir traiter l'ontologie avec le solveur ∃-ASPeRiX nous devons transformer l'ontologie en programme NME. Pour cela, nous traduisons l'ontologie en deux phases, la première phase consiste à traduire l'ontologie dans le format DLGP correspondant aux règles existentielles, la seconde phase consiste à retraiter le programme DLGP pour qu'il soit utilisable avec le solveur ∃-ASPeRiX. Les requêtes sont quant à elles traduites manuellement et intégrées au programme NME, mais un module de traduction permettrait de faciliter ce traitement par la suite. Le schéma de la figure 5.4 illustre le procédé de traduction et l'intégration de l'interrogation au programme.

### Traduction de OWL vers DLGP

La transformation de OWL vers DLGP se fait avec l'outil OWL2DLGP du LIRMM programmé en java récupérable à cette adresse

https://graphik-team.github.io/graal/owl2dlgp

Il faut effectuer la transformation pour la TBOX et l'ensemble des ABOX de l'ontologie. en exécutant de la manière suivante.

```
java -jar owl2dlgp-*.jar -f entrée -o sortie
```

Pour que la traduction fonctionne correctement il faut que les entêtes soient correctes. Pour la TBOX récupérée sur le site du benchmark l'entête est

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns="http://swat.cse.lehigh.edu/onto/univ-bench.owl#"
xml:base="http://swat.cse.lehigh.edu/onto/univ-bench.owl"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#">
```

Pour les fichiers générés par l'outil UBA il faut spécifier lors de la génération l'adresse de l'ontologie

```
http://swat.cse.lehigh.edu/onto/univ-bench.owl
```

pour générer convenablement l'entête suivant dans les ABOX :

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:ub="http://swat.cse.lehigh.edu/onto/univ-bench.owl#">

<owl:Ontology rdf:about="">
<owl:imports rdf:resource=
            "http://swat.cse.lehigh.edu/onto/univ-bench.owl" />
</owl:Ontology>
```

Dans le fichier DLGP résultat l'ensemble des règles et faits sont traduits avec les prédicats préfixés par " :" dans la TBOX et "ub :" dans les ABOX. Ce format n'est pas compris par ∃-ASPeRiX et ne sert que lors de l'utilisation d'ontologies avec multi-source. Nous traduisons alors ces préfixes pour obtenir des prédicats interprétables par le solveur.

De même les constantes sont typées dans les ontologies et possèdent un type attaché à celles-ci non utilisé en ASP. Soit la constante suivante :

```
<ub:University rdf:about="http://www.University0.edu">
   <ub:name>University0</ub:name>
</ub:University>
```

traduite en :

```
ub:name(<http://www.University0.edu>,
"University0"=^^<http://www.w3.org/1999/02/22-rdf-syntax-ns# >).
```

Il faut là aussi retraduire pour que les constantes soient interprétables avec ∃-ASPeRiX.

Une fois la traduction de l'ontologie en règles existentielles (format DLGP) effectuée, le programme résultant comprend 106 règles dont 8 règles existentielles avec tête multi-atomique, 7 règles avec un corps multi atomique les autres étant des règles avec un seul atome en tête et dans le corps. Il n'y a que des atomes unaires et binaires en quantité presque égale. Il n'existe pas de négation par défaut dans l'ontologie originale.

#### Adaptation du format DLGP aux programme NME

La transformation de DLGP vers un programme NME se fait en traduisant les éléments DLGP non compatibles avec ∃-ASPeRiX. Pour cela, il suffit d'utiliser un script avec la commande sed dans un terminal pour effectuer le traitement suivant :

#### Suppression des entêtes de fichiers

- Suppression des lignes d'entête commençant par @

```
sed /^@.*/d;
```

- remplacement des lignes de constantes typées avec les ^^

```
sed s/"\(.*\)"^^<.*>/\<\1\>/g;
```

- remplacement des préfixes, par exemple " :" et "ub :", par des quotes autour des prédicats.

```
sed s/[^ ]*:\([^ ]*\)(/"\1"(/g
```

Cette traduction n'est pas assez générale pour être utilisée sur toutes les ontologies. Un module de traduction basé sur OWL2DLGP permettant de traduire directement de OWL2 vers un programme NME est à développer pour faciliter la traduction.

#### Requêtes

Ci-dessous l'ensemble des requêtes traduites à la main en interrogation pour ∃-ASPeRiX :

Q1 = ans(X) :- ”GraduateStudent”(X),
”takesCourse”(X,<http ://www.Department0.University0.edu/GraduateCourse0>).
Q2 = ans(X) :- ”GraduateStudent”(X), ”University”(Y), ”Department”(Z), ”memberOf”(X,Z),
”subOrganizationOf”(Z,Y), ”undergraduateDegreeFrom”(X,Y).
Q3 = ans(X) :- ”Publication”(X),
”publicationAuthor”(X,<http ://www.Department0.University0.edu/AssistantProfessor0>).
Q4 = ans(X,Y1,Y2,Y3) :- ”Professor”(X), ”name”(X,Y1), ”emailAddress”(X,Y2),
”telephone”(X,Y3),”worksFor”(X,<http ://www.Department0.University0.edu>).
Q5 = ans(X) :- ”Person”(X), ”memberOf”(X,<http ://www.Department0.University0.edu>).
Q6 = ans(X) :- ”Student”(X).
Q7 = ans(X) :- ”Student”(X), ”Course”(Y), ”takesCourse”(X,Y),
”teacherOf”(<http ://www.Department0.University0.edu/AssociateProfessor0>,Y).
Q8 = ans(X,Y,Z) :- ”Student”(X), ”Department”(Y), ”memberOf”(X,Y), ”emailAddress”(X,Z),
”subOrganizationOf”(Y,<http ://www.University0.edu>).
Q9 = ans(X,Y,Z) :- ”Student”(X), ”Faculty”(Y), ”Course”(Z), ”advisor”(X,Y), ”takesCourse”(X,Z),
”teacherOf”(Y,Z).
Q10 = ans(X) :- ”Student”(X),
”takesCourse”(X,<http ://www.Department0.University0.edu/GraduateCourse0>).
Q11 = ans(X) :- ”ResearchGroup”(X), ”subOrganizationOf”(X,<http ://www.University0.edu>).
Q12 = ans(X,Y) :- ”Chair”(X), ”Department”(Y), ”worksFor”(X,Y),
”subOrganizationOf”(Y,<http ://www.University0.edu>).
Q13 = ans(X) :- ”Person”(X), ”hasAlumnus”(<http ://www.University0.edu>,X).
Q14 = ans(X) :- ”UndergraduateStudent”(X).

