Interrogation de grandes bases de connaissances : algorithmes de réécriture de requêtes conjonctives en présence de règles existentielles

Mélanie König

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'**Université Montpellier II**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **UMR 5506**

Spécialité: **Informatique**

Présentée par **Mélanie König**

Interrogation de grandes bases de connaissances

algorithmes de réécriture de requêtes conjonctives en présence de règles existentielles

Soutenue le 24 octobre 2014 devant le jury composé de :

Encadrants		
Mme. Marie-Laure Mugnier	Professeur	Univ. Montpellier II
M. Michel Leclère	Maître de conférence	Univ. Montpellier II
Rapporteurs		
M. Bernardo Cuenca Grau	Tenure Associate Professor	Univ. Oxford
M. Igor Stephan	Maître de conférence HDR	Univ. Angers
Examinateurs		
Mme. Marianne Huchard	Professeur	Univ. Montpellier II
M. Lhouari Nourine	Professeur	Univ. Blaise Pascal
Invités		
Mme. Meghyn Bienvenu	Chargé de recherche CNRS	Univ. Paris Sud
M. Remi Coletta	Maître de conférence	Univ. Montpellier II

UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

Sommaire

Chapitre 1

Introduction

Dans ce manuscrit, nous abordons une problématique qui suscite beaucoup d'intérêt actuellement dans les domaines de la représentation de connaissances, des bases de données, et du web sémantique, celle de l'interrogation de données en présence d'une ontologie (que nous noterons OBQA pour "ontology-based query answering"). Cette problématique est aussi appelée interrogation de bases de connaissances, une base de connaissances étant alors composée de données, ou faits, et d'une ontologie, qui exprime typiquement des connaissances générales sur le domaine d'application. L'objectif est d'exploiter les connaissances exprimées par l'ontologie lors de l'interrogation des données, de façon à obtenir des réponses plus riches. De façon générale, l'ontologie permet d'inférer de nouveaux faits qui ne sont pas explicitement stockés dans la base de données. Considérer une "couche" ontologique au-dessus des données permet également d'enrichir le vocabulaire utilisé dans les requêtes, et ainsi de faire abstraction de la façon dont les données sont effectivement stockées. Lorsque plusieurs sources de données utilisent des vocabulaires différents, ceci permet également de fournir une interface unifiée d'accès aux données.

L'exploitation de connaissances ontologiques de manière automatique est un défi qui intéresse autant le monde industriel que le monde académique. Elle nécessite de pouvoir représenter les connaissances de manière formelle. La famille de formalismes la plus utilisée pour représenter des ontologies et effectuer les raisonnements associés est celle des logiques de description [Baader et al., 2003]. Les logiques de description (DLs), qui ont fait leur apparition dans les années 80, peuvent être vues comme des fragments décidables de la logique du premier ordre. Dans ces langages, la représentation des connaissances du domaine s'appuie sur des concepts, qui correspondent à des prédicats unaires en logique du premier ordre, et des rôles, qui correspondent à des prédicats binaires. Des constructeurs, qui diffèrent selon la logique de description considérée, permettent de construire des concepts ou des rôles plus complexes à partir d'autres. Les relations entre concepts et rôles sont exprimées par des axiomes d' inclusion de concepts ou de rôles. Historiquement, les problèmes considérés dans le cadre des logiques de description correspondaient à des raisonnements sur les ontologies, tels que la vérification de subsomption entre concepts, la détermination de

l'appartenance d'une instance à un concept, ou encore la vérification de la satisfiabilité d'une base de connaissances. Les logiques de description couramment utilisées avaient une expressivité élevée, et le problème OBQA s'est avéré très complexe dans ces logiques.

Dans le domaine des logiques de description, et plus généralement en représentation de connaissances, de nombreux travaux visent à trouver un compromis satisfaisant entre l'expressivité du formalisme considéré et la complexité des problèmes de raisonnement associés. L'intérêt pour le problème OBQA a ainsi motivé l'étude de logiques de description dites légères, comme la famille DL-Lite qui a été proposée spécifiquement dans le cadre OBQA [Calvanese et al., 2005], ou la famille $\mathcal{EL}$ qui, si elle a été introduite pour raisonner sur de grosses ontologies [Baader, 2003] , s'avère bien adaptée à OBQA. Dans le cadre du web sémantique, l'exploitation automatique d'ontologies a été rendue possible grâce à l'élaboration de standards du W3C (World Wide Web Consortium) tels que RDFS, OWL et OWL2. Le langage OWL2 comporte en particulier des profils dits "traitables", qui s'appuient sur les familles de logiques de description légères $\mathcal{EL}$ et DL-Lite (ainsi que sur le langage de règles Datalog pour l'un des profils citeabiteboul95).

Dans ce manuscrit, nous représentons les ontologies dans le formalisme des règles existentielles [Baget et al., 2011a, Krötzsch and Rudolph, 2011], aussi connu sous le nom de Datalog± [Calì et al., 2008, Calì et al., 2009]. Les règles existentielles ont la même forme que les "tuple-generating dependencies", des dépendances très expressives étudiées de longue date en bases de données [Abiteboul et al., 1995]. Elles correspondent aussi à la traduction logique des règles de graphes conceptuels [Salvat and Mugnier, 1996, Chein and Mugnier, 2009]. Les règles existentielles sont de la forme $H \rightarrow C$, où H, appelé hypothèse ou prémisse de la règle, et C, appelée conclusion de la règle, sont des conjonctions d'atomes (sans fonction en dehors des constantes). Leur spécificité réside dans le fait que les variables qui apparaissent uniquement dans C sont quantifiées existentiellement, ce qui permet de représenter des entités non identifiées, potentiellement non présentes dans la base initiale. Cette capacité est reconnue comme cruciale pour la représentation de connaissances dans le cadre d'un monde ouvert ("open world") où l'on ne peut considérer que les données sont complètes. Les règles existentielles généralisent les logiques de description légères, en permettant la description de structures cycliques, et pas seulement arborescentes, ainsi qu'en autorisant une arité quelconque des prédicats. Le fait de ne pas borner l'arité des prédicats permet notamment une association naturelle entre les prédicats du vocabulaire et un schéma de base de données relationnelle.

Exemple 1 (Règles existentielles) *Considérons un vocabulaire permettant de décrire le monde du cinéma, avec notamment les concepts (prédicats unaires)* film *et* acteur*, et le rôle (prédicat binaire)* joue*; intuitivement,* joue(x, y) *signifie que "x joue dans y"). L'ontologie bâtie sur ce vocabulaire comporte la connaissance que "tout acteur joue dans un film", ce que l'on exprime par la règle existentielle suivante :*

$$R = \forall x(acteur(x) \rightarrow \exists y(joue(x, y) \wedge film(y)))$$

La variable y qui n'apparaît pas dans l'hypothèse de R est quantifiée existentiellement. La règle R permet de dire qu'il existe un film dans lequel x joue, sans que l'on sache quel est ce film.

Comme l'ontologie, les données peuvent être représentées dans différents modèles de représentation, tels que les bases de données relationnelles, les bases de graphes, de triplets RDFS, etc. Pour faire abstraction de ces représentations particulières, nous considérons les données, ou faits, comme des formules (existentielles, positives et conjonctives) de la logique du premier ordre. Enfin, en ce qui concerne les requêtes, nous étudions les requêtes conjonctives, qui sont considérées comme les requêtes fondamentales par la communauté des bases de données, car elles sont à la fois évaluables efficacement et fréquemment utilisées. Dans ce document, pour simplifier les notions de base, nous restreignons les requêtes aux requêtes conjonctives booléennes, que l'on peut voir comme des formules positives et conjonctives existentiellement closes. Toutefois, comme nous l'expliquerons, tous les résultats peuvent être facilement étendus aux requêtes conjonctives non booléennes, ainsi qu'aux unions de requêtes conjonctives.

Le problème que nous abordons peut donc être reformulé de la manière suivante : étant donné une base de connaissances K, composée de faits et de règles existentielles, et une requête booléenne conjonctive Q, est-ce que la réponse à Q dans K est positive, c'est-à-dire Q est-elle logiquement impliquée par K ? La présence de variables existentielles en conclusion de règle, associée à des ensembles d'atomes quelconques dans l'hypothèse de la règle, rend ce problème indécidable avec des règles existentielles quelconques (voir notamment [Beeri and Vardi, 1981, Chandra et al., 1981] sur les tuple-generating dependencies). Pour cette raison, de nombreux travaux de recherche se sont attachés à trouver des sous-ensembles de règles qui rendent le problème décidable, avec un bon compromis entre expressivité et complexité du problème.

Exemple 2 (Problème OBQA) *Reprenons l'exemple du monde du cinéma. Considérons la requête booléenne conjonctive* $Q = \exists y\ joue(b, y)$*, où b est une constante. Q demande si b joue quelque part. Supposons que la base de connaissances K contienne le fait* $F = \text{acteur}(b)$*, ainsi que la règle R. Q est impliquée logiquement par K. La réponse à Q est donc "oui" sur cette base de connaissances.*

Il existe deux approches principales pour aborder le problème OBQA, qui sont liées aux deux paradigmes de traitement des règles, en marche avant ou en marche arrière. Ces deux approches peuvent être vues comme une façon de réduire le problème OBQA à un problème d'interrogation classique de bases de données, en éliminant les règles. La marche avant consiste à appliquer les règles sur les faits pour les enrichir dans le but d'ajouter toutes les informations qui sont logiquement impliquées par la base de connaissances initiale. La requête est ensuite évaluée sur la base de faits enrichie. La marche arrière procède de manière "inverse" : elle utilise les règles pour réécrire la requête "de toutes les façons possibles". La requête réécrite est ensuite

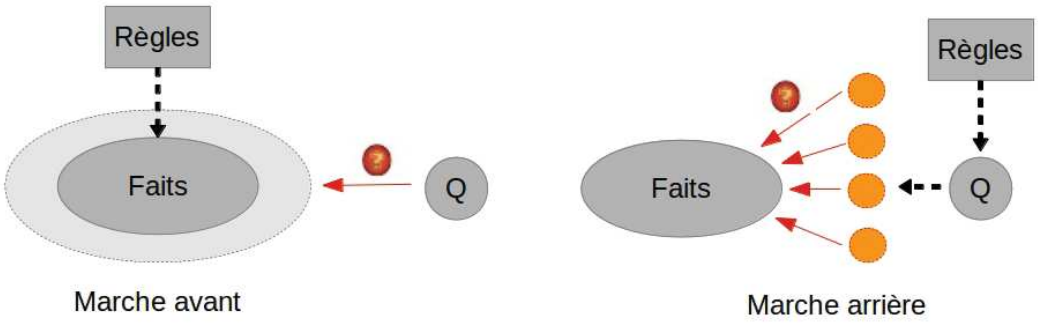

FIGURE 1.1 – Marche avant - Marche arrière

évaluée sur la base de faits initiale. Ces deux approches sont illustrées dans la figure 1.1. Le traitement des règles en marche arrière présente un avantage évident lorsque la taille des faits est trop importante pour que la matérialisation de toutes les informations inférables de la base de connaissances soit raisonnable. De plus, elle ne nécessite aucun accès en écriture aux données, cet accès pouvant être interdit, ou difficile si la base de données est distribuée. Comme elle ne modifie pas les données, ne se pose pas non plus le problème de mettre à jour les inférences calculées lorsque les données évoluent. Néanmoins, la taille de la réécriture peut être importante, parfois exponentielle en la taille de la requête initiale, et donc peu efficacement évaluable sur les faits, au moins avec les techniques actuelles de bases de données. Enfin, des techniques qui combinent les deux approches ont été développées, en particulier la technique appelée "approche combinée"[Lutz et al., 2009, Kontchakov et al., 2011].

Exemple 3 (Deux approches pour résoudre OBQA) *Reprenons notre exemple. Si Q est posée sur la base de faits réduite à F, la réponse à Q est non. En effet, il est nécessaire de prendre en compte la règle R pour répondre positivement à Q. En marche avant, R est appliquée à F : la connaissance $\exists y_0(joue(B, y_0) \wedge film(y_0))$ est ajoutée à la base de faits, où y_0 est une nouvelle variable. La base de faits enrichie permet de répondre positivement à Q. La règle R peut aussi être utilisée en marche arrière pour réécrire Q (intuitivement : "puisque tout acteur joue dans un film, si on trouve que B est un acteur, on répond à Q"). La requête initiale Q est ainsi réécrite en $Q \vee Q'$, où $Q' = \text{acteur}(B)$. Q' ayant une réponse positive sur la base de faits initiale, la réponse à la requête initiale est positive.*

Organisation du mémoire

Dans ce document, nous nous concentrons sur les techniques de réécritures et plus spécifiquement sur les techniques de réécritures d'une requête conjonctive initiale Q en une union de requêtes conjonctives (UCQ) ? Une UCQ peut aussi être vue

comme un ensemble de requêtes conjonctives, que nous appellerons réécritures de Q. Notre but est de calculer un ensemble de réécritures qui soit à la fois adéquat (si l'une des réécritures s'envoie sur les faits alors Q est impliqué par la base de connaissances) et complet (si Q est impliqué par la base de connaissances alors l'une des réécritures s'envoie sur les faits). Une autre propriété souhaitable de l'ensemble de réécritures est sa minimalité : plus l'ensemble est petit, plus son évaluation sera rapide. Étant donné que le problème n'est pas décidable avec un ensemble de règles existentielles quelconques, il n'existe pas forcément d'ensemble de réécritures *fini* ayant les propriétés d'adéquation et de complétude. Un ensemble de règles existentielles qui assure l'existence d'un tel ensemble de réécritures pour n'importe quelle requête est appelé ensemble à unification finie (finite unification set, *fus*) [Baget et al., 2011a].

Après l'introduction dans le chapitre 2, des notions de base nécessaires à la compréhension de notre travail nous proposons dans le chapitre 3 un cadre théorique permettant l'étude des techniques de réécritures. Nous définissons d'abord les propriétés souhaitables d'un ensemble de réécritures (adéquation, complétude, minimalité). Puis nous étudions un algorithme générique qui, étant donné une requête et un ensemble de règles, calcule un ensemble de réécritures. Cet algorithme est paramétré par un *opérateur de réécriture*, c'est-à-dire une fonction qui, étant donné une requête et un ensemble de règles existentielles, retourne les réécritures "directes" de cette requête par cet ensemble de règles. L'algorithme effectue une exploration en largeur de l'espace des réécritures. A chaque étape, l'algorithme calcule l'ensemble des réécritures directes des requêtes obtenues à l'étape précédente et conservées (toutes les requêtes n'étant pas nécessairement conservées, pour des raisons que nous détaillerons au chapitre 3). Nous définissons des propriétés d'un opérateur de réécriture qui assurent que l'ensemble de réécritures calculé par l'algorithme est adéquat, complet, et minimal si l'ensemble de règles existentielles est *fus*.

Dans le chapitre 4, nous présentons ensuite une famille d'opérateurs de réécritures. Ces opérateurs de réécritures s'inspirent de travaux précédents sur les graphes conceptuels [Salvat and Mugnier, 1996] et s'appuient sur la notion d'*unificateur par pièce*. Comme pour la marche arrière classique, nos techniques s'appuient sur une opération d'unification entre la requête et la conclusion de la règle. Nous nous arrêtons d'abord sur le problème que posent les variables existentielles dans les conclusions de règles lors de cette phase d'unification, ce qui justifie la notion d'unificateur par pièce qui remplace l'unification usuelle. Puis, nous étudions plusieurs opérateurs de réécriture basés sur l'unification par pièce à la lumière des propriétés définies dans le chapitre 3.

Dans le chapitre 5, nous proposons deux optimisations qui nous permettent de traiter d'une manière plus efficace des règles simples mais très présentes dans les ontologies réelles. Ces optimisations ouvrent sur d'autres formes de réécritures que les UCQs, ce qui pose la question de la façon de les évaluer sur des données.

Nous détaillons l'implémentation de nos algorithmes dans le chapitre 6, en commençant par la présentation de l'API sur laquelle s'appuie notre implémentation, et de la hiérarchie de classes mise en place. Nous zoomons ensuite sur les algorithmes de

calcul des unificateurs par pièce et finissons par les structures de données utilisées dans le cadre de nos deux optimisations. Nous avons réalisé une évaluation pratique de nos algorithmes sur des benchmarks existants, présentée dans le chapitre 7. Notre expérimentation comporte une comparaison interne de nos différents opérateurs, puis une comparaison externe avec d'autres systèmes existants. Finalement, nous concluons sur nos travaux et esquissons des perspectives dans le chapitre 8.

Publications associées

Les travaux présentés dans ce mémoire ont fait l'objet des publications suivantes :

Mélanie König, Michel Leclère, Marie-Laure Mugnier, Michaël Thomazo : Sound, Complete, and Minimal UCQ-Rewriting for Existential Rules. Semantic Web Journal (à paraître, `http://www.semantic-web-journal.net/content/sound-complete-and-minimal-ucq-rewriting-existential-rules-0`)

Mélanie König, Michel Leclère, Marie-Laure Mugnier, Michaël Thomazo : Sound, Complete, and Minimal Query Rewriting for Existential Rules. IJCAI 2013

Mélanie König, Michel Leclère, Marie-Laure Mugnier, Michaël Thomazo : On the Exploration of the Query Rewriting Space with Existential Rules. 7th International Conference on Web Reasoning and Rule Systems (RR 2013) : 123-137

Mélanie König, Michel Leclère, Marie-Laure Mugnier, Michaël Thomazo : A Sound and Complete Backward Chaining Algorithm for Existential Rules. 6th International Conference on Web Reasoning and Rule Systems (RR 2012) : 122-138 - Best paper award

Les résultats relatifs aux optimisations introduites dans le chapitre 5 n'ont pas encore été publiés.

Chapitre 2

Notions de base

Tout au long de ce manuscrit, nous nous intéressons au problème de l'interrogation d'une base de connaissances (OBQA). Une base de connaissances $\mathcal{K} = (\mathcal{F}, \mathcal{O})$ est composée d'un ensemble de faits $\mathcal{F}$ et d'une ontologie $\mathcal{O}$. La problématique générale est de prendre en compte des connaissances générales sur le domaine, exprimées dans l'ontologie, lors de l'interrogation des faits. Comme nous l'avons expliqué dans l'introduction, nous utiliserons la logique du premier ordre pour exprimer les différents composants du problème. Les notations logiques utilisées seront présentées dans la première section. Dans les deux suivantes, nous formaliserons les trois composants du problème : requête, faits puis ontologie avec laquelle nous présenterons plusieurs langages ontologiques. Nous commencerons par présenter deux familles de logiques de description $\mathcal{EL}$ et DL-Lite, puis nous continuerons avec les règles existentielles et la traduction de $\mathcal{EL}$ et DL-Lite en règles existentielles. Pour finir ce chapitre, nous parlerons de l'indécidabilité du problème étudié et de ses sous-cas connus décidables.

2.1 Bases de la logique du premier ordre

Nous avons choisi d'utiliser le formalisme de la logique du premier ordre pour exprimer les composants de notre problème, notamment les faits et la requête. Ce choix est motivé d'une part par l'utilisation des règles existentielles qui utilisent aussi ce formalisme et d'autre part pour faire abstraction de la technologie de stockage des faits et du langage de requêtes associé. La lecture de ce manuscrit nécessite des connaissances classiques sur la logique du premier ordre, que nous rappelons dans cette section.

Définition 2.1 (Langage du premier ordre) *Un* langage du premier ordre $\mathcal{L} = (\mathcal{P}, \mathcal{C})$ *est composé de deux ensembles disjoints :* $\mathcal{P}$ *est fini et contient des prédicats,* $\mathcal{C}$ *peut être infini et contient des constantes. Chaque prédicat de* $\mathcal{P}$ *est associé à un entier positif ou nul qui fixe son* arité.

Du point de vue des langages formels, les formules sont des mots construits sur l'alphabet formé :

- d'un langage du premier ordre $\mathcal{L} = (\mathcal{P}, \mathcal{C})$,
- des connecteurs $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$,
- des parenthèses $(,)$,
- des quantificateurs universels $\forall$ et existentiel $\exists$,
- d'un ensemble infini de symboles de variables noté $\mathcal{V}$ (disjoint de $\mathcal{P}$ et $\mathcal{C}$).

Le langage du premier ordre $\mathcal{L}$ fait donc parti de l'alphabet toutefois l'expression "langage" du premier ordre est traditionnel en logique.

Par convention, nous notons les constantes avec les lettres du début de l'alphabet $a, b, c, \ldots$ et les variables avec les lettres de fin de l'alphabet $x, y, z, \ldots$.

Il faut aussi noter que nous ne considérons pas les symboles fonctionnels en dehors des constantes qui peuvent être vues comme des symboles fonctionnel d'arité nulle. Un *terme* de $\mathcal{L}$ est donc, soit un élément de $\mathcal{C}$ c'est-à-dire une constante, soit une variable.

Définition 2.2 (atome) *Soit $\mathcal{L}$ un langage du premier ordre. Un* atome *de $\mathcal{L}$ est de la forme $p(t_1, \ldots, t_k)$ où p est un prédicat de $\mathcal{P}$ d'arité k et $t_1, \ldots, t_k$ sont des termes de $\mathcal{L}$. L'ensemble des termes d'un atome A est noté* term(A)*, l'ensemble de ses variables* var(A)*, l'ensemble de ses constantes* const(A) *et son prédicat* pred(A)*. L'arité d'un prédicat p est notée* arité(p)*.*

Définition 2.3 (Formule bien formée) *Soit $\mathcal{L} = (\mathcal{P}, \mathcal{C})$ un langage logique, $\mathcal{V}$ un ensemble infini de variables, $\mathcal{Q} = \{\forall, \exists\}$ l'ensemble des quantificateurs, $\mathcal{N} = \{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ l'ensemble des connecteurs et $\mathcal{D} = \{(,)\}$ un jeu de parenthèse. On définit par induction $FBF(\mathcal{L})$ l'ensemble des* formules bien formées *(fbf), construites sur $\mathcal{L}$:*

- *base :*
  - *$FBF(\mathcal{L})$ contient l'ensemble des atomes construits sur $\mathcal{L}$.*
  - *$FBF(\mathcal{L})$ contient $\bot$ et $\top$ dans la mesure où ces symboles sont admis.*
- *induction : soit $\varphi, \psi \in FBF(\mathcal{L})$ et $x \in \mathcal{V}$:*
  - *$\neg\varphi \in FBF(\mathcal{L})$.*
  - *$(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi) \in FBF(\mathcal{L})$.*
  - *$\forall x\ \varphi$, $\exists x\ \varphi \in FBF(\mathcal{L})$.*

Dans la fbf $\forall x\ \varphi$, respectivement $\exists x\ \varphi$, x est la variable *quantifiée* et φ est la *portée* de la quantification $\forall x$, respectivement $\exists x$. Une occurrence d'une variable x est *liée* si elle est dans la portée d'une quantification portant sur x, sinon cette occurrence est *libre*.

Définition 2.4 (formule fermée) *Une fbf est dite* fermée *lorsqu'elle n'a aucune variable libre.*

Nous rappelons maintenant la définition de l'interprétation d'un langage logique :

Définition 2.5 (Interprétation d'un langage logique) *L'*interprétation (D, I) *d'un langage logique $\mathcal{L} = (\mathcal{P}, \mathcal{C})$ est constituée d'un ensemble non vide D appelé domaine d'interprétation et de I une fonction d'interprétation des symboles de $\mathcal{L}$ telle que :*

- *pour tout $c \in \mathcal{C}$, $I(c) \in D$*
- *pour tout $p \in \mathcal{P}$ d'arité k, $I(p) \subseteq D^k$*

Une interprétation de $\mathcal{L}$ est un *modèle* d'une formule sur $\mathcal{L}$ si elle rend vrai cette formule, en considérant l'interprétation classique des connecteurs et quantificateurs logiques.

Définition 2.6 (Satisfiabilité) *Une formule est* satisfiable *si elle admet au moins un modèle.*
Une formule est insatisfiable *si elle n'admet aucun modèle.*

Lorsque deux formules ont les mêmes modèles ont dira qu'elles sont équivalentes.

Définition 2.7 (Équivalence logique) *Soit F_1 et F_2 deux fbf fermées d'un même langage du premier ordre $\mathcal{L}$. On dit qu'elles sont* logiquement équivalentes *si pour toute interprétation elles ont la même valeur de vérité. On note $F_1 \equiv F_2$.*

Lorsque l'ensemble des modèles d'une formule est inclus dans celui d'un ensemble de formules, on dira que cette formule est conséquence logique de l'ensemble de formules.

Définition 2.8 (Conséquence logique) *Soit $H_1, \ldots, H_n$ et C des fbf fermées d'un même langage du premier ordre $\mathcal{L}$. On dit que C est une* conséquence logique *de (ou est impliquée par) $H_1, \ldots, H_n$ lorsque toute interprétation I de $\mathcal{L}$ qui est un modèle de $H_1 \wedge \cdots \wedge H_n$ est un modèle de C. On note $H_1, \ldots, H_n \models C$.*

On a immédiatement que pour toutes fbf fermées F_1 et F_2, $F_1 \equiv F_2$ si et seulement si $F_1 \models F_2$ et $F_2 \models F_1$.

Définition 2.9 (Littéral, clause, clause vide, clause de Horn, forme clausale) *Un* littéral *est un atome ou la négation d'un atome.*

Une clause *est une disjonction de littéraux quantifiée universellement. La* clause vide *est la disjonction de zéro littéraux, elle s'évalue toujours à faux.*

Une clause de Horn *est une clause dans laquelle il y a au plus un littéral positif.*

Une forme clausale *est une conjonction de clauses.*

On voit souvent les clauses comme des ensembles d'atomes et les formes clausales comme des ensembles d'ensembles d'atomes.

Définition 2.10 (Substitution) *Soit X un ensemble de variables et T un ensemble de termes. Une* substitution σ *de X dans T est une application de X dans T.*

Appliquer une substitution σ à une formule (ou à tout objet contenant des termes) consiste à remplacer toute variable $x \in X$ par son image $\sigma(x) \in T$.

Une substitution σ de X dans T sera représentée par un ensemble de couples $\{(x_1, \sigma(x_1)), \ldots, (x_k, \sigma(x_k))\}$ où les x_i sont les éléments de X.

Deux listes de termes sont unifiables s'il existe une substitution qui les rend identiques.

Définition 2.11 (Unificateur logique) *Soit $L_1, \ldots, L_n$ des listes de termes, un* unificateur *(logique) de $L_1, \ldots, L_n$ est une substitution de $\text{var}(L_1) \cup \cdots \cup \text{var}(L_n)$ dans $\text{term}(L_1) \cup \cdots \cup \text{term}(L_n)$ telle que $u(L_1) = \cdots = u(L_n)$.*

On étend cette définition à des atomes : des atomes sont unifiables s'ils ont même prédicat et si leur liste de termes sont unifiables. En général, il existe plusieurs unificateurs d'un ensemble E d'atomes ou de termes mais on s'intéressera à ceux qui "transforment" le moins possible les éléments de E.

Définition 2.12 (Unificateur le plus général) *Un unificateur u d'un ensemble E est un* unificateur le plus général *(upg) si tout autre unificateur u' s'obtient par une substitution supplémentaire s : $u' = s \circ u$.*

Un ensemble peut avoir plusieurs upg mais dans ce cas, on peut passer de l'un à l'autre par un simple renommage de variables.

Exemple 4 *Soit $E = \{p(x, y, z), p(u, a, z)p(b, y, v)\}$ où a et b sont des constantes et u, v, x, y, z sont des variables. $\{(x, b), (u, b), (y, a), (z, v)\}$ est un upg de E mais $\{(x, b), (u, b), (y, a), (v, z)\}$ est aussi un upg, on peut passer de l'un à l'autre en renommant v en z et vice-versa. En revanche, l'unificateur $\{(x, b), (u, b), (y, a), (z, b), (v, b)\}$ n'est pas un upg car il s'obtient du premier (par exemple) par la substitution supplémentaire $\{(v, b)\}$.*

La *méthode de résolution* due à J.A. Robinson 1965 permet en s'appuyant sur l'unification de tester si un ensemble de clauses est satisfiable, c'est-à-dire s'il admet au moins un modèle. Ainsi, elle permet de tester si une formule C est une conséquence logique d'un ensemble de formules $\{H_1, \ldots, H_n\}$ en vérifiant si la forme clausale associée à $H_1 \wedge \cdots \wedge H_n \wedge \neg C$ est insatisfiable. Toute formule F peut-être mise sous forme clausale, qui ne lui est pas forcément équivalente, mais qui préserve la satisfiabilité : F est insatisfiable si et seulement si sa forme clausale est insatisfiable. Cette mise sous forme clausale utilise la skolémisation pour supprimer les quantificateurs existentiels : après avoir mis la formule sous forme prénexe (c'est-à-dire avec tous les quantificateurs en tête de formule), chaque variable quantifiée existentiellement est remplacée par une nouvelle fonction dite de Skolem portant sur les variables quantifiées universellement qui précèdent.

Exemple 5 *Soit* $F = \forall x\ (q(x) \vee \exists y\ (p(x,y) \wedge q(y,x)))$.
Une forme prénexe de F *est :* $\forall x \exists y (q(x) \vee (p(x,y) \wedge q(y,x)))$.
Pour obtenir la forme clausale associée, il reste à skolémiser : $\forall x\ (q(x) \vee (p(x, f(x)) \wedge q(f(x), x)))$, *la variable* y *étant remplacée par une fonction de Skolem portant sur* x *car son quantificateur est dans la portée de celui de* x.
Pour finir on distribue les $\wedge$ *et* $\vee$ *pour obtenir une conjonction de clauses :* $\forall x\ ((q(x) \vee p(x, f(x))) \wedge (q(x) \vee q(f(x), x)))$.

La méthode de résolution s'appuie sur une règle dites de résolution qui unifie les parties complémentaires de deux clauses grâce à un upg pour produire une nouvelle clause. Si un littéral est positif, son complémentaire est obtenu en ajoutant une négation devant, sinon il est obtenu en supprimant sa négation. Le complémentaire d'un ensemble de littéraux est obtenu en remplaçant chaque littéral par son complémentaire.

Définition 2.13 (Règle de résolution) *Soit* C_1 *et* C_2 *deux clauses sans variable en commun et soit* $L_1 \subseteq C_1$ *et* $L_2 \subseteq C_2$ *tel que* L_1 *et le complémentaire de* L_2 *sont unifiables par un upg* u. *La résolvante de* C_1 *et* C_2 *selon* L_1, L_2 *et* u *est la clause* $(u(C_1) \setminus u(L_1)) \cup (u(C_2) \setminus u(L_2))$

La méthode de résolution est adéquate et complète : il existe une suite finie de résolutions menant à la clause vide *si et seulement si* la forme clausale est insatisfiable. On peut lui associer un algorithme de recherche en largeur [1] : partant de l'ensemble de clauses initiales, à chaque étape on calcule toutes les résolvantes d'un ensemble de clauses, les ajoute à l'ensemble de clauses et on recommence jusqu'à obtenir la clause vide ou qu'il n'existe plus de nouvelles résolvantes. Cet algorithme assure que l'on produira la clause vide s'il existe une suite finie de résolution y menant.

1. "breath-first" en anglais

2.2 Les faits et la requête

Les faits sont des données spécifiques sur les individus de la base. Il existe de nombreuses technologies permettant de stocker des faits, les plus connues sont bien sûr les bases de données relationnelles mais d'autres comme les bases de graphes ou les "triple-stores" peuvent aussi être pertinentes en fonction du contexte d'utilisation. Nous utilisons la logique du premier ordre pour faire abstraction de ces technologies. Habituellement en logique, les faits sont des ensemble d'atomes qui contiennent seulement des constantes, ici nous généralisons cette notion pour que les faits puissent contenir aussi des variables existentiellement quantifiées. La première raison est que les règles existentielles produisent des variables existentielles, mais cela permet aussi d'inclure naturellement les "blank nodes" de RDF et les valeurs manquantes de bases de données relationnelles.

Définition 2.14 (Fait) *Soit $\mathcal{L}$ un langage logique, un fait sur $\mathcal{L}$ est une conjonction d'atomes de $\mathcal{L}$ existentiellement fermée.*

Nous étendons les notations de l'ensemble des termes et de l'ensemble des variables à un fait. Dans un souci de simplicité, nous omettons les quantificateurs des faits dans les chapitres suivants, car les formules qui représentent les faits sont toujours existentiellement fermées. De plus, nous considérons souvent les faits comme des ensembles d'atomes afin de pouvoir utiliser les notions classiques de théorie des ensembles, comme l'inclusion ou l'union, directement sur les faits. Cette vision supprime les duplications d'atomes, en effet, une formule avec un atome dupliqué telle que $\exists x(p(x) \wedge p(x))$ sera vu par le même ensemble $\{p(x)\}$ que la formule $\exists x\ p(x)$ mais ce n'est pas gênant car ces deux formules sont trivialement équivalentes. Il faut aussi noter que l'on ne fera pas de distinction entre un seul fait et un ensemble de faits, en effet, un ensemble de faits est une conjonction de faits et donc assimilable à un fait. De la même manière, on utilisera indifféremment les termes de fait et de base de faits. Voici un exemple illustrant les notions vues précédemment :

Exemple 6 *Soit F la formule logique du premier ordre suivante :*

$$\exists x \exists y \exists z (r(x, y) \wedge p(x, x, a) \wedge r(a, z))$$

F est existentiellement fermée donc est un fait et $\text{var}(F) = \{x, y, z\}$, $\text{term}(F) = \{x, y, z, a\}$.
F peut aussi être vu comme l'ensemble $\{r(x, y), p(x, x, a), r(a, z)\}$

Il faut noter qu'il faut prendre quelques précautions avec les variables de même nom lorsque l'on considère des variables existentielles dans les faits.

Exemple 7 *Soit $F_1 = \exists x \exists y (p(x, y) \wedge q(y, a))$, où a est une constante, et $F_2 = \exists x\ p(x, x)$. F1 et F2 sont des faits. La formule $F_1 \wedge F_2$ peut être vue comme un seul fait et obtenu en considérant une forme prénexe, ce qui implique de renommer la variable x dans F_1 ou F_2 . On obtient par exemple $\exists x \exists y \exists z (p(x, y) \wedge q(y, a) \wedge p(z, z))$,qui peut aussi être vu comme l'ensemble d'atomes $\{p(x, y), q(y, a), p(z, z)\}$.*

Nous considérons classiquement des requêtes conjonctives, c'est-à-dire une conjonction d'atomes dont certaines variables sont existentiellement quantifiées. Les variables libres de la requête sont nommées *variables réponses*, une réponse à la requête est une instanciation de ces variables par des termes de la base de connaissances.

Définition 2.15 (Réponse à une requête) *Soit $\mathcal{F}$ un ensemble de faits, vu comme une seule formule et Q une requête avec comme ensemble ordonné de variables réponses $(x_1, \ldots, x_k)$. Le tuple $(a_1, ..., a_k)$ où chaque a_i est une constante est une réponse à Q si $\mathcal{F} \models Q_a$, où Q_a est obtenue de Q en remplaçant chaque x_i par a_i*

Les requêtes conjonctives sont les requêtes de base de tous les langages de requêtes, de plus ce sont les plus fréquemment utilisées en base de données relationnelle.
Voici un exemple de traduction d'une requête conjonctive dans différents langages de requêtes.

Exemple 8 *Requête : "trouver les x qui jouent dans un film".*

- Logique du premier ordre *:* $\exists y(film(y) \wedge joue(x, y))$
- Datalog *:* $answer(x)$ *:-* $film(y), joue(x, y).$
- SQL *:*
  SELECT joue.acteur FROM joue,films
  WHERE joue.œuvre = films.titre
- SPARQL *:*
  SELECT ?x WHERE {
  ?y rdf :type <http ://schema.org/Movie> .
  ?y <http ://schema.org/actor> ?x }

Lorsque la formule est fermée, on l'appelle requête conjonctive booléenne. Par la suite, nous considérerons seulement des requêtes conjonctives booléennes sans perte de généralité (voir section 4.3), nous les appellerons requête ou CQ. La réponse à une requête booléenne est vrai si la requête se déduit de la base de connaissances. On peut remarquer qu'il n'y a pas de différence entre la représentation des requêtes booléennes et des faits, la seule différence est conceptuelle.

De la même manière que pour les faits, nous omettons ensuite les quantificateurs puisque les formules pour les requêtes seront toujours existentiellement fermées. Les requêtes seront aussi vues comme des ensembles d'atomes. Il est aussi important de noter que tout le travail qui suit peut être facilement étendu aux unions de requêtes conjonctives. Une union de requête conjonctive est une disjonctions de requête conjonctives, si les requêtes sont booléennes, sa réponse est positive seulement si la réponse à l'un des requêtes qui la composent est positive. Si les requêtes ne sont pas booléennes, les réponses à une union de requêtes conjonctives est l'union des réponses des requêtes qui la composent.

Exemple 9 *Soit $Q = \exists x \exists y (r(x, y) \wedge q(y))$ une requête booléenne. Q peut aussi être vue comme l'ensemble $\{r(x, y), q(y)\}$*

Avant de pouvoir prendre en compte une ontologie, il est nécessaire de se pencher sur le problème de la conséquence logique. On sait que la conséquence logique est équivalente à l'existence d'un homomorphisme de G dans F (voir par exemple le théorème d'homomorphisme dans [Abiteboul et al., 1995]). Un homomorphisme de G dans F est une application des variables de G dans les termes de F qui préserve les atomes.

Définition 2.16 (Homomorphisme, isomorphisme) *Un* homomorphisme *d'un ensemble d'atomes* source *G vers un ensemble d'atomes* cible *F est une substitution π de $\text{var}(G)$ dans $\text{term}(F)$ telle que $\pi(G) \subseteq F$. On dit que G s'envoie sur F par π ou que G subsume F.*

Un isomorphisme *d'un ensemble d'atomes G vers un ensemble d'atomes F est une substitution bijective σ de $\text{var}(G)$ dans $\text{var}(F)$ telle que $\sigma(G) = F$; on appellera σ un renommage bijectif de variables.*

Lorsqu'il existe un homomorphisme d'un ensemble d'atomes G dans un ensemble d'atomes F, on dira que G est *plus général* que F ou F est *plus spécifique* que G, on note $G \geq F$. Les ensemble d'atomes les plus généraux d'un ensemble sont ceux qui ne sont plus spécifiques à aucun autre de l'ensemble. Dans la suite du manuscrit, on utilisera fréquemment le pré-ordre $\geq$ induit par l'homomorphisme pour comparer les faits ou requêtes entre eux.

Théorème 1 *Soit F et G deux faits, avec F possiblement infini. $F \models G$ si et seulement s'il existe un homomorphisme de G dans F.*

Exemple 10 *Soit deux faits $G = q(x) \wedge p(x, y) \wedge p(z, y)$ et $F = q(a) \wedge p(a, t) \wedge q(a, t)$. Soit la substitution $\pi = \{(x, a), (y, t), (z, a)\}$, $\pi(G) = q(a) \wedge p(a, t) \wedge p(a, t)$ donc π est un homomorphisme de G dans F et $F \models G$.*

Certains atomes d'un fait peuvent être redondants et ne pas apporter d'informations supplémentaires. Lorsque l'on voudra faire référence seulement à la partie "essentielle" d'un fait, on parlera de *noyau*[2] qui est une notion classique pour les graphes mais est aisément transférable aux faits (voir [Hell and Nesetril, 1992] qui situe son introduction à la fin des années 60).

Définition 2.17 (Noyau) *Le* noyau *d'un fait F, noté $\text{noyau}(F)$, est un sous-ensemble minimal (selon l'inclusion) de F équivalent à F.*

2. par traduction de “core” en anglais

Comme nous le montre l'exemple suivant, un fait peut avoir deux noyaux différents mais ils sont toujours isomorphes donc identiques à un renommage de variables près, c'est la raison pour laquelle nous nous permettons de considérer que le noyau est unique. De plus, il est bien connu que deux faits équivalents ont des noyaux isomorphes.

Exemple 11 *Soit* $F = t(y) \wedge p(x, y) \wedge p(z, y)$ F *a deux noyaux,* $t(y) \wedge p(x, y)$ *et* $t(y) \wedge p(z, y)$ *qui sont isomorphes.*

2.3 L'ontologie

L'ontologie contient des connaissances générales sur le domaine, telles qu'une hiérarchie de types par exemple. L'un des formalismes privilégiés pour la représentation d'ontologies est celui des logiques de description (DLs). Il s'agit d'une famille de langages dans laquelle les différentes DLs proposées se différencient par leur compromis expressivité/complexité. Historiquement, les DLs ont été conçues pour mettre en œuvre des raisonnements par classification sur une ontologie (c'est-à-dire pour comparer deux classes ou un individu et une classe par subsomption). Elles sont malheureusement peu adaptées au problème de l'interrogation d'une base de connaissances pour lequel elles ont une complexité trop importante. Pour répondre à ce problème, de nouvelles DLs, moins expressives, mais ayant une complexité moindre, ont été proposées et baptisées DLs légères. Dans la section qui suit, nous présenterons les DLs légères les plus connues DL-Lite et $\mathcal{EL}$. Une autre manière de formaliser les connaissances d'une ontologie est d'utiliser les règles existentielles ; c'est ce formalisme qui est principalement considéré dans ce manuscrit et qui est présenté dans la seconde section.

2.3.1 Logiques de description

Les DLs sont composées d'une ABox représentant la base de faits et d'une TBox décrivant l'ontologie. La TBox est un ensemble d'axiomes ontologiques qui s'appuient sur des concepts, qui sont les catégories auxquelles peuvent appartenir les individus de la base, et des rôles qui permettent d'exprimer une relation entre deux individus de la base. L'axiome le plus couramment utilisé est l'axiome d'inclusion entre deux concepts C_1 et C_2, qui signifie que tous les individus du concept C_1 appartiennent aussi au concept C_2. Les concepts et les rôles peuvent être atomiques ou construits au moyen d'autres concepts et rôles, et de constructeurs. L'expressivité d'une DL est plus ou moins grande selon les constructeurs autorisés. Voici un aperçu des principaux constructeurs de concepts et de rôles :

intersection de concepts notée $C_1 \sqcap C_2$, est composé des individus qui appartiennent à la fois au concept C_1 et au concept C_2.

union de concepts notée $C_1 \sqcup C_2$, est le concept regroupant les individus des concepts C_1 et C_2.

restriction universelle notée $\forall r.C$, est composé des individus qui ne sont en relation par le rôle r qu'avec des individus du concept C.

restriction existentielle notée $\exists r.C$, est composé des individus qui sont en relation par le rôle r à un individu du concept C.

restriction existentielle non qualifiée notée $\exists r$, il s'agit du concept composé des individus qui sont en relation par le rôle r à un individu quelconque.

négation d'un concept notée $\neg C$, 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 noté r^-, 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^-

Les logiques de description utilisent aussi le concept universel $\top$ auquel appartiennent tous les individus et le concept $\bot$ auquel n'appartient aucun individu. La ABox des logiques de description considérées ici contient uniquement des assertions de la forme :

$$A(a) \qquad r(a,b)$$

où A est un concept atomique, r un rôle atomique et a et b des individus.

La logique de description $\mathcal{EL}$

Pour des raisons historiques, les premières logiques de description étudiées favorisaient l'utilisation de la restriction universelle à la restriction existentielle. Pourtant la restriction universelle est un facteur important de complexité pour les problèmes de raisonnements même basiques. Ceci a motivé l'étude de $\mathcal{EL}$ [Baader, 2003], dont l'objectif était de gérer de grandes ontologies en permettant le raisonnement "intra TBox" (satisfiabilité, subsomption de concepts...) en temps polynomial. De plus, son expressivité s'avère suffisante pour un certain nombre de cas pratiques importants tels que l'ontologie biomédicale SNOMED.

La TBox quant à elle peut contenir des inclusions de concepts de la forme :

$$C_1 \sqsubseteq C_2$$

où C_1 et C_2 sont des concepts construits selon la règle suivante :

$$C ::= \top | A | C_1 \sqcap C_2 | \exists r.C_1$$

où A est un concept atomique, C_1 et C_2 des concepts construits et r un nom de rôle. Il existe un nombre important d'extensions de $\mathcal{EL}$ dont la plus fréquemment

rencontrée pour l'interrogation d'une base de connaissances est $\mathcal{ELHI}$. $\mathcal{ELHI}$ étend $\mathcal{EL}$ en autorisant l'utilisation des rôles inverses et l'inclusion de rôles dans la $TBox$ $r \sqsubseteq s$ pour l'inclusion du rôle r dans le rôle s, qui signifie que deux individus en relation par le rôle r le sont aussi par le rôle s.

La logique de description DL-Lite

DL-Lite est une famille de logiques de description qui a été introduite dans [Calvanese et al., 2005]. Elle a été conçue pour exprimer des ontologies simples tout en conservant une complexité basse pour l'interrogation, c'est-à-dire polynomiale en fonction de la taille de l'ontologie en complexité de données. Le but étant, non plus de vérifier simplement la subsomption de concepts ou la satisfiabilité de la base de connaissances comme c'était le cas pour les DLs antérieures, mais de répondre à des requêtes complexes et notamment aux requêtes conjonctives sur une base stockée en mémoire secondaire. L'idée principale est d'utiliser la TBox pour reformuler la requête en un ensemble de requêtes qui sont directement évaluées sur la ABox stockée en mémoire secondaire et gérée par un système de gestion des bases de données relationnelle. Cette méthode a l'avantage de permettre de séparer la ABox de la TBox et de profiter des optimisations pour le requêtage implémentées dans les systèmes de gestion de base de données. Nous présenterons ici les deux membres les plus simples de la famille DL-Lite $_{core}$ et DL-Lite$_{\mathcal{R}}$ qui constituent la base du langage du web sémantique OWL 2 QL.

Le premier membre de la famille DL-Lite $_{core}$ constitue la base de tous les autres membres de la famille. Il permet d'exprimer une négation très restreinte puisqu'elle ne peut porter que sur un rôle ou un concept basique et ne peut apparaître qu'en partie droite d'une inclusion. De plus, seule la restriction existentielle non qualifiée de rôles basiques est autorisée. Les rôles de DL-Lite $_{core}$ sont de deux types :

- rôles basiques : $q = p|p^-$
- rôles généraux : $r = q|\neg q$

où p est un rôle atomique et p^- est l'inverse d'un rôle atomique. De la même manière il y a deux types de concepts :

- concepts basiques : $B = A|\exists q$
- concepts généraux : $C = B|\neg B$

où A est un concept atomique et q un rôle basique.

De plus, la TBox ne contient que des inclusions de concepts de la forme suivante :

$$B \sqsubseteq C$$

où B est un concept basique et C est un concept général.

