

PROJET TUTORÉ

Optimisation d'analyse syntaxique par réécriture de programme

Auteurs :

Maxime GUILLAUME

Guilherme RAZET

Superviseur :

Sylvain POGODALLA

INSTITUT DES SCIENCES DU DIGITAL - MANAGEMENT ET COGNITION

UNIVERSITÉ DE LORRAINE

LORIA

6 juin 2019

Table des matières

Table des matières	1
I Présentation du sujet	5
1 Datalog	7
1.1 Syntaxe	7
1.2 Sémantique	9
1.3 Évaluation	10
2 Construction de profils de liage uniques	13
2.1 Profils de liage	13
2.2 Construction du Rule/Goal Graph	14
2.3 Séparation des prédicats	16
3 Réécriture Magic set	17
3.1 Définition	17
3.2 Algorithme de réécriture	17
3.3 Variante	20
3.4 Exemple de réécriture	20
II Travail réalisé	23
4 Implémentation	25
4.1 Environnement	25
4.2 Livrables	25
4.3 Résultats	26
5 Réécriture des preuves magiques	29
5.1 Intuition	29
5.2 Algorithmes et propriétés	31
Bibliographie	35
Annexes	37
Exemple de réécriture magique	37

Introduction

Ce rapport présente notre projet tutoré portant sur l'optimisation d'analyse syntaxique par réécriture de programme, réalisé sous la tutelle de Sylvain POGODALLA, chercheur au sein de l'équipe Sémagramme du Laboratoire lorrain de recherche en informatique et ses applications.

Notre projet s'inscrit dans le formalisme des grammaires catégorielles abstraites (ACG) (de Groote 2001), un objet mathématique permettant de représenter différents formalismes grammaticaux comme par exemple les grammaires d'arbres adjoints (Joshi et Schabes 1997) ou les grammaires hors-contexte.

L'analyse syntaxique d'une classe particulière d'ACG peut être traduite sous la forme d'un programme Datalog (Kanazawa 2007), un langage de programmation logique pour la gestion de bases de données (Ullman 1983). Réaliser l'analyse syntaxique d'un point de vue des ACG est isomorphe à la recherche d'une preuve d'une requête dans ce programme Datalog.

Si nous prenons l'exemple d'une grammaire hors-contexte permettant de générer le langage $\{a^n b^n \mid n \geq 0\}$, il est possible de réduire l'analyse syntaxique dans le contexte de cette grammaire à un programme Datalog et une requête. (voir exemple 1).

Il existe plusieurs algorithmes pour évaluer un programme, avec plus ou moins d'efficacité en fonction des spécificités de chaque algorithme. On peut par exemple citer l'algorithme Query Sub-Query (QSQ) (Abiteboul, Hull et Vianu 1995, chapitre 13) ou l'algorithme d'évaluation dite "semi-naïve".

Pour améliorer les performances de ces algorithmes, il est également possible de réécrire les programmes Datalog avant leur évaluation, en suivant certaines méthodes spécifiques comme par exemple la réécriture "Magic Set".

Cette réécriture entraîne une modification profonde des programmes et par conséquent des preuves, ce qui compromet l'isomorphisme de l'analyse syntaxique des ACG avec la recherche de preuve pour une requête et un programme donnés.

L'objectif de ce projet tutoré est de développer la réécriture Magic Set dans ACGtk (Pogodalla 2016), une implémentation des ACG qui comprend un prouveur Datalog, et de réfléchir à un processus pour retrouver l'isomorphisme après réécriture.

Dans un premier temps, nous allons présenter le sujet de notre projet tutoré, et plus particulièrement Datalog, la réécriture Magic Set et les prétraitements nécessaires, puis nous expliciterons le travail réalisé avec l'implémentation de la réécriture Magic Set ainsi que la réécriture des preuves.

Exemple 1 (Transformation d'une grammaire hors-contexte en programme Datalog). *Grammaire hors-contexte générant le langage $\{a^n b^n \mid n \geq 0\}$:*

$$S \rightarrow aSb \quad (g1)$$

$$S \rightarrow \epsilon \quad (g2)$$

Cette grammaire peut être traduite dans le programme Datalog suivant :

$$S(i, l) \leftarrow a(i, j), S(j, k), b(k, l) \quad (d1)$$

$$S(i, i) \leftarrow \quad (d2)$$

Prenons un mot appartenant au langage $aabb$, il est possible de le découper en utilisant un système de position :

$${}_0a_1a_2b_3b_4$$

— *a est présent entre la position 0 et 1 et également entre la position 1 et 2, $a(0, 1)$ et $a(1, 2)$ constituent des faits.*

— b est présent entre la position 2 et 3 et également entre la position 3 et 4, $b(2, 3)$ et $b(3, 4)$ sont donc acquis.

Déterminer si $aabb$ appartient au langage revient à se demander si le programme, à partir des faits énumérés précédemment, génère une preuve du fait $S(0, 4)$. Cette preuve est tout à fait productible à partir de ce programme voir figure 1. Pour constituer $S(0, 4)$, notre requête, l'utilisation de la règle d1 est nécessaire, nous avons connaissance de $a(0, 1)$ et $b(3, 4)$, il reste à déterminer si il est possible de produire $S(1, 3)$, pour cela nous utilisons encore la règle d1. $a(1, 2)$ et $b(2, 3)$ sont des faits connus, il reste à prouver que $S(2, 2)$ est un fait prouvable, pour cela, nous utilisons la règle d2.

Si nous sélectionnons un mot qui n'appartient au langage comme aab , le découpage donnera :

$${}_0a_1a_2b_3$$

Nous cherchons donc à prouver la requête $S(0, 3)$, la figure 2 présente l'échec d'une telle preuve. Pour obtenir le fait $S(0, 3)$, nous avons pour fait $a(1, 2)$ et $S(2, 2)$ mais il nous manque un fait à propos de b qui correspond.

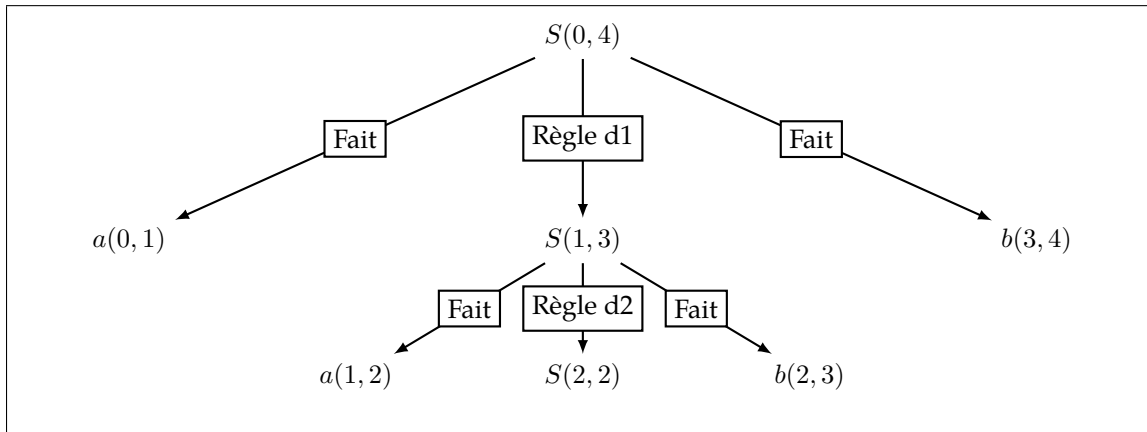


FIGURE 1: Preuve de aabb

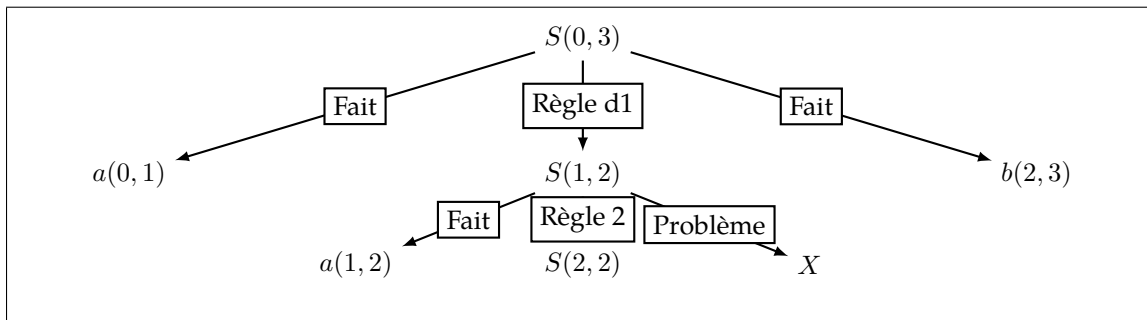


FIGURE 2: Tentative de preuve de aab

Première partie

Présentation du sujet

Chapitre 1

Datalog

Datalog est un langage basé sur la logique formelle, notamment utilisé pour interroger des bases de données déductives. Nous entendons par base de données déductive, un système qui permet de déterminer de nouveaux faits à partir de faits initiaux et de règles.

Datalog est la brique élémentaire de ce projet, c'est la raison pour laquelle nous présentons sa syntaxe, sa sémantique et certaines manières d'évaluer des programmes. Le travail présenté se base sur l'étude de Abiteboul, Hull et Vianu 1995 et d' Ullman 1990.

1.1 Syntaxe

Nous introduisons dans cette section les éléments de la syntaxe et un certain nombre de définitions qui sont nécessaires pour aborder la sémantique et l'évaluation d'un programme.

Définition 1 (Constante). *Une constante est une séquence de chiffres ou de lettres qui commence par une majuscule.*

Définition 2 (Variable). *Une variable est une séquence de lettres qui commence par une minuscule.*

Définition 3 (Terme). *Un terme est une constante ou une variable.*

Définition 4 (Symbole prédicatif). *Un symbole prédicatif est une séquence de lettres.*

Définition 5 (Symbole atomique). *Un symbole atomique (ou prédicat) est une chaîne de la forme $P(t_1, \dots, t_n)$ avec P , un symbole prédicatif et t_1, \dots, t_n des termes.*

Définition 6 (Règle Datalog). *Une règle Datalog est une expression de la forme :*

$$P_0(u_0) \leftarrow P_1(u_1), \dots, P_n(u_n)$$

avec :

- u_i : un vecteur de termes ;
- $R_k(u_k)$: un symbole atomique ;
- $n \geq 0$.

Définition 7 (Tête de la règle). *Dans une règle quelconque :*

$$P_0(u_0) \leftarrow P_1(u_1), \dots, P_n(u_n)$$

la partie gauche de la règle $P_0(u_0)$ est appelée la tête.

Définition 8 (Corps de la règle). *Dans une règle quelconque :*

$$P_0(u_0) \leftarrow P_1(u_1), \dots, P_n(u_n)$$

la partie droite $P_1(u_1), \dots, P_n(u_n)$ est appelée le corps.

Remarque 1 (Absence de corps). *La définition syntaxique des règles autorise les règles sans corps.*

Définition 9 (Sous-buts). *Les prédicats qui appartiennent au corps de la règle sont appelés sous-buts.*

Exemple 2 (Règle Datalog).

$$\text{MusiqueDisco}(\text{titre}, \text{artiste}) \leftarrow \text{Musique}(\text{titre}, \text{artiste}), \text{Genre}(\text{titre}, \text{artiste}, 1980)$$

Cette règle est composé de plusieurs éléments :

- La tête de cette règle est $\text{MusiqueDisco}(\text{titre}, \text{artiste})$;
- Le corps de cette règle est $\text{Musique}(\text{titre}, \text{artiste}), \text{Genre}(\text{titre}, \text{artiste}, 1980)$;
- Les deux sous-buts de cette règle sont par conséquent : $\text{Musique}(\text{titre}, \text{artiste})$ et $\text{Genre}(\text{titre}, \text{artiste}, 1980)$;
- Les trois prédicats de cette règle sont $\text{MusiqueDisco}(\text{titre}, \text{artiste})$, $\text{Musique}(\text{titre}, \text{artiste})$ et $\text{Genre}(\text{titre}, \text{artiste}, 1980)$;
- Les termes de cette règle sont :
 - titre ;
 - artiste ;
 - 1980;
- Les variables de cette règle sont titre et artiste ;
- La seule constante de cette règle est 1980.