### 5.3.3 Série de tests

Pour effectuer les interrogations avec ∃−ASPeRiX il faut fusionner l'ensemble des règles de la TBOX avec les données provenant des ABOX + une requête. Par exemple, le programme NME avec la TBOX, la première ABOX générée par uba et la première requête de `university`.

```
cat univ-bench.owl.asp > university.asp
cat University0_0.owl.asp >> university.asp
cat query1.asp >> university.asp
```

#### `university` classique

Pour réaliser les tests avec ∃−ASPeRiX nous utilisons une partie d'une seule université générée, l'interrogation n'étant pas encore implémentée de manière efficace, le solveur n'est pas en mesure de traiter efficacement l'ontologie complète notamment car il n'isole pas uniquement les règles dont la requête dépend ni les instances strictement nécessaires. Les éléments présentés dans le chapitre 4 n'étant pas encore implémentés, il est nécessaire pour le solveur de calculer entièrement les answer set pour chaque requête jusqu'à l'obtention d'une réponse. Une fois l'ensemble des fonctionnalités implémentées il sera possible de ne calculer qu'une partie des answer set pour y répondre. Soit les ABOX University0_0 à University0_14 pour la génération d'une université, nous sélectionnons une ABOX permettant d'obtenir une réponse à une interrogation choisie parmi les 14 proposées par le benchmark `university`. Il n'y a bien sûr ici qu'un seul answer set étant donné qu'il n'y a pas de négation par défaut. Pour toutes les requêtes la ABOX University0_0.owl est suffisante pour obtenir une réponse sauf pour la requête Q2 où la réponse est vide. Pour tester les requêtes avec ∃−ASPeRiX :

./asperix -QC universityq1.asp

pour chaque requête avec universityq1.asp correspondant au programme composé de la TBOX de la première ABOX University0_0.owl.asp et de la première requête au format d'un programme NME ce qui donne les résultats suivants

| $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $Q_5$ | $Q_6$ | $Q_7$ | $Q_8$ | $Q_9$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21s | 20s | 28s | 20,5s | 21 | 20,6 | 22s | 21s | 1m31 | 21s | 21s | 21s | 21s | 21s |

On peut aussi juste calculer les answer set en enlevant l'option -QC, le temps reste alors identique mis à part l'affichage de l'answer set. Le temps beaucoup plus long de la requête 9 provient du fait qu'ASPeRiX teste l'ensemble des instanciations possibles dans l'ordre de l'apparition des variables. Soit

$$Q9 = \; ans(X,Y,Z) : -"Student"(X),"Faculty"(Y),"Course"(Z),"advisor"(X,Y), \\ "takesCourse"(X,Z),"teacherOf"(Y,Z).$$

ASPeRiX va essayer toutes les combinaisons possibles pour X, Y, Z alors que si l'on déplace "Faculty"(Y), "Course"(Z) après "advisor"(X,Y), "takesCourse"(X,Z), ASPeRiX ne testera que les possibilités pour Y et Z correspondant aux instances de "advisor"(X,Y), "takesCourse"(X,Z). Donc le temps de calcul pour :

$$Q9 = \; ans(X,Y,Z) : -"Student"(X),"advisor"(X,Y),"takesCourse"(X,Z), \\ "Faculty"(Y),"Course"(Z),"teacherOf"(Y,Z).$$

est de 20,5 secondes.
Ces tests sur l'ontologie permettent seulement de vérifier le bon fonctionnement de l'interrogation avec le solveur ∃−ASPeRiX, les temps obtenus ne sont pas comparables avec ceux obtenus par Protégé avec NoHR car le traitement de l'interrogation n'est pas encore optimisé, de même ∃−ASPeRiX n'est pas optimisé pour traiter de gros fichiers de données, le parser de celui-ci doit-être réécrit pour traiter l'ontologie complète.
Il est néanmoins possible d'obtenir des temps plutôt bons avec les solveurs DLV et Clasp une fois l'ontologie traduite en ASP. Pour preuve le premier tableau de test sur le même exemple qu'∃−ASPeRiX, avec 1 université et 1 département :

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $Q_5$ | $Q_6$ | $Q_7$ | $Q_8$ | $Q_9$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DLV | 0.7s | 1s | 0.6s | 0.8s | 1s | 1s | 0.7s | 1.9s | 1s | 0.7s | 0.7s | 0.8s | 0.7s | 0.9s |
| Clasp | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s | 1s |

Et le second tableau de test avec cette fois-ci 20 universités complètes :

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $Q_5$ | $Q_6$ | $Q_7$ | $Q_8$ | $Q_9$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DLV | 11s | 22s | 12s | 31s | 35s | 23s | 35s | 38s | 23s | 30s | 11s | 31s | 29s | 23s |
| Clasp | 27s | 28s | 27s | 27s | 27s | 27s | 27s | 28s | 29s | 29s | 29s | 29s | 29s | 29s |

### `university` avec négation par défaut

Nous avons souhaité tester aussi le résultat d'une requête lors de la présence d'une règle avec négation par défaut dans l'ontologie university, nous avons donc dans un premier temps ajouté la règle extraite de [30] :

```
"Replacement"(X,Y) :- "Professor"(X), "worksFor"(X,Y),
     "lowTeachingLoad"(X), not "onSabbatical"(X), not "ill"(X).
```

Nous ajoutons alors la requête suivante permettant d'interroger le programme sur une déduction provenant de l'application de la règle avec négation par défaut :

ans(X,Y) :- "Replacement"(X,Y).

pour obtenir une réponse à cette requête nous ajoutons aussi les faits suivants :

```
"lowTeachingLoad"(<http://www.Department0.University0.edu/FullProfessor0>).
"lowTeachingLoad"(<http://www.Department0.University0.edu/AssistantProfessor0>).
"lowTeachingLoad"(<http://www.Department0.University0.edu/AssociateProfessor0>).
"lowTeachingLoad"(<http://www.Department0.University0.edu/FullProfessor1>).
"onSabbatical"(<http://www.Department0.University0.edu/FullProfessor0>).
"ill"(<http://www.Department0.University0.edu/AssistantProfessor0>).
"onSabbatical"(<http://www.Department0.University0.edu/FullProfessor1>).
"ill"(<http://www.Department0.University0.edu/FullProfessor1>).
```