Dans une TBox DL-Lite$_\mathcal{R}$, on autorise aussi certaines inclusions de rôles :

$$q \sqsubseteq r$$

où q est un rôle basique et r est un rôle général.

2.3.2 Les règles existentielles

Nous arrivons maintenant au formalisme adopté dans ce manuscrit, les règles existentielles. Les règles ont dès le début été associées à l'intelligence artificielle, notamment dans les systèmes experts, puis dans la programmation logique. Les règles existentielles [Baget et al., 2011a] apparaissent sous différentes formes équivalentes dans la littérature : tuple-generating dependencies (TGD) [Abiteboul et al., 1995], règles *Datalog*$^\exists$ [Calì et al., 2008] renommées Datalog± dans [Calì et al., 2009], règles de graphes conceptuels [Salvat and Mugnier, 1996],∀∃-rules [Baget et al., 2009], ...

Définition 2.18 (Règles existentielles) *Les* règles existentielles *sur un langage logique $\mathcal{L}$ sont des formules fermées de la forme :*

$$R = \forall \vec{X} \forall \vec{Y} (H[\vec{X}, \vec{Y}] \rightarrow \exists \vec{Z} (C(\vec{Y}, \vec{Z})))$$

où H et C sont des conjonctions finies d'atomes sur $\mathcal{L}$ appelées respectivement hypothèse*,* prémisse *ou* corps *en programmation logique et* conclusion *ou* tête *de la règle et notées $hyp(R)$ respectivement $concl(R)$. L'ensemble de variables $\vec{Y}$, partagé par l'hypothèse et la conclusion, est appelé* frontière *de la règle et noté $fr(R)$.*

Par souci de concision, dans les chapitres suivants, les règles existentielles seront appelées simplement règles et les quantificateurs et le parenthésage seront omis dans les règles puisqu'il n'y a pas d'ambiguïté.

Exemple 12 *Voici une règle existentielle :*

$$R = \forall x \forall y ((p(x, y) \wedge q(y)) \rightarrow \exists z (r(y, z) \wedge q(z)))$$

ou plus simplement :

$$R = p(x, y) \wedge q(y) \rightarrow r(y, z) \wedge q(z)$$

et ses principaux composants : $hyp(R) = p(x, y) \wedge q(y)$, $concl(R) = r(y, z) \wedge q(z)$ et $fr(R) = \{y\}$.

Une règle existentielle n'est pas une clause (a fortiori une clause de Horn) à cause de la présence des variables existentielles, les deux sont cependant fortement liées car par skolémisation, on obtient des clauses de Horn (avec symbole fonctionnel). Dans le cas des règles existentielles, cette skolémisation revient à remplacer chaque variable existentielle de la conclusion par une fonction de skolem portant sur les variables de la frontière.

Exemple 13 *Soit* $R = \forall x \forall y(p(x,y) \rightarrow \exists z \exists t(s(x,z) \wedge s(z,t) \wedge s(t,x)))$.
La formule logique associée, non skolémisée, obtenue en faisant remonter les quantificateurs existentiels le plus en avant possible est la suivante :

$$\forall x \exists z \exists t \forall y(\neg p(x,y) \vee (s(x,z) \wedge s(z,t) \wedge s(t,x)))$$

En la skolémisant on obtient :

$$\forall x \forall y(\neg p(x,y) \vee (s(x,f(x)) \wedge s(f(x),g(x)) \wedge s(g(x),x)))$$

à partir de laquelle on obtient trois clauses de Horn :

$$\neg p(x,y) \vee s(x,f(x)) \qquad \neg p(x,y) \vee s(f(x),g(x)) \qquad \neg p(x,y) \vee s(g(x),x)$$

Ces clauses auraient été obtenues directement en remplaçant les variables existentielles par des fonctions de skolem portant sur la frontière et en découpant la règle en trois règles à conclusion atomique :

- $\forall x \forall y(p(x,y) \rightarrow s(x,f(x)))$
- $\forall x \forall y(p(x,y) \rightarrow s(f(x),g(x)))$
- $\forall x \forall y(p(x,y) \rightarrow s(g(x),x))$

Les règles existentielles permettent de produire de nouveaux faits à partir des faits existants.

Définition 2.19 (Application d'une règle) *Soit F un fait, et $R = H \rightarrow C$ une règle existentielle. R est* applicable *à F s'il existe un homomorphisme π de H dans F. Dans ce cas, l'application de R sur F produit un fait $\alpha(F,R,\pi) = F \cup \pi^{safe}(C)$, avec $\pi^{safe}(C) = \pi(\delta(C))$ où δ est une substitution qui renomme chaque variable de C qui n'appartient pas au domaine de π, par une variable fraîche, c'est-à-dire une nouvelle variable n'apparaissant nulle part ailleurs.*

Voici un exemple d'application de la règle précédente :

Exemple 14 *Soit $R = p(x,y) \wedge q(y) \rightarrow r(y,z) \wedge q(z)$ et $F = p(a,b) \wedge q(b) \wedge r(a,b)$. R est applicable à F par $\pi = \{(x,a),(y,b)\}$ et produit le fait :*

$$p(a,b) \wedge q(b) \wedge r(a,b) \wedge r(b,z_1) \wedge q(z_1)$$

où z_1 est une nouvelle variable quantifiée existentiellement.

Il existe aussi deux types de règles particulières, les règles avec égalité et les contraintes négatives. Les règles avec égalité généralisent la dépendance fonctionnelle.

Définition 2.20 (Règle avec égalité) *Les* règles avec égalité *sur un langage logique $\mathcal{L}$ sont des formules fermées de la forme :*

$$R = \forall \vec{X}(H[\vec{X}] \rightarrow x_i = x_j))$$

où H est une conjonction finie d'atomes sur $\mathcal{L}$ et x_i et x_j sont des variables distinctes de $\vec{X}$.

Les contraintes négatives expriment qu'un certain fait ne doit pas être déductible de la base. Elles sont souvent utilisées pour exprimer la disjonction de concepts ou l'incompatibilité de relation.

Définition 2.21 (Contrainte négative) *Les* contraintes négatives *sur un langage logique $\mathcal{L}$ sont des formules fermées de la forme :*

$$R = \forall \vec{X}(H[\vec{X}] \rightarrow \bot))$$

où H est une conjonction finie d'atomes sur $\mathcal{L}$.

Comme pour les règles existentielles classiques, les règles avec égalité et les contraintes négatives sont applicables à un fait lorsqu'il existe un homomorphisme π de leur hypothèse dans le fait. Pour l'application, par un homomorphisme π, d'une règle avec égalité dont la conclusion est $x_i = x_j$, il faut remplacer dans le fait toutes les occurrences de $\pi(x_i)$ par $\pi(x_j)$ (ou indifféremment toutes les occurrences de $\pi(x_j)$ par $\pi(x_i)$). Lorsqu'une contrainte négative est applicable sur un fait, celui-ci devient inconsistant et tout est déductible de lui.

Le principal avantage des règles existentielles est leur capacité à attester de l'existence d'entités non identifiées. Cette propriété, appelée invention de valeur en base de données, est essentielle pour la représentation de connaissances ontologiques en domaine ouvert. De plus, les règles existentielles généralisent la plupart des langages ontologiques utilisés pour l'interrogation. Notamment, les logiques de description vues précédemment sont toutes traduisibles en règles existentielles. C'est aussi le cas des principaux fragments du langage du web sémantique OWL qui sont dits "traitable" tels que OWL 2 QL, OWL 2 RL, OWL 2 EL (http ://www.w3.org/TR/owl2-profiles/). DL-Lite$_\mathcal{R}$ forme le cœur de OWL 2 QL tandis que OWL 2 EL est formé par $\mathcal{EL}$ et plus exactement $\mathcal{EL}$ ++ qui autorise en plus des constructeurs de $\mathcal{EL}$, le concept $\bot$ qui permet d'exprimer la disjonction de concepts, la composition de rôles dans des axiomes d'inclusion de rôles qui permet notamment d'exprimer la transitivité sur les rôles, les concepts nominaux (concept composé d'un seul individu) et une version restreinte des domaines concrets [Baader et al., 2005]. $\mathcal{EL}$ ++ étant indécidable pour l'interrogation, on se ramènera plus communément à $\mathcal{ELHI}$ pour l'interrogation. Le troisième fragment OWL 2 RL est lui étroitement lié au langage de règles Datalog. Toutes les logiques de descriptions légères sont exprimables au moyen des règles existentielles.

2.3.3 Traduction d'une base de logique de description en une base avec règles existentielles

La traduction d'une base de DL en une base avec règles existentielles se fait naturellement. Pour chaque individu présent dans la base on associe une constante du même nom dans le langage logique, pour chaque concept atomique un prédicat unaire du même nom et pour chaque rôle un prédicat binaire du même nom. Ainsi, si on appelle Φ la fonction de traduction, si A est un concept atomique $\Phi_A(x) = A(x)$ et si r est un rôle atomique, $\Phi_r(x, y) = r(x, y)$.

La $ABox$ est traduite à partir de cette fonction en un ensemble de faits. Par exemple, l'assertion $A(a)$ sera traduite en $A(a)$ où A est un prédicat unaire et a une constante, $r(a, b)$ sera traduite en $r(a, b)$ où r est un prédicat binaire et a et b des constantes.

La $TBox$ sera traduite en un ensemble de règles. Pour chaque assertion de la forme $B \sqsubseteq C$, on crée une règle $\forall x \ \ (\Phi_B(x) \rightarrow \Phi_C(x))$ où Φ_B est la traduction du concept B et Φ_C celle du concept C. Pour un axiome d'inclusion de rôles $r \sqsubseteq s$ dans une TBox, on utilisera la traduction suivante $\forall x, y \ \ (\Phi_r(x, y) \rightarrow \Phi_s(x, y))$ où Φ_r est la traduction du rôle r et Φ_s celle du rôle s.

Les concepts et rôles construits se traduisent de la manière suivante :

- $\Phi_{\exists r}(x) = \exists y \ \ (\Phi_r(x, y))$
- $\Phi_{r^-}(x, y) = \Phi_r(y, x)$
- $\Phi_{C \sqcap D}(x) = \Phi_C(x) \wedge \Phi_D(x)$
- $\Phi_{\exists \ r.C}(x) = \exists y \ \ (\Phi_r(x, y) \wedge \Phi_C(y))$

La traduction des concepts composés d'une restriction existentielle de rôles ($\exists r$) font apparaître des quantificateurs existentiels au milieu des formules. Il faut remonter ces quantificateurs en tête de formule et remplacer ceux portant sur l'hypothèse par des quantificateurs universels. En effet, une règle de la forme :

$$\forall x, y(\ \exists x' \ \ (P(x, x', y)) \rightarrow \exists z \ \ C(y, z))$$

est équivalente à la règle :

$$\forall x, y, x'(\ \ P(x, x', y) \rightarrow \exists z \ \ C(y, z))$$

Traduction de $\top$ en $\mathcal{EL}$

Une base $\mathcal{EL}$ est traduite de la manière décrite précédemment, la seule différence vient du concept universel $\top$. Pour le traduire, il faut d'abord créer un prédicat unaire $\top$ puis ajouter à l'ensemble de règles les règles suivantes :

- pour chaque prédicat unaire A : $\forall x \ \ (A(x) \rightarrow \top(x))$

- pour chaque prédicat binaire r : $\forall x, y \ \ (r(x, y) \rightarrow \top(x))$ et $\forall x, y \ \ (r(x, y) \rightarrow \top(y))$

Dans l'exemple suivant, on propose une traduction d'une base de connaissances $\mathcal{ELHI}$ en une base de connaissances avec règles existentielles.

Exemple 15

- $TBox = \{A \sqsubseteq \exists r.C, B \sqcap C \sqsubseteq A, \exists r^{-}.A \sqsubseteq C, s \sqsubseteq r^{-}\}$
- $ABox = \{A(a), B(b), r(a, b)\}$

La traduction est décrite sur le langage logique $\mathcal{L}$ = ($\mathcal{P}$, $\mathcal{C}$) où :

- $\mathcal{P} = \{r, s, A, B, C, \top\}$ *où* $A, B, C, \top$ *sont unaires et* r *et* s *sont binaires.*
- $\mathcal{C} = \{a, b\}$

Base de faits $= \{A(a), B(b), r(a, b)\}$
Base de règles :

- $\forall x \ \ (A(x) \rightarrow \exists y \ \ (r(x, y) \wedge C(y)))$
- $\forall x \ \ (B(x) \wedge C(x) \rightarrow A(x))$
- $\forall x, y \ \ (r(y, x) \wedge A(y) \rightarrow C(x))$
- $\forall x, y \ \ (s(x, y) \rightarrow r(y, x))$

Il reste ensuite à rajouter les règles liées au concept universel $\top$ *:*

- $\forall x \ \ (A(x) \rightarrow \top(x))$
- $\forall x \ \ (B(x) \rightarrow \top(x))$
- $\forall x \ \ (C(x) \rightarrow \top(x))$
- $\forall x, y \ \ (r(x, y) \rightarrow \top(x))$
- $\forall x, y \ \ (r(x, y) \rightarrow \top(y))$
- $\forall x, y \ \ (s(x, y) \rightarrow \top(x))$
- $\forall x, y \ \ (s(x, y) \rightarrow \top(y))$

Traduction de la négation en DL-Lite

La traduction d'une base DL-Lite est la même que celle présentée précédemment. La seule différence est au niveau des axiomes de la TBox qui contiennent des négations et sont traduits par des contraintes négatives. En effet, une règle de la forme :

$$\forall x(P(x) \rightarrow \neg C(x))$$

est équivalente à la règle :

$$\forall x(P(x) \wedge C(x) \rightarrow \bot)$$

Ainsi, les inclusions de concepts de la forme $B \sqsubseteq \neg C$ sont traduits par une règle $\forall x \ \ (\Phi_B(x) \wedge \Phi_C(x) \rightarrow \bot)$ où Φ_B est la traduction du concept B et Φ_C celle du concept C. Dans une TBox en DL-Lite$_\mathcal{R}$, on trouve aussi des inclusions de rôles de la forme $p \sqsubseteq \neg q$ qui sont traduits par une règle $\forall x, y \ \ (\Phi_p(x, y) \wedge \Phi_q(x, y) \rightarrow \bot)$, où Φ_p est la traduction du rôle p et Φ_q celle du rôle q. Puis, de la même manière que pour les inclusions classiques, les quantificateurs existentielles de l'hypothèse de la règle sont transformés en quantificateurs universels.

Voici un exemple de traduction d'une TBox DL-Lite$_\mathcal{R}$ en un ensemble de règles.

Exemple 16 *Soit une TBox* $\{A \sqsubseteq \exists q, p \sqsubseteq q^-, \exists q \sqsubseteq \neg\exists p\}$. *Sa traduction est décrite sur le langage logique* $\mathcal{L} = (\mathcal{P}, \mathcal{C})$ *où :*

- $\mathcal{P} = \{A, p, q\}$ *où* A *est un prédicat unaire et* p *et* q *sont des prédicats binaires.*

Traduction en règle :

- $\forall x \ \ (A(x) \rightarrow \exists y \ \ q(x, y))$
- $\forall x, y \ \ (p(x, y) \rightarrow q(y, x))$
- $\forall x, y, z \ \ (q(x, y) \wedge p(x, z) \rightarrow \bot)$

2.4 Les différentes approches

Le problème d'interrogation d'une base de connaissances étudié dans cette thèse peut donc être reformulé de la manière suivante : Étant donné une base de faits $\mathcal{F}$, une base de règles $\mathcal{R}$ et une requête conjonctive booléenne Q, est-ce qu'il existe une réponse à Q dans $\mathcal{F}$ selon $\mathcal{R}$? C'est-à-dire est-ce que Q est une conséquence logique de $\mathcal{F}$ et $\mathcal{R}$? Ce que l'on note $(\mathcal{F}, \mathcal{R}) \models Q$.

Il existe deux paradigmes classiques de traitement des règles, le premier, appelé marche avant, consiste à faire grossir la base de faits en appliquant toutes les règles possibles. S'il existe une réponse à la requête, on la trouvera dans la base de faits enrichie. Le second, appelé marche arrière, consiste à réécrire la requête en fonction des

règles. S'il existe une réponse à la requête, une des réécritures de la requête aura une réponse dans la base de faits initiale. Dans le contexte de l'interrogation d'une base de connaissances, on parle aussi de méthode avec matérialisation, respectivement sans matérialisation, de l'inférence des règles.

Il est connu depuis longtemps que le problème de l'interrogation d'une base de connaissances est indécidable [Beeri and Vardi, 1984], même avec une seule règle, ou en se restreignant à des prédicats unaires et binaires [Baget et al., 2011a]. Dans le cas général, la base de faits peut donc grossir indéfiniment en marche avant et le nombre de réécritures de la requête en marche arrière être infini. En revanche, les recherches ont mis en évidence des restrictions sur les ensembles de règles avec lesquelles le problème redevient décidable. La plupart de ces restrictions peuvent être classées en trois catégories reposant sur des propriétés abstraites apparentées pour deux d'entre elles à la marche avant et pour la dernière à la marche arrière.

2.4.1 Marche avant

La *marche avant* enrichit la base de faits en appliquant toutes les règles possibles puis interroge la base enrichie avec la requête initiale. L'étape d'application des règles, appelée saturation, se fait avec une stratégie en largeur pour garantir la complétude. On part d'un fait initial $\mathcal{F}_0$. Chaque étape i consiste à produire un fait appelé $\mathcal{F}_i$ à partir du fait courant, noté $\mathcal{F}_{i-1}$, en calculant tous les homomorphismes des hypothèses de chaque règle avec $\mathcal{F}_{i-1}$ puis en effectuant toutes les applications de règles correspondantes. Le fait $\mathcal{F}_k$ obtenu à l'étape k est appelé la k-saturation de $\mathcal{F}_0$ avec l'ensemble de règles. La marche avant peut-être aussi retrouvée sous le nom de *chase* dans la littérature de base de données, néanmoins il faut noter qu'il existe différentes variantes du *chase* qui se différencientpar leur manière de traiter la redondance (*oblivious* [Calì et al., 2008], *skolem* [Marnette, 2009], *restricted* [Fagin et al., 2005], *core chase* [Deutsch et al., 2008])

Définition 2.22 (k-saturation) *Soit F un fait, $\mathcal{R}$ un ensemble de règles et $\Pi(\mathcal{R}, F) = \{(R, \pi) | R \in \mathcal{R}$ et π est un homomorphisme de hyp(R) dans $F\}$ l'ensemble des homomorphismes de l'hypothèse d'une règle de $\mathcal{R}$ avec F. La* saturation directe *de F avec $\mathcal{R}$ est définie par :*

$$\alpha(F, \mathcal{R}) = F \cup \bigcup_{\substack{(R,\pi)\in\Pi(\mathcal{R},F) \\ avec\ R=H\to C}} \pi^{safe}(C)$$

*La k-*saturation *de F avec $\mathcal{R}$, notée $\alpha_k(F, \mathcal{R})$ est définie par induction de la façon suivante :*

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

La définition suivante donne la terminologie utilisée avec la saturation.

Définition 2.23 (Dérivation) *Soit F un fait et $\mathcal{R}$ un ensemble de règles. On appelle* $\mathcal{R}$-dérivation *de F un fait F' tel qu'il existe une séquence finie, appelée* séquence de dérivation*, $F = F_0, F_1, ..., F_k = F'$, où $\forall 1 \leq i \leq k$ il existe une règle $R = H \rightarrow C$ de $\mathcal{R}$ et un homomorphisme π de H dans F_{i-1} avec $F_i = \alpha(F_{i-1}, R, \pi)$.*

Le fait obtenu en saturant le fait initial F avec toutes les applications possibles d'une règle de l'ensemble de règles $\mathcal{R}$, est appelé la saturation de F par $\mathcal{R}$.

Définition 2.24 (Saturation) *Soit F un fait et $\mathcal{R}$ un ensemble de règles. La* saturation *de F par $\mathcal{R}$, noté $\alpha_\infty(F, \mathcal{R})$ est définie par :*

$$\alpha_\infty(F, \mathcal{R}) = \bigcup_{k \in \mathbb{N}} \alpha_k(F, \mathcal{R})$$

Ce fait a la particularité d'avoir comme modèle isomorphe le *modèle canonique* aussi connu sous le nom de *modèle universel* dans la littérature base de données. Le modèle canonique s'envoie sur n'importe quel modèle de F et $\mathcal{R}$, donc pour savoir si une requête q est induite par F et $\mathcal{R}$ il suffit de vérifier si le modèle canonique de F et $\mathcal{R}$ est un modèle de q.

Définition 2.25 (Modèle isomorphe) *Soit F un fait construit sur le langage logique $\mathcal{L} = (\mathcal{P}, \mathcal{C})$. Le* modèle isomorphe *à F, (D, I), est tel que :*

- *D est en bijection avec $\text{term}(F) \cup \mathcal{C}$ (pour simplifier les notations on considère que cette bijection est l'identité) ;*
- *pour tout $c \in \mathcal{C}$, $I(c) = c$;*
- *pour tout $p \in \mathcal{P}$, $I(p) = \{(t_1, \ldots, t_k) \mid p(t_1, \ldots, t_k) \in F\}$ si p apparaît dans F sinon $I(p) = \emptyset$.*

A partir de ces notions nous pouvons présenter le théorème suivant qui est fondamental pour résoudre le problème interrogation d'une base de connaissances.

Théorème 2 ([Baget et al., 2011a]) *Soit F un fait, q une requête et $\mathcal{R}$ un ensemble de règles. Les propriétés suivantes sont équivalentes :*

- *$(F, \mathcal{R}) \models q$;*
- *il existe un homomorphisme de q dans $\alpha_\infty(F, \mathcal{R})$;*
- *il existe un entier k tel qu'il y a un homomorphisme de q dans $\alpha_k(F, \mathcal{R})$.*

Après l'application d'une règle R par un homomorphisme π sur un fait F, R reste applicable par π sur $\alpha(F, R, \pi)$ mais cette application n'apporte aucune nouvelle information. Donc en pratique, lors de la saturation, nous ne considérons que les nouvelles applications de règles, c'est-à-dire les applications utilisant un nouvel homomorphisme. L'exemple suivant illustre une saturation finie.

Exemple 17 *Soit* $F = r(a,b) \wedge q(b)$, $\mathcal{R} = \{R_1, R_2\}$, $R_1 = r(x,y) \rightarrow s(x,y)$ *et* $R_2 = q(x) \rightarrow r(x,y)$. R_1 *est applicable à* F *par* $\pi_1 = \{(x,a),(y,b)\}$ *et* R_2 *est applicable à* F *par* $\pi_2 = \{(x,b)\}$, *on obtient donc le fait suivant en réalisant les applications correspondantes :*

$$F_1 = F \wedge s(a,b) \wedge r(b,y_1)$$

Ensuite, seule R_2 *est applicable avec un nouvel homomorphisme sur* F_1 *par* $\pi_3 = \{(x,b),(y,y_1)\}$ *pour obtenir :*

$$F_2 = F_1 \wedge s(b,y_1)$$

Plus aucune nouvelle application de règle ne peut être faite, donc la saturation s'arrête et $\alpha_\infty(F,\mathcal{R}) = F_2$.

Mais comme le montre l'exemple suivant, la saturation peut aussi produire des séquences de dérivation de longueur infinie, la saturation est alors infinie.

Exemple 18 *Soit* $F = q(a)$ *et* $\mathcal{R} = \{q(x) \rightarrow r(x,y) \wedge q(y)\}$.

$$\begin{aligned}
\alpha_1(F,\mathcal{R}) &= F \wedge r(a,y_1) \wedge q(y_1)\\
\alpha_2(F,\mathcal{R}) &= F_1 \wedge r(y_1,y_2) \wedge q(y_2)\\
\alpha_3(F,\mathcal{R}) &= F_2 \wedge r(y_2,y_3) \wedge q(y_3)\\
\alpha_4(F,\mathcal{R}) &= \ldots
\end{aligned}$$

La saturation de F *par* $\mathcal{R}$ *est infinie.*

Une manière de rendre le problème de l'interrogation d'une base de connaissances décidable est d'avoir une saturation équivalente à un fait fini. Ce sera la première propriété abstraite intéressante d'un ensemble de règles. Un ensemble de règles $\mathcal{R}$ est à *expansion finie* si pour tout fait F, la saturation de F par $\mathcal{R}$ est équivalente à un fait fini.

Définition 2.26 (Ensemble à expansion finie) *Un ensemble de règles* $\mathcal{R}$ *est appelé* ensemble à expansion finie *(fes pour "finite expansion set") si et seulement si, pour tout fait* F, *il existe un entier* k *tel que* $\alpha_k(F,\mathcal{R}) \equiv \alpha_\infty(F,\mathcal{R})$.

Dans l'exemple suivant, on peut voir une saturation infinie équivalente à un fait fini.

Exemple 19 *Soit* $F = q(a)$ *et* $\mathcal{R} = \{q(x) \rightarrow r(x,y) \wedge r(y,y) \wedge q(y)\}$.

$$\begin{aligned}
\alpha_1(F,\mathcal{R}) &= F \wedge r(a,y_1) \wedge r(y_1,y_1) \wedge q(y_1)\\
\alpha_2(F,\mathcal{R}) &= F_1 \wedge r(y_1,y_2) \wedge r(y_2,y_2) \wedge q(y_2)\\
\alpha_3(F,\mathcal{R}) &= F_2 \wedge r(y_2,y_3) \wedge r(y_3,y_3) \wedge q(y_3)\\
\alpha_4(F,\mathcal{R}) &= \ldots
\end{aligned}$$

$\alpha_\infty(F,\mathcal{R})$ *est infini mais est équivalent à* $\alpha_1(F,\mathcal{R})$, *en effet,* $\alpha_1(F,\mathcal{R}) \subseteq \alpha_\infty(F,\mathcal{R})$ *et chaque* $r(y_i,y_{i+1}) \wedge r(y_{i+1},y_{i+1}) \wedge q(y_{i+1})$ *de* $\alpha_\infty(F,\mathcal{R})$ *s'envoie sur* $r(y_1,y_1) \wedge q(y_1)$ *par* $\{(y_i,y_1),(y_{i+1},y_1)\}$.

Le problème de savoir si un ensemble de règles est *fes* est indécidable [Baget et al., 2011a], les ensembles de règles *fes* ne sont donc pas reconnaissables.

La seconde propriété abstraite d'un ensemble de règles liée à la marche avant n'est pas reconnaissable non plus, elle définit les ensembles à *largeur arborescente bornée*, c'est-à-dire que la saturation peut-être infinie mais sa structure est proche de celle d'un arbre.

Définition 2.27 (Ensemble à largeur arborescente bornée) *Un ensemble de règles $\mathcal{R}$ est appelé* ensemble à largeur arborescente bornée *(bts pour "bounded tree-width set") si et seulement si, pour tout fait F, il existe un entier b (dépendant de F et $\mathcal{R}$) tel que pour toute $\mathcal{R}$-dérivation F' de F, la largeur arborescente*[3] *de noyau(F) est inférieure ou égale à b.*

La borne b dépend de F ce qui implique que tout *fes* est aussi *bts*, il suffit de choisir b égal au nombre de termes du fait équivalent à la saturation de F et $\mathcal{R}$. En s'appuyant sur un résultat de [Courcelle, 1990], il a été prouvé que le problème d'interrogation d'une base de connaissances est décidable si l'ensemble de règles est *bts* [Calì et al., 2008, Baget et al., 2011a]. La preuve n'est pas constructive, elle ne fournit donc pas d'algorithme pour l'interrogation d'une base de connaissances avec des ensembles de règles *bts*. En revanche [Baget et al., 2011b, Thomazo et al., 2012, Thomazo, 2013b] propose un algorithme pour une sous-classe expressive de *bts* appelée *gbts*. Cette classe couvre la plupart des classes de règles concrètes connues qui sont *bts* et non *fes*. Il est à noter que cet algorithme est optimal en complexité combinée et en complexité de données dans le pire des cas.

2.4.2 Marche arrière

Historiquement, la marche arrière a d'abord été utilisée en programmation logique, notamment avec Prolog. Un programme logique positif est un ensemble de clauses de Horn représentant des faits (atomes sans variable) et des règles, pouvant comporter des symboles fonctionnels. On prouve qu'une requête conjonctive Q est conséquence logique d'un programme logique P en montrant que $P \wedge \neg Q$ est insatisfiable, à l'aide de la méthode de résolution (à noter que Prolog par exemple suit une stratégie en profondeur pour des raisons d'efficacité, et que cette stratégie n'est pas complète). Lorsque la clause vide est produite, on dit que l'on a "effacé" Q. A chaque étape, on unifie un atome de Q, appelé le but, avec un atome positif d'une clause (donc un fait ou une conclusion de règle) et on produit la réécriture correspondante. On peut découper le processus de production de la clause vide en deux parties. La première partie crée de nouvelles clauses à partir des buts et des règles (pour que ce découpage soit applicable, cela nécessite bien sûr que ce processus soit fini). La seconde partie produit la clause vide à partir d'une clause créée par la première

3. Voir définition 8.1 en annexe

phase et des faits. On remarque que si l'on efface Q avec des faits cela revient à trouver un homomorphisme de Q dans ces faits.

Nous en venons à une autre vision de la marche arrière introduite par l'article fondateur en OBQA [Calvanese et al., 2005] pour la logique de description DL-Lite. La marche arrière y est décomposée en deux étapes :

1. on calcule un ensemble de réécritures de la requête initiale, qui est un ensemble de requêtes conjonctives vu comme une union de requêtes conjonctives.

2. on interroge la base de faits avec cette union de requêtes conjonctives ce qui est équivalent d'un point de vue logique à chercher des homomorphismes (bien que le mécanisme soit implémenté en SQL).

Cette séparation des faits et de l'ontologie présente d'indéniables avantages, par exemple dans le cas où les données sont réparties dans plusieurs bases ou que l'on ne dispose pas des droits d'écritures sur les faits. Outre les problèmes d'accès aux données, la marche arrière évite les problèmes liés au grossissement d'une base de faits causé par la marche avant.

Nous avons montré dans la section 2.3.3 qu'une TBox DL-Lite se traduit en règles existentielles, et donc pas directement en clauses. L'unification doit donc être adaptée pour tenir compte des variables existentielles, ou bien les règles obtenues doivent être skolémisées. Ces deux approches ont été utilisées par la suite.

Les techniques de réécriture de la littérature peuvent être classées en deux catégories en fonction du type de la réécriture. La première technique consiste à réécrire la requête sous forme d'une union de requêtes conjonctives [Gottlob et al., 2011, Chortaras et al., 2011, Rodriguez-Muro et al., 2013], la seconde réécrit la requête en un programme Datalog [Pérez-Urbina et al., 2010, Gottlob and Schwentick, 2012, Eiter et al., 2012, Trivela et al., 2013].

L'existence d'une réécriture sous forme d'une union de requêtes conjonctives est assurée lorsqu'un ensemble de règles est reformulable en requête du premier ordre ("first-order rewritable"). Cette notion très commune dans la littérature concerne l'existence d'une réécriture, adéquate et complète, en requête du premier ordre ("first order query"). En pratique, ces requêtes sont équivalentes à des requêtes SQL.

Définition 2.28 (Reformulable en requête du premier ordre) *Soit $\mathcal{R}$ un ensemble de règles. $\mathcal{R}$ est* reformulable en requête du premier ordre *(FO-reformulable) si pour toute requête q, il existe q' une réécriture de q, en requête du premier ordre, telle que pour tout fait $\mathcal{F}$, on a $(\mathcal{F}, \mathcal{R}) \models q$ si et seulement si $\mathcal{F} \models q'$. On dit que q' est adéquate et complète en fonction de $\mathcal{R}$.*

Une autre propriété d'un ensemble de règles, assurant l'existence d'une réécriture sous forme d'une union de requêtes conjonctives, peut être trouvée dans la littérature [Baget et al., 2011a]. Elle assure directement l'existence d'une réécriture sous la forme d'une disjonction de conjonctions d'atomes, c'est-à-dire d'une union de requêtes conjonctives (UCQ pour "union of conjonctive queries").

Définition 2.29 (Ensemble à unification finie) *Soit $\mathcal{R}$ un ensemble de règles. $\mathcal{R}$ est appelé* ensemble à unification finie *(fus pour "finite unification set") si pour toute requête q, il existe $\mathcal{Q}$, une union de requêtes conjonctives, telle que pour tout fait F on a $(F, \mathcal{R}) \models q$ si et seulement s'il existe $q' \in \mathcal{Q}$ telle que $F \models q'$. On dit que la réécriture $\mathcal{Q}$ est adéquate et complète en fonction de $\mathcal{R}$.*

Même si la présence d'une union de requêtes conjonctives semble plus restrictive que celle d'une requête du premier ordre, nous sommes enclin à croire que les notions *fus* et FO-reformulable sont équivalentes. Cependant, aucune preuve de cela n'a été publiée à notre connaissance.

Comme pour les ensembles à expansion finie ou à largeur arborescente bornée, les ensembles de règles à unification finie ne sont pas reconnaissables [Baget et al., 2011a], ces classes sont donc dites abstraites. En revanche, il existe de nombreuses classes de règles, dites concrètes, qui sont reconnaissables et dont on connaît l'appartenance ou non aux trois classes abstraites. Ces classes de règles et leur classification sont répertoriées dans [Baget et al., 2011a].

Nous rappelons ici la définition des principales classes de règles *fus*.

Définition 2.30 (Règle à hypothèse atomique[Baget et al., 2011a]) *Une règle R est à* hypothèse atomique*, noté ah, si $hyp(R)$ contient un seul atome.*

La notion d'ensemble de règles *ah* est équivalente à la notion d'ensemble de règles "linear Datalog±". Ces règles Datalog contiennent un seul atome en hypothèse et en conclusion mais tout ensemble de règles peut être décomposé en un ensemble équivalent de règles à conclusion atomique (voir 7.4).

Exemple 20 *$R = p(x, x, z) \rightarrow r(x, y, z) \wedge A(z)$ est une règle ah, en effet, son hypothèse ne contient qu'un seul atome $p(x, x, z)$.*

Définition 2.31 (Règle à domaine restreint [Baget et al., 2011a]) *Une règle R est à* domaine restreint*, noté dr, si chaque atome de sa conclusion contient toutes ou aucune des variables de son hypothèse.*

Exemple 21 *$R = p(x, y) \wedge B(y) \rightarrow r(x, y, z) \wedge A(z)$ est une règle dr, en effet, $r(x, y, z)$ contient toutes les variables de l'hypothèse et $A(z)$ aucune.*

Définition 2.32 (Ensemble de règles "sticky" [Calì et al., 2010b]) *Soit $\mathcal{R}$ un ensemble de règles. On marque toutes les variables qui apparaissent dans l'hypothèse des règles de la manière suivante. D'abord, pour chaque règle $R \in \mathcal{R}$ et chaque variable v de $hyp(R)$, s'il existe un atome a de $concl(R)$ tel que v n'apparaît pas dans a alors on marque chaque occurrence de v dans $hyp(R)$. Ensuite on applique jusqu'à l'obtention d'un point fixe la procédure suivante : pour chaque règle $R \in \mathcal{R}$ si une variable marquée apparaît dans $hyp(R)$ à la position π alors pour chaque règle $R' \in \mathcal{R}$ (y compris $R = R'$) on marque chaque occurrence des variables de $hyp(R')$ qui apparaissent dans $concl(R')$ à la même position π. On dit que $\mathcal{R}$ est "sticky" s'il n'existe aucune règle $R \in \mathcal{R}$ telle qu'une variable marquée apparaît dans $hyp(R)$ plus d'une fois.*

Exemple 22 *Soit* $R = A(x) \wedge r(x, y) \rightarrow r(y, z)$, $\{R\}$ *n'est pas "sticky" car* x *est marqué et apparaît deux fois dans l'hypothèse de la règle.*

Soit $R_1 = r(x_1, y_1) \wedge s(y_1, z_1) \rightarrow t(y_1, u_1)$ *et* $R_2 = t(x_2, y_2) \rightarrow r(y_2, x_2)$ *à l'initialisation,* x_1 *et* z_1 *sont marquées, puis le marquage de* x_1 *se propage à* y_2 *car* y_2 *apparaît dans* R_2 *à la première position de* r *comme* x_1 *dans* R_1*. Finalement, les variables marquées sont* x_1,z_1 *et* y_2 *donc* $\{R_1, R_2\}$ *est "sticky" puisqu'aucune n'apparaît deux fois dans l'hypothèse d'une règle.*

On peut noter que les ensembles de *ah*, *dr* et "sticky" sont incomparables. La règle de l'exemple 20 est *ah* mais ni *dr* car $A(z)$ contient une seule des variables l'hypothèse, ni "sticky" car x est marquée et apparaît deux fois dans l'hypothèse. Celle de l'exemple 21 est *dr* mais n'est pas *ah* elle contient deux atomes en hypothèse ou "sticky" car y est marquée et apparaît deux fois dans l'hypothèse. Enfin, celle de l'exemple 22 est "sticky" mais ni *ah*, elle a deux atomes en hypothèse, ni *dr*, $t(y_1, u_1)$ ne contient pas z_1.

Enfin, les règles "sticky-join" généralisent les règles "sticky" et à hypothèse atomique [Calì et al., 2010a], la définition s'appuie aussi sur un marquage de variables mais qui est plus sophistiqué que celui pour les règles "sticky".

Il existe encore d'autres ensembles de règles *fus*, comme les *a-GRD* (pour "acyclic graph of rule dependencies" [Baget et al., 2009]) qui ont une condition d'acyclicité sur un graphe de dépendance entre règles, elles sont aussi *fes* et *bts*.

L'ensemble de règles qui traduit une ontologie DL-Lite est *fus*, de nombreux systèmes mettent à profit cette propriété et font de la réécriture de requêtes conjonctives en UCQ par une ontologie DL-Lite (voir section 7.3). En revanche, cette méthode n'est pas applicable pour une ontologie $\mathcal{EL}$ dont l'ensemble de règles correspondant n'est pas *fus*. Par contre, l'ensemble de règles qui traduit une ontologie $\mathcal{EL}$ admet une réécriture d'une requête conjonctive sous la forme d'un programme Datalog. Il existe d'autres ensembles de règles, tels que ceux qui traduisent les ontologies $\mathcal{ELHI}$, qui admettent une réécriture sous forme d'un programme Datalog mais pas sous forme d'une UCQ. De plus, les réécritures Datalog ont aussi un intérêt lorsque les règles sont *fus*, car elles permettent une réécriture plus compacte.

Définition 2.33 (Règle Datalog) *Une règle Datalog est une expression de la forme* α *:-* $\beta_1, \ldots, \beta_n$ *où* α*,* β_1*, ...,* β_n *sont des atomes et chaque variable de* α *doit apparaître au moins une fois dans* $\beta_1, \ldots, \beta_n$*.* α *est appelé la tête de la règle et* $\beta_1, \ldots, \beta_n$ *est appelé le corps.*

Une règle Datalog est donc une règle existentielle qui a un seul atome en conclusion et aucune variable existentielle. Une réécriture Datalog ou programme Datalog est simplement un ensemble de règles Datalog avec un prédicat particulier "réponse", qui ne fait pas partie des prédicats présents dans la base et qui ne peut apparaître qu'en conclusion d'une règle. Le prédicat réponse a évidemment la même arité partout dans le programme.

Le principal inconvénient des réécritures Datalog est qu'elles nécessitent un système de gestion des bases de données implémentant Datalog, ces systèmes étant peu développés et leurs performances restant à prouver. En revanche, ce problème peut être évité si le programme Datalog est non récursif, il peut alors être traduit simplement en une UCQ et être exécuté sur un système de gestion de bases de données classique.

Définition 2.34 (Programme Datalog non-récursif) *Un programme Datalog est non-récursif s'il existe un ordre total $r_1, \ldots, r_n$ sur ses règles tel que le prédicat de la tête d'une règle r_i n'apparaît pas dans le corps d'une règle r_j telle que $i \leq j$.*

Chapitre 3

Un cadre théorique pour la réécriture en UCQ

Dans ce chapitre, nous définissons les propriétés souhaitées des ensembles de réécritures que nous allons calculer. Puis nous proposons un algorithme générique de réécritures ainsi que la preuve de sa correction lorsque l'ensemble de règles donné est *fus*.

3.1 Propriétés d'un ensemble de réécritures

Les techniques de réécritures en UCQ produisent, à partir d'une requête et d'un ensemble de règles, un ensemble de requêtes, que l'on appellera souvent ensemble de réécritures. Puisque le but est d'interroger la base de faits avec ces réécritures pour obtenir les réponses de la requête initiale dans la base de connaissances, il faut que cet ensemble de réécritures soit adéquat et complet pour que les réponses soient bien celles souhaitées. De plus, pour que l'interrogation soit rapide, il faut que cet ensemble soit aussi minimal. En résumé, nous désirons que notre ensemble de réécritures ait trois propriétés : adéquation, complétude et minimalité.

Définition 3.1 (Ensemble adéquat et complet) *Soit $\mathcal{R}$ un ensemble de règles, Q une requête et $\mathcal{Q}$ un ensemble de requêtes. $\mathcal{Q}$ est* adéquat *(en fonction de $\mathcal{R}$ et Q) si pour tout fait F et toute requête $Q' \in \mathcal{Q}$, $F \models Q'$ implique $(F, \mathcal{R}) \models Q$. $\mathcal{Q}$ est* complet *(en fonction de $\mathcal{R}$ et Q) si pour tout fait F, si $(F, \mathcal{R}) \models Q$ alors il existe $Q' \in \mathcal{Q}$ telle que $F \models Q'$.*

Pour obtenir la propriété de minimalité tout en conservant la complétude, il faut ne garder que les éléments les plus généraux de l'ensemble. En effet, soit deux requêtes Q_1 et Q_2 telles que $Q_1 \geq Q_2$ (autrement dit, Q_1 subsume Q_2, voir définition 2.16), pour tout fait F, l'ensemble des réponses de Q_2 est inclus dans l'ensemble des réponses de Q_1. Cette propriété est due au fait que l'homomorphisme est transitif, s'il existe un homomorphisme de Q_1 dans Q_2 et de Q_2 dans F alors il y en a un de

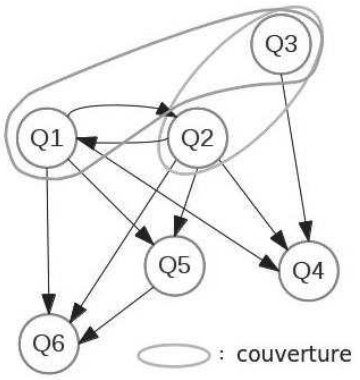

FIGURE 3.1 – Couverture (exemple 23)

Q_1 dans F. Ainsi, enlever les éléments plus spécifiques ne va pas compromettre la complétude. L'ensemble de réécritures désiré est donc un ensemble de requêtes adéquates et incomparables deux à deux qui "couvre" l'ensemble de toutes les réécritures adéquates de la requête initiale.

Définition 3.2 (Relation de couverture) *Soit $\mathcal{Q}_1$ et $\mathcal{Q}_2$ deux ensembles de réécritures. On dit que $\mathcal{Q}_1$ couvre $\mathcal{Q}_2$, noté $\mathcal{Q}_1 \geq \mathcal{Q}_2$, si pour chaque requête $Q_2 \in \mathcal{Q}_2$ il existe $Q_1 \in \mathcal{Q}_1$ telle que $Q_1 \geq Q_2$.*

Un ensemble de réécritures est minimal au sens de l'inclusion selon cette relation de couverture.

Définition 3.3 (Minimalité d'un ensemble de requêtes) *Soit $\mathcal{Q}$ un ensemble de réécritures, $\mathcal{Q}$ est minimal s'il n'existe pas de requêtes $Q \in \mathcal{Q}$ telle que $\mathcal{Q} \setminus \{Q\} \geq \mathcal{Q}$.*

Un sous-ensemble minimal de réécritures qui couvre l'ensemble des réécritures est appelée couverture.

Définition 3.4 (Couverture d'un ensemble de requêtes) *Soit $\mathcal{Q}$ un ensemble de réécritures, une couverture de $\mathcal{Q}$ est un ensemble minimal de requêtes $\mathcal{Q}_c \subseteq \mathcal{Q}$ tel que $\mathcal{Q}_c \geq \mathcal{Q}$.*

Voici un exemple de couverture d'un ensemble de requêtes illustré par la figure 3.1.

Exemple 23 *Soit $\mathcal{Q} = \{Q_1, \ldots, Q_6\}$ sur lequel on a les relations suivantes : $Q_1 \geq Q_2$, Q_4, Q_5, Q_6 ; $Q_2 \geq Q_1$, Q_4, Q_5, Q_6 ; $Q_3 \geq Q_4$; $Q_5 \geq Q_6$. Q_1 et Q_2 sont donc équivalentes, et il y a deux couvertures de $\mathcal{Q}$, $\{Q_1, Q_3\}$ et $\{Q_2, Q_3\}$.*

Étant donné qu'une couverture est un ensemble minimal, tous ses éléments sont deux à deux incomparables. On peut aussi prouver que deux couvertures d'un même ensemble ont la même cardinalité. Si de plus, on considère que chaque élément des couvertures est de taille minimale, c'est-à-dire qu'il s'agit de noyaux, les deux couvertures sont identiques à un isomorphisme près. Ainsi, quelle que soit la technique utilisée pour le calculer, il existe un ensemble unique (à un isomorphisme près) de réécritures adéquat, complet et minimal dont les éléments sont de taille minimale.

Théorème 3 ([König et al., 2012]) *soit $\mathcal{R}$ un ensemble de règles fus et Q une requête. Il existe un unique ensemble fini de réécritures de Q selon $\mathcal{R}$ adéquat, complet et minimal dont les éléments sont de taille minimale.*

Preuve : Soit $\mathcal{Q}_1$ et $\mathcal{Q}_2$ deux ensembles de réécritures de Q avec $\mathcal{R}$ adéquats et complets, par définition de *fus*, on sait que de tels ensembles existent. Soit $\mathcal{Q}_1^c$, $\mathcal{Q}_2^c$ une de leurs couvertures respectives. $\mathcal{Q}_1^c$ et $\mathcal{Q}_2^c$ sont aussi adéquats et complets et par définition minimales au sens de l'inclusion. Nous montrons qu'ils ont la même cardinalité. Soit $Q_1 \in \mathcal{Q}_1^c$, il existe $Q_2 \in \mathcal{Q}_2^c$ telle que $Q_1 \leq Q_2$ car $\mathcal{Q}_2^c$ est complet. De la même manière, il existe $Q_1' \in \mathcal{Q}_1^c$ telle que $Q_2 \leq Q_1'$. Ainsi, $Q_1 \leq Q_1'$ ce qui signifie que $Q_1' = Q_1$ puisque $\mathcal{Q}_1^c$ est une couverture. Donc pour tout $Q_1 \in \mathcal{Q}_1^c$, il existe $Q_2 \in \mathcal{Q}_2^c$ telle que $Q_1 \leq Q_2$ et $Q_2 \leq Q_1$. Une telle Q_2 est unique puisque les éléments de $\mathcal{Q}_2^c$ sont incomparables deux à deux. La fonction associant Q_1 à Q_2 est donc une bijection de $\mathcal{Q}_1^c$ dans $\mathcal{Q}_2^c$, ce qui montre que les deux ensembles ont la même cardinalité. Si nous imposons de plus, que les éléments de $\mathcal{Q}_1^c$ et $\mathcal{Q}_2^c$ soient de taille minimale, Q_1 et Q_2 seront isomorphes à leur noyau et donc isomorphes. □

Il est aussi important de noter que même si l'ensemble des réécritures adéquates d'une requête est infinie, sa couverture peut être finie.

