Swan ROCHER

Outils d'Analyse d'une Base de Règles

2012 Université Montpellier 2

Remerciements

Nous tenons à remercier Marie-Laure Mugnier ainsi que Michel Leclere et Michaël Thomazo pour leur encadrement tout au long de ce projet. De plus, merci par avance aux examinateurs de la soutenance pour leur attention. Enfin, merci également à Mountaz Hascoët pour l'organisation de l'ensemble des TER.

Table des matières

Chapitre 1

Contexte

Base de données sans ontologie =¿ ne permet pas de répondre à la requête-exemple. On ajoute l'ontologie =¿ on peut déduire de nouveaux faits, ...

Requête / base de connaissance (vision globale).

Universelles / Existentielles.

Non décidabilité de la réponse à une requête dans une base contenant des existentielles.

Exemples avec des MOTS.

Malgré le fait que de manière général, il n'existe aucun algorithme permettant de répondre à ce problème, certaines règles peuvent entrer dans des catégories (qui seront nommées *classes de règles*) qui en ajoutant des contraintes sur la forme des règles s'assurent que le problème soit décidable. Selon quelles contraintes sont satisfaites, il est nécessaire d'appliquer différentes méthodes de réponse sur différents sous-ensembles des règles.

L'objectif de ce TER est donc d'implémenter un outil permettant d'analyser une base de règles afin de construire son graphe de dépendances associés, de déterminer quelles contraintes sont satisfaites, sur quel sous-ensemble, et si la base est décidable d'en déduire quels algorithmes utiliser sur chacun d'eux. De plus, cet outil doit pouvoir charger des bases de règles à partir de fichiers, ainsi que les y écrire, et être suffisamment modulable pour permettre l'ajout de nouvelles vérifications de contrainte.

Chapitre 2

Notions de base

Avant toute chose il est nécessaire de définir un certain nombre de notions et de termes.

2.1 Prédicat

Un prédicat noté $p\backslash n$ est un symbole relationnel d'arité n. Dans la suite, on supposera que tout nom de prédicat est unique et on notera p_i la i^{eme} position de p.

2.2 Atome

Un atome $a = p(a_1, a_2, ..., a_n)$ associe un terme à chaque position d'un prédicat $p\backslash n$. On note :

- a_i le terme en position i dans a. Un terme peut être une *constante* ou une *variable*. Une variable peut être *libre* ou *quantifiée* universellement (notée $\forall - var$) ou existentiellement (notée $\exists - var$).
- $dom(a) = \{a_i : \forall\ i \in [1, n]\}$, l'ensemble des termes de a
- $var(a)$ l'ensemble des variables de a
- $cst(a)$ l'ensemble des constantes de a

2.3 Conjonction d'atomes

Une conjonction de n atomes A est définie telle que :
$A = \bigwedge_{i=1}^{n} k_i$ avec $\forall\ i \in [1, n]$ $a_i = p_i(a_{i1}, a_{i2}, ..., a_{in_i})$ un atome de prédicat $p_i\backslash n_i$.

2.4 Représentation graphique d'une conjonction d'atomes

Une conjonction d'atomes peut être représentée par le graphe non orienté $G_A = (V_A, E_A, \omega)$ avec V_A son ensemble de sommets, E_A, son ensemble d'arêtes et ω une fonction de poids sur les arêtes construits de la manière suivante :
- $V_A = P_A \cup T_A$ avec $P_A = \{i : a_i \in A\}$ et $T_A = \{t_j \in dom(A)\}$
- $E_A = \{(i, t_j) : \forall a_i \in A, \forall t_j \in dom(a_i)\}$
- $\omega : E_A \to \mathbb{N}$ telle que $\omega(i, t_j) = j : \forall (i, t_j) \in E_A$

Cette représentation a de nombreux avantages, elle permet notamment de parcourir rapidement les atomes liés à un terme (et réciproquement), ainsi que de pouvoir être visualisée agréablement (voir figure 2.1).

FIGURE 2.1 – Représentation de $\forall x, \forall y(salle(x) \wedge date(y) \wedge reservee(x, y))$

On remarque que G_A admet une bipartition de ses sommets, en effet toutes les arêtes ont une extrémité dans P_A et l'autre dans T_A, or par construction $P_A \cap T_A = \emptyset$.

2.5 Règle

Une règle $R = (H, C)$ est constituée de deux conjonctions d'atomes H et C représentant respectivement l'hypothèse (le corps) et la conclusion (la tête) de R. Toutes les variables apparaissant dans H sont quantifiées universellement tandis que celles apparaissant uniquement dans C le sont existentiellement. Ainsi une règle est toujours sous la forme $R : \forall x_i(H \to \exists z_j(C))$.

On note :
- $dom(R) = dom(H) \cup dom(C)$, le domaine de R
- $var(R) = var(H) \cup var(C)$, les variables de R
- $cst(R) = cst(H) \cup cst(C)$, les constantes de R
- $fr(R) = var(H) \cap var(C)$, l'ensemble des variables frontières de R
- $cutp(R) = fr(R) \cup cst(R)$, l'ensemble des points de coupure de R

2.6 Représentation graphique d'une règle

Tout comme une simple conjonction d'atomes, une règle $R = (H, C)$ peut être représentée par un graphe similaire, en ajoutant une coloration à deux couleurs : une pour les atomes de l'hypothèse, l'autre pour ceux de la conclusion.