Définition 10 (Programme Datalog). Un programme Datalog noté \mathcal{P} est un ensemble fini de règles.

Nous avons désormais à notre disposition tous les éléments pour construire syntaxiquement un programme. Nous introduisons maintenant une distinction de nature des prédicats, ainsi que des notions basées sur ces définitions qui seront utilisées tout au long de ce rapport.

Définition 11 (Prédicat extensionnel). Un prédicat extensionnel est un prédicat qui n'existe que dans le corps des règles de \mathcal{P} .

Définition 12 (Prédicat intentionnel). Un prédicat intentionnel est un prédicat qui apparaît dans la tête des règles de \mathcal{P} .

Définition 13 (Schéma extensionnel). le schéma extensionnel, $\text{edb}(\mathcal{P})$ est l'ensemble des prédicats extensionnels. C'est intuitivement l'entrée d'un programme Datalog.

Définition 14 (Schéma intentionnel). le schéma intentionnel, $\text{idb}(\mathcal{P})$ est l'ensemble des prédicats intentionnels. C'est, en quelque sorte, la sortie d'un programme Datalog.

Il est nécessaire de comprendre la notion d'instance et d'instanciation pour pouvoir expliquer comment évaluer un programme.

Définition 15 (Instanciation). Une instanciation d'une règle R correspond à :

$$P_0(\nu(u_0)) \leftarrow P_1(\nu(u_1)), \dots, P_n(\nu(u_n))$$

Où chaque variable dans les vecteurs de termes est remplacé par l'application de ν à cette variable, avec ν une fonction de substitution définie sur l'ensemble des variables vers l'ensemble des constantes.

Exemple 3 (Instanciation de la règle musicale). Soit

$$\text{MusiqueDisco}(\text{titre}, \text{artiste}) \leftarrow \text{Musique}(\text{titre}, \text{artiste}), \text{Genre}(\text{titre}, \text{artiste}, 1980)$$

une règle, et ν une fonction de substitution définit comme :

$$\begin{aligned} \nu(\text{titre}) &= \text{TakeOnMe} \\ \nu(\text{artiste}) &= \text{Aha} \end{aligned}$$

Une instance de cette règle peut être :

$$\text{MusiqueDisco}(\text{TakeOnMe}, \text{Aha}) \leftarrow \text{Musique}(\text{TakeOnMe}, \text{Aha}), \text{Genre}(\text{TakeOnMe}, \text{Aha}, 1980)$$

Définition 16 (Fait). Un fait est une instanciation d'un prédicat.

Définition 17 (Instance). Une instance I est un ensemble de faits.

Exemple 4 (Programme Datalog musicale).

$$\text{MusiqueDisco}(\text{titre}, \text{artiste}) \leftarrow \text{Musique}(\text{titre}, \text{artiste}), \text{Genre}(\text{musique}, \text{artiste}, 1980)$$

$$I = \left\{ \begin{array}{l} \text{Musique}(\text{Born_to_be_alive}, \text{Patrick_Hernandez}), \\ \text{Genre}(\text{Born_to_be_alive}, \text{Patrick_Hernandez}, 1980) \end{array} \right\}$$

Le programme est composé d'une règle et nous avons une instance composée de deux faits provenant de prédicats dans l'edb.

1.2 Sémantique

Nous présentons dans cette section deux manières équivalentes de voir la sémantique de Datalog. La théorie des points fixes est un moyen particulièrement simple de l'introduire, c'est la raison pour laquelle nous avons fait le choix de l'explicitier. Quant à la perspective "théorie de la preuve", elle est motivée par le fait que dans Acgk nous avons besoin des preuves pour produire des transformations par isomorphisme.

Points fixes

Il est possible de construire la sémantique de Datalog en se basant sur la théorie des points fixes. Pour cela, nous définissons la notion de conséquence immédiate et en présentons une exemplification exemple 5.

Définition 18 (Conséquence immédiate). *Un fait F est une conséquence immédiate pour un programme \mathcal{P} et une instance K dans les cas suivant :*

- $F \in K$ avec F un prédicat dans edb ;
- F est la tête d'une règle instancié : $F \leftarrow R_1, \dots, R_n$ dans \mathcal{P} où chaque $R_i \in K$.

Exemple 5 (Conséquence immédiate sur le programme musicale).

$$\text{MusiqueDisco}(\text{titre}, \text{artiste}) \leftarrow \text{Musique}(\text{titre}, \text{artiste}), \text{Genre}(\text{musique}, \text{artiste}, 1980)$$

$$K = \left\{ \begin{array}{l} \text{Musique}(\text{Born_to_be_alive}, \text{Patrick_Hernandez}), \\ \text{Genre}(\text{Born_to_be_alive}, \text{Patrick_Hernandez}, 1980) \end{array} \right\}$$

$\text{Musique}(\text{Born_to_be_alive}, \text{Patrick_Hernandez})$ est une conséquence immédiate de \mathcal{P} car il appartient à K .

$\text{MusiqueDisco}(\text{Born_to_be_alive}, \text{Patrick_Hernandez})$ est une conséquence immédiate de \mathcal{P} car :

- $\text{Musique}(\text{Born_to_be_alive}, \text{Patrick_Hernandez}) \in K$;
- $\text{Genre}(\text{Born_to_be_alive}, \text{Patrick_Hernandez}, 1980) \in K$.

Cette sémantique opérationnelle est basée sur un opérateur de conséquence immédiate noté $T_{\mathcal{P}}$ qui pour une instance I et un programme \mathcal{P} calcule toutes les conséquences immédiates de I .

Propriété 1 (Point fixe). K est un point fixe si $T_{\mathcal{P}}(K) = K$

Théorème 1 (Monotonie). Soit \mathcal{P} un programme Datalog, $T_{\mathcal{P}}$ est monotone.

Théorème 2 (Point fixe minimum). Pour chaque programme Datalog \mathcal{P} et une instance I , $T_{\mathcal{P}}$ possède un point fixe minimum qui est égale à $\mathcal{P}(I)$.

Théorie de la preuve

L'intuition derrière cette interprétation de la sémantique est que la sortie d'un programme \mathcal{P} sur une instance I est l'ensemble des faits qui peuvent être prouvés.

Nous introduisons la notion de preuve pour Datalog.

Définition 19 (Arbre de preuve). *Un arbre de preuve d'un fait F produit par \mathcal{P} et I est un arbre étiqueté où :*

- Chaque nœud de l'arbre est étiqueté par un fait ;

- Les feuilles sont étiquetées par un fait de l'instance I ;
- La racine est étiquetée par le fait F ;
- Pour chaque nœud interne, il existe une instanciation d'une règle R de \mathcal{P} tel que le nœud est étiqueté par le prédicat de tête et ses enfants par le corps de la règle.

Nous présentons figure 3 un exemple d'arbre de preuve pour le programme suivant :

$$\begin{aligned} S(x_1, x_3) &\leftarrow T(x_1, x_2), R(x_2, a, x_3) \\ T(x_1, x_4) &\leftarrow R(x_1, a, x_2), R(x_2, b, x_3), T(x_3, x_4) \\ T(x_1, x_3) &\leftarrow R(x_1, a, x_2), R(x_2, a, x_3) \end{aligned}$$

et l'instance I :

$$I = \{R(1, a, 2), R(2, b, 3), R(3, a, 4), R(4, a, 5), R(5, a, 6)\}$$

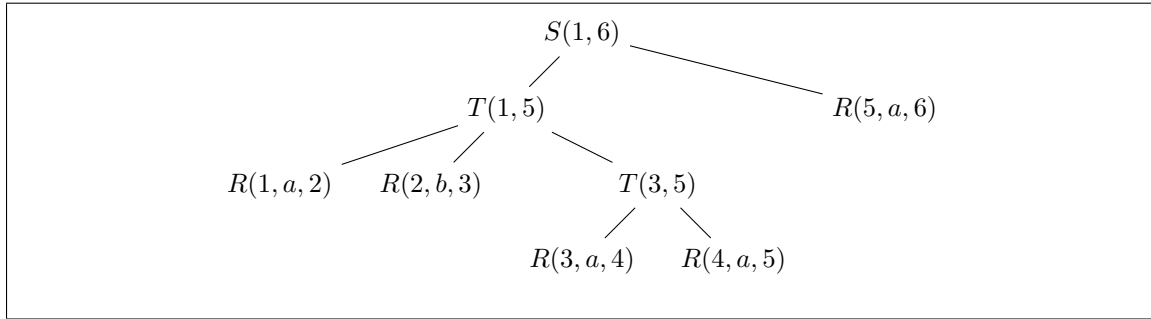


FIGURE 3: Arbre de preuve du fait $S(1, 6)$

Avec :

- $S(1, 6)$ qui correspond au fait F ;
- Les feuilles $\{R(5, a, 6), R(1, a, 2), R(2, b, 3), R(3, a, 4), R(4, a, 5)\}$ qui appartiennent à l'instance
- $T(1, 5)$ et $T(3, 5)$ qui sont des nœuds internes et $T(3, 5)$ qui appartient au corps de la règle instanciée qui donne $T(1, 5)$.

1.3 Évaluation

Nous avons désormais une sémantique pour Datalog, l'évaluation d'un programme Datalog est une question qui reste encore à aborder. Il existe 2 grandes familles d'algorithmes d'évaluation, la famille ascendante et la famille descendante. Nous avons besoin d'introduire la notion de requête et de réponse à une requête pour expliciter la famille descendante et la réécriture Magic set.

Définition 20 (Requête). Une requête q est un prédicat.

Définition 21 (Réponse à une requête). La réponse à une requête q sur un programme \mathcal{P} correspond à vrai si dans tous les faits qu'il est possible de produire il existe une instanciation de la requête ou faux si il n'en existe aucune.

Évaluation descendante (Bottom-up) Nous commençons avec les faits de l'instance, et nous essayons d'appliquer les règles pour produire de nouveaux faits tant que c'est possible ;

Évaluation ascendante (Top-down) Nous définissons une requête et nous nous demandons si il est possible de l'obtenir par le biais des règles et de l'instance.

Évaluation ascendante dite "naïve"

Nous introduisons l'algorithme qui possède l'intuition la plus simple. L'idée est simplement de calculer les faits tant qu'il est possible produire des nouveaux faits.

Algorithme 1 : Évaluation naïve d'un programme Datalog
Données : \mathcal{P} : : Un programme I : : Une instance Résultat : K : : Une instance qui contient tout les faits dérivables $J \leftarrow I$; répéter $K \leftarrow J$; $J \leftarrow T_{\mathcal{P}}(K)$; jusqu'à $K = J$; retourner K

Exemple 6 (Évaluation naïve). *pour un programme qui calcule la fermeture transitive d'un graphe :*

$$\begin{aligned} T(x, y) &\leftarrow G(x, y) \\ T(x, y) &\leftarrow G(x, z), T(z, y) \end{aligned}$$

et une instance I :

$$I = \{G(1, 2), G(2, 3)\}$$

Le déroulement de l'algorithme donne :

$$\begin{aligned} K_1 &= I \\ K_2 &= K_1 \cup \{T(1, 2), T(2, 3)\} \\ K_3 &= K_1 \cup K_2 \cup \{T(1, 3)\} \\ K_4 &= K_3 \end{aligned}$$

Évaluation ascendante dite "semi-naïve"

Il existe une version plus optimisée de l'algorithme précédent qu'on appelle l'évaluation semi-naïve. Dans l'exemple précédent 1, nous voyons très clairement qu'on recalcule tout à chaque itération. Cette évaluation permet de prendre en considérant ce fait en utilisant uniquement les nouveaux faits générés à chaque itération.