Exemple 24 *Soit $Q = t(u)$, et $R_1 = t(x) \wedge p(x,y) \rightarrow r(y)$, $R_2 = r(x) \wedge p(x,y) \rightarrow t(y)$. R_1 et R_2 ont une conclusion réduite à un seul atome et aucune variable existentielle donc on peut utiliser des unificateurs les plus généraux classiques, qui unifient le premier atome de la requête avec l'atome de la conclusion de la règle. L'ensemble des réécritures de Q avec $\{R_1, R_2\}$ est infini : Les premières requêtes générées sont les suivantes (noter que les variables des règles sont renommées quand c'est nécessaire) :*
$Q_0 = t(u)$
$Q_1 = r(x) \wedge p(x,y)$ *// à partir de Q_0 et R_2 avec $\{(u,y)\}$*
$Q_2 = t(x_0) \wedge p(x_0,y_0) \wedge p(y_0,y)$ *// à partir de Q_1 et R_1 avec $\{(x,y_0)\}$*
$Q_3 = r(x_1) \wedge p(x_1,y_1) \wedge p(y_1,y_0) \wedge p(y_0,y)$ *// à partir de Q_2 et R_2 avec $\{(x_0,y_1)\}$*
$Q_4 = t(x_2) \wedge p(x_2,y_2) \wedge p(y_2,y_1) \wedge p(y_1,y_0) \wedge p(y_0,y)$ *// à partir de Q_3 et R_1*
et ainsi de suite ...

En revanche, l'ensemble des réécritures les plus générales est $\{Q_0, Q_1\}$ puisque toutes les autres que l'on peut obtenir sont plus spécifiques.

3.2 Algorithme de réécriture

L'algorithme présenté est un algorithme classique de réécriture, il prend en paramètre un opérateur de réécriture et une fonction de couverture et produit en sortie un ensemble de réécritures. La question est de savoir quelles sont les propriétés de l'opérateur de réécriture qui assurent que l'ensemble de réécritures produit est adéquat, complet et minimal. Voici d'abord une définition d'un opérateur de réécriture.

Définition 3.5 (Opérateur de réécriture) *Un opérateur de réécriture rew, ou plus simplement opérateur, est une fonction qui prend en entrée une requête conjonctive Q et un ensemble de règles $\mathcal{R}$ et qui donne en sortie un ensemble de requêtes conjonctives, noté* $rew(Q, \mathcal{R})$.

Un opérateur de réécriture va permettre de calculer des réécritures directes d'une requête Q, qui vont à leur tour être réécrites en utilisant ce même opérateur et ainsi de suite. Pour cette raison, nous étendons cette définition à un ensemble de requêtes $\mathcal{Q}$ au lieu d'une seule requête Q : $rew(\mathcal{Q}, \mathcal{R}) = \bigcup_{Q \in \mathcal{Q}}(rew(Q, \mathcal{R}))$. Nous aurions pu considérer qu'un opérateur de réécriture prend en entrée une requête Q et une seule règle R et fournit les réécritures directes de Q avec R, mais notre définition permet de considérer des réécritures qui utilisent plusieurs règles simultanément (voir section 4.6).

L'algorithme 1 cherche à produire une couverture de l'ensemble des réécritures d'une requête. Il effectue une exploration en largeur de l'espace des réécritures. A chaque pas, on calcule les réécritures de l'ensemble précédent. Seules les réécritures les plus générales sont gardées grâce à une fonction de couverture *cover* qui calcule la couverture d'un ensemble de requêtes. Ce sont ces requêtes que l'on va tenter de réécrire au tour suivant ; lorsque l'on tente de réécrire une requête on dira qu'on l'explore. Pour des raisons de terminaison, la fonction *cover* retourne une couverture qui garde en priorité les requêtes déjà explorées, c'est-à-dire que si $\mathcal{Q}_c \cup \{q\}$ et $\mathcal{Q}_c \cup \{q'\}$ sont toutes deux des couvertures de $\mathcal{Q}_F \cup rew(\mathcal{Q}_E, \mathcal{R})$, avec q et q' équivalente par homomorphisme et $\{q\}$ appartenant à $\mathcal{Q}_F$, alors *cover* ne produit pas $\mathcal{Q}_c \cup \{q'\}$.

La suite de cette section est dédiée à la présentation des propriétés de l'opérateur de réécriture ainsi que les preuves associées qui garantissent que premièrement l'algorithme s'arrête et produit une couverture de l'ensemble de toutes les réécritures productibles par l'opérateur et deuxièmement, que cette couverture est adéquate et complète.

3.2.1 Terminaison et correction de l'algorithme

Nous avons précédemment présenté la définition d'un opérateur de réécriture *rew* qui à partir d'une requête Q produit un ensemble de réécritures $rew(Q, \mathcal{R})$. Puisque les éléments de $rew(Q, \mathcal{R})$ sont aussi des requêtes conjonctives, il est possible de leur appliquer récursivement plusieurs pas de réécritures. Une réécriture obtenue après k étapes de réécriture est une k-réécriture.

Algorithme 1 : Algorithme de réécriture

Données : Une requête Q, un ensemble de règles $\mathcal{R}$
Paramètres : Un opérateur de réécriture *rew*, une fonction de couverture *cover*
Résultat : Une couverture de l'ensemble de toutes les réécritures de Q selon $\mathcal{R}$
début
  $\mathcal{Q}_F \leftarrow \{Q\}$; `// ensemble de réécritures final`
  $\mathcal{Q}_E \leftarrow \{Q\}$; `// ensemble de réécritures à explorer`
  tant que $\mathcal{Q}_E \neq \emptyset$ **faire**
    $\mathcal{Q}_C \leftarrow cover(\mathcal{Q}_F \cup rew(\mathcal{Q}_E, \mathcal{R}))$; `// mise à jour de la couverture`
    $Q_E \leftarrow \mathcal{Q}_C \setminus \mathcal{Q}_F$;
    $\mathcal{Q}_F \leftarrow \mathcal{Q}_C$;
  fin tant que
fin

Définition 3.6 (k-réécriture) *Soit Q une requête, $\mathcal{R}$ un ensemble de règles et rew un opérateur de réécriture. Une* 1*-réécriture de Q (selon $\mathcal{R}$ et rew) ou* réécriture directe *est un élément de $rew(Q, \mathcal{R})$. Une k-réécriture de Q (selon $\mathcal{R}$ et rew), $k > 1$, est une* 1*-réécriture d'une $(k-1)$-réécriture de Q (selon $\mathcal{R}$ et rew). On note l'ensemble des k-réécritures de Q (selon rew et $\mathcal{R}$) par $rew_k(Q, \mathcal{R})$.*

Les réécritures atteignables en, au plus k étapes de réécritures, sont appelées $\leq k$-réécritures.

Définition 3.7 ($\leq k$-réécriture) *Soit Q une requête, $\mathcal{R}$ un ensemble de règles et rew un opérateur de réécriture. Une $\leq k$-réécriture de Q (selon $\mathcal{R}$ et rew), est une i-réécriture de Q (selon $\mathcal{R}$ et rew) telle que $i \leq k$. On note l'ensemble des $\leq k$-réécritures de Q (selon rew et $\mathcal{R}$) par $\mathcal{W}_k(Q, \mathcal{R})$ et $\mathcal{W}_\infty(Q, \mathcal{R}) = \bigcup_{k \in \mathbb{N}} \mathcal{W}_k(Q, \mathcal{R})$.*

Ces définitions peuvent être étendues à un ensemble de requêtes $\mathcal{Q}$ au lieu d'une seule requête :

$$rew_k(\mathcal{Q}, \mathcal{R}) = \bigcup_{Q \in \mathcal{Q}} (rew_k(Q, \mathcal{R}))$$

$$\mathcal{W}_k(\mathcal{Q}, \mathcal{R}) = \bigcup_{Q \in \mathcal{Q}} (\mathcal{W}_k(Q, \mathcal{R}))$$

$$\mathcal{W}_\infty(\mathcal{Q}, \mathcal{R}) = \bigcup_{Q \in \mathcal{Q}} (\mathcal{W}_\infty(Q, \mathcal{R}))$$

Nous présentons maintenant une propriété qui assure que l'algorithme 1 calcule bien une couverture de $\mathcal{W}_\infty(Q, \mathcal{R})$. En effet, pour un opérateur quelconque la sélection des réécritures les plus générales à chaque pas de réécriture peut mener à l'incomplétude de la couverture. Dans l'exemple qui suit, une requête subsumée par une autre est éliminée sans être explorée, et il se peut que l'une de ses réécritures fasse partie de la couverture, alors celle-ci ne sera jamais calculée.

Exemple 25 *Soit rew un opérateur, $\mathcal{R}$ un ensemble de règles et Q une requête tels que $rew(Q,\mathcal{R}) = \{Q_1\}$, $rew(Q_1,\mathcal{R}) = \{Q_2\}$ et $rew(Q_2,\mathcal{R}) = \emptyset$ donc $\mathcal{W}_\infty(Q,\mathcal{R}) = \{Q, Q_1, Q_2\}$. Si Q et Q_2 sont incomparables et que Q est plus générale que Q_1 alors la seule couverture de $\mathcal{W}_\infty(Q,\mathcal{R})$ est $\{Q, Q_2\}$. Or, lors du premier pas de l'algorithme 1, Q_1 sera calculée par rew mais éliminée directement par cover puisqu'elle est plus spécifique que Q, et l'exécution s'arrêtera puisqu'il ne restera plus de requête non explorée, ainsi Q_2 ne sera jamais calculée et l'ensemble de sortie $\{Q\}$ ne sera pas une couverture de $\mathcal{W}_\infty(Q,\mathcal{R})$.*

La propriété suivante d'élagabilité sur un opérateur garantit que le cas de figure précédent ne peut pas se produire. Un opérateur est élagable si lorsqu'une requête est plus générale qu'une autre, l'ensemble des réécritures directes de la première couvre l'ensemble des réécritures directes de la seconde, ainsi, éliminer la requête la plus spécifique n'empêchera pas d'obtenir toutes les réécritures de la couverture.

Définition 3.8 (Élagable) *Soit rew un opérateur de réécriture. rew est* élagable *si pour tout ensemble de règles $\mathcal{R}$ et pour toutes requêtes Q_1, Q_2, Q'_2 telles que $Q_1 \geq Q_2$, $Q'_2 \in rew(Q_2,\mathcal{R})$ et $Q_1 \not\geq Q'_2$, il existe $Q'_1 \in rew(Q_1,\mathcal{R})$ telle que $Q'_1 \geq Q'_2$.*

Le lemme qui suit est une application récursive de la définition sur un ensemble de requêtes.

Lemme 1 *Soit rew un opérateur de réécriture élagable, et soit $\mathcal{Q}_1$ et $\mathcal{Q}_2$ deux ensembles de requêtes. Pour tout ensemble de règles $\mathcal{R}$, si $\mathcal{Q}_1 \geq \mathcal{Q}_2$, alors $\mathcal{W}_\infty(\mathcal{Q}_1,\mathcal{R}) \geq \mathcal{W}_\infty(\mathcal{Q}_2,\mathcal{R})$.*

Preuve : On prouve par induction sur i que $\mathcal{W}_i(\mathcal{Q}_1,\mathcal{R}) \geq rew_i(\mathcal{Q}_2,\mathcal{R})$.
Pour $i = 0$, $\mathcal{W}_0(\mathcal{Q}_1,\mathcal{R}) = \mathcal{Q}_1 \geq \mathcal{Q}_2 = rew_0(\mathcal{Q}_2,\mathcal{R})$.
Pour $i > 0$, pour tout $Q_2 \in \mathcal{W}_i(\mathcal{Q}_1,\mathcal{R})$, soit $Q_2 \notin rew_i(\mathcal{Q}_2,\mathcal{R})$, et on peut conclure directement avec l'hypothèse d'induction. Soit $Q_2 \in rew_i(\mathcal{Q}_2,\mathcal{R})$, et il existe une requête $Q'_2 \in rew_{i-1}(\mathcal{Q}_2,\mathcal{R})$ telle que $Q_2 \in rew(Q'_2,\mathcal{R})$. Selon l'hypothèse d'induction, il existe $Q'_1 \in \mathcal{W}_{i-1}(\mathcal{Q}_1,\mathcal{R})$ telle que $Q'_1 \geq Q'_2$. *rew* est élagable, donc par définition soit $Q'_1 \geq Q_2$, soit il existe $Q_1 \in rew(Q'_1,\mathcal{R})$ telle que $Q_1 \geq Q_2$. Puisque $\mathcal{W}_{i-1}(\mathcal{Q}_1,\ R)$ et $rew(Q'_1,\mathcal{R})$ sont tous deux inclus dans $\mathcal{W}_i(\mathcal{Q}_1,\mathcal{R})$, on peut conclure. □

Pour faciliter la preuve de la correction de l'algorithme 1, nous aurons besoin d'une autre version "technique" de ce lemme qui nous permettra de prouver dans l'algorithme 1 qu'il est suffisant de calculer les réécritures de $\mathcal{Q}_E$ pour obtenir une couverture de $\mathcal{W}_\infty(Q,\mathcal{R})$.

Lemme 2 *Soit rew un opérateur de réécriture élagable, et soit $\mathcal{Q}_1$ et $\mathcal{Q}_2$ deux ensembles de requêtes. Pour tout ensemble de règles $\mathcal{R}$, si $(\mathcal{Q}_1 \cup \mathcal{Q}_2) \geq rew(\mathcal{Q}_1,\mathcal{R})$, alors $(\mathcal{Q}_1 \cup \mathcal{W}_\infty(\mathcal{Q}_2,\mathcal{R})) \geq \mathcal{W}_\infty(\mathcal{Q}_1 \cup \mathcal{Q}_2,\mathcal{R})$.*

Preuve : On prouve pas induction sur i que $\mathcal{Q}_1 \cup \mathcal{W}_i(\mathcal{Q}_2, \mathcal{R}) \geq rew_i(\mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{R})$.
Pour $i = 0$, $rew_0(\mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{R}) = \mathcal{Q}_1 \cup \mathcal{Q}_2 = \mathcal{Q}_1 \cup \mathcal{W}_0(\mathcal{Q}_2, \mathcal{R})$.
Pour $i > 0$, pour tout $Q_i \in rew_i(\mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{R})$, il existe une requête $Q_{i-1} \in rew_{i-1}(\mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{R})$ telle que $Q_i \in rew(Q_{i-1}, \mathcal{R})$. Selon l'hypothèse d'induction, il existe $Q'_{i-1} \in \mathcal{Q}_1 \cup \mathcal{W}_{i-1}(\mathcal{Q}_2, \mathcal{R})$ telle que $Q'_{i-1} \geq Q_{i-1}$. Puisque *rew* est élagable, par définition, soit $Q'_{i-1} \geq Q_i$ et donc $\mathcal{Q}_1 \cup \mathcal{W}_{i-1}(\mathcal{Q}_2, \mathcal{R}) \geq \{Q_i\}$ soit il existe $Q'_i \in rew(Q'_{i-1}, \mathcal{R})$ telle que $Q'_i \geq Q_i$. Ensuite, il y a deux possibilités :

- soit $Q'_{i-1} \in \mathcal{Q}_1$: puisque $\mathcal{Q}_1 \cup \mathcal{Q}_2 \geq rew(\mathcal{Q}_1, \mathcal{R})$, on a $\mathcal{Q}_1 \cup \mathcal{Q}_2 \geq \{Q'_i\}$ et donc $\mathcal{Q}_1 \cup \mathcal{W}_i(\mathcal{Q}_2, \mathcal{R}) \geq \{Q'_i\}$;
- soit $Q'_{i-1} \in \mathcal{W}_{i-1}(\mathcal{Q}_2, \mathcal{R})$: alors $Q'_i \in \mathcal{W}_i(\mathcal{Q}_2, \mathcal{R})$.

□

Dans les preuves qui suivent, on notera $\mathcal{Q}_F^i$, respectivement $\mathcal{Q}_E^i$, la valeur de l'ensemble $\mathcal{Q}_F$, respectivement $\mathcal{Q}_E$, à la $i^{\text{ème}}$ boucle de l'algorithme 1.

La propriété suivante présente les invariants de boucle de l'algorithme qui sont utilisés pour prouver sa correction.

Propriété 1 (Invariants de l'algorithme 1) *Soit rew un opérateur de réécriture. Après chaque itération de la boucle "tant que" de l'algorithme 1, les propriétés suivantes sont vérifiées :*

1. $\mathcal{Q}_E \subseteq \mathcal{Q}_F \subseteq \mathcal{W}_\infty(Q, \mathcal{R})$;
2. $\mathcal{Q}_F \geq rew(\mathcal{Q}_F \setminus \mathcal{Q}_E, \mathcal{R})$;
3. *si rew est élagable alors* $(\mathcal{Q}_F \cup \mathcal{W}_\infty(\mathcal{Q}_E, \mathcal{R})) \geq \mathcal{W}_\infty(Q, \mathcal{R})$;
4. *pour tout* $Q, Q' \in \mathcal{Q}_F$ *distincts,* $Q \not\geq Q'$ *et* $Q' \not\geq Q$.

Preuve : Les invariants sont prouvés par récurrence sur le nombre d'itérations de la boucle “tant que”.

Invariant 1. $\mathcal{Q}_E \subseteq \mathcal{Q}_F \subseteq \mathcal{W}_\infty(Q, \mathcal{R})$.

base : $\mathcal{Q}_E^0 = \mathcal{Q}_F^0 = \{Q\} = \mathcal{W}_0(Q, \mathcal{R}) \subseteq \mathcal{W}_\infty(Q, \mathcal{R})$.

récurrence : Par construction, $\mathcal{Q}_E^i \subseteq \mathcal{Q}_F^i$ et $\mathcal{Q}_F^i \subseteq \mathcal{Q}_F^{i-1} \cup rew(\mathcal{Q}_E^{i-1}, \mathcal{R})$. Pour toute requête $Q' \in \mathcal{Q}_F^i$ on a : soit $Q' \in \mathcal{Q}_F^{i-1}$ et donc par hypothèse de récurrence $Q' \in \mathcal{W}_\infty(Q, \mathcal{R})$; soit $Q' \in rew(\mathcal{Q}_E^{i-1}, \mathcal{R})$ et donc par hypothèse de récurrence on a $\mathcal{Q}_E^{i-1} \subseteq \mathcal{W}_\infty(Q, \mathcal{R})$, ce qui implique $Q' \in \mathcal{W}_\infty(Q, \mathcal{R})$.

Invariant 2. $\mathcal{Q}_F \geq rew(\mathcal{Q}_F \setminus \mathcal{Q}_E, \mathcal{R})$.

base : $rew(\mathcal{Q}_F^0 \setminus \mathcal{Q}_E^0, \mathcal{R}) = rew(\emptyset, \mathcal{R}) = \emptyset$ et donc tout ensemble le couvre.

récurrence : Par construction, $\mathcal{Q}_F^i \geq \mathcal{Q}_F^{i-1} \cup rew(\mathcal{Q}_E^{i-1}, \mathcal{R})$; par hypothèse de récurrence $\mathcal{Q}_F^{i-1} \geq rew(\mathcal{Q}_F^{i-1} \setminus \mathcal{Q}_E^{i-1}, \mathcal{R})$, donc on a $\mathcal{Q}_F^i \geq rew(\mathcal{Q}_F^{i-1} \setminus \mathcal{Q}_E^{i-1}, \mathcal{R}) \cup rew(\mathcal{Q}_E^{i-1}, \mathcal{R}) = rew(\mathcal{Q}_F^{i-1}, \mathcal{R})$ on a donc *(i)* $\mathcal{Q}_F^i \geq rew(\mathcal{Q}_F^{i-1}, \mathcal{R})$. De plus, par construction, $\mathcal{Q}_E^i = \mathcal{Q}_F^i \setminus \mathcal{Q}_F^{i-1}$; donc $\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i \subseteq \mathcal{Q}_F^{i-1}$ et donc par définition de *rew* sur un ensemble de requête *(ii)* $rew(\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i, \mathcal{R}) \subseteq rew(\mathcal{Q}_F^{i-1}, \mathcal{R})$. Ainsi par *(i)* et *(ii)* on a $\mathcal{Q}_F^i \geq rew(\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i, \mathcal{R})$.

Invariant 3. Si *rew* est élagable alors $(\mathcal{Q}_F \cup \mathcal{W}_\infty(\mathcal{Q}_E, \mathcal{R})) \geq \mathcal{W}_\infty(Q, \mathcal{R})$.

base : $(\mathcal{Q}_F^0 \cup \mathcal{W}_\infty(\mathcal{Q}_E^0, \mathcal{R})) = (\{Q\} \cup \mathcal{W}_\infty(\{Q\}, \mathcal{R})) = \mathcal{W}_\infty(Q, \mathcal{R})$.

récurrence : On montre d'abord que *(i)* : $(\mathcal{Q}_F^i \cup \mathcal{W}_\infty(\mathcal{Q}_E^i, \mathcal{R})) \geq \mathcal{W}_\infty(\mathcal{Q}_F^i, \mathcal{R})$, ensuite on montre par induction que *(ii)* : $\mathcal{W}_\infty(\mathcal{Q}_F^i, \mathcal{R}) \geq \mathcal{W}_\infty(Q, \mathcal{R})$:

(i) Par construction, $\mathcal{Q}_E^i \subseteq \mathcal{Q}_F^i$, donc $(\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i) \cup \mathcal{Q}_E^i = \mathcal{Q}_F^i$, et grâce à l'invariant 2, on a $(\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i) \cup \mathcal{Q}_E^i \geq rew(\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i, \mathcal{R})$. Le lemme 2 implique que $((\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i) \cup \mathcal{W}_\infty(\mathcal{Q}_E^i, \mathcal{R})) \geq \mathcal{W}_\infty((\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i) \cup \mathcal{Q}_E^i, \mathcal{R})$ et on peut conclure puisque $\mathcal{Q}_F^i = (\mathcal{Q}_F^i \setminus \mathcal{Q}_E^i) \cup \mathcal{Q}_E^i$.

(ii) Par construction, on a $\mathcal{Q}_F^i \geq \mathcal{Q}_F^{i-1} \cup rew(\mathcal{Q}_E^{i-1}, \mathcal{R})$; donc, grâce au lemme 1, on a $\mathcal{W}_\infty(\mathcal{Q}_F^i, \mathcal{R}) \geq \mathcal{W}_\infty(\mathcal{Q}_F^{i-1} \cup rew(\mathcal{Q}_E^{i-1}, \mathcal{R}), \mathcal{R}) = \mathcal{W}_\infty(\mathcal{Q}_F^{i-1}, \mathcal{R}) \cup \mathcal{W}_\infty(rew(\mathcal{Q}_E^{i-1}, \mathcal{R}), \mathcal{R})$. De plus, $\mathcal{Q}_E^{i-1} \subseteq \mathcal{Q}_F^{i-1} \subseteq \mathcal{W}_\infty(\mathcal{Q}_F^{i-1}, \mathcal{R})$, donc $\mathcal{W}_\infty(\mathcal{Q}_F^i, \mathcal{R}) \geq \mathcal{Q}_F^{i-1} \cup \mathcal{Q}_E^{i-1} \cup \mathcal{W}_\infty(rew(\mathcal{Q}_E^{i-1}, \mathcal{R}), \mathcal{R}) = \mathcal{Q}_F^{i-1} \cup \mathcal{W}_\infty(\mathcal{Q}_E^{i-1}, \mathcal{R})$. En utilisant *(i)*, on a $\mathcal{W}_\infty(\mathcal{Q}_F^i, \mathcal{R}) \geq \mathcal{W}_\infty(\mathcal{Q}_F^{i-1}, \mathcal{R})$ et on peut conclure avec l'hypothèse de récurrence.

Invariant 4. pour tout $Q, Q' \in \mathcal{Q}_F$ distincts, $Q \not\geq Q'$ et $Q' \not\geq Q$.

Trivialement satisfait par la fonction *cover*.

□

Voici ensuite une preuve de l'arrêt de l'algorithme lorsque l'ensemble des réécritures de la requête avec l'ensemble de règles et un opérateur élagable admet une couverture finie.

Propriété 2 *Soit rew un opérateur de réécriture, $\mathcal{R}$ un ensemble de règles et Q une requête. Si $W_\infty(Q, \mathcal{R})$ admet une couverture finie et rew est élagable alors l'algorithme 1 s'arrête.*

Preuve : Soit $\mathcal{Q}$ une couverture finie de $\mathcal{W}_\infty(Q, \mathcal{R})$ et soit m le plus grand k d'une k-réécriture dans $\mathcal{Q}$. On a donc $\mathcal{W}_m(Q, \mathcal{R}) \geq \mathcal{Q} \geq \mathcal{W}_\infty(Q, \mathcal{R})$. Puisque *rew* est élagable, on a $\mathcal{Q}_F^i \geq \mathcal{W}_i(Q, \mathcal{R})$ pour tout $i \geq 0$ (peut être prouvé par une simple induction sur i) donc $\mathcal{Q}_F^m \geq \mathcal{W}_\infty(Q, \mathcal{R})$. Comme $rew(\mathcal{Q}_E^m, \mathcal{R})$ est couvert par $\mathcal{Q}_F^m$, et puisque les requêtes déjà explorées sont choisies en premier pour le calcul de *cover*, on a $\mathcal{Q}_E^{m+1} = \emptyset$. Ainsi, l'algorithme 1 s'arrête. □

Le théorème qui suit garantit que, si elle existe, l'algorithme calcule bien la couverture finie de l'ensemble de réécritures d'une requête avec un ensemble de règles et un opérateur élagable.

Théorème 4 *Soit rew un opérateur de réécriture, $\mathcal{R}$ un ensemble de règles et Q une requête. Si $W_\infty(Q, \mathcal{R})$ admet une couverture finie et rew est élagable alors l'algorithme 1 calcule cette couverture (à une équivalence de requêtes près).*

Preuve : Grâce à la propriété 2, on sait que l'algorithme 1 s'arrête. Grâce à l'invariant 3, $(\mathcal{Q}_F^f \cup \mathcal{W}_\infty(\mathcal{Q}_E^f, \mathcal{R})) \geq \mathcal{W}_\infty(Q,\ R)$ où $\mathcal{Q}_F^f$ et $\mathcal{Q}_E^f$ représentent les valeurs finales de $\mathcal{Q}_F$ et $\mathcal{Q}_E$ dans l'algorithme 1. Puisque $\mathcal{Q}_E^f = \emptyset$ et l'algorithme 1 s'arrête, on a $\mathcal{Q}_F^f \geq \mathcal{W}_\infty(Q, \mathcal{R})$. Grâce aux invariants 1 et 4, on peut conclure que $\mathcal{Q}_F^f$ est une couverture de $\mathcal{W}_\infty(Q, \mathcal{R})$. □

3.2.2 Adéquation et complétude

Nous avons prouvé que si l'opérateur de réécriture est élagable, l'algorithme 1 produit une couverture finie de $W_\infty(Q, \mathcal{R})$ si elle existe. Nous allons maintenant définir des notions d'adéquation et de complétude d'un opérateur de réécriture qui garantissent que la couverture calculée est un ensemble de réécritures adéquat et complet de Q selon $\mathcal{R}$.

Définition 3.9 *Soit* rew *un opérateur de réécriture.* rew *est adéquat si pour tout ensemble de règles $\mathcal{R}$, toute requête Q, pour toute réécriture $Q' \in$ rew$(Q, \mathcal{R})$, pour tout fait F, $F \models Q'$ implique $(F, \mathcal{R}) \models Q$.* rew *est complet si pour tout ensemble de règles $\mathcal{R}$, toute requête Q, pour tout fait F tel que $(F, \mathcal{R}) \models Q$, il existe une réécriture $Q' \in \mathcal{W}_\infty(Q, \mathcal{R})$ telle que $F \models Q'$.*

En s'appuyant sur ces conditions, le prochain théorème assure que l'algorithme 1 calcule un ensemble fini minimal, adéquat et complet de réécritures d'une requête avec un ensemble de règles *fus*.

Théorème 5 *Si* rew *est un opérateur de réécriture adéquat, complet et élagable, et si $\mathcal{R}$ est un ensemble de règles à unification finie (*fus*), alors pour toute requête Q, l'algorithme 1 calcule un ensemble (fini) de cardinalité minimale, adéquat et complet de réécritures de Q avec $\mathcal{R}$.*

Preuve : Si $\mathcal{R}$ est *fus* et si *rew* est un opérateur adéquat et complet alors $\mathcal{W}_\infty(Q, \mathcal{R})$ possède une couverture finie. Grâce au Théorème 4 et au fait que *rew* soit élagable on sait que l'algorithme calculera cette couverture qui est par définition minimale au sens de l'inclusion et par le théorème 3, cette couverture est également de taille minimale. Si *rew* est un opérateur adéquat et complet, alors par définition $\mathcal{W}_\infty(Q, \mathcal{R})$ est adéquat et complet selon Q et $\mathcal{R}$ et donc sa couverture aussi. □

Dans le chapitre suivant, nous allons définir des opérateurs adéquats, complets et élagables qui calculent efficacement l'ensemble de réécritures cible.

Chapitre 4

Une famille d'opérateurs de réécriture en union de requêtes conjonctives

Ce chapitre présente une famille d'opérateurs de réécriture qui permettent de calculer une réécriture sous la forme d'une union de requêtes conjonctives. Ces opérateurs satisfont les propriétés d'adéquation, de complétude et d'élagabilité présentées dans le chapitre précédent. Le chapitre finit par une section présentant une perspective d'amélioration de ces opérateurs.

4.1 Unification en présence de variables existentielles

Les opérateurs présentés dans ce chapitre s'appuient sur les notions de *pièce* et d'*unificateur par pièce* initialement développés pour les graphes conceptuels ([Salvat and Mugnier, 1996]) et plus tard repris pour les règles existentielles ([Baget et al., 2009]). Ces notions ont été reformulées et adaptées afin de mieux correspondre à nos besoins, mais l'essence reste la même. Son principe de base, couramment utilisé en programmation logique, est d'effectuer une unification entre une conclusion de règle et une partie de la requête et de remplacer la partie de requête unifiée par l'hypothèse de la règle. Voici un exemple simple et intuitif pour illustrer ce principe.

Exemple 26 *Nous disposons d'une requête $Q = animal(x)$ qui interroge la présence d'un animal dans une base de faits et d'une règle $R = chat(y) \rightarrow animal(y)$ qui code l'information que tout chat est un animal. L'atome $animal(x)$ de la requête Q peut être unifié avec l'atome $animal(y)$ de la conclusion de la règle R par l'unificateur $u = \{(y, x)\}$. L'atome unifié de la requête est ensuite remplacé par l'hypothèse de la règle sur laquelle est appliqué l'unificateur. La réécriture qui découle de cette unification*

avec u est $Q' = chat(x)$, en effet, si dans la base de fait il y a un chat alors, grâce à la règle R, nous savons qu'il y a aussi un animal. L'ensemble de réécritures obtenu est $\{Q, Q'\}$.

L'opération clé de ce mécanisme est l'unification entre la conclusion de la règle et une partie de la requête. En programmation logique, les règles ont une tête atomique sans variable existentielle, l'unification est simple. Dans le cas des règles existentielles où la conclusion est plus complexe, l'unification l'est aussi. Voici sur un exemple le problème rencontré lorsque l'on effectue une réécriture avec une unification simple sur des règles existentielles.

Exemple 27 *$Q = partenaire(Alice, z) \wedge dentiste(z)$, cette requête interroge la base de connaissances pour savoir si Alice a un partenaire qui est dentiste.*
Soit le fait $F = pompier(Alice) \wedge dentiste(Bob)$ et la règle $R = pompier(x) \rightarrow partenaire(x, y)$. Cette règle indique que, les pompiers travaillant en binôme, tout pompier a un partenaire.
L'atome de la requête $partenaire(Alice, z)$ et celui de la conclusion de R $partenaire(x, y)$ sont unifiables par $\{(x, Alice), (y, z)\}$. Pourtant si on effectue la réécriture selon cette unification, on obtient $Q' = pompier(Alice) \wedge dentiste(z)$. On voit bien que cette réécriture n'est pas adéquate car elle se déduit de F alors que la requête initiale ne se séduit pas de (F, R). En effet, le lien entre les termes Alice et z a été perdu.

Pour conserver l'adéquation des réécritures, des contraintes supplémentaires doivent être ajoutées à l'unification. Elles assurent le fait que lorsqu'une occurrence d'une variable disparaît lors de la réécriture (c'est-à-dire qu'elle est unifiée à une variable existentielle) toutes les autres occurrences de cette variable disparaissent aussi (c'est-à-dire que toutes les autres occurrences apparaissent dans la partie enlevée donc unifiée)

4.2 Substitution et partition

En plus d'ajouter des contraintes supplémentaires à l'unification, nous modifions aussi l'outil utilisé pour l'unification. En effet, les substitutions ont l'inconvénient d'imposer un "sens", c'est-à-dire que lorsque deux variables doivent être unifiées, l'une est substituée par l'autre. Ce sens de substitution arbitraire complique la comparaison des substitutions et des images qu'elles génèrent. Par exemple, dans l'exemple suivant, deux substitutions différentes génèrent deux réécritures trivialement équivalentes. Dans un souci de simplicité, nous aimerions que ces deux unifications soient confondues.

Exemple 28 *Soit $A = p(x, a)$ et $B = p(y, z)$ deux atomes à unifier. Soit $u_1 = \{(z, a), (x, y)\}$ et $u_2 = \{(z, a), (y, x)\}$ deux unificateurs possibles donnant respectivement $p(y, a)$ et $p(x, a)$ comme image, ces deux atomes sont équivalents.*

Étant donné que seul nous intéresse le fait de savoir quels sont les termes unifiés et non le sens dans lequel on va les substituer, nous allons remplacer les unificateurs logiques par des partitions sur les termes. A chaque substitution u de l'ensemble de variables A dans l'ensemble de termes B peut être associée une partition P sur $A \cup B$ telle que deux termes sont dans une même classe de P si et seulement s'ils sont unifiés par u. Plus exactement, nous considérons la classe d'équivalence de la fermeture transitive, symétrique et réflexive de la relation $\sim : t \sim t'$ si $u(t) = t'$. Les partitions associées à des substitutions sont appelées admissibles et on peut les définir de la manière suivante :

Définition 4.1 (Partition admissible) *Soit une partition P sur un ensemble de termes, P est* admissible *si aucune de ses classes ne contient deux constantes.*

A l'inverse, chaque partition admissible peut être associée à une substitution unique u qu'on appelle *substitution associée* à la partition.

Définition 4.2 (Substitution associée à une partition) *Soit P une partition sur un ensemble de termes E et un ordre total sur cet ensemble E tel que les constantes sont plus petites que les variables. Pour chaque classe $\{t_1, \ldots, t_n\}$ de P, on choisit le* représentant *t_i le plus petit de la classe ; on fixe $u(t_j) = t_i$. On appelle u la* substitution associée *à P.*

Pour ne pas avoir à faire référence à la substitution associée à une partition à chaque fois que l'on veut faire une réécriture, on définira directement l'image par une partition comme étant l'image par la substitution associée à la partition.

Définition 4.3 (Image par une partition admissible) *Soit t un terme, P une partition admissible sur un ensemble de termes et u sa substitution associée. L'image de t par P est l'image de t par u, $P(t) = u(t)$.*

La définition précédente peut directement être étendue à un atome et un fait.

Exemple 29 *Soit $P = \{\{x, y, z\}, \{u, v, a\}, \{t\}\}$ une partition sur un ensemble de termes. Cette partition est admissible car aucune classe ne contient deux constantes. L'ordre lexicographique sur les termes, permet de choisir en priorité les constantes, et sera utilisé dans le reste du document. On choisit x comme représentant de la classe $\{x, y, z\}$, a comme représentant de la classe $\{u, v, a\}$ car a est une constante donc à choisir en priorité, t est le représentant de sa classe, il en découle que la substitution associée à P est $\{(y, x), (z, x), (u, a), (v, a)\}$.*

L'ensemble des partitions sur un ensemble peut être structuré en un treillis par la relation “plus fine”.

Définition 4.4 (Partition plus fine) *Soit P_1 et P_2 deux partitions sur un ensemble E. P_1 est* plus fine *que P_2, notée $P_1 \geq P_2$* [1] *si toute classe de P_1 est incluse dans une classe de P_2.*

Les deux définitions techniques ci-dessous ne seront utiles que dans les preuves au cours de ce chapitre. Une substitution préservante est une substitution des variables d'une partition dans les termes d'une autre partition. Elle est telle que si deux éléments sont dans une même classe dans la partition de départ alors leurs images par la substitution sont dans une même classe dans la partition d'arrivée.

Définition 4.5 (Substitution préservante) *Soit P_1 et P_2 deux partitions admissibles sur E_1 et E_2 respectivement. Soit f une substitution des variables de E_1 dans les termes de E_2 préservant les classes de P_1 vers P_2, c'est-à-dire que $f(P_1) \geq P_2$. On dira que f est une* substitution préservante *de P_1 dans P_2.*

La substitution complémentaire est la substitution qui "manque" à la substitution associée à la partition de départ pour obtenir la substitution associée à la partition d'arrivée, lorsque les deux partitions sont liées par une substitution préservante.

Définition 4.6 (Substitution complémentaire) *Soit P_1 et P_2 deux partitions admissibles sur E_1 et E_2 respectivement, et f une substitution préservante de P_1 dans P_2. On appelle* substitution complémentaire *de P_1 dans P_2 selon f la substitution s qui, à chaque représentant d'une classe P de P_1 associe le représentant de la classe de P_2 qui contient les images par f des éléments de P, c'est-à-dire que pour tout $t \in E_1$, $s(P_1(t)) = P_2(f(t))$.*

Exemple 30 *Soit $E_1 = \{v_1, w_1, x_1, y_1, z_1\}$, $E_2 = \{u_2, v_2, w_2, x_2, y_2, z_2, a\}$ et f la substitution qui à chaque terme t_1 de E_1 associe le terme t_2 de E_2. Soit $P_1 = \{\{v_1, w_1\}, \{x_1, y_1\}, \{z_1\}\}$ et $P_2 = \{\{v_2, w_2, a\}, \{u_2, x_2, y_2, z_2\}\}$. Soit $s_1 = \{(w_1, v_1), (y_1, x_1)\}$ la substitution associée à P_1 et $s_2 = \{(v_2, a), (w_2, a), (x_2, u_2), (y_2, u_2), (z_2, u_2)\}$ la substitution associée à P_2. f est une substitution préservante de P_1 dans P_2. La substitution complémentaire de P_1 dans P_2 selon f est $s = \{(v_1, a), (x_1, u_2), (z_1, u_2)\}$.*

On peut remarquer que lorsqu'une substitution préservante de P_1 dans P_2 est telle que $P_2 = f(P_1)$, la substitution complémentaire de P_1 dans P_2 est une bijection. En effet, chaque classe de P_2 contient les images d'exactement une classe de P_1 donc chaque représentant d'une classe de P_2 est l'image d'exactement un représentant d'une classe de P_1.

Appliquée sur un fait, une partition plus fine produit une image plus générale.

1. Habituellement, la relation "plus fine" est notée par $\leq$ mais nous utilisons la convention contraire car elle est plus en accord avec les substitutions et le pré-ordre $\geq$ sur les requêtes

Propriété 3 *Soit P_1 et P_2 deux partitions admissibles sur un même ensemble E telles que $P_1 \geq P_2$. Pour tout fait F, il existe une substitution s telle que $s(P_1(F)) = P_2(F)$ donc $P_1(F) \geq P_2(F)$.*

Preuve : Par hypothèse, on sait que $P_1 \geq P_2$ donc toute classe de P_1 est contenue dans une classe de P_2. Soit id la substitution qui à tout terme associe lui même, id préserve les classes de P_1 vers P_2. Soit s la substitution complémentaire de P_1 dans P_2 selon id, on a $s(P_1(F)) = P_2(id(F))$ donc $s(P_1(F)) = P_2(F)$. □

Les partitions peuvent être associées grâce à la jointure.

Définition 4.7 (Jointure de partitions) *Soit $\mathcal{P}$ un ensemble de partitions, la* jointure *de cet ensemble, notée $join(\mathcal{P})$, est obtenue en faisant l'union de toutes leurs classes non-disjointes.*

La jointure de deux partitions admissibles peut ne pas être admissible. Si leur jointure est admissible, on dira que les partitions sont *compatibles*. Voici un exemple pour illustrer ces notions.

Exemple 31 *Soit $P_1 = \{\{x, y\}, \{z, a\}\}$, $P_2 = \{\{x, u\}, \{z, b\}\}$ et $P_3 = \{\{x, u\}, \{y, b\}\}$. La jointure de P_1 et P_2 est $\{\{x, y, u\}, \{z, a, b\}\}$ elle n'est pas admissible car elle contient deux constantes a et b dans une même classe, donc P_1 et P_2 ne sont pas compatibles. En revanche la jointure de P_1 et P_3 $\{\{x, y, u, b\}, \{z, a\}\}$ est admissible donc P_1 et P_3 sont compatibles.*

4.3 Unificateur par pièce

Comme nous l'avons vu dans la première section de ce chapitre, afin de conserver l'adéquation des réécritures, il faut ajouter des contraintes supplémentaires à l'unification. Ces contraintes permettent d'assurer que toutes les occurrences des termes qui disparaissent lors de la réécriture (qui sont unifiées avec des variables existentielles) appartiennent à la partie unifiée. Ces termes particuliers sont appelés variables liantes, et un ensemble d'atomes minimal qui contient aucune ou toutes les occurrences d'une même variable liante est une pièce.

Définition 4.8 (Pièce [Baget et al., 2011a]) *Soit Q un ensemble d'atomes et X un sous ensemble de $\text{var}(Q)$ dont les éléments sont nommés variables liantes. Une* pièce *de Q selon X est un sous-ensemble P_c de Q minimal et non vide tel que pour chaque pair d'atomes A et A' de Q, si $A \in P_c$ et que A et A' partagent une variable de X, c'est-à-dire $\text{var}(A) \cap \text{var}(A') \cap X \neq \emptyset$, alors $A' \in P_c$.*

Si on considère une pièce dans la conclusion d'un règle R selon l'ensemble des variables existentielles de R, elle peut être vue comme une "unité" de connaissances apportée par une application de R en marche avant. En effet, R peut être décomposée en un ensemble de règles équivalent ayant la même hypothèse et exactement une pièce en conclusion.

Propriété 4 ([Baget et al., 2011a]) *Soit $R = H \to C$ une règle et $Pc_1, \dots Pc_k$ les pièces de C selon les variables existentielles de R. R est équivalente à la conjonction des règles $R_1, \dots R_k$ où $R_i = H \to Pc_i$.*

Les conclusions des règles obtenues ne peuvent pas être découpées davantage en gardant la même sémantique que R si l'hypothèse n'est pas modifiée. En revanche, en modifiant l'hypothèse et en ajoutant des prédicats supplémentaires il est possible de découper une règle quelconque en un ensemble de règles à conclusion atomique équivalent (voir section 7.4). Voici un exemple de la décomposition d'une règle en un ensemble de règles ayant une seule pièce en conclusion.

Exemple 32 *Soit $R = A(x) \to r(x, y) \wedge s(z, x) \wedge t(y)$ une règle. La conclusion de R est composée de deux pièces $\{r(x, y), t(y)\}$ et $\{s(z, x)\}$ selon $\{y, z\}$. Les règles obtenues en décomposant R en règles ayant une seule pièce en conclusion sont : $A(x) \to r(x, y) \wedge t(y)$ et $A(x) \to s(z, x)$*

Nous allons aussi avoir besoin de distinguer les variables qui sont à la limite entre la pièce et le reste de la requête.

Définition 4.9 (Variables séparatrices) *Soit Q un ensemble d'atomes et $Q' \subseteq Q$. On appelle* variables séparatrices *(de Q' dans Q) les variables qui apparaissent à la fois dans Q' et dans $(Q \setminus Q')$, on les note $sep(Q') = var(Q') \cap var(Q \setminus Q')$.*

La réécriture d'une requête avec une règle existentielle sera donc faite au moyen d'un unificateur appelé "unificateur par pièce", qui unifiera non plus seulement des atomes mais un ensemble de pièces. La définition qui suit s'inspire de celle présente dans [Baget et al., 2011a] mais en utilisant la notion de partition au lieu de la notion habituelle de substitution.

Définition 4.10 (Unificateur par pièce, variables liantes) *Soit Q une requête et R une règle. Un* unificateur par pièce *de Q avec R est un triplet $\mu = (Q', C', P)$ où Q' est un sous-ensemble de Q, C' est un sous-ensemble de $concl(R)$ et P est une partition sur les termes de Q' et C' telle que :*

- *P est admissible.*
- *Si une classe de P contient une variable existentielle de R les autres termes qu'elle contient sont des variables non séparatrices de Q'.*
- *$P(C') = P(Q')$.*

Les variables qui apparaissent dans la classe d'une variable existentielle de R sont appelées variables liantes *et notées $glue(\mu)$.*

On peut remarquer que si $\mu = (Q', C', P)$ est un unificateur par pièce, Q' est un ensemble de pièces de Q selon l'ensemble des variables liantes $glue(\mu)$ et $glue(\mu) \cap sep(Q') = \emptyset$. Il s'agit maintenant de définir la réécriture effectuée au moyen de l'unificateur par pièce.

Définition 4.11 (1-réécriture avec un unificateur par pièce, réécriture directe) *Soit Q un fait, R une règle et $\mu = (Q', C', P)$ un unificateur par pièce de Q avec R. La* 1-réécriture *ou* réécriture directe *de Q selon μ et R produit le fait $\beta(Q, R, \mu) = P(hyp(R)) \cup P(Q \setminus Q')$.*

Voici un exemple pour illustrer les différentes notions vues précédemment.