Nous avons donc notre règle appliquée uniquement pour l'instanciation avec AssociateProfessor0 les autres instances étant bloquées soit par une instance de "ill", une instance de OnSabbatical" ou les deux. La réponse sceptique/crédule obtenue est :

```
{ [ X <- "http://www.Department0.University0.edu/AssociateProfessor0",
    Y <- "http://www.Department0.University0.edu" ],
  [ X <- "http://www.Department0.University0.edu/AssociateProfessor0",
    Y <- skR8X3("http://www.Department0.University0.edu/AssociateProfessor0") ] }
```

Il n'y a là aussi qu'un seul answer set car il n'y a pas de point de choix donc la réponse sceptique et crédule est identique. Le temps de réponse obtenu pour cette requête avec $\exists$−ASPeRiX est de 22s, le temps correspond au temps nécessaire à la majorité des requêtes testées sur university car le calcul de l'answer set est le même à quelques règles près. La réponse est correcte car seules les instances avec "http ://www.Department0.University0.edu/AssociateProfessor0" ne sont pas bloquées. Nous pouvons aussi tester une requête booléenne conjonctive comme :

$$\texttt{ans} : \texttt{-}\ \text{"}\texttt{Replacement}\text{"}(\texttt{X}, \texttt{Y}).$$

dont le calcul de la réponse renvoie true.

#### university avec négation par défaut et multiple answer set

Dernier exemple avec l'ajout des règles avec négation par défaut suivantes à l'exemple précédent :

$$\text{"}\texttt{phdStudent}\text{"}(\texttt{X}) : \texttt{-}\ \text{"}\texttt{Person}\text{"}(\texttt{X}), \texttt{not}\ (\text{"}\texttt{Student}\text{"}(\texttt{X}), \text{"}\texttt{takesCourse}\text{"}(\texttt{X}, \texttt{Y})).$$
$$\text{"}\texttt{takesCourse}\text{"}(\texttt{X}, \texttt{Y}), \text{"}\texttt{Student}\text{"}(\texttt{X}) : \texttt{-}\ \text{"}\texttt{Person}\text{"}(\texttt{X}), \texttt{not}\ \text{"}\texttt{phdStudent}\text{"}(\texttt{X}).$$

et la requête :

$$\texttt{ans}(\texttt{X}) : \texttt{-}\ \text{"}\texttt{phdStudent}\text{"}(\texttt{X}).$$

Le calcul des answer set est très long car dépendant du nombre d'instances de "Person" qui est très grand. La réponse est correcte si nous réduisons le nombre d'instances de "Person".
De même pour une réponse à une requête booléenne conjonctive comme par exemple

$$\texttt{ans} : \texttt{-}\ \text{"}\texttt{phdStudent}\text{"}(\texttt{X}).$$

### `university` DL-Lite

Le solveur `ASPeRiX` n'étant pas performant sur l'ontologie originale `university` au format OWL-Lite($\mathcal{SHIF}$). Nous proposons une série de tests sur une version DL-Lite de celle-ci, au format OWL 2 QL, cette version ne possède que des règles linéaires ce qui facilite le traitement de l'ontologie. C'est aussi l'approche adoptée dans l'article [30] dont l'outil NoHR passe par une traduction soit au format EL (plus lent) ou au format QL. Dans cette partie nous comparons uniquement les solveurs sur le format QL.

Pour ce test nous utilisons la traduction de l'ontologie `university` proposée dans [63] pour les solveurs DLV, clasp et `∃-ASPeRiX`, et l'ontologie `university` originale traduite en pré-traitement par l'outil NoHR. Afin de comparer de manière correcte tous les solveurs nous détaillons les temps nécessaires à l'obtention d'un format compatible avec les solveurs et les temps nécessaires à l'initialisation pour NoHR.

Dans un premier temps NoHR charge l'ontologie dans le programme ce qui demande 48,5 secondes puis effectue une traduction de OWL-Lite vers OWL QL qui n'est pas nécessaire pour les autres solveurs car nous utilisons la traduction proposée dans l'article [30]. Le temps nécessaire pour la traduction est de 16,5 secondes pour NoHR.

D'un autre côté, pour être compatible avec les solveurs ASP nous avons besoin de traduire dans un premier temps l'ontologie au format DL-Lite QL vers le format DLGP des règles existentielles. Pour cela il faut 610 secondes pour la traduction. Puis dans le format d'`∃-ASPeRiX` qui demande 43 secondes et enfin en programme ASP 30 secondes.

Ensuite, pour la résolution, NoHR nécessite une phase préalable à l'interrogation pour charger le programme en mémoire pour XSB ce qui demande 42,1 secondes, mais ce chargement n'est appliqué qu'une seule fois, les réponses aux requêtes étant calculées directement sur le fichier chargé dans XSB.

| | chgmt OWL | OWL → QL | chgmt + OWL → DLGP | chgmt + DLGP → NME | chgmt + NME → ASP | chgmt XSB |
|---|---|---|---|---|---|---|
| *NoHR* | 48,5s | 16,5s | - | - | - | 42,1s |
| `∃-ASPeRiX` | - | - | 610s | 43s | 30s | - |
| *DLV* | - | - | 610s | 43s | 30s | - |
| *Clasp* | - | - | 610s | 43s | 30s | - |

Les initialisations pour NoHR et ASP sont très différentes, étant donné que NoHR ne charge l'ontologie qu'une seule fois en mémoire alors que pour ASP le chargement est fait à chaque étape de la traduction (la traduction étant effectuée par des outils séparés). La traduction ASP n'est pas optimisée car considérée comme un pré-traitement, de plus une fois effectuée nous n'avons plus besoin de la recommencer pour effectuer le calcul des modèles, il serait même possible de créer un générateur de faits directement en ASP et alors le temps serait grandement amélioré.

Une fois ces initialisations faites nous interrogeons l'ontologie au format OWL QL pour NoHR, ASP pour Clasp et DLV, et sous forme de programme NME pour `∃-ASPeRiX`. Pour l'interrogation NoHR pose les 14 requêtes sur l'ontologie entièrement initialisée tandis que les requêtes sont posées une par une pour les solveurs ASP et `∃-ASPeRiX`. Les requêtes étant de simples règles en ASP, il est en théorie possible de poser les 14 requêtes en une seule fois si l'on ajoute un atome réponse par requête et que l'on demande d'afficher les substitutions pour ces atomes. Ainsi pour `∃-ASPeRiX` et Clasp le temps pour répondre à une interrogation serait quasiment le même que pour répondre aux 14 requêtes en une fois, comme aucune optimisation sur la requête n'est effectuée (les answer set sont entièrement calculés). Pour DLV et NoHR qui ont un module d'interrogation optimisé le temps dépend de la requête (le modèle n'est pas calculé entièrement).

