U.F.R.
Sciences

Laboratoire d'Informatique Fondamentale d'Orléans
FRE CNRS 2490

Rapport de recherche RR-2005-07

MEMOIRE
Représentation des langages de n-uplets d'arbres par des programmes logiques et applications

présenté dans le cadre de l'école doctorale Sciences et Technologies pour l'obtention

de l'Habilitation à Diriger des Recherches
Université d'Orléans

Sébastien Limet

présenté et soutenu publiquement le 28 novembre 2005

devant le jury composé de

Gaétan Hains, Professeur à l'Université d'Orléans
Nicolas Hermann, Chargé de Recherche CNRS, Ecole Polytechnique
Hélène Kirchner, Directrice de Recherche CNRS au LORIA Nancy
Gernot Salzer, Professeur à l'Université Technique de Vienne
Frédéric Saubion, Professeur à l'Université d'Angers
Sophie Tison, Professeur à l'Université des Sciences et Technologies de Lille

Remerciements

A Benoît, à Philippe

Je tiens tout d'abord à remercier Miki Hermann, Sophie Tison et Moshe Vardi, les rapporteurs du présent document, ainsi que tous les membres du jury d'avoir accepté de prendre de leur temps précieux pour s'intéresser à mes travaux.

Je remercie tous les chercheurs avec qui j'ai eu l'honneur de travailler. Les discussions que nous avons eues, m'ont permis d'améliorer et de faire évoluer mes travaux de recherche.

Je tiens aussi à remercier mes collègues du LIFO et du département Informatique de l'IUT d'Orléans pour leur disponibilité, leurs conseils et leur soutien. Je remercie les personnels IATOSS du LIFO et de l'IUT qui m'ont permis de travailler dans les meilleurs conditions possibles.

Je remercie plus particulièrement Emmanuel, Valérie et Gérard pour leur soutien précieux dans des moments particulièrement difficiles.

Enfin, je dédie mon travail à toute ma petite famille : Agnès, Marianne, Fanny et Noémie. Je les remercie pour leur patience et leur compréhension lorsque mon travail empiète sur la vie de famille.

Résumé

Ce document présente différents outils pour représenter et manipuler des ensembles infinis de n-uplets d'arbres appelés langages de n-uplets d'arbres. Ces outils sont très utiles dans la plupart des domaines de l'informatique théorique où l'on raisonne sur des relations sur les termes du 1^{er} ordre, à condition de pouvoir effectivement calculer les opérations ensemblistes sur ces relations et de pouvoir tester si un n-uplet appartient à la relation ou non d'une part et savoir si la relation est vide ou non.

Les automates et les grammaires d'arbres permettent de représenter des ensembles de termes et ont les propriétés requises. Des extensions naturelles de ces outils aux langages de n-uplets d'arbres, préservant au moins en partie les propriétés de clôture par les opérations ensemblistes et ayant des tests d'appartenance et du vide décidables ont été étudiées ce qui a permis d'obtenir de nombreux résultats dans différents domaines de l'informatique théorique. Par contre l'utilisation des ces outils est assez complexe car leur formalisme n'est pas très intuitif d'une part et reste de bas niveau ce qui rend la phase de transcription du problème initial en termes de langages de n-uplets d'arbres ardue en général. Par ailleurs, la programmation logique est un outil de haut niveau, très largement répandu en informatique, qui permet de représenter n'importe quelle relation récursivement énumérable sur les termes. Cette puissance d'expression ne permet pas de garantir des tests du vide et d'apartenance décidables qui sont essentiels pour la plupart des applications des langages de n-uplet d'arbres.

Nous présentons tout d'abord la classe des langages synchronisés qui est une classe de relations pour laquelle on sait tester le vide et l'appartenance, en termes de grammaires synchronisées et de systèmes de contraintes. Nous étudions ensuite une classe de programmes logiques, appelés cs-programmes, qui permet de représenter les langages synchronisés et nous donnons un algorithme à base de transformations de programmations qui permet de calculer des différentes opérations sur ces langages. Grâce à cette procédure, nous décrivons différentes classes de programmes logiques, qui correspondent à des classes de langages de n-unplets d'arbres connues.

Nous montrons ensuite comment ces résultats peuvent s'appliquer à des problèmes liés à la réécriture de termes. Ces résultats se basent sur une procédure de construction d'un programme logique qui décrit la clôture réflexive transitive de la relation induite par un système de réécriture. Grâce à cette transformation nous pouvons utiliser les propriétés des cs-programmes pour donner de nouveaux résultats en réécriture concernant les ensembles de termes accessibles par réécriture à partir d'un ensemble de termes initial. Ces résultats sont ensuite appliqués à la vérification de propriétés de sûreté de systèmes à états infinis, d'une part, et d'autre part pour permettre de résoudre les formules pseudo-régulières du premier ordre.

Mots clés : Langages d'arbres, programmation logique, réécriture, vérification formelle

Table des matières

Chapitre 1

Introduction

Ce document est la synthèse d'une partie des travaux que j'ai effectués au sein de l'équipe Vérification, Parallèlisme et Sécurité du Laboratoire d'Informatique Fondamentale d'Orléans. Ces travaux portent sur l'axe Vérification et sont le fruit de collaborations locales, nationales et internationales. Ils ont tous été publiés dans les actes de conférences d'audience internationale ou dans des revues internationales. Leurs derniers développements sont l'objet de la thèse de Pierre Pillot qui a débuté en Septembre 2004 et que j'encadre en totalité.

Les travaux qui vont être présentés, trouvent leurs fondements dans les domaines de l'informatique théorique comme la logique, la démonstration automatique ou la vérification de systèmes à états infinis. En effet dans ces domaines, la structure de données manipulée est le plus souvent la structure de termes. Les termes sont en quelque sorte à ces domaines ce que les entiers et les réels sont au calcul numérique. Les *arbres* que nous manipulons dans ce document, sont en fait les termes finis du premier ordre.

J'ai commencé à m'intéresser à la représentation des ensembles infinis d'arbres, aussi appelés *langages d'arbres*, lors de ma thèse où je cherchais à représenter les ensembles de solutions de problèmes de R-unification. La R-unification consiste à rechercher les instances de deux termes qui sont égales modulo le système de réécriture R. Ce type d'unification est utilisé par exemple comme mécanisme opérationnel des langages de programmation logico-fonctionnels comme Babel [49] ou Curry [3] ou pour faire de la vérification de protocole cryptographique [46, 71, 72]. En règle générale, l'ensemble des solutions d'un problème de R-unification peut-être infini et surtout le mécanisme pour retrouver les solutions est semi-décidable et donc peut boucler à l'infini. Il peut donc être intéressant dans certains cas de trouver une représention finie des solutions afin, par exemple, d'éviter des boucles infinies ou pour faire de l'analyse statique du problème. L'ensemble des solutions d'un problème de R-unification peut être vu comme un langage de n-uplets d'arbres où chaque composant du n-uplet correspond à une instance d'une variable.

En fait, la question de la représentation finie d'ensembles infinis d'arbres se pose dans de nombreux domaines de l'informatique dès lors qu'on a besoin de travailler sur les modèles ou que l'on veut éviter des calculs infinis (par exemple en déduction automatique avec la construction de modèles [29, 53] ou en analyse de programmes pour définir des domaines abstraits [32]). Pour pouvoir représenter et manipuler des langages d'arbres, il faut des outils qui permettent de calculer des opérations sur ces ensembles comme l'union, l'intersection ou le complément, et surtout qui permettent de faire le test d'appartenance d'un arbre à l'ensemble et le test qui indique si le langage représenté est vide ou non (appelé test du vide). Donc une difficulté est de trouver un bon compromis entre l'expressivité de l'outil choisi et la possibilité de calculer effectivement les opérations et de tester les langages obtenus. Les automates d'arbres sont l'outil privilégié pour représenter des ensembles infinis d'arbres. Ces automates, introduits dans les années 1950 pour faire de la vérification de circuits, sont une extension des automates finis à la structure d'arbres et ont toutes les qualités requises en terme de clôture par les opérations ensemblistes et leurs tests du vide et d'appartenance sont décidables. Ces automates ont été étendus pour permettre de représenter des ensembles de n-uplets d'arbres par des techniques de fusion de termes. Les langages ainsi reconnus sont appelés relations régulières. Les relations régulières préservent toutes les propriétés des langages reconnus par les automates d'arbres, elles sont en plus closes pour la plupart des opérations propres aux relations (comme la composition ou la projection par exemple). Les deux principaux inconvénients de cette classe sont son manque d'expressivité d'une part et son formalisme à base de termes mélangés d'autre part qui rend la transcription du problème initial vers les automates d'arbres assez ardu en général. Des extensions d'automates d'arbres ont été étudiées afin d'améliorer leur expressivité, en ajoutant notamment des contraintes sur l'application des transitions [9, 10].

Lors de ma thèse, j'ai introduit un outil appelé *grammaires synchronisées* [55], pour permettre de représenter des ensembles infinis de solutions de la R-unification. Les langages définis par ces grammaires s'appellent *langages synchronisés*. Lors de ce travail, il est apparu trois types de problèmes. Le premier concerne la représentation des ensembles infinis de n-uplets de termes ; en effet les grammaires synchronisées nous ont permis de montrer un certain nombre de résultats dans le domaine de la réécriture mais le formalisme a montré des faiblesses pour le calcul des opérations ensemblistes. Le second problème concerne l'utilisation de ce type d'outils, c'est à dire la transcription du problème initial en terme de grammaires synchronisées, n'est pas aisée du fait du caractère relativement basique de cet outil. Enfin l'implantation des algorithmes de manipulation [52, 85] de ces langages s'est avérée très difficile.

A partir de ces constats, la suite de mes travaux a consisté à rechercher des formalismes plus adéquats pour la représentation des ensembles infinis de n-uplets d'arbres d'une part et à diversifier leurs applications en essayant de fournir des mécanismes généraux de transformation d'un problème comme la R-unification en terme d'ensembles de n-uplets d'arbres.

Le but des grammaires synchronisées était de proposer un formalisme plus expressif que les relations régulières, quitte à perdre quelques propriétés de clôture par les opérations ensemblistes mais en gardant des tests d'appartenance et du vide décidables. D'autres formalismes dont le but est d'étendre l'expressivité des relations régulières, ont été étudiés. On peut citer par exemple [25, 79, 87] mais ils restent assez difficiles à manipuler. Dans le cadre d'une collaboration avec H. Seidl, nous avons introduit [40, 59] une représentation des langages synchronisés, basée sur des contraintes ensemblistes, qui résout un certain nombre de problèmes liés au formalisme des grammaires synchronisées et qui a permis d'exhiber des sous-classes intéressantes des langages synchronisés qui sont closes par intersection.