Exemple 33 *Soit $R = q(x) \rightarrow p(x, y)$ et $Q = p(u, v) \wedge p(w, v) \wedge p(w, t) \wedge r(u, w)$. Soit $C' = \{p(x, y)\}$. Il y a trois unificateurs par pièce de Q avec R :*
$\mu_1 = (Q'_1, C', P_1)$ avec $Q'_1 = \{p(u, v), p(w, v)\}$ et $P_1 = \{\{x, u, w\}, \{y, v\}\}$
$\mu_2 = (Q'_2, C', P_2)$ avec $Q'_2 = \{p(w, t)\}$ et $P_2 = \{\{x, w\}, \{y, t\}\}$
$\mu_3 = (Q'_3, C', P_3)$ avec $Q'_3 = \{p(u, v), p(w, v), p(w, t)\}$ et $P_3 = \{\{x, u, w\}, \{y, v, t\}\}$
Notez que Q'_1 et Q'_2 sont composés d'une unique pièce ; $Q'_3 = Q'_1 \cup Q'_2$ et P_3 est la jointure de P_1 et P_2.

Les 1*-réécritures de Q avec μ_1, respectivement μ_2, sont $q(u) \wedge p(u, t) \wedge r(u, u)$, respectivement $p(u, v) \wedge p(w, v) \wedge q(w) \wedge r(u, w)$. La* 1*-réécriture de Q avec μ_3 est $q(u) \wedge r(u, u)$.*

Comme précisé dans la section 2.2, pour simplifier, nous nous sommes concentrés sur les requêtes conjonctives booléennes. Pour les rendre applicables à des requêtes non booléennes, il faudrait étendre les définitions proposées, pour traiter les variables libres d'une manière particulière. Cependant, ne considérer que des requêtes booléennes n'est pas une restriction car il existe une transformation très simple d'une requête non booléenne à une requête booléenne qui assure un traitement correct de la requête d'origine. Soit Q une requête conjonctive non booléenne avec des variables libres $x_1, \ldots, x_k$, on peut traduire Q en une requête booléenne Q' en lui ajoutant un atome $ans(x_1, \ldots, x_k)$, où *ans* est un prédicat spécial n'apparaissant pas dans la base de connaissances, et en fermant existentiellement la formule obtenue. Cette technique assure que si on réécrit Q' en une UCQ Q'', puis que l'on enlève de Q'' tous les prédicats *ans*, on obtient l'UCQ que l'on aurait obtenue en réécrivant Q.

Exemple 34 *Soit $Q = \exists x\ p(x, y)$ et $R = A(u) \rightarrow p(u, v)$. La variable y de Q est libre, l'ensemble des réponses à la requête est l'ensemble des instanciations de cette requête dans la base de faits. Ainsi, cette variable ne doit pas disparaître de la requête. Or si nous effectuons une réécriture de Q avec la règle R, y est unifiée avec la variable existentielle v au dessus et donc disparaîtra lors de la réécriture. La traduction proposée au dessus résout ce problème, la requête conjonctive booléenne associée à Q est $Q' = \exists x \exists y\ ans(y) \wedge p(x, y)$. L'important étant que le prédicat ans n'apparaisse jamais dans la base de connaissances. Ainsi, puisque le prédicat ans*

n'existe nulle part ailleurs, l'atome $ans(y)$ ne sera jamais réécrit et sa variable sera traitée de manière adéquate puisque étant partagée elle ne sera jamais unifiée avec une variable existentielle et donc ne disparaîtra pas.

Pour la suite, on appellera rew_β l'opérateur de réécriture qui à une requête associe toutes les 1-réécritures de cette requête calculables au moyen d'un unificateur par pièce. L'opérateur rew_β est connu pour être adéquat et complet.

Théorème 6 ([Salvat and Mugnier, 1996, Baget et al., 2011a]) *Soit $\mathcal{K} = (\mathcal{F}, \mathcal{R})$ une base de connaissances et Q une requête. $(\mathcal{F}, \mathcal{R}) \models Q$ si et seulement s'il existe une k-réécriture Q' de Q selon $\mathcal{R}$ et rew_β telle que $\mathcal{F} \models Q'$.*

Le lemme suivant assure que cet opérateur est aussi élagable.

Lemme 3 *Soit R une règle, Q_1 et Q_2 deux requêtes telles que $Q_1 \geq Q_2$. Pour tout unificateur par pièce μ_2 de Q_2 avec R,*

(i) soit $Q_1 \geq \beta(Q_2, R, \mu_2)$

(ii) soit il existe un unificateur par pièce μ_1 de Q_1 avec R tel que $\beta(Q_1, R, \mu_1) \geq \beta(Q_2, R, \mu_2)$.

Preuve : Soit h un homomorphisme de Q_1 dans Q_2, soit $\mu_2 = (Q'_2, C'_2, P_2)$ un unificateur par pièce de Q_2 avec R. On considère deux cas :

(i) Si $h(Q_1) \subseteq Q_2 \setminus Q'_2$, alors $P_2 \circ h$ est un homomorphisme de Q_1 dans $P_2(Q_2 \setminus Q'_2) \subseteq \beta(Q_2, R, \mu_2)$ donc $Q_1 \geq \beta(Q_2, R, \mu_2)$.

(ii) Sinon soit Q'_1 le sous ensemble non vide de Q_1 qui se projette sur Q'_2 par h, c'est-à-dire $h(Q'_1) \subseteq Q'_2$ et C'_1 le sous ensemble de C'_2 dont l'image par P_2 est unifiée avec $P_2(h(Q'_1))$, c'est-à-dire $P_2(C'_1) = P_2(h(Q'_1))$. Soit P'_1 une partition telle que $h(P'_1) = P_2$, soit P_1 la restriction de P'_1 aux $term(Q'_1) \cup term(C'_1)$, h est une substitution préservante de P_1 dans P_2. Par construction, $\mu_1 = (Q'_1, C'_1, P_1)$ est un unificateur par pièce, il vérifie toutes les conditions de la définition 4.10 puisque P_2 les vérifie. Soit s la substitution complémentaire de P_1 dans P_2 selon h, on sait que pour tout terme t de P_1, $s(P_1(t)) = P_2(h(t))$.

On construit maintenant une substitution h' de $var(\beta(Q_1, R, \mu_1))$ dans $term(\beta(Q_2, R, \mu_2))$, avec trois possibilités selon la partie de $\beta(Q_1, R, \mu_1)$ où la variable apparaît, c'est-à-dire dans Q_1 mais pas Q'_1, dans $hyp(R)$ mais pas C'_1 ou dans la partie restante qui correspond aux images de $sep(Q'_1)$ par P_1 :

1. si $x \in var(Q_1) \setminus var(Q'_1)$, $h'(x) = h(x)$
2. si $x \in hyp(R) \setminus var(C'_1)$, $h'(x) = P_2(x)$
3. si $x \in u_1(sep(Q'_1))$ (ou de manière équivalente $x \in P_1(fr(R) \cap var(C'_1))$), $h'(x) = s(x)$

On conclut en montrant que h' est un homomorphisme de $\beta(Q_1, R, \mu_1) = P_1(hyp(R)) \cup P_1(Q_1 \setminus Q'_1)$ dans $\beta(Q_2, R, \mu_2) = P_2(hyp(R)) \cup P_2(Q_2 \setminus Q'_2)$, en deux points :

1. $h'(P_1(hyp(R))) = P_2(hyp(R))$, en effet, pour toute variable x de $hyp(R)$ soit
   - $x \notin var(C'_1)$, donc $h'(P_1(x)) = h'(x) = P_2(x)$ (P_1 n'unifie pas les variables de $Q_1 \setminus Q'_1$),
   - ou $x \in fr(R) \cap var(C'_1)$, donc $h'(P_1(x)) = s(P_1(x)) = P_2(h(x)) = P_2(x)$ (h est une substitution des variables de Q_1).
2. $h'(P_1(Q_1 \setminus Q'_1)) \subseteq P_2(Q_2 \setminus Q'_2)$. On montre que $h'(P_1(Q_1 \setminus Q'_1)) = P_2(h(Q_1 \setminus Q'_1)))$, et puisque $h(Q_1 \setminus Q'_1) \subseteq Q_2 \setminus Q'_2$, on a $h'(P_1(Q_1 \setminus Q'_1)) \subseteq P_2(Q_2 \setminus Q'_2)$. Pour montrer que, $h'(P_1(Q_1 \setminus Q'_1)) = P_2(h(Q_1 \setminus Q'_1)))$, il faut souligner que, pour toute variable x de $Q_1 \setminus Q'_1$ soit :
   - $x \in var(Q'_1)$, alors $h'(P_1(x)) = s(P_1(x)) = P_2(h(x))$
   - ou $x \in var(Q_1) \setminus var(Q'_1)$, alors $h'(P_1(x)) = h'(x) = h(x) = P_2(h(x))$ (P_1 et P_2 n'unifie pas les éléments de $var(Q_1) \setminus var(Q'_1)$ et $h(x) \notin var(Q'_2 \cup C'_2)$).

□

La première possibilité pour calculer un ensemble de réécritures minimal, adéquat et complet est donc d'utiliser les unificateurs par pièce. Malheureusement, deux sources de complexité rendent cette technique peu efficace.

D'abord, le problème de déterminer s'il existe un unificateur par pièce entre une requête et une règle est NP-complet dans le cas général. En effet, si on considère une règle ayant une frontière vide, il y a un unificateur par pièce de Q avec R si et seulement s'il existe un homomorphisme de Q dans $concl(R)$ ce qui est un problème NP-complet.

Ensuite, le nombre d'unificateurs par pièce peut être exponentiel en la taille de la requête. En effet, même en restreignant la conclusion de la règle à un seul atome C, si on considère que chaque atome de la requête Q est unifiable avec C et forme sa propre pièce, il y a au moins $2^{|Q|}$ unificateurs par pièce car il faut considérer tous les sous-ensembles de Q. Le nombre important d'unificateurs par pièce présente un double inconvénient : premièrement cela augmente leur temps de calcul, de plus, chaque unificateur produit une réécriture qu'il faudra ensuite comparer à chaque autre pour ne garder que les plus générales.

Dans un premier temps, nous avons donc cherché à éliminer les unificateurs qui produisent des requêtes trop spécifiques dont nous n'avons pas besoin dans notre ensemble final. Nous avons donc adapté la notion d'upg (unificateur le plus général) aux unificateurs par pièce.

Définition 4.12 (Unificateur par pièce le plus général) *Soit* $\mu_1 = (Q', C', P_1)$ *et* $\mu_2 = (Q', C', P_2)$ *deux unificateurs par pièce sur les mêmes ensembles* Q' *et* C'*. On dit que* μ_1 *est plus général que* μ_2 *noté* $\mu_1 \geq \mu_2$ *si* $P_1 \geq P_2$*. Un unificateur par pièce le plus général est un unificateur* $\mu = (Q', C', P)$ *tel que quelque soit un unificateur* $\mu' = (Q', C', P')$*,* μ *est plus général que* μ'*.*

On peut noter que dans l'exemple 33 tous les unificateurs par pièce sont des unificateurs par pièce les plus généraux. Les unificateurs les plus généraux nous intéressent car un unificateur plus spécifique qu'un autre produit une réécriture plus spécifique que celle produite par le plus général.

Lemme 4 *Soit* Q *une requête,* R *une règle,* $\mu_1 = (Q', C', P_1)$ *et* $\mu_2 = (Q', C', P_2)$ *deux unificateurs par pièce de* Q *avec* R *sur les mêmes ensembles* Q' *et* C' *tels que* $\mu_1 \geq \mu_2$*. Alors* $\beta(Q, R, \mu_1) \geq \beta(Q, R, \mu_2)$*.*

Preuve : On sait que $\beta(Q, R, \mu_1) = P_1(hyp(R)) \cup P_1(Q \setminus Q')$ et $\beta(Q, R, \mu_2) = P_2(hyp(R)) \cup P_2(Q \setminus Q')$. Par la propriété 3 et sa preuve, on sait qu'il existe une substitution s qui est un homomorphisme de $P_1(hyp(R))$ dans $P_2(hyp(R))$ et de $P_1(Q \setminus Q')$ dans $P_2(Q \setminus Q')$. s est donc un homomorphisme de $\beta(Q, R, \mu_1)$ dans $\beta(Q, R, \mu_2)$ donc $\beta(Q, R, \mu_1) \geq \beta(Q, R, \mu_2)$. □

On appelle $rew_{\beta+}$ l'opérateur de réécriture qui a une requête associe toutes les 1-réécritures de cette requête calculables au moyen d'un unificateur par pièce le plus général. Le lemme précédent nous permet de prouver que l'opérateur $\beta+$ est adéquat, complet et élagable.

Théorème 7 *L'opérateur de réécriture* $rew_{\beta+}$ *est adéquat, complet et élagable.*

Preuve : L'adéquation découle directement du fait que les unificateurs par pièce les plus généraux sont des unificateurs par pièce et que l'opérateur rew_β basé sur les unificateurs par pièce est adéquat. La complétude s'appuie sur le lemme 4, qui assure que toute 1-réécriture par l'opérateur rew_β qui n'est pas une 1-réécriture par l'opérateur $rew_{\beta+}$ est subsumée par une 1-réécriture par $rew_{\beta+}$. Soit Q_2 une réécriture produite par un unificateur μ_1 de Q avec R qui n'est pas le plus général, soit μ_1 l'unificateur le plus général tel que $\mu_1 \geq \mu_2$ le lemme assure que $\beta(Q, R, \mu_1) \geq \beta(Q, R, \mu_2)$ or $\beta(Q, R, \mu_1)$ est une 1-réécriture de Q par $rew_{\beta+}$. L'élagabilité vient du fait que le lemme 3 reste valable si on considère seulement des unificateurs les plus généraux. En effet, soit Q_1 et Q_2 deux requêtes telles que $Q_1 \geq Q_2$ et soit Q'_2 une 1-réécriture par un unificateur par pièce μ_2 de Q_2 avec R, le lemme 3 assure que $Q_1 \geq Q'_2$ ou qu'il existe Q'_1 une 1-réécriture par μ_1 un unificateur par pièce de Q_1 avec R telle que $Q'_1 \geq Q'_2$. Le lemme 4 assure que si μ_1 n'est pas le plus général, l'unificateur le plus général produit une réécriture plus générale que Q'_1 et donc a fortiori plus générale que Q'_2. □

4.4 Unificateur mono-pièce

Dans le double but de simplifier le calcul des unificateurs et de diminuer le nombre d'unificateurs à calculer à chaque pas, la seconde idée clé de nos recherches a été de considérer uniquement des *unificateurs mono-pièce*, c'est-à-dire des unificateurs par pièce de la forme $\mu = (Q', C', P)$ où Q' est une unique pièce selon $glue(\mu)$. La restriction aux unificateurs mono-pièce ne conduit pas à manquer des réécritures puisque la réécriture produite en enchaînant des unificateurs mono-pièce est au moins aussi générale que celle produite par l'unificateur multi-pièce correspondant. Voici sur un exemple une illustration de ce principe.

Exemple 35 *Soit* $R = p(x, y) \rightarrow q(x, y)$ *et* $Q = q(u, v) \wedge r(v, w) \wedge q(t, w)$.

Soit $\mu_1 = (Q'_1, C', P_1)$ *un unificateur mono-pièce de* Q *avec* R *où* $Q'_1 = \{q(u, v)\}$, $C' = \{q(x, y)\}$ *et* $P_1 = \{\{u, x\}, \{v, y\}\}$. *Soit* $\mu_2 = (Q'_2, C', P_2)$ *un unificateur mono-pièce de* Q *avec* R' *où* $Q'_2 = \{q(t, w)\}$ *et* $P_2 = \{\{t, x\}, \{w, y\}\}$. *Soit* $\mu_3 = (Q'_1 \cup Q'_2, C', P_3)$ *où* $P_3 = \{\{u, t, x\}, \{v, w, y\}\}$.

Si on applique successivement μ_1 *et* μ_2 *on obtient* $Q_s = p(u, v) \wedge r(v, w) \wedge p(t, w)$. *Si on applique* μ_3, *on obtient* $Q_3 = p(t, v) \wedge r(v, v) \wedge p(t, v)$ *et on constate* Q_s *est plus générale que* Q_3.

Nous montrons d'abord que l'opérateur de réécriture basé sur les unificateurs mono-pièce les plus généraux est adéquat et complet. En revanche, il n'est pas élagable, il s'ensuit que l'ensemble de réécritures produit par l'algorithme 1 avec cet opérateur peut ne pas être complet. Pour retrouver l'élagabilité, nous présentons dans la section suivante un nouvel opérateur exploitant les unificateurs mono-pièce pour calculer des unificateurs mono-pièce agrégés. Cette section présente aussi un algorithme pour calculer l'ensemble des unificateurs mono-pièce les plus généraux entre une requête et une règle.

Définition 4.13 (Unificateur mono-pièce) *Soit* $\mu = (Q', C', P)$ *un unificateur par pièce, on dit que* μ *est mono-pièce si* Q' *est une unique pièce par rapport à* $glue(\mu)$.

On appellera rew_{β_1+} l'opérateur de réécriture qui a une requête associe toutes les réécritures de cette requête calculables au moyen d'un unificateur mono-pièce le plus général. La complétude de l'opérateur rew_{β_1+} s'appuie sur le lemme suivant.

Lemme 5 *Pour tout unificateur par pièce* μ *de* Q *avec* R, *il existe un entier* k *et* Q_s *une* $\leq k$*-réécriture de* Q *selon* R *et* rew_{β_1+} *telle que* $Q_s \geq \beta(Q, R, \mu)$.

Preuve : Pour prouver ce lemme, nous allons construire une séquence $Q = Q_0, Q_1, \ldots, Q_k = Q_s$ telle que $Q_s \geq \beta(Q, R, \mu)$ et que chaque Q_i, $1 \leq i \leq k$, est une 1-réécriture par μ_i un unificateur mono-pièce le plus général de Q_{i-1} avec R_i une copie de R.

Soit $Pc_1, \ldots, Pc_k$ les pièces de Q' selon $\mu = (Q', C', P)$. On construit chaque $\mu_i = (Q'_i, C'_i, P_i)$ de Q_{i-1} avec R_i de la manière suivante (Q'_i sera un sous-ensemble de Q_{i-1}) :

- R_i est une copie de R obtenue en renommant chaque variable par une nouvelle variable avec une substitution de renommage h_i.
- C'_i est l'image par h_i du sous-ensemble de C' unifié par P avec Pc_i.
- P_i est obtenu de la partition $h_i(P)$ en (1) la restreignant au termes de Q'_i et C'_i (2) l'affinant autant que possible en gardant la propriété que $P_i(Q'_i) = P_i(C'_i)$.
- $Q'_1 = Pc_1$ et pour $i > 1$, $Q'_i = P^\circ_{i-1}(Pc_i)$ où P°_{i-1} est la jointure de $P_1, \ldots P_{i-1}$.

Nous supprimons de la séquence les unificateurs inutiles comme indiqué dans l'explication qui suit pour garantir les deux propriétés suivantes :

i) $\forall i,\ P^\circ_{i-1}(Pc_i) \cap P^\circ_{i-1}(Q \setminus Q') = \emptyset$

ii) $\forall i, \forall j > i,\ P^\circ_{i-1}(Pc_i) \cap P^\circ_{i-1}(Pc_j) = \emptyset$

explication de la propriété i) : Si $P^\circ_{i-1}(Pc_i) \cap P^\circ_{i-1}(Q \setminus Q') \neq \emptyset$, nous enlevons μ_i de la séquence car il est inutile puisque $P^\circ_{i-1}(Pc_i) \subseteq P^\circ_{i-1}(Q \setminus Q')$. En effet, soit $a \in P^\circ_{i-1}(Pc_i) \cap P^\circ_{i-1}(Q \setminus Q')$, il existe $b \in Pc_i$ et $b' \in Q \setminus Q'$, $b \neq b'$ tels que $P^\circ_{i-1}(b) = P^\circ_{i-1}(b') = a$, donc $term(b) \subseteq sep(Pc_i)$, donc $\{b\}$ est une pièce selon $glue(\mu)$, donc $Pc_i = \{b\}$. Par conséquent, $P^\circ_{i-1}(Pc_i) = \{a\} \subseteq P^\circ_{i-1}(Q \setminus Q')$ et donc il serait inutile de considérer cet unificateur. La propriété *ii)* s'appuie sur des raisons similaires.

Nous allons maintenant montrer que :

1. μ_i est un unificateur par pièce
2. μ_i est un unificateur par pièce le plus général
3. μ_i est un unificateur mono-pièce

Pour le premier point :

- $Q'_i \subseteq Q_{i-1}$ puisque $\forall i,\ P^\circ_{i-1}(Pc_i) \cap P^\circ_{i-1}(Q \setminus Q') = \emptyset$ et $\forall i, \forall j > i\ P^\circ_{i-1}(Pc_i) \cap P^\circ_{i-1}(Pc_j) = \emptyset$, voir propriétés *i)* et *ii)*.
- $C'_i \subseteq concl(R_i)$ par construction.
- P_i satisfait les conditions de la définition de l'unificateur par pièce puisque P les satisfait et que toute classe de P_i est incluse dans une classe de $h_i(P)$.

Pour le second point, puisque P_i est la partition la plus fine associée à un unificateur par pièce de C'_i et Q'_i, on est sûr que μ_i est un unificateur par pièce le plus général.

Pour le troisième point, il faut remarquer que chaque atome de Q'_i correspond à au moins un atome de Pc_i. Ainsi, si Pc_i est composé d'un unique atome, Q'_i aussi et donc forme une unique pièce. Sinon, Pc_i est une pièce qui contient plus d'un atome ; chaque atome A de Pc_i contient une variable x qui dans P est dans la même classe qu'une variable existentielle y, y vient du sous-ensemble C' unifié par P avec Pc_i. Ainsi, l'atome $P^\circ_{i-1}(A)$ dans Q'_i est tel que la classe de P_i qui contient $P^\circ_{i-1}(x)$ contient aussi la variable existentielle $h_i(y)$. Donc Q'_i forme une pièce unique.

A la fin de la séquence, $Q_k \subseteq P^\circ_k(Q \setminus Q') \cup \bigcup_{j\in 1..k}(P_k(\dots P_j(hyp(R_j))))$. Soit P^h la jointure de tous les $h_i(P)$. Puisque P°_k et P^h sont définis sur le même ensemble et que chaque classe des P_i est incluse dans une classe de $h_i(P)$ donc de P^h, P°_k est plus fine que P^h. Ainsi, par la propriété 3, il existe une substitution s telle que $s(P^\circ_k(Q \setminus Q')) = P^h(Q \setminus Q')$.

Soit h la substitution obtenue en faisant l'union des inverses des h_i, alors $h(P^h(Q \setminus Q')) = P(Q \setminus Q')$ donc $h \circ s$ est un homomorphisme de $P^\circ_k(Q \setminus Q')$ dans $P(Q \setminus Q')$.

Maintenant, nous prouvons que pour $1 \leq j \leq k$, $h(s(P_k(\dots P_j(hyp(R_j))))) = P(hyp(R))$. En effet, $P_k(\dots P_j(hyp(R_j))) = P_k(\dots P_1(hyp(R_j)))$ puisque les termes de $hyp(R_j)$ n'apparaissent pas dans P_i $(i < j)$.

Pour conclure la preuve, on a $h(s(Q_k)) \subseteq P(hyp(R)) \cup P(Q \setminus Q') = \beta(Q, \mu, R)$, donc $h \circ s$ est un homomorphisme de Q_k dans $\beta(Q, \mu, R)$, et $Q_k \geq \beta(Q, \mu, R)$. □

Il nous reste à conclure par le théorème d'adéquation et de complétude suivant.

Théorème 8 *L'opérateur de réécriture* rew_{β_1+} *est adéquat et complet.*

Preuve : L'adéquation découle trivialement du fait que les unificateurs mono-pièce sont des unificateurs par pièce. Pour la complétude, grâce au théorème 6, il nous suffit de montrer par induction sur k la longueur de la séquence de réécritures, que : pour toute k-réécriture Q_r de Q selon rew_β et $\mathcal{R}$ il existe Q_s et i un entier telle que Q_s est une i-réécriture de Q selon rew_{β_1+} et que $Q_s \geq Q_r$.

Pour $k = 0$ la propriété est trivialement satisfaite.
Pour $k \geq 1$, on a $Q_r = \beta(Q'_r, R, \mu)$, avec Q'_r une (k-1)-réécriture de Q selon rew_β. Par hypothèse d'induction, il existe Q'_s une i-réécriture de Q selon rew_{β_1+} telle que $Q'_s \geq Q'_r$. Par le lemme 3, soit $Q'_s \geq Q_r$, soit il existe un unificateur par pièce μ' de Q'_s avec R tel que $\beta(Q'_s, R, \mu') \geq Q_r$. Dans le dernier cas, grâce au lemme 5, il existe Q_s une j-réécriture de Q'_s selon rew_{β_1+} telle que $Q_s \geq \beta(Q'_s, R, \mu') \geq Q_r$. □

Malheureusement, même s'il est adéquat et complet, nous allons voir au travers de quelques exemples que l'opérateur rew_{β_1+} n'est pas élagable et que l'ensemble retourné par l'algorithme 1 peut ne pas être complet. Pour corriger ce problème, on pourrait avoir envie d'enlever le calcul de couverture à chaque étape pour ne la

faire qu'une fois toutes les réécritures calculées mais comme le montre l'exemple 24, l'ensemble des réécritures peut-être infinie alors que la couverture est finie. Dans ces cas là, l'algorithme ne s'arrêterait pas alors qu'il se serait arrêté en calculant la couverture à chaque pas. Une autre solution serait de savoir à quel moment une réécriture qui n'est pas la plus générale doit être gardée car elle est l'unique moyen de produire une réécriture appartenant à la couverture finale. Malheureusement, nous n'avons réussit à concevoir aucun moyen pour le déterminer.

Exemple 36 *Soit* $Q_1 = \{p(y), r(x,y), r(x,a)\}$ *et* $Q_2 = \{p(a), r(z,a)\}$ *deux requêtes, on a* $Q_1 \geq Q_2$. *Soit* $R = s(u,v) \rightarrow r(u,v)$. *Il y a un unificateur mono-pièce* $\mu_2 = (\{r(z,a)\}, \{r(u,v)\}, \{\{z,u\}, \{a,v\}\})$ *de* Q_2 *avec* R *qui produit une requête* $Q'_2 = \{p(a), s(z,a)\}$. *Il y a deux unificateurs mono-pièce de* Q_1 *avec* R $\mu_{11} = (\{r(x,y)\}, \{r(u,v)\}, \{\{x,u\}, \{y,v\}\})$ *et* $\mu_{12} = (\{r(x,a)\}, \{r(u,v)\}, \{\{x,u\}, \{a,v\}\})$ *qui produisent respectivement* $Q'_{11} = \{p(y), s(x,y), r(x,a)\}$ *et* $Q'_{12} = \{p(y), r(x,y), s(x,a)\}$. *Ni* Q'_{11} *ni* Q'_{12} *ni* Q_1 *elle-même ne sont plus générales que* Q'_2 *donc* rew_{β_1+} *n'est pas élagable.*

Non seulement, l'opérateur rew_{β_1+} n'est pas élagable mais en plus, si on l'utilise dans l'algorithme 1, la sortie peut être incomplète. En effet, la restriction aux unificateurs mono-pièce n'est pas compatible avec la sélection des réécritures les plus générales à chaque pas comme le fait l'algorithme.

Exemple 37 (Exemple simple) *Soit* $Q = p(y,z) \wedge p(z,y)$ *et* $R = r(x,x) \rightarrow p(x,x)$. *Il y a deux unificateurs mono-pièce de* Q *avec* R, $\mu_1 = (\{p(y,z)\}, \{p(x,x)\}, \{\{x,y,z\}\})$ *et* $\mu_2 = (\{p(z,y)\}, \{p(x,x)\}, \{\{x,y,z\}\})$, *qui mènent à la même réécriture* $Q_1 = r(x,x) \wedge p(x,x)$. *Il y a aussi un unificateur par pièce composé de deux pièces* $\mu = (Q, \{p(x,x)\}, \{\{x,y,z\}\})$, *qui produit* $Q' = r(x,x)$. *Une requête équivalente à* Q' *peut être obtenue à partir de* Q_1 *par un pas supplémentaire d'unification mono-pièce. Maintenant, si on considère que l'on se restreint aux unificateurs mono-pièce* et *que l'on ne garde que les réécritures les plus générales à chaque pas,* Q_1 *ne sera pas gardée puisque* $Q \geq Q_1$. *Si* Q_1 *n'est pas gardée,* Q' *ne sera jamais générée, alors qu'elle est incomparable à* Q, *donc l'ensemble de sortie sera incomplet.*

Dans l'exemple précédent, soit P_1, respectivement P_2, les partitions associées à μ_1, respectivement μ_2, on peut noter que $P_1(Q)$ est redondant ainsi que $P_2(Q)$ donc le problème pourrait être réglé en calculant $u_1(Q) \setminus u_1(Q')$ à la place de $u_1(Q \setminus Q')$ et en rendant $P_1(Q)$ non redondant, c'est-à-dire identique à $p(x,x)$, avant de calculer $u_1(Q) \setminus u_1(Q')$ qui donc serait vide. Cependant, le problème est plus profond comme l'illustre l'exemple suivant.

Exemple 38 (Prédicats ternaires) *Soit* $Q = r(u,v,w) \wedge r(w,t,u)$ *et* $R = p(x,y) \rightarrow r(x,y,x)$. *De nouveau, il y a deux unificateurs mono-pièce de* Q *avec* R *:* $\mu_1 = (\{r(u,v,w)\}, \{r(x,y,x)\}, \{\{u,w,x\}, \{v,y\}\})$ *et* $\mu_2 =$

$(\{r(w,t,u)\},\{r(x,y,x)\},\{\{u,w,x\},\ \{t,y\}\})$. *On obtient deux réécritures plus spécifiques que* Q, $Q_1 = p(x,y) \wedge r(x,v,x)$, *et* $Q_2 = p(x,y) \wedge r(x,t,x)$, *qui sont isomorphes. Il y a aussi un unificateur composé de deux pièces* $(Q,\{r(x,y,x)\},\{\{u,w,x\},\{v,t,y\}\})$, *qui produit* $p(x,y)$. *Si nous enlevons* Q_1 *et* Q_2, *parce qu'elles sont plus spécifiques que* Q, *aucune requête équivalente à* $p(x,y)$ *ne pourra être générée.*

Le dernier exemple est intéressant car il n'utilise que des prédicats binaires et unaires et que la règle est très simple et peut être exprimée dans n'importe quelle logique de description légère.

Exemple 39 *Soit* $Q = r(u,v) \wedge r(v,w) \wedge p(u,z) \wedge p(v,z) \wedge p(v,t) \wedge p(w,t) \wedge p_1(u) \wedge p_2(w)$ *(voir la figure 4.1 où les sommets représentent les termes, les arcs représentent les prédicats binaires et les prédicats unaires étiquettent les sommets-termes qu'ils ont pour argument) et* $R = b(x) \rightarrow p(x,y)$. *Notez que* Q *n'est pas redondante. Il y a deux unificateurs mono-pièce de* Q *avec* R, *nommés* μ_1 *et* μ_2, *avec les pièces* $Q'_1 = \{p(u,z),p(v,z)\}$ *et* $Q'_2 = \{p(v,t),p(w,t)\}$ *respectivement. Les réécritures obtenues sont dessinées dans la figure 4.1. Ces requêtes sont toutes deux plus spécifiques que* Q. *Les enlever va empêcher le calcul d'une requête équivalente à* $r(x,x) \wedge p_1(x) \wedge p_2(x) \wedge b(x)$, *qui peut être générée à partir de* Q *avec un unificateur composé de deux pièces.*

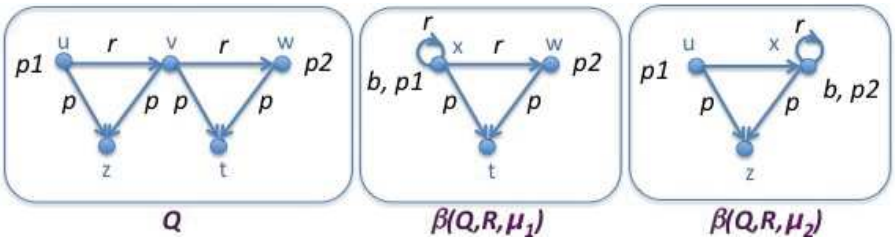

FIGURE 4.1 – Les requêtes de l'exemple 39

Même si les unificateurs mono-pièce n'ont pas la propriété d'élagabilité pour être directement utilisés pour l'algorithme 1, ils ont l'intérêt d'être adéquats et complets et sont plus simples et moins nombreux à calculer que les unificateurs par pièce. C'est la raison pour laquelle nous les avons utilisés comme "briques de base" pour le calcul d'un opérateur élagable qui sera présenté dans la section suivante.

4.5 Unificateur mono-pièce agrégé

L'opérateur présenté dans cette section allie les avantages des opérateurs précédemment présentés en essayant d'éviter leurs inconvénients. Ainsi, il s'appuiera sur des unificateurs les plus généraux et mono-pièce pour profiter de la rapidité

de calcul, mais pour éviter le problème de non élagabilité des unificateurs mono-pièce, il calculera l'agrégation des unificateurs mono-pièce d'une même règle. Un unificateur par pièce peut être découpé en plusieurs unificateurs mono-pièce (voir preuve du lemme 5) qui devront être appliqués les uns après les autres ; la requête obtenue n'est pas équivalente à celle qui serait obtenue avec l'unificateur par pièce, mais plus générale, ce qui satisfait notre objectif. Toutefois, il faut donc plusieurs pas de réécritures pour obtenir une requête qui ne nécessitait qu'un pas avec un unificateur classique ; si les réécritures intermédiaires sont éliminées par le test de subsomption, la requête voulue ne sera jamais atteinte. Afin d'éviter ce problème et donc retrouver l'élagabilité, nous allons construire une agrégation d'unificateurs mono-pièce qui permettra d'appliquer en un pas plusieurs unificateurs mono-pièce. Néanmoins, tous les unificateurs ne peuvent pas être appliqués ensemble, il faut qu'ils soient compatibles.

Définition 4.14 (Unificateur compatible, agrégation d'unificateur) *Soit* $\mathcal{U} = \{\mu_1 = (Q'_1, C'_1, P_1) \dots \mu_k = (Q'_k, C'_k, P_k)\}$ *un ensemble d'unificateurs par pièce de* Q *avec* R_1, ..., R_k *respectivement. On suppose que les règles ont toujours des ensembles de variables disjoints, sinon, on renommera les variables en commun. On dit que* $\mathcal{U}$ *est* compatible *si*

1. *pour tout i,j,* $i \neq j$, $Q'_i \cap Q'_j = \emptyset$
2. *la jointure de* $P_1, \dots, P_k$ *est admissible*

Soit $\mathcal{U}$ *un tel ensemble compatible, l'unificateur* agrégé *de* $\mathcal{U}$*, noté* $\mu_1 \diamond \dots \diamond \mu_k$*, est* $\mu = (Q', C', P)$ *où*

1. $Q' = Q'_1 \cup \dots \cup Q'_k$
2. $C' = C'_1 \cup \dots \cup C'_k$
3. P *est la jointure de* $P_1, \dots, P_k$

μ *est dit* le plus général *si tous les unificateurs par pièce de* $\mathcal{U}$ *sont les plus généraux.* μ *est dit* mono-pièce *si tous les unificateurs par pièce de* $\mathcal{U}$ *sont mono-pièce.*

Pour prouver la correction des unificateurs agrégés nous allons utiliser le fait qu'ils sont des unificateurs par pièce pour une règle dite agrégée, que nous définissons ci-dessous.

Définition 4.15 (Agrégation de règles) *Soit* $\mathcal{R} = \{R_1 \dots R_k\}$ *un ensemble de règles avec des ensembles de variables disjoints. L'*agrégation *de* $\mathcal{R}$*, noté* $R_1 \diamond \dots \diamond R_k$*, est la règle* $hyp(R_1) \wedge \dots \wedge hyp(R_k) \rightarrow concl(R_1) \wedge \dots \wedge concl(R_k)$.

Pour le moment, les unificateurs que l'on agrège sont tous des unificateurs entre une requête et des copies d'une même règle, mais, dans la section 4.6 , nous envisagerons l'agrégation entre unificateurs de règles différentes, les définitions et propriétés sont donc écrites de manière générale pour rester valables.

Propriété 5 *Soit Q une requête et $\mathcal{U} = \{\mu_1 = (Q'_1, C'_1, P_1) \dots \mu_k = (Q'_k, C'_k, P_k)\}$ un ensemble compatible d'unificateurs par pièce de Q avec R_1, ..., R_k respectivement. L'unificateur agrégé de $\mathcal{U}$ est un unificateur par pièce de Q avec l'agrégation de $\{R_1$, ..., $R_k\}$.*

Preuve : Nous allons d'abord montrer que l'unificateur agrégé $\mu = (Q', C', P_u)$ de $\mathcal{U}$ satisfait les conditions de la définition 4.10 d'un unificateur par pièce. La première condition est satisfaite puisque la jointure de $P_1 \dots P_k$ est admissible. La seconde condition est aussi satisfaite puisque comme $P_1 \dots P_k$ la satisfont, leur jointure la satisfait également. En effet, si une classe contient une variable existentielle, elle ne peut pas être fusionnée avec une autre lors de l'agrégation puisque les autres termes de la classe sont des variables non-séparatrices, donc n'apparaissent pas dans d'autres classes. Pour la dernière condition, pour tout $1 \leq i \leq k$, on a $P_i(C'_i) = P_i(Q'_i)$. Puisque $Q' = \bigcup_{i=1}^{k} Q'_i$ et $C' = \bigcup_{i=1}^{k} C'_i$, P étant la jointure de $P_1 \dots P_k$, on a $P(C') = P(Q')$. □

Grâce à cette propriété, nous pouvons définir la réécriture produite par un unificateur agrégé μ comme la réécriture $\beta(Q, R_1 \diamond \dots \diamond R_k, \mu)$. Comme le montre l'exemple suivant, cette réécriture correspond à la réécriture obtenue en appliquant les unificateurs associés aux R_i les uns après les autres.

Exemple 40 *Soit $R = p(x, y) \rightarrow q(x, y)$, $R' = p(x', y') \rightarrow q(x', y')$ une copie de R et $Q = q(u, v) \wedge r(v, w) \wedge q(t, w)$.*

Soit $\mu_1 = (Q'_1, C'_1, P_1)$ un unificateur par pièce de Q avec R où $Q'_1 = \{q(u, v)\}$, $C'_1 = \{q(x, y)\}$ et $P_1 = \{\{u, x\}, \{v, y\}\}$. Et $\mu_2 = (Q'_2, C'_2, P_2)$ un unificateur par pièce de Q avec R' où $Q'_2 = \{q(t, w)\}$, $C'_2 = \{q(x', y')\}$ et $P_2 = \{\{t, x'\}, \{w, y'\}\}$.

Si on applique successivement μ_1 et μ_2 on obtient $Q_s = p(u, v) \wedge r(v, w) \wedge p(t, w)$.

L'agrégation $R \diamond R'$ est la règle $p(x, y) \wedge p(x', y') \rightarrow q(x, y) \wedge q(x', y')$. L'unificateur agrégé de μ_1 et μ_2 est $(\{q(u, v), q(t, w)\}, \{q(x, y), q(x', y')\}, \{\{u, x\}, \{v, y\}, \{t, x'\}, \{w, y'\}\})$ et il produit exactement Q_s.

En fait, la seule différence entre un unificateur multi-pièces et l'agrégation des unificateurs mono-pièce correspondants est que pour un unificateur par pièce on considère une partition sur les termes de $Q \cup \mathit{concl}(R)$ alors que pour l'unificateur agrégé on considère une partition sur les termes de $Q \cup \bigcup_{i=1}^{k} \mathit{concl}(R_i)$ où k est le nombre de pièces considéré et chaque R_i est une nouvelle copie de R. En d'autres mots, si on considérait que les R_i étaient identiques à R au lieu d'être des copies, l'agrégation des R_i serait exactement R après suppression des atomes en double et l'unificateur agrégé serait un unificateur classique.

La propriété suivante montre que tout unificateur par pièce peut être remplacé par une agrégation d'unificateurs mono-pièce.

Propriété 6 *Pour tout unificateur par pièce μ d'une requête Q avec une règle R, il existe un unificateur agrégé mono-pièce le plus général $\mu_\diamond$ de Q avec $R_1, \ldots, R_i$ copies de R tel que $\beta(Q, R_1 \diamond \cdots \diamond R_i, \mu_\diamond) \geq \beta(Q, R, \mu)$.*

Preuve : Soit $Q'_1, \ldots, Q'_k$ les pièces de Q' selon $\mu = (Q', C', P)$. Soit $R_1 \ldots R_k$ les nouvelles copies de R. Soit h_i la bijection renommant les variables utilisées pour produire R_i à partir de R. Soit $\mathcal{U} = \{\mu_1 = (Q'_1, C'_1, P^1_u), \ldots, \mu_k = (Q'_k, C'_k, P^k_u)\}$ l'ensemble des unificateurs par pièce de Q avec $R_1, \ldots, R_k$ construit de la manière suivante pour chaque i :

- C'_i est l'image par h_i du sous-ensemble de C' unifié par P avec Q'_i
- soit $h_i(P)$ la partition construite à partir de P en remplaçant chaque $x \in var(C')$ par $h_i(x)$; puis, P_i est obtenue de $h_i(P)$ (1) en la restreignant aux termes de Q'_i et C'_i, et (2) en la raffinant autant que possible en gardant la propriété que $P_i(C'_i) = P_i(Q'_i)$.

Pour tout $\mu_i = (Q'_i, C'_i, P_i)$ on peut immédiatement vérifier que :

1. μ_i est un unificateur par pièce le plus général
2. μ_i est un unificateur mono-pièce.
3. pour tout $\mu_i, \mu_j \in \mathcal{U}$, avec $\mu_i \neq \mu_j$, μ_j et μ_i sont compatibles.

Soit $\mu_\diamond = (Q'_\diamond, C'_\diamond, P_\diamond)$ l'unificateur agrégé de $\mathcal{U}$. On note que $Q'_\diamond = Q'$. Les propriétés précédentes vérifiées pour chaque μ_i de $\mathcal{U}$ assurent que $\mu_\diamond$ est un unificateur agrégé mono-pièce le plus général.

On note $R_\diamond = R_1 \diamond \ldots \diamond R_k$. Il reste à prouver que $\beta(Q, R_\diamond, \mu_\diamond) \geq \beta(Q, R, \mu)$. Soit h l'inverse de l'union des h_i, h est une substitution préservant les classes de $P_\diamond$ vers P, Soit s la substitution complémentaire de $P_\diamond$ dans P selon h, pour tout terme t de $P_\diamond$, c'est-à-dire de $Q' \cup C'_\diamond$, $s(P_\diamond(t)) = P(h(t))$.

On construit maintenant une substitution h' de $var(\beta(Q, R_\diamond, \mu_\diamond))$ dans $term(\beta(Q, R, \mu))$, avec trois possibilités selon la partie de $\beta(Q, R_\diamond, \mu_\diamond)$ où la variable apparaît, c'est-à-dire dans Q mais pas Q', dans $hyp(R_\diamond)$ mais pas $C'_\diamond$ ou dans la partie restante qui correspond aux images de $sep(Q')$ par $P_\diamond$:

1. si $x \in var(Q) \setminus var(Q')$, $h'(x) = x$;
2. si $x \in var(hyp(R_\diamond)) \setminus var(C'_\diamond)$, $h'(x) = h(x)$;
3. si $x \in P_\diamond(sep(Q'))$(ou indifféremment $x \in P_\diamond(fr(R_\diamond) \cap var(C'_\diamond))$), $h'(x) = s(x)$;

On conclut en montrant que h' est un homomorphisme de $\beta(Q, R_\diamond, \mu_\diamond) = P_\diamond(hyp(R_1) \cup \cdots \cup hyp(R_k)) \cup P_\diamond(Q \setminus Q')$ dans $\beta(Q, R, \mu) = P(hyp(R)) \cup u(Q \setminus Q')$, avec deux points :

1. pour tout i, $h'(P_\diamond(hyp(R_i))) = P(hyp(R))$. En effet, pour toute variable $x \in var(hyp(R_i))$:
   - soit $x \in var(hyp(R_\diamond)) \setminus var(C'_\diamond)$, donc $h'(P_\diamond(x)) = h'(x) = h(x) = P(h(x))$ (P ne substitue pas les variables de $var(hyp(R)) \setminus var(C')$),
   - ou $x \in fr(R_\diamond) \cap var(C'_\diamond)$, donc $h'(P_\diamond(x)) = s(P_\diamond(x)) = P(h(x))$;

   et pour tout i, $P(h(hyp(R_i))) = P(hyp(R))$.
2. $h'(P_\diamond(Q \setminus Q')) = P(Q \setminus Q')$. En effet, pour toute variable $x \in var(Q \setminus Q')$:
   - soit $x \in var(Q')$, alors $h'(P_\diamond(x)) = s(P_\diamond(x)) = P(h(x)) = P(x)$ (h ne substitue pas les variables de Q),
   - ou $x \in var(Q) \setminus var(Q')$, alors $h'(P_\diamond(x)) = h'(x) = x = P(x)$ ($P_\diamond$ et P ne substitue pas les variables de $var(Q) \setminus var(Q')$).

□

On appelle opérateur mono-pièce agrégé, noté sra ("single rule aggregator"), l'opérateur de réécriture qui a toute requête associe les réécritures produites par des unificateurs mono-pièce les plus généraux mais aussi les réécritures produites par toutes les agrégations possibles d'unificateurs mono-pièce les plus généraux entre Q et une même règle R.

Exemple 41 *Soit $R_1 = p(x, y) \rightarrow q(x, y)$, $R_2 = s(x', y') \rightarrow r(x', y')$ une copie de R et $Q = q(u, v) \wedge r(v, w) \wedge q(t, w)$. Il y a trois unificateurs mono-pièce entre Q et $\{R_1, R_2\}$. $\mu_1 = (Q'_1, C'_1, P_1)$ unificateur mono-pièce de Q avec R_1 où $Q'_1 = \{q(u, v)\}$, $C'_1 = \{q(x, y)\}$ et $P_1 = \{\{u, x\}, \{v, y\}\}$, $\mu_2 = (Q'_2, C'_2, P_2)$ unificateur mono-pièce de Q avec R_1 où $Q'_2 = \{q(t, w)\}$, $C'_2 = \{q(x, y)\}$ et $P_2 = \{\{t, x\}, \{w, y\}\}$ et $\mu_3 = (Q'_3, C'_3, P_3)$ unificateur mono-pièce de Q avec R_2 où $Q'_3 = \{r(v, w)\}$, $C'_3 = \{r(x', y')\}$ et $P_3 = \{\{v, x'\}, \{w, y'\}\}$. L'opérateur sra associe à Q les réécritures produites par μ_1, μ_2, μ_3 les unificateurs mono-pièce et par $\mu_1 \diamond \mu_2$ l'agrégation des unificateurs de la règle R_1.*

La propriété 6 permet de prouver que l'opérateur *sra* est adéquat, complet et élagable et peut donc être utilisé pour l'algorithme 1 pour produire une sortie adéquate et complète.