Les $\mathcal{A}Box$ utilisées sont les mêmes que dans les parties précédentes avec 10 universités.

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $Q_5$ | $Q_6$ | $Q_7$ | $Q_8$ | $Q_9$ | $Q_{10}$ | $Q_{11}$ | $Q_{12}$ | $Q_{13}$ | $Q_{14}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∃−ASPeRiX | 390s | 394s | 392s | 405s | 385s | 396s | 396s | 397s | 420s | 390s | 386s | 395s | 390s | 400s |
| $NoHR$ | - | 0,8s | 0,01s | 0,121s | 0,135s | 71,36s | 1,6s | 45s | 13s | 0,02s | - | - | 0,01s | 75s |
| $DLV$ | 5s | 10s | 6s | 15s | 16s | 11s | 16s | 18s | 11s | 15s | 5s | 15s | 14s | 10s |
| $Clasp$ | 12s | 12s | 12s | 12s | 12s | 12s | 12s | 12s | 12 | 12s | 12s | 12s | 12s | 12s |

Notons que pour l'utilisation d'NoHR avec 10 universités il faut environ 50 secondes pour initialiser l'ontologie puis 16 secondes pour la traduire, plus 42 secondes pour la charger dans XSB, temps nécessaire à chaque exécution alors qu'une fois traduite en ASP le temps d'initialisation fait partie du temps nécessaire au calcul des modèles et il n'est pas possible de dissocier ces temps pour les solveurs ASP.

Pour conclure, NoHR est efficace sur des requêtes spécifiques et propose une réponse presque instantanée une fois l'initialisation faite, mais certaines requêtes ne permettent pas l'obtention d'une réponse. D'un autre côté les solveurs ASP proposent des temps plus long que NoHR pour les requêtes qui étaient instantanées mais ils sont beaucoup plus efficaces pour les requêtes qui demandent beaucoup de traitement (comme la requête 6 par exemple). Seul DLV propose une interrogation ne calculant pas tous les modèles pour les solveurs ASP, mais dans tous les cas le calcul complet du modèle pour DLV et Clasp est plus court que le temps de réponse cumulé de toute les requêtes avec NoHR, nous pouvons donc en déduire que si nous posons les 14 requêtes en une fois avec les solveurs ASP le temps sera meilleur que celui d'NoHR demandant environ 190 secondes contre 12 secondes pour Clasp et environ 20 secondes pour DLV. Pour terminer, ∃−ASPeRiX a du retard sur ses concurrents à cause de l'absence d'optimisations pour le calcul des answer set dans le solveur initial.

### 5.3.4 Discussion

Dans ce chapitre nous avons mis en avant le développement d'une extension du solveur ASPeRiX pour traiter des programmes NME. En l'état actuel ∃−ASPeRiX a l'avantage d'être le seul solveur mettant en pratique l'ensemble des éléments théoriques présentés dans cette thèse, il permet ainsi de traiter des programmes NME et donc de traiter directement des ontologies OWL2 traduites en programmes de règles existentielles et contenant des variables existentielles en tête. De plus, Il permet grâce à une syntaxe plus riche de simplifier l'écriture de certains programmes avec l'utilisation des têtes multi-atomiques et des corps négatifs multi-atomiques ainsi que l'utilisation de requête pour l'interrogation de programmes NME

Une première série de tests techniques a permis de mettre en avant les différentes fonctionnalités d'∃−ASPeRiX et son efficacité sur les programmes NME, en comparant ∃−ASPeRiX avec les solveurs ASP classiques (après traduction des programmes en ASP) nous remarquons qu'il reste moins efficace que les autres à cause du manque d'optimisation du solveur ASPeRiX, mais se distingue sur certains cas précis.

Dans un cas plus concret, avec le benchmark university, nous avons présenté une traduction possible de ce benchmark vers un programme de règles existentielles, celui-ci est donc compatible directement avec ∃−ASPeRiX. Après l'ajout de négation par défaut, nous pouvons constater qu'il est capable de traiter des ontologies avec exceptions de petite taille à partir d'une ontologie en OWL traduite en règles existentielles. Il est néanmoins difficile de traiter de grosses instances avec celui-ci à partir du moment où les règles ne sont plus linéaires (règles avec corps mono atomique). ∃−ASPeRiX reste le seul solveur capable de traiter nativement une ontologie sous la forme d'un programme NME, mais il est possible après traduction en ASP d'utiliser d'autres solveurs pour le calcul d'answer set.

Les autres solveurs ASP sont bien plus performant mais l'optimisation de ∃−ASPeRiX pourrait permettre de resserrer cette différence de performance avec des améliorations notamment sur l'analyse syntaxique, et sur l'ordre des instanciations. Nous pouvons mettre en avant plusieurs avantages

comme la possibilité d'avoir plusieurs atomes dans les corps négatifs et dans la tête des règles, ce qui permet une écriture plus simple pour certains programmes, de plus les têtes multi-atomiques permettent un gain de temps dans certains cas.

L'interrogation est fonctionnelle mais demande au solveur de calculer les answer set entièrement avant d'obtenir une réponse. L'implémentation d'un module d'interrogation reprenant les éléments présentés dans la section 4, avec notamment une marche arrière utilisant les constantes pour instancier les règles dont la requête dépend, peuvent permettre d'obtenir des résultats proches de ceux obtenus par NoHR sur `university`. De même l'implémentation d'un module testant la décidabilité et la consistance d'un programme en prétraitement pourrait permettre d'optimiser l'interrogation.

# Conclusion

Pour raisonner à partir d'ontologies avec exceptions, ou avec des données incomplètes, nous avons défini, dans cette thèse, un langage de règles non-monotones existentielles qui permet de représenter à la fois les programmes provenant des règles existentielles et d'ASP. Les programmes NME autorisent l'utilisation de variables existentielles en tête de règle et de négations par défaut, de plus nous autorisons les têtes multi-atomiques et des ensembles d'atomes liés à une négation par défaut. À partir de cette étude nous avons proposé une traduction de ce langage vers un ASP classique ce qui permet de traiter des logiques de description légères avec exception à l'aide de solveurs ASP.