Ainsi le graphe associé $G_R = (V_R, E_R, \omega, \chi)$ est défini comme :
- $V_R = P_R \cup T_R$ avec $P_R = \{i : r_i \in R\}$ et $T_R = \{t_j \in dom(R)\}$

– $E_R = \{(i, t_j) : \forall r_i \in R, \forall t_j \in dom(R_i)\}$
– $\omega : E_R \to [1, p]$ telle que $\omega(i, t_j) = j : \forall (i, t_j) \in E_R$
– $\chi : P_R \to \{1, 2\}$ telle que $\chi(r_i) = 1$ si $r_i \in H$ et $\chi(r_i) = 2$ si $r_i \in C$.

Par exemple la règle $R : \forall x \forall y\ (salle(x) \wedge date(y) \wedge reservee(x, y) \to \exists z\ (cours(z) \wedge aLieu(z, x, y)))$ peut être visualisée de la façon suivante :

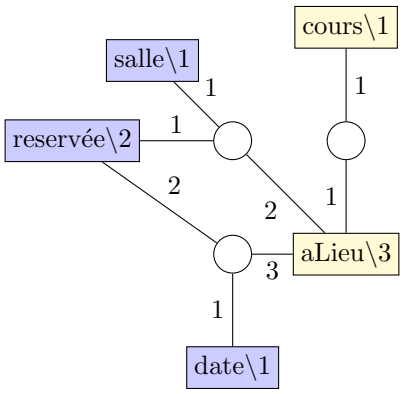

FIGURE 2.2 – Exemple de représentation d'une règle

Sur la figure suivante (2.3) représentant la même règle, on peut voir les différentes parties de celle-ci, de plus les variables ont été étiquetées pour mieux visualiser les différents termes.

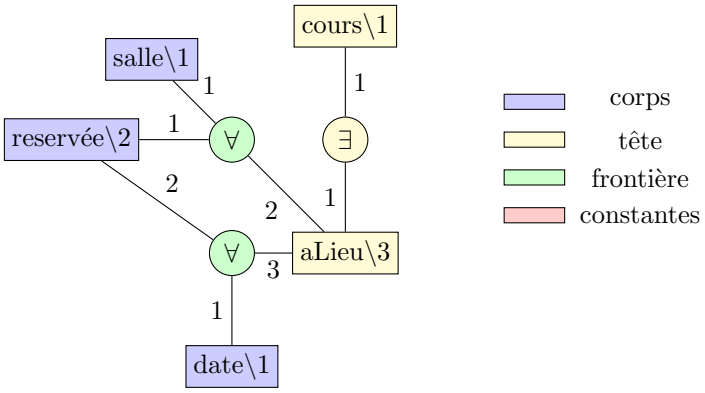

FIGURE 2.3 – Les différents éléments d'une règle

2.7 Règle à conclusion atomique

Une règle à conclusion atomique ajoute une contrainte sur la forme de sa conclusion qui ne doit contenir qu'un seul atome. Ces règles ont l'avantage d'être plus simples à unifier (voir section 3.2), et la plupart des algorithme présentés dans ce rapport sont plus efficaces sur ce type de règle, tandis que d'autres ne fonctionnent uniquement sur celles-ci.

Mais ceci n'est pas un problème puisqu'il est possible de réécrire une règle à conclusion non atomique en un ensemble de règles à conclusions atomiques équivalent.

En effet, quelle que soit une règle $R = (H, C)$, nous pouvons définir un nouveau prédicat p_R d'arité $|var(R)|$ ainsi qu'un nouvel ensemble de règles à conclusions atomiques R^A dont le premier élément aura la même hypothèse que R et une conclusion de prédicat p_R contenant toutes ses variables, et dont les suivants auront pour hypothèse cette nouvelle conclusion, et comme conclusion atomique les atomes de C.

Cet ensemble est donc défini de la manière suivante :
$R^A = \{R_i^A = (H_i^A, C_i^A) : \forall\ i \in [0, |C|]\}$, tels que :

$$R_0^A = \begin{cases} H_0^A = H, \\ C_0^A = p_R(\{x_j \in var(R)\}) \end{cases} \quad \forall\ i \in [1, |C|]\ R_i^A = \begin{cases} H_i^A = C_0^A, \\ C_i^A = c_i \in C \end{cases}$$

Ainsi nous pouvons utiliser l'algorithme 1 afin d'effectuer cette conversion.

Algorithm 1 Conversion d'une règle à conclusion non atomique

Require: $R = (H, C)$: une règle quelconque
Ensure: R^A : un ensemble de règles à conlusions atomiques équivalent à R

```
H_0^A ← H
C_0^A ← p_R({x_i ∈ var(R)})
R^A ← {(H_0^A, C_0^A)}
for all atome c_i ∈ C do
    R_i^A ← (C_0^A, c_i)
    R^A ← R^A ∪ {R_i^A}
end for
return R^A
```

Toute règle pouvant donc se réécrire de manière équivalente en un ensemble de règles à conclusion atomique, dans la suite nous ne considèrerons que des règles sous cette forme.

2.8 Base de connaissance

Une base de connaissance $K = (F, R)$ est constituée d'un ensemble de *faits* représenté par une conjonction d'atomes F, ainsi que d'une *ontologie* représentée par un ensemble de règles R.

Les faits sont souvent considérés comme complètement instanciés, c'est à dire ne contenant que des constantes, mais ici, les règles contenant des variables existentielles peuvent générer de nouveaux individus. Donc nous définissons F comme une conjonction d'atomes existentiellement fermée.