C'est cet algorithme qui est utilisé dans Acgtk, La réécriture syntaxique "Magic Set" est conçue pour fonctionner avec cet algorithme.

Évaluation descendante

Une autre famille d'évaluation demeure, c'est l'évaluation descendante avec notamment Query-Subquery (QSQ). Cet algorithme débute par un programme et une requête et tente de dériver un arbre de preuve de la requête.

L'accent est mis uniquement sur les faits qui sont intéressants. Pour cela, nous considérons qu'un fait est utile si et seulement si il apparaît comme fait dans l'arbre de preuve de la requête.

Réécriture et évaluation

Nous présentons un tableau récapitulatif 1 des différentes possibilités pour évaluer un programme Datalog issu de Ceri, Gottlob et Tanca 1989. En dehors de l'évaluation, la littérature s'est également intéressée à la réécriture des programmes pour faire des optimisations, c'est-à-dire la construction d'un nouveau programme plus efficace qui donne exactement la même réponse que le

programme d'origine. Il existe deux familles de réécriture : les réécritures logiques et les réécritures algébriques. Le tableau 2 issu de Ceri, Gottlob et Tanca 1989 donne un aperçu des réécritures possibles.

Nous nous intéresserons dans ce rapport à une réécriture logique en particulier : Magic Set. Celle-ci produit un programme qui simule l'évaluation QSQ quand elle est évaluée par l'algorithme d'évaluation semi-naïf.

	Évaluation ascendante	Évaluation descendante
Méthodes d'évaluation	Naïve Semi-naïve Gauss-Seidel Henschen-Naqv	QSQ

TABLE 1 – Évaluations de programme Datalog

	Réécriture logique	Réécriture algébrique
Méthodes de réécriture	Magic Set Counting Magic Counting Static filtering	Variables and constants réduction

TABLE 2 – Réécritures de programme Datalog

Chapitre 2

Construction de profils de liage uniques

La technique de réécriture Magic set nécessite que le programme Datalog possède une propriété particulière que nous allons introduire. Pour cela, nous allons commencer par définir les principes du profil de liage, puis nous aborderons une manière de représenter un programme Datalog comme un graphe.

2.1 Profils de liage

Définition 22 (Terme lié et terme libre). *Un terme peut être lié ou libre (bound ou free en anglais). Il y a plusieurs cas :*

- Si le terme est une constante, il est lié;
- Si le terme est une variable qui n'est jamais introduite dans les sous-buts de la règle, il est libre;
- Si le terme est une variable qui est déjà introduite dans la règle, il est lié.

Pour noter les différents états, nous associons la lettre b (pour bound) si le terme est lié ou la lettre f (pour free) si le terme est libre.

Définition 23 (Profil de liage). *Un profil de liage α contient les lettres définissant l'état des termes. Il est rattaché à un prédicat p , nous le notons p^α . Ce prédicat présente un ornement, nous l'appelons donc "prédicat orné".*

Exemple 7 (Profil de liage d'un prédicat). *pour un prédicat $P(a, b, 3)$ où :*

- a est une variable libre;
- b une variable liée;
- 3 une constante.

le profil de liage α associé est fbb , nous notons donc le prédicat orné $P^{fbb}(a, b, 3)$.

Le profil de liage du prédicat de tête d'une règle est défini par la requête associée au programme Datalog. il est très courant d'utiliser le Rule/Goal Graph présenté dans la prochaine section pour le récupérer, mais pour cet exemple nous allons supposer que le profil de liage de la tête de la règle est donné.

Exemple 8 (profil de liage d'une règle). *pour une règle r :*

$$S^{bf}(a, b) \leftarrow S_1(a, b, c, 3), S_2(a, b, c)$$

Nous traitons les sous-buts dans leur ordre d'apparition, nous commençons donc par S_1 :

- a est un terme utilisé dans la tête, il est donc lié;
- b est un terme utilisé dans la tête, il est donc lié;
- c n'est pas utilisé avant dans la règle, il est donc libre;
- 3 est une constante, il est donc lié.

Le prédicat orné S_1 se présente comme cela :

$$S_1^{bbfb}(a, b, c, 3)$$

Nous traitons maintenant S_2 :

- a apparaît avant dans la règle (tête et S_1), il est donc lié;
 - b apparaît avant dans la règle (tête et S_1), il est donc lié;
 - c apparaît avant dans la règle (S_1), il est donc lié.
- Le prédicat orné S_2 se présente comme cela :

$$S_2^{bbb}(a, b, c)$$

Nous orons donc les prédicats de cette règle de cette manière :

$$S^{bf}(a, b) \leftarrow S_1^{bbfb}(a, b, c, 3), S_2^{bbb}(a, b, c)$$

2.2 Construction du Rule/Goal Graph

L'objectif est ici de produire un graphe qui représente les façons de produire un fait. Les éléments nécessaires à la construction seront donc un programme et une requête. Ce graphe possédera deux types de nœuds bien distincts : les nœuds buts qui représenteront les prédicats ornés et les nœuds règles qui sont des représentations d'une règle et d'une position de lecture de la règle. L'exemple 9 présente une construction détaillée de ce graphe pour le programme Datalog qui produit le langage $a^n b^n$.

La première étape est de construire un graphe avec un unique nœud but p^α , le prédicat p de la requête et son ornement α . Nous appliquons les règles ci-dessous pour générer la totalité du graphe.

1. Un nœud but dans edb n'a pas de successeur.
2. Un nœud but p^α dans idb possède des successeurs. Ce sont toutes les règles qui ont pour tête p . Pour une règle R qui satisfait cette condition, nous créons le nœud :

$$R_{i,0}^{B,F}$$

avec

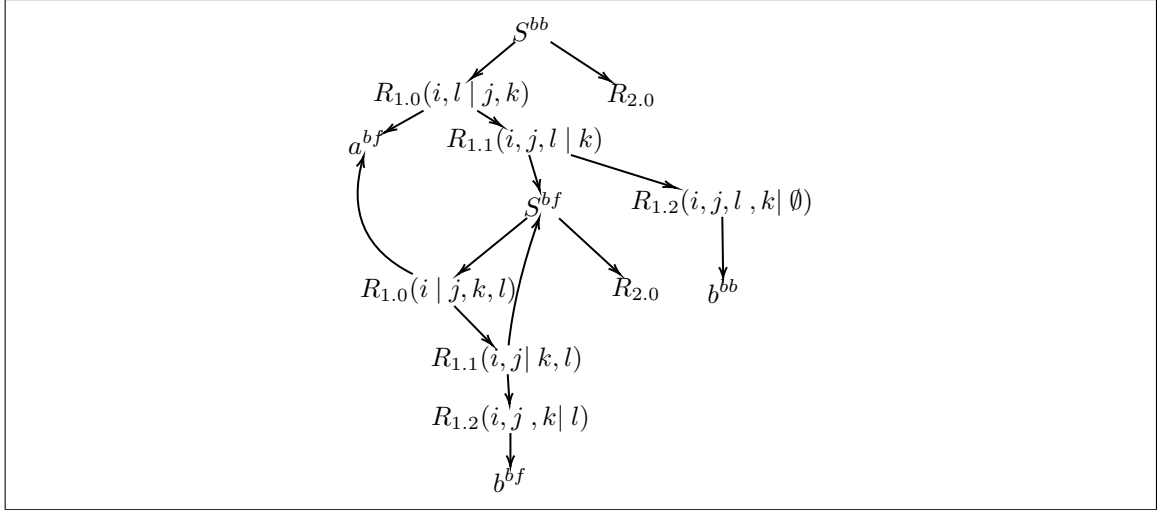
- B les variables liées dans la tête;
- F les autres variables de la règle;
- i l'identifiant de la règle;
- 0 la position de lecture initiale.

3. Un nœud règle $R_{i,j}$ correspondant à la règle $i : P_0(U_0) \leftarrow$ n'a pas de successeur;
4. Un nœud règle $R_{i,j}$ correspondant à la règle $i : P_0(U_0) \leftarrow P_1(U_1), \dots, P_n(U_n)$ et $j = n$ possède un successeur, Le nœud but du sous-but j avec son profil de liage.
5. Un nœud règle $R_{i,j}$ correspondant à la règle $i : P_0(U_0) \leftarrow P_1(U_1), \dots, P_n(U_n)$ et $j < n$ possède deux successeurs :
 - Le nœud but du sous-but j avec son profil de liage;
 - Le nœud règle :

$$R_{i,j+1}^{B',F}$$

avec :

- B' l'union de l'ensemble des variables liées à la position j pour la règle i et le profil de liage α et des variables de P_j ;
- F les autres variables de la règle.

FIGURE 4: Rule/Goal Graph de $a^n b^n$

Exemple 9 (Rule/Goal Graph du programme Datalog représentant $a^n b^n$). Soit \mathcal{P} le programme Datalog qui représente une grammaire qui génère le langage $a^n b^n$:

$$S(i, l) \leftarrow a(i, j), S(j, k), b(k, l) \quad (1)$$

$$S(i, i) \quad (2)$$

et q :

$$S_{bb}(0, 4)$$

La figure 4 présente le Rule/Goal Graph du programme \mathcal{P} pour la requête q . Nous commençons par la racine, qui est la requête q . Il s'agit d'un nœud but qui est la tête de 2 règles dans \mathcal{P} : les règles (1) et (2). Nous créons donc les nœuds $R_{1.0}(i, l | j, k)$ et $R_{2.0}$. Les nœuds à traiter sont :

$$\overline{\text{Nœuds à traiter : } R_{1.0}(i, l | j, k) \quad R_{2.0}}$$

$R_{1.0}(i, l | j, k)$ est un nœud règle et il a deux successeurs : a^{bf} et $R_{1.1}(i, l, j | jk)$. Les nœuds à traiter sont :

$$\overline{\text{Nœuds à traiter : } R_{2.0} \quad a^{bf} \quad R_{1.1}(i, l, j | jk)}$$

$R_{2.0}$ est un nœud règle sans corps, il n'a donc pas de successeur.

Les nœuds à traiter sont :

$$\overline{\text{Nœuds à traiter : } a^{bf} \quad R_{1.1}(i, l, j | jk)}$$

a^{bf} est un nœud but dans l'edb, il n'a donc pas de successeur. Les nœuds à traiter sont :

$$\overline{\text{Nœuds à traiter : } R_{1.1}(i, l, j | jk)}$$

$R_{1.1}(i, l, j | jk)$ est un nœud règle et il a deux successeurs : s^{bf} et $R_{1.2}(i, j, l, k | \emptyset)$. Les nœuds à traiter sont :

$$\overline{\text{Nœuds à traiter : } s^{bf} \quad R_{1.2}(i, j, l, k | \emptyset)}$$

Et ainsi de suite.

2.3 Séparation des prédicats

Nous pouvons désormais expliciter la propriété qui est nécessaire pour la réécriture.

Propriété 2 (Unicité des profils de liages). *Un programme P possède des profils de liage unique si lors de la construction du rgg, nous ne trouvons pas un prédicat avec deux ornements différents.*

Il est toujours possible de construire un programme P' qui possède cette propriété. Pour cela, nous construisons le RGG de P et nous appliquons l'algorithme ci-dessous.

1. Pour chaque ornement α d'un prédicat p , si p^α est un nœud but du RGG, alors nous construisons un nouveau prédicat p_α ;
2. Pour chaque règle avec p en tête, nous construisons une copie de la règle avec p_α en tête;
3. Nous parcourons les sous-buts, si c'est un prédicat dans idb , nous utilisons la version ornée du prédicat qui correspond dans le RGG.