La définition de programmes NME nous a mené à étudier les cas décidables pour ces programmes. Un programme NME pouvant avoir une infinité d'answer set potentiellement infinis il est important de définir des classes de programmes pour lesquels nous savons qu'il existe des modèles calculables en un temps fini. Nous avons dans un premier temps redéfinis de façon homogène, à l'aide d'un graphe de positions, les classes concrètes de décidabilité traitant de l'acyclicité et appartenant à la classe abstraite FES. Dans un second temps nous avons étendu ces notions pour prendre en compte la dépendance entre règles. Pour finir nous avons considéré la négation par défaut pour affiner encore un peu les classes de décidabilité pour les programmes NME.

Les programmes NME pouvant être traduits en programmes ASP classiques, nous avons traiter l'interrogation en ASP plutôt qu'en programme NME. La réponse à une interrogation étant équivalente à une réécriture près. Pour cela nous avons défini la notion de requête et de réponse en ASP où nous avons considéré les réponses sceptique et crédule proposant respectivement une réponse à une requête correcte pour l'ensemble des answer set et pour au moins un answer set. Le problème principal posé dans cette thèse pour l'interrogation est celui de l'inconsistance. Pour la traiter, il a été proposé la détection de règles dangereuses, représentant les règles qui peuvent provoquer une inconsistance dans un programme. Une fois que nous avons prouvé que le programme n'est pas inconsistant, nous avons isolé à partir d'une requête le plus petit ensemble de règles nécessaire pour y répondre, afin de limiter le nombre de calculs nécessaires pour obtenir une réponse.

La dernière partie de cette thèse a traité de l'extension d'un solveur ASP permettant de traiter ces nouveaux programmes. Cette extension est celle du solveur ASPeRiX développé au LERIA, qui a pour particularité d'instancier les règles à la volée. Nous avons choisi ce solveur pour l'accessibilité du code et notre volonté d'exploiter l'instanciation à la volée pour traiter les variables existentielles en tête de règle plus efficacement. L'extension s'appelle `∃-ASPeRiX`, et propose le calcul d'un modèle pour les programmes non-monotones existentiels, ainsi que l'interrogation sur ceux-ci. Le module d'interrogation ne fait que calculer une réponse à partir des answer set calculés entièrement, les améliorations grâce aux règles dangereuses n'ayant pas encore été implémentées. Nous avons aussi mis en avant les résultats obtenus sur le benchmark `university` permettant de confirmer l'efficacité des solveurs ASP. Malheureusement ASPeRiX se retrouve un peu pénalisé par l'absence d'optimisation dans le code.

## Perspectives

La définition des règles NME ouvre beaucoup de perspectives, la première concerne le domaine de la décidabilité où seule la classe FES a fait l'objet d'une étude pour les programmes NME. Il est ainsi possible de déterminer si un programme va se terminer à l'issue d'un algorithme de marche avant, mais il existe deux autres classes de décidabilité provenant des règles existentielles à étudier. Les classes FUS, et BTS, peuvent ainsi être approfondies pour élargir le nombre de programmes dont nous savons qu'ils sont décidables. La définition de la classe BTS pour les programmes NME permettrait de couvrir l'ensemble des programmes DL-Lite en terme de décidabilité, mais il faut pour cela redéfinir la classe BTS dans le cas d'un programme non-monotone. Il en est de même pour la classe FUS qui nécessite de prendre en compte la négation par défaut pour la réécriture de la requête. Néanmoins après l'étude de l'interrogation et de l'utilisation des règles dangereuses, cela donne quelques indices pour approfondir la classe FUS. Nous savons désormais qu'en ajoutant les règles dangereuses à l'ensemble des règles dont la requête dépend, il est possible de répondre correctement à cette requête sur l'ensemble du programme.

Une autre perspective pourrait être l'amélioration de l'algorithme de marche avant pour les programmes NME. Nous avons mis en avant que l'utilisation de la skolémisation avec l'opérateur de conséquence en ASP correspond à l'utilisation du skolem-chase en règles existentielles. Mais comme nous l'avons fait remarquer, il existe d'autres chase qui calculent un modèle équivalent à un homomorphisme près en détectant les redondances au sein d'un programme. Ainsi pour certains programmes dont le skolem-chase ne s'arrête pas, le restricted-chase ou le core-chase peuvent s'arrêter en détectant une redondance dans le modèle, et ainsi calculer un modèle équivalent. Une étude plus approfondie de l'utilisation du restricted-chase et du core-chase sur les programmes NME permettrait d'élargir le nombre de programmes se terminant en un temps fini. Le problème est de définir une nouvelle sémantique définissant les modèles obtenus après l'utilisation de ces chase sur un programme NME. En effet, si en règles existentielles la sémantique permet d'obtenir des modèles équivalents avec les différents chase ce n'est pas le cas avec un programme NME où les answer set obtenus peuvent contenir des instances n'apparaissant pas avec un autre chase, même à l'aide d'un homomorphisme. Les answer set obtenus avec les programmes NME n'ont pas les mêmes propriétés que les modèles universels, il faut donc redéfinir une sémantique adéquate pour permettre l'utilisation de ces chase. Nous pouvons alors imaginer calculer un answer set universel qui serait défini sur une nouvelle sémantique différente de celles d'ASP et des règles existentielles.

Du point de vue de l'interrogation nous pouvons voir deux perspectives différentes. La première concerne le type de réponse attendu lors de l'interrogation d'un programme. Nous avons pour le moment défini seulement deux types de réponses qui sont la réponse sceptique et la réponse crédule. Il est alors possible d'imaginer d'autres types de réponses, notamment une réponse sous forme de classement où l'on attribuerait un rang pour chaque réponse en fonction du nombre d'answer set dans laquelle elle apparaît et ainsi donner plus d'importance à certaines réponses récurrentes. Ce type de réponse prendrait sens lors de la fusion de plusieurs sources de données, dans ce cas une réponse avec un rang élevé aurait plus de valeur qu'une réponse avec un rang moins élevé. Toujours concernant l'interrogation, il est possible d'optimiser l'interrogation dans un premier temps en effectuant une instanciation intelligente à partir des données de la requête, en instanciant par exemple les règles du programme à partir des constantes présentes dans la requête, en passant par la réécriture de la requête ou alors par la réécriture des règles du programme. Ceci permettrait de ne calculer que les instances nécessaires à la requête. Basé sur le même principe, il serait possible de limiter le nombre d'instances nécessaires sur les règles dangereuses en ne calculant que celles ne pouvant pas être bloquées par une règle dangereuse négative. Cela permettrait de réduire encore plus le nombre d'instanciations nécessaires pour le calcul d'une réponse à une requête en ASP.