2.9 Chaînage avant

Afin de déterminer si une requête bouléenne Q peut être déduite d'une base de connaissance $B = (R, F)$, le chaînage avant *dérive* de manière itérative la base de faits F (qui peut être vue comme un fait unique) par l'ensemble de règles R de manière à générer (en cherchant de nouveaux homomorphismes) de nouveaux faits qui devront à leur tour être étudiés. Avant chaque dérivation, l'algorithme vérifie si F^i contient Q auquel cas la réponse est positive, et si $F^i = F^{i-1}$ afin de savoir si la chaîne de dérivation est finie et dans ce cas donner une réponse négative.

2.10 Chaînage arrière

Le chaînage arrière quant à lui consiste à réécrire la requête Q en plusieurs nouvelles requêtes juqu'à ce qu'une de celles-ci soit dans F. Pour cela les conclusions des règles de la base sont *unifiées* avec des sous-ensembles de Q (voir section 3.2 pour plus de détails).

Chapitre 3

Graphe de dépendances des règles

Le graphe de dépendances de règles est une représentation d'une base de règles très intéressante. En effet il permet de vérifier rapidement quelles règles pourront éventuellement être déclenchées après l'application d'une règle donnée.

De plus il permet de déterminer l'appartenance à certaines classes de règles, et le calcul de ses composantes fortement connexes permet de "découper" la base de manière à effectuer les requêtes de manières différentes selon celles-ci.

3.1 Définition

Le graphe de dépendances des règles associé à une base de règles B_R est défini comme le graphe orienté $GRD = (V_{GRD}, E_{GRD})$ avec :
- $V_{GRD} = \{R_i \in B_R\}$,
- $E_{GRD} = \{(R_i, R_j) : \exists$ un bon unificateur $\mu : \mu(C_i) = \mu(H_j)\}$.

Intuitivement, on crée un sommet par règle et on relie R_i à R_j si R_i *"peut amener à déclencher"* R_j (R_j dépend de R_i). La notion d'unificateur est abordée dans la section suivante.

3.2 Unification de règles

Afin de pouvoir construire ce graphe, il faut donc pouvoir déterminer si une règle peut en déclencher une autre, c'est à dire s'il existe un unificateur entre la conclusion de la première et l'hypothèse de la seconde. Tout d'abord, l'unification est définie, s'en suit un algorithme permettant de vérifier si un tel unificateur existe, puis la correction de celui-ci ainsi que ses complexités.

3.2.1 Définitions

Substitution

Une substitution de taille n d'un ensemble de symboles X dans un ensemble de symboles Y est une fonction de X vers Y représentée par l'ensemble de couples suivants (avec $n \leq |X|$) :
- $s = \{(x_i, y_i) \ : \ \forall i \in [1, n] \ x_i \in X, \ y_i \in Y, \forall j \neq i \ x_i \neq x_j\}$
- $s(x_i) = y_i \ \forall i \in [1, n]$
- $s(x_i) = x_i \ \forall i \in [n+1, |X|] \ x_i \in X$

Unificateur logique

Un unificateur logique entre deux atomes a_1 et a_2 est une substitution μ telle que :
- $\mu : var(a_1) \cup var(a_2) \rightarrow dom(a_1) \cup dom(a_2)$
- $\mu(a_1) = \mu(a_2)$

Cette définition s'étend aux conjonctions d'atomes.

Unificateur de conclusion atomique

Un unificateur de conclusion atomique est un unificateur logique $\mu = \{(x_i, t_i) : \forall i \in [1, n]\}$ entre l'hypothèse d'une règle $R_1 = (H_1, C_1)$ et la conclusion atomique d'une règle $R_2 = (H_2, C_2)$ et est défini de la manière suivante :
- $\mu : fr(R_2) \cup var(H_1) \rightarrow dom(C_2) \cup cst(H_1)$
- $\forall (x_i, t_i) \in \mu \ si \ x_i \in fr(R_2) \ alors \ t_i \in cutp(R_2) \cup cst(H_1)$

Bonne unification atomique

Un bon unificateur de conclusion atomique est un unificateur de conclusion atomique $\mu = \{x_i, t_i) : \forall i \in [1, n]\}$ entre un sous ensemble Q de l'hypothèse d'une règle $R_1 = (H_1, C_1)$ et la conclusion d'une règle atomique $R_2 = (H_2, C_2)$ tel que :
$\forall (x_i, t_i) \in \mu : si \ x_i \in H_1 \setminus Q \ alors \ t_i \ n'est \ pas \ une \ \exists - var$

Un tel ensemble Q est appelé un *bon ensemble d'unification atomique* de l'hypothèse de R_1 par la conclusion de R_2. On note que Q est donc défini comme suit :
- $Q \subseteq H_1$
- $\forall$ *position* i *de* $\exists - var$ *dans* $C_2, \forall$ *atome* $a \ \in Q, si \ a_i \in var(H_1) \ alors \ \forall$ *atome* $b \in H_1 : si \ \exists \ b_j \in b : a_i = b_j, alors \ b \in Q$

Bon ensemble d'unification atomique minimal

Un *bon ensemble d'unification atomique minimal* Q de H_1 par C_2 enraciné en a est défini tel que :
- Q est un bon ensemble d'unification atomique de H_1 par C_2
- $a \in Q$
- $|Q| = min(|Q_i| : Q_i \ est \ un \ bon \ ensemble \ d'unification \ atomique \ de \ H_1 \ par \ C_2 \ et \ a \in Q_i)$

3.2.2 Algorithmes

Vérifier qu'une règle atomique R_i peut déclencher R_j consiste donc à trouver un bon unificateur atomique entre R_i et R_j. Dans cette section, un algorithme permettant de répondre à ce problème est détaillé.