Malgré ces progrès, les systèmes de contraintes restent assez peu intuitifs à utiliser et c'est pour cela que nous avons cherché à améliorer encore la représentation des langages synchronisés. Finalement, en travaillant sur cette question en collaboration avec G. Salzer, nous sommes arrivés à la conclusion que l'outil le plus adapté et le plus standard pour représenter des langages de n-uplets d'arbres est la programmation logique. Le principal inconvénient de ce formalisme est sa trop grande expressivité qui rend notamment le test du vide indécidable. Nous nous sommes donc attachés à étudier la classe de programmes logiques, appelés *cs-programmes* [60, 63], qui correspondent aux langages synchronisés. Nous avons également étudié certaines de leurs sous-classes. Il nous a fallu aussi définir un algorithme pour calculer effectivement les opérations ensemblistes et faire les tests d'appartenance et du vide. Cet algorithme se base sur des techniques bien connues de transformations de programmes logiques [78, 90].

Parallèlement à ces travaux sur la représentation des langages de n-uplets d'arbres, j'ai cherché à appliquer ces langages dans différents domaines de l'informatique théorique. Pour la plupart de ces applications, je me suis efforcé de définir des algorithmes suffisamment généraux pour qu'ils puissent être utilisés avec d'autres classes de langages de n-uplets d'arbres que les langages synchronisés. Un premier travail dans ce sens [65], en collaboration avec F. Saubion a consisté à généraliser l'algorithme de R-unification de [55] afin de s'abstraire de la classe de langages utilisée pour représenter les ensembles de solutions. Cela nous a permis de montrer que l'on pouvait l'appliquer avec les grammaires primales de [43, 44] par exemple. Nous avons aussi appliqué nos résultats sur les langages d'arbres à la validation partielle de programmes logiques [64]. Dans le domaine de la réécriture, en collaboration avec P. Réty nous avons étudié l'utilisation des grammaires synchronisées pour des problèmes de dis-unification [57] et pour la théorie de la réécriture en une étape [58]. Avec G. Salzer, nous avons étudié le problème de la préservation de la régularité des systèmes de réécriture [61], ces travaux nous ont ammené à définir une procédure générale de transformation d'un système de réécriture en un programme logique qui code la relation de réécriture. Nous avons aussi montré [62] que nos résultats pouvaient servir dans le domaine de la vérification de systèmes infinis. Enfin, dans le cadre de la thèse de P. Pillot, nous avons étudié la résolution de formules du premier ordre d'une théorie dont l'unique prédicat concerne la joignabilité modulo un système de réécriture [54], ces travaux reprennent les principes de [65] pour la résolution des équations et de [61] pour la transcription du problème en termes d'opérations sur les langages de n-uplets d'arbres.

Dans la suite de ce document, nous revenons plus en détail sur la plupart des points cités ci-dessus en insistant sur les derniers développements de cette recherche. Le chapitre 2 fixe les notions concernant les termes, la programmation logique, la réécriture et les langages d'arbres et de n-uplets d'arbres. Le chapitre 3 présente les trois outils que nous avons définis pour représenter les langages synchronisés, c'est à dire les grammaires synchronisées, les systèmes de contraintes et les programmes logiques. Ce chapitre décrit aussi nos règles de transformation de programmes logiques ainsi que les cas dans lesquels la transformation aboutie à un cs-programme. Le chapitre 4 présente la procédure qui permet de transformer un système de réécriture en un programme logique puis montre comment combiner ce résultat avec ceux obtenus sur les programmes logiques représentant des langages d'arbres afin d'obtenir de nouveaux résultats ou d'en étendre d'autres dans les domaines de la réécriture et de la vérification. Enfin le chapitre 5 donne des perspectives pour des travaux de recherches futurs basés sur les résultats décrits dans ce document.

Chapitre 2

Notations et définitions utilisées

Dans ce chapitre, nous introduisons les définitions et notations utilisées dans ce mémoire. La plupart notions de ce chapitre sont standard. Plus de détails sur les automates et grammaires d'arbres peuvent être trouvés dans [21, 83], en ce qui concerne la réécriture vous pouvez consulter [5] et enfin [66] pour les bases de la programmation logique.

2.1 Arbres et Termes

Dans le contexte de ce mémoire, les arbres sont des termes du 1$^{\text{er}}$ ordre définis comme suit.

Soit Σ un ensemble fini de *symboles avec arité*, Var un ensemble infini de *variables*, et $\mathcal{T}(\Sigma, \mathit{Var})$ l'*algèbre de termes* du 1$^{\text{er}}$ ordre sur Σ et Var. Un terme est *linéaire* si aucune variable n'apparaît plus d'une fois dans celui-ci ; il est *clos* s'il ne contient pas de variables. $\mathit{Var}(t)$ est l'ensemble des variables de t.

Σ est constituée de deux sous-ensembles disjoints : l'ensemble $\mathcal{F}$ des *symboles de fonction* et l'ensemble $\mathcal{P}r$ des *symboles de prédicat*. Parfois l'ensemble $\mathcal{F}$ est lui-même partitionné en $\mathcal{D}$ l'ensemble des *fonctions définies* et $\mathcal{C}$ l'ensemble des *constructeurs*. Les termes de la forme $P(\vec{t})$ où P est un symbole de prédicat d'arité n et $\vec{t}$ un vecteur de $\mathcal{T}(\mathcal{F}, \mathit{Var})^n$ sont appelés des *atomes*. Les termes construits sur $\mathcal{T}(\mathcal{C}, \mathit{Var})$ sont appelés *termes sur les constructeurs* ou *data-termes*. Un terme de la forme $f(x_1, \ldots, x_n)$ où f est un symbole d'arité n est dit *plat*, si $f \in \mathcal{P}r$ on parle d'*atome plat*.

De manière standard, nous identifions les *positions* dans les termes par des chaînes d'entiers positifs. ε représente la chaîne vide, $u.v$ est la concaténation des chaînes u et v, et $<$ est l'ordre préfixe sur les chaînes, c'est à dire $u < v$ si et seulement si $v = u.w$ pour une chaîne w. Deux positions sont incomparable si aucune n'est le préfixe de l'autre. Pour un ensemble V de positions, on définit $u.V = \{\, u.v \mid v \in V \,\}$. Pour un terme t, $\mathit{Pos}(t)$ est l'ensemble des positions de t, $|t| = |\mathit{Pos}(t)|$ la *taille* de t, et $t|_u$ le *sous-terme* de t à la position u. Le terme $t[u \leftarrow s]$ est obtenu à partir de t en substituant s au sous-terme à la position u de t. De même, si U est un ensemble de positions incomparables deux à deux, $t[U \leftarrow s]$ est obtenu en substituant simultanément par s tous les sous-termes aux positions $u \in U$ dans t. L'ensemble $\Sigma\mathit{Pos}(t) \subseteq \mathit{Pos}(t)$ est l'ensemble des positions non variable de t, c'est à dire $t|_u \notin \mathit{Var}$ pour tout $u \in \Sigma\mathit{Pos}(t)$ et $t|_u \in \mathit{Var}$ pour tout $u \in \mathit{Pos}(t) \setminus \Sigma\mathit{Pos}(t)$. La *profondeur d'un terme* t de $\mathcal{T}(\mathcal{F}, \mathit{Var})$, noté $\mathit{Depth}(t)$, est 0 si $t \in \mathit{Var}$ et $\max(\{\, |p| \mid p \in \Sigma\mathit{Pos}(t) \,\})$ sinon. La *profondeur d'un atome* $P(t_1, \ldots, t_n)$, noté aussi $\mathit{Depth}(P(t_1, \ldots, t_n))$, est $\max(\{\, \mathit{Depth}(t_i) \mid 1 \leq i \leq n \,\})$.

Une *substitution* est une application de Var vers $\mathcal{T}(\Sigma, \mathit{Var})$ qui s'étend naturellement à une application $\mathcal{T}(\Sigma, \mathit{Var})$ vers $\mathcal{T}(\Sigma, \mathit{Var})$. Le *domaine* d'une substitution σ, noté $\mathit{Dom}(\sigma)$, est l'ensemble $\{\, x \in \mathit{Var} \mid x\sigma \neq x \,\}$. Pour $V \subseteq \mathit{Var}$, $\sigma|_V$ représente la *restriction de σ aux variables de V*, c'est à dire, $x\sigma|_V = x\sigma$ pour $x \in V$ et $x\sigma|_V = x$ sinon. Un terme s *subsume* un terme t si $s\sigma = t$ pour une substitution σ. Une substitution σ est un *renommage de variables* pour un ensemble de termes $\mathcal{R}$, s'il existe une substitution σ^{-1} telle que $\mathcal{R}\sigma\sigma^{-1} = \mathcal{R}$. Une substitution σ telle que

$x\sigma \in \mathcal{T}(\Sigma)$ pour tout $x \in Dom(\sigma)$ est appelée *substitution close*. Une substitution σ telle que $x\sigma \in \mathcal{T}(\mathcal{C}, Var)$ pour tout $x \in Var$ est appelée une *data-substitution*. Une substitution μ telle que $\forall x \in Var\ x\mu$ est linéaire et $Var(x\mu) \cap Var(y\mu) = \emptyset$ si $y \in Var$ et $x \neq y$, est dite *linéaire*.

Deux termes s et t sont dits *unifiables* s'il existe une substitution σ telle que $s\sigma = t\sigma$. σ est un *unificateur* de s et t. Lorsque deux termes s et t sont unifiables, il existe un *unificateur μ le plus général* (noté $mgu(s,t)$) unique à un renommage près tel pour tout unificateur σ de s et t il existe μ' et $\sigma = \mu \circ \mu'$.

Soit $CVar = \{ \square_i \mid i \geq 1 \}$ un ensemble de variables de contexte disjoint de Var. Un *n-contexte* est un terme t de $T(\mathcal{F}, Var \cup CVar)$ tel que $\square_1, \ldots, \square_n$ sont les seules variables de contexte de t et chaque variable de contexte apparaît exactement une fois dans t. Un contexte est dit *propre* s'il contient au moins un symbole de fonction, c'est à dire s'il n'appartient pas à $Var \cup CVar$. Le contexte $\square$ est appelé *contexte vide*. Pour un n-contexte C, l'expression $C[t_1, \ldots, t_n]$ représente le terme $C\{ \square_i \mapsto t_i \mid 1 \leq i \leq n \}$.

2.2 Programmation logique

Un *programme logique* est un ensemble fini de clauses de Horn $H \leftarrow A_1, \ldots, A_n$ où$H, A_1, \ldots, A_n$ sont des atomes ($n \geq 0$). Notons que dans la suite on considère $A_i \neq A_j$ si $i \neq j$. H s'appelle la *tête* de la clause et $A_1, \ldots, A_n$ le *corps* de la clause. Les éléments de $Var(A_1, \ldots, A_n) \setminus Var(H)$ sont appelés *variables existentielles*. Une clause est dite *linéaire* si sa tête et son corps sont linéaires, c'est à dire si toute variable apparaît au plus une fois dans H et au plus une fois dans $A_1, \ldots, A_n$. Une clause dont le corps est vide, notée $H \leftarrow$ est appelée un *fait*.