La troisième perspective porte sur l'optimisation du solveur ∃−ASPeRiX qui permettrait d'être

en concurrence avec les différents solveurs présentés dans cette thèse. Dans un premier temps l'amélioration servirait à l'amélioration du calcul d'answer set avec une optimisation de la gestion des têtes multi-atomiques, de l'instanciation des règles et de leur application. Ainsi ASPeRiX pourrait se trouver une place dans les compétitions concernant à la fois ASP et les logiques de description. Dans un second temps, il serait intéressant d'ajouter les résultats théoriques provenant de cette thèse à propos de la décidabilité et de l'interrogation. Il serait ainsi possible de rajouter un test de décidabilité et de consistance (optionnel) avec l'utilisation des règles dangereuses et les graphes de positions. Ces algorithmes pouvant être utilisés en pré-traitement permettraient d'optimiser les autres traitements sans ralentir le programme principal. Nous pourrions améliorer le module d'interrogation qui n'est pour le moment que dans une version naïve. Les optimisations sur l'interrogation telles que l'isolation d'un ensemble de règles minimal pour répondre à une requête et une instanciation intelligente utilisant les constantes présentes dans la requête et les règles dangereuses négatives. Certains traitements pourraient aussi être parallélisés (l'algorithme actuel ne tirant la puissance que d'un seul cœur d'un processeur).

Enfin, des tests sur des applications réelles ont besoin d'être entrepris, notamment sur le benchmark ARPENTEUR proposé dans le cadre du projet ANR ASPIQ. Ce benchmark propose d'interroger des données extraites lors de la fouille de sites archéologiques sous-marins. Ces informations représentent des données issues de la photogrammétrie du site de fouille permettant d'aider les archéologues à étudier les sites sous-marins. À l'aide de plusieurs photographies numériques en 2D, une représentation 3D des fonds marins est calculée puis convertie sous forme d'ontologie. Ces données représentent l'ensemble des données concernant des amphores situées en profondeur et que l'on souhaiterait interroger, par exemple pour connaître le nombre d'amphores respectant une même propriété. Un point important serait l'ajout d'exceptions dans les données ontologiques. Cette application sera par ailleurs réalisée par la suite dans le cadre du projet ASPIQ, les données pour ce benchmark n'étant pas disponible lors de l'écriture de cette thèse, nous n'avons pas pu effectuer de tests sur celui-ci pour le moment.

Il existe pour le moment très peu d'ontologies avec exception, la proposition du formalisme du langage NME peut permettre la réalisation de nouveaux benchmarks intégrant à la fois variables existentielles en tête et négation par défaut, comme proposé pour le benchmark `university`. La majorité des exemples proposés comportent une négation par défaut donnant lieu à un programme stratifié, et donc ne comportant qu'un seul answer set. Par la suite la proposition d'un benchmark cohérent, voire issu d'un cas réel comme ARPENTEUR, comportant plusieurs answer set permettrait d'utiliser l'ensemble du potentiel des programmes NME.

# Table des matières

# Liste des tableaux

# Table des figures

# Publications personnelles

## Articles de conférences internationales avec comité de lecture

**Extending acyclicity notions for existential rules.** avec J.-F. Baget, M.-L. Mugnier et S. Rocher. In Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014), 18-22 August 2014, Prague, Czech Republic, pages 39–44, 2014.

**Revisiting chase termination for existential rules and their extension to nonmonotonic negation.** avec J.-F. Baget, M.-L. Mugnier et S. Rocher. In Proceedings of the 15th International Workshop on Non-Monotonic Reasoning (NMR 2014), volume INFSYS Research Report Series, Vienna, Austria, July 2014.

## Articles de workshops internationaux avec comité de lecture

**∃-ASP.** avec L. Garcia, C. Lefèvre, et I. Stéphan. In Proceedings of the 1st Ontologies and Logic Programming for Query Answering workshop (ONTOLP 2015) workshop of the 24th International Joint Conference on Artificial Intelligence (IJCAI), Buenos Aires, July 2015.

## Articles de revues

**Bringing existential variables in answer set programming and bringing non-monotony in existential rules : two sides of the same coin.** avec J.-F. Baget, L. Garcia, C. Lefèvre, S. Rocher, et I. Stéphan. En cours de soumission dans Annals of Mathematics and Artificial Intelligence 2016.

# Bibliographie


[1] Benchmark university. http://swat.cse.lehigh.edu/projects/lubm. Dernière visite : 2016-04-04. 8, 11

[2] Projet ANR ASPIQ référence ANR-12-BS02-0003. http://aspiq.lsis.org/. 6

[3] DLV - manuel d'utilisation. http://www.dlvsystem.com/html/DLV_User_Manual.html. 89

[4] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases : The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995. 25

[5] M. Alviano, F. Calimeri, W. Faber, G. Ianni, and N. Leone. Function symbols in asp : Overview and perspectives. In *In Non-Monotonic Reasoning (NMR 2011) – Essays Celebrating Its 30th Anniversary*, pages 1–24. College Publications, 2011. 17

[6] M. Alviano and W. Faber. Dynamic magic sets for super-consistent answer set programs. *Computing Research Repository*, abs/1011.4377, 2010. 8, 84, 89

[7] M. Alviano, W. Faber, and S. Woltran. Complexity of super-coherence problems in asp. *Theory and Practice of Logic Programming*, 14 :339–361, 5 2014. 84, 100

[8] M. Alviano, N. Leone, M. Manna, G. Terracina, and P. Veltri. *Proceedings of the Second International Workshop on Datalog in Academia and Industry, Datalog 2.0, Vienna, Austria, September 11-13, 2012.*, chapter Magic-Sets for Datalog with Existential Quantifiers, pages 31–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 84, 91, 117

[9] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The dl-lite family and relations. *Computting Research Repository*, abs/1401.3487, 2014. 25

[10] F. Baader, S. Brandt, and C. Lutz. Pushing the el envelope. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, IJCAI'05, pages 364–369, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. 5

[11] J.-F. Baget. Improving the forward chaining algorithm for conceptual graphs rules. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), Whistler, Canada, June 2-5, 2004*, pages 407–414, 2004. 61

[12] J.-F. Baget. Ontologies and large databases : querying algorithms for the Web of Data. Invited Talk, Artificial Intelligence meets the Web of Data, ESWC'13 Workshop, 2013. 33