Dans la suite, les règles sont supposées représentées par des graphes (tels que définis en 2.6) et à conclusion atomique.

Le premier algorithme fait appel aux deux suivants de manière à déterminer si il existe au moins un unificateur entre les deux conjonctions d'atomes. En première phase, il vérifie l'exitence d'unificateurs avec chaque atome de manière indépendante. S'ensuit une extension à partir des atomes préselectionnés, et dès qu'un bon ensemble d'unification est entièrement unifié, l'algorithme s'arrête en répondant avec succès.

Algorithm 2 Unification

Require: H_1 : conjonction d'atomes, $R = (H_2, C_2)$: règle à conclusion atomique
Ensure: succès si C_2 peut s'unifier avec H_1, i.e. si $\exists H \subseteq H_1, \mu$ *une substitution* : $\mu(H_1) = \mu(C_2)$, échec sinon

```
1  ▷ Précoloration
2  for all sommet atome a ∈ H1 do
3      if UnificationLocale(a, R) ≠ échec then
4          couleurLogique[a] ← noir
5      else
6          couleurLogique[a] ← blanc
7      end if
8  end for
9  ▷ Initialisation du tableau contenant les positions des variables existentielles de C2
10 E ← {i : ci est une ∃ − var de C2}
11 ▷ Extension des ensembles
12 for all sommet atome a ∈ H1 : couleurLogique[a] = noir do
13     if Q ← Extension(H1, a, couleurLogique, E) ≠ échec then
14         if UnificationLocale(Q, R) ≠ échec then
15             return succès
16         end if
17     end if
18     couleurLogique[a] ← blanc
19 end for
20 return échec
```

Le deuxième algorithme est utilisé pour le calcul des bons ensembles d'unification à partir d'un atome racine. Tant qu'aucune erreur n'est détectée il *avale* les atomes voisins aux termes en positions existentielles. Les positions existentielles sont les indices des variables existentielles dans l'atome de conclusion.

Algorithm 3 Extension

Require: H_1 : conjonction d'atomes, $a \in H_1$: sommet atome racine, *couleurLogique* : tableau de taille égal au nombre d'atomes dans H_1 tel que $couleurLogique[a] = noir$ ssi $UnificationLocale(a, R) =$ succès, E : ensemble des positions des variables existentielles

Ensure: Q : bon ensemble d'unification minimal des atomes de H_1 construit à partir de a s'il existe, échec sinon.

```
1  ▷ Initialisation du parcours
2  for all sommet atome a ∈ H1 do
3      if couleurLogique[a] = noir then
4          for all sommet terme t ∈ voisins(a) do
5              couleur[t] = blanc
6          end for
7          couleur[a] = blanc
8      end if
9  end for
10 couleur[a] ← noir
11 Q ← {a} ▷ conjonction d'atomes à traiter
12 attente ← {a} ▷ file d'attente du parcours
13 while attente ≠ ∅ do
14     u ← haut(attente)
15     if u est un atome then
16         for all i ∈ E do
17             v ← voisin(u, i)
18             if v est une constante then
19                 return échec
20             else if couleur[v] = blanc then
21                 ▷ v est une ∀ − var non marquée par le parcours
22                 couleur[v] ← noir
23                 attente ← attente ∪ {v}
24             end if
25         end for
26     else
27         ▷ u est un terme
28         for all v ∈ voisins(u) do
29             if couleurLogique[v] = blanc then
30                 return échec
31             else
32                 if couleur[v] = blanc then
33                     couleur[v] ← noir
34                     attente ← attente ∪ {v}
35                     Q ← Q ∪ {v}
36                 end if
37             end if
38         end for
39     end if
40 end while ▷ Fin du parcours
41 return Q
```

Remarque :
La phase d'initialisation du parcours pourrait simplement parcourir tous les sommets de H_1 et mettre leur couleur à blanc. En pratique, cette solution serait sans doute plus efficace, mais dépendrait donc du nombre de sommets total dans H_1. Ce qui en théorie amènerait la complexité de cette boucle en $\bigcirc(nombre\ d'atomes\ \times arite\ max\ de\ H_1)$. Or ici, la complexité ne dépend pas de cette arité max, mais uniquement de larité du prédicat de la conclusion C_2.

Le dernier algorithme est celui qui teste réellement si il existe un unificateur entre une conclusion atomique, et un bon ensemble d'unification atomique minimal. Il est appelé une première fois pour tester les atomes de la conjonction séparement, et permettre une préselection des atomes (qui vont servir de racine). Durant la dernière phase (lorsqu'il est appelé sur les ensembles étendus), s'il trouve un unificateur, celui-ci assure que la règle peut amener à déclencher la conjonction.