Le *domaine de Herbrand* est l'ensemble des atomes clos, c'est à dire $\{ P(t_1, \ldots, t_n) \mid P \in \mathcal{P}r$ d'arité $n, t_i \in \mathcal{T}(\mathcal{F})$ pour $1 \leq i \leq n \}$. Un *modèle de Herbrand* pour une clause $H \leftarrow A_1, \ldots, A_n$ est un ensemble d'atomes clos $\mathcal{M}$ tel que pour toute substitution close σ telle que $Var(H \leftarrow A_1, \ldots, A_n) \subseteq Dom(\sigma)$ $\{H\sigma, A_1\sigma, \ldots, A_n\sigma\} \subseteq \mathcal{M}$. $\mathcal{M}$ est un modèle de Herbrand de $\mathcal{P}$ ssi $\mathcal{M}$ est un modèle de chacune des clause de $\mathcal{P}$. Pour chaque programme logique, il existe un *plus petit modèle de Herbrand* au sens de l'inclusion. Le plus petit modèle de Herbrand d'un programme $\mathcal{P}$ est en fait le plus petit point fixe de l'opérateur de déduction $T_{\mathcal{P}}$ défini de la manière suivante ($\mathcal{G}$ étant un ensemble d'atomes clos) :

$$T_{\mathcal{P}}(\mathcal{G}) = \mathcal{G} \cup \{ H\sigma \text{ clos } \mid \exists H \leftarrow \mathcal{B} \in \mathcal{P} \text{ et } \mathcal{B}\sigma \subseteq \mathcal{G} \}$$

Pour un programme logique $\mathcal{P}$ et un atome clos A on note $\mathcal{P} \models A$ si A appartient au plus petit modèle de Herbrand de $\mathcal{P}$. Le *langage décrit par un symbole de prédicat* P d'arité n dans un programme $\mathcal{P}$ est l'ensemble de n-uplets de termes clos $\{ (t_1, \ldots, t_n) \mid \mathcal{P} \models P(t_1, \ldots, t_n) \}$. Soient $\mathcal{P}$ un programme logique, $\mathcal{G}$ un ensemble d'atomes, on appelle *preuve* de $\mathcal{G}$ une suite finie $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$ où $\mathcal{G}_0 = \mathcal{G}$ et $\mathcal{G}_n = \emptyset$ et où $\mathcal{G}_i, \mathcal{G}_{i+1}$ sont tels que $\mathcal{G}_i = A, \mathcal{B}$, $A = H\sigma$, $G_{i+1} = \mathcal{B}'\sigma, \mathcal{B}$ et $H \leftarrow \mathcal{B}' \in \mathcal{P}$. n est la longueur de la preuve, les $\mathcal{G}_i$ $(1 \leq i \leq n)$ sont appelés buts. On note $\mathcal{P} \models \mathcal{G}$ s'il existe une preuve pour $\mathcal{G}$ utilisant les clauses de $\mathcal{P}$. Notons que si $\mathcal{P} \models \mathcal{G}$ alors toute instance close $\mathcal{G}\mu$ de $\mathcal{G}$ est incluse dans le plus petit modèle de Herbrand de $\mathcal{P}$. On étend la notion $\models$ à des atomes non-clos de la manière suivante $\mathcal{P} \models A$ ssi pour toutes les instances closes $\mathcal{A}\sigma$ de A $\mathcal{P} \models A\sigma$.

Le mécanisme opérationnel de la programmation logique est la *résolution*. Ce mécanisme se base sur l'algorithme d'unification de termes. Cet algorithme, inventé par A. Robison [82], a pour but de trouver l'unificateur le plus général de deux termes. Il existe de nombreuses manières de le présenter, la figure 2.1 utilise une approche à base de règles inspirée de [6]. Un problème d'unification est un ensemble d'équations $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$, l'algorithme d'unification consiste à appliquer les règles jusqu'à ce que plus aucune ne soit applicable. Quelque soit la stratégie, la dérivation se termine toujours avec pour résultat soit $fail$ ce qui signifie que le problème d'unification n'a pas de solutions, soit un ensemble d'équations E dit en *forme résolue* c'est à dire de la forme $x \stackrel{?}{=} t$ avec $x \notin Var(t)$ et x n'apparaît dans aucune autre équation de E. Dans ce cas l'unificateur le plus général est $\mu = \{ x \mapsto t \mid x \stackrel{?}{=} t \in E \}$. Notons que nous inversons les équations $t \stackrel{?}{=} x$ de manière à ce que les variables apparaissent toujours à gauche des équations.

Décomposition	$$\frac{\{f(t_1,\ldots,t_n) \stackrel{?}{=} f(s_1,\ldots,s_n)\} \cup E}{\{\, s_i \stackrel{?}{=} t_i \mid 1 \leq i \leq n \,\} \cup E}$$
Collision	$$\frac{\{f(t_1,\ldots,t_n) \stackrel{?}{=} g(s_1,\ldots,s_n)\} \cup E}{fail} \text{si } f \neq g$$
Instanciation	$$\frac{\{x \stackrel{?}{=} t\} \cup E}{\{x \stackrel{?}{=} t\} \cup E\{x \mapsto t\}} \text{si } t \neq x \text{ et } x \in Var(E)$$
Test d'occurrence	$$\frac{\{x \stackrel{?}{=} t\} \cup E}{fail} \text{si } t \neq x \text{ et } x \in Var(t)$$
Elimination des équations triviales	$$\frac{\{x \stackrel{?}{=} x\} \cup E}{E}$$

FIG. 2.1 – Unification

Nous montrons quelques propriétés intéressantes de l'unificateur le plus général dans le cas où l'un des deux termes ou les deux termes considérés sont linéaires.

Dans la suite τ_t est la profondeur du terme t, et $\tau_{x,t}$ est la profondeur minimale d'une occurrence de la variable x dans t. $ran(\mu)$ est l'ensemble $\{\, x\mu \mid x \in Dom(\mu) \,\}$. Enfin nous dirons d'un ensemble de termes qu'il est *linéaire* si une variable n'apparaît au plus qu'une seule fois dans cet ensemble de termes[1]

Lemme 2.2.1. *Soient s et t deux termes linéaires sans variables communes qui sont unifiables avec le mgu μ. $ran(\mu)$, l'ensemble des termes introduits par μ, est un ensemble linéaire de termes et $\tau_{s\mu} = \max(\tau_s, \tau_t)$.*

Preuve. Partant du problème d'unification $\{s \stackrel{?}{=} t\}$ on calcule μ en utilisant les règles de la figure 2.1. Comme s et t sont linéaires et sans variables communes, toutes les variables n'apparaissent au plus qu'une seule fois dans le problème d'unification, et seule la règle de décomposition est utilisée pour calculer l'unificateur. Les deux cotés des équations du système final sont donc constitués de sous-termes distincts et qui ne se recouvrent pas, de s et t par conséquent ils sont linéaires et n'ont pas de variables communes les uns avec les autres. L'ensemble $ran(\mu)$ (μ étant le mgu calculé) est un sous-ensemble de ces cotés d'équations par conséquent il est linéaire. De plus l'ensemble des positions du terme unifié $s\mu$ ($= t\mu$) est l'union des positions de s et de t, donc $\tau_{s\mu} = \max(\tau_s, \tau_t)$.

Notons que le problème d'unification $s \stackrel{?}{=} t$ est équivalent au problème $\{x \stackrel{?}{=} s, x \stackrel{?}{=} t\}$ où $x \notin Var(s) \cup Var(t)$. En itérant le lemme 2.2.1 on obtient le résultat pour l'unification d'un ensemble de termes linéaires sans variables communes.

Corollaire 2.2.2. *Soit $\{x \stackrel{?}{=} t_1, \ldots, x \stackrel{?}{=} t_n\}$ un problème d'unification où $\{t_1, \ldots, t_n\}$ est un ensemble linéaire de termes unifiables simultanément par l'unificateur le plus général μ. $ran(\mu)$ est un ensemble linéaire de termes et $\tau_{t_1\mu} = \max\{\tau_{t_1}, \ldots, \tau_{t_n}\}$.*

Dans le cas où seulement un des deux termes est linéaire, l'unificateur le plus général a aussi des propriétés intéressantes.

Lemme 2.2.3. *Soient s et t deux termes sans variables communes unifiables avec le mgu μ, et tels que t est linéaire. Soit μ_s la substitution μ restreinte aux variables de s ($\mu_s = \mu|_{Var(s)}$). $ran(\mu_s)$ est un ensemble linéaire de termes dont les variables sont incluses dans celles de t et $\tau_{x\mu_s} \leq \tau_t - \tau_{x,s}$ pour toute variable $x \in Var(s)$.*

[1]Notez qu'un ensemble linéaire de termes est plus restrictif qu'un ensemble de termes linéaires car dans le deuxième cas les termes peuvent avoir des variables communes alors que pas dans le premier cas.

Preuve. Partant du problème d'unification $\{s \stackrel{?}{=} t\}$ on calcule le mgu μ en utilisant les règles de la figure 2.1. On peut appliquer les règles de telle manière que l'unification se resolve en trois étapes :

1. Appliquer la règle de décomposition tant que c'est possible. Le problème résultant, P, contient deux types d'équations : $x \stackrel{?}{=} t'$ où x est une variable de s et t' est un sous-terme de t, et $y \stackrel{?}{=} s'$ où y est une variable de t et s' est un sous-terme de s. Pour les variables x de s il peut y avoir plusieurs équations de la forme $x \stackrel{?}{=} t'$ alors que ce n'est pas le cas pour les variables de t puisqu'il est linéaire. On peut noter que pour deux équations différentes $x_1 \stackrel{?}{=} t_1$ et $x_2 \stackrel{?}{=} t_2$ où x_1 et x_2 sont des variables de s, les termes t_1 et t_2 sont sans variables communes et linéaires
2. Soit P_x l'ensemble des équations de P suivant $\{x \stackrel{?}{=} t_1, \ldots, x \stackrel{?}{=} t_n\}$ tel que $x \in Var(s)$. Soit μ_x la solution de ce problème et P'_x la forme résolue de P_x. On peut remarquer que $\{t_1, \ldots, t_n\}$ est un ensemble linéaire de termes D'après le corollaire 2.2.2, l'ensemble $ran(\mu_x)$ est linéaire et $\tau_{t_1\mu_x} = \max\{\tau_{t_1}, \ldots, \tau_{t_n}\}$. Les t_i ne contiennent que des variables de t, par conséquent les variables de $ran(\mu_x)$ appartiennent donc à $Var(t)$. De plus, la profondeur des t_i est bornée par $\tau_t - \tau_{x,s}$, car ce sont des sous-termes de t apparaissant à la même position qu'une occurrence de x dans s. Par conséquent $t_x = t_1\mu_x$ est linéaire et $\tau_{t_x} = \tau_{t_1\mu_x} \leq \tau_t - \tau_{x,s}$. Remarquons que pour $x_1 \neq x_2$ les termes unifiés t_{x_1} et t_{x_2} (correspondant à P_{x_1} et P_{x_2}, respectivement) sont sans variables communes.
3. Soit P' le système obtenu en remplaçant dans P chaque P_x ($x \in Var(s)$) par sa forme résolue P'_x. En général P' n'est pas encore en forme résolue car les variables de x peuvent apparaître dans les équations $y \stackrel{?}{=} s'$ où $y \in Var(t)$. Pour trouver une forme résolue il suffit d'appliquer la règle instanciation autant de fois que possible. Il faut noter que l'application de cette règle ne va jamais instancier les équations de P'_x $x \in Var(s)$ par conséquent on obtient bien une forme résolue et $\mu_s = \{\, x \mapsto t_x \mid x \in Var(s) \,\}$. Les propriétés de μ_s découlent directement de la discussion de la seconde étape.