Théorème 9 *L'opérateur sra est adéquat, complet et élagable.*

Preuve : L'adéquation vient de la propriété 5 et du fait que pour tout ensemble de règles $\mathcal{R}$, soit $\mathcal{R}_\diamond$ l'agrégation de $\mathcal{R}$, on a $\mathcal{R} \models \mathcal{R}_\diamond$. La complétude et l'élagabilité viennent de la propriété 6 qui assure que toute réécriture produite en un pas par rew_β est subsumée par une réécriture produite en un pas par *sra* et du fait que rew_β est complet et élagable. □

4.6 Perspective d'amélioration

Au cours de notre thèse, nous avons aussi développé un opérateur de réécriture supplémentaire *aram* dans le but de diminuer le nombre de requêtes générées lors de la réécriture avec *sra*. Bien qu'adéquat et complet, cet opérateur n'est pas élagable au sens de la définition 3.8. Nous pensons néanmoins que l'algorithme 1 associé à cet opérateur produit un ensemble de réécritures complet grâce à une propriété d'élagabilité moins locale que celle proposée précédemment. En revanche, la propriété qui nous permettrait de prouver que la sortie de l'algorithme 1 est complète avec *aram* reste à trouver.

Le but de cet opérateur est d'éviter de générer des requêtes identiques. Lorsque deux unificateurs par pièce μ_1 et μ_2 portent sur des morceaux différents de la requête, on a la possibilité de les appliquer l'un après l'autre dans les deux ordres, μ_1 puis μ_2 et μ_2 puis μ_1, cela produira la même réécriture obtenue par deux "chemins" différents. Voici, dans l'exemple suivant, une illustration de ce phénomène.

Exemple 42 *Soit $Q = p(u, w) \wedge r(u, v)$ une requête, $R_1 = q(x) \rightarrow p(x, y)$ et $R_2 = t(x') \rightarrow r(x', y')$ deux règles. Il existe deux unificateurs mono-pièce agrégés $\mu_1 = (\{p(u, w)\}, \{p(x, y)\}, \{\{u, x\}, \{w, y\}\})$ de Q avec R_1 et $\mu_2 = (\{r(u, v)\}, \{r(x', y')\}, \{\{u, x'\}, \{v, y'\}\})$ de Q avec R_2.*

En appliquant μ_1 sur Q on obtient $Q_1 = q(u) \wedge r(u, v)$ sur laquelle on peut appliquer μ_2 et obtenir $Q_{12} = q(u) \wedge t(u)$. En appliquant μ_2 sur Q on obtient $Q_2 = p(u, w) \wedge t(u)$ sur laquelle on peut appliquer μ_1 et obtenir $Q_{21} = q(u) \wedge t(u)$.

Or Q_{12} et Q_{21} sont identiques.

Afin d'éviter la génération de requêtes identiques liée à ce phénomène, nous avons développé un opérateur de réécriture *aram* qui calcule *en une étape* les réécritures obtenues avec les unificateurs mono-pièce agrégés et, en utilisant l'agrégation, les réécritures que l'on aurait obtenues après l'enchaînement de plusieurs de ces unificateurs compatibles. En effet, soit μ_1 et μ_2 deux unificateurs mono-pièce agrégés compatibles, la réécriture produite par $\mu_1 \diamond \mu_2$ est plus générale que la réécriture produite par l'application de μ_1 puis μ_2 ou μ_2 puis μ_1. Ainsi, on obtient en un pas toutes les réécritures impliquant ces unificateurs. Pour éviter de calculer à nouveau ces réécritures, on utilise un système de marquage qui empêche que ces unificateurs soient utilisés une autre fois. Le système de marquage marque tous les atomes nouvellement ajoutés, et un unificateur doit pour être utilisé unifier au moins un atome marqué.

Définition 4.16 (Marquage d'atomes) *Soit Q une requête, $\mu = (Q', C', P)$ un unificateur par pièce de Q avec une règle R, $\beta(Q, R, \mu) = P(hyp(R)) \cup P(Q \setminus Q')$. Les* atomes marqués *de $\beta(Q, R, \mu)$ sont les atomes de $P(hyp(R))$.*

Une requête marquée est une requête qui contient des atomes marqués selon la règle de marquage précédente. Tous les atomes de la requête initiale sont supposés marqués.

L'opérateur de réécriture *aram* associe à une requête marquée Q et à un ensemble de règles, toutes les réécritures produites par un unificateur mono-pièce agrégé $\mu = (Q', C', P)$ dont Q' contient au moins un atome marqué. De plus, il associe aussi toutes les réécritures produites par toutes les agrégations possibles des unificateurs compatibles utilisés pour produire la première partie des réécritures.

Dans la figure 4.2, on peut retrouver l'illustration des réécritures produites par *sra* à partir de la requête et des règles de l'exemple 42, dans la figure 4.3, celle des réécritures produites pas l'opérateur *aram* s'il ne tenait pas compte du marquage et enfin dans la figure 4.4 celles produites par l'opérateur *aram* tenant compte du marquage (les atomes soulignés sont les atomes marqués).

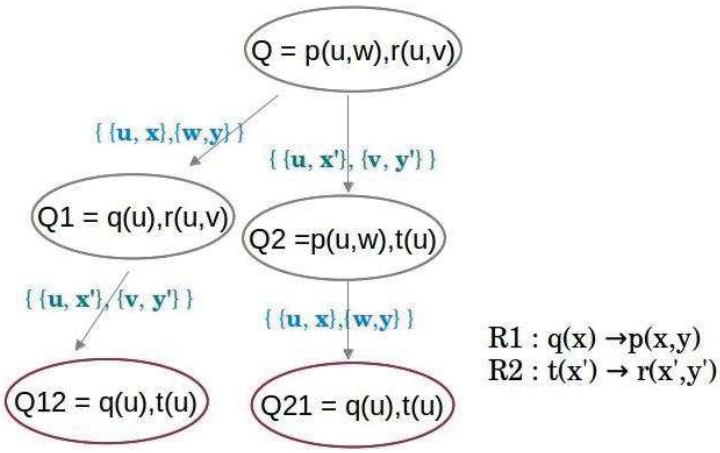

FIGURE 4.2 – Exemple 42 - *sra*

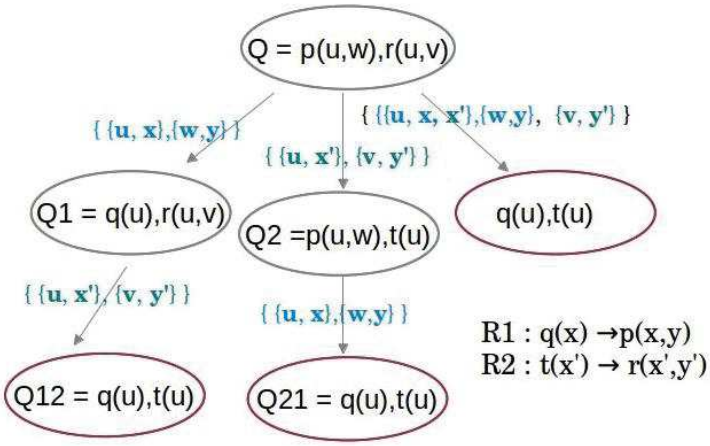

FIGURE 4.3 – Exemple 42 - *aram* sans marquage

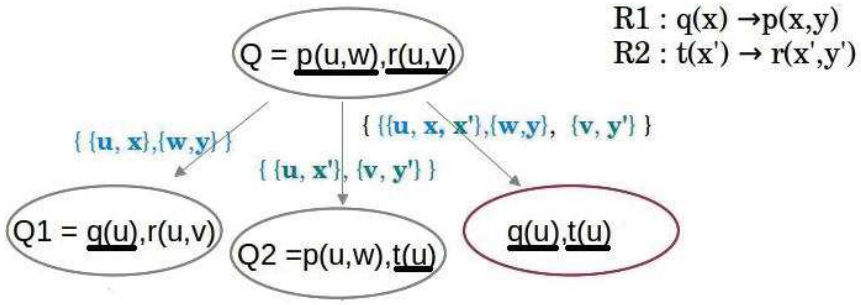

FIGURE 4.4 – Exemple 42 - *aram*

Voici sur un exemple le comportement de *aram*, illustrer par la figure 4.5, où le soulignement signifie qu'un atome est marqué.

Exemple 43 *Soit* $Q = \underline{p(u,w)} \wedge \underline{r(u,v)} \wedge \underline{s(u,z)}$ *une requête,* $\mathcal{R} = \{R_1, R_2, R_3, R_4, R_5\}$ *un ensemble de règles avec* $R_1 = \underline{q(x)} \rightarrow \underline{p(x,y)}$, $R_2 = t(x) \rightarrow r(x,y)$, $R_3 = h(x) \rightarrow s(x,y)$, $R_4 = g(x) \rightarrow h(x)$ *et* $R_5 = f(x) \rightarrow h(x)$. *Q étant la requête initiale, tous ses atomes sont marqués.*

Il existe trois unificateurs mono-pièce agrégés utilisant un atome marqué :

- $\mu_1 = (\{p(u,w)\}, \{p(x,y)\}, \{\{u,x\}, \{w,y\}\})$ *de Q avec* R_1,
- $\mu_2 = (\{r(u,v)\}, \{r(x,y)\}, \{\{u,x\}, \{v,y\}\})$ *de Q avec* R_2,
- $\mu_3 = (\{s(u,z)\}, \{s(x,y)\}, \{\{u,x\}, \{z,y\}\})$ *de Q avec* R_3.

μ_1, μ_2 *et* μ_3 *sont compatibles.* aram *produit donc à partir de Q et* $\mathcal{R}$ *les réécritures avec* μ_1, μ_2, μ_3, $\mu_1 \diamond \mu_2$, $\mu_1 \diamond \mu_3$ *et* $\mu_1 \diamond \mu_2 \diamond \mu_3$ *respectivement qui correspondent aux requêtes* $Q_1 = \underline{q(u)} \wedge r(u,v) \wedge s(u,z)$, $Q_2 = p(u,w) \wedge \underline{t(u)} \wedge s(u,z)$, $Q_3 = p(u,w) \wedge r(u,v) \wedge \underline{h(u)}$, $Q_4 = \underline{q(u)} \wedge \underline{t(u)} \wedge s(u,z)$, $Q_5 = p(u,w) \wedge \underline{t(u)} \wedge \underline{h(u)}$ *et* $Q_6 = \underline{q(u)} \wedge \underline{t(u)} \wedge \underline{h(u)}$ *respectivement.*

A partir de Q_1, aram *ne produit aucune réécriture car les unificateurs mono-pièce agrégés* μ_2, μ_3, *qui existent sur* Q_1 *ne réécrivent pas d'atomes marqués.*

A partir de Q_2, aram *ne produit aucune réécriture car les unificateurs mono-pièce agrégés* μ_1, μ_3, *qui existent sur* Q_2 *ne réécrivent pas d'atomes marqués.*

A partir de Q_3, aram *ne produit aucune réécriture avec* μ_1, μ_2 *car ils ne réécrivent pas d'atomes marqués. En revanche, il produit deux réécritures avec* $\mu_4 = (\{h(u)\}, \{h(x)\}, \{\{u,x\}\})$ *et* $\mu_5 = (\{h(u)\}, \{h(x)\}, \{\{u,x\}\})$ *deux unificateurs par pièce agrégés de Q avec* R_4 *et* R_5 *respectivement. Ces réécritures sont* $Q_7 = p(u,w) \wedge r(u,v) \wedge \underline{g(u)}$ *et* $Q_8 = p(u,w) \wedge r(u,v) \wedge \underline{f(u)}$ *respectivement. Par contre,* μ_4 *et* μ_5 *ne sont pas compatibles, il ne produit donc pas de réécritures avec* $\mu_4 \diamond \mu_5$.

A partir de Q_4, aram *ne produit aucune réécriture avec* μ_3, *car il ne réécrit pas d'atomes marqués.*

A partir de Q_5, aram ne produit aucune réécriture avec μ_1, car il ne réécrit pas d'atomes marqués. En revanche, il produit deux réécritures avec μ_4 et μ_5 qui sont $Q_9 = p(u,w) \wedge t(u) \wedge \underline{g(u)}$ et $Q_{10} = p(u,w) \wedge t(u) \wedge \underline{f(u)}$ respectivement. Par contre, μ_4 et μ_5 ne sont pas compatibles, il ne produit donc pas de réécritures avec $\mu_4 \diamond \mu_5$.

A partir de Q_6, aram produit deux réécritures avec μ_4 et μ_5 qui sont $Q_{11} = q(u) \wedge t(u) \wedge \underline{g(u)}$ et $Q_{12} = q(u) \wedge t(u) \wedge \underline{f(u)}$ respectivement. Par contre, μ_4 et μ_5 ne sont pas compatibles, il ne produit donc pas de réécritures avec $\mu_4 \diamond \mu_5$.

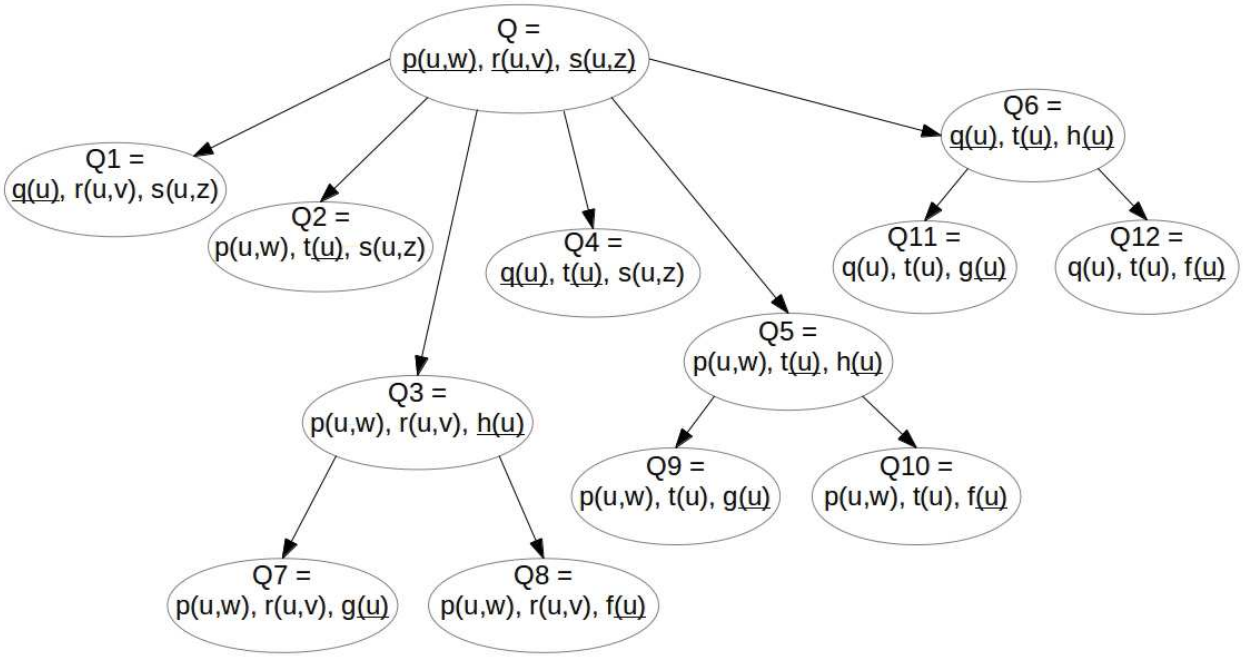

FIGURE 4.5 – Exemple 43

Il est trivial de voir que cet opérateur n'est pas élagable au sens de la définition 3.8, en effet, si on prend une requête identique à une autre mais dont aucun atome n'est marqué, elle est plus générale que la première pourtant elle ne produira aucune réécriture.

Dans [König et al., 2013], on a pu confirmer, par des expérimentations préliminaires, le fait que *aram* produit moins de requêtes intermédiaires que *sra*. L'opérateur de réécriture *aram* applique en un pas tous les unificateurs mono-pièce agrégés et leurs agrégations compatibles. Ensuite, il suffit de ne considérer que les unificateurs impliquant un atome nouvellement créé. Ce comportement permet de mettre à profit le graphe de dépendance des règles (GRD pour "graph of rule dependencies") [Baget et al., 2009]. En effet, le GRD indique la dépendance entre règles, une règle R_1 dépend d'une règle R_2 s'il existe un atome de $concl(R_2)$ qui est unifiable par

un unificateur par pièce avec un atome de $hyp(R_1)$. Soit Q une requête et R_1, R_2 deux règles, soit Q' la réécriture de Q avec un unificateur par pièce de Q avec R_1. Il est facile de vérifier que si R_1 ne dépend pas de R_2, il ne peut y avoir d'unificateur par pièce de Q' avec R_2 utilisant un atome nouvellement ajouté (marqué). Ainsi, le GRD nous permet de ne chercher des unificateurs mono-pièce agrégés d'une requête Q qu'avec les règles dont dépendent la règle qui a permis de produire Q.

4.7 Comparaison aux algorithmes existants

Nous présentons maintenant les principales méthodes utilisées en OBQA pour calculer des réécritures sous forme d'UCQ et les mettons en relation avec nos travaux.

La première méthode, nommée PerfectRef et parfois algorithme CGLLR, par exemple dans [Pérez-Urbina et al., 2009], est développée dans [Calvanese et al., 2005] pour DL-Lite$_\mathcal{R}$ et forme la base du raisonneur QuOnto (`http://www.dis.uniroma1.it/~quonto/`). Avant de commencer l'étape de réécriture, la TBox est mise sous forme normale, c'est-à-dire que chaque inclusion de la forme $B \sqsubseteq C_1 \sqcap C_2$ est remplacée par deux inclusions $B \sqsubseteq C_1$ et $B \sqsubseteq C_2$. Il ne reste alors plus dans la TBox que des inclusions positives de la forme $B_1 \sqsubseteq B_2$, des inclusions négatives de la forme $B_1 \sqsubseteq \neg B_2$ (où B_1 et B_2 sont des concepts ou des rôles basiques) et des axiomes de fonctionnalité. Les inclusions négatives et les axiomes de fonctionnalité n'interviennent pas dans l'étape de réécriture, nous les laisserons donc de côté.

On considère parfois qu'une TBox DL-Lite$_\mathcal{R}$ contient aussi des inclusions de la forme $B \sqsubseteq \exists r.A$ où B est un concept basique, r un rôle basique et A un concept atomique. Une telle inclusion peut être mise sous forme normale en la remplaçant par les inclusions suivantes :

$$B \sqsubseteq \exists p \qquad \exists p^- \sqsubseteq A \qquad p \sqsubseteq r$$

où p est un nouveau rôle atomique. Cette normalisation correspond exactement à transformer l'inclusion en règle et à la découper en règles à conclusion atomique (voir section 7.4 pour plus de détails sur cette décomposition). En effet, la règle $B(x) \rightarrow r(x, y) \wedge A(y)$ est décomposée en trois règles à conclusion atomique :

$$B(x) \rightarrow p(x, y) \qquad p(x, y) \rightarrow A(y) \qquad p(x, y) \rightarrow r(x, y)$$

où p est un nouveau prédicat.

Pour effectuer les réécritures, l'algorithme PerfectRef procède en deux étapes : la première effectue des réécritures atomiques de la requête grâce aux inclusions positives tandis que la seconde fusionne des paires d'atomes de la requête. Les termes de la requête peuvent être de deux types : si ce sont des constantes, des variables partagées (apparaissant dans plusieurs atomes) ou des variables réponses, on les appelle termes *liés* sinon ils sont dits *non liés* et notés par $-$.

Dans la première étape, il s'agit d'appliquer une inclusion positive à un atome de la requête pour réécrire cet atome en un autre atome. Soit I une inclusion positive. I est applicable à un atome $B(x)$ de la requête si elle possède B en partie droite. I est applicable à un atome $p(x, y)$ de la requête si : soit elle possède p en partie droite, soit y est une variable non liée et la partie droite de I est $\exists p$, soit x est une variable non liée et la partie droite de I est $\exists p^-$. La réécriture que l'on obtient en appliquant une inclusion I sur un atome D de la requête Q est calculée en remplaçant D dans Q selon les règles suivantes :

- Si $D = A(x)$: soit $I = B \sqsubseteq A$ alors on remplace D par $B(x)$, soit $I = \exists p \sqsubseteq A$ alors on remplace D par $p(x, -)$, soit $I = \exists p^- \sqsubseteq A$ alors on remplace D par $p(-, x)$.
- Si $D = p(x, -)$: soit $I = A \sqsubseteq \exists p$ alors on remplace D par $A(x)$, soit $I = \exists q \sqsubseteq \exists p$ alors on remplace D par $q(x, -)$, soit $I = \exists q^- \sqsubseteq \exists p$ alors on remplace D par $q(-, x)$.
- Si $D = p(-, x)$: soit $I = A \sqsubseteq \exists p^-$ alors on remplace D par $A(x)$, soit $I = \exists q \sqsubseteq \exists p^-$ alors on remplace D par $q(x, -)$, soit $I = \exists q^- \sqsubseteq \exists p^-$ alors on remplace D par $q(-, x)$.
- Si $D = p(x, y)$: soit $I = q \sqsubseteq p$ ou $I = q^- \sqsubseteq p^-$ alors on remplace D par $q(x, y)$, soit $I = q \sqsubseteq p^-$ ou $I = q^- \sqsubseteq p$ alors on remplace D par $q(y, x)$.

Comme nous l'avons vu dans la section 2.3.3, toute inclusion positive de concepts I de la forme considéré plus haut, peut être traduite en une règle existentielle à hypothèse et conclusion atomique, soit R cette règle. Autrement dit, en se plaçant du point de vue des règles existentielles, I est applicable à un atome de la requête si l'atome en conclusion de R est unifiable par un upg u avec cet atome, et si, de plus, les constantes, les variables partagées et les variables réponses ne sont pas unifiées par u avec des variables existentielles de R. La réécriture de la requête sera obtenue en remplaçant l'atome unifié par l'atome de l'hypothèse R auquel on applique u et dont on transforme les variables n'appartenant pas à la frontière par $-$.

Dans la seconde étape, il s'agit simplement de fusionner deux atomes unifiables de la requête par un upg u. La réécriture est obtenue en appliquant u à la requête et en éliminant l'atome redondant. Cette étape a pour but de faire disparaître des variables partagées pour permettre de nouvelles applications d'inclusions positives. L'algorithme applique les deux étapes alternativement jusqu'à ne plus produire de nouvelles réécritures (à un isomorphisme près). Voici sur un exemple le fonctionnement de cet algorithme :

Exemple 44 *Soit $Q = ans(x, z)$:- $p(x, y) \wedge p(z, y)$ une requête et $I = A \sqsubseteq \exists p$ une inclusion positive. Les variables liées de Q sont x et z car ce sont des variables réponses et y car c'est une variable partagée. I n'est pas applicable à $p(x, y)$ car y est une variable liée, pour la même raison elle n'est pas applicable à $p(z, y)$.*

On passe donc à l'étape d'unification qui unifie $p(x, y)$ *et* $p(z, y)$ *pour obtenir* $ans(x, x)$ *:-* $p(x, y)$ *sur laquelle* I *est applicable et permet d'obtenir* $ans(x, x)$ *:-* $A(x)$*.*

On peut relever un certain nombre d'avantages qu'a notre algorithme par rapport à PerfectRef :

1. Tout d'abord, il est important de remarquer que PerfectRef a autant de règles de réécriture que d'axiomes au lieu d'avoir une règle générale basée sur les unificateurs par pièce comme c'est notre cas. Lorsque l'on change l'expressivité de la DL, il faut ajouter ou enlever des règles de réécriture alors que nos opérateurs peuvent être utilisés sur toutes les règles existentielles sans modification.

2. Une autre différence réside dans le fait que nos opérateurs permettent de réécrire plusieurs atomes à la fois alors que PerfectRef effectue des réécritures atome par atome.

3. A cause de cette dernière particularité, il a fallu, pour retrouver la complétude, introduire une autre opération, l'unification d'atomes de la requête. Le point faible de cette solution est que toutes les unifications possibles sont effectuées, et pas seulement celles qui vont mener à une nouvelle application d'une inclusion positive. Ainsi, dans certains cas, des unifications inutiles vont être faites comme le montre l'exemple suivant :

   Exemple 45 *Soit* $Q = ans(u)$ *:-* $p(u, v) \wedge p(v, w) \wedge b(w)$ *une requête et* $I = A \sqsubseteq \exists p$ *une inclusion positive. Les variables liées de* Q *sont* v *et* w *car ce sont des variables partagées et* u *car c'est une variable réponse.* I *n'est pas applicable à* $p(u, v)$ *car* v *est une variable liée et elle n'est pas applicable à* $p(v, w)$ *car* w *est une variable liée.*
   On passe donc à l'étape d'unification qui unifie $p(u, v)$ *et* $p(v, w)$ *pour obtenir* $ans(u)$ *:-* $p(u, u) \wedge b(u)$ *sur lequel* I *n'est toujours pas applicable car* u *est une variable liée, l'étape d'unification a donc été inutile. Par comparaison, notre algorithme aurait détecté directement qu'il n'y a pas d'unificateur par pièce de* Q *avec* I*.*

4. Comme nous l'avons noté précédemment, la mise sous forme normale de la TBox correspond à une décomposition en règles à conclusion atomique. Cette décomposition pour les axiomes de la forme $B \sqsubseteq \exists r.A$, entraîne l'ajout de nouveaux prédicats et provoque une grosse augmentation du nombre de réécritures calculables (voir la section 7.4 pour l'analyse de cet impact).

5. Enfin, il n'y a pas de test de subsomption entre requêtes, PerfectRef ne produit donc pas forcément un ensemble de réécritures minimal.

La seconde méthode que nous décrivons est un algorithme basé sur des règles de résolution (qui peuvent être vues comme des cas particulier d'application de

la règle de résolution générales). Cet algorithme, nommé RQR, est introduit dans [Pérez-Urbina et al., 2009] et implémenté dans REQUIEM. L'algorithme prend une requête conjonctive Q et une TBox T en entrée et produit une réécriture de Q selon T qui a différentes formes selon l'expressivité de la TBox. Si T est une TBox $\mathcal{ELHI}$, la réécriture de Q est un programme Datalog. Si T est une TBox DL-Lite$_\mathcal{R}$, la réécriture de Q est une UCQ, c'est dans ce dernier cas que nous nous plaçons. L'algorithme est composé de quatre étapes : *clausification*, *saturation*, *unfolding* et *pruning*. La première étape, *clausification* consiste à réécrire la TBox sous forme d'un ensemble de clauses, les clauses étant vues comme des règles. Les clauses obtenues sont définies pour chaque "type" d'axiome mais cela revient à traduire les axiomes de la TBox en règles existentielles, comme défini dans la section 2.3.3, et à skolémiser ces règles pour obtenir des clauses de Horn (voir exemple 13). Lors de l'étape de *saturation*, l'algorithme produit de nouvelles clauses à partir de la requête et des clauses issues de la TBox. On peut classifier les clauses de cette étape en deux ensembles, les clauses "ontologiques" qui viennent des axiomes de la TBox et les clauses "réécritures" qui viennent de la requête. Pour effectuer la *saturation*, l'algorithme utilise un ensemble de *templates*, qui permettent de combiner deux clauses "ontologiques" pour produire une nouvelle clause "ontologique", ou alors qui combine une clause "réécriture" avec une clause "ontologique" pour produire une nouvelle clause "réécriture". Les *templates* qui combinent deux clauses "ontologiques" correspondent exactement à l'application de la règle de résolution entre une conclusion et une hypothèse de règle, les deux étant réduites à un seul atome. Cette part de la saturation peut être rapprochée de la saturation que nous définissons dans la section 5.2, sur des règles dites compilables, la différence résidant dans la présence de symboles fonctionnels. Les autres templates appliquent sous certaines conditions la règle de résolution sur un atome d'une clause "réécriture" et un atome d'une clause "ontologique" (tête de règle) pour former une nouvelle clause "réécriture". La phase d'*unfolding* supprime d'abord de l'ensemble produit à l'étape précédente les clauses dans lesquelles apparaissent des symboles fonctionnels. Elle utilise ensuite les clauses restantes pour produire de nouvelles clauses avec une règle de réécriture qui correspond à la règle de résolution. La dernière étape de *pruning* enlève de l'ensemble les clauses qui ne possèdent pas le prédicat réponse et effectue des tests de subsomption pour calculer la couverture de l'ensemble de clauses. Par rapport à PerfectRef, RQR évite certaines unifications inutiles en ne découpant pas en deux sortes d'opération, mais il ne les évite pas toutes, comme on peut le voir en reprenant l'exemple 45.

Exemple 46 *Soit $p(u,v) \wedge p(v,w) \wedge b(w) \rightarrow ans(u)$ une requête et $I = A \sqsubseteq \exists p$ une inclusion positive traduite par la clause $A(x) \rightarrow p(x, f(x))$. L'étape de saturation produira trois nouvelles clauses :*

- *$C = A(u) \wedge p(f(u), w) \wedge b(w) \rightarrow ans(u)$ à partir du premier atome de la requête et de la clause issue de I*
- *$p(u,v) \wedge A(v) \wedge b(f(v)) \rightarrow ans(u)$ à partir du second atome de la requête et de la clause issue de I*

- $A(u) \wedge A(f(u)) \wedge b(f(f(u)) \rightarrow ans(u)$ *à partir* C *et de la clause issue de* I

Ces clauses seront supprimées à l'étape unfolding suivante elles ont donc été calculées inutilement. En comparaison, un opérateur basé sur les unificateurs par pièce aurait repéré directement qu'aucune réécriture de la requête n'était possible avec I*.*

Depuis ces travaux précurseurs, plusieurs algorithmes ont été proposés pour DL-Lite. Presto [Rosati and Almatelli, 2010] décompose la génération d'une réécriture de la requête en deux étapes :

1. il élimine les variables partagées et non réponses (dites "existential join"), ce qui remplace l'étape de fusion de PerfectRef avec ses unifications "à l'aveugle".

2. Il effectue une réécriture similaire à celle de PerfectRef sur la requête produite à la première étape.

De plus, Presto réécrit la requête en un programme Datalog non récursif, ce qui génère une réécriture plus compacte. Ce programme Datalog peut ensuite être expansé en une UCQ.

Les dernières méthodes implémentées dans les systèmes Rapid [Chortaras et al., 2011] et tw-rewriting/Ontop [Rodriguez-Muro et al., 2013] mettent particulièrement à profit les spécificités de DL-Lite et en particulier le fait que le modèle canonique de la base de connaissances est un arbre sur la partie "anonyme" (celle qui ne porte pas uniquement sur les constantes de la ABox). Nous n'avons pas fait de comparaison théorique avec ces algorithmes qui sont trop éloignés du nôtre, et qui ne semblent pas directement généralisables à des classes de règles existentielles qui ne sont pas des DLs.

Concernant les règles existentielles, un seul algorithme alternatif a été proposé à notre connaissance. Cet algorithme est introduit dans [Calì et al., 2010b, Gottlob et al., 2011] et implémenté dans le système Nyaya. Certaines erreurs de la première version ont été corrigées dans [Gottlob et al., 2014a], où le système prend le nom de SYSNAME. Cet algorithme a été proposé pour les règles sticky, mais il est applicable à n'importe quel type de règles existentielles comme le nôtre, avec la garantie d'arrêt si une UCQ existe. Cependant, il faut préciser que l'ensemble de règles est supposé être à conclusion atomique ; il est donc nécessaire de décomposer des règles en introduisant des nouveaux prédicats. Ceci n'a aucune incidence sur l'expressivité, mais réduit l'efficacité en pratique (voir la discussion à la section 7.4). Ajoutons qu'une optimisation applicable si l'ensemble de règles est linéaire, (équivalent à hypothèse atomique) ce qui inclus DL-Lite$_{\mathcal{R}}$, permet de supprimer des parties de la requête qui sont redondantes par rapport aux règles. La technique de réécriture utilisée par Nyaya décompose l'unification en deux étapes similairement à PerfectRef : une étape de factorisation, qui permet de fusionner des atomes de la requête, de façon à transformer des variables partagées entre plusieurs atomes en des variables non-partagées ; et une étape d'unification proprement dite, qui unifie des atomes de la requête avec la conclusion de la règle (qui est atomique) ; cette

étape d'unification ne permet pas d'unifier une variable existentielle de la conclusion avec des variables partagées de la requête, d'où la nécessité de l'étape préalable de factorisation. A nouveau, l'intérêt de la notion d'unificateur par pièce est qu'elle évite de calculer des factorisations de la requête qui ne mèneront finalement pas à des réécritures.

Il est possible d'accorder PerfectRef au cadre défini dans le chapitre précédent en définissant un opérateur de réécriture, qui à une requête et à un ensemble de règles correspondantes aux inclusions positives de la TBox, associe les requêtes obtenues par fusion d'atomes et par application des inclusions positives sur la requête. On peut facilement vérifier que cet opérateur est adéquat et complet mais ne satisfait pas la propriété d'élagabilité. De plus, il n'est pas possible de conserver la complétude en effectuant à chaque étape un test de subsomption comme on peut le voir sur l'exemple suivant.

Exemple 47 *Soit une inclusion positive* $I = A \sqsubseteq \exists r$ *et une requête* $Q = ans() :- r(a,u) \wedge r(v,u) \wedge B(v)$ *où* a *est une constante. L'opérateur produit la requête* $Q_2 = ans() :- r(a,u) \wedge B(a)$ *par fusion, elle n'en produit aucune par application de* I *puisque* u *est partagée donc liées. Si on effectue un test de subsomption* Q_2 *est éliminée car elle est subsumée par* Q*. On ne pourra alors pas produire* $Q_3 = ans() :- A(a) \wedge B(a)$ *en appliquant* I *sur* Q_2 *alors que cette requête appartient nécessairement à l'ensemble de réécritures. On peut vérifier que l'on obtient* Q_3 *avec un unificateur par pièce.*

Une autre solution serait de définir un opérateur de réécriture s'appuyant sur PerfectRef qui associe à une requête et à un ensemble de règles correspondantes aux inclusions positives de la TBox, les requêtes obtenues par application des inclusions positives sur la requête de départ et sur les requêtes que l'on peut obtenir par fusion d'atomes à partir de la requête de départ. Cela revient à reconstruire des unificateurs par pièce. Cet opérateur sera alors adéquat complet et élagable. Concernant RQR, il semble plus difficile à exprimer sous forme d'un opérateur de réécriture s'appliquant sur une requête selon des règles puisque les opérations de RQR sont sur des paires de clauses qui ne correspondent pas forcément à une requête et une règle. Une réflexion plus approfondie doit être menée à ce sujet. Dans les mêmes conditions que PerfectRef, l'algorithme de Nyaya peut être transposé en un opérateur de réécriture élagable.

Le chapitre suivant est dédié aux améliorations développées pour rendre plus rapide le calcul des réécritures avec notre opérateur de réécriture *sra*.

Chapitre 5

Compilation de règles

Dans ce chapitre, nous proposons un nouvel opérateur de réécriture basé sur l'unification par pièce mais prenant en compte un pré-ordre sur les atomes. Ce pré-ordre est extrait de la compilation des règles dites compilables qui n'auront alors plus besoin d'être considérées lors de la phase de réécriture, diminuant ainsi le nombre de réécritures à effectuer. La définition de cet opérateur nécessite la redéfinition des deux opérations de base de l'algorithme de réécriture : l'unification et l'homomorphisme. Dans un premier temps, nous expliquerons le principe de cette technique sur les règles hiérarchiques puis nous l'étendrons à un ensemble de règles plus important dites compilables.

5.1 Prise en compte de règles hiérarchiques

Quel que soit le langage ontologique ou le domaine, la première chose que l'on voudra exprimer au sein d'une ontologie est une hiérarchie de concepts et dans une moindre mesure de relations. Dans les logiques de description, cette notion est représentée par l'inclusion atomique entre concepts ou rôles $\sqsubseteq$. En OWL ou RDFS, on la retrouvera avec les assertions *subClassOf* et *subPropertyOf*. Dans notre cadre en logique du premier ordre, cette relation hiérarchique est représentée par des règles de la forme $p(x_1, \ldots, x_n) \rightarrow q(x_1, \ldots x_n)$, où pour tout $i \neq j$, on a $x_i \neq x_j$. Lorsque p et q sont des concepts, $n = 1$ et la règle exprime que p est plus spécifique que q. Ces règles, que l'on appellera simplement *règles hiérarchiques*, sont des causes bien connues d'explosion combinatoire en réécriture de requêtes comme l'illustre l'exemple qui suit.

Exemple 48 *Soit $\{R_1, \ldots, R_n\}$ un ensemble de règles de la forme $R_i = p_i(x) \rightarrow p_{i-1}(x)$. Soit $Q = p_0(x_1), \ldots, p_0(x_k)$. Chaque atome $p_0(x_i)$ de Q est réécrit en $p_1(x_i)$ qui à son tour est réécrit en $p_2(x_i)$ et ainsi de suite. A la fin, il y a $(n+1)^k$ réécritures de Q.*

Les règles hiérarchiques peuvent être compilées en un pré-ordre partiel sur les prédicats, c'est-à-dire une relation réflexive et transitive, que nous noterons $\leq_p$.

En effet, les règles hiérarchiques définissent une relation $<$ sur les prédicats : pour chaque règle $p(x,y) \rightarrow q(x,y)$ on a $p < q$. En calculant la fermeture réflexive et transitive de $<$ on obtient le pré-ordre partiel $\leq_p$ sur les prédicats. Ce pré-ordre $\leq_p$ sur les prédicats induit un pré-ordre partiel $\preccurlyeq$ sur les atomes que l'on peut définir de la manière suivante :

Définition 5.1 ($\preccurlyeq$) *Soit $\mathcal{R}_h$ un ensemble de règles hiérarchiques et $\leq_p$ le pré-ordre sur les prédicats associé. Soit $p(x_1,\ldots,x_k)$ et $q(x_1,\ldots,x_k)$ deux atomes ayant mêmes arguments. On a $p(x_1,\ldots,x_k) \preccurlyeq q(x_1,\ldots,x_k)$ si on a $p \leq_p q$.*

Comme nous allons le montrer de manière informelle dans l'exemple suivant, en étendant les opérations d'homomorphisme et d'unification pour prendre en compte ce pré-ordre, il n'est plus nécessaire de considérer les règles hiérarchiques lors de la phase de réécriture.

Exemple 49 *Soit $Q = s(u,v)$, $F = p(a)$, $R_1 = r(x,y) \rightarrow s(x,y)$, $R_2 = q(x) \rightarrow r(x,y)$ et $R_3 = p(x) \rightarrow q(x)$. Les règles hiérarchiques R_1 et R_3 induisent le pré-ordre sur les prédicats où $r \leq_p s$ et $p \leq_p q$.*

En prenant en compte le pré-ordre sur les atomes induit par $\leq_p$ on peut remarquer que Q est unifiable avec la conclusion de R_2. En effet, si on considère la partition $\{\{x,u\},\{y,v\}\}$, on peut unifier $r(x,y)$ et $s(u,v)$ puisque $r(u,v) \preccurlyeq s(u,v)$ car $r \leq_p s$. Il y a donc un $\preccurlyeq$-unificateur (prenant en compte le pré-ordre) entre Q et R_2 $\mu_{\preccurlyeq} = (\{s(u,v)\},\{r(x,y)\},\{\{x,u\},\{y,v\}\})$ qui mène à la réécriture $Q_2 = q(u)$.

En ne considérant plus les règles hiérarchiques pour effectuer des réécritures, on diminue le nombre de réécritures mais le processus conserve sa complétude si l'homomorphisme est étendu pour prendre en compte le pré-ordre sur les atomes. En effet, il n'y a pas d'homomorphisme classique entre Q_2 et F en revanche il y a un $\preccurlyeq$-homomorphisme (prenant en compte le pré-ordre) $\pi_{\preccurlyeq} = \{(u,a)\}$ de Q_2 dans F. En effet, $p(a) \preccurlyeq \pi_{\preccurlyeq}(q(u)) = q(a)$ car $p \leq_p q$.

On peut se convaincre que $(F,R) \models Q$ en remarquant que la réécriture de Q par R_1 produit $Q_1 = r(u,v)$ puis que la réécriture de Q_1 par R_2 produit Q_2, que la réécriture de Q_2 par R_3 produit $Q_3 = p(u)$ et enfin que $\{(u,a)\}$ est un homomorphisme classique de Q_3 dans F. Les définitions formelles d'$\preccurlyeq$-homomorphisme et $\preccurlyeq$-unificateur peuvent être trouvées dans les sections 5.3 et 5.4.

Nous allons maintenant étendre l'ensemble de règles compilées, le principe reste le même mais le calcul du pré-ordre sur les atomes induit par les règles se complexifie.

5.2 Extension aux règles compilables

C'est la simplicité des règles hiérarchiques qui les rende compilables en un pré-ordre sur les prédicats. En effet, lorsqu'on les utilise pour effectuer des réécritures mono-pièces, ces règles remplacent un atome par un seul autre atome. De plus, elles

ne contiennent pas de variables existentielles donc lorsqu'un atome s'unifie à une conclusion de règle, on sait qu'il s'agit d'un unificateur par pièce sans avoir besoin de vérifications supplémentaires. Les règles qui regroupent ces deux particularités sont dites compilables. En revanche, le pré-ordre qu'elles induisent sur les atomes est plus complexe à calculer que pour les règles hiérarchiques.

Définition 5.2 (Règle compilable) *Un règle est dite* compilable *si elle a un seul atome en hypothèse, aucune variable existentielle et aucune constante.*

La condition sur les constantes sert à simplifier les définitions. Les règles compilables peuvent être vue comme des règles Datalog à corps atomique. On supposera aussi qu'une règle compilable a une conclusion atomique, autrement, elle peut être directement décomposée en autant de règles qu'elle possède d'atomes en conclusion (voir propriété 4).

Nous pouvons voir dans l'exemple suivant que les règles compilables expriment différents axiomes ontologiques couramment utilisés.

Exemple 50 *Soit l'ensemble de règles compilables $\mathcal{R}_{\preccurlyeq}$ suivant :*
$R_1 = r(x, y) \rightarrow t(x, y)$ (r est une spécialisation de t)
$R_2 = s(x, y) \rightarrow t(y, x)$ (s et t sont des relations inverses)
$R'_2 = t(x, y) \rightarrow s(y, x)$
$R_3 = t(x, y) \rightarrow q(x)$ (q est le domaine de la relation t)
$R_4 = t(x, y) \rightarrow q(y)$ (q est le co-domaine [1] de la relation t)
$R_5 = p(x, y, z) \rightarrow r(x, z)$ (r est une projection de p)
$R_6 = p(x, x, z) \rightarrow s(x, x)$ (introduction d'une boucle)

Avec les règles hiérarchiques, il suffisait de calculer la fermeture réflexive et transitive de la relation entre prédicats donnée par les règles pour obtenir un pré-ordre sur les prédicats $\leq_p$ aboutissant au pré-ordre $\preccurlyeq$ sur les atomes. Avec les règles compilables, cela devient plus compliqué, les règles n'exprimant plus simplement une relation de subsomption entre prédicats mais plutôt une relation entre prédicats associés à des positions d'arguments. Comme nous ne savons pas calculer la fermeture transitive de cette relation directement, nous allons donc procéder au niveau des règles elles mêmes pour les combiner jusqu'à obtenir un ensemble de règles, appelé saturation, exprimant la fermeture transitive de la relation initiale.

Définition 5.3 (Règle inférée, saturation) *Soit R_1 et R_2 deux règles compilables telles que $concl(R_1)$ et $hyp(R_2)$ sont unifiables par un unificateur logique classique le plus général u. La règle* inférée *à partir de (R_1, R_2) est $R_1 \bullet R_2 = u(hyp(R_1)) \rightarrow u(concl(R_2))$.*

Soit $\mathcal{R}_{\preccurlyeq}$ un ensemble de règles compilables, la saturation *de $\mathcal{R}_{\preccurlyeq}$, notée $\mathcal{R}^*_{\preccurlyeq}$, est la fermeture de $\mathcal{R}_{\preccurlyeq}$ par l'opération $\bullet$.*

1. "range" en anglais

Voici la saturation de l'ensemble de règles compilables de l'exemple 50.

Exemple 50 (Suite) *Les règles inférées à partir de $\mathcal{R}_{\preccurlyeq}$ sont les suivantes :*
$R_1 \bullet R'_2 = r(x,y) \rightarrow s(y,x)$
$R_1 \bullet R_3 = r(x,y) \rightarrow q(x)$
$R_1 \bullet R_4 = r(x,y) \rightarrow q(y)$
$R_2 \bullet R'_2 = s(x,y) \rightarrow s(x,y)$
$R_2 \bullet R_3 = s(x,y) \rightarrow q(y)$
$R_2 \bullet R_4 = s(x,y) \rightarrow q(x)$
$R'_2 \bullet R_2 = t(x,y) \rightarrow t(x,y)$
$R_5 \bullet R_1 = p(x,y,z) \rightarrow t(x,z)$
$R_6 \bullet R_2 = p(x,x,z) \rightarrow t(x,x)$
$R_5 \bullet R_1 \bullet R'_2 = p(x,y,z) \rightarrow s(z,x)$
$R_5 \bullet R_1 \bullet R_3 = p(x,y,z) \rightarrow q(x)$
$R_5 \bullet R_1 \bullet R_4 = p(x,y,z) \rightarrow q(z)$
$R_6 \bullet R_2 \bullet R_3 = R_6 \bullet R_2 \bullet R_4 = p(x,x,z) \rightarrow q(x)$
Nous rappelons que nous considérons que les règles ont des ensembles de variables disjoints, même si pour simplifier nous utilisons les mêmes variables.

Si nous considérons que l'arité des prédicats est bornée, la taille de la saturation $\mathcal{R}^*_{\preccurlyeq}$ d'un ensemble de règles compilables $\mathcal{R}_{\preccurlyeq}$ est polynomiale en la taille de $\mathcal{R}_{\preccurlyeq}$.