Algorithm 4 UnificationLocale

Require: H_1 : conjonction d'atomes, $R = (H_2, C_2)$: règle à conclusion atomique
Ensure: succès si C_2 peut s'unifier avec H_1

```
1  ▷ Vérification des prédicats
2  for all atome a ∈ H1 do
3      if prédicat(a) ≠ prédicat(C2) then
4          return échec
5      end if
6  end for
7  u ← ∅   ▷ substitution
8  for all terme t_i ∈ C2 do
9      ▷ def : a_i = terme de a en position i
10     E ← {a_i : ∀ atome a ∈ H1}
11     if t_i est une constante then
12         if ∃ v ∈ E : v est une constante et v ≠ t_i, ou v est une ∃ − variable then
13             return échec
14         else
15             u ← {(v, t_i) : v ∈ E et v ≠ t_i}
16         end if
17     else if t_i est une ∃ − variable then
18         if ∃ v ∈ E : v est une ∃ − variable et v ≠ t_i, ou v est une constante then
19             return échec
20         else
21             u ← {(v, t_i) : v ∈ E et v ≠ t_i}
22         end if
23     else
24         if ∃ v1, v2 ∈ E : v1 ≠ v2 et v1, v2 ne sont pas des ∀ − variables then
25             return échec
26         else if ∃ c ∈ E : c est une constante then
27             u ← {(v, c) : v ∈ E ∪ {t_i} et v ≠ c}
28         else
29             u ← {(v, t_i) : v ∈ E et v ≠ t_i}
30         end if
31     end if
32     H1 ← u(H1)
33     C2 ← u(C2)
34 end for
35 return succès
```

3.2.3 Correction

3.2.4 Complexites

Soit R une règle, et G_R son graphe associé. On note :
– k : nombre d'atomes dans R
– p : arité maximum des prédicats de R

– $t \leq n \times p$: nombre de termes de R

et :

– $n = k + t$: nombre de sommets dans G_R
– $m \leq n \times p$: nombre d'arêtes dans G_R

Avec notre représentation nous avons donc les complexités suivantes :

– Parcourir les sommets prédicats : $\bigcirc(k)$.
– Parcourir tous les sommets : $\bigcirc(n)$.
– Accéder au i^{eme} voisin d'un sommet : $\bigcirc(1)$.

– $n =$ nombre d'atomes dans H_1
– $p =$ arité de C_2
– $t =$ nombre de termes "colorables" dans H_1
– $m =$ nombre d'arêtes "suivables" dans H_1

On remarque que dans le pire des cas on a :

– $t = n \times p$
– $m = n \times p$

Temps

– UnificationLocale (pire des cas) = $\bigcirc(np)$
– Extension (pire des cas) = $\bigcirc(np)$
– Unification (pire des cas) = $\bigcirc(n^2p)$

Extension

On effectue simplement un parcours en largeur à partir d'un sommet donné qui peut éventuellement s'arrêter plus tôt qu'un parcours classique. La complexité en temps dans le pire des cas est donc au plus la même, c'est à dire linéaire au nombre de sommets plus le nombre d'arcs. Le graphe représentant la conjonction d'atomes H_1 possèdent $n \times ariteMax(H_1)$ arêtes et $n \times (ariteMax(H_1) + 1)$ sommets. On sait donc que $C^{temps}_{Extension} = \bigcirc(n \times ariteMax(H_1))$.

Deux cas :

i) Découverte d'un sommet atome :

parcours classique =¿ on récupère tous les voisins blancs

parcours extension =¿ on récupére tous les voisins blancs de couleur logique noire (en effet arrêt immédiat si un voisin de couleur logique noire est découvert.

ii) Découverte d'un sommet variable :

pas de différence

On a donc que l'algo ne suit que les sommets atomes pouvant être unifies localement. C'est à dire (entre autres) que leur prédicat est égal à prédicat(C_2).

...

Espace

Chapitre 4

Classes de règles

4.1 Classes abstraites

Trois classes abstraites ont été définies, chacune permettant l'usage de certains algorithmes sur l'ensemble de règles considéré. Elles sont dites *abstraites* puisque déterminer si un ensemble de règles appartient à l'une de ces classes est un problème non décidable. De plus elles sont incomparables entre elles, et non exclusives.

4.1.1 Finite Expansion Set

La première classe est définie comme assurant la finition des algorithmes de chaînage avant. Ainsi tout ensemble de règles appartenant à cette classe peut être utilisé pour les dérivations de ces algorithmes.

4.1.2 Bounded Treewidth Set

La deuxième définit quant à elle les ensembles de règles où la production de nouveaux faits suit la forme d'un arbre de largeur bornée. Cette classe ne permet pas l'utilisation direct d'algorithmes, mais par contre la classe abstraite Greedy Bounded Treewidth qui est une spécialisation de celle-ci, s'assure que le chaînage avant s'exécute en temps fini, et ce via un algorithme glouton permettant de construire la décomposition de l'arbre et en utilisant un algorithme d'arrêt spécifique.

4.1.3 Finite Unification Set

Enfin la dernière classe abstraite assure la finition des algorithmes de chaînage arrière utilisant les méthodes en *largeur* qui ne garderait que les faits *le plus généraux* lors de leur génération.

4.2 Classes concrètes

Comme expliqué dans la section précédente, il est impossible de déterminer à coup sûr si un ensemble de règles appartient à l'une de ces classes, par contre de nombreuses classes dites *concrètes* ont été définies. Chacune d'entre elle, en ajoutant des contraintes sur la forme des règles voir sur l'ensemble des règles, permet de s'assurer que celui-ci appartient à certaines des classes abstraites.