Pour un terme t on note par $\Delta Depth(t)$ est la différence maximale de profondeur entre deux occurrences de la même variable de t.

Corollaire 2.2.4. *Soient s et t deux termes sans variables communes unifiables avec le mgu μ, et tels que t est linéaire. $\Delta Depth(s\mu) = \Delta Depth(s)$ et le nombre maximal d'occurrences d'une variable est le même dans s et dans $s\mu$.*

Preuve. Les variables dupliquées de $s\mu$ sont soit des variables de s qui n'ont pas été instanciées par μ et dans ce cas la différence de hauteur maximale entre ces variables ne changent pas ni leur nombre d'occurrences, soit proviennent de l'instanciation d'une variable dupliquée de s car $ran(\mu_s)$ est linéaire et dans ce cas encore la différence de hauteur des différentes occurrences variables dans $s\mu$ est celle des variables instanciées par μ dans s et le nombre d'occurrences des variables de $x\mu_s$ est celui de x pour $x \in Dom(\mu_s)$.

Corollaire 2.2.5. *Soient s et t deux termes sans variables communes, unifiables avec le mgu μ, et tels que t est linéaire. $\tau_{x\mu} \leq \max(\tau_s, \tau_t + \Delta Depth(s))$.*

Preuve. Pour toute variable x de s $x\mu$ est un terme t' tel que t' est le terme unifié d'un problème d'unification entre sous-termes de t dont la profondeur maximale est $\tau_t - m_x$ où m_x est la profondeur minimale à laquelle apparaît x dans s. Donc d'après le corollaire 2.2.2, $\tau_{t'} \leq \tau_t - m_x \leq \tau_t \leq \tau_t + \Delta Depth(s)$ donc la propriété est vérifiée pour les variables de s.

Les variables y de t sont elles instanciées par des termes de la forme $s'\mu_s$ où s' est un sous-terme de s. Si $s'\mu = s'$ alors la propriété est vérifiée car $\tau_{s'} \leq \tau_s$. Sinon $\tau_{s'\mu}$ est par définition égal à $\max(\{\, u \mid u \in Pos(s'\mu) \,\})$ donc soit $\tau_{s'\mu} = \tau'_s$ qui est inférieur à τ_s donc la propriété est vérifiée, soit il existe une position $u \in Pos(s'\mu)$ qui n'est pas dans $Pos(s')$ est qui a une longueur plus grande que τ'_s cela signifie que $u = v.v'$ avec $s'|_v = x$ une variable de s et x est instanciée par μ. Supposons que la variable y apparaît à la profondeur p dans t, que $|v| = p'$ que $x\mu = s'$. On sait que $p' \leq \tau_s - p$. On sait aussi que la profondeur minimale à laquelle x apparaît dans s est bornée par $p + p' - \Delta Depth(s)$ donc $\tau_{t'} \leq \tau_t - (p + p' - \Delta Depth(s))$ par conséquent $\tau_{t'} \leq \tau_t - (p + \tau_s - p - \Delta Depth(s))$ or $\tau_t - (p + \tau_s - p - \Delta Depth(s)) = \tau_t - \tau_s + \Delta Depth(s)$ donc $\tau_{t'} \leq \tau_t - \tau_s + \Delta Depth(s)$. Donc $\tau_{s'\mu} \leq p' + \tau_t - \tau_s + \Delta Depth(s)$ or $p' \leq \tau_s$ ce qui signifie que $\tau_{s'\mu} \leq \tau_s + \tau_t - \tau_s + \Delta Depth(s) = \tau_t + \Delta Depth(s)$

2.3 Réécriture

Un *système de réécriture* ou *TRS* (pour Term Rewrite System) est un ensemble fini de *règles de réécriture* $l \to r$, où $l, r \in \mathcal{T}(\mathcal{F}, Var)$. Si le membre gauche de chaque règle est un terme linéaire, le TRS est dit *linéaire gauche* ; si le membre droit de chaque règle est linéaire, le TRS est dit *linéaire droit*. Un TRS qui est à la fois linéaire droit et linéaire gauche est dit *linéaire*.

Pour un TRS R, la relation de réécriture $\to_R$ est définie comme suit : $s \to_R t$ ssi il existe une règle $l \to r$ de R, une position non-variable u de s, et une substitution σ telle que $s|_u = l\sigma$ et $t = s[u \leftarrow r\sigma]$. La transformation de s en t est appelée une *étape de réécriture* et s'écrit $s \to_{[u,l\to r]} t$, $s|_u$ est un *redex* de s et u une *position de redex*. Si un terme t ne peut être réduit par aucune règle de réécriture, il est dit *irreductible*. La clôture réflexive-transitive de $\to_R$ est notée $\to_R^*$. La relation $\to_R^n$ représente n étapes de la relation de réécriture. Une séquence $t_0 \to_R t_1 \to_R \cdots \to_R t_n$ est appelée une *dérivation*. Un système de réécriture R est dit *confluent* si pour tous termes s, s_1, s_2 tels $s \to_R^* s_1$ et $s \to_R^* s_2$, il existe un terme t tel que $s_1 \to_R^* t$ et $s_2 \to_R^* t$. Pour un ensemble de termes E, on définit $R^*(E) = \{ t \mid s \to_R^* t \text{ avec } s \in E \}$. Deux termes s et t sont dits *R-joignables* (on note $s \downarrow_R r$) si il existe une substitution σ et un terme s' tels que $s\sigma \to_R^* s'$ et $t \to_R^* s'$. Lorsque R est confluent on dit que s et t sont *R-unifiables* (on note $s =_R t$) et que σ est un *R-unificateur*.

Un système de réécriture R est dit *basé sur les constructeurs* si toutes ses règles sont de la forme $f(t_1, \ldots, t_n) \to r$ où $f \in \mathcal{D}$, $t_i \in \mathcal{T}(\mathcal{C}, Var)$ et $r \in \mathcal{T}(\mathcal{F}, Var)$. S'il existe $u, v \in Pos(r)$ telles que $u < v$, $t(u) \in \mathcal{D}$ et $t(v) \in \mathcal{D}$, on dit que r contient des *fonctions définies imbriquées*. Une étape de réécriture $s \to_{[u,l\to r]} t$ est dite une *data-étape* si σ telle que $s|_u = l\sigma$ est une data-substitution. Un dérivation de réécriture est une *data-dérivation* si toutes ses étapes sont des data-étapes. Les termes s et t sont dits *data-joignables* s'il existe un data-substitution σ et un data-terme s' tels que $s\sigma \to_R^* s'$ et $t\sigma \to_R^* s'$ par des data-dérivations. Lorsque R est confluent on dit que s et t sont *data-R-unifiables* et que σ est un *data-R-unificateur*.

Les définitions suivantes sont moins standard. Leur objectif est d'identifier les positions d'un terme t qui peuvent être réécrites dans t ou dans une de ses instances.

Définition 2.3.1. Soient R un TRS et t un terme. On dit que la position $p \in \Sigma Pos(t)$ est une *position de redex potentiel*, s'il existe $t\sigma$, une instance close de t telle que $t\sigma \to^* t'$ sans jamais réécrire à des occurrences plus petites que p et p est une position de redex de t'. L'ensemble des positions de redex potentiel du terme t est notée $Pot_R(t)$.

Il est en général indécidable de savoir si une position d'un terme t est une position de redex potentiel car répondre à ce problème revient à décider de l'unifiabilité modulo R qui est connu pour être indécidable. Afin de pouvoir utiliser concrètement la notion de redex potentiel nous utiliserons des sur-approximations de cet ensemble. De manière générique on appellera ensemble de *positions de redex possible*, noté $PRedPos_R(t)$, un ensemble tel que $Pot_R(t) \subseteq PRedPos_R(t) \subseteq \Sigma Pos(t)$. Les définitions 2.3.4 et 2.3.5 sont deux exemples d'approximations possibles qui seront utilisées par la suite.

Définition 2.3.2. Soient R un TRS et t un terme et $PRedPos(t)$ un ensemble de positions de redex posible de t. Tout élément de $PRedPos(t)$ est appelé *position de redex possible*. Si $u \in PRedPos(t)$, on appelle *redex possible* à la position u de t, le contexte C tel que $Pos(C) = \{ v \mid u.v \in Pos(t), \nexists v' \text{tel que } u < u.v' < u.v \text{ et } u.v' \in PRedPos(t) \}$ et $C(v) = t(u.v)$ si $u.v \notin PRedPos(t)$ et $C(v) \in CVar$ sinon. L'ensemble de tous les redex possibles de t est noté $PRed(t)$. L'ensemble des variables apparaissant dans un redex possible de t est le suivant $PRedVar_R(t) = Var(t) \cap \bigcup_{C \in PRed_R(t)} Var(C)$. Pour une variable $x \in PRedVar_R(t)$, le nombre $PRedDepth_R(x)$ est la profondeur maximale à laquelle x apparaît dans un redex possible de t. Le contexte C qui ne contient aucun redex possible et qui est tel que $t = C[t_1, \dots, t_n]$ où u_i pour $1 \leq i \leq n$ est une position de redex possible, est appelé la *partie irreductible* de t et se note $Irr_R(t)$. Si u et v sont deux positions de redex possible telles que $u < v$ alors v est dite *imbriquée*.

Exemple 2.3.3. Soient R le TRS $\{f(s(x)) \to c(f(p(x)), f(f(x)))\}$ et t le terme $c(f(p(x)), f(f(x)))$. Si on pose $PRedPos_R(t) = \{2, 2.1\}$ qui est bien un ensemble de redex possibles de ce terme, on a alors $PRed_R(t) = \{f(\Box_1), f(x)\}$, $Irr_R(t) = c(f(p(x)), \Box_1)$, $PRedVar(t) = \{x\}$, et $PRedDepth(x) = 1$. t contient une position de redex imbriquée : 2.1.

Les deux définitions suivantes sont des approximations intéresssantes de $Pot_R(t)$ qui seront utilisées dans le chapitre 4.

Définition 2.3.4. Soient R un système de réécriture et t un terme. L'ensemble de *positions de redex possible* de t, noté $PRedPos_R(t)$, est le plus petit ensemble vérifiant
- $PRedPos_R(t) = \emptyset$ si $t \in Var$;
- $PRedPos_R(t) = S_{\text{sub}} \cup S_\varepsilon$ si $t = f(t_1, \dots, t_n)$,
  avec $S_{\text{sub}} = \bigcup_{i=1}^n i.PRedPos_R(t_i)$ et S_ε vaut
  - $\{\varepsilon\}$ si t est de la forme $C[s_1, \dots, s_k]$ telle que C est un contexte propre s'unifiant avec au moins un membre gauche de R et tel que l'ensemble $\{u_1, \dots, u_k\}$ des positions de variables de contexte vaut $\{ u \in S_{\text{sub}} \mid v \not< u \text{ pour tout } v \in S_{\text{sub}} \}$
  - $\emptyset$ sinon.