Théorème 3 (Correction). *Soit \mathcal{P} un programme Datalog, q^α une requête ornée, \mathcal{P}' la version du programme \mathcal{P} avec des profils de liage unique, et q_α la nouvelle requête : la réponse de \mathcal{P}' à la requête q_α est la même que celle de \mathcal{P} à la requête q^α .*

Exemple 10. *Séparation des prédicats*

Soit P :

$$S(p1, p3) \leftarrow A(p1, p3, p2, p2) \quad (2.1)$$

$$A(p1, p8, p, p5) \leftarrow A(p2, p7, p3, p6), a(p1, p2), b(p3, p4), c(p5, p6), d(p7, p8) \quad (2.2)$$

$$A(p1, p2, p1, p2) \leftarrow \quad (2.3)$$

L'algorithme donne pour la requête $q = S(0, 8)$:

$$S_bb(p1, p3) \leftarrow A_bbff(p1, p3, p2, p2) \quad (2.4)$$

$$A_ffff(p1, p8, p4, p5) \leftarrow A_ffff(p2, p7, p3, p6), a(p1, p2), b(p3, p4), c(p5, p6), d(p7, p8) \quad (2.5)$$

$$A_bbff(p1, p8, p4, p5) \leftarrow A_ffff(p2, p7, p3, p6), a(p1, p2), b(p3, p4), c(p5, p6), d(p7, p8) \quad (2.6)$$

$$A_bbff(p1, p2, p1, p2) \leftarrow \quad (2.7)$$

$$A_ffff(p1, p2, p1, p2) \leftarrow \quad (2.8)$$

Chapitre 3

Réécriture Magic set

3.1 Définition

La réécriture Magic Set s'applique sur une paire $\langle q, \mathcal{P} \rangle$ avec \mathcal{P} un programme Datalog et q une requête. Son objectif est d'améliorer l'efficacité de l'évaluation de $\langle \mathcal{P}, q \rangle$ en construisant un nouveau programme qui simule l'évaluation descendante QSQ par l'ajout de prédicats particuliers.

Théorème 4 (Correction de la technique Magic set Abiteboul, Hull et Vianu 1995 page 327). Soit :

- $\langle q, \mathcal{P} \rangle$ une paire contenant une requête et un programme Datalog ;
- $\langle m_q, m_{\mathcal{P}} \rangle$ la paire réécrite par Magic set.

La réponse de $\langle m_q, m_{\mathcal{P}} \rangle$ sur I une instance est strictement identique à celle de $\langle q, \mathcal{P} \rangle$ sur I .

L'ensemble des faits produits par l'évaluation semi-naïve de $\langle m_q, m_{\mathcal{P}} \rangle$ est le même que l'évaluation QSQ de $\langle q, \mathcal{P} \rangle$.

3.2 Algorithme de réécriture

Prédicats magiques

Une fois que nous avons assuré l'unicité des profils de liage, nous créons pour chaque prédicat p_α de i_{db} un prédicat dit "magique", $magic_p_\alpha$. Les termes de $magic_p_\alpha$ correspondent aux termes liés de p tels que définis par la construction de profils de liage uniques.

Exemple 11. Création d'un prédicat magique $magic_p$ à partir d'un prédicat p :

Prédicat : $p_bbff(p1, p3, p2, p4)$

Prédicat magique : $magic_p_bbff(p1, p3)$

Le nom du prédicat magique correspond à celui du prédicat, précédé de $magic_$. Les termes de $magic_p$ correspondent aux termes liés de p , à savoir $(p1, p3)$.

Prédicats supplémentaires

Nous créons un prédicat dit "supplémentaire" pour chaque sous-but de chaque règle. Un prédicat supplémentaire est noté $sup_{r,i}$ où :

- r est l'indice de la règle d'origine ;
- i correspond à la position du prédicat supplémentaire dans la règle. Nous commençons à 0 (avant le premier sous-but de la règle), jusqu'à $k - 1$ (k étant la position du dernier sous-but de la règle).

Un prédicat supplémentaire contient :

- Les termes liés dans la tête de la règle r ;
- Les termes présents au moins une fois dans les sous-buts de 0 à $i - 1$ et au moins une fois dans les sous-buts de i à k .

Exemple 12. Création des prédicats supplémentaires à partir d'une règle :

$$S_bb(p1, p4) \leftarrow a(p1, p2), S_bf(p2, p3), b(p3, p4) \quad (1)$$

La règle (1) possède trois sous-buts, nous allons donc créer trois prédicats supplémentaires :

1. $sup_{1.0}(p1, p4);$
2. $sup_{1.1}(p1, p4, p2);$
3. $sup_{1.2}(p1, p4, p3);$

$sup_{1.0}$ étant placé avant les autres sous-buts, il ne contient que les termes liés contenus dans le prédicat de tête $(p1, p4)$.

$sup_{1.1}$ étant placé après $a(p1, p2)$, il contient les termes liés contenus dans le prédicat de tête $(p1, p4)$ ainsi que le terme $p2$, apparaissant dans $a(p1, p2)$ et dans $S_bf(p2, p3)$.

$sup_{1.2}$ étant placé après $S_bf(p2, p3)$, il contient les termes liés contenus dans le prédicat de tête $(p1, p4)$ ainsi que le terme $p3$, mais pas le terme $(p2)$ car il n'apparaît pas dans les sous-buts suivants.

Maintenant que les prédicats magiques et supplémentaires sont créés, nous allons décrire la création ou la modification des règles intégrant ces nouveaux prédicats.

Création des règles magiques

Chaque prédicat magique $magic_p_α$ va permettre de créer une nouvelle règle magique r_magic . Il y a deux étapes à la création d'une règle magique :

1. Construction de la tête de r_magic ;
2. Construction du corps de r_magic ;

La tête de r_magic est constitué du prédicat $magic_p_α$. Elle contient les variables liées de $p_α$ uniquement.

Remarque 2. Si $p_α$ est le sous-but de plusieurs règles, il y aura pour chaque règle r une règle magique r_magic .

Pour chaque r_magic , nous récupérons le prédicat supplémentaire précédent $p_α$ dans r : il s'agit de l'unique sous-but de r_magic .

Exemple 13. Création d'une règle magique à partir d'une règle :

$$S_bb(p1, p4) \leftarrow a(p1, p2), S_bf(p2, p3), b(p3, p4) \quad (1)$$

$S_bf(p2, p3)$ est le seul sous-but de la règle à apparaître dans l'idb, le prédicat magique associé est $magic_S_b(p2)$, et le prédicat supplémentaire associé est $sup_{1.1}(p1, p4, p2)$.

La règle magique créée avec ces éléments est donc :

$$magic_S_bf(p2) \leftarrow sup_{1.1}(p1, p4, p2) \quad (m1)$$

Création de règles pour le premier prédicat supplémentaire

Pour le prédicat supplémentaire noté $sup_{r.0}$ (où r est l'index d'une règle), la création de règle est différente de celle des autres prédicats supplémentaires. Le prédicat de tête de cette règle supplémentaire est le prédicat supplémentaire $sup_{r.0}$, contenant les variables liées du prédicat de tête p de la règle r . L'unique sous-but est le prédicat de tête p de la règle r dans sa version magique (noté $magic_p$); contenant les variables liées de p .

Exemple 14. Création de la règle du premier prédicat supplémentaire à partir d'une règle :

$$S_bb(p1, p4) \leftarrow a(p1, p2), S_bf(p2, p3), b(p3, p4) \quad (1)$$

$sup_{1.0}(p1, p4)$ est le premier prédicat supplémentaire, voici sa règle associée :

$$sup_{1.0}(p1, p4) \leftarrow magic_S_bb(p1, p4) \quad (s1.0)$$

Création de règles pour les autres prédicats supplémentaires

Pour chaque prédicat supplémentaire $sup_{r,i}$ (r est l'indice d'une règle et i la position du prédicat dans cette règle), nous créons une règle qui prend en tête $sup_{r,i}$ avec les mêmes variables que pour le prédicat supplémentaire. Il y a deux sous-buts dans cette règle : le prédicat supplémentaire précédent ($sup_{r,i-1}$) et le prédicat à la position i de la règle r .

Exemple 15. Création des règles supplémentaires à partir d'une règle :

$$S_bb(p1, p4) \leftarrow a(p1, p2), S_bf(p2, p3), b(p3, p4) \quad (1)$$

Prédicats supplémentaires générés :

1. $sup_{1.0}(p1, p4)$;
2. $sup_{1.1}(p1, p4, p2)$;
3. $sup_{1.2}(p1, p4, p3)$.

Nous créons 2 règles supplémentaires :

$$sup_{1.1}(p1, p4, p2) \leftarrow sup_{1.0}(p1, p4), a(p1, p2) \quad (s1.1)$$

avec le prédicat $sup_{1.1}(p1, p4, p2)$ en tête, le prédicat supplémentaire précédent ($sup_{1.0}(p1, p4)$) en premier sous-but et le premier prédicat de la règle (1) ($a(p1, p2)$) en second sous-but.

$$sup_{1.2}(p1, p4, p3) \leftarrow sup_{1.1}(p1, p4, p2), S_bf(p2, p3) \quad (s1.2)$$

avec le prédicat $sup_{1.2}(p1, p4, p3)$ en tête, le prédicat supplémentaire précédent ($sup_{1.1}(p1, p4, p2)$) en premier sous-but et le deuxième prédicat de la règle (1) ($S_bf(p2, p3)$) en second sous-but.

Transformation de règles pour les prédicats de l'idb

Maintenant que nous avons traité tous les prédicats créés par la réécriture (prédicats magiques et supplémentaires), nous allons transformer les règles ayant pour tête un prédicat de l'idb.

Pour chaque règle r correspondant à ce critère, nous gardons uniquement le dernier prédicat supplémentaire correspondant à r , ainsi que le dernier sous-but de r .

Exemple 16. Transformation d'une règle :

$$S_bb(p1, p4) \leftarrow a(p1, p2), S_bf(p2, p3), b(p3, p4) \quad (1)$$

Dernier prédicat supplémentaire : $sup_{1.2}(p1, p4, p3)$

Règle transformée :

$$S_bb(p1, p4) \leftarrow sup_{1.2}(p1, p4, p3), b(p3, p4) \quad (1)$$

Il existe une exception à cette étape : si une règle r ne possède pas de sous-buts, nous construisons la règle avec l'unique prédicat supplémentaire lié à r .

Exemple 17. Transformation d'une règle sans sous-buts :

$$S_bb(p1, p4) \leftarrow \quad (1)$$

Prédicat supplémentaire créé : $sup_{1.0}(p1, p4)$.

Règle transformée :

$$S_bb(p1, p4) \leftarrow sup_{1.0}(p1, p4) \quad (1)$$

Création de la règle d'initialisation

Maintenant que les règles du programme original \mathcal{P} ont été modifiées, nous utilisons la requête q pour créer une règle d'initialisation qui contiendra la version magique de la requête : $magic_q$. Il est important de noter que toute la réécriture dépend de la requête qu'on souhaite traiter, et donc qu'une réécriture est valide uniquement pour la requête qui lui est associée.

Exemple 18. *Création de la règle d'initialisation à partir d'une requête :*

Requête du programme original : $S^{bb}(0, 4)$

Règle d'initialisation associée : $magic_S^{bb}(0, 4)$

3.3 Variante

General Magic Set

Il existe une faiblesse à la version "classique" de Magic Set : en effet, les prédicats magiques et supplémentaires ne contiennent que les termes liés, alors qu'il y a d'autres informations utiles qui pourraient être stockées dans les prédicats. Si nous stockons tous les termes (liés et libres), les informations supplémentaires qui sont contenues dans les prédicats magiques et supplémentaires correspondent parfaitement aux règles du RGG. Cette modification permet aux règles magiques d'être évaluables plus efficacement avec une approche ascendante que dans la version "classique" de Magic Set.