4.2.1 Acyclicité du graphe de dépendance des règles

Le seul fait que le graphe de dépendance soit sans circuit suffit à certifier que le chaînage avant et arrière s'exécutent en temps fini, impliquant que si cette contrainte est satisfaite, l'ensemble des règles appartient à *FES* et à *FUS* .

4.2.2 Faiblement acyclique

Cette classe est quant à elle un peu particulière puisqu'elle demande la génération d'une autre structure de graphes et qu'elle s'applique directement sur un ensemble de règles et pas uniquement sur chacune des règles indépendamment des autres.

On crée donc un graphe de dépendances des positions dont les sommets sont les positions des prédicats et dont la construction des arcs est la suivante : pour chaque variable x d'une règle R apparaissant dans l'hypothèse en position p_i, si x appartient à la frontière de R alors il existe un arc de p_i vers chacune des positions r_j de la conclusion de R dans laquelle apparaît x, de plus pour chacune des variables existentielles apparaîssant en position q_k il existe un arc *spécial* de p_i vers q_k.

Si le graphe de dépendances des positions associés à un ensemble de règle ne contient aucun circuit passant par un arc *spécial* alors il est dit *faiblement acyclique* et appartient à *FES* .

Voir [1] pour les détails et les preuves.

4.2.3 Sticky

Un ensemble de règles satisfait cette contrainte si certaines de ses variables (marquées) n'apparaissent pas plusieurs fois dans l'hypothèse d'une de ses règles.

Plus précisemment, on marque tout d'abord les variables en deux étapes :

– premièrement, pour chaque règle R, et pour chacune des variables x de l'hypothèse H de R, si x n'apparait pas dans la conclusion de R alors on marque chaque occurence de x dans H.

– Deuxièmement, pour chaque règle R si une variable marquée apparaît en position p_i alors pour chaque règle R' (incluant $R = R'$), et pour chacune des variables x apparaîssant en position p_i dans la conclusion de R', on marque chaque occurence de x dans l'hypothèse de R'.

Ensuite si aucune règle ne contient plusieurs occurence d'une variable marquée dans son hypothèse, l'ensemble est dit *sticky*, et assure la finition des algorithmes basés sur le chaînage arrière, cette classe appartient donc à la classe abstraite *FUS* (voir [1]).

4.2.4 Faiblement sticky

Cette classe est une généralisation des deux précédentes qui malheureusement n'appartient à aucune des classes abstraites citées dans la section précédente (cf [1]).

Tout comme la vérification de l'appartenance à la classe *faiblement acyclique (4.2.2)*, il est nécessaire de construire le graphe de dépendances des positions. De plus, on dit qu'une position (un sommet de ce graphe) est *sûre* si elle n'apparait dans aucun circuit contenant un arc *spécial*, et on marque chacune des variables en suivant le même algorithme que pour la classe *sticky (4.2.3)*.

On peut maintenant définir un ensemble de règles *faiblement sticky* comme ne contenant que des variables sûres ou non marquées.

4.2.5 Guardée

Une règle gardée est définie comme étant une règle dont un atome de son hypothèse (nommé *garde*) contient toutes les variables de celle-ci. Si toutes les règles de l'ensemble contiennent un garde, alors l'ensemble appartient à *GBTS* .

4.2.6 Frontière gardée

On dit qu'une règle a une *frontière gardée* si un atome de son hypothèse possède toutes les variables de la frontière. On peut remarquer que cette classe est une généralisation de la précédente. Et dans le cas où toutes les règles de l'ensemble possède cette propriété, celui-ci appartient également à *GBTS* .

4.2.7 Frontière-1

Cette classe contient les règles dont la frontière est de taille 1, elle est donc une spécialisation des règles à frontière gardée, Ainsi un ensemble de règles satisfaisant cette propriété appartient à *GBTS* .

4.2.8 Hypothèse atomique

Les règles ne contenant que des hypothèses atomiques s'assurent que la règle est *gardée (4.2.5)*, et de plus assurent que les algorithmes basés sur le chaînage arrière se terminent en temps fini. Donc si toutes les règles d'un ensemble sont à hypothèse atomique alors celui-ci est $GBTS$ et FUS .

4.2.9 Domaine restreint

Une règle satisfait cette contrainte si tous les atomes de sa conclusion contiennent soit toutes les variables de l'hypothèse, soit aucune. Dans le cas des règles à conclusion atomique, cela revient à s'assurer que la frontière de chaque règle est soit égale à 0 soit au nombre de variables universelles. Cette contrainte est suffisante pour que l'ensemble de règles appartienne à FUS .

4.2.10 Règle déconnectée

Une règle est dite déconnectée si sa frontière est vide, cette classe est donc une spécialisation des règles à *domaine restreint (4.2.9)*, à *frontière gardée (4.2.6)* et *faiblement acyclique (4.2.2)*. Ce type de règle n'est pas très utilisé étant donné que seules des constantes sont partagées entre l'hypothèse et la conclusion ce qui limite leur usage, mais elles ont l'avantage d'être à la fois FES , $GBTS$ et FUS .

4.2.11 Règles universelles

Les règles ne contenant aucune variable existentielle ($vars(C) \subseteq vars(H)$) forment bien entendu un ensemble décidable. Celles-ci sont à la fois FES et $GBTS$. On peut également remarquer que toute ensemble de règles universelles est également *faiblement acyclique (4.2.2)*.