Par une induction sur la hauteur du terme t, on montre facilement que cette définition nous donne bien un ensemble de positions de redex possible. Dans l'exemple 2.3.3, l'ensemble des positions de redex possible donné correspond à cette définition. La définition suivante se place dans le cadre des systèmes basés sur les constructeurs et collecte simplement les positions étiquetées par des symboles de fonction définie.

Définition 2.3.5. Soient R un système de réécriture basé sur les constructeurs et t un terme. L'ensemble de *positions de redex possible* de t, noté $PRedPos_R(t)$, et l'ensemble $\{ u \in \Sigma Pos(t) \mid t(u) \in \mathcal{D} \}$

Il est facile de voir que l'ensemble défini ainsi est bien un ensemble de positions de redex possible puisque les seuls symboles de fonction qui se réécrivent sont dans $\mathcal{D}$ dans le cadre de la réécriture basée sur les constructeurs.

Remarque 2.3.6. Si $PRedPos(t) = \emptyset$ alors t est irreductible.

2.4 Automates et grammaires d'arbres

Un *automate d'arbres* $\mathcal{A}$ est un quadruplet $(\Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ où Σ est un ensemble fini de symboles de fonction avec arité, $\mathcal{Q}$ un ensemble fini d'*états* disjoint de Σ, $\mathcal{Q}_f$ un ensemble fini d'*états finaux* inclus dans $\mathcal{Q}$ et Δ un ensemble fini de *transitions* de la forme $f(q_1, \dots, q_n) \to q$ où $f \in \Sigma$ et $q, q_1, \dots, q_n \in \mathcal{Q}$. Δ est considéré comme un système de réécriture sur la signature $\mathcal{T}(\Sigma \cup \mathcal{Q})$. Un terme clos $t \in \mathcal{T}(\Sigma)$ est *reconnu* par $\mathcal{A}$ ssi il existe $q \in \mathcal{Q}_f$ est $t \to_\Delta^* q$. Le *langage reconnu* par $\mathcal{A}$, noté $\mathcal{L}(\mathcal{A})$, est l'ensemble des termes clos reconnus par $\mathcal{A}$.

Une *grammaire d'arbres* $\mathcal{G}$ est un quadruplet $(\Sigma, NT, \mathcal{P}, I)$ où Σ est un ensemble fini de symboles de fonctions avec arité, NT un ensemble fini de *non-terminaux* d'arité 0, disjoint de Σ, $\mathcal{P}$ un ensemble fini de *productions* de la forme $X \Rightarrow t$ où $X \in NT$, $t \in \mathcal{T}(\Sigma \cup NT)$ et $I \in NT$ est l'*axiome* de la grammaire. $\mathcal{P}$ est considéré comme un système de réécriture clos sur le signature $\mathcal{T}(\Sigma \cup NT)$. Un terme clos de $\mathcal{T}(\Sigma)$ est *généré* par $\mathcal{G}$ ssi $I \Rightarrow^* t$. Le *langage généré* par $\mathcal{G}$, noté $\mathcal{L}(\mathcal{G})$, est l'ensemble des termes clos générés par $\mathcal{G}$. Pour toute grammaire d'arbres $\mathcal{G}$, il existe une grammaire d'arbres $\mathcal{G}'$ dont toutes les productions sont de la forme $X \Rightarrow f(X_1, \dots, X_n)$ où $X, X_1, \dots, X_n$ sont des non-terminaux et f est un symbole de fonction d'arité n et telle que $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$. $\mathcal{G}'$ est dite *normalisée*.

L'ensemble des langages reconnus par les automates d'arbres et le même que l'ensemble des langages générés par les grammaires d'arbres. Cet ensemble constitue la classe des *langages réguliers d'arbres*. L'intuition de ce résultat est que les productions des grammaires normalisées sont des transitions d'un automate qui ont été renversées. On parle de génération dans le cadre des grammaires d'arbres car on part de l'axiome pour générer les termes du langage (vision "top-down"). On parle de reconnaissance pour les automates d'arbres car on part d'un terme du langage et on essaie de le réduire en un état final (vision "bottom-up").

Le *test du vide* (c'est à dire est-ce que le langage est vide ?) et *d'appartenance* (c'est à dire est-ce qu'un terme particulier appartient au langage ?) sont décidables pour les langages réguliers. De plus les langages réguliers sont clos par les opérations ensemblistes : union, intersection et complément.

2.5 Langages de n-uplets d'arbres

Dans cette section nous étendons la notion de langages d'arbres aux langages de n-uplets d'arbres. Ces langages permettent, par exemple, de représenter des relations sur les termes clos. Nous verrons dans le chapitre 4 que ces langages permettent de représenter la relation de réécriture. Un *langage de n-uplets d'arbres* $\mathcal{L}$ est donc un ensemble de n-uplets de la forme $(t_1, \ldots, t_n)$ où $t_i \in \mathcal{T}(\Sigma)$ pour $1 \leq i \leq n$. n est l'*arité du langage* notée $\mathrm{ar}(\mathcal{L})$. Comme pour les langages d'arbres, on s'intéresse aux tests d'appartenance et du vide des langages de n-uplets d'arbres. Les opérations ensemblistes comme l'union, l'intersection et le complément se définissent de la même manière que leurs équivalentes sur les langages d'arbres, à la seule différence qu'il faut que les deux langages considérés soient de même arité. Enfin on s'intéresse à de nouvelles opérations spécifiques aux langages de n-uplets.

Définition 2.5.1. Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages de n-uplets d'arbres sur la signature Σ d'arité respective n_1 et n_2, soit π une permutation sur $[1, n_1]$.

- La *restriction*[2] de $\mathcal{L}_1$, notée $\mathcal{L}_1\backslash_i$, est le langage d'arité i_1-1 $\{\, (s_1, \ldots, s_{i_1-1}) \mid \exists (t_1, \ldots, t_{n_1}) \in \mathcal{L}_1$ et $s_j = t_j$ pour $1 \leq j < i$ et $s_j = t_{j-1}$ pour $i \leq j < n_1 \,\}$. On peut facilement étendre la restriction à un ensemble de composants $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n_1\}$ de la manière suivante : $\mathcal{L}_1\backslash_{i_1,\ldots,i_k} = (\mathcal{L}_1\backslash_{i_1})\backslash_{j_2,\ldots,j_k}$ où $j_l = i_l$ si $j_l < i_1$ et $j_l = i_l - 1$ so $j_l \geq i_1$.
- La *projection* de $\mathcal{L}_1$ sur l'ensemble des composants $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n_1\}$, notée $\mathcal{P}_{i_1,\ldots,i_k}(\mathcal{L}_1) = \mathcal{L}_1\backslash_{\{j \in \{1,\ldots,n_1\} \backslash \{i_1,\ldots,i_k\}\}}$
- La *cylindrification* de $\mathcal{L}_1$ sur le composant i $(0 \leq i \leq n_1)$, noté $C_i(\mathcal{L}_1)$, est le langage d'arité $n_1 + 1$ $\{\, (s_1, \ldots, s_{n_1+1}) \mid \exists (t_1, \ldots, t_{n_1}) \in \mathcal{L}_1, s_j = t_j$ pour $1 \leq j \leq i$ et $s_j = t_{j-1}$ pour $i < j \leq n_1 + 1$ et $s_i \in \mathcal{T}(\mathcal{F}) \,\}$. On peut facilement étendre la cylindrification à un ensemble de composants $\{i_1, \ldots, i_k\} \subseteq \{0, \ldots, n_1\}$ de la manière suivante : $C_{i_1,\ldots,i_k}(\mathcal{L}_1) = C_{j_2,\ldots,j_k}(C_{i_1}(\mathcal{L}_1))$ où $j_l = i_l$ si $j_l \leq i_1$ et $j_l = i_l + 1$ si $j_l > i_1$.
- La *permutation suivant* σ de $\mathcal{L}_1$, notée $\sigma(L_1)$, est le langage d'arité n_1 $\{\, (s_1, \ldots, s_{n_1}) \mid \exists (t_1, \ldots, t_{n_1}) \in \mathcal{L}_1$ et $s_j = t_{j\sigma}$ pour $1 \leq j \leq n_1 \,\}$.
- Le *produit cartésien* de $\mathcal{L}1$ et $\mathcal{L}_2$, noté $\mathcal{L}_1 \times \mathcal{L}_2$ est le langage d'arité $n_1 + n_2$ $\{\, (s_1, \ldots, s_{n_1}, t_1, \ldots, t_{n_2}) \mid (s_1, \ldots, s_{n_1}) \in \mathcal{L}_1$ et $(t_1, \ldots, t_{n_2}) \in \mathcal{L}_2 \,\}$.
- La *jointure* i, j de $\mathcal{L}_1$ avec $\mathcal{L}_2$ $(1 \leq i \leq n_1$ et $1 \leq j \leq n_2)$, notée $\mathcal{L}_1 \ltimes_{i,j} \mathcal{L}_2$ est le langage d'arité $n_1 + n_2 - 1$: $\{\, (s_1, \ldots, s_{n_1}, t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_{n_2}) \mid (s_1, \ldots, s_{n_1}) \in \mathcal{L}_1$ et $(t_1, \ldots, t_{j-1}, s_i, t_{j+1}, \ldots, t_{n_2}) \in \mathcal{L}_2 \,\}$.

Exemple 2.5.2. Soient $\mathcal{L}_1 = \{(s^n(0), s^m(0), s^{n+m}(0))\}$ et $\mathcal{L}_2 = \{(s^n(0), s^{2n}(0))\}$ où $s^n(0)$ représente le terme $\overbrace{s(s(\ldots s(}^{n \text{ fois}}0)\ldots))$. Soit $\sigma = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$

- $\mathcal{L}_1\backslash_2 = \{\, (s^n(0), s^m(0)) \mid \exists (s^n(0), s^k(0), s^m(0)) \in \mathcal{L}_1 \,\}$. En d'autres termes c'est l'ensemble des couples $(s^n(0), s^m(0))$ tels que $m \geq n$.

[2]Dans [21] cette opération est appelée projection mais elle conserve les composants inverses de ceux attendus par une projection.

- $\Pi_{1,3}(\mathcal{L}_1) = \mathcal{L}_1 \backslash_2$.
- $C_1(\mathcal{L}_2) = \{(s^n(0), s^k(0), s^{2n}(0))\}$.
- $\sigma(\mathcal{L}_1) = \{(s^{n+m}(0), s^n(0), s^m(0))\}$
- $\mathcal{L}_1 \times L_2 = \{(s^n(0), s^m(0), s^{n+m}(0), s^k(0), s^{2k}(0))\}$
- $\mathcal{L}_1 \bowtie_{1,2} \mathcal{L}_2 = \{(s^{2k}(0), s^m(0), s^{2k+m}(0), s^k(0))\}$.

On peut remarquer que la jointure peut s'exprimer en fonction des autres opérations.

$$\mathcal{L}_1 \bowtie_{i,j} \mathcal{L}_2 = C_{n_1+n_2-2}(C_{n_1+n_2-3}(\dots C_{n_1}(\mathcal{L}_1)\dots)) \cap \sigma(\overbrace{C_0(C_0(\dots C_0(}^{(n_1-1)fois}\mathcal{L}_2)\dots)))$$

où σ est une permutation sur $[1, n_1+n_2-1]$ telle que $k\sigma = k$ si $k < i$ ou $k > n_1+j$, $k\sigma = k+1$ si $i \leq k < n_1+j$ et $n_1+j\sigma = i$.