À la lumière de cette faiblesse, voici les changements qui sont appliqués à la méthode de réécriture magique :

- Les prédicats magiques contiennent tous les termes, au lieu de ne contenir que les termes liés ;
- Les prédicats supplémentaires contiennent tous les variables, au lieu de ne contenir que les variables liés.

Ces modifications dans la création des prédicats magiques et supplémentaires ont un impact sur toutes les règles magiques et supplémentaires. Nous appelons cette variante "General Magic Set" (Ullman 1990 (chapitre 13, section 6, pages 860-862)).

Théorème 5 (Efficience de la Technique général magique Ullman 1990 page 870-871). *Soit $\langle gm_q, gm_P \rangle$, une requête magique et un programme version général Magic Set :*

L'évaluation semi-naïve de $\langle gm_q, gm_P \rangle$ est proportionnel au temps de l'évaluation QSQ de $\langle q, P \rangle$ le programme et la requête original.

3.4 Exemple de réécriture

Nous allons ici vous montrer un exemple de réécriture "Classique" et "General" pour $\langle \mathcal{P}, q \rangle$ avec \mathcal{P} :

$$S(i, l) \leftarrow a(i, j), S(j, k), b(k, l) \quad (1)$$

$$S(i, i) \leftarrow \quad (2)$$

et q :

$$S_{bb}(0, 4)$$

Nous n'indiquons ici que les résultats de la réécriture, les détails de la réécriture sont disponibles dans l'annexe 1.

Voici le programme Datalog \mathcal{P} et la requête q après la réécriture magique "Classique" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bf(i, l) &\leftarrow sup_{2.2}(i, k), b(k, l) & (2) \\
sup_{2.0}(i) &\leftarrow magic_S_bf(i) & (s2.0) \\
sup_{2.1}(i, j) &\leftarrow sup_{2.0}(i), a(i, j) & (s2.1) \\
sup_{2.2}(i, k) &\leftarrow sup_{2.1}(i, j), S_bf(j, k) & (s2.2) \\
magic_S_bf(j) &\leftarrow sup_{2.1}(i, j) & (m2) \\
S_bb(i, i) &\leftarrow sup_{3.0}(i, i) & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow sup_{4.0}(i, i) & (4) \\
sup_{4.0}(i) &\leftarrow magic_S_bf(i) & (s4.0) \\
magic_S_bb(0, 4) &\leftarrow & (q)
\end{aligned}$$

Et le programme Datalog \mathcal{P} et la requête q après la réécriture magique "General" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j, k) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (2) \\
sup_{2.0}(i, l) &\leftarrow magic_S_bf(i, l) & (s2.0) \\
sup_{2.1}(i, l, j) &\leftarrow sup_{2.0}(i, l), a(i, j) & (s2.1) \\
sup_{2.2}(i, l, k) &\leftarrow sup_{2.1}(i, l, j), _bf(j, k) & (s2.2) \\
magic_S_bf(j, k) &\leftarrow sup_{2.1}(i, l, j) & (m2) \\
S_bb(i, i) &\leftarrow sup_{3.0}(i, i) & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow sup_{4.0}(i, i) & (4) \\
sup_{4.0}(i) &\leftarrow magic_S_bf(i, i) & (s4.0) \\
magic_S_bb(0, 4) &\leftarrow & (q)
\end{aligned}$$

Les règles présentant un changement entre les 2 différentes versions sont les règles m1, 2, s2.0, s2.1, s2.2, m2 et s4.0.

Deuxième partie

Travail réalisé

Chapitre 4

Implémentation

4.1 Environnement

Ce projet est en interaction avec un environnement logiciel, nous en donnons une brève explication.

Acgtk

Acgtk est une boîte à outils développée au sein de l'équipe Sémagramme par Sylvain Pogodalla pour le test et le développement des ACG.

Ocaml

Ocaml est un langage de programmation écrit par Xavier Leroy, Damien Doligez, Jérôme Vouillon et Didier Rémy au sein de L'INRIA. Il fait partie de la famille des langages ML (Méta-langage). Il est multiparadigme. Ses caractéristiques principales sont :

- Le typage statique;
- L'inférence de types;
- Le polymorphisme paramétrique;
- Le filtrage par motif.

C'est le langage utilisé pour le logiciel Acgtk et pour notre projet tutoré.

OCamlgraph

OCamlgraph (Conchon, Filliâtre et Signoles 2007) est une bibliothèque Ocaml de structures de données efficaces représentant des graphes, basée sur l'utilisation des foncteurs. Elle est utilisée dans notre projet pour modéliser le Rule/Goal Graph.

4.2 Livrables

Nous avons conçu une bibliothèque logicielle qui permet de construire pour un programme Datalog \mathcal{P} , pour chaque prédicat $p \in idb$ et pour chaque profil de liage α , $\mathcal{P}_{magic_p_alpha}$ la version General Magic Set pour la requête p^α .

Cette bibliothèque est basée sur le patron architectural "Filtres et tubes" (très commun dans le domaine du traitement automatique des langues et de la compilation), chaque étape de la réécriture est un filtre (voir figure 5). Elle utilise les composants Datalog définis dans Acgtk ce qui la rend totalement intégrable. En excluant les commentaires et la documentation, elle représente environ 1300 lignes de code source.

Avec l'aide de cette bibliothèque, nous avons programmé un exécutable autonome capable de d'appliquer la réécriture General Magic Set, l'exemple 19 présente pour un programme Datalog, le résultat de cet exécutable.

Exemple 19 (Résultat de l'exécutable). *Pour le programme \mathcal{P} :*

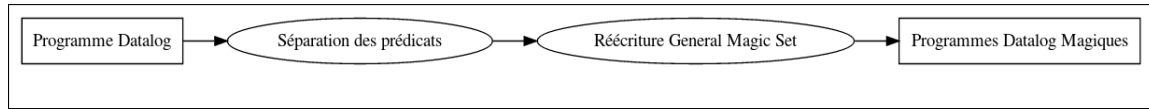


FIGURE 5: Représentation simplifiée de la bibliothèque

$$S(i, l) \leftarrow a(i, j), S(j, k), b(k, l) \quad (\text{d1})$$

$$S(i, i) \leftarrow \quad (\text{d2})$$

L'idb de ce programme est un constitué d'un unique prédicat S d'arité 2. L'ensemble des profils de liage est $\{bb, bf, fb, ff\}$.

notre exécutable va donc construire le programme en version General Magic Set pour chaque prédicat dans $\{S^{bb}, S^{bf}, S^{fb}, S^{ff}\}$, le résultat sera donc une collection qui contient tout les programmes avec un accès direct par requête tel que présenté ci-dessous :

Requête	Programme
S^{bb}	$P_magic_S_bb$
S^{bf}	$P_magic_S_bf$
S^{fb}	$P_magic_S_fb$
S^{ff}	$P_magic_S_ff$

4.3 Résultats

Nous présentons un comparatif (figure 6) entre l'évaluation semi-naïve de \mathcal{P} (en gris) le programme qui produit le langage non-contextuel $\{a^n b^n \mid n \geq 0\}$ et l'évaluation semi-naïve de \mathcal{P}' (en noir) la réécriture General Magic Set de \mathcal{P} .

Dans la figure 6, l'axe des abscisses représente le nombre de a et de b , en ordonnée, nous retrouvons le temps d'évaluation en seconde. La courbe . Nous avons pris pour valeur la moyenne sur dix itérations d'exécution pour chaque n . Nous observons que les deux courbes sont croissantes. Néanmoins, la courbe d'évaluation de \mathcal{P} a une accélération bien plus importante et des valeurs plus grandes à partir de $n > 30$.

La figure 7 présente la comparaison pour un programme \mathcal{P} différent, qui génère le langage contextuel $\{a^n b^n c^n d^n \mid n \geq 0\}$ dans les même conditions que la comparaison précédente avec n le nombre de a, b, c, d . La différence est flagrante, l'évaluation de \mathcal{P} prend de plus en plus de temps tandis qu'à cette échelle, l'évaluation de \mathcal{P}' semble constante. Si nous nous concentrons uniquement sur l'évaluation de \mathcal{P}' , nous observons une courbe semblable à celle de la courbe (en noir) de la figure 6. Pour donner un ordre d'idée, l'évaluation avec $n = 1000$ donne un temps d'évaluation de une seconde.

Il est assez aisé de voir graphiquement que dans les deux cas, l'évaluation semi-naïve simple est significativement plus longue que l'évaluation semi-naïve de la réécriture (en noir). Cela est confirmé par les statistiques inférentielles, nous ne rejetons pas l'hypothèse d'une moyenne significativement différente avec $\alpha = 0.01$ dans les deux cas :

- Pour le cas $a^n b^n$: $p - value = 0,0000188$ et $Q = -4,872$;
- Pour le cas $a^n b^n c^n d^n$: $p - value = 0.000015$ et $Q = -4,94$.

Ces résultats expérimentaux confirment ce qui est attendu d'un point de vue théorique et montre qu'une intégration de cette librairie à Acgk est bénéfique d'un point de vue de l'optimisation de l'analyse syntaxique.

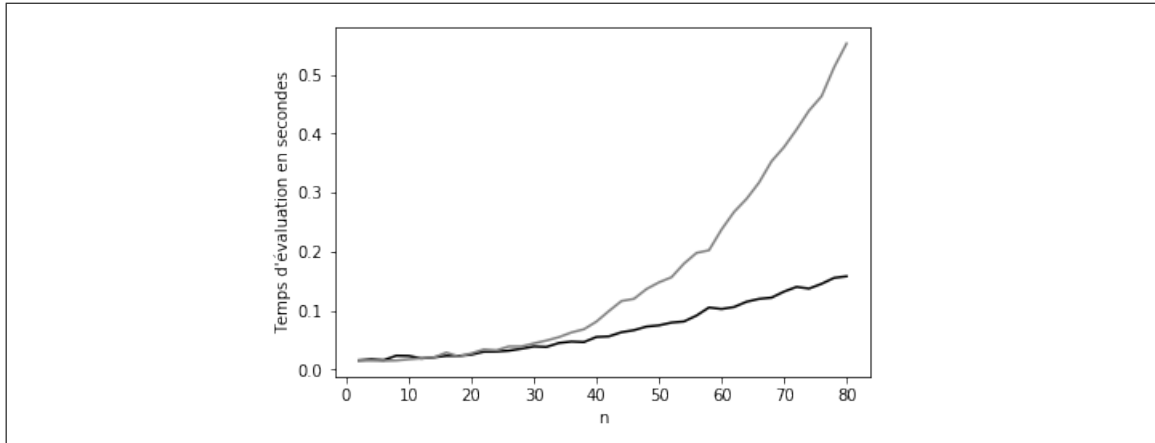


FIGURE 6: Évaluation d'un programme qui génère le langage $\{a^n b^n \mid n > 0\}$

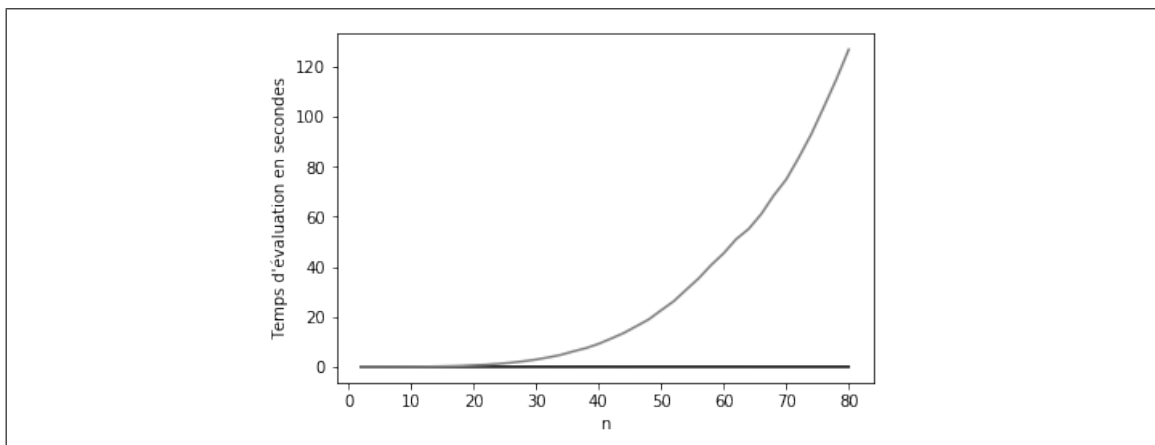


FIGURE 7: Évaluation d'un programme qui génère le langage $\{a^n b^n c^n d^n \mid n > 0\}$

Chapitre 5

Réécriture des preuves magiques

Pour pouvoir pleinement utiliser la réécriture Magic set dans le contexte de Acgk, il est nécessaire d'élaborer un algorithme qui permet de retrouver à partir d'une preuve d'un programme magique, la preuve du même fait dans le programme originale.