4.3 Schéma d'inclusion des classes de règles

Sur la figure 4.1, les flèches représentent la spécialisation tandis que les couleurs l'appartenance aux classes abstraites. De plus, seules les classes cités ci-dessus apparaissent sur celle-ci.

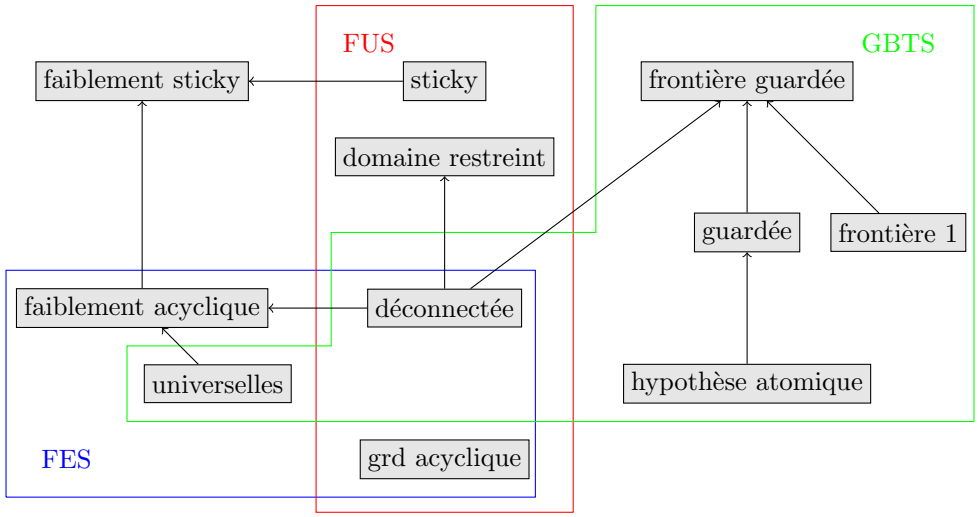

FIGURE 4.1 – Schéma récapitulatif des classes de règles

4.4 Combinaisons

Si l'ensemble des règles appartient à une des classes concrètes, alors il suffit d'appliquer les algorithmes correspondant pour répondre aux requêtes. Toutefois, il est possible que ce ne soit pas le cas, et il faut donc pouvoir *découper* l'ensemble des règles de manière à appliquer des méthodes de réponse différentes en fonction du sous-ensemble.

Dans cette optique nous utilisons les composantes fortement connexes du graphe de dépendances de façon à attribuer des étiquettes différentes à celles-ci en fonction des classes concrètes auxquelles leurs ensembles de règles appartiennent.

Puis une fois chaque sous ensemble étiqueté, il faut encore vérifier que celles-ci sont compatibles entre elles. Pour cela, nous définissons le graphe orienté des composantes fortement connexes associé $G_C = (V_C, E_C)$ tel que son ensemble de sommets est l'ensemble des composantes du graphe, et qu'il existe un arc entre deux composantes C_i et C_j si et seulement s'il existe un arc d'un sommet de C_i vers un sommet de C_j. Par définition des composantes fortement connexes, ce graphe est évidemment sans circuit.

On dit que C_i *précède* C_j s'il n'existe aucun arc de C_j vers C_i, et on note cette relation $C_i \triangleright C_j$.

De plus, on définit une fonction $etiquette : V_C \to \{FES, GBTS, FUS\}$ qui associe à chaque C_i une étiquette qui déterminera la classe abstraite considérée pour cette composante.

La propriété suivante s'assure que l'ensemble des règles est décidable : $\{C_i : etiquette(C_i) = FES\} \triangleright \{C_i : etiquette(C_i) = GBTS\} \triangleright \{C_i : etiquette(C_i) = FUS\}$ C'est à dire qu'aucune règle *FES* ne doit dépendre d'une règle *FUS* ou *GBTS* , et qu'aucune règle *GBTS* ne doit dépendre d'une règle *FUS* .

En effet les algorithmes de chaînage arrière par exemple réécrivent la requête jusqu'à ce qu'elle corresponde à la base, tandis que ceux avant ajoutent des faits jusqu'à obtenir

la requête. Il est donc évident que si une composante n'accepte que le chaînage arrière, il ne doit exister aucune règle de celle-ci de laquelle dépende une règle de la composante acceptant uniquement le chaînage avant (si tel était le cas, cette règle ne serait jamais déclenchée).

Chapitre 5

Implémentation

5.1 Structures de donnée

Nous utilisons de nombreux graphes différents que ce soit pour stocker des informations ou même pour vérifier les contraintes de certaines classes de règles. Ainsi nous avons mis en place une structure générique, permettant de choisir le type de sommets et d'arcs, ainsi que de coder différemment les ensembles d'arêtes (ou d'arcs) et de sommets de manière différentes selon la situation, tout en fournissant les algorithmes de graphes classiques.

5.1.1 Règle

Nous avons donc choisi d'utiliser la représentation graphique des règles telle que définie dans la section 2.6. Ainsi une structure de graphe biparti non orienté a été mise en place, l'ensemble des arêtes est codé par une liste de voisinage pour chaque sommet. En effet l'opération de parcours des voisins doit être la plus efficace possible, étant donné qu'elle nécessaire pour grand nombre d'algorithmes ainsi pour la création des atomes. Le graphe étant biparti, l'ensemble de sommets est divisé en deux parties permettant des accès rapides à l'une comme à l'autre, mais perdant de l'efficacité sur certaines opérations peu fréquentes telles que la suppression d'un sommet. Les arêtes sont quant à elle typées par un entier, permettant de connaître la position d'un terme dans un atome, les termes devraient être de préférence ajoutés dans l'ordre (et après les prédicats) de manière à faciliter leur parcours (et à optimiser le temps nécessaire à la création de la règle).