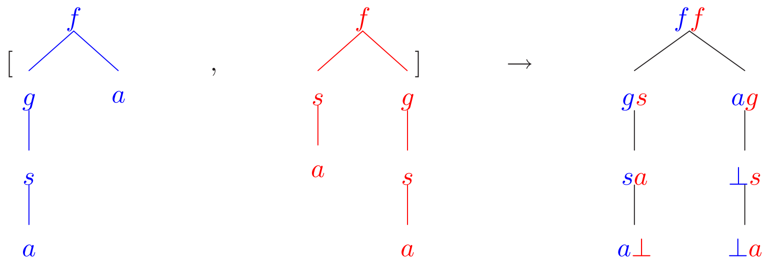
Par exemple, $\mathcal{L}_1 \bowtie_{1,2} \mathcal{L}2 = C_3(\mathcal{L}_1)) \cap \sigma(C_0(C_0(\mathcal{L}_2)))$ avec $\sigma = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 1\}$. En effet $C_3(\mathcal{L}_1) = \{(s^n(0), s^m(0), s^{n+m}(0), s^k(0))\}$ et $C_0(C_0(\mathcal{L}_2)) = \{(s^n(0), s^m(0), s^k(0), s^{2k}(0))\}$ d'où $\sigma(C_0(C_0(\mathcal{L}_2))) = \{(s^{2k}(0), s^n(0), s^m(0), s^k(0))\}$ et donc par conséquent, on retrouve bien que $C_3(\mathcal{L}_1)) \cap \sigma(C_0(C_0(\mathcal{L}_2))) = \{(s^{2k}(0), s^m(0), s^{2k+m}(0), s^k(0))\}$.

Les propriétés recherchées pour les langages de n-uplets d'arbres sont celles des langages d'arbres plus la clôture par les opérations définies ci-dessus. Un classe de langages bien connue possède ces propriétés, c'est la classe des *relations régulières* présentée dans [21] sous le nom de relations reconnaissables. Pour définir les relations régulières nous avons besoin de définir une opération de *superposition* entre termes dont l'objectif est de représenter un n-uplet de termes en un seul terme.

Définition 2.5.3. Soient $f(t_1, \dots, t_n)$ et $g(s_1, \dots, s_k)$ deux termes. $[f(t_1, \dots, t_n), g(s_1, \dots, s_k)] = fg([t_1, s_1], \dots, [t_m s_m])$ où $m = Max(n, k)$ et $\forall i > n\ t_i = \perp$ et $\forall i > k\ s_i = \perp$

On généralise $[t_1, t_2, \dots, t_n] = [[t_1, t_2], \dots, t_n]$

Exemple 2.5.4. La superposition des termes $f(g(s(a)), a)$ et $f(s(a), g(s(a)))$ donne le résultat suivant :



Définition 2.5.5. La classe des *relations régulières* est l'ensemble des relations $R \in \mathcal{T}(\Sigma)^n$ telles que $\{\ [t_1, \dots, t_n] \mid (t_1, \dots, t_n) \in R\ \}$ est reconnu par un automate d'arbres construit sur $(\Sigma \cup \{\perp\})^n$.

La clôture des relations régulières par union, intersection et complément se déduit directement des propriétés de langages réguliers d'arbres. La clôture par restriction s'obtient de la manière suivante : pour chaque transition $f_1 \dots f_{i-1} f_i f_{i+1} \dots f_n(\vec{q}) \rightarrow q$ de l'automate qui reconnaît la relation R d'arité n, on fabrique la transition $f_1 \dots f_{i-1} f_{i+1} \dots f_n(\vec{q}) \rightarrow q$ pour reconnaître la relation $R\backslash_i$. La clôture par cylindrification s'obtient en créant pour chaque transition $f_1 \dots f_{i-1} f_i \dots f_n(\vec{q}) \rightarrow q$ de R l'ensemble de transitions $\{\ f_1 \dots f_{i-1} f f_i \dots f_n(\vec{q}) \rightarrow q \mid f \in \Sigma\ \}$. La clôture par permutation se déduit assez facilement en transformant les transitions de E de la manière suivante : $f_{1\sigma} \dots f_{n\sigma}(\vec{q}) \rightarrow q$. La clôture par les autres opérations se déduit des précédentes.

Par conséquent, les relations régulières ont toutes les qualités en ce qui concerne les propriétés de décidabilité et de clôture par les opérations sur les relations. Elles ont par contre deux inconvénients majeurs. Le premier est leur faible pouvoir d'expression ; en effet les relations régulières ne permettent de contrôler que les symboles de fonctions qui apparaissent à la même position dans chacun des composants du n-uplet et par conséquent des relations du type $(s^n(0), s^{2n}(0))$ ne sont pas des relations régulières. Le deuxième inconvénient est le formalisme utilisé pour les représenter, c'est à dire la superposition de terme, n'est pas très facile d'utilisation. Des extensions des relations régulières ont été étudiées, comme dans [87], en utilisant la notion d'automates avec tests, mais ces extensions perdent rapidement la décidabilité du test du vide.

Dans la section suivante, nous présenterons différents formalismes pour représenter des langages de n-uplets d'arbres qui permettent d'aller au delà des langages réguliers tout en préservant la décidabilité du test du vide ainsi qu'un certain nombre de propriétés de clôture par les opérations sur les relations.

Chapitre 3

Représentation des langages de n-uplets d'arbres

Dans ce chapitre nous présentons trois formalismes de représentation des langages de n-uplets d'arbres que nous avons étudiés. Le premier formalisme, appellé "grammaires synchronisées", a été introduit dans [56] et est une extension des grammaires d'arbres. Il a permis de définir une classe de langages de n-uplets d'arbres très intéressante en terme d'expressivité, préservant de bonnes propriétés de clôture pour certaines opérations sur les relations ainsi que des tests du vide et d'appartenance décidables. Ce formalisme comportent cependant un certains nombres de défauts, ce qui nous a amené à introduire dans [40, 59] un autre formalisme appelé "systèmes de contraintes" se basant sur les contraintes ensemblistes. Les systèmes de contraintes permettent de résoudre un certain nombre de problèmes posés par les grammaires synchronisées mais restent assez difficiles à utiliser c'est pourquoi dans [60] nous avons étudié les liens entre les systèmes de contraintes et la programmation logique, ce qui nous a conduit à définir des classes de programmes logiques qui permettent de définir les différentes classes de langages de n-uplets d'arbres présentées dans ce mémoire. Nous avons pu aussi donner un algorithme de transformation de programmes logiques qui calcule les différentes opérations sur les relations.

3.1 Les grammaires synchronisées

Dans cette section nous donnons simplement les définitions liées aux grammaires synchronisées dans une version simplifiée ainsi que les résultats que nous avons obtenus. Enfin nous discutons des inconvénients de ce formalisme.

Intuitivement, une grammaire synchronisée est une grammaire d'arbres dont les productions sont soit libres (c'est à dire qu'on les utilise comme des productions des grammaires standard) soit synchronisées, dans ce cas elles sont regroupées dans un paquet de productions qui doivent être toutes appliquées en même temps. Pour contrôler l'application des paquets de productions, on associe à chaque non-terminal apparaissant dans un arbre dérivé de l'axiome, un entier de contrôle. Un paquet de productions synchronisés ne peut s'appliquer que sur des non-terminaux qui ont des entiers de contrôle indentiques.

Définition 3.1.1. Soit Σ une signature et NT un ensemble fini de symboles d'arité 0 disjoint de Σ.

Un *paquet de productions* est un ensemble de productions associé à un entier. Il se note $\{X_1 => t_1, \ldots, X_n => t_n\}_k$.

– Lorsque $k = 0$ le paquet est un singleton de la forme $\{X_1 => f(Y_1, \ldots, Y_n)\}_0$ où $f \in \Sigma$ et $Y_1, \ldots, Y_n \in NT$ et sont deux à deux distincts. Cette production est dite *libre*, elle s'écrit plus simplement $X_1 => c(Y_1, \ldots, Y_n)$.

- Lorsque $k > 0$ le paquet est de la forme $\{X_1 => Y_1, \ldots, X_n => Y_n\}_k$ où les X_i sont tous différents entre eux ainsi que les Y_i, $Y_1, \ldots, Y_n \in NT$. Les productions du paquet sont dites *synchronisées*.

Définition 3.1.2. Une *grammaire synchronisée* (abrégée en TTSG pour Tree Tuple Synchronized Grammar) est définie par (Σ, NT, PP, TI) où
- Σ est l'ensemble des symboles terminaux,
- NT est un ensemble fini de non-terminaux,
- PP est un ensemble fini de paquets de productions tel que si $X \Rightarrow t_1$ et $Y \Rightarrow t_2$ sont des productions libres de PP alors $X \neq Y$ implique que t_1 et t_2 n'ont pas de non-terminaux commun,
- TI est l'axiome de la TTSG. c'est un n-uplet $((I_1, 0), \ldots, (I_n, 0))$ où chaque I_i est un non-terminal.

La définition suivante permet de préciser comment sont appliquées les productions d'une TTSG. Il faut bien remarquer que ce que nous nommons calcul n'est pas une dérivation mais un n-uplet d'arbres (contenant éventuellement des non-terminaux) obtenu par une dérivation de la grammaire.

Définition 3.1.3. L'ensemble des *calculs* d'une TTSG
$Gr = (Sz, \mathcal{C}, NT, PP, TI)$ est noté $Comp(Gr)$. C'est le plus petit ensemble vérifiant :
- $TI \in Comp(Gr)$,
- Si $tp \in Comp(Gr)$ et $tp|_u = (X, ct)$ et la production libre $X => c(Y_1, \ldots, Y_n) \in PP$ alors $tp' = tp[u \leftarrow c((Y_1, ct), \ldots, (Y_n, ct))] \in Comp(Gr)$ (on note $tp => tp'$),
- Si $tp \in Comp(Gr)$ et s'il existe n occurrences $u_1, \ldots, u_n$ de tp, différentes deux à deux, telles que $\forall i \in [1, n]$ $tp|_{u_i} = (X_i, c_i)$ et $c_i = a$ ($a \in I\!N$) et le paquet de productions synchronisées $\{X_1 => Y_1, \ldots, X_n => Y_n\}_k \in PP$, alors $tp' = tp[u_1 \leftarrow (Y_1, b)] \ldots [u_n \leftarrow (Y_n, b)]$ (où b est un nouvel entier non encore utilisé dans tp) est dans $Comp(Gr)$, (on note $tp => tp'$).

Une *dérivation* de Gr est une suite ordonnée de calculs de Gr, $tp_1, tp_2, \ldots, tp_n$, tels que $tp_1 = TI$ et pour tout $i \in [2, n]$, $tp_{i-1} => tp_i$.

Remarquons que dans un calcul, les non-terminaux apparaissent seulement aux feuilles et que les entiers de contrôle ne sont pas associés aux symboles terminaux mais seulement aux non-terminaux.