En effet, l'analyse syntaxique dans une certaine classe d'ACG peut se ramener à la recherche de la preuve d'une requête dans un programme Datalog \mathcal{P} , toutefois si nous construisons un nouveau programme \mathcal{P}_m par le biais de la réécriture magique les arbres de preuves de \mathcal{P}_m vont différer de ceux de \mathcal{P} . Ce chapitre présente une résolution automatique de cette problématique.

5.1 Intuition

Nous présentons dans cette section, une instance du problème à résoudre ainsi que l'intuition de notre algorithme.

Exemple 20 (Des arbres de preuves différents). Soit \mathcal{P} , le programme qui génère le langage $\{a^n \mid n \geq 0\}$:

$$S(x, z) \leftarrow a(x, y), S(y, z) \quad (5.1)$$

$$S(i, i) \leftarrow \quad (5.2)$$

une instance :

$$I = \{a(0, 1), a(1, 2)\}$$

et $q = S(0, 2)$ une requête.

Nous présentons une preuve de $S(0, 2)$ figure 8 et l'équivalent de cette preuve dans le programme version magique figure 9. Nous observons que le nombre de nœuds et la hauteur de l'arbre diffèrent significativement. Cela est dû aux différentes étapes de la réécriture Magic Set. Cependant, nous constatons que tout les nœuds¹ de la figure 8 sont présents dans la figure 9. La problématique qui nous concerne ici est de transformer l'arbre 9 en l'arbre figure 8 de manière automatique.

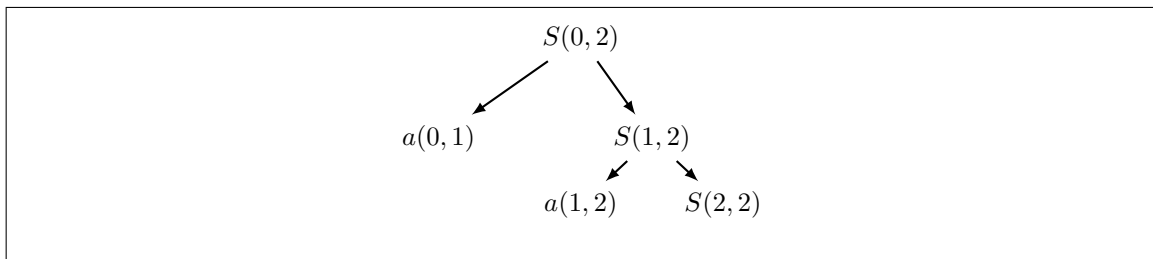
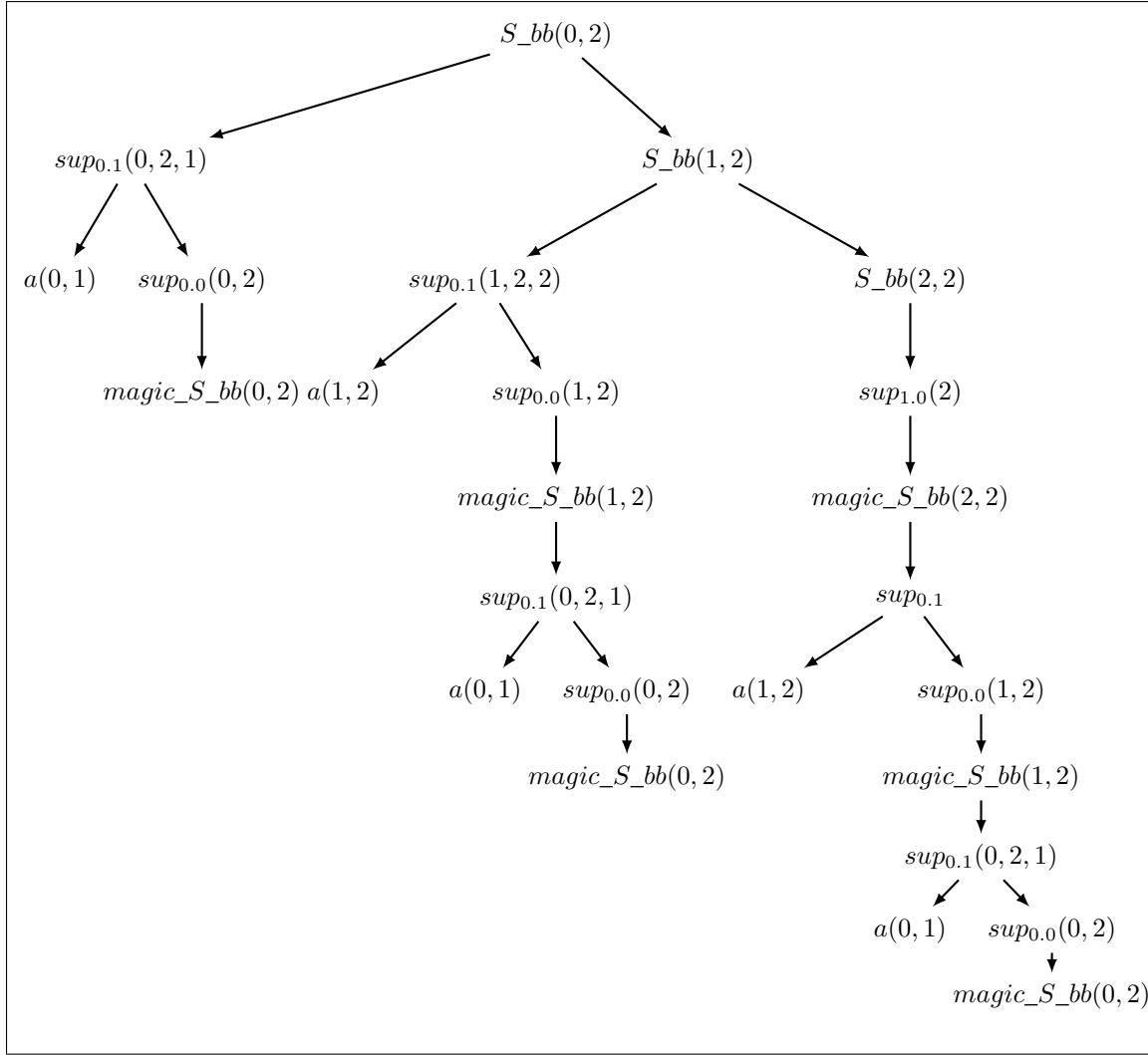


FIGURE 8: Preuve de $S(0, 2)$ dans le programme \mathcal{P}

1. en négligeant les ornements

FIGURE 9: Preuve de $S(0, 2)$ dans le programme \mathcal{P}_m

Aplatissement des preuves des relations supplémentaires

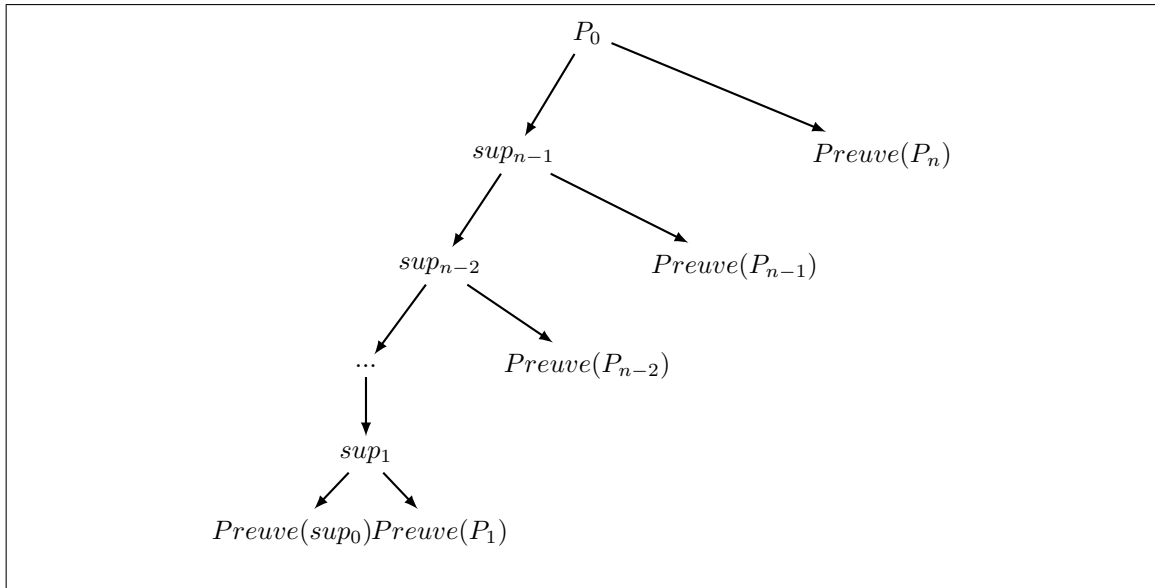
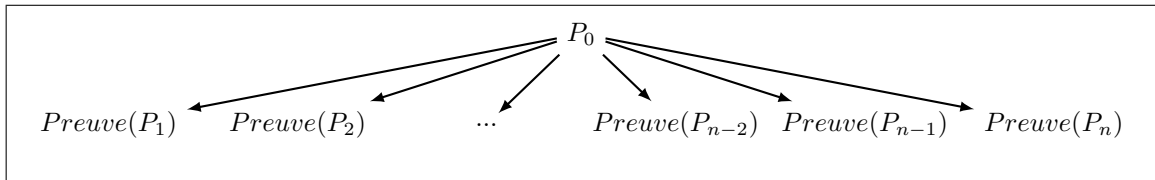
En négligeant les étapes de la réécriture Magic Set qui ne concerne pas les relations supplémentaire et en ne décrivant pas les termes des prédicats, pour une règle de la forme :

$$P_0 \leftarrow P_1, \dots, P_{n-1}, P_n$$

nous obtenons les règles suivantes :

$$\begin{aligned} sup_1 &\leftarrow sup_0, P_1 \\ &\dots \\ sup_{n-1} &\leftarrow sup_{n-2}, P_{n-1} \\ P_0 &\leftarrow sup_{n-1}, P_n \end{aligned}$$

La preuve d'une instance de P_0 dans le programme magique sera donc de la même forme que la figure 10. Pour obtenir la figure 11, il suffit donc de supprimer les faits issues des prédicats supplémentaires et rattacher les fils à P_0 . Cependant, il existe un cas particulier qui est celui des preuves des prédicats supplémentaires zéro. Le rattachement des fils ne doit pas s'effectuer dans ce cas car nous ne sommes pas intéressés par les preuves des faits magiques, ils agissent comme déclencheur lors de l'évaluation mais n'apportent rien d'un point de vue de la démonstration. C'est la raison pour laquelle nous supprimons toutes les preuves des faits issues des prédicats supplémentaires zéro.