Propriété 7 *Soit $\mathcal{R}_{\preccurlyeq}$ un ensemble de règles compilables. La taille de sa saturation $\mathcal{R}^*_{\preccurlyeq}$ est bornée par $|\mathcal{R}_{\preccurlyeq}|^2 \times B_k$, où k est l'arité maximum des prédicats et B_k est le $k^{\text{ème}}$ nombre de Bell.*

Preuve : Le nombre de spécialisations possibles d'un atome d'arité k est borné par le nombre de partitions de $\{1, \ldots, k\}$, connu comme le $k^{\text{ème}}$ nombre de Bell B_k. Ainsi, le nombre de règles distinctes (à un renommage de variable près) qui peuvent être obtenues est bornée par $|\mathcal{R}_{\preccurlyeq}|^2 \times B_k$. On précise que $B_k < 2^{k^2}$. □

La saturation de $\mathcal{R}_{\preccurlyeq}$ peut facilement être rendue minimale en enlevant les règles tautologiques ou redondantes. Une règle est tautologique si tout interprétation est un modèle de la règle. Les règles compilables sont tautologiques si et seulement si elles ont un même atome en conclusion et en hypothèse ($H \rightarrow H$). On dira qu'une règle est redondante par rapport à un ensemble de règles si l'ensemble contient une autre règle dont elle se déduit :

Définition 5.4 (Règle redondante) *Soit $\mathcal{R}$ un ensemble de règles et R_j une règle de cet ensemble. R_j est redondante par rapport à $\mathcal{R}$ s'il existe $R_i \in \mathcal{R}$, $i \neq j$, telle que $R_i \models R_j$.*

On peut vérifier que $R_i \models R_j$ si et seulement si R_i subsume R_j selon la définition suivante.

Définition 5.5 (Subsomption de règles) *Soit R_i et R_j deux règles compilables. On dit que R_i* subsume *R_j s'il existe un homomorphisme h de $hyp(R_i)$ dans $hyp(R_j)$ tel que $h(concl(R_i)) = concl(R_j)$.*

Propriété 8 *Soit R_i et R_j deux règles compilables (sans variables communes). $R_i \models R_j$ si et seulement si R_i subsume R_j*

Preuve : Soit $R_j = H_j \rightarrow C_j$ et $R_i = H_i \rightarrow C_i$.

$\Leftarrow$ Soit h_i de H_i dans H_j avec $h_i(C_i) = C_j$. On montre que tout modèle I de R_i est un modèle de R_j . Soient p et q les prédicats respectifs de H_i et H_j, et de C_i et C_j . Pour simplifier l'écriture, étant donné un prédicat p, si $(d_1...d_k) \in I(p)$, on note $p(d_i...d_k) \in I$. On prouve que pour toute assignation s des termes de H_j dans le domaine de I telle que $s(H_j) \in I$, on a $s(C_j) \in I$. Soit s telle que $s(H_j) \in I$. Alors $s \circ h_i(H_i) \in I$. Puisque I est un modèle de R_i , $s \circ h_i(C_i) \in I$, autrement dit $s(C_j) \in I$.

$\Rightarrow$ Supposons que $R_i \models R_j$. S'il n'existe pas d'homomorphisme de H_i dans H_j, on peut construire une interprétation vérifiant la formule existentielle $\neg H_i \wedge H_j \wedge \neg C_j$ (rappel : R_j n'est pas valide), qui sera un modèle de R_i mais pas de R_j. Soit donc h_i l'unique homomorphisme de H_i dans H_j . Soit I un modèle isomorphe la formule existentielle $h_i(H_i) \wedge h_i(C_i) = H_j \wedge h_i(C_i)$. I est un modèle de R_i , donc de R_j par hypothèse. Il existe donc un homomorphisme h de $H_j \wedge C_j$ dans $H_j \wedge h_i(C_i)$, qui laisse H_j invariant. Comme R_j n'est pas valide, h envoie forcément C_j dans $h_i(C_i)$, autrement dit $h(C_j) = h_i(C_i)$. Comme $vars(C_j) \subseteq vars(H_j)$, on a $h(x) = x$ pour toute variable x de C_j. Donc $h(C_j) = C_j = h_i(C_i)$. □

Exemple 50 (Suite) *Les règles $R_2 \bullet R'_2 = s(x, y) \rightarrow s(x, y)$ et $R'_2 \bullet R_2 = t(x, y) \rightarrow t(x, y)$ sont des règles tautologiques. $R_6 \bullet R_2 \bullet R_3 = R_6 \bullet R_2 \bullet R_4 = p(x, x, z) \rightarrow q(x)$ est subsumée par $R_5 \bullet R_1 \bullet R_3 = p(x, y, z) \rightarrow q(x)$.*

Nous pouvons maintenant définir le pré-ordre sur les atomes induit par les règles compilables qui permet d'éviter de les considérer lors de la réécriture de requêtes. Ce pré-ordre sur les atomes est étendu aux ensembles d'atomes.

Définition 5.6 ($\preccurlyeq$) *Soit $\mathcal{R}_{\preccurlyeq}$ un ensemble de règles compilables et A et B deux atomes. On note $A \preccurlyeq B$ si (i) $A = B$ ou si (ii) il existe une règle $R \in \mathcal{R}^*_{\preccurlyeq}$ qui subsume la règle $(A \rightarrow B)$ (de manière équivalente : l'application de R sur A produit exactement B).*

Soit $\mathcal{A}$ et $\mathcal{B}$ des ensembles d'atomes. On note $\mathcal{A} \preccurlyeq \mathcal{B}$ s'il existe une application surjective f de $\mathcal{B}$ dans $\mathcal{A}$ telle que pour tout $B \in \mathcal{B}$, $f(B) \preccurlyeq B$.

On peut remarquer que si $A \preccurlyeq B$ et qu'on est dans le cas *(ii)* de la définition, alors il existe une réécriture directe de B avec R par un unificateur par pièce qui est plus générale que A ou identique à A.

Exemple 51 *Soit $R = p(x, y) \rightarrow t(x)$ une règle et $A = p(c_1, c_2)$ et $B = t(c_1)$ deux atomes. $A \preccurlyeq B$ car $p(c_1, c_2) \rightarrow t(c_1)$ est subsumée par R. On peut noter que la réécriture de B par un unificateur par pièce le plus général produit $p(c_1, y)$ qui est plus général que A.*

Plus généralement, en considérant des ensembles d'atomes :

Propriété 9 *Soit $\mathcal{A}$ et $\mathcal{B}$ deux ensembles d'atomes et $\mathcal{R}_{\preccurlyeq}$ un ensemble de règles compilables associé au pré-ordre $\preccurlyeq$ sur les atomes. On a $\mathcal{A} \preccurlyeq \mathcal{B}$ si et seulement s'il existe une k-réécriture $\mathcal{B}'$ de $\mathcal{B}$ selon rew_β (l'opérateur de réécriture utilisant les unificateurs par pièce) et $\mathcal{R}_{\preccurlyeq}$, qui s'envoie sur $\mathcal{A}$ par une substitution s de $var(\mathcal{B}') \setminus var(\mathcal{B})$ dans $term(\mathcal{A})$ telle que $s(\mathcal{B}') = \mathcal{A}$.*

Preuve :
$\Rightarrow$ Soit $\mathcal{A} \preccurlyeq \mathcal{B}$, c'est-à-dire qu'il existe une application surjective f de $\mathcal{B}$ dans $\mathcal{A}$ telle que pour tout $B \in \mathcal{B}$, $f(B) \preccurlyeq B$. Pour tout $B \in \mathcal{B}$, soit R_B une règle de $\mathcal{R}^*_{\preccurlyeq}$ telle que R_B subsumes $f(B) \rightarrow B$: alors, il y a un homomorphisme h_B de $hyp(R_B)$ dans $f(B)$ tel que $h_B(concl(R_B)) = B$; nous divisons h_B en h_B^{hyp} la partie de h_B ayant pour domaine $var(hyp(R_B)) \setminus var(concl(R_B))$ et h_B^{concl} la partie de h_B ayant pour domaine $var(concl(R_B))$. Soit $\mu = (\{B\}, concl(R_B), P)$ où P est la partition associée à h_B^{concl} ; puisque $h_B^{concl}(concl(R_B)) = B$, μ est un unificateur par pièce de $\mathcal{B}$ avec R_B. Les unificateurs par pièce μ associés à chaque $B \in \mathcal{B}$ peuvent être itérativement appliqués à $\mathcal{B}$; soit $\mathcal{B}'$ la réécriture obtenue à partir de $\mathcal{B}$ et de ces unificateurs par pièce. $\mathcal{B}'$ est composé de chaque $h_B^{concl}(hyp(R_B))$, de plus, $h_B^{hyp}(h_B^{concl}(hyp(R_B))) = f(B)$. Enfin, soit s l'union de tous les h_B^{hyp}. On a $s(\mathcal{B}') = \mathcal{A}$; de plus, puisque chaque h_B^{hyp} est une substitution de $var(hyp(R_B)) \setminus var(concl(R_B))$ dans $term(f(B)))$, s est une substitution de $var(\mathcal{B}') \setminus var(\mathcal{B})$ dans $term(\mathcal{A})$.
$\Leftarrow$ On suppose qu'il existe $\mathcal{B}'$ une k-réécriture de $\mathcal{B}$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$ qui s'envoie sur $\mathcal{A}$ par une substitution s de $var(\mathcal{B}') \setminus var(\mathcal{B})$ dans $term(\mathcal{A})$ telle que $s(\mathcal{B}') = \mathcal{A}$. Nous prouvons qu'il existe une application surjective f' de $\mathcal{B}$ dans $\mathcal{A}$ telle que pour tout $B \in \mathcal{B}$, $f'(B) \preccurlyeq B$, donc $\mathcal{A} \preccurlyeq \mathcal{B}$. Puisque $\mathcal{B}'$ est une k-réécriture de $\mathcal{B}$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$, on a $\mathcal{B}' \preccurlyeq \mathcal{B}$; donc, il existe une application surjective f de $\mathcal{B}$ dans $\mathcal{B}'$ telle que pour tout $B \in \mathcal{B}$, $f(B) \preccurlyeq B$. A partir de f, nous pouvons définir une application f' de $\mathcal{B}$ dans $\mathcal{A}$ de la manière suivante : pour tout $B \in \mathcal{B}$, $f'(B) = s(f(B))$. f' est surjective puisque f est surjective et que $s(\mathcal{B}') = \mathcal{A}$. Puisque $f(B) \preccurlyeq B$, il existe une règle R telle que R subsume $f(B) \rightarrow B$. Aucune variable de B n'apparaît dans $var(\mathcal{B}') \setminus var(\mathcal{B})$, par conséquence, $s(B) = B$. Ceci implique que R subsume $s(f(B)) \rightarrow B$, donc $f'(B) = s(f(B)) \preccurlyeq B$. □

Il faut noter que la propriété précédente considère que l'on garde lors de la réécriture en priorité les variables de la requête. Sur l'exemple suivant, nous pouvons voir une illustration de cette propriété.

Exemple 52 *Soit $\mathcal{R}^*_{\preccurlyeq}$ de l'exemple 50. Soit $\mathcal{A} = \{p(u, u, c_1), r(c_2, c_1)\}$ et $\mathcal{B} = \{s(u, u), s(c_1, u), t(c_2, c_1), q(c_2)\}$, où c_1 et c_2 sont des constantes.*

On a $\mathcal{A} \preccurlyeq \mathcal{B}$ *puisque :*

- $p(u,u,c_1) \to s(u,u)$ *est subsumée par* $R_6 = p(x,x,z) \to s(x,x)$,
- $p(u,u,c_1) \to s(c_1,u)$ *est subsumée par* $R_5 \bullet R_1 \bullet R'_2 = p(x,y,z) \to s(z,x)$,
- $r(c_2,c_1) \to t(c_2,c_1)$ *est subsumée par* $R_1 = r(x,y) \to t(x,y)$
- $r(c_2,c_1) \to q(c_2)$ *est subsumée par* $R_1 \bullet R_3 = r(x,y) \to q(x)$.

Selon la propriété 9, on peut aussi vérifier que $\mathcal{B}' = \{p(u,u,z), p(u,y,c_1), r(c_2,c_1), r(c_2,y')\}$ *est une k-réécriture de* $\mathcal{B}$ *selon* rew_β *et* $\mathcal{R}_{\preccurlyeq}$ *et que* $\mathcal{B}'$ *se projette sur* $\mathcal{A}$. *On obtient* $\mathcal{B}'$ *de la manière suivante :*

- *en réécrivant* $\mathcal{B} = \{s(u,u), s(c_1,u), t(c_2,c_1),\ q(c_2)\}$ *avec* $R_6 = p(x,x,z) \to s(x,x)$ *on obtient* $\mathcal{B}_1 = \{p(u,u,z), s(c_1,u), t(c_2,c_1),\ q(c_2)\}$,
- *en réécrivant* $\mathcal{B}_1 = \{p(u,u,z), s(c_1,u), t(c_2,c_1),\ q(c_2)\}$ *avec* $R'_2 = t(x,y) \to s(y,x)$ *on obtient* $\mathcal{B}_2 = \{p(u,u,z), t(u,c_1), t(c_2,c_1),\ q(c_2)\}$,
- *en réécrivant* $\mathcal{B}_2 = \{p(u,u,z), t(u,c_1), t(c_2,c_1),\ q(c_2)\}$ *avec* $R_1 = r(x,y) \to t(x,y)$ *on obtient* $\mathcal{B}_3 = \{p(u,u,z), r(u,c_1), t(c_2,c_1),\ q(c_2)\}$,
- *en réécrivant* $\mathcal{B}_3 = \{p(u,u,z), r(u,c_1), t(c_2,c_1),\ q(c_2)\}$ *avec* $R_5 = p(x,y,z) \to r(x,z)$ *on obtient* $\mathcal{B}_4 = \{p(u,u,z), p(u,y,c_1), t(c_2,c_1),\ q(c_2)\}$,
- *en réécrivant* $\mathcal{B}_4 = \{p(u,u,z), p(u,y,c_1), t(c_2,c_1),\ q(c_2)\}$ *avec* $R_1 = r(x,y) \to t(x,y)$ *on obtient* $\mathcal{B}_5 = \{p(u,u,z), p(u,y,c_1), r(c_2,c_1),\ q(c_2)\}$,
- *en réécrivant* $\mathcal{B}_5 = \{p(u,u,z), p(u,y,c_1), r(c_2,c_1),\ q(c_2)\}$ *avec* $R_3 = t(x,y) \to q(x)$ *on obtient* $\mathcal{B}_6 = \{p(u,u,z), p(u,y,c_1), r(c_2,c_1),\ t(c_2,y')\}$,
- *en réécrivant* $\mathcal{B}_6 = \{p(u,u,z), p(u,y,c_1), r(c_2,c_1),\ t(c_2,y')\}$ *avec* $R_1 = r(x,y) \to t(x,y)$ *on obtient* $\mathcal{B}' = \{p(u,u,z), p(u,y,c_1), r(c_2,c_1),\ r(c_2,y')\}$.

Nous allons maintenant redéfinir les deux opérations de base de notre algorithme de réécriture : l'homomorphisme et l'unification, pour qu'elles prennent en compte le pré-ordre sur les atomes que nous avons calculé.

5.3 Prise en compte d'un pré-ordre sur les atomes pour l'homomorphisme

L'implication logique entre faits et requêtes peut être calculée grâce à un test d'homomorphisme. Il nous faut maintenant étendre ce test pour prendre en compte le pré-ordre $\preccurlyeq$.

Définition 5.7 ($\preccurlyeq$-homomorphisme) *Soit $\mathcal{A}$ et $\mathcal{B}$ deux ensembles d'atomes et un pré-ordre $\preccurlyeq$ sur les atomes. Un $\preccurlyeq$-homomorphisme de $\mathcal{B}$ dans $\mathcal{A}$ est une substitution h de $\text{var}(\mathcal{B})$ dans $\text{term}(\mathcal{A})$ telle que pour tout $B \in \mathcal{B}$, il existe $A \in \mathcal{A}$ avec $A \preccurlyeq h(B)$.*

Exemple 53 *Nous considérons à nouveau l'ensemble de règles $\mathcal{R}_{\preccurlyeq}$ de l'exemple 50. Soit $Q_1 = t(u,v) \wedge q(v)$ et $Q_2 = q(w) \wedge s(z,w) \wedge c(w)$. La substitution $h = \{(u,w),(v,z)\}$ est un $\preccurlyeq$-homomorphisme de Q_1 dans Q_2. En effet, $t(u,v)$ et $q(v)$ sont tous deux envoyés sur $s(z,w)$, en utilisant $R_2 = s(x,y) \rightarrow t(y,x)$ et $R_2 \bullet R_4 = s(x,y) \rightarrow q(x)$ respectivement.*

Il est important de noter le lien entre la relation $\preccurlyeq$ sur les ensembles d'atomes et la présence d'un $\preccurlyeq$-homomorphisme.

Propriété 10 *Soit $\mathcal{A}$ et $\mathcal{B}$ deux ensembles d'atomes et un pré-ordre $\preccurlyeq$ sur les atomes. Si $\mathcal{A} \preccurlyeq \mathcal{B}$ alors il existe un $\preccurlyeq$-homomorphisme de $\mathcal{B}$ dans $\mathcal{A}$.*

Preuve : $\mathcal{A} \preccurlyeq \mathcal{B}$ signifie qu'il existe une application surjective f de $\mathcal{B}$ dans $\mathcal{A}$ telle que pour tout $B \in \mathcal{B}$, $f(B) \preccurlyeq B$. Il faut noter que les termes de B sont inclus dans les termes de $f(B)$ et que donc la substitution identité id est une substitution des variables de $\mathcal{B}$ dans les termes de $\mathcal{A}$ telle que pour chaque atome B de $\mathcal{B}$ il existe $f(B)$ un atome de $\mathcal{A}$ tel que $f(B) \preccurlyeq id(B)$, la substitution id est donc un $\preccurlyeq$-homomorphisme. □

5.4 Prise en compte d'un pré-ordre sur les atomes pour l'unification

Nous définissons maintenant la notion d'$\preccurlyeq$-unificateur par pièce qui est un unificateur par pièce prenant en compte un pré-ordre $\preccurlyeq$ sur les atomes induit par un ensemble de règles compilables ou hiérarchiques.

Définition 5.8 ($\preccurlyeq$-unificateur par pièce) *Soit $\preccurlyeq$ un pré-ordre sur les atomes, Q une requête et R une règle. Un* $\preccurlyeq$-unificateur *par pièce de Q avec R est un triplet $\mu_{\preccurlyeq} = (Q', C', P)$ où Q' est un sous-ensemble de Q, C' est un sous-ensemble de $\text{concl}(R)$ et P est une partition sur les termes de Q' et C' telle que :*

- *P est admissible.*
- *Si une classe de P contient une variable existentielle de R les autres termes qu'elle contient sont des variables non séparatrices de Q'.*
- *$P(C') \preccurlyeq P(Q')$.*

Les variables qui apparaissent dans la classe d'une variable existentielle de R sont appelées variables liantes *et notées $\text{glue}(\mu_{\preccurlyeq})$.*

Le calcul d'une réécriture produite par un $\preccurlyeq$-unificateur est le même que celui d'une réécriture produite par un unificateur par pièce classique : $\beta(Q, R, \mu_{\preccurlyeq}) = P(hyp(R)) \cup P(Q \setminus Q')$.

5.5 Introduction de l'opérateur de réécriture $rew_{\preccurlyeq}$

Le seul changement dans l'opérateur de réécriture consiste donc à remplacer les unificateurs par pièce par des $\preccurlyeq$-unificateurs par pièce. On appelle $rew_{\preccurlyeq}$ l'opérateur de réécriture qui, selon un pré-ordre $\preccurlyeq$ et un ensemble de règles $\mathcal{R}$, associe à une requête Q toutes les réécritures produites par un $\preccurlyeq$-unificateur par pièce de Q avec une règle de $\mathcal{R}$.

Exemple 54 *Nous considérons à nouveau les règles de l'exemple 50 ainsi que la règle* $R = b(x) \rightarrow t(x, y)$ *et la requête* $Q_1 = t(u, v) \wedge q(v)$. *Il n'y a pas d'unificateur par pièce de* Q_1 *avec* R *mais il y a un* $\preccurlyeq$*-unificateur par pièce* $\mu = (Q_1, concl(R), \{\{x, u\}, \{y, v\}\})$ *de* Q_1 *avec* R, *grâce à* $R_4 = t(x, y) \rightarrow q(y)$ *qui subsume* $t(u, v) \rightarrow q(v)$. *La 1-réécriture de* Q_1 *selon* μ *est* $\beta(Q_1, R, \mu) = b(u)$.

On considère maintenant $Q_2 = q(w) \wedge s(z, w) \wedge c(w)$. *Avec le* $\preccurlyeq$*-unificateur par pièce* $\mu' = (\{q(w), s(z, w)\}, concl(R), \{\{x, w\}, \{y, z\}\})$ *de* Q_2 *avec* R, *grâce à* $R_3 = t(x, y) \rightarrow q(x)$ *et* $R'_2 = t(x, y) \rightarrow s(y, x)$, *on obtient* $\beta(Q_2, R, \mu') = b(w) \wedge c(w)$.

Nous souhaitons utiliser l'opérateur $rew_{\preccurlyeq}$ dans l'algorithme 1 en association avec une fonction de couverture utilisant non plus l'homomorphisme mais l'$\preccurlyeq$-homomorphisme. Nous allons pour cela nous attacher à prouver l'adéquation et la complétude de l'opérateur $rew_{\preccurlyeq}$ associé au test d'$\preccurlyeq$-homomorphisme. Enfin, nous prouverons l'élagabilité de l'opérateur $rew_{\preccurlyeq}$ vis à vis du test d'$\preccurlyeq$-homomorphisme.

La propriété suivante fait le lien entre l'implication logique et la notion d'$\preccurlyeq$-homomorphisme.

Propriété 11 *Soit* $\mathcal{A}$ *et* $\mathcal{B}$ *deux ensembles d'atomes et* $\mathcal{R}_{\preccurlyeq}$ *un ensemble de règles compilables et son pré-ordre* $\preccurlyeq$ *associé. Il existe un* $\preccurlyeq$*-homomorphisme de* $\mathcal{B}$ *dans* $\mathcal{A}$ *si et seulement si* $(\Phi(\mathcal{A}), \mathcal{R}_{\preccurlyeq}) \models \Phi(\mathcal{B})$, *où* $\Phi(\mathcal{A})$ *et* $\Phi(\mathcal{B})$ *sont les formules existentiellement close assignées à* $\mathcal{A}$ *et* $\mathcal{B}$.

Preuve :
$\Rightarrow$ Nous supposons qu'il existe un $\preccurlyeq$-homomorphisme de $\mathcal{B}$ dans $\mathcal{A}$, c'est-à-dire une substitution h de $var(\mathcal{B})$ dans $term(\mathcal{A})$ telle que pour tout $B \in \mathcal{B}$, il y a $A \in \mathcal{A}$ avec $A \preccurlyeq h(B)$; puisque la règle $A \rightarrow h(B)$ est une conséquence de $\mathcal{R}_{\preccurlyeq}$, $h(B)$ est une conséquence de A et $\mathcal{R}_{\preccurlyeq}$. Ainsi, $(\Phi(\mathcal{A}), \mathcal{R}_{\preccurlyeq}) \models \Phi(h(\mathcal{B}))$. Puisque $\Phi(h(\mathcal{B})) \models \Phi(\mathcal{B})$, on obtient $(\Phi(\mathcal{A}), \mathcal{R}_{\preccurlyeq}) \models \Phi(\mathcal{B})$.
$\Leftarrow$ Nous supposons que $(\Phi(\mathcal{A}), \mathcal{R}_{\preccurlyeq}) \models \Phi(\mathcal{B})$. Donc, il existe $\mathcal{B}'$ une k-réécriture de $\mathcal{B}$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$ telle qu'il existe un homomorphisme h de $\mathcal{B}'$ dans $\mathcal{A}$ (donc, $h(\mathcal{B}') \subseteq \mathcal{A}$). Puisque $\mathcal{B}'$ est une k-réécriture de $\mathcal{B}$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$, il existe une

application surjective f de $\mathcal{B}$ dans $\mathcal{B}'$ telle que, pour tout $B \in \mathcal{B}$, $f(B) \preccurlyeq B$ (donc, $f(\mathcal{B}) = \mathcal{B}'$). A partir de $h(\mathcal{B}') \subseteq \mathcal{A}$ et $f(\mathcal{B}) = \mathcal{B}'$, on obtient $h(f(\mathcal{B})) \subseteq \mathcal{A}$. Puisque pour tout $B \in \mathcal{B}$, on a $f(B) \preccurlyeq B$, on peut en déduire que $h(f(B)) \preccurlyeq h(B)$. Puisque $h(f(B)) \in \mathcal{A}$, pour tout $B \in \mathcal{B}$, il existe $A \in \mathcal{A}$ avec $A \preccurlyeq h(B)$. Ainsi, h est un $\preccurlyeq$-homomorphisme de $\mathcal{B}$ dans $\mathcal{A}$. □

Les deux lemmes suivants établissent la correspondance entre une 1-réécriture avec un $\preccurlyeq$-unificateur par pièce et une séquence de réécritures avec les unificateurs par pièce classiques. Ils montrent qu'un $\preccurlyeq$-unificateur peut être remplacé par une séquence d'unificateurs par pièce classiques avec d'abord des règles compilables puis une règle non compilable. La réciproque n'est pas vrai, une séquence d'unificateurs par pièce classiques avec d'abord des règles compilables puis une règle non compilable n'est pas remplacée par un $\preccurlyeq$-unificateur seulement mais par un $\preccurlyeq$-unificateur suivi d'une séquence d'unificateurs par pièce classiques avec des règles compilables.

Lemme 6 *Soit Q une requête, R une règle et $\mathcal{R}_{\preccurlyeq}$ un ensemble de règles compilables et son pré-ordre $\preccurlyeq$ associé. S'il existe un $\preccurlyeq$-unificateur par pièce $\mu_{\preccurlyeq}$ de Q avec une règle R produisant $Q_2 = \beta(Q, R, \mu_{\preccurlyeq})$, alors il existe une k-réécriture Q_1 de Q selon $\mathcal{R}_{\preccurlyeq}$ et rew_β telle que Q_2 est une 1-réécriture selon un unificateur par pièce de Q_1 avec R.*

Preuve : Soit $\mu_{\preccurlyeq} = (Q', C', P_{\preccurlyeq})$. On sait que $P_{\preccurlyeq}(C') \preccurlyeq P_{\preccurlyeq}(Q')$. Grâce à la propriété 9, il existe Q'' une k-réécriture de $P_{\preccurlyeq}(Q')$ selon $\mathcal{R}_{\preccurlyeq}$ et rew_β qui s'envoie sur $P_{\preccurlyeq}(C')$ par une substitution s de $var(Q'') \setminus var(P_{\preccurlyeq}(Q'))$ dans $term(P_{\preccurlyeq}(C'))$ telle que $s(Q'') = P_{\preccurlyeq}(C')$. En se rapportant à la preuve du lemme 3 et au fait que $Q' \geq P_{\preccurlyeq}(Q')$, on prouve qu'il existe une k-réécriture Q'_1 de Q' selon $\mathcal{R}_{\preccurlyeq}$ et rew_β et h un homomorphisme de Q'_1 dans Q'' telle que $h(Q'_1) = Q''$. On appelle Q_1 la k-réécriture de Q selon $\mathcal{R}_{\preccurlyeq}$ et rew_β obtenue en réécrivant Q' en Q'_1 ($Q' \subseteq Q$), et P_1 la partition associée à cette réécriture (c'est-à-dire la jointure des partitions de chacun des unificateurs par pièce utilisés pour produire Q_1). On a $Q_1 = P_1(Q \setminus Q') \cup Q'_1$.

De plus, on sait que $s(h(Q'_1)) = P_{\preccurlyeq}(C')$. Soit u_n la substitution de $var(C') \cup var(Q'_1)$ dans $term(C') \cup term(Q'_1)$ construite de la manière suivante :

- $\forall x \in Q'_1$, $u_n(x) = s(h(x))$
- $\forall x \in C'$, $u_n(x) = P_{\preccurlyeq}(x)$

Donc, $u_n(Q'_1) = u_n(C')$. Nous construisons maintenant $\mu_n(Q'_1, C'_{\preccurlyeq}, P_n)$, un unificateur par pièce de Q_1 avec R, où P_n est la partition associée à u_n. Il nous reste à prouver que $Q_2 = \beta(Q_1, R, \mu_n)$. On sait que $Q_2 = P_{\preccurlyeq}(hyp(R)) \cup P_{\preccurlyeq}(Q \setminus Q')$ et $\beta(Q_1, R, \mu_n) = P_n(hyp(R)) \cup P_n(Q_1 \setminus Q'_1)$. Par construction de u_n, $P_n(hyp(R)) = P_{\preccurlyeq}(hyp(R))$, donc il reste à vérifier que $P_n(Q_1 \setminus Q'_1) = P_{\preccurlyeq}(Q \setminus Q')$. Puisque $Q_1 = P_1(Q \setminus Q') \cup Q'_1$, on a $P_n(Q_1 \setminus Q'_1) = P_n(P_1(Q \setminus Q'))$, et, par construction de u_n, $P_n(P_1(Q \setminus Q')) = s(h(P_1(Q \setminus Q')))$. En raison du domaine de s, $s(h(P_1(Q \setminus Q'))) = h(P_1(Q \setminus Q'))$.

Nous prouvons maintenant que $\forall x \in var(Q \setminus Q')$, $P_{\preccurlyeq}(x) = h(P_1(x))$:

- si $x \notin var(Q')$, $P_{\preccurlyeq}(x) = x$ et $h(u_1(x)) = x$
- sinon, $x \in var(Q')$,
  - si $x \notin var(Q'_s)$, où Q'_s est la partie de Q' réécrite avec P_1, $P_1(x) = x$ et, par construction de h dans la preuve du lemme 3 assurant l'élagabilité de rew_β, on sait que $h(x) = P_{\preccurlyeq}(x)$,
  - sinon, $x \in var(Q'_s)$, et dans la même preuve du lemme 3, on sait que $h(P_1(x)) = P_2(P_{\preccurlyeq}(x))$, où P_2 est la partition utilisée pour réécrire $P_{\preccurlyeq}(Q')$ en Q'_s. Comme on peut le vérifier dans la preuve de la propriété 9, P_2 ne contient aucune variable de $P_{\preccurlyeq}(Q')$, donc, $P_2(P_{\preccurlyeq}(x)) = P_{\preccurlyeq}(x)$.

□

Le lemme suivant n'est pas, comme on pourrait s'y attendre la réciproque du précédent. En effet, comme on peut le voir dans l'exemple suivant, tous les unificateurs par pièce avec une règle compilable ne sont pas remplacés par l'$\preccurlyeq$-unificateur par pièce.

Exemple 55 *Soit $R = p(x) \rightarrow r(x, y)$ une règle, $\mathcal{R}_{\preccurlyeq} = \{R_1, R_2\}$ un ensemble de règles compilables où $R_1 = r(x, y) \rightarrow s(x, y)$ et $R_2 = q(x) \rightarrow t(x)$, et $Q = s(u, v) \wedge t(u)$ une requête. Si on considère la séquence de réécritures suivante utilisant une séquence d'unificateurs par pièce avec une règle compilable, puis un unificateur par pièce avec R :*

- *à partir de $Q = s(u, v) \wedge t(u)$ et $R_2 = q(x) \rightarrow t(x)$ avec $\mu_1 = (\{t(u)\}, \{t(x)\}, \{\{u, x\}\})$ on produit $Q_1 = s(u, v) \wedge q(u)$,*
- *à partir de $Q_1 = s(u, v) \wedge q(u)$ et $R_1 = r(x, y) \rightarrow s(x, y)$ avec $\mu_2 = (\{s(u, v)\}, \{s(x, y)\}, \{\{u, x\}, \{v, y\}\})$ on produit $Q_2 = r(u, v) \wedge q(u)$,*
- *à partir de $Q_2 = r(u, v) \wedge q(u)$ et $R = p(x) \rightarrow r(x, y)$ avec $\mu_3 = (\{r(u, v)\}, \{r(x, y)\}, \{\{u, x\}, \{v, y\}\})$ on produit $Q_3 = p(u) \wedge q(u)$.*

On peut remplacer cette séquence par une séquence employant un $\preccurlyeq$-unificateur par pièce à la place de μ_2 et μ_3, néanmoins il faudra ensuite appliquer μ_1 pour obtenir Q_3

- *à partir de $Q = s(u, v) \wedge t(u)$ et $R = p(x) \rightarrow r(x, y)$ avec $\mu_{\preccurlyeq} = (\{s(u, v)\}, \{r(x, y)\}, \{\{u, x\}, \{\{v, y\}\}\})$ on produit $Q'_1 = p(u) \wedge t(u)$,*
- *à partir de $Q'_1 = p(u) \wedge t(u)$ et $R_2 = q(x) \rightarrow t(x)$ avec $\mu_1 = (\{t(u)\}, \{t(x)\}, \{\{u, x\}\})$ on produit $Q_3 = p(u) \wedge q(u)$.*

Une séquence d'unificateurs par pièce avec des règles compilables suivie par un unificateur par pièce sur une règle quelconque est donc remplacée par une séquence composée d'un $\preccurlyeq$-unificateur par pièce suivi par les unificateurs par pièce qui ne sont pas remplacés par l'$\preccurlyeq$-unificateur par pièce.

Lemme 7 *Soit Q une requête, R une règle et $\mathcal{R}_{\preccurlyeq}$ un ensemble de règles compilables et son pré-ordre $\preccurlyeq$ associé. S'il existe une k-réécriture Q_1 de Q selon rew_β et $\mathcal{R}_{\preccurlyeq}$ et une 1-réécriture Q_2 par un unificateur par pièce de Q_1 avec R, alors il existe un $\preccurlyeq$-unificateur par pièce $\mu_{\preccurlyeq}$ de Q avec R tel que Q_2 est une k-réécriture de $\beta(Q, R, \mu_{\preccurlyeq})$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$.*

Preuve : On prouve que

1. Il existe un $\preccurlyeq$-unificateur par pièce $\mu_{\preccurlyeq}$ de Q avec R.
2. Q_2 est une k-réécriture de $\beta(Q, R, \mu_{\preccurlyeq})$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$.

1. Q_1 est une k-réécriture de Q selon rew_β et $\mathcal{R}_{\preccurlyeq}$, donc chaque atome de Q_1 est un atome de Q, ou une réécriture d'un atome de Q par une règle de $\mathcal{R}^*_{\preccurlyeq}$, et donc on peut construire une application surjective f de Q dans Q_1 qui à un atome de Q associe l'atome par lequel il est réécrit dans Q_1. De la même manière, chaque terme de Q_1 est un terme de Q ou la réécriture d'au moins un terme de Q, il y a donc une substitution surjective f' des variables de Q dans les termes de Q_1. Ces deux substitutions sont telles que pour tout $B \in Q$, $f(B) \preccurlyeq f'(B)$, et comme $f(Q) = Q_1$, on a $Q_1 \preccurlyeq f'(Q)$.

   Soit $\mu_1 = (Q'_1, C', P_1)$ l'unificateur par pièce de Q_1 avec R produisant Q_2. Soit $\mu_{\preccurlyeq} = (Q', C', P_{\preccurlyeq})$, où $Q' = \{A \in Q \mid f(A) \in Q'_1\}$ ou de manière équivalente $Q' = \{A \in Q \mid f'(A) \in Q'_1\}$, $P_{\preccurlyeq} = f'(P_1)$, f' est donc une substitution préservante de $P_{\preccurlyeq}$ dans P_1 et la substitution complémentaire s de $P_{\preccurlyeq}$ dans P_1 selon f' est une bijection (voir section 4.2).

   Nous prouvons maintenant que $\mu_{\preccurlyeq}$ est un $\preccurlyeq$-unificateur par pièce de Q avec R. Les deux premières conditions de la définition des unificateurs par pièce sont respectées par P_1 donc par construction, $P_{\preccurlyeq}$ les respecte aussi. Il nous reste à prouver que $P_{\preccurlyeq}(C') \preccurlyeq P_{\preccurlyeq}(Q')$. Par construction de Q', on a $Q'_1 \preccurlyeq f'(Q')$ donc $P_1(Q'_1) \preccurlyeq P_1(f'(Q'))$. Par la définition de la substitution complémentaire on a $P_1(f'(Q')) = s(P_{\preccurlyeq}(Q'))$ donc $P_1(Q'_1) \preccurlyeq s(P_{\preccurlyeq}(Q'))$. Comme $P_1(Q'_1) = P_1(C')$, on a $P_1(C') \preccurlyeq s(P_{\preccurlyeq}(Q'))$. Enfin, comme s est une bijection, s^{-1} est telle que pour tout $t \in var(s(Q'_1 \cup C'))$, $s^{-1}(P_1(t)) = P_{\preccurlyeq}(t)$, donc $P_{\preccurlyeq}(C') \preccurlyeq P_{\preccurlyeq}(Q')$.

2. On sait que $Q_2 = P_1(hyp(R)) \cup P_1(Q_1 \setminus Q'_1)$ et $\beta(Q, R, \mu_{\preccurlyeq}) = P_{\preccurlyeq}(hyp(R)) \cup P_{\preccurlyeq}(Q \setminus Q') = s^{-1}(P_1(hyp(R))) \cup s^{-1}(P_1(Q \setminus Q'))$. s^{-1} est une bijection qui renomme des variables. Il reste à prouver que $P_1(Q_1 \setminus Q'_1)$ est une k-réécriture de $P_1(Q \setminus Q')$ selon $\mathcal{R}_{\preccurlyeq}$ et rew_β. On sait que Q_1 (respectivement Q'_1) est une k-réécriture de Q (respectivement Q') selon rew_β et $\mathcal{R}_{\preccurlyeq}$, donc $Q_1 \setminus Q'_1$ est une k-réécriture de $Q \setminus Q'$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$. On peut en conclure que $P_1(Q_1 \setminus Q'_1)$ est une k-réécriture de $P_1(Q \setminus Q')$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$.

□

Le prochain théorème assure que $rew_{\preccurlyeq}$ associé au test d'$\preccurlyeq$-homomorphisme est adéquat et complet par rapport à la conséquence logique.

Théorème 10 (Adéquation et complétude de $rew_{\preccurlyeq}$) *Soit $\mathcal{K} = (F, \mathcal{R})$ une base de connaissances, où $\mathcal{R}$ peut être partitionné en $\mathcal{R}_e$ et $\mathcal{R}_{\preccurlyeq}$ un ensemble de règles compilables et son pré-ordre $\preccurlyeq$ associé. Soit Q une requête. On a $(F, \mathcal{R}) \models Q$ si et seulement s'il existe $Q' \in \mathcal{W}_\infty^{\preccurlyeq}(Q, \mathcal{R})$ et un $\preccurlyeq$-homomorphisme de Q' dans F.*

Preuve : On rappelle que $\mathcal{W}_\infty^{\preccurlyeq}(Q, \mathcal{R})$ représente l'ensemble des $\leq k$-réécritures de Q selon $rew_{\preccurlyeq}$ et $\mathcal{R}$ pour tout $k \in \mathbb{N}$ et $\mathcal{W}_\infty^{\beta}(Q, \mathcal{R})$ représente l'ensemble des $\leq k$-réécritures de Q selon rew_β et $\mathcal{R}$ pour tout $k \in \mathbb{N}$.

Nous nous appuyons sur l'adéquation et la complétude des unificateurs par pièce et nous prouvons que :

1. Si $Q' \in \mathcal{W}_\infty^{\preccurlyeq}(Q, \mathcal{R}_e)$ alors $Q' \in \mathcal{W}_\infty^{\beta}(Q, \mathcal{R})$.
2. Si $Q' \in \mathcal{W}_\beta^{\infty}(Q, \mathcal{R})$ tel que $F \models Q'$ alors il existe $Q'' \in \mathcal{W}_\infty^{\preccurlyeq}(Q, \mathcal{R}_e)$ telle qu'il existe un $\preccurlyeq$-homomorphisme de Q'' dans F.

1. A partir du lemme 6, chaque $\preccurlyeq$-unificateur par pièce d'une séquence menant à Q' peut être remplacé par une séquence d'unificateurs par pièce.
2. Une séquence d'unificateurs par pièce menant à Q' peut être décomposée en sous-séquences, telles que chaque sous-séquence est composée d'une séquence (pouvant être vide) d'unificateurs par pièce utilisant des règles de $\mathcal{R}_{\preccurlyeq}$ suivi d'un unificateur par pièce utilisant une règle de $\mathcal{R}_e$. Selon le lemme 7, chaque sous-séquence peut être remplacée par un $\preccurlyeq$-unificateur par pièce suivi d'unificateurs par pièce utilisant des règles de $\mathcal{R}_{\preccurlyeq}$; ces derniers unificateurs par pièce sont ajoutés à la sous-séquence suivante. A la fin, nous obtenons une séquence d'$\preccurlyeq$-unificateurs par pièce menant à Q'', suivi d'une séquence (pouvant être vide) d'unificateurs par pièce utilisant des règles de $R_{\preccurlyeq}$ et menant à Q'. Ainsi, Q' est une k-réécriture de Q'' selon rew_β et $\mathcal{R}_{\preccurlyeq}$ donc $(Q', \mathcal{R}_{\preccurlyeq}) \models Q''$. Or $F \models Q'$ donc $(F, \mathcal{R}_{\preccurlyeq}) \models Q''$. Par la propriété 11, si $(F, \mathcal{R}_{\preccurlyeq}) \models Q''$, alors il existe un $\preccurlyeq$-homomorphisme de Q'' dans F.

□

Enfin, la propriété suivante assure l'élagabilité de $rew_{\preccurlyeq}$ vis-à-vis du test d'$\preccurlyeq$-homomorphisme. Dans la définition 3.8, on introduisait la notion d'élagabilité avec le test d'homomorphisme, ici, on étend cette notion pour prendre en compte le pré-ordre en remplaçant l'homomorphisme par l'$\preccurlyeq$-homomorphisme.

Propriété 12 (Élagabilité) *Soit Q_1 et Q_2 deux requêtes, R une règle, et $\preccurlyeq$ un pré-ordre sur les atomes. S'il existe un $\preccurlyeq$-homomorphisme de Q_1 dans Q_2 alors pour tout $\preccurlyeq$-unificateur par pièce $\mu_{\preccurlyeq 2}$ de Q_2 avec R, soit il existe un $\preccurlyeq$-homomorphisme de Q_1 dans $\beta(Q_2, R, \mu_{\preccurlyeq 2})$, soit il existe un $\preccurlyeq$-unificateur par pièce $\mu_{\preccurlyeq 1}$ de Q_1 avec R tel qu'il existe un $\preccurlyeq$-homomorphisme de $\beta(Q_1, R, \mu_{\preccurlyeq 1})$ dans $\beta(Q_2, R, \mu_{\preccurlyeq 2})$.*

Preuve : Soit $Q'_2 = \beta(Q_2, R, \mu_{\preccurlyeq 2})$.
On suppose qu'il existe un $\preccurlyeq$-homomorphisme de Q_1 dans Q_2. De part la propriété 11, on a $\Phi(Q_2), \mathcal{R}_{\preccurlyeq} \models \Phi(Q_1)$, et grâce à la correction du mécanisme de réécriture avec les unificateurs par pièce, on sait qu'il existe une k-réécriture Q_3 de Q_1 selon rew_β et $\mathcal{R}_{\preccurlyeq}$ telle qu'il existe un homomorphisme de Q_3 dans Q_2. De part le lemme 6 en prenant $Q = Q_2$, on peut déduire qu'il existe une k-réécriture Q''_2 de Q_2 selon rew_β et $\mathcal{R}_{\preccurlyeq}$ telle que Q'_2 est une 1-réécriture selon un unificateur par pièce de Q''_2 avec R. Grâce à l'élagabilité de rew_β (lemme 3), et puisque $Q_3 \geq Q_2$, on sait qu'il existe une k-réécriture Q''_3 de Q_3 selon rew_β et $\mathcal{R}_{\preccurlyeq}$ et une 1-réécriture Q'_3 selon un unificateur par pièce de Q''_3 avec R, telle que $Q''_3 \geq Q''_2$ et $Q'_3 \geq Q'_2$. Enfin, puisque Q_3 est une k-réécriture de Q_1 selon rew_β et $\mathcal{R}_{\preccurlyeq}$ et puisque Q''_3 est une k-réécriture de Q_3 selon rew_β et $\mathcal{R}_{\preccurlyeq}$, on peut en déduire que Q''_3 est une k-réécriture de Q_1 selon rew_β et $\mathcal{R}_{\preccurlyeq}$. Par le lemme 7, on sait qu'il existe un $\preccurlyeq$-unificateur par pièce $\mu_{\preccurlyeq 1}$ de Q_1 avec R tel que Q'_3 est une k-réécriture de $Q'_1 = \beta(Q_1, R, \mu_{\preccurlyeq 1})$ selon rew_β et $\mathcal{R}_{\preccurlyeq}$. Puisque $Q'_3 \geq Q'_2$, on peut conclure à partir de la propriété 9 (l'homomorphisme de Q'_3 dans Q'_2 vérifie les conditions de la substitution s de la propriété si on considère que l'on garde en priorité les variables de la requête lors de la réécriture) que $Q'_2 \preccurlyeq Q'_1$ et donc qu'il existe un $\preccurlyeq$-homomorphisme de Q'_1 dans Q'_2 (propriété 10). □

Nous avons donc prouvé que l'algorithme 1 utilisant $rew_{\preccurlyeq}$ comme opérateur et l'$\preccurlyeq$-homomorphisme pour le test de subsomption fournit un ensemble de réécritures adéquat et complet si on utilise l'$\preccurlyeq$-homomorphisme pour l'interrogation des données. Cet ensemble de réécritures sera nommé UCQ-pivot. En effet, comme nous allons le voir dans la section suivante, cette UCQ est à l'intersection entre plusieurs représentations. Voici pour finir un exemple de calcul d'UCQ-pivot.

Exemple 56 *Soit* $\mathcal{R} = \mathcal{R}_{\preccurlyeq} \cup \mathcal{R}_e$ *et* $Q = D(u) \wedge s(u, v)$ *où* $\mathcal{R}_{\preccurlyeq} = \{R_1, R_2, R_3, R_4\}$ *et* $\mathcal{R}_e = \{R_5, R_6\}$ *avec :*

$$R_1 = A(x) \rightarrow B(x)$$
$$R_2 = r(x, y) \rightarrow s(x, y)$$
$$R_3 = t(x, y) \rightarrow r(x, y)$$
$$R_4 = C(x) \rightarrow D(x)$$
$$R_5 = A(x) \rightarrow r(x, y)$$
$$R_6 = B(x) \rightarrow t(x, y)$$

La saturation de $\mathcal{R}_{\preccurlyeq}$ *produit une seule nouvelle règle* $R_3 \bullet R_2 = t(x, y) \rightarrow s(x, y)$. *Soit* $\preccurlyeq$ *le pré-ordre calculé à partir de* $\mathcal{R}_{\preccurlyeq}$. *Il y a deux* $\preccurlyeq$*-unificateurs par pièce de* Q *avec les règles de* $\mathcal{R}_e$ *:*

- $\mu_{\preccurlyeq 1} = (\{s(u, v)\}, \{r(x, y)\}, \{(u, x), (v, y)\})$ *est un* $\preccurlyeq$*-unificateur par pièce de* Q *avec* R_5 *grâce à* R_2. $\beta(Q, R_5, \mu_{\preccurlyeq 1}) = Q_1 = D(u) \wedge A(u)$.