Définition 3.1.4. Le langage *généré* par une TTSG Gr est noté $Rec(Gr)$. C'est l'ensemble des n-uplets de termes clos qui sont des calculs de Gr.

Exemple 3.1.5. Soit Gr la grammaire synchronisée définie sur la signature $\{c, a\}$ où c est d'arité 2 et a est d'arité 0. Soient $\{X, Y, X_1, Y_1, X_2, Y_2\}$ les non-terminaux de cette grammaire et $((X, 0), (Y, 0))$ son axiome. Les paquets de productions de Gr sont les suivants.

$$X \Rightarrow c(X_1, Y_1), Y \Rightarrow c(Y_2, X_2), X \Rightarrow a, Y \Rightarrow a, \{X_1 \Rightarrow X, X_2 \Rightarrow Y\}_1, \{Y_1 \Rightarrow X, Y_2 \Rightarrow Y\}_1$$

Voici une dérivation de Gr :

$$\begin{array}{ll} ((X,0),(Y,0)) & \Rightarrow (c((X_1,0),(Y_1,0)), c((Y_2,0),(X_2,0))) \\ & \Rightarrow (c((X,1),(Y_1,0)), c((Y_2,0),(Y,1))) \\ & \Rightarrow (c((X,1),(X,2)), c((Y,2),(Y,1))) \end{array}$$

A ce niveau de la dérivation on peut voir l'intérêt de l'entier de contrôle associé à chaque non terminal. En effet les non-terminaux X et Y apparaissent chacun deux fois dans le calcul, sans entier de contrôle, nous n'aurions plus aucun moyen de savoir quel X est synchronisé avec quel Y

$$\begin{array}{ll} (c((X,1),(X,2)), c((Y,2),(Y,1))) & \Rightarrow (c(a,(X,2)), c((Y,2),(Y,1))) \\ & \Rightarrow (c(a,(X,2)), c((Y,2),a)) \\ & \Rightarrow^* (c(a, c((X_1,2),(Y_1,2))), c(c((Y_2,2),(X_2,2)),a)) \\ & \Rightarrow^* (c(a, c((X,3),(X,4))), c(c((Y,4),(X,4)),a)) \\ & \Rightarrow^* (c(a, c(a,a)), c(c(a,a),a)) \end{array}$$

On peut montrer que la grammaire Gr génère l'ensemble des couples (t_1, t_2) où t_1 est le symétrique de t_2.

Définition 3.1.6. Un langage généré par une grammaire synchronisée est appelé *langage synchronisé*.

Les propriétés suivantes ont été démontrées dans [55].

Théorème 3.1.7. *Il est décidable de savoir si le langage généré par une TTSG est vide.*

La preuve de ce théorème se base sur le fait que dans tout calcul d'une grammaire synchronisée tous les non-terminaux qui porte le même numéro de contrôle sont différents deux à deux, ce qui permet trouver des cycles dans les dérivations des grammaires.

Théorème 3.1.8. *Il est décidable de savoir si un n-uplet de termes clos généré par une TTSG.*

Théorème 3.1.9. *L'union de deux langages générés par des TTSG est reconnue par une TTSG.*

Pour cela il suffit de supposer que les deux grammaires initiales n'ont pas de non-terminaux communs, et de construire la grammaire qui contient l'union des transitions des deux grammaires initiales.

Proposition 3.1.10. *Il est indécidable de savoir si le langage généré par l'intersection deux TTSG est vide ou non.*

Preuve. On peut décrire le problème de correspondance de Post par un problème d'intersection de langages générés par des TTSG. Soit b un constructeur binaire, considérons les langages de 1-uplets $L = \{b(\alpha, \phi(\alpha)) | \, \alpha \in A^+\}$ et $L' = \{b(\alpha, \phi'(\alpha)) | \, \alpha \in A^+\}$. On suppose que $\phi(a_i) = c_{i,1}...c_{i,p_i}$ pour chaque $a_i \in A$. Alors le langage L est généré par la TTSG Gr définie par les productions suivantes :

- $I => b(X, Y)$
- $\{Z => X, T => Y\}$,
- $\{Z => \perp, T => \perp\}$,
- et pour chaque $a_i \in A$:
  - $\{X => X_i, Y => Y_i\}$,
  - $X_i => a_i Z$, $Y_i => c_{i,1}(Y_{i,2}), Y_{i,2} => c_{i,2}(Y_{i,3}), ..., Y_{i,p_i} => c_{i,p_i}(T)$,

et l'axiome I.

La grammaire Gr' qui génère L' est construite de la même façon. Si l'on veut savoir si $L \cap L'$ est vide il nous faudra chercher l'existence de $\alpha \in A^+$ tel que $\phi(\alpha) = \phi'(\alpha)$, ce que revient à résoudre le problème de correspondance de Post.

La jointure de deux langages générés par des TTSG est un langage généré par une TTSG si on impose des restrictions sur le composant de jointure d'un des deux langages.

Définition 3.1.11. Soit Gr une TTSG, le i^{eme} composant de Gr est dit *libre* si pour chaque paquet de productions synchronisées $\{X_1 \Rightarrow Y_1, \dots, X_n \Rightarrow Y_n\}$ et chaque calcul tp de Gr, deux non-terminaux parmi $X_1, \dots, X_n$ ne peuvent apparaître avec le même entier de contrôle dans le i^{eme} composant de tp.

Théorème 3.1.12. *La jointure $i-j$ de deux langages générés par deux grammaires synchronisées est généré par une grammaire synchronisées si soit le i^{eme} composant de la première grammaire, soit le j^{eme} composant de la seconde grammaire est libre.*

La démonstration de ce théorème nécessite d'étendre la définition des TTSG en replaçant les entiers de contrôle par des n-uplets d'entiers de contrôle ce qui complique passablement le formalisme. De plus, la jointure de deux grammaires utilisant des n-plets de contrôle de tailles respectives n_1 et n_2 produit une grammaire qui utilise des n-uplets de contrôle de taille $n_1 + n_2 - 1$. Par conséquent, ce formalisme comportent deux inconvénients majeurs. Le premier est que bien qu'étant très proche des grammaire d'arbres, il n'est pas très intuitif. Le second est que chaque jointure engendre une complexification de la grammaire résultante au niveau du contrôle des synchronisations. C'est la raison pour laquelle d'autres formalismes ont été étudiés pour tenter de représenter la même classe de langages tout en évitant les inconvénients des TTSG.

3.2 Les systèmes de contraintes

L'idée de ce formalisme est de representer les langages de n-plets d'arbre par des variables ensemblistes, qui sont définies par une ensemble de contraintes du type $X \subseteq exp$ où exp est une expresion sur les ensembles de n-uplet d'arbres. L'avantage principal de ce formalisme est qu'il n'y pas plus besoin de la notion de contrôle ce qui permet de clarifier les résultats concernant la jointure de deux langages synchronisés.

3.2.1 Le formalisme

Les définitions qui vont être données dans cette section sont celles présentées dans [60]. Elles sont plus générales que les définitions initialement introduites dans [40] et [59] et permettent de faire le lien avec la programmation logique dans la section suivante.

Dans la suite, nous considèrerons $A \times (B \times C)$, $(A \times B) \times C$, et $A \times B \times C$ comme équivalents et nous omettrons les parenthèses. La notation $A_1 \times \cdots \times A_k$ représente $\mathcal{T}(\Sigma)^0 = \{()\}$ (l'ensemble contenant uniquement le zero-uplet) si $k = 0$, elle représente A_1 si $k = 1$, et $A_1 \times (A_2 \times \cdots \times A_k)$ dans les autres cas.

Un *template* est un terme clos qui peut aussi contenir des entiers positifs comme symbole de constante. Formellement, $\mathcal{T}(\Sigma \cup \mathbb{N})$ est l'ensemble des templates de Σ. Ici nous ne distinguons pas les entiers et les symboles qui les représentent. Par conséquent, suivant le contexte une variable i peut représenter à la fois un symbole de constante lorsque qu'elle apparaît dans un template et et l'entier correspondant lorsqu'elle apparaît dans une expression mathématique. Les entiers sont appelés *indices* et sont utilisés pour désigner un composant particulier d'un langage de n-uplets.

L'arité d'un objet O est noté $\mathrm{ar}(O)$, où O est soit un symbol de fonction ou de prédicat, une variable de langage, ou une expression sur les langages.

Soient A un langage n-uplets, et $\Box$ un k-uplet de templates où l est le plus grand indice apparaissant dans $\Box$, où 0 s'il n'en contient aucun. Les opérations de *construction* et de *filtrage* sont définies comme suit :

$$\begin{array}{rcl} \Box \circ A & \stackrel{\text{def}}{=} & \{\, \Box\{1 \mapsto t_1, \ldots, l \mapsto t_l\} \mid (t_1, \ldots, t_{n+l}) \in A \times \mathcal{T}(\Sigma)^l \,\} \\ A/\Box & \stackrel{\text{def}}{=} & \{\, (t_1, \ldots, t_l) \in \mathcal{T}(\Sigma)^l \mid \Box\{1 \mapsto t_1, \ldots, l \mapsto t_l\} \in A \,\} \end{array}$$

On peut remarquer que l'opération de construction remplace les indices de $\Box$ qui sont plus grands que l'arité de A par des termes clos arbitraires. Les l produits cartésiens de $\mathcal{T}(\Sigma)$ permettent d'assurer que l'arité de $A \times \mathcal{T}(\Sigma)^l$ n'est pas inférieure à l, même si A est le langage d'arité 0. Construction et filtrage sont en quelque sorte l'inverse l'un de l'autre : si l'arité de A est égal au plus grand indice de $\Box$, c'est à dire $n = l$, et si $\Box$ contient tous les indices de 1 à l alors $(\Box \circ A)/\Box = A$.

Exemple 3.2.1. Soient $\Sigma = \{a/0, f/1, g/2\}$, $A = \{(a, f(a)), (f(a), a)\}$, et $\Box = [g(1, 3), 2]$ un couple de templates. On a :

$$\begin{array}{rcl} \Box \circ A & = & \{\, (g(a, t), f(a)) \mid t \in \mathcal{T}(\Sigma) \,\} \cup \{\, (g(f(a), t), a) \mid t \in \mathcal{T}(\Sigma) \,\} \\ A/\Box & = & \emptyset \\ (\Box \circ A)/\Box & = & \{\, (a, f(a), t) \mid t \in \mathcal{T}(\Sigma) \,\} \cup \{\, (f(a), a, t) \mid t \in \mathcal{T}(\Sigma) \,\} \end{array}$$

Pour être capable de spécifier des relations intéressantes par des langages de n-uplets d'arbres, nous avons besoin d'un mécanisme itératif ou récursif. L'idée est d'utiliser les définitions inductives avec des relations d'inclusion sur des expressions de langages comme cadre général.

Soit $\mathcal{X}$ un ensemble de variables de langages avec arité. L'ensemble des *expressions de langages* est défini par la grammaire suivante

$$e \quad ::= \quad A \mid \{()\} \mid (e \times e) \mid (\Box \circ e) \mid (e/\Box)$$

où A est une variable de langage et $\Box$ est un n-uplet de templates.