FIGURE 10: Forme d'une preuve d'un fait originaire de \mathcal{P} dans le programme magiqueFIGURE 11: Forme d'une preuve d'un fait dans \mathcal{P}

5.2 Algorithmes et propriétés

Nous définissons donc un premier algorithme 2 qui prend un arbre binaire de preuve d'un fait issue d'un prédicat supplémentaire et qui aplatit cet arbre en liste de preuves de faits de sorte à n'avoir que les preuves des faits qui n'instancient pas des prédicats supplémentaires ou magiques. Le second algorithme 3 construit à partir d'une preuve dans un programme magique, la preuve originelle du programme en version originelle. La complexité spatiale et temporelle de cet algorithme est a priori dans le pire des cas équivalente à un simple parcours d'arbre.

Algorithme 2 : FlattenSupp

Données : A est un arbre de preuve binaire d'un fait supplémentaire

Résultat : Une liste de preuves de faits

si $Racine(A)$ correspond à $sup_{i,0}$ **alors**

retourner \square

sinon

$preuve \leftarrow [FilsDroit(A)]$

retourner $append(FlattenSupp(FilsGauche(A)), preuve)$

Conjecture 1 (Terminaison). *Unmagic s'arrête.*

Conjecture 2 (Validité). *Unmagic est valide.*

Propriété 3 (Complexité temporelle). *La complexité temporelle de Unmagic est :*

$$T(n) = O(n)$$

Propriété 4 (Complexité spatiale). *la complexité spatiale de Unmagic est :*

$$S(n) = \Theta(1)$$

Algorithme 3 : Unmagic

Données : \mathcal{P}_m : un programme magique,
 \mathcal{P} : le programme d'origine,
 A : un arbre de preuve binaire du programme \mathcal{P}_m

Hypothèse : A est un arbre de preuve d'un fait $\notin Sup \cup Magic$ **Résultat :** A' un arbre de preuve n-aire du programme \mathcal{P}

si A correspond à racine F **et** $F \in edb(\mathcal{P}_m)$ **alors**
 | **retourner** A

sinon si A correspond à $\langle racine F_{\alpha}, Fils \rangle$ **alors**
 | // Nous supprimons l'ornement ;
 | $racine \leftarrow F$;
 | $preuves \leftarrow []$;
 | **si** $Non\ Vide(Fils[1])$ **alors**
 | | $preuves \leftarrow Fils[1] :: preuves$
 | $preuves \leftarrow append(FlattenSupp(Fils[0]), preuves)$;
 | $preuvesNaire \leftarrow [Unmagic(p) \mid p \in preuves]$;
 | **retourner** $ArbreNaire(racine, preuvesNaire)$

Conclusion

Bilan

La finalité de ce projet tutoré est de fournir une bibliothèque compatible avec ACGtk, capable de transformer un programme Datalog en utilisant la réécriture Magic Set. Cette réécriture transforme le programme, ce qui a pour conséquence de modifier les preuves générées. C'est un problème dans la mesure où l'analyse syntaxique avec les ACG est isomorphe à la preuve d'une requête pour le programme Datalog non transformé. Un des objectifs est donc de proposer une transformation de ces arbres pour pouvoir utiliser la réécriture.

Pour mener à bien ce travail, d'autres objectifs ont découlés de cet objectif général. Tout d'abord, il a fallu assimiler le langage de programmation d'ACG (OCaml), puis assurer une compatibilité totale entre notre bibliothèque et le reste de l'outil, ce qui a demandé un travail d'adaptation par rapport à la démarche théorique. En effet, la réécriture Magic Set telle que présentée par Ullman 1989 est purement conceptuelle, nous avons donc transformé des descriptions de haut-niveau en algorithmes, puis en programmes.

Nous avons consacré beaucoup de ressources à l'étude approfondie de la littérature sur les différentes dimensions nécessaires de notre projet : les différentes réécritures Magic Set, le langage Datalog, les ACG et les TAG.

Enfin, le dernier objectif identifié est d'assurer la pérennité de ce projet : dans ce but, nous avons mis en place une documentation claire de cette bibliothèque et un rapport détaillé afin d'explicitier notre réflexion sur ce projet.

À la lumière de ces différents objectifs, nous avons fourni une librairie de réécriture fonctionnelle et correctement intégrée à ACGtk, ainsi qu'une documentation complète. Pour assurer l'efficacité de notre implémentation, nous avons également produit un comparatif automatisé de l'évaluation d'un programme réécrit par rapport à sa version originale et un programme de test de réécriture General Magic Set. Enfin, pour compléter l'intégration de la librairie dans ACGtk, nous avons défini une transformation des arbres de preuves, qui sera amené à être implémentée dans le future.

Perspectives d'amélioration

Pour avoir la certitude que la transformation des arbres de preuves proposée est valide, il reste à démontrer la véracité des deux conjonctures présenté dans le chapitre 5.

Dans le but de poursuivre les optimisations, nous avons identifié à l'aide de la littérature trois axes sur lequel il est possible de travailler pour potentiellement réduire le temps d'évaluation.

Ordonnancement des sous-buts des règles (Ullman et Vardi 1988) L'ordre des sous-buts dans les règles a un effet sur la performance pendant la phase d'évaluation ;

Évaluation ascendante parallèle (Ganguly, Silberschatz et Tsur 1992) Une version parallèle des algorithmes ascendants ;

Réécriture Subsumptive tabling (Tekle et Liu 2011) Une méthode de réécriture qui semble donner des performances meilleures que Magic set.

Bibliographie

- Abiteboul, Serge, Richard Hull et Victor Vianu (1995). *Foundations of Databases*. 1^{re} éd. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc. ISBN : 0201537710. URL : <http://webdam.inria.fr/Alice/pdfs/all.pdf>.
- Beeri, Catriel et Raghu Ramakrishnan (1991). « On the power of magic ». In : *The Journal of Logic Programming* 10.3. Special Issue : Database Logic Programming, p. 255–299. DOI : [10.1016/0743-1066\(91\)90038-Q](https://doi.org/10.1016/0743-1066(91)90038-Q).
- Ceri, Stefano, Georg Gottlob et Letizia Tanca (1989). « What you Always Wanted to Know About Datalog (And Never Dared to Ask) ». In : *IEEE Trans. Knowl. Data Eng.* 1, p. 146–166. DOI : [10.1109/69.43410](https://doi.org/10.1109/69.43410).
- Conchon, Sylvain, Jean-Christophe Filliâtre et Julien Signoles (2007). « Designing a Generic Graph Library Using ML Functors. » In : *Trends in Functional Programming 2007*, p. 124–140. URL : <http://www.lri.fr/~filliatr/ftp/publis/ocamlgraph.ps>.
- de Groote, Philippe (2001). « Towards Abstract Categorical Grammars ». In : *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics*, p. 148–155. Anthologie ACL : [P01-1033](https://aclanthology.org/P01-1033).
- Ganguly, Sumit, Avi Silberschatz et Shalom Tsur (1992). « Parallel Bottom-up Processing of Datalog Queries ». In : *J. Log. Program.* 14.1-2, p. 101–126. ISSN : 0743-1066. DOI : [10.1016/0743-1066\(92\)90048-8](https://doi.org/10.1016/0743-1066(92)90048-8).
- Joshi, Aravind K. et Yves Schabes (1997). « Tree-adjoining grammars ». In : *Handbook of formal languages*. Sous la dir. de Grzegorz Rozenberg et Arto K. Salomaa. T. 3. Springer. Chap. 2. DOI : [10.1007/978-3-642-59126-6_2](https://doi.org/10.1007/978-3-642-59126-6_2).
- Kanazawa, Makoto (2007). « Parsing and Generation as Datalog Queries ». In : *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL 2007)*. Prague, Czech Republic : Association for Computational Linguistics, p. 176–183. Anthologie ACL : [P07-1023](https://aclanthology.org/P07-1023).
- Pogodalla, Sylvain (2016). « ACGTK : un outil de développement et de test pour les grammaires catégorielles abstraites ». In : *Actes de la 23ème Conférence sur le Traitement Automatique des Langues Naturelles, 31ème Journées d’Études sur la Parole, 18ème Rencontre des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (JEP-TALN-RECITAL 2016)*. Sous la dir. de Laurence Danlos et Thierry Hamon. Démonstration. Association pour le Traitement Automatique des Langues, Association Francophone pour la Communication Parlée. HAL archive ouverte : [hal-01328702](https://hal.archives-ouvertes.fr/hal-01328702).
- Tekle, K Tuncay et Yanhong A Liu (2011). « More efficient datalog queries : subsumptive tabling beats magic sets ». In : *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, p. 661–672. DOI : [10.1145/1989323.1989393](https://doi.org/10.1145/1989323.1989393).
- Ullman, Jeffrey D. (1983). *Principles of Database and Knowledge-Base Systems : Volume I*. 2nd. New York, NY, USA : W. H. Freeman & Co. ISBN : 0716780690.
- Ullman, Jeffrey D. (1989). « Bottom-up Beats Top-down for Datalog ». In : *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS ’89. Philadelphia, Pennsylvania, USA : ACM, p. 140–149. ISBN : 0-89791-308-6. DOI : [10.1145/73721.73736](https://doi.org/10.1145/73721.73736).
- Ullman, Jeffrey D. (1990). *Principles of Database and Knowledge-Base Systems : Volume II : The New Technologies*. New York, NY, USA : W. H. Freeman & Co. ISBN : 071678162X.
- Ullman, Jeffrey D. et Moshe Y. Vardi (1988). « The Complexity of Ordering Subgoals ». In : *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS ’88. Austin, Texas, USA : ACM, p. 74–81. ISBN : 0-89791-263-2. DOI : [10.1145/308386.308417](https://doi.org/10.1145/308386.308417).

Annexes

Exemple de réécriture magique

Nous allons ici vous montrer un exemple de réécriture pour $\langle \mathcal{P}, q \rangle$ avec \mathcal{P} :

$$S(i, l) \leftarrow a(i, j), S(j, k), b(k, l) \quad (1)$$

$$S(i, i) \leftarrow \quad (2)$$

et q :

$$S_{bb}(0, 4)$$

On applique la première partie de la réécriture avec :

— Le Rule/Goal Graph présenté en figure 4;

— La mise en place du "Unique Binding Pattern".

Voici le programme Datalog \mathcal{P} sur lequel nous allons appliquer les réécritures Magic Set (noté "classique") et General Magic Set (noté "General") :

$$S_{bb}(i, l) \leftarrow a(i, j), S_{bf}(j, k), b(k, l) \quad (1)$$

$$S_{bf}(i, l) \leftarrow a(i, j), S_{bf}(j, k), b(k, l) \quad (2)$$

$$S_{bb}(i, i) \leftarrow \quad (3)$$

$$S_{bf}(i, i) \leftarrow \quad (4)$$

Création des prédicats magiques et supplémentaires

Maintenant que nous avons notre programme Datalog \mathcal{P} pré-traité, nous pouvons commencer la réécriture, avec la création des prédicats magiques et supplémentaires. Nous n'allons pas détailler les étapes de création, mais nous allons lister tous les prédicats créés par la première étape de la réécriture magique.

Règle traitée	prédicat magique ("Classique")	prédicat magique ("General")
(1)	$magic_S_bf(j)$	$magic_S_bf(j, k)$
(2)	$magic_S_bf(j)$	$magic_S_bf(j, k)$

Règle traitée, position traitée	prédicat supp ("Classique")	prédicat supp ("General")
(1), position 0	$sup_{1.0}(i, l)$	$sup_{1.0}(i, l)$
(1), position 1	$sup_{1.1}(i, l, j)$	$sup_{1.1}(i, l, j)$
(1), position 2	$sup_{1.2}(i, l, k)$	$sup_{1.2}(i, l, k)$
(2), position 0	$sup_{2.0}(i)$	$sup_{2.0}(i, l)$
(2), position 1	$sup_{2.1}(i, j)$	$sup_{2.1}(i, l, j)$
(2), position 2	$sup_{2.2}(i, k)$	$sup_{2.2}(i, l, k)$
(3), position 0	$sup_{3.0}(i, i)$	$sup_{3.0}(i, i)$
(4), position 0	$sup_{4.0}(i)$	$sup_{4.0}(i, i)$

Maintenant que nous avons créé nos prédicats magiques et supplémentaires, nous allons créer les règles magiques à partir des prédicats magiques.