- $\mu_{\preccurlyeq 2} = (\{s(u,v)\}, \{t(x,y)\}, \{(u,x),(v,y)\})$ *est un* $\preccurlyeq$*-unificateur par pièce de* Q *avec* R_6 *grâce à* $R_3 \bullet R_2$. $\beta(Q, R_6, \mu_{\preccurlyeq 2}) = Q_2 = D(u) \wedge B(u)$.

Or il y a un $\preccurlyeq$*-homomorphisme de* Q_2 *dans* Q_1 *et* Q_1 *est donc éliminée. Par ailleurs* Q *et* Q_2 *sont incomparables par* $\preccurlyeq$*-homomorphisme. Comme il n'y a pas d'*$\preccurlyeq$*-unificateur par pièce de* Q_2 *avec une règle de* $\mathcal{R}_e$*, l'UCQ-pivot finale est donc* $\{Q, Q_2\}$.

5.6 Évaluation d'une UCQ-pivot

Nous nous intéressons maintenant aux diverses façons d'exploiter une UCQ-pivot en phase d'évaluation sur des données : soit directement, soit par transformation en d'autres formes de requêtes, selon le type de système associé à la base de données.

Évaluation directe

On peut envisager deux manières d'évaluer la requête sans avoir besoin de la transformer. Comme nous l'avons prouvé dans la section précédente, l'UCQ-pivot produite avec $rew_{\preccurlyeq}$ et l'algorithme 1 est adéquate et complète si on utilise l'$\preccurlyeq$-homomorphisme pour l'interrogation. Le premier moyen d'évaluer une UCQ-pivot est donc d'utiliser un système implémentant l'$\preccurlyeq$-homomorphisme.

Une autre solution simple consiste à appliquer en marche avant, jusqu'à saturation, les règles compilées sur la base de faits. Comme les règles compilées ne contiennent pas de variables existentielles, le processus est fini. On pourra ensuite évaluer par un homomorphisme simple l'UCQ-pivot sur cette base de faits saturée.

Transformation en UCQ classique par déploiement

Une UCQ classique $\mathcal{Q}$' est obtenue à partir de l'UCQ-pivot $\mathcal{Q}$ en ajoutant pour chaque $Q \in \mathcal{Q}$ toutes les requêtes Q' telles que $Q' \preccurlyeq Q$. On appellera cette phase le *déploiement* de $\mathcal{Q}$. On peut simplement construire Q' à partir de Q en choisissant, grâce au pré-ordre $\preccurlyeq$, pour chaque atome $A \in Q$ un atome A' construit de la manière suivante : pour chaque règle $R \in \mathcal{R}_c^*$ telle qu'il existe un homomorphisme π de $concl(R)$ dans A, on génère l'atome $\pi^{safe}(hyp(R))$ (π^{safe} est l'extension de l'homomorphisme π qui substitue les variables n'appartenant pas au domaine de π par de nouvelles variables n'apparaissant nulle part ailleurs). Pour que $\mathcal{Q}$' soit minimale, il restera à calculer la couverture avec le test d'homomorphisme classique.

Exemple 56 (Suite) *Nous reprenons l'exemple précédent pour lequel nous allons déployer l'UCQ-pivot* $\{Q, Q_2\}$.

$Q = D(u) \wedge s(u,v)$. *Les atomes* $\preccurlyeq$ *aux atomes de* Q *sont respectivement* $\{D(u), C(u)\}$ *et* $\{s(u,v), r(u,v), t(u,v)\}$. *Les requêtes déployées à partir de* Q *sont donc* $\{D(u) \wedge s(u,v), D(u) \wedge r(u,v), D(u) \wedge t(u,v), C(u) \wedge s(u,v), C(u) \wedge r(u,v), C(u) \wedge t(u,v)\}$.

$Q_2 = D(u) \wedge B(u)$. Les atomes $\preccurlyeq$ aux atomes de Q_2 sont respectivement $\{D(u), C(u)\}$ et $\{B(u), A(u)\}$. Les requêtes déployées à partir de Q_2 sont donc $\{D(u) \wedge B(u), C(u) \wedge B(u), D(u) \wedge A(u), C(u) \wedge A(u)\}$.

Aucune des requêtes déployées n'en subsume une autre. L'UCQ finale classique est donc $\{D(u) \wedge s(u,v), D(u) \wedge r(u,v), D(u) \wedge t(u,v), C(u) \wedge s(u,v), C(u) \wedge r(u,v), C(u) \wedge t(u,v), D(u) \wedge B(u), C(u) \wedge B(u), D(u) \wedge A(u), C(u) \wedge A(u)\}$

Il est clair que l'évaluation directe de l'UCQ-pivot sur la base de données saturée par les règles compilables est bien plus efficace que l'évaluation de l'UCQ classique associée, qui peut-être exponentiellement plus grande (voir exemple 48). Ceci nécessite cependant de saturer la base de données, il faut donc avoir accès en écriture aux données. D'autre part, il peut y avoir un problème de taille de la base de données saturée.

Lorsque les règles compilables sont de simples règles hiérarchiques, on peut éviter la saturation en utilisant des techniques d'index sémantiques [Rodriguez-Muro and Calvanese, 2012] . Il ne semble pas cependant que cette technique soit extensible aux règles compilables non hiérarchiques.

Transformation en union de requêtes semi-conjonctives

Une requête semi-conjonctive (SCQ) est une conjonction de disjonctions, avec des contraintes sur les occurrences des variables [Thomazo, 2013a]. Toute SCQ est équivalente à une UCQ tout en étant plus compacte. Une union de SCQ est donc équivalente à une UCQ. Chaque requête Q de l'UCQ-pivot peut être transformée en une requête semi-conjonctive en remplaçant chaque atome A de Q par la disjonction de tous les atomes A' tels que $A' \preccurlyeq A$.

Exemple 56 (Suite) *Nous reprenons l'exemple précédent, ainsi Q se traduit en une SCQ : $(D(u) \vee C(u)) \wedge (s(u,v) \vee r(u,v) \vee t(u,v))$*

Similairement pour Q_2 : $(D(u) \vee C(u)) \wedge (B(u) \vee A(u))$.

Finalement, l'union de SCQ associée à l'UCQ-pivot est l'union des deux SCQ : $\{(D(u) \vee C(u)) \wedge (s(u,v) \vee r(u,v) \vee t(u,v)), (D(u) \vee C(u)) \wedge (B(u) \vee A(u))\}$

Une union de SCQ est plus compacte qu'une UCQ, mais les disjonctions internes à une SCQ peuvent être coûteuses à calculer. Selon [Thomazo, 2013a], l'évaluation d'une union de SCQ reste cependant plus efficace que l'évaluation de l'UCQ associée.

Transformation en Programme Datalog

L'UCQ-pivot associée à l'ensemble de règles compilables peut être vue comme un programme Datalog. En effet, les règles compilables n'ont pas de variables existentielles, elles peuvent donc être directement traduites en règles Datalog. De la même manière, on sait qu'une UCQ est directement traduisible en un programme Datalog. Afin de ne conserver dans le programme Datalog que les règles indispensables, on

peut se servir du pré-ordre $\preccurlyeq$ pour ne garder que les règles de $\mathcal{R}_{\preccurlyeq}$ dont la conclusion est unifiable à un atome inférieur (au sens $\preccurlyeq$) à un atome de la requête. Une autre possibilité est de partir de $\mathcal{R}^*_{\preccurlyeq}$ et de ne conserver que les règles dont la conclusion est unifiable au sens logique classique à un atome de la requête. Le second ensemble obtenu est la saturation du premier, il nécessite donc moins de calcul de la part du système Datalog mais a une taille plus importante, il peut donc être intéressant de comparer le temps d'évaluation du programme Datalog avec ces deux ensembles de règles.

Exemple 56 (Suite) *Nous reprenons l'exemple précédent, le programme Datalog associé à l'UCQ pivot est :*

$$\begin{aligned} B(x) &: -A(x) \\ s(x,y) &: -r(x,y) \\ r(x,y) &: -t(x,y) \\ D(x) &: -C(x) \\ ans() &: -D(u) \wedge s(u,v) \\ ans() &: -D(u) \wedge B(u) \end{aligned}$$

Ici on peut remarquer que toutes les règles compilables sont nécessaires dans le programme Datalog.

Nous n'avons pas comparé l'évaluation d'une UCQ et celle d'une réécriture Datalog correspondante, principalement parce que cela nécessiterait de comparer des systèmes différents (SGBD relationnel et système Datalog). Ce que l'on mesurerait alors n'est pas clair, car entre en jeu l'efficacité de l'implémentation de chacun de ces systèmes. Par contre, il serait intéressant de comparer en termes de temps l'évaluation sur les données de la transformation Datalog de notre UCQ-pivot à l'évaluation des réécritures Datalog produites par d'autres systèmes si elles sont différentes.

Chapitre 6

Implémentation

Dans ce chapitre, nous commençons par décrire brièvement l'architecture des classes sur laquelle repose notre implémentation. Nous présentons alors la technique et les algorithmes utilisés pour le calcul des unificateurs mono-pièce. Nous détaillons ensuite les structures de données utilisées par la compilation des règles et la manière dont nous les exploitons pour le calcul des réécritures. Enfin nous terminons par quelques optimisations pratiques permettant d'accélérer le processus de réécriture.

L'implémentation de nos algorithmes a été faite en Java, à l'aide d'une API ALASKA développée au cours de la thèse de Bruno Paiva Lima da Silva [da Silva, 2014] et reprise dans une plateforme en cours de développement par Clément Sipieter. En particulier, cette API modélise un sous-ensemble positif de la logique du premier ordre, elle est composée d'un ensemble d'interfaces qui représentent les notions principales : terme, prédicat, atome, fait, requête et règle. Un atome, interface "IAtom" est naturellement composé d'un prédicat (interface "IPredicate") et d'un ensemble de termes (interface "ITerm"). Un fait, interface "IFact", est un ensemble d'atomes. Une règle, interface "IRule", possède une tête et un corps qui sont des faits. Et enfin, une requête, interface "IQuery", est un fait qui peut posséder un ensemble de variables réponse.

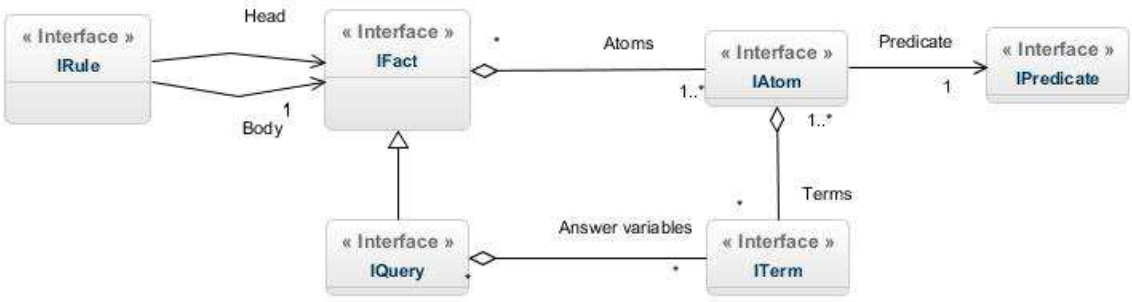

FIGURE 6.1 – Hiérarchie de l'API

A chacune de ces interfaces, nous avons ajouté une spécialisation par une classe concrète, ainsi que les notions qui nous étaient nécessaires : partition, substitution,

unificateur par pièce, compilation de règles et différents outils pour des tâches telles que l'acquisition de fichiers de règles et de requêtes et le calcul d'homomorphisme.

6.1 Calcul des unificateurs par pièce

Nous allons maintenant présenter les algorithmes qui permettent de calculer les unificateurs mono-pièce les plus généraux.

Pour nos algorithmes, nous allons avoir besoin de plusieurs notions. La première est la notion de *partition valide*, ce sont les partitions qui vérifient le fait qu'elles sont admissibles et qu'une variable existentielle contient dans sa classe uniquement des variables de la requête.

Définition 6.1 (Partition valide) *Soit Q une requête, R une règle, $Q' \subseteq Q$, $C' \subseteq concl(R)$, et P une partition sur $term(Q') \cup term(C')$. P est* valide *si aucune classe de P ne contient deux constantes, ou deux variables existentielles de R, ou une constante et une variable existentielle de R, ou une variable existentielle de R et une variable de la frontière de R.*

La méthodologie mise en place pour calculer les unificateurs consiste à construire chaque Q' en partant d'un atome de la requête que l'on cherche à unifier (au sens classique) à un atome de la conclusion de la règle (qui constituera son C' associé) et on vérifie que la partition induite est valide. Il reste alors à vérifier que les variables séparatrices ne sont pas fusionnées avec des variables existentielles. Si c'est le cas on étend alors Q' avec les atomes de Q qui contiennent les variables unifiées aux variables existentielles. On étend alors le C' afin de l'unifier au nouveau Q' en vérifiant la condition de validité de la partition. Et on itère avec les nouvelles variables séparatrices. La structure Q', C', P où P est la partition induite par l'unification classique des atomes de Q' et C' est appelée un pré-unificateur.

Définition 6.2 (Pré-unificateur) *Soit Q une requête, R une règle, $Q' \subseteq Q$, $C' \subseteq concl(R)$, et P une partition sur $term(Q') \cup term(C')$. $\mu = (Q', C', P)$ est un* pré-unificateur *de Q avec R si* (1) *P est valide, et* (2) *$P(C') = P(Q')$.*

La propriété suivante assure qu'un pré-unificateur dont l'ensemble des variables liantes et l'ensemble des variables séparatrices sont disjoints (voir définition 4.10) est un unificateur par pièce et réciproquement.

Propriété 13 *Soit Q une requête, R une règle, $Q' \subseteq Q$, $C' \subseteq concl(R)$, et P une partition sur $term(Q') \cup term(C')$. $\mu = (Q', C', P)$ est un unificateur par pièce de Q avec R si et seulement si μ est un pré-unificateur et $glue(\mu) \cap sep(Q') = \emptyset$.*

Preuve : Un pré-unificateur est un unificateur qui ne remplit pas forcément la seconde condition de la définition qui est que les termes apparaissant dans la classe

d'une variable existentielle ne peuvent être que des variables non séparatrices. $glue(\mu)$ désigne les variables qui apparaissent dans la classe d'une variable existentielle, $glue(\mu) \cap sep(Q') = \emptyset$ impose que ces variables ne soit pas des variables séparatrices. Le fait que la partition est valide impose le fait qu'elles ne soient pas des constantes, des variables de la frontière ou d'autres variables existentielles, donc par élimination, elles sont nécessairement des variables non séparatrices. □

Le principe de l'algorithme sera donc de calculer des pré-unificateurs et de les étendre avec les atomes extérieurs à Q' contenant des variables liantes jusqu'à obtenir une pièce et donc des unificateurs par pièce. Le calcul des unificateurs mono-pièce est décomposé en quatre algorithmes.

L'algorithme 2 principal calcule l'ensemble des unificateurs mono-pièce les plus généraux d'une requête avec une règle en s'appuyant sur deux autres algorithmes. La première étape consiste à calculer tous les pré-unificateurs entre un atome de la requête et un atome de la conclusion grâce à l'algorithme 3. Ces pré-unificateurs seront ensuite étendus grâce à l'algorithme 4 pour obtenir les unificateurs mono-pièce.

Algorithme 2 : Calcul des unificateurs mono-pièce les plus généraux

Données : Une requête Q, une règle R.
Résultat : L'ensemble des unificateurs mono-pièce les plus généraux de Q avec R.

```
début
    U ← ∅;
    APU ← Pré-Unificateur_Atomique(Q, R) ; // voir algorithme 3
    while APU ≠ ∅ do
        choisir et supprimer ({a}, {b}, P) dans APU;
        U ← U ∪ extend({a}, {b}, P, APU) ; // voir algorithme 4
    end
    retourner U;
fin
```

L'algorithme "Pre-Unificateur_Atomique" (algorithme 3) est utilisé pour calculer tous les pré-unificateurs les plus généraux entre un atome de la requête et un atome de la règle. Il s'agit donc de trouver la partition la plus fine, appelée *partition par position*, qui unifie deux atomes a et b ayant même prédicat. Pour former un pré-unificateur, la partition par position associée à $\{a, b\}$ doit être valide. La définition qui suit définit formellement cette partition.

Définition 6.3 (Partition par position) *Soit A un ensemble d'atomes ayant un même prédicat p. La* partition par position *associée à A, notée $P_p(A)$, est la partition la plus fine sur $term(A)$ telle que deux termes de A apparaissant à une même position i ($1 \leq i \leq arity(p)$) sont dans une même classe de $P_p(A)$.*

Exemple 57 *Soit $R = p(x) \rightarrow r(x, y) \wedge q(y)$ et $Q = r(u, v) \wedge q(v) \wedge r(u, u)$.*

Si on considère $r(x, y)$ et $r(u, v)$ la partition par position associée est $\{\{u, x\}, \{v, y\}\}$ cette partition est valide puisque la seule variable existentielle y est dans la classe d'une variable de la requête seulement.

Si on considère $r(x, y)$ et $r(u, u)$ la partition par position associée est $\{\{u, x, y\}\}$ cette partition n'est pas valide puisque la variable existentielle y est dans la classe d'une variable de la frontière x.

On note par APU l'ensemble des pré-unificateurs atomiques les plus généraux, c'est-à-dire $APU = \{(\{a\}, \{b\}, P) \mid a \in Q,\ b \in concl(R),\ et\ (\{a\}, \{b\}, P)\ est\ un\ pré\text{-}unificateur\ de\ Q\ avec\ R\ \}$. L'algorithme 3 détaille le calcul de APU.

Algorithme 3 : Pré-Unificateur_Atomique

Données : Une requête Q, une règle R.
Résultat : L'ensemble des pré-unificateurs les plus généraux d'un atome de Q avec un atome de $concl(R)$.

```
début
    APU ← ∅;
    pour chaque a ∈ concl(R) faire
        pour chaque b ∈ Q faire
            si pred(a) = pred(b) et P_p({a, b}) est valide alors
                APU ← APU ∪ {({a}, {b}, P_p({a, b}))};
            fin si
        fin pour
    fin pour
    retourner APU;
fin
```

L'algorithme "extend" (algorithme 4) vérifie d'abord si le pré-unificateur est un unificateur par pièce, c'est-à-dire qu'aucun atome extérieur à Q' ne contient une variable liante. Si c'est le cas, il est retourné, sinon l'algorithme est récursivement appelé sur les extensions de ce pré-unificateur après un appel à l'algorithme 5 pour obtenir la liste des extensions possibles du pré-unificateur.

Algorithme 4 : extend

Données : (Q', C', P) est un pré-unificateur de Q avec R, APU l'ensemble de tous les pré-unificateurs atomiques.

Résultat : L'ensemble des unificateurs mono-pièce les plus généraux de Q avec R calculés à partir de (Q', C', P).

début
  si $glue((Q', C', P)) \cap sep(Q') = \emptyset$ **alors**
    retourner $\{(Q', C', P')\}$;
  sinon
    $res \leftarrow \emptyset$;
    $Q_{ext} \leftarrow \{a \in Q \setminus Q' \mid \exists x \in var(a) \cap glue((Q', C', P))\}$;
    $Ext \leftarrow$ extend-aux$((Q', C', P'), Q_{ext}, APU)$; `// voir algorithme 5`
    pour chaque $(Q_{ext}, H_{ext}, P_{ext}) \in Ext$ **faire**
      si $join(P_{ext}, P)$ *est valide* **alors**
        $res \leftarrow res \cup extend(Q_{ext} \cup Q', H_{ext} \cup C', P_{ext} \cup P, APU)$;
        `// appel récursif à l'algorithme 4`
      fin si
    fin pour
    retourner res ;
  fin si
fin

Ce dernier algorithme (algorithme 5) calcule toutes les possibilités d'étendre un pré-unificateur donné à un ensemble d'atomes donné.

Algorithme 5 : extend-aux

Données : (Q', C', P') un pré-unificateur de Q avec R,Q_{ext} un sous-ensemble de Q (disjoint de Q'), APU l'ensemble des pré-unificateurs atomiques de Q avec R.
Résultat : l'ensemble des pré-unificateurs étendant (Q', C', P') selon Q_{ext} et APU.

```
début
    si Q_ext = ∅ alors
        return {(Q', C', P')}
    sinon
        Ext ← ∅;
        choisir un atome a ∈ Q_ext;
        pour chaque (a, b, P_a) ∈ APU faire
            si join(P', P_a) est valide alors
                Ext ← Ext ∪ extend-aux((Q' ∪ {a}, C' ∪ {b}, join(P, P_a)),
                    Q_ext \ {a}, APU) ; // appel récursif à l'algorithme 5
            fin si
        fin pour
        retourner Ext;
    fin si
fin
```

On reprend l'exemple précédent pour illustrer le fonctionnement du calcul d'un unificateur .

Exemple 58 *Soit* $R = p(x) \to r(x, y) \wedge q(y)$ *et* $Q = r(u, v) \wedge q(v) \wedge r(u, u)$.

On considère $r(x, y)$ *et* $r(u, v)$ *dont la partition par position associée est* $\{\{u, x\}, \{v, y\}\}$ *cette partition est valide donc* $\mu_1 = (\{r(u, v)\}, \{r(x, y)\}, \{\{u, x\}, \{v, y\}\})$ *est un pré-unificateur. Les variables séparatrices de* sep$(\{r(u, v)\})$ *sont* u *et* v*, la seule variable liante de* glue(μ_1) *est* v*. Ces deux ensembles ne sont donc pas disjoints,* μ_1 *n'est pas un unificateur par pièce. Il va falloir l'étendre pour en faire un unificateur par pièce.*

On étend μ_1 *avec les atomes de* $Q \setminus \{r(u, v)\}$ *qui contiennent les variables qui sont à la fois séparatrices et liantes, le seul atome à ajouter est donc* $q(v)$*. Le seul pré-unificateur atomique possible entre* $q(v)$ *et un atome de* concl(R) *est* $(\{q(v)\}, \{q(y)\}, \{\{v, y\}\})$*, on utilise ce pré-unificateur pour étendre* μ_1 *ce qui donne* $\mu_2 = (\{r(u, v), q(v)\}, \{r(x, y), q(y)\}, \{\{u, x\}, \{v, y\}\})$ *la partition est valide c'est donc un pré-unificateur. La seule variable séparatrice de* sep$(\{r(u, v), q(v)\})$ *est* u*, la seule variable liante de* glue(μ_2) *est* v*, ces deux ensembles sont disjoints,* μ_2 *est donc un unificateur par pièce.*

6.2 Mise en place de la compilation

Compilation de règles hiérarchiques Pour la compilation de règles hiérarchiques, on utilise une matrice bi-valeur représentant le pré-ordre $\leq_p$ sur les prédicats calculé à partir des règles hiérarchiques et induisant le pré-ordre sur les atomes. A chaque prédicat de la base, nous associons un nombre, la case (i, j) de la matrice est à 1 si le prédicat i est supérieur au prédicat j et à 0 sinon.

Nous allons maintenant détailler la mise en œuvre de l'amélioration avec les règles compilables.

6.2.1 Saturation

Pour calculer la saturation d'un ensemble de règles, nous effectuons un calcul de point fixe. Nous calculons d'abord l'ensemble de règles inférées par toutes les paires de règles, puis nous supprimons les règles tautologiques et les règles subsumées. L'ensemble obtenu est appelé l'ensemble de règles inférées au rang 1. Ensuite, nous utilisons l'ensemble de règles inférées au rang 1 et l'ensemble de règles initial pour obtenir l'ensemble de règles inférées au rang 2 après avoir supprimé les règles tautologiques et les règles subsumées. Ce processus est répété jusqu'à ce que plus aucune nouvelle règle ne soit obtenue.

Exemple 59
Voici le détails du calcul de la saturation pour les règles compilables de l'exemple 50.
$R_1 = r(x, y) \rightarrow t(x, y)$
$R_2 = s(x, y) \rightarrow t(y, x)$
$R'_2 = t(x, y) \rightarrow s(y, x)$
$R_3 = t(x, y) \rightarrow q(x)$
$R_4 = t(x, y) \rightarrow q(y)$
$R_5 = p(x, y, z) \rightarrow r(x, z)$
$R_6 = p(x, x, z) \rightarrow s(x, x)$

Les règles inférées au rang 1 sont les suivantes :
$R_1 \bullet R'_2 = r(x, y) \rightarrow s(y, x)$
$R_1 \bullet R_3 = r(x, y) \rightarrow q(x)$
$R_1 \bullet R_4 = r(x, y) \rightarrow q(y)$
$R_2 \bullet R'_2 = s(x, y) \rightarrow s(x, y)$ *est tautologique*
$R_2 \bullet R_3 = s(x, y) \rightarrow q(y)$
$R_2 \bullet R_4 = s(x, y) \rightarrow q(x)$
$R'_2 \bullet R_2 = t(x, y) \rightarrow t(x, y)$ *est tautologique*
$R_5 \bullet R_1 = p(x, y, z) \rightarrow t(x, z)$
$R_6 \bullet R_2 = p(x, x, z) \rightarrow t(x, x)$

Les règles inférées au rang 2 sont les suivantes :
$R_5 \bullet R_1 \bullet R'_2 = p(x, y, z) \rightarrow s(z, x)$
$R_5 \bullet R_1 \bullet R_3 = p(x, y, z) \rightarrow q(x)$
$R_5 \bullet R_1 \bullet R_4 = p(x, y, z) \rightarrow q(z)$
$R_6 \bullet R_2 \bullet R_3 = R_6 \bullet R_2 \bullet R_4 = p(x, x, z) \rightarrow q(x)$ *est subsumée par* $R_5 \bullet R_1 \bullet R_3$

La seule règle inférée au rang 3 est la suivante :
$R_5 \bullet R_1 \bullet R'_2 \bullet R_2 = p(x, y, z) \rightarrow t(x, z)$ *est subsumée par* $R_5 \bullet R_1$

Comme aucune règle inférée au rang 3 n'est conservée, la saturation est terminée.

La saturation d'un ensemble de règles peut prendre un temps relativement long. Nous conservons donc dans un fichier texte, le résultat de la saturation de l'ensemble de règles afin d'éviter de la recalculer à chaque test.

6.2.2 Codage

Chaque règle compilable peut être codée d'une part en explicitant les co-occurences de terme dans l'hypothèse par des égalités sur les positions du prédicat de l'atome en hypothèse, et d'autre part, en indiquant à quelle variable de l'hypothèse correspond une variable de la conclusion en donnant pour chaque position du prédicat de l'atome en conclusion la plus petite position du prédicat en hypothèse contenant cette variable.

Définition 6.4 (Codage d'une règle) *Soit* $R : p(x_1, \ldots x_k) \rightarrow q(x'_1, \ldots x'_{k'})$ *une règle compilable. Le codage de* R *est* $cod(R) = (p, q, E_p, L_q)$*, où* E_p *est l'ensemble d'égalités* $\{i = j | i \neq j, x_i = x_j\}$*, où* i *est la plus petite position telle que* $x_i = x_j$ *(ceci assure l'unicité du codage) et* L_q *est une liste de positions* $(pos_1, \ldots, pos_{k'})$ *telle que* pos_i *est la position du premier terme de* $hyp(R)$ *identique à* x'_i*.*

Exemple 60 *Soit* $R = p(x, x, y) \rightarrow q(y, y)$*. Alors* $cod(R) = (p, q, \{1 = 2\}, (3, 3))$

La saturation de l'ensemble de règles est stockée dans une structure codant une relation binaire sur les prédicats. Dans le but de coder le pré-ordre sur les atomes, nous avons pour chaque pairs (p, q) de prédicats, une liste (possiblement vide) de conditions à vérifier afin de savoir si $a \preccurlyeq b$, où a et b sont deux atomes donnés ayant pour prédicat p et q respectivement. Chaque condition code une règle inférée de la forme $a_i \rightarrow b_i$ où a_i a pour prédicat p et b_i a pour prédicat q.

Exemple 61 (Suite de l'exemple 59) *La table 6.1, illustre la table obtenue avec l'ensemble de règles saturé calculé dans l'exemple 59 :*

$R_1 = r(x, y) \to t(x, y)$
$R_2 = s(x, y) \to t(y, x)$
$R'_2 = t(x, y) \to s(y, x)$
$R_3 = t(x, y) \to q(x)$
$R_4 = t(x, y) \to q(y)$
$R_5 = p(x, y, z) \to r(x, z)$
$R_6 = p(x, x, z) \to s(x, x)$
$R_1 \bullet R'_2 = r(x, y) \to s(y, x)$
$R_1 \bullet R_3 = r(x, y) \to q(x)$
$R_1 \bullet R_4 = r(x, y) \to q(y)$
$R_2 \bullet R_3 = s(x, y) \to q(y)$
$R_2 \bullet R_4 = s(x, y) \to q(x)$
$R_5 \bullet R_1 = p(x, y, z) \to t(x, z)$
$R_6 \bullet R_2 = p(x, x, z) \to t(x, x)$
$R_5 \bullet R_1 \bullet R'_2 = p(x, y, z) \to s(z, x)$
$R_5 \bullet R_1 \bullet R_3 = p(x, y, z) \to q(x)$
$R_5 \bullet R_1 \bullet R_4 = p(x, y, z) \to q(z)$

$\preccurlyeq$	q	r	s	t
p	$R_5 \bullet R_1 \bullet R_3 : \{\}, (1)$ $R_5 \bullet R_1 \bullet R_4 : \{\}, (3)$	$R_5 : \{\}, (1, 3)$	$R_6 : \{1 = 2\}, (1, 1)$ $R_5 \bullet R_1 \bullet R'_2 : \{\}, (3, 1)$	$R_5 \bullet R_1 : \{\}, (1, 3)$ $R_6 \bullet R_2 : \{1 = 2\}, (1, 1)$
r	$R_1 \bullet R_3 : \{\}, (1)$ $R_1 \bullet R_4 : \{\}, (2)$		$R_1 \bullet R'_2 : \{\}, (2, 1)$	$R_1 : \{\}, (1, 2)$
s	$R_2 \bullet R_4 : \{\}, (1)$ $R_2 \bullet R_3 : \{\}, (2)$			$R_2 : \{\}, (2, 1)$
t	$R_3 : \{\}, (1)$ $R_4 : \{\}, (2)$		$R'_2 : \{\}, (2, 1)$	

TABLE 6.1 – Compilation des règles de l'exemple 50.

6.2.3 Intégration du pré-ordre sur les atomes

Cette section présente la manière dont est pris en compte le pré-ordre sur les atomes dans les algorithmes d'homomorphisme et d'unification. L'opération de base consistera à vérifier si un atome est $\preccurlyeq$ ou non à un autre. Ceci permettra de calculer les $\preccurlyeq$-unificateurs par pièce et les images possibles d'un atome du fait source lors du calcul d'homomorphisme.

Vérifier que $a \preccurlyeq b$. La comparaison de deux atomes a et b de prédicats p et q respectivement, est faite de la manière suivante. Soit C une liste de codages de règles représentant les conditions associées à la paire (p, q). La relation $a \preccurlyeq b$ est vérifiée s'il existe $(p, q, E_p, L_q) \in C$ telle que :

- pour tout $i = j$ dans E_p, le terme en position i dans a est identique au terme en position j dans a ;
- pour tout $j \in [1 \dots \textit{arité}(q)]$, le terme en position j dans b est identique au terme en position $L_q[j]$ dans a.

L'accès à la liste de conditions associée aux deux atomes se fait en temps constant. La complexité de la vérification dépend donc principalement du nombre de conditions associées à la paire de prédicats, mais en pratique, ce nombre reste très bas, on peut donc considérer que la vérification est en temps constant.

Calculer les $\preccurlyeq$-unificateurs par pièce. Pour le calcul des $\preccurlyeq$-unificateurs par pièce, nous reprenons le même principe que pour le calcul des unificateurs par pièce classique. La notion de pré-unificateur est étendue à celle de pré-$\preccurlyeq$-unificateur.

Définition 6.5 (pré-$\preccurlyeq$-unificateur) *Soit Q une requête, R une règle, $Q' \subseteq Q$, $C' \subseteq concl(R)$, et P une partition sur $term(Q') \cup term(C')$. Alors $\mu = (Q', C', P)$ est un* pré-$\preccurlyeq$-unificateur *de Q avec R si* (1) *P est valide, et* (2) *$P(C') \preccurlyeq P(Q')$.*

La propriété suivante, similaire à celle proposée pour les unificateurs par pièce, assure qu'un pré-$\preccurlyeq$-unificateur dont l'ensemble des variables liantes et l'ensemble des variables séparatrices sont disjoints (voir définition 4.10) est un $\preccurlyeq$-unificateur par pièce et réciproquement. La preuve est similaire à celle de la propriété 13.

Propriété 14 *Soit Q une requête, R une règle, $Q' \subseteq Q$, $C' \subseteq concl(R)$, et P une partition sur $term(Q') \cup term(C')$. $\mu = (Q', C', P)$ est un $\preccurlyeq$-unificateur par pièce de Q avec R si et seulement si μ est un pré-$\preccurlyeq$-unificateur et $glue(\mu) \cap sep(Q') = \emptyset$.*

Pour calculer les $\preccurlyeq$-unificateurs par pièce entre une requête Q et une règle R, il nous faut donc modifier notre algorithme 3 de calcul des pré-unificateurs atomiques pour obtenir l'algorithme 6 qui calcule l'ensemble des pré-$\preccurlyeq$-unificateurs atomiques les plus généraux, c'est-à-dire $APU = \{\mu = (\{a\}, \{b\}, P) \mid a \in Q,\ b \in concl(R),$ *et μ est un pré-$\preccurlyeq$-unificateur de Q avec R* $\}$. Pour cela, nous nous appuyons sur la définition suivante qui pour deux atomes a et b permet de calculer une partition P telle que $P(a) \preccurlyeq P(b)$.

Définition 6.6 (Partition de codage) *Soit a et b deux atomes et $c = (pred(a), pred(b), E_a, L_b)$ un codage d'une règle. La* partition de codage *associée à a et b selon c, notée par $P_c(a, b, c)$, est la partition la plus fine sur $term(a) \cup term(b)$ telle que :*

- *pour chaque $i = j \in E_a$, les termes de a apparaissant en position i et j sont dans une même classe de $P_c(a, b, c)$.*
- *soit $L_b = (p_1, \ldots, p_k)$, pour chaque $p_i \in L_b$, le termes de a apparaissant en position p_i et le terme de b apparaissant en position i sont dans une même classe de $P_c(a, b, c)$.*

Exemple 62 *Soit $a = p(x, y, z)$ et $b = q(u, v)$ deux atomes et $c = (p, q, \{1 = 3\}, (2, 1))$ le codage d'une règle. La partition de codage associée à a et b selon c est la partition $\{\{x, z, v\}, \{y, u\}\}$. Les variables x et z sont dans la même classe car $1 = 3$, u est dans la même classe que y la $2^{ème}$ variable de a et v est dans la même classe que x la $1^{ère}$ variable de a.*

Algorithme 6 : pré-unificateur

Données : Une requête Q, une règle R, C une liste de codages de règles
Résultat : L'ensemble des pré-$\preccurlyeq$-unificateurs les plus généraux d'un atome de Q avec un atome de $concl(R)$.

```
début
    APU ← ∅;
    pour chaque a ∈ concl(R) faire
        pour chaque b ∈ Q faire
            pour chaque c = (pred(a), pred(b), E_a, L_b) ∈ C faire
                si P_c(a, b, c) est valide alors
                    APU ← APU ∪ {({a}, {b}, P_c(a, b, c)};
                fin si
            fin pour
        fin pour
    fin pour
    retourner APU;
fin
```

6.2.4 Déploiement d'une requête

Lors du déploiement d'une requête (voir 5.6), nous avons besoin de connaître les atomes $\preccurlyeq$ à un atome de la requête. Nous allons voir comment il est possible de construire à partir d'un atome a tous les atomes a' les plus généraux tel que $a' \preccurlyeq a$. Soit q le prédicat de a, on construit la liste des atomes $a' \preccurlyeq a$ en prenant, pour tout prédicat p, C_p une liste de codages de règles représentant les conditions associées à la paire (p, q). On choisit tous les $(p, q, E_p, L_q) \in C_p$, avec $L_q = (p_1, \ldots, p_m)$, tels qu'on a pour chaque $p_i = p_j$ le i^{eme} et le j^{eme} termes de a qui sont identiques. A partir de ces (p, q, E_p, L_q), on construit l'atome $a' = p(t_1, \ldots t_k)$ de la manière suivante :

1. soit $L_q = (p_1, \ldots, p_m)$ pour tout $i \in [1, m]$ le terme de a' en position p_i est identique au terme de a à la position i.

2. pour chaque $i = j \in E_p$, $t_i = t_j$ dans a'

3. Pour tous les termes de a' dont la valeur n'a pas été définie par les deux conditions précédentes, on crée une nouvelle variable.

Exemple 63 *Soit $a = q(x, y)$ un atome et $c = (p, q, \{1 = 4\}, (2, 1))$ une condition où p est d'arité 4. l'atome, $a' \preccurlyeq a$ construit à partir de c est $a' = p(y, x, z, y)$ où z est une nouvelle variable. x est identique à la $2^{ème}$ variable de a', y est identique à la $1^{ière}$ et la $4^{ème}$ variable de a' est identique à la $1^{ère}$ donc à y, enfin la $3^{ème}$ position n'est concernée par aucune des conditions donc on crée une nouvelle variable.*

6.3 Structures de données et optimisations pratiques

Faits Les faits sont des listes d'atomes sans doublons que nous avons ordonnés selon l'ordre lexicographique. Ce classement à l'avantage de regrouper les atomes par prédicats et de permettre des tests d'inclusion entre atomes de complexité linéaire.

Stockage des règles Nous avons choisi de stocker nos règles dans un tableau associatif, qui associe à chaque prédicat la liste de règles qui contiennent ce prédicat en conclusion. Ainsi, lors du calcul d'unificateur par pièce entre une requête et l'ensemble de règles, nous ne lançons l'algorithme que sur les règles ayant en conclusion des prédicats susceptibles de s'unifier avec les prédicats présents dans la requête.

Calcul d'homomorphisme Le processus de réécriture produit des requêtes très similaires et dans la majorité des cas, lorsque deux réécritures sont comparables, elles sont soit identiques, soit l'une est un sous-fait donc incluse dans l'autre. Pour cette raison, nous avons choisi de faire un simple test d'inclusion ensembliste avant le test d'homomorphisme. Les atomes d'un fait étant ordonnés, ce test d'inclusion est linéaire alors que le test d'homomorphisme est NP-complet.

Le test d'homomorphisme en lui même est un algorithme de backtrack simple qui cherche à associer chaque atome du fait source à un atome du fait cible en vérifiant que l'association est compatible avec une substitution sur les termes. Pour chaque atome du fait source, l'algorithme choisit une liste d'image possible parmi les atomes du fait cible. Les images possibles sont les atomes dont les prédicats permettent une unification, lorsqu'il n'y a pas de pré-ordre il s'agit des atomes ayant le même prédicat ; lorsque les règles compilées sont hiérarchiques, il s'agit des atomes ayant un prédicat inférieur ou égal à celui du fait source ; sinon, il s'agit des atomes ayant même prédicat qu'un atome $\preccurlyeq$ à l'atome source.

Exemple 64 *Soit* $G = p(x, y) \wedge q(y)$ *et* $F = p(u, v) \wedge q(u) \wedge p(v, u)$. *L'algorithme va choisir pour chaque atome de* G *les images possibles parmi les atomes de* F. *Les images possibles pour* $p(x, y)$ *sont* $p(u, v)$ *et* $p(v, u)$. *La seule image possible pour* $q(y)$ *est* $q(u)$. *L'algorithme va associer* $p(x, y)$ *à* $p(u, v)$ *puis* $q(y)$ *à* $q(u)$ *mais il aboutira à une erreur car cette association n'est pas compatible avec une substitution sur les termes puisque la première association associe* y *à* v *alors que la seconde associe* y *à* u. *Il n'y a pas d'autre image possible pour* $q(y)$, *il revient donc en arrière et choisit l'image suivante de* $p(x, y)$, $p(v, u)$ *puis associe* $q(y)$ *à* $q(u)$ *ce qui est bien compatible avec la substitution* $\{(x, v)(y, u)\}$.

Simplification de requêtes Nous avons trouvé une utilité supplémentaire aux pré-ordres calculés à partir de la compilation des règles. En effet, dans les requêtes proposées, certains atomes peuvent se déduire des autres avec les règles et donc

peuvent être enlevés de la requête sans perte d'informations. Grâce au pré-ordre, nous pouvons vérifier en un temps constant si un atome se déduit d'un autre par les règles compilées. Nous nous sommes servis de cela pour enlever de la requête initiale tous les atomes a tels qu'il existe dans la requête un atome b avec $a \preccurlyeq b$. Ce test est quadratique mais peut diminuer la taille de la requête initiale donc le nombre de réécritures à effectuer et surtout le nombre de requêtes à déployer.

Nous arrivons maintenant au chapitre qui présente une évaluation pratique de nos algorithmes.

Chapitre 7

Évaluation

Le but de ce chapitre est de présenter une évaluation de notre algorithme de réécriture avec l'opérateur *sra* et les deux optimisations utilisant les pré-ordres. Nous commencerons par présenter les ontologies et les requêtes choisies pour faire les tests. Puis nous passerons à l'évaluation proprement dite en comparant les trois versions en termes de temps et de taille de réécritures. Enfin, nous effectuerons une comparaison avec différents systèmes de réécritures déjà existants dans la littérature. Il existe de nombreux systèmes, nous avons choisi de nous comparer à ceux produisant des UCQ afin que le type de sortie soit similaire. En revanche, la plupart des systèmes ne traite que des ontologies issues des familles de logique de description DL-Lite et $\mathcal{EL}$. Puisque les règles traduisant les axiomes $\mathcal{EL}$ ne sont pas *fus*, les tests se feront donc sur des ontologies DL-Lite$_\mathcal{R}$. Cette restriction est liée aux types d'ontologies acceptées par les autres systèmes mais aussi au fait qu'à notre connaissance, il n'existe pas de benchmark d'ontologies *fus* qui ne soient pas DL-Lite$_\mathcal{R}$. Pour finir, nous présenterons une analyse annexe de l'impact de la décomposition en règle à conclusion atomique en termes du nombre de réécritures intermédiaires.

Tous les tests ont été réalisés sur une machine DELL avec un processeur de 3.60 GHz et 16 Go de RAM, et nous avons alloué 4 Go à la machine virtuelle Java.

7.1 Présentation du benchmark

Les ontologies et les requêtes sont celles utilisées dans la littérature pour les systèmes acceptant DL-Lite$_\mathcal{R}$. Ces ontologies ont été développées dans le cadre d'applications réelles en même temps que les requêtes de tests qui sont basées sur les requêtes habituellement utilisées avec les applications correspondantes. Adolena (A) est une ontologie qui donne des informations à propos des dispositifs et des handicaps, et a été développée pour permettre l'interrogation d'une base de connaissances pour "the South African National Accessibility Portal" [Keet et al., 2008]. StockExchange (S) capture des informations sur les institutions financières de l'Union Européenne et a été développée pour l'interrogation d'une base de connaissances [Rodriguez-Muro et al., 2008]. University (U) est une version DL-Lite$_\mathcal{R}$ de l'onto-

logie bien connue LUMB qui décrit la structure organisationnelle des universités, elle a été développée par l'université de Lehigh pour tester des systèmes de raisonnement et de gestion d'ontologies (`http://swat.cse.lehigh.edu/projects/lubm/`). Vicodi (V) est une ontologie sur l'histoire européenne développée dans le projet européen du même nom (`http://www.vicodi.org/`). Ce benchmark de quatre ontologies a d'abord été introduit dans [Pérez-Urbina et al., 2009] et plus tard repris dans [Gottlob et al., 2011, Chortaras et al., 2011, Imprialou et al., 2012, König et al., 2012]. Nous remercions Giorgio Orsi de nous avoir fourni les versions datalog± de ces ontologies. Bien que couramment utilisées dans la littérature pour effectuer des tests, ces quatre ontologies restent de taille modeste. Afin d'effectuer des tests avec des ontologies plus conséquentes, nous avons aussi utilisé les versions DL-Lite$_\mathcal{R}$ des ontologies OpenGALEN2 (`http://www.opengalen.org/`), OBOprotein (`http://www.obofoundry.org/`), et NCI 3.12e (`http://evs.nci.nih.gov/ftp1/NCI_Thesaurus`), que nous noterons G, O et N, respectivement. Les versions DL-Lite$_\mathcal{R}$ de ces ontologies au format OWL peuvent être trouvées sur `http://www.image.ece.ntua.gr/~achort/rapid/testsuite.zip`, et nous les avons ensuite traduites en règles existentielles.

Afin de mieux connaître les ontologies et de permettre une meilleure compréhension des résultats, nous avons fait une analyse des ontologies. Le but est d'avoir une idée des types de règles qui composent ces ontologies. Nous avons donc compté le nombre de règles hiérarchiques, de règles compilables et le nombre de règles restantes.

Ontologie	# Règles hiérarchiques	# Règles compilables non hiérarchiques	# Règles compilables	# Règles totales
A	72	4	76	102
S	16	28	44	52
U	36	36	72	77
V	202	20	222	222
G	26980	836	27816	50764
O	35390	0	35390	43351
N	34319	132	34451	53341

TABLE 7.1 – Types de règles dans les ontologies

Comme écrit précédemment, l'analyse confirme que beaucoup de règles qui composent les ontologies du monde réel sont des règles hiérarchiques ou des règles compilables, ce qui justifie notre volonté de traiter ces règles de manière spécifique pour gagner en efficacité. Nous pouvons déjà noter (voir table 7.1) que dans certaines ontologies, beaucoup de règles compilables ne sont pas des règles hiérarchiques comme dans StockExchange (S) et University (U) alors que pour d'autres, par exemple NCI 3.12e (N), la différence est moindre voire inexistante pour OBOprotein (O). Il est donc prévisible que l'optimisation concernant les règles compilables aura un impact

différent sur ces ontologies.