[13] J.-F. Baget, F. Garreau, M.-L. Mugnier, and S. Rocher. Extending acyclicity notions for existential rules. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014), 18-22 August 2014, Prague, Czech Republic*, pages 39–44, 2014. 63, 71, 75

[14] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. Extending Decidable Cases for Rules with Existential Variables. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 677–682, Pasadena, CA, United States, July 2009. AAAI. 5, 25, 35, 73

[15] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On rules with existential variables : Walking the decidability line. *Artificial Intelligence*, 175(9-10) :1620–1654, 2011. 5, 25, 34, 74, 75, 83

[16] J.-F. Baget, F. Garreau M.-L. Mugnier, and S. Rocher. Revisiting chase termination for existential rules and their extension to nonmonotonic negation. In Sébastien Konieczny and Hans Tompits, editors, *Proceedings of the 15th International Workshop on Non-Monotonic Reasoning (NMR 2014)*, volume INFSYS Research Report Series, Vienna, Austria, July 2014. 73, 76

[17] J.-F. Baget and M.-L. Mugnier. The Complexity of Rules and Constraints. *Journal of Artificial Intelligence Research (JAIR)*, 16 :425–465, 2002. 28, 54

[18] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM. 8, 84, 90

[19] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003. 5, 16

[20] C. Beeri and M. Y. Vardi. The implication problem for data dependencies. In *Proceedings of the 8th Colloquium Automata, Languages and Programming, Acre (Akko), Israel, July 13-17, 1981,*, pages 73–85, 1981. 53

[21] M. Cadoli and M. Schaerf. Special issue : Non-monotonic reasoning and logic programming a survey of complexity results for non-monotonic logics. *The Journal of Logic Programming*, 17(2) :127 – 160, 1993. 53

[22] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase : Query answering under expressive relational constraints. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 70–80, 2008. 25

[23] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2009)*, pages 77–86, 2009. 25

[24] A. Calì, G. Gottlob, and T. Lukasiewicz. Tractable query answering over ontologies with datalog+/-. In *Proceedings of the 22nd International Workshop on Description Logics (DL 2009)*, 2009. 6

[25] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP : theory and implementation. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008), Udine, Italy, December 9-13 2008*, pages 407–424, 2008. 56

[26] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. Asp-core-2, input language format. https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf. version 2.03b. 109

[27] D. Calvanese, G. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics : The dl-lite family. *Journal of Automated Reasoning*, 39(3) :385–429, 2007. 5

[28] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Dl-lite : Tractable description logics for ontologies. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005)*, pages 602–607, 2005. 15

[29] A. K. Chandra, H. R. Lewis, and J. A. Makowsky. Embedded implicational dependencies and their inference problem. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 342–354, 1981. 53

[30] Nuno Costa, Matthias Knorr, and João Leite. *Next Step for NoHR : OWL 2 QL*, pages 569–586. Springer International Publishing, Cham, 2015. 122, 128, 130

[31] J. de Bruijn, D. Pearce, A. Polleres, and A. Valverde. A semantical framework for hybrid knowledge bases. *Knowledge and Information System*, 25(1) :81–104, 2010. 6

[32] J. Delgrande and W. Faber, editors. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2011. 108, 117, 147

[33] A. Deutsch, A. Nash, and J.B. Remmel. The chase revisited. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2008)*, pages 149–158, 2008. 7, 32

[34] P.M. Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105(1) :7 – 25, 1992. 90

[35] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13) :1495–1539, 2008. 6

[36] W. Faber, G. Greco, and N. Leone. Magic sets and their application to data integration. *Journal of Computer and System Sciences*, 73(4) :584 – 609, 2007. Special Issue : Database Theory 2005. 8, 84, 90

[37] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange : Semantics and query answering. In *,Proceedings of the 9th International Conference on Database Theory (ICDT 2003), Siena, Italy, January 8-10, 2003*, pages 207–224, 2003. 55

[38] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange : semantics and query answering. *Theoritical Computer Science*, 336(1) :89–124, 2005. 7, 31, 55

[39] P. Ferraris, J. Lee, and V. Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175 :236–263, 2011. 6

[40] F. Garreau, L. Garcia, C. Lefèvre, and I. Stéphan. ∃-asp. In *Proceedings of the 1st Ontologies and Logic Programming for Query Answering workshop (ONTOLP'15), Buenos Aires, July 2015*, 2015.

[41] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In Delgrande and Faber [32], pages 345–351. 108, 117

[42] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving : From theory to practice. *Artificial Intelligence*, 187-188 :52–89, 2012. 108, 117

[43] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP 1988)*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press. 5, 16, 19

[44] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4) :365–386, 1991. 6

[45] G. Gottlob, A. Hernich, C. Kupke, and T. Lukasiewicz. Equality-friendly well-founded semantics and applications to description logics. In J. Hoffmann and B. Selman, editors, *Proceedings of the 26th National Conference on Artificial Intelligence (AAAI 2012), Toronto, Ontario, Canada, July 2012*. AAAI Press, 2012. 6

[46] B. Cuenca Grau, I. Horrocks, M. Krötzsch, C. Kupke, D. Magka, B. Motik, and Z. Wang. Acyclicity conditions and their application to query answering in description logics. In *Proceedings of the 13th International Conference Principles of Knowledge Representation and Reasoning (KR 2012), Rome, Italy, June 10-14, 2012*, 2012. 54

[47] Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *Journal of Artificial Intelligence Research (JAIR)*, 47 :741–808, 2013. 54, 61, 62, 63

[48] P. Hell and J. Nesetril. The core of a graph. *Discrete Mathematics*, 109(1) :117 – 126, 1992. 32

[49] G. Ianni, T. Eiter, H. Tompits, and R. Schindlauer. Nlp-dl : A kr system for coupling nonmonotonic logic programs with description logics. In *Proceedings of the 4th International Semantic Web Conference (ISWC 2005)*, 2005. 6

[50] M. Krötzsch and S. Rudolph. Extending decidable existential rules by joining acyclicity and guardedness. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 963–968, 2011. 59

[51] J. Lee and R. Palla. Integrating rules and ontologies in the first-order stable model semantics (preliminary report). In *Proceedings of the 11th International Conference in Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, Canada, May 16-19, 2011.*, pages 248–253, 2011. 6

[52] C. Lefèvre, C. Béatrix, I. Stéphan, and L. Garcia. Asperix, a first order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming (TPLP 2015)*, page to appear, 2015. 108