Création des règles magiques

règle d'origine	prédicat magique ("Classique")	règle magique associée
(1)	$magic_S_bf(j)$	$magic_S_bf(j) \leftarrow sup_{1.1}(i, l, j)$
(2)	$magic_S_bf(j)$	$magic_S_bf(j) \leftarrow sup_{2.1}(i, j)$

règle d'origine	prédicat magique ("General")	règle magique associée
(1)	$magic_S_bf(j, k)$	$magic_S_bf(j, k) \leftarrow sup_{1.1}(i, l, j)$
(2)	$magic_S_bf(j, k)$	$magic_S_bf(j, k) \leftarrow sup_{2.1}(i, l, j)$

Voici le programme Datalog \mathcal{P} avec les règles magiques en version "Classique" :

$$\begin{aligned}
 S_bb(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (1) \\
 magic_S_bf(j) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
 S_bf(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (2) \\
 magic_S_bf(j) &\leftarrow sup_{2.1}(i, j) & (m2) \\
 S_bb(i, i) &\leftarrow & (3) \\
 S_bf(i, i) &\leftarrow & (4)
 \end{aligned}$$

Et le programme Datalog \mathcal{P} avec les règles magiques en version "General" :

$$\begin{aligned}
 S_bb(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (1) \\
 magic_S_bf(j, k) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
 S_bf(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (2) \\
 magic_S_bf(j, k) &\leftarrow sup_{2.1}(i, l, j) & (m2) \\
 S_bb(i, i) &\leftarrow & (3) \\
 S_bf(i, i) &\leftarrow & (4)
 \end{aligned}$$

Création des règles supplémentaires

Ici, nous allons créer les règles supplémentaires, à partir des prédicats supplémentaires précédemment générés.

règle d'origine	prédicat supp traité ("classique")	règle supp associée
(1)	$sup_{1.0}(i, l)$	$sup_{1.0}(i, l) \leftarrow magic_S_bb(i, l)$
(1)	$sup_{1.1}(i, l, j)$	$sup_{1.1}(i, l, j) \leftarrow sup_{1.0}(i, l), a(i, j)$
(1)	$sup_{1.2}(i, l, k)$	$sup_{1.2}(i, l, k) \leftarrow sup_{1.1}(i, l, j), S_bf(j, k)$
(2)	$sup_{2.0}(i)$	$sup_{2.0}(i) \leftarrow magic_S_bf(i)$
(2)	$sup_{2.1}(i, j)$	$sup_{2.1}(i, j) \leftarrow sup_{2.0}(i), a(i, j)$
(2)	$sup_{2.2}(i, k)$	$sup_{2.2}(i, k) \leftarrow sup_{2.1}(i, j), S_bf(j, k)$
(3)	$sup_{3.0}(i, i)$	$sup_{3.0}(i, i) \leftarrow magic_S_bb(i, i)$
(4)	$sup_{4.0}(i)$	$sup_{4.0}(i) \leftarrow magic_S_bf(i)$

règle d'origine	prédicat supp traité ("General")	règle supp associée
(1)	$sup_{1.0}(i, l)$	$sup_{1.0}(i, l) \leftarrow magic_S_bb(i, l)$
(1)	$sup_{1.1}(i, l, j)$	$sup_{1.1}(i, l, j) \leftarrow sup_{1.0}(i, l), a(i, j)$
(1)	$sup_{1.2}(i, l, k)$	$sup_{1.2}(i, l, k) \leftarrow sup_{1.1}(i, l, j), S_bf(j, k)$
(2)	$sup_{2.0}(i, l)$	$sup_{2.0}(i, l) \leftarrow magic_S_bf(i, l)$
(2)	$sup_{2.1}(i, l, j)$	$sup_{2.1}(i, l, j) \leftarrow sup_{2.0}(i, l), a(i, j)$
(2)	$sup_{2.2}(i, l, k)$	$sup_{2.2}(i, l, k) \leftarrow sup_{2.1}(i, l, j), S_bf(j, k)$
(3)	$sup_{3.0}(i, i)$	$sup_{3.0}(i, i) \leftarrow magic_S_bb(i, i)$
(4)	$sup_{4.0}(i, i)$	$sup_{4.0}(i, i) \leftarrow magic_S_bb(i, i)$

Voici le programme Datalog \mathcal{P} avec les règles magiques et les règles supplémentaires en version "Classique" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bf(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (2) \\
sup_{2.0}(i) &\leftarrow magic_S_bf(i) & (s2.0) \\
sup_{2.1}(i, j) &\leftarrow sup_{2.0}(i), a(i, j) & (s2.1) \\
sup_{2.2}(i, k) &\leftarrow sup_{2.1}(i, j), S_bf(j, k) & (s2.2) \\
magic_S_bf(j) &\leftarrow sup_{2.1}(i, j) & (m2) \\
S_bb(i, i) &\leftarrow & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow & (4) \\
sup_{4.0}(i) &\leftarrow magic_S_bf(i) & (s4.0)
\end{aligned}$$

Et le programme Datalog \mathcal{P} avec les règles magiques et supplémentaires en version "General" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j, k) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bf(i, l) &\leftarrow a(i, j), S_bf(j, k), b(k, l) & (2) \\
sup_{2.0}(i, l) &\leftarrow magic_S_bf(i, l) & (s2.0) \\
sup_{2.1}(i, l, j) &\leftarrow sup_{2.0}(i, l), a(i, j) & (s2.1) \\
sup_{2.2}(i, l, k) &\leftarrow sup_{2.1}(i, l, j), S_bf(j, k) & (s2.2) \\
magic_S_bf(j, k) &\leftarrow sup_{2.1}(i, l, j) & (m2) \\
S_bb(i, i) &\leftarrow & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow & (4) \\
sup_{4.0}(i, i) &\leftarrow magic_S_bf(i, i) & (s4.0)
\end{aligned}$$

Transformation des règles initiales

Maintenant que nous avons créé nos règles magiques et supplémentaires, nous allons transformer les règles initialement présentes dans le programme Datalog \mathcal{P} :

règle initiale ("classique")	règle transformée
(1)	$S_bb(i, l) \leftarrow sup_{1.2}(i, l, k), b(k, l)$
(2)	$S_bf(i, l) \leftarrow sup_{2.2}(i, k), b(k, l)$
(3)	$S_bb(i, i) \leftarrow sup_{3.0}(i, i)$
(4)	$S_bf(i, i) \leftarrow sup_{4.0}(i)$
règle initiale ("General")	règle transformée
(1)	$S_bb(i, l) \leftarrow sup_{1.2}(i, l, k), b(k, l)$
(2)	$S_bb(i, l) \leftarrow sup_{1.2}(i, l, k), b(k, l)$
(3)	$S_bb(i, i) \leftarrow sup_{3.0}(i, i)$
(4)	$S_bf(i, i) \leftarrow sup_{4.0}(i, i)$

Voici le programme Datalog \mathcal{P} avec les règles magiques, les règles supplémentaires et les règles initiales transformées en version "Classique" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bf(i, l) &\leftarrow sup_{2.2}(i, k), b(k, l) & (2) \\
sup_{2.0}(i) &\leftarrow magic_S_bf(i) & (s2.0) \\
sup_{2.1}(i, j) &\leftarrow sup_{2.0}(i), a(i, j) & (s2.1) \\
sup_{2.2}(i, k) &\leftarrow sup_{2.1}(i, j), S_bf(j, k) & (s2.2) \\
magic_S_bf(j) &\leftarrow sup_{2.1}(i, j) & (m2) \\
S_bb(i, i) &\leftarrow sup_{3.0}(i, i) & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow sup_{4.0}(i, i) & (4) \\
sup_{4.0}(i) &\leftarrow magic_S_bf(i) & (s4.0)
\end{aligned}$$

Et le programme Datalog \mathcal{P} avec les règles magiques, les règles supplémentaires et les règles initiales transformées en version "General" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j, k) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (2) \\
sup_{2.0}(i, l) &\leftarrow magic_S_bf(i, l) & (s2.0) \\
sup_{2.1}(i, l, j) &\leftarrow sup_{2.0}(i, l), a(i, j) & (s2.1) \\
sup_{2.2}(i, l, k) &\leftarrow sup_{2.1}(i, l, j), _bf(j, k) & (s2.2) \\
magic_S_bf(j, k) &\leftarrow sup_{2.1}(i, l, j) & (m2) \\
S_bb(i, i) &\leftarrow sup_{3.0}(i, i) & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow sup_{4.0}(i, i) & (4) \\
sup_{4.0}(i) &\leftarrow magic_S_bf(i, i) & (s4.0)
\end{aligned}$$

Création de la règle d'initialisation

On utilise la requête q pour générer la règle d'initialisation associée à cette réécriture :

requête initiale	règle d'initialisation associée
$S_bb(0, 4)$	$magic_S_bb(0, 4)$

Voici le programme Datalog \mathcal{P} et la requête q après la réécriture magique "Classique" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bf(i, l) &\leftarrow sup_{2.2}(i, k), b(k, l) & (2) \\
sup_{2.0}(i) &\leftarrow magic_S_bf(i) & (s2.0) \\
sup_{2.1}(i, j) &\leftarrow sup_{2.0}(i), a(i, j) & (s2.1) \\
sup_{2.2}(i, k) &\leftarrow sup_{2.1}(i, j), S_bf(j, k) & (s2.2) \\
magic_S_bf(j) &\leftarrow sup_{2.1}(i, j) & (m2) \\
S_bb(i, i) &\leftarrow sup_{3.0}(i, i) & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow sup_{4.0}(i, i) & (4) \\
sup_{4.0}(i) &\leftarrow magic_S_bf(i) & (s4.0) \\
magic_S_bb(0, 4) &\leftarrow & (q)
\end{aligned}$$

Et le programme Datalog \mathcal{P} et la requête q après la réécriture magique "General" :

$$\begin{aligned}
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (1) \\
sup_{1.0}(i, l) &\leftarrow magic_S_bb(i, l) & (s1.0) \\
sup_{1.1}(i, l, j) &\leftarrow sup_{1.0}(i, l), a(i, j) & (s1.1) \\
sup_{1.2}(i, l, k) &\leftarrow sup_{1.1}(i, l, j), S_bf(j, k) & (1.2) \\
magic_S_bf(j, k) &\leftarrow sup_{1.1}(i, l, j) & (m1) \\
S_bb(i, l) &\leftarrow sup_{1.2}(i, l, k), b(k, l) & (2) \\
sup_{2.0}(i, l) &\leftarrow magic_S_bf(i, l) & (s2.0) \\
sup_{2.1}(i, l, j) &\leftarrow sup_{2.0}(i, l), a(i, j) & (s2.1) \\
sup_{2.2}(i, l, k) &\leftarrow sup_{2.1}(i, l, j), _bf(j, k) & (s2.2) \\
magic_S_bf(j, k) &\leftarrow sup_{2.1}(i, l, j) & (m2) \\
S_bb(i, i) &\leftarrow sup_{3.0}(i, i) & (3) \\
sup_{3.0}(i, i) &\leftarrow magic_S_bb(i, i) & (s3.0) \\
S_bf(i, i) &\leftarrow sup_{4.0}(i, i) & (4) \\
sup_{4.0}(i) &\leftarrow magic_S_bf(i, i) & (s4.0) \\
magic_S_bb(0, 4) &\leftarrow & (q)
\end{aligned}$$