Notons tout de même qu'une règle est une spécialisation d'une conjonction d'atomes, et que c'est cette-dernière qui est en charge de la gestion du graphe hormis la séparation entre l'hypothèse et la conclusion.

5.1.2 Graphe de dépendances des règles

Ensuite, le graphe de dépendances des règles est quant à lui un graphe orienté dont les arcs sont également codés par listes de voisinage et ne possèdent pas de poids. Il est évident que les arcs d'un graphe de dépendances ont plus intérêt à être implémentés de cette manière puisque l'objectif de construire un tel graphe est de connaître rapidement de quel sommet dépend quel autre.

5.1.3 Graphe de dépendances des positions

Enfin le graphe de dépendances des positions utilisés pour vérifier l'appartenance à certaines classes de règles (faiblement acyclique (4.2.2) et faiblement sticky (4.2.4)) est aussi un graphe orienté, mais ici la fonction de poids sur les arcs permet de savoir si un arc est *spécial* ou non. Les sommets représentent les positions des prédicats et sont stockées à la suite. De plus, une table de hachage gère l'accès à la première position d'un prédicat en fonction de son nom, ainsi les opérations d'accès sont aussi légères que possible, et ce type de graphe n'étant conservé en mémoire que le temps de la vérification des contraintes, l'espace supplémentaire utilisé par la table est négligeable.

5.2 GRDAnalyser

Les règles (les données) sont donc stockées directement dans le graphe de dépendances des règles dont une instance est encapsulée dans le *GRDAnalyser*. De plus le *GRDAnalyser*est en fait constitué de deux parties distinctes supplémentaires : la première est en charge de la détermination des classes concrètes, tandis que la seconde vérifie si la base est décidable et combine les classes abstraites de manière à savoir quels algorithmes utiliser.

5.3 Détermination d'une classe concrète

En section 4.2 nous avons défini de nombreuses classes de règles qu'il faut donc pouvoir déterminer. Pour cela, nous avons déclaré une interface de fonction *DecidableClassCheck* fournissant une méthode renvoyant une étiquette à partir d'un ensemble de règles. L'analyseur de classes concrètes contient une liste des contraintes à tester, et lors de son exécution, il vérifie tout d'abord l'ensemble complet des règles sur chacune de celles-ci, puis ensuite sur chaque composante fortement connexe du graphe de dépendances.

5.4 Combinaison des classes abstraites

Comme expliqué plus en détails dans la section 4.4 il est ensuite nécessaire de vérifier si l'ensemble est bel et bien décidable. L'analyseur de classes abstraites regarde donc tout d'abord si l'ensemble des règles est étiqueté par une classe concrète, si tel est le cas, une des approches pour répondre à une requête est donc d'exécuter l'algorithme correspondant. Si ce n'est pas le cas, il associe la valeur 1 à l'étiquette FES, 2 à $GBTS$ et 3 à FUS de manière à avoir un ordre sur celles-ci, puis il effectue un parcours en largeur du graphe des composantes fortement connexes à partir de l'ensemble des sources de celui-ci, attribuant à chaque sommet découvert (qu'il soit déjà traité ou non) la plus petite étiquette fournie par ses classes concrètes et supérieure à celle de son prédécesseur ou 0 si ce n'est pas possible. Une fois cette opération effectuée, si tous les sommets sont étiquetés par des valeurs strictement positives, l'ensemble de règles est décidable.

5.5 Formats de fichiers

Le graphe de dépendances des règles est capable de charger une base à partir d'un fichier, celui-ci devant être écrit dans un format spécifique : chaque ligne doit être une règle de la forme suivante :
$atome_1; atome_2; ...; atome_n - - > atome_c$
avec n le nombre d'atomes dans l'hypothèse et $atome_c$ l'unique atome de la conclusion et où chaque atome i est écrit : $p_i(t_{i1}, t_{i2}, ..., t_{ik})$
Les termes sont interprêtés comme des constantes s'ils sont encadrés par des simple guillemets.
Par exempe la règle $\forall x(\rightarrow (\exists z))$ doit être écrite :

En plus du format interne ci-dessus, il est également possible de fournir un fichier Datalog (.dtg) qui ne contient que des règles à hypothèse atomique. Chaque ligne est soit une règle, soit un commentaire auquel cas elle doit débuter par //. Ici les règles sont sous le format suivant :
[!]$atome_c$:- $atome_h$.
Le point d'explamation est utilisé pour signaler la négation d'une conclusion, celle-ci sera convertie en une règle contenant ses deux atomes actuels dans son hypothèse et ayant une conclusion au prédicat spécial $ABSURD$. De plus, les termes sont maintenant considérés comme des variables s'ils commencent par un point d'interrogation et comme des constantes sinon.
Ainsi la ligne du fichier correspondant à la règle $\forall x(\rightarrow (\exists z))$ doit être :
.

Chapitre 6

Perspectives

Bibliographie

[1] Andrea Calì, Georg Gottlob, and Andreas Pierris. Query Answering under Non-Guarded Rules in Datalog+/-.

[2] Marie-Laure Mugnier. Ontological Query Answering with Existential Rules.