7.2 Expérimentation des différentes versions de l'algorithme de réécriture

Chacune des ontologies et requêtes ont été transformées en format DLGP (`www2.lirmm.fr/~mugnier/graphik/kiabora/downloads/datalog-plus_en.pdf`) soit depuis un autre format de règles existentielles pour les ontologies A, S, U et V à l'aide de Kiabora (`www2.lirmm.fr/~mugnier/graphik/kiabora/index.html`) soit depuis le format OWL pour les autres. Pour chaque requête de chaque ontologie, l'algorithme 1 avec l'opérateur *sra* (voir section 4.5) a été lancé sur l'ensemble de règles associées de 3 manières différentes :

- dans sa version basique sur l'ensemble complet de règles que nous appelons PURE (Piece-Unifier REwriting)
- en considérant le pré-ordre induit par les règles hiérarchiques et donc en n'appliquant les réécritures que sur l'ensemble des règles non hiérarchiques. On nommera cette version PURE_H
- en considérant le pré-ordre sur les atomes correspondant aux règles compilables et donc en n'appliquant les réécritures que sur l'ensemble des règles non compilables. On nommera cette version PURE_C

Pour chaque exécution, on mesure :

- la taille de l'UCQ résultat c'est-à-dire le nombre de requêtes qui la composent
- le nombre de requêtes générées durant la phase de réécriture
- le nombre de requêtes déployées c'est-à-dire le nombre de requête que l'on peut construire à partir de l'UCQ-pivot et du pré-ordre (voir section 5.6) avant suppression des réécritures trop spécifiques
- le temps de réécriture qui correspond uniquement à la phase de réécriture avec les unificateurs par pièce
- le temps de déploiement et des tests de subsomption nécessaire pour une UCQ minimale
- le temps total qui correspond pour PURE_H et PURE_C à la somme des deux précédents et pour PURE seulement à la phase de réécritures.

Un temps limite de 10 minutes a été accordé pour chaque exécution (incluant la phase de déploiement quant il y a lieu). Dans certains cas, une partie seulement du processus (réécriture et non déploiement) termine dans le temps imparti ce qui

explique que pour certaines requêtes la taille finale de l'UCQ n'est pas connue. Enfin, il faut préciser que les phases de compilation des règles pour calculer le pré-ordre ne sont pas comptées dans les temps d'exécution. En effet, ce calcul est indépendant des requêtes et est donc considéré comme un pré-traitement.

7.2.1 Comparaison en termes de nombre de requêtes

La table 7.2 donne des informations sur la taille de l'UCQ calculée par les différentes versions de PURE, la taille de l'UCQ-pivot calculée par PURE_H et PURE_C, le nombre requêtes générées par les trois versions de l'algorithme et enfin le nombre de requêtes déployées par PURE_H et PURE_C avant les tests de subsomption effectués pour obtenir l'UCQ finale. Le nombre de requêtes générées correspond au nombre de requêtes produites par la phase de réécriture, dont certaines ne sont pas conservées si elles sont plus spécifiques que d'autres. Les valeurs manquantes sont due à des *timeout* (voir table 7.3).

Le plus important à remarquer concernant la table 7.2 est la différence de taille parfois extrême entre la taille de l'UCQ et la taille des UCQ-pivot (par exemple 33887 et 1 pour Q5 de O). La taille des UCQ parfois très importante nous amène à renforcer notre interrogation quant à la pertinence de vouloir produire dans toutes les situations des UCQ. En effet, est-il réaliste de vouloir interroger une base de faits avec des UCQ composées de 30000 requêtes ou plus ? Même si ces requêtes sont calculables en un temps acceptable, leur évaluation ne l'est pas forcément.

Il est aussi flagrant que PURE_C et PURE_H génèrent beaucoup moins de requêtes intermédiaires que PURE, le résultat final est atteint en beaucoup moins d'étapes, nous verrons par la suite que cela se remarque aussi en termes de temps de calcul. Il faut aussi remarquer que PURE_C a toujours des valeurs inférieures ou égales à PURE_H que ce soit en termes de requêtes générées, de taille d'UCQ-pivot ou de requêtes déployées. PURE_C est donc un peu plus efficace sur le nombre, nous vérifierons si cela se confirme aussi en termes de temps.

On peut noter aussi qu'il n'est pas pertinent de comparer PURE_H et PURE_C sur les ontologies O et N. Il n'y a pas plus de règles compilables que de règles hiérarchiques dans O et les chiffres montrent que pour les requêtes proposées sur N, les règles compilables non hiérarchiques n'interviennent pas.

Enfin, on remarque que le nombre de requêtes déployées notamment pour PURE_C est identique à la taille de l'UCQ finale dans la majorité des cas. En revanche, pour les cas où ces deux chiffres ne sont pas identiques, la différence est significative (par exemple Q2 et Q4 pour A). Il serait certainement très intéressant de déterminer le phénomène à l'origine de cette différence pour essayer de le contrer lors du déploiement par exemple et obtenir directement le bon nombre de requêtes après le déploiement sans avoir besoin de refaire des tests de subsomption entre requêtes.

		taille UCQ	taille UCQ-pivot		# CQs Générées			# CQs Déployées	
			PURE_H	PURE_C	PURE	PURE_H	PURE_C	PURE_H	PURE_C
A	Q1	27	3	2	459	15	13	702	702
	Q2	50	3	2	171	3	1	104	104
	Q3	104	4	1	316	8	0	104	104
	Q4	224	3	2	826	7	5	520	520
	Q5	624	4	1	1416	8	0	624	624
S	Q1	6	2	1	9	1	0	6	6
	Q2	2	2	1	137	51	0	2	2
	Q3	4	4	1	275	176	0	4	4
	Q4	4	4	1	450	90	0	4	4
	Q5	8	8	1	688	358	0	8	8
U	Q1	2	1	1	1	0	0	2	2
	Q2	1	1	1	105	28	0	1	1
	Q3	4	2	1	42	19	0	4	4
	Q4	2	1	1	2142	348	0	2	2
	Q5	10	2	1	153	76	0	130	10
V	Q1	15	5	1	14	4	0	15	15
	Q2	10	1	1	9	0	0	10	10
	Q3	72	1	1	117	0	0	72	72
	Q4	185	1	1	328	0	0	185	185
	Q5	30	1	1	59	0	0	30	30
G	Q1	2	1	1	2	0	0	3	2
	Q2	1152	1	1	1275	0	0	1152	1152
	Q3	488	6	5	1514	34	4	492	488
	Q4	147	1	1	154	0	0	147	147
	Q5	324	20	19	908	28	18	327	324
O	Q1	27	20	20	28	21	21	27	27
	Q2	1356	1264	1264	1355	1263	1263	1356	1356
	Q3	33887	1	1	-	0	0	33887	33887
	Q4	-	682	682	-	793	793	34733	34733
	Q5	-	-	-	-	-	-	-	-
N	Q1	3619	2	2	7914	1	1	5005	5005
	Q2	1765	1	1	1803	0	0	1765	1765
	Q3	-	134	134	-	180	180	2905190	2905190
	Q4	-	25	25	-	28	28	222390	222390
	Q5	-	123	123	-	214	214	65016	65016

TABLE 7.2 – Impact des optimisations sur le nombre de requêtes

7.2.2 Comparaison en termes de temps

Les temps de la table 7.3 sont exprimés en millisecondes. Le *timeout* a été fixé à 10 minutes soit 60000 millisecondes et est noté TO dans la table. Il faut noter que le temps est mesuré de 10 ms en 10 ms, un temps de 0 ms correspond donc à un temps effectif entre 0 et 9 ms.

Le temps de réécriture contient la phase de calcul des unificateurs et de calcul des réécritures proprement dites mais aussi l'élimination des requêtes trop spécifiques à chaque pas de réécriture. Le temps de déploiement contient le temps mis à calculer le déploiement de l'UCQ-pivot mais surtout celui pris pour faire le test de subsomption entre les requêtes obtenues pour ne garder que les plus générales. Le temps total est la somme de ces deux temps pour PURE_H et PURE_C, pour PURE il s'agit du temps de réécriture puisqu'il n'y a pas de phase de déploiement.

La table 7.3 nous permet tout d'abord de confirmer que PURE_H et PURE_C sont plus efficaces que PURE en termes de temps. La seule exception à cela est la requête Q2 de O, où PURE est significativement plus rapide que PURE_H et PURE_C. Cela s'explique par le fait que la taille de l'UCQ-pivot n'est que très légèrement inférieure à celle de l'UCQ, donc les règles compilables ne sont que peu utiles pour la réécriture de cette requête, d'où le manque d'efficacité des optimisations.

Concernant la comparaison entre PURE_H et PURE_C, on peut remarquer que PURE_C est globalement plus rapide, notamment sur les ontologies S et U qui sont celles ayant la plus grosse différence entre le nombre de règles hiérarchiques et le nombre de règles compilables. De plus, sur les ontologies telles que O et N où soit les nombres de règles compilables et hiérarchiques sont identiques soit les règles compilables n'interviennent pas dans la réécriture, PURE_C n'est pas moins performant que PURE_H.

Ensuite, il est important de noter que le temps total de calcul est composé en très grande majorité par le déploiement. Le temps de déploiement est lui même utilisé essentiellement par le test de subsomption, cela n'est pas visible dans les chiffres mais se comprend aisément. Le calcul des requêtes déployées est linéaire tandis que le test de subsomption est NP-complet et doit se faire dans les deux sens entre chaque paire de requêtes déployées. Comme nous l'avons remarqué précédemment, dans la majorité des cas le test de subsomption n'est pas utile, mais il reste néanmoins quelques cas pour lesquels il s'avère nécessaire. Réussir à les éviter pour se passer de ce test serait une grande amélioration des performances de nos algorithmes.

Enfin, dans l'optique où nous choisirions d'évaluer l'UCQ-pivot au lieu de l'UCQ, on peut noter qu'à une exception près, elle est calculée en quelques secondes pour les plus longues et quelques millisecondes pour les autres.

Pour mettre en relation la taille et le temps, on peut remarquer que le temps total est particulièrement influencé par la taille de l'UCQ finale. En effet, nous avons noté que le temps était principalement utilisé par le test de subsomption final, qui est d'autant plus long que l'UCQ cible est grande.

		Temps de réécriture		Temps de déploiement + tests de subsomption		Temps total		
		PURE_H	PURE_C	PURE_H	PURE_C	PURE	PURE_H	PURE_C
A	Q1	20	10	130	120	180	150	130
	Q2	10	0	20	40	90	30	40
	Q3	20	10	10	20	170	30	30
	Q4	0	0	120	130	280	120	130
	Q5	10	0	370	440	1500	380	440
S	Q1	10	0	0	0	0	10	0
	Q2	50	0	10	0	100	60	0
	Q3	70	0	0	0	70	70	0
	Q4	30	0	0	0	110	30	0
	Q5	90	10	0	10	120	90	20
U	Q1	0	0	0	0	10	0	0
	Q2	30	0	0	0	100	30	0
	Q3	30	0	0	0	30	30	0
	Q4	220	0	0	0	1500	220	0
	Q5	20	0	10	10	20	30	10
V	Q1	10	0	10	0	10	20	0
	Q2	0	0	10	10	10	10	10
	Q3	0	0	90	70	110	90	70
	Q4	0	0	70	60	120	70	60
	Q5	0	0	0	10	10	0	10
G	Q1	0	0	10	10	0	10	10
	Q2	80	50	570	570	1060	650	620
	Q3	80	70	210	190	1020	290	260
	Q4	0	10	10	0	20	10	10
	Q5	40	30	70	60	890	110	90
O	Q1	150	130	20	10	440	170	140
	Q2	1390	1110	910	760	1160	2300	1870
	Q3	90	90	570240	557900	TO	570330	557990
	Q4	520	430	TO	TO	TO	TO	TO
	Q5	TO	TO	TO	TO	TO	TO	TO
N	Q1	20	10	12330	12520	16240	12350	12530
	Q2	130	120	1200	1190	1650	1330	1310
	Q3	10140	10000	TO	TO	TO	TO	TO
	Q4	1990	2000	TO	TO	TO	TO	TO
	Q5	800	800	TO	TO	TO	TO	TO

TABLE 7.3 – Impact des optimisations sur le temps d'exécution (en ms)

7.3 Comparaison à l'existant

Nous allons maintenant comparer nos algorithmes en termes de temps aux systèmes existants qui produisent le même type de sortie et acceptent les ontologies du benchmark. Nous commençons par une rapide présentation des systèmes répondant à nos critères : Nyaya, tw-rewriting, Presto, Rapid et REQUIEM.

7.3.1 Systèmes comparés

Nyaya

Le système de réécriture Nyaya est l'un des plus comparables au nôtre puisqu'il traite des règles Datalog± qui sont exactement nos règles existentielles et non seulement des ontologies de logique de description légère. Il effectue des réécritures de requêtes conjonctives en UCQ et avec des règles à hypothèse atomique ou des ensemble de règles "sticky". Il est présenté dans [Calì et al., 2010b] pour les ensembles de règles "sticky" et propose une optimisation pour les règles à hypothèse atomique (nommées "linear Datalog rule", dans [Gottlob et al., 2011]). De plus, ce système n'admet que des règles à conclusion atomique ce qui n'est pas une perte en termes d'expressivité mais qui est un facteur d'explosion du nombre de requêtes produites (voir section 7.4). En raison de cette restriction, les requêtes ont été réécrites par Nyaya à partir des ontologies traduites en règles à conclusion atomiques (voir section 7.4).

tw-rewriting

tw-rewriting est un algorithme de réécriture [Rodriguez-Muro et al., 2013] implémenté dans un système d'interrogation d'une base de connaissances nommé Ontop qui stocke les faits dans une base de données relationnelle. Ontop ne se contente pas de faire de la réécriture de requêtes, il prend aussi en charge l'interrogation de la base pour laquelle il propose plusieurs optimisations [Rodriguez-Muro et al., 2013].

Presto

Presto réécrit des requêtes conjonctives en programme Datalog non récursif selon une ontologie DL-Lite$_\mathcal{R}$ [Rosati and Almatelli, 2010]. Il s'appuie fortement sur la hiérarchie de rôles et concepts sous-jacente à l'ontologie et sur la notion de "most general subsumees" MGS qui est le sous-concept ou le sous rôle commun le plus général à un ensemble de concepts ou de rôles donné. La réécriture Datalog produite est composée d'une part d'une UCQ et d'autre part d'un ensemble de règles qui encode les relations de hiérarchies manquantes pour que cette UCQ soit complète. Néanmoins, le système permet aussi de "redéployer" les requêtes du programme Datalog avec les règles hiérarchiques afin de produire une UCQ. Ce principe est assez proche de celui de l'amélioration concernant les hiérarchies présentée dans le chapitre précédent.

Rapid

Le système de réécriture Rapid a d'abord été développé pour traiter des requêtes non booléennes sous ontologies DL-Lite$_\mathcal{R}$ ([Chortaras et al., 2011]). Il utilise deux règles d'inférence pour produire une réécriture en UCQ. La première règle nommée "Shrinking" permet de diminuer le nombre d'occurrences d'une variable en réécrivant des atomes pour les fusionner. La seconde, nommée "Unfolding", est une règle de réécriture simple qui interdit l'unification d'une variable ayant plusieurs occurrences avec une variable existentielle de la règle. Les règles d'inférences ont ensuite été étendues pour réécrire des requêtes sous ontologies $\mathcal{ELHI}$ en programme Datalog [Trivela et al., 2013].

REQUIEM

REQUIEM ([Pérez-Urbina et al., 2009]) réécrit des requêtes conjonctives en présence d'ontologies DL-Lite$_\mathcal{R}$ et $\mathcal{ELHIO}\neg$. Les réécritures peuvent être de deux formes, UCQ lorsque l'ontologie est en DL-Lite$_\mathcal{R}$ et Datalog lorsque l'ontologie est une ontologie $\mathcal{ELHIO}\neg$. Comme pour Rapid, la méthode se base sur une règle d'inférence.

7.3.2 Comparaison expérimentale

La table 7.4 présente les temps de calcul des différents systèmes en millisecondes. Les systèmes produisent tous une UCQ correcte, complète et minimal avec ce benchmark. L'abréviation OM signifie que l'exécution s'est arrêtée sur une exception de type "out of memory", TO signifie que l'exécution a dépassé le temps imparti de 10 minutes soit 60000 millisecondes. Il faut noter que pour nos différentes versions de PURE le temps est mesuré de 10 ms en 10 ms, un temps de 0 ms correspond donc

à un temps effectif entre 0 et 9 ms. Ceci induit une différence avec les autres systèmes qui mesure leur temps toutes les 1 ms néanmoins une divergence de quelques millisecondes n'est pas représentative.

Le système le plus performant globalement est Rapid qui réussit à réécrire toutes les requêtes sauf une et dont toutes les réécritures prennent moins d'une seconde à l'exception des dernières requêtes des ontologies O et N.

Le système tw-rewriting est très légèrement plus rapide que Rapid sur toutes les requêtes sauf celles où il dépasse le *timeout*.

REQUIEM ne supporte pas bien le passage à l'échelle, ses temps d'exécution sont relativement élevés sur les trois grosses ontologies G, O et N.

Nyaya obtient des temps relativement élevés sur l'ontologie A découpée en règles à conclusion atomique. Nous évoquons dans la section suivante l'impact du découpage en règle non atomique sur la réécriture. De plus, nous n'avons pu faire tourner Nyaya sur les trois ontologies les plus grosses, le pré-traitement de l'ontologie avant la phase de réécritures mettant un temps trop long (supérieur à 48h).

Enfin, notre système PURE$_C$ est capable de réécrire des requêtes en présence de grosses ontologies en un temps acceptable et similaire aux autres systèmes. Ceci malgré le fait qu'il soit capable de traiter des ontologies *fus* plus riches que DL-Lite$_\mathcal{R}$. En revanche, il souffre d'un manque de performances lorsque la taille de l'UCQ cible est importante. Néanmoins, dans ces cas là, la question de la pertinence de vouloir une réécriture sous forme d'UCQ se pose puisque même si l'UCQ peut être calculée en un temps acceptable malgré sa taille, son évaluation sur une base de faits risque d'être problématique. Il est sûrement intéressant dans ces cas là, de trouver d'autres formes de réécritures dont l'évaluation sera plus rapide. Dans ce cadre là, notre système fournit, pour tous les tests effectués sauf un, une UCQ-pivot en quelques secondes voire millisecondes.

Néanmoins, il faut garder à l'esprit que ces comparaisons ne peuvent donner que des indices sur l'efficacité des algorithmes sous-jacent, d'une part parce que le jeu de test reste trop réduit et mal caractérisé. D'autre part parce que les différents systèmes ont des "niveaux" d'implémentation différents, certains ne sont que des prototypes alors que d'autres mettent en œuvre des optimisations pratiques plus élaborées. Comme nous l'avons remarqué au court de notre propre développement, ces optimisations ont un fort impact sur les performances d'un système mais elles sont rarement mentionnées dans les publications scientifiques, il est donc difficile de juger de l'efficacité d'un algorithme par rapport à un autre si les différences ne sont pas flagrantes.

		PURE	PURE$_H$	PURE$_C$	Nyaya	tw-rewriting	Presto [1]	Rapid	REQUIEM
A	Q1	180	150	130	1122	12	-	18	264
	Q2	90	30	40	862	11	-	23	105
	Q3	170	30	30	2363	9	-	34	134
	Q4	280	120	130	5557	12	-	48	252
	Q5	1500	380	440	33206	10	-	93	467
S	Q1	0	10	0	4	5	-	7	6
	Q2	100	60	0	4	7	-	9	164
	Q3	70	70	0	46	8	-	13	443
	Q4	110	30	0	7	8	-	12	674
	Q5	120	90	20	8	8	-	15	5981
U	Q1	10	0	0	8	8	-	6	5
	Q2	100	30	0	4	6	-	9	128
	Q3	30	30	0	12	8	-	7	211
	Q4	1500	220	0	6	8	-	13	1091
	Q5	20	30	10	10	8	-	15	2943
V	Q1	10	20	0	13	5	254	9	10
	Q2	10	10	10	51	9	255	5	10
	Q3	110	90	70	21	8	315	25	62
	Q4	120	70	60	28	8	392	32	136
	Q5	10	0	10	22	10	316	26	73
G	Q1	0	10	10	-	5	TO	5	47
	Q2	1060	650	620	-	11	TO	74	209048
	Q3	1020	290	260	-	21	TO	59	259103
	Q4	20	10	10	-	6	TO	10	190250
	Q5	890	110	90	-	26	TO	40	238459
O	Q1	440	170	140	-	19	TO	23	3447
	Q2	1160	2300	1870	-	578	TO	953	21786
	Q3	TO	570330	557990	-	77	TO	611	TO
	Q4	TO	TO	TO	-	1238	TO	14698	TO
	Q5	TO	TO	TO	-	TO	TO	562223	TO
N	Q1	16240	12350	12530	-	18	TO	317	7356
	Q2	1650	1330	1330	-	15	TO	66	14618
	Q3	TO	TO	TO	-	TO	TO	OM	TO
	Q4	TO	TO	TO	-	TO	TO	3867	TO
	Q5	TO	TO	TO	-	TO	TO	148253	TO

TABLE 7.4 – Temps d'exécution des différents systèmes de réécriture (en ms)

7.4 Impact de la décomposition en règles à conclusion atomique

Certains systèmes de réécriture tels que Nyaya ne traite que des règles à conclusion atomique. D'un point de vue théorique, ce n'est pas une perte d'expressivité. En effet, comme nous allons le détailler plus bas n'importe quelle règle peut être décomposée en règles à conclusion atomique en ajoutant des prédicats supplémentaires. Pour conserver un ensemble de réécritures adéquat, il faut alors en fin de processus supprimer les requêtes contenant ces prédicats supplémentaires. En revanche, l'expérience présentée dans cette section montre très clairement que d'un point de vue pratique, cela conduit à une perte significative d'efficacité.

Un ensemble de règles quelconques peut être très simplement décomposé en un ensemble de règles à conclusion atomique de la manière suivante. Pour chaque règle R à conclusion non atomique, on introduit un prédicat auxiliaire aux n'apparaissant nulle part ailleurs dont l'arité est le nombre de variables de la conclusion. Puis on remplace la règle par l'ensemble de règles composé d'une règle ayant comme hypothèse $hyp(R)$ et comme conclusion le prédicat aux dont les arguments sont les variables de $concl(R)$. Enfin, pour chacun des atomes a de la conclusion de R on crée une règle ayant le prédicat aux avec comme arguments les variables de $concl(R)$ en hypothèse et l'atome a en conclusion. Voici un exemple pour illustrer cette transformation.

Exemple 65 (Décomposition en règles à conclusion atomique)
Soit $R_1 = s(x, y) \rightarrow r(x, z) \wedge q(z)$ *et* $R_2 = q(x) \wedge q(x, y) \rightarrow s(y, z) \wedge p(z) \wedge r(y, x)$. $\{R_1, R_2\}$ *peut être transformé en un ensemble de règles à conclusion atomique équivalent* $\{R_{11}, R_{12}, R_{13}, R_{21}, R_{22}, R_{23}, R_{24}\}$ *où*

$$
\begin{aligned}
R_{11} &= s(x, y) \rightarrow aux_1(x, z) \\
R_{12} &= aux_1(x, z) \rightarrow r(x, z) \\
R_{13} &= aux_1(x, z) \rightarrow q(z) \\
R_{21} &= q(x) \wedge q(x, y) \rightarrow aux_2(x, y, z) \\
R_{22} &= aux_2(x, y, z) \rightarrow s(y, z) \\
R_{23} &= aux_2(x, y, z) \rightarrow p(z) \\
R_{24} &= aux_2(x, y, z) \rightarrow r(y, x)
\end{aligned}
$$

Les ontologies A et U contiennent des règles à conclusion non atomique, on appelle AX respectivement UX les ontologies obtenues en les décomposant en règles à conclusion atomique.

La table 7.5 retrace le comportement de l'algorithme de réécriture avec *sra* sur AX par rapport à A et sur UX par rapport à U selon les trois paramètres de temps

	Time (ms)		Output		Generated	
	A	AX	A	AX	A	AX
Q1	170	330	27	41	459	720
Q2	90	4'900	50	1431	171	4567
Q3	240	47'290	104	4466	316	13838
Q4	440	28'620	224	3159	826	14526
Q5	2'100	1**h**36	624	32921	2416	215523
	U	UX	U	UX	U	UX
Q1	0	10	2	5	1	4
Q2	0	0	1	1	105	120
Q3	10	20	4	12	42	155
Q4	1'370	4'190	2	5	2142	4720
Q5	20	20	10	25	153	351

TABLE 7.5 – Impact de la décomposition en règle à conclusion atomique

d'exécution, de taille de sortie (nombre de CQs) et de nombre de requêtes générées durant la réécriture. La taille de la sortie pour AX et UX est mesurée avant élimination des CQs contenant des prédicats auxiliaires. On peut voir qu'éviter la décomposition en règle à conclusion non atomique mène à une différence significative à tout point de vue. Le gain est particulièrement important avec Q_5 sur A/AX, pour les trois paramètres. De plus, il faut noter que seulement 29 règles sur 102 pour A et 5 sur 77 pour U ont plusieurs atomes en conclusion (dans notre cas exactement deux). On peut raisonnablement penser que le gain augmente avec la proportion de règles à conclusion non atomique et la taille de ces conclusions.

Chapitre 8

Conclusion

Tout au long de cette thèse, nous nous sommes intéressé au problème d'interrogation de bases de connaissances (OBQA) qui est un problème fondamental à la croisée du domaine de la représentation des connaissances et du domaine des bases de données. Le problème consiste à prendre en compte des connaissances générales, typiquement une ontologie de domaine, lors de l'évaluation d'une requête. Nous avons choisi de représenter nos informations ontologiques au moyen des règles existentielles au lieu du formalisme utilisé classiquement des logiques de description. Les règles existentielles généralisent les DLs légères utilisées pour OBQA, apportant un point de vue plus général et uniforme sur le problème.

Nous considérons une approche couramment utilisée, qui consiste à réécrire la requête en exploitant les règles de façon à se ramener à un problème classique d'interrogation d'une base de données. Après avoir présenté les notions de base nécessaires à la compréhension de ce document, nous définissons, dans le chapitre 3, un cadre théorique d'étude des algorithmes de réécriture d'une requête conjonctive en une union de requêtes conjonctives, appelée ensemble de réécritures. Dans ce cadre théorique, nous avons notamment défini les propriétés souhaitables d'un ensemble de réécritures : adéquation, complétude et minimalité, et la notion d'opérateur de réécriture qui à une requête et un ensemble de règles associe l'ensemble de ses réécritures directes. Nous avons aussi proposé un algorithme de réécriture générique, prenant en paramètre un opérateur de réécriture, qui explore l'espace des réécritures en largeur. Cet algorithme maintient un ensemble de réécritures les plus générales, en calculant à chaque pas, à l'aide de l'opérateur de réécriture, les réécritures directes des requêtes non réécrites de l'ensemble et met à jour l'ensemble avec les réécritures nouvellement produites les plus générales. Cet algorithme s'accompagne de propriétés sur l'opérateur de réécriture qui garantissent que l'ensemble de réécritures calculé par l'algorithme a bien les propriétés souhaitées d'adéquation, de complétude et de minimalité.

Par la suite, dans le chapitre 4, nous proposons une famille d'opérateurs de réécriture basés sur la notion d'unificateur par pièce. La présentation de cette famille aboutit à un opérateur de réécriture, appelé *sra*, qui est plus efficace que l'opérateur

qui calculerait brutalement tous les unificateurs par pièce, et qui a les propriétés garantissant que l'ensemble de réécritures calculé par notre algorithme est adéquat, complet et minimal.

Pour améliorer nos opérateurs, nous avons développé, dans le chapitre 5, deux optimisations qui permettent de traiter une partie des règles de manière spécifique et plus efficace. La première compile les règles hiérarchiques en un pré-ordre sur les prédicats qui induit un pré-ordre sur les atomes, la seconde est une généralisation de la première et compile les règles dites compilables directement en un pré-ordre sur les atomes. L'utilisation de ces optimisations avec nos opérateurs de réécriture, produit une UCQ-pivot associée aux règles hiérarchiques ou compilables. Une UCQ-pivot peut être traduite simplement dans différents formalismes dans le but d'être évaluée sur les faits, notamment sous la forme d'un programme Datalog.

Nous avons effectué une première implémentation de nos algorithmes dont certains détails sont présentés dans le chapitre 6. Certaines optimisations pratiques simples ont été mises en place mais il reste encore beaucoup de points à améliorer. L'opération clef de notre système est le test d'homomorphisme, qui effectué de manière répétée devient très coûteux en temps. Il apparaît donc primordial que ces tests soient faits de la manière la plus efficace possible. On sait notamment que les requêtes générées sont relativement similaires les unes aux autres. En effet, si on considère une requête obtenue à partir d'une autre requête par réécriture avec un unificateur par pièce, les deux requêtes ont en commun la partie sur laquelle ne s'applique pas l'unificateur. Une optimisation envisagée serait de mettre à profit cette similarité pour bénéficier des résultats obtenus lors des exécutions précédentes : si grâce à un test d'homomorphisme antécédent, on sait que la partie commune à deux requêtes ne s'envoie pas sur une troisième requête, il n'y a pas besoin d'effectuer de test supplémentaire pour savoir qu'aucune des deux requêtes ne s'envoie sur la troisième. D'autres optimisations pratiques sont envisageables, par exemple pour le stockage des requêtes : avec une structure de données adéquate, on pourrait obtenir en temps quasi constant les requêtes possédant les prédicats nécessaires pour qu'une requête s'envoie dessus. En effet, si dans une requête, on retrouve les prédicats p et q, on sait qu'elle ne peut pas être envoyée par un homomorphisme classique sur une requête qui ne contient pas les prédicats p et q. La prise en compte d'un pré-ordre lors de l'homomorphisme complexifie cette vérification mais elle reste néanmoins faisable car il reste possible de déterminer les prédicats sur lesquels peut être envoyé un autre prédicat. Nous évoquons aussi, dans ce chapitre, comme première ouverture, un opérateur de réécriture *aram* qui permettrait d'effectuer une réécriture selon plusieurs règles à la fois. L'objectif de cet opérateur est de diminuer le nombre de réécritures équivalentes produites par l'enchaînement dans un ordre différent de plusieurs unificateurs portant sur des parties disjointes de la requête. Les premières expérimentations confirment l'intérêt pratique de cet opérateur, cependant ses propriétés liées à l'élagabilité restent à étudier.

Nous avons clôturé notre travail dans le chapitre 7 en évaluant notre opérateur de réécriture *sra* et nos deux optimisations, d'abord les uns avec les autres puis vis-

à-vis d'autres systèmes de réécritures. Ces évaluations montrent que notre système est capable de réécrire des requêtes en présence de grosses ontologies en un temps acceptable et comparable aux autres systèmes. Ceci malgré le fait qu'il soit capable de traiter des ontologies *fus* plus riches que celles des autres systèmes considérés. Nous notons toutefois qu'il souffre d'une manque de performance lorsque l'UCQ cible est importante. D'autre part, la forme même de réécriture en UCQ est connue pour ses problèmes de taille, qui la rendent parfois difficilement évaluable sur des données. Ceci renforce notre conviction qu'il faut étendre nos recherches pour produire d'autres formes de réécritures plus compactes telles que les programmes Datalog. Ces évaluations ont été faites sur des benchmarks du domaine couramment utilisés, néanmoins il nous faut remarquer que ces benchmarks sont insuffisants pour évaluer des systèmes acceptant des règles existentielles. En effet, toutes les ontologies ont été écrites dans des logiques de descriptions dont l'expressivité est bien plus restreinte que celle des règles existentielles. Pour observer dans sa globalité le comportement de notre système, il faudrait disposer d'ontologies *fus* dont l'expressivité n'est pas restreinte à DL-Lite$_{\mathcal{R}}$. En plus des ontologies, il faudrait élargir le nombre de requêtes et envisager la création d'un générateur de requêtes qui à partir d'une ontologie produirait un ensemble de requêtes ayant certaines propriétés, par exemple en termes de taille des réécritures ou en termes de nombre et de types des règles impliquées dans le processus de réécriture.

La principale perspective envisagée pour ce travail est de proposer des opérateurs qui produisent des réécritures sous forme de programme Datalog. Une partie de ce travail a été fait avec les UCQ-pivots qui se transforment simplement en programme Datalog. Cette traduction reste néanmoins à implémenter, il faudra ensuite proposer une manière appropriée de comparer l'évaluation de ces programmes Datalog et celle des UCQ correspondantes. Comme nous l'avons fait remarquer, cela nécessiterait de comparer des systèmes différents (SGBD relationnel et système Datalog). Le problème étant alors que rentre alors en jeu l'efficacité de l'implémentation des systèmes. En plus de permettre de produire des réécritures plus compactes, la réécriture sous forme de programme Datalog permet de traiter des ensembles de règles dont l'expressivité est plus importante que les ensembles de règle *fus*. En effet, la traduction de $\mathcal{EL}$, une des DLs légères les plus utilisées, en règles existentielles n'est pas *fus*. Les ontologies $\mathcal{EL}$ ne peuvent pas être utilisées pour réécrire des requêtes en UCQ, en revanche elles peuvent l'être pour les réécrire en programme Datalog. Cela est aussi le cas pour les ontologies $\mathcal{ELHIO}\neg$ [Pérez-Urbina et al., 2010] et d'autres fragments plus complexes, le plus expressif étant Horn-$\mathcal{SHIQ}$ à notre connaissance [Eiter et al., 2012]. Le premier travail consistera à déterminer quelles sont les propriétés de ces ensembles de règles qui leur permettent d'être réductibles en un programme Datalog, puis d'en déduire une procédure mettant en œuvre les unificateurs par pièce pour produire ce programme Datalog. Les ensembles de règles pour lesquels la complexité du problème d'implication d'une requête conjonctive booléenne est dans P en complexité de données sont susceptibles d'admettre une réécriture sous forme d'un programme Datalog. En effet, la complexité de déterminer

si un programme Datalog admet une réponse sur une base de données est dans P en complexité de données, et le processus de réécriture est lui-même dans P en complexité de données puisqu'il ne concerne pas les données. On ne peut donc espérer réécrire des ensembles de règles appartenant à des classes pour lesquelles le problème d'implication est plus complexe. Il a par exemple été récemment montré que les classes de règles gardées et frontière-gardées [Calì et al., 2008, Baget et al., 2010] admettent des réécritures en Datalog [Gottlob et al., 2014b]. Cependant, ce résultat n'est pas encore accompagné d'un algorithme qui serait utilisable en pratique.

Finalement, une approche intéressante est l'approche combinée, introduite dans [Kontchakov et al., 2010] : on utilise les règles pour modifier à la fois la base de faits et la requête de façon à avoir : $(F, \mathcal{R}) \models Q$ si et seulement si $F'| = Q'$, où F' est obtenu de F et $\mathcal{R}$ et Q' de Q et $\mathcal{R}$. L'optimisation que nous avons développée au chapitre 5, qui consiste à compiler des règles en un pré-ordre, peut être vue comme un exemple de cette approche, qui permet d'obtenir des réécritures plus petites (sans dépasser le fragment *fus* toutefois) : ici, F' est obtenu en saturant F avec les règles compilables, et Q' est la requête pivot. Les réécritures en Datalog peuvent également être vues sous l'angle de l'approche combinée : on peut utiliser une partie des règles pour saturer la base de faits, et on conserve l'autre en tant que requête, soit une UCQ, soit une requête Datalog. Plus généralement, l'approche combinée permet d'obtenir des réécritures plus compactes, mais elle permet également de dépasser le fragment *fus*, c'est pourquoi elle paraît particulièrement prometteuse.

Nous avons de plus laissé de côté le traitement des contraintes négatives. Comme nous l'avons vu, la traduction de DL-Lite par exemple en règles existentielles produit des contraintes négatives (voir section 2.3.3). Ces contraintes n'ont pas du tout été prises en compte lors des réécritures. Leur présence a pour conséquence d'empêcher de produire les réécritures pour lesquelles il existe un homomorphisme de l'hypothèse d'une contrainte dans la requête. Il faudrait intégrer la gestion de ces contraintes notamment dans l'implémentation. Un autre type de règles n'a pas été traité, il s'agit des règles avec égalité, leur intégration est plus complexe et nécessite des travaux théoriques préalables. En effet, les règles avec égalité conduisent très rapidement à l'indécidabilité du problème d'interrogation. Quelques travaux préliminaires ont été réalisés, dans [Calì et al., 2012] on peut notamment retrouver les restrictions les plus notables qui assurent la décidabilité en présence de règles à égalité. Nous savons aussi qu'un ensemble de règles *fus* auquel on ajoute des règles avec égalité n'est pas garanti de conserver la propriété *fus* [Baget et al., 2011a]. Néanmoins, il est possible que certains ensembles de règles associés aux règles à égalité sous certaines contraintes restent *fus*, il faudra alors étendre la notion d'unificateur par pièce et vérifier que les opérateurs de réécriture gardent leurs bonnes propriétés avec cette extension.

Annexes

Définition 8.1 (Largeur arborescente d'un fait) *Soit F un fait (possiblement infini). Une décomposition arborescente de F est un arbre (possiblement infini) $T = (\mathcal{X} = \{X_1, \ldots, X_k, \ldots\}, U)$ où :*

1. *les X_i sont des ensembles de termes de F ;*
2. *pour chaque atome a de F, il existe $X_i \in \mathcal{X}$ tel que $\mathrm{term}(a) \subseteq X_i$;*
3. *pour chaque terme t de F, le sous-graphe de T induit par les nœuds X_i tels que $t \in X_i$ est connexe.*

La largeur d'une décomposition arborescente est la taille de son nœud le plus grand moins 1. La largeur arborescente d'un fait est la largeur minimale de toutes ses décompositions arborescentes.

Index

Bibliographie

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Baader, 2003] Baader, F. (2003). Terminological cycles in a description logic with existential restrictions. In *IJCAI'03*, pages 325–330.

[Baader et al., 2005] Baader, F., Brandt, S., and Lutz, C. (2005). Pushing the el envelope. In *IJCAI'05*, pages 364–369.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook : Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA.

[Baget et al., 2010] Baget, J.-F., Leclère, M., and Mugnier, M.-L. (2010). Walking the decidability line for rules with existential variables. In *KR'10*.

[Baget et al., 2009] Baget, J.-F., Leclère, M., Mugnier, M.-L., and Salvat, E. (2009). Extending decidable cases for rules with existential variables. In *IJCAI'09*, pages 677–682.

[Baget et al., 2011a] Baget, J.-F., Leclère, M., Mugnier, M.-L., and Salvat, E. (2011a). On rules with existential variables : Walking the decidability line. *Artif. Intell.*, 175(9-10) :1620–1654.

[Baget et al., 2011b] Baget, J.-F., Mugnier, M.-L., Rudolph, S., and Thomazo, M. (2011b). Walking the complexity lines for generalized guarded existential rules. In *IJCAI'11*, pages 712–717.

[Beeri and Vardi, 1981] Beeri, C. and Vardi, M. (1981). The implication problem for data dependencies. In *ICALP'81*, volume 115 of *LNCS*, pages 73–85.

[Beeri and Vardi, 1984] Beeri, C. and Vardi, M. Y. (1984). Formal systems for tuple and equality generating dependencies. *SIAM J. Comput.*, 13(1) :76–98.

[Calì et al., 2008] Calì, A., Gottlob, G., and Kifer, M. (2008). Taming the infinite chase : Query answering under expressive relational constraints. In *KR'08*, pages 70–80.

[Calì et al., 2009] Calì, A., Gottlob, G., and Lukasiewicz, T. (2009). A general datalog-based framework for tractable query answering over ontologies. In *PODS'09*, pages 77–86.

[Calì et al., 2012] Calì, A., Gottlob, G., Orsi, G., and Pieris, A. (2012). On the interaction of existential rules and equality constraints in ontology querying. In *Correct Reasoning*, pages 117–133.

[Calì et al., 2010a] Calì, A., Gottlob, G., and Pieris, A. (2010a). Query answering under non-guarded rules in datalog+/-. In *RR'10*.

[Calì et al., 2010b] Calì, A., Gottlob, G., and Pieris, A. (2010b). Query rewriting under non-guarded rules. In *AMW'10*.

[Calvanese et al., 2005] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2005). DL-Lite : Tractable description logics for ontologies. In *AAAI'05*, pages 602–607.

[Chandra et al., 1981] Chandra, A. K., Lewis, H. R., and Makowsky, J. A. (1981). Embedded implicational dependencies and their inference problem. In *STOC'81*, pages 342–354.

[Chein and Mugnier, 2009] Chein, M. and Mugnier, M.-L. (2009). *Graph-based Knowledge Representation and Reasoning—Computational Foundations of Conceptual Graphs*. Advanced Information and Knowledge Processing. Springer.

[Chortaras et al., 2011] Chortaras, A., Trivela, D., and Stamou, G. B. (2011). Optimized query rewriting for owl 2 ql. In *CADE'11*, pages 192–206.

[Courcelle, 1990] Courcelle, B. (1990). The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1) :12–75.

[da Silva, 2014] da Silva, B. P. L. (2014). Accès aux données en présence d'ontologies.

[Deutsch et al., 2008] Deutsch, A., Nash, A., and Remmel, J. B. (2008). The chase revisited. In *PODS'08*, pages 149–158.

[Eiter et al., 2012] Eiter, T., Ortiz, M., Simkus, M., Tran, T.-K., and Xiao, G. (2012). Query rewriting for horn-shiq plus rules. In *AAAI'12*.

[Fagin et al., 2005] Fagin, R., Kolaitis, P. G., Miller, R. J., and Popa, L. (2005). Data exchange : semantics and query answering. *Theor. Comput. Sci.*, 336(1) :89–124.

[Gottlob et al., 2011] Gottlob, G., Orsi, G., and Pieris, A. (2011). Ontological queries : Rewriting and optimization. In *ICDE'11*, pages 2–13.

[Gottlob et al., 2014a] Gottlob, G., Orsi, G., and Pieris, A. (2014a). Query rewriting and optimization for ontological databases. *CoRR'14*, abs/1405.2848.

[Gottlob et al., 2014b] Gottlob, G., Rudolph, S., and Simkus, M. (2014b). Expressiveness of guarded existential rule languages. In *PODS'14*, pages 27–38.

[Gottlob and Schwentick, 2012] Gottlob, G. and Schwentick, T. (2012). Rewriting ontological queries into small nonrecursive datalog programs. In *KR'12*.

[Hell and Nesetril, 1992] Hell, P. and Nesetril, J. (1992). The core of a graph. *Discrete Mathematics*, 109(1-3) :117–126.

[Imprialou et al., 2012] Imprialou, M., Stoilos, G., and Grau, B. C. (2012). Benchmarking ontology-based query rewriting systems. In *AAAI'12*.

[Keet et al., 2008] Keet, C. M., Alberts, R., Gerber, A., and Chimamiwa, G. (2008). Enhancing web portals with ontology-based data access : The case study of south africa's accessibility portal for people with disabilities. In *OWLED'08*.

[König et al., 2012] König, M., Leclère, M., Mugnier, M.-L., and Thomazo, M. (2012). A sound and complete backward chaining algorithm for existential rules. In *RR'02*, pages 122–138.

[König et al., 2013] König, M., Leclère, M., Mugnier, M.-L., and Thomazo, M. (2013). On the exploration of the query rewriting space with existential rules. In *RR'03*, pages 123–137.

[Kontchakov et al., 2010] Kontchakov, R., Lutz, C., Toman, D., Wolter, F., and Zakharyaschev, M. (2010). The combined approach to query answering in dl-lite. In *KR'10*.

[Kontchakov et al., 2011] Kontchakov, R., Lutz, C., Toman, D., Wolter, F., and Zakharyaschev, M. (2011). The combined approach to ontology-based data access. In *IJCAI'11*, pages 2656–2661.

[Krötzsch and Rudolph, 2011] Krötzsch, M. and Rudolph, S. (2011). Extending decidable existential rules by joining acyclicity and guardedness. In *IJCAI'11*, pages 963–968.

[Lutz et al., 2009] Lutz, C., Toman, D., and Wolter, F. (2009). Conjunctive query answering in the description logic el using a relational database system. In *IJCAI'09*, pages 2070–2075.

[Marnette, 2009] Marnette, B. (2009). Generalized schema-mappings : from termination to tractability. In *PODS'09*, pages 13–22.

[Pérez-Urbina et al., 2009] Pérez-Urbina, H., Horrocks, I., and Motik, B. (2009). Efficient query answering for owl 2. In *ISWC'09*, pages 489–504.

[Pérez-Urbina et al., 2009] Pérez-Urbina, H., Motik, B., and Horrocks, I. (2009). A comparison of query rewriting techniques for dl-lite. In *DL'09*.

[Pérez-Urbina et al., 2010] Pérez-Urbina, H., Motik, B., and Horrocks, I. (2010). Tractable query answering and rewriting under description logic constraints. *J. Applied Logic*, 8(2) :186–209.

[Rodriguez-Muro and Calvanese, 2012] Rodriguez-Muro, M. and Calvanese, D. (2012). High performance query answering over dl-lite ontologies. In *KR'12*.

[Rodriguez-Muro et al., 2013] Rodriguez-Muro, M., Kontchakov, R., and Zakharyaschev, M. (2013). Query rewriting and optimisation with database dependencies in ontop. In *DL'13*, pages 917–929.

[Rodriguez-Muro et al., 2008] Rodriguez-Muro, M., Lubyte, L., and Calvanese, D. (2008). Realizing ontology based data access : A plug-in for protg. In *ICDE'08*, pages 286–289.

[Rosati and Almatelli, 2010] Rosati, R. and Almatelli, A. (2010). Improving query answering over dl-lite ontologies. In *KR'10*.

[Salvat and Mugnier, 1996] Salvat, E. and Mugnier, M.-L. (1996). Sound and complete forward and backward chainingd of graph rules. In *ICCS'96*, pages 248–262.

[Thomazo, 2013a] Thomazo, M. (2013a). Compact rewritings for existential rules. In *IJCAI'13*.

[Thomazo, 2013b] Thomazo, M. (2013b). Conjonctive query answering under existential rules.

[Thomazo et al., 2012] Thomazo, M., Baget, J.-F., Mugnier, M.-L., and Rudolph, S. (2012). A generic querying algorithm for greedy sets of existential rules. In *KR'12*.

[Trivela et al., 2013] Trivela, D., Stoilos, G., Chortaras, A., and Stamou, G. B. (2013). Optimising resolution-based rewriting algorithms for dl ontologies. In *DL'13*, pages 464–476.