[53] C. Lefèvre, C. Béatrix, I. Stéphan, and L. Garcia. Asperix, a first order forward chaining approach for answer set computing. *Computing Research Repository*, abs/1503.07717 :(to appear in TPLP), 2015. 108

[54] C. Lefèvre and P. Nicolas. A first order forward chaining approach for answer set computing. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *LNCS*, pages 196–208. Springer, 2009. 8

[55] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3) :499–562, July 2006. 108, 117

[56] Y. Lierler and V. Lifschitz. One more decidable class of finitely ground programs. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009), Pasadena, CA, USA, July 14-17, 2009*, pages 489–493, 2009. 57

[57] L. Liu, E. Pontelli, T. C. Son, and M. Truszczynski. Logic programs with abstract constraint atoms : The role of computations. *Artificial Intelligence*, 174(3-4) :295–315, 2010. 108

[58] D. Magka, M. Krötzsch, and I. Horrocks. Computing stable models for nonmonotonic existential rules. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013), Beijing, China, August 3-9, 2013*, 2013. 6, 77

[59] B. Marnette. Generalized schema-mappings : From termination to tractability. In *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2009)*, PODS '09, pages 13–22, New York, NY, USA, 2009. ACM. 7, 60

[60] B. Motik and R. Rosati. Reconciling description logics and rules. *ACM*, 57(5), 2010. 6

[61] M.-L. Mugnier. Ontological query answering with existential rules. In *Proceedings of the 5th International Conference on Web Reasoning and Rule Systems (RR 2011), Galway, Ireland, August 29-30, 2011. Proceedings*, pages 2–23, 2011. 7

[62] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4) :241–273, 1999. 5, 16

[63] H. Pérez-urbina, B. Motik, and I. Horrocks. A comparison of query rewriting techniques for dl-lite. In *Proceedings of DL 2009*, 2009. 130

[64] R. Rosati. Dl+log : Tight integration of description logics and disjunctive datalog. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), Lake District of the United Kingdom, June 2-5, 2006*, pages 68–78, 2006. 6

[65] M. Salvat and M-L. Mugnier. Sound and complete forward and backward chainings of graph rules. pages 248–262. 1996. 25, 78

[66] T. Schaub and M. Thielscher. *Proceedings of the International Conference on Formal and Applied Practical Reasoning (FAPR'96), Bonn, Germany, June 3–7, 1996*, chapter Skeptical query-answering in Constrained Default Logic, pages 567–581. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. 89

[67] P. Simons, I. Niemelä, and T. Soininen. Knowledge representation and logic programming extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1) :181 – 234, 2002. 108

[68] T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics, 1998. 108

[69] T. Syrjänen. *Proceedings of the 6th International Conference on Logic Programming and Nonmotonic Reasoning (LPNMR 2001), Vienna, Austria, September 17–19, 2001*, chapter Omega-Restricted Logic Programs, pages 267–280. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 53

[70] M. Thomazo. Compact Rewriting for Existential Rules. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'2013)*, page N/A, Beijing, China, 2013. 78

[71] M. Thomazo, J.-F. Baget, M.-L. Mugnier, and S. Rudolph. A Generic Querying Algorithm for Greedy Sets of Existential Rules. In *KR : Principles of Knowledge Representation and Reasoning*, pages 096–106, Rome, Italy, June 2012. 80

[72] M. Y. Vardi. The implication and finite implication problems for typed template dependencies. *Journal of Computer and System Sciences*, 28(1) :3 – 28, 1984. 28

UNIVERSITE BRETAGNE LOIRE

université angers

# Thèse de Doctorat

Fabien GARREAU

## Extension d'ASP pour couvrir des fragments DL traitables : étude théorique et implémentation

## Extension of ASP to cover treatable DL fragments : theorical study and implementation

**Résumé**
Les ontologies sont utilisées pour la représentation et l'interrogation de connaissances d'un domaine précis et peuvent être représentées en partie à l'aide des logiques de description légères. Ces ontologies peuvent être issues de plusieurs sources dont les données sont plus ou moins complètes, ainsi certaines données peuvent être incomplètes ou incohérentes empêchant la déduction d'autres données. L'Answer Set Programming (ASP) est un langage de programmation logique non-monotone à base de règles permettant de représenter des données incomplètes mais il ne permet pas de représenter les logiques de description légères. Les règles existentielles généralisent les logiques de description légères et forment aussi un langage de programmation logique mais ne permettant pas la définition d'exceptions. À partir d'une étude théorique d'ASP et des règles existentielles nous proposons de regrouper en un seul formalisme ces deux langages, nous définissons le formalisme des programmes non-monotones existentiels permettant de traiter un programme provenant d'une ontologie avec exceptions. Cette extension a pour but de généraliser à la fois ASP et les règles existentielles et d'utiliser la puissance des solveurs ASP pour raisonner sur des ontologies avec exceptions. Cette étude propose d'approfondir les travaux sur la décidabilité d'un programme avec l'extension aux programmes non-monotones existentiels. Nous proposons aussi d'améliorer les résultats liés à l'interrogation d'un programme ASP ainsi qu'une implémentation d'une extension du solveur ASPeRiX pour traiter les programmes non-monotones existentiels.

**Mots clés**
Answer Set Programming, Règles existentielles, Interrogation, Décidabilité, Inconsistance.

**Abstract**
Ontologies are meant to represent or to query knowledge from a precise domain and can be represented, in part, by logic formalisms such that description logics. These ontologies can be provided by several sources where knowledge is more or less complete, hence some data can be incomplete or incoherent preventing the deduction of other data. Answer Set Programming (ASP) formalism is a non-monotonic logic programming language based on rules, often used in knowledge representation, which has the feature to represent incomplete data. However, it's impossible to represent lite description logics in ASP, because of existential variables in rules. Existential rules generalize lite description logics and also form a programmation logic language that but doesn't offer the possibility to represent exceptions. Based on a theoritical study of ASP and existential rules, we propose to gather both languages in a unique formalism, we define non-monotonic existential program allowing to deal with ontology with exceptions. This extension aims to generalize both ASP and existential rules program and to use the efficiency of ASP solvers to reason on ontologies with exceptions. This thesis propose to deepen works about entailment and decidability of a non-monotonic existential program. Another result from this study is the improvement of interrogation in ASP and the implementation of an extension of the ASPeRiX solver to deal with non-monotonic existential programs.

**Key Words**
Answer Set Programming, Existential rules, Query Answering, Decidability, Inconsistency.