Une interpretation λ est une application associant un langage de n-uplets d'arbres à chaque variable de langage d'arité n. Elle s'étend naturellement à une interprétation sur les expressions de langage :

$$\begin{array}{rcl} \lambda(\{()\}) & = & \{()\} \\ \lambda(e_1 \times e_2) & = & \lambda(e_1) \times \lambda(e_2) \\ \lambda(\Box \circ e) & = & \Box \circ \lambda(e) \\ \lambda(e/\Box) & = & \lambda(e)/\Box \end{array}$$

Une interprétation, λ, est plus petite ou égale à une autre, λ', si $\lambda(A) \subseteq \lambda'(A)$ pour tout $A \in \mathcal{X}$.

Une *définition inductive* est un ensemble de contraintes de la forme $A \supseteq e$ tel que les arités de A et e coïncident.[1] Les langages définis par une définition inductive D sont donnés par la plus petite interprétation λ telle que $\lambda(A) \supseteq \lambda(e)$ pour toutes les contraintes ; elle est définie de manière unique et peut être obtenue grâce au plus petit point fixe des contraintes. On note $\mathcal{L}_\mathcal{D}$ la plus petite interprétation ou simplement par $\mathcal{L}$ si la définition inductive peut être déduite du contexte.

Exemple 3.2.2. Les opérations sur les langages de n-uplets d'arbres comme l'union, l'intersection, la jointure ou la projection peuvent s'exprimer en termes d'expressions de langages. De plus, le test d'appartenance peut se réduire à un test du vide :

$$\begin{array}{rcl} \{\vec{t}_1, \ldots, \vec{t}_n\} & = & \vec{t}_1 \circ \{()\} \cup \cdots \cup \vec{t}_n \circ \{()\} \\ e \cup f & = & A \qquad \text{avec les contraintes } A \supseteq e \text{ et } A \supseteq f \\ e \cap f & = & (e \times f)/[1, \ldots, m, 1, \ldots, n] \qquad \text{pour } m = n \\ e \bowtie_{i,j} f & = & (e \times f)/[1, \ldots, m+j-1, i, m+j, \ldots, m+n-1] \\ e \bowtie_{i,n} f & = & (e \times f)/[1, \ldots, m+n-1, i] \\ \Pi_{i_1,\ldots,i_k} e & = & [i_1, \ldots, i_k] \circ e \\ \vec{t} \in \mathcal{L}(A) & \equiv & \mathcal{L}(A') \neq \emptyset \qquad \text{avec la contrainte } A' \supseteq A/\vec{t} \end{array}$$

où $\vec{t}, \vec{t}_1, \ldots, \vec{t}_n$ représente des n-uplets de termes ou de templates sans indices, e et f sont des expressions de langages d'arité respective m et n et $1 \leq i, i_1, \ldots, i_k \leq m$, $1 \leq j < n$.

Exemple 3.2.3. Soit $\Sigma = \{a/0, f/2\}$ et $\mathcal{X} = \{IdB/2, Sym/2\}$. Les contraintes

$$\begin{array}{rcl} IdB & \supseteq & \{(a,a)\} \cup [f(1,3), f(2,4)] \circ (IdB \times IdB) \\ Sym & \supseteq & \{(a,a)\} \cup [f(1,3), f(4,2)] \circ (Sym \times Sym) \end{array}$$

définit les langages

$$\begin{array}{rcl} \mathcal{L}(IdB) & = & \{\, (t,t) \mid t \in \mathcal{T}(\Sigma) \,\} \\ \mathcal{L}(Sym) & = & \{\, (t,t') \mid t \in \mathcal{T}(\Sigma),\ t' \text{ est l'arbre symétrique de } t \,\} \ . \end{array}$$

Les langages définissables par des définitions inductives sont exactement les langages de n-uplets d'arbres récursivement énumérables ; il y a une traduction directe des contraintes vers les clauses de Horn et vice versa (voir section 3.2.2). L'inconvénient d'une telle expressivité est que les propriétés comme les test de vide ou d'appartenance sont indécidables. Par conséquent on restreint notre formalisme afin d'exhiber des sous-classes intéressantes.

Les *systèmes de contraintes* ont été introduits dans [40] comme une version simplifiée des grammaires synchronisées. On peut les définir comme des définitions inductives sans opérations de filtrage. Dans ce cas, toutes les contraintes peuvent être normalisées sous la forme suivante $A \supseteq \Box \circ (A_1 \times \cdots \times A_k)$ pour les variables de langages $A, A_1, \ldots, A_k$.

[1] L'arité d'une expression de langage est le nombre de composants du langage de n-uplets qu'il définit ; elle peut être déterminée facilement car l'arité des variables de langage est donnée.

$$\{()\} \rightsquigarrow \langle (), \{\} \rangle$$

$$A \rightsquigarrow \langle (x_1, \ldots, x_{\mathrm{ar}(A)}), \{P_A(x_1, \ldots, x_{\mathrm{ar}(A)})\} \rangle$$
où $x_1, \ldots, x_{\mathrm{ar}(A)}$ sont des variables 2 à 2 distinctes.

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m), \mathcal{G} \rangle \quad f \rightsquigarrow \langle (t_1, \ldots, t_n), \mathcal{H} \rangle}{(e \times f) \rightsquigarrow \langle (s_1, \ldots, s_m, t_1, \ldots, t_n), \mathcal{G} \cup \mathcal{H} \rangle}$$
si les prémisses n'ont pas de variables communes.

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m), \mathcal{G} \rangle}{(\Box \circ e) \rightsquigarrow \langle \Box\{1 \mapsto s_1, \ldots, m \mapsto s_m, m{+}1 \mapsto x_1, m{+}2 \mapsto x_2, \ldots\}, \mathcal{G} \rangle}$$
où $x_1, x_2, \ldots$ sont des variables 2 à 2 distinctes n'apparaissant pas dans les premisses.

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m), \mathcal{G} \rangle}{(e/\Box) \rightsquigarrow \langle (x_1, \ldots, x_l)\mu, \mathcal{G}\mu \rangle}$$
si μ, le mgu de $(s_1, \ldots, s_m)$ et $\Box\{1 \mapsto x_1, \ldots, l \mapsto x_l\}$, existe, où l désigne l'indice maximal apparaissant dans $\Box$ ou 0 si $\Box$ n'en contient pas, et $x_1, \ldots, x_l$ sont des variables 2 à 2 distinctes n'apparaissant pas dans les prémisses.

FIG. 3.1 – Conversion des expressions de langages en clauses de Horn

3.2.2 Correspondance avec les clauses de Horn

Dans cette section, nous définissons une traduction des expressions de langages vers des programmes logiques équivalents et vice versa. Ce qui nous permettra dans la section suivante de montrer comment utiliser les techniques de transformations de programmes logiques pour résoudre les problèmes de décidabilité et de clôture des langages de n-uplets d'arbres.

Pour toute variable de langage A d'arité n, on note P_A le symbole de prédicat d'arité n associé de manière unique à A. On définit la clause correspondant à une contrainte de la manière suivante :

$$\mathrm{horn}(A \supseteq e) \stackrel{\mathrm{def}}{=} \begin{cases} \{P_A(s_1, \ldots, s_n) \leftarrow \mathcal{G}\} & \text{si } e \rightsquigarrow \langle (s_1, \ldots, s_n), \mathcal{G} \rangle \\ \{\} & \text{sinon} \end{cases}$$

La définition de la relation $\rightsquigarrow$ est donnée par la figure 3.1. Pour une définition inductive $\mathcal{D}$, c'est à dire pour un ensemble de contraintes, on définit le programme logique correspondant par $\mathrm{horn}(\mathcal{D}) \stackrel{\mathrm{def}}{=} \bigcup_{C \in \mathcal{D}} \mathrm{horn}(C)$.

Exemple 3.2.4. Soit $\Sigma = \{a/0, s/1\}$. Les contraintes

$$\begin{array}{lcl@{\qquad}lcl} X & \supseteq & [a, 1, 1] \circ \{()\} & X & \supseteq & [s(1), 2, s(3)] \circ X \\ Y & \supseteq & [a, 1, a] \circ \{()\} & Y & \supseteq & [s(4), 1, 3] \circ ((X \times Y)/[1, 2, 3, 4, 1, 2]) \end{array}$$

définissent les langages

$$\begin{array}{lcl} \mathcal{L}(X) & = & \{\, (s^m(a), s^n(a), s^{m+n}(a)) \mid m, n \geq 0 \,\} \\ \mathcal{L}(Y) & = & \{\, (s^m(a), s^n(a), s^{m*n}(a)) \mid m, n \geq 0 \,\} \end{array}$$

La traduction en clauses de Horn donne le résultat suivant :

$$\begin{array}{l@{\qquad}l} P_X(a, x_1, x_1) \leftarrow & P_X(s(x_1), x_2, s(x_3)) \leftarrow P_X(x_1, x_2, x_3) \\ P_Y(a, x_1, a) \leftarrow & P_Y(s(x_4), x_1, x_3) \leftarrow P_X(x_1, x_2, x_3), P_Y(x_4, x_1, x_2) \end{array}$$

La transformation est détaillée dans la figure 3.2.

Pour chaque symbole de prédicat P d'arité n, on associe de manière unique la variable de langage A_P d'arité n. Soit σ un substitution donnée qui remplace chaque variable du premier

$$\frac{\{()\} \rightsquigarrow \langle (),\ \{\}\rangle}{(a,1,1)\circ\{()\} \rightsquigarrow \langle (a,x_1,x_1),\ \{\}\rangle} \qquad \frac{\{()\} \rightsquigarrow \langle (),\ \{\}\rangle}{(a,1,a)\circ\{()\} \rightsquigarrow \langle (a,x_1,a),\ \{\}\rangle}$$

$$\frac{X \rightsquigarrow \langle (x_1,x_2,x_3),\ \{P_X(x_1,x_2,x_3)\}\rangle}{(s(1),2,s(3))\circ X \rightsquigarrow \langle (s(x_1),x_2,s(x_3)),\ \{P_X(x_1,x_2,x_3)\}\rangle}$$

$$\cfrac{\cfrac{\cfrac{X \rightsquigarrow \langle (x_1,x_2,x_3),\ \{P_X(x_1,x_2,x_3)\}\rangle \qquad Y \rightsquigarrow \langle (x_4,x_5,x_6),\ \{P_Y(x_4,x_5,x_6)\}\rangle}{X\times Y \rightsquigarrow \langle (x_1,x_2,x_3,x_4,x_5,x_6),\ \{P_X(x_1,x_2,x_3), P_Y(x_4,x_5,x_6)\}\rangle}}{X\times Y/[1,2,3,4,1,2] \rightsquigarrow \langle (x_1,x_2,x_3,x_4,x_1,x_2),\ \{P_X(x_1,x_2,x_3), P_Y(x_4,x_1,x_2)\}\rangle}}{[s(4),1,3]\circ(X\times Y/[1,2,3,4,1,2]) \rightsquigarrow \langle (s(x_4),x_1,x_3),\ \{P_X(x_1,x_2,x_3), P_Y(x_4,x_1,x_2)\}\rangle}$$

FIG. 3.2 – Conversion des expressions de l'exemple 3.2.4 en clause de Horn

ordre par un entier unique, c'est à dire $\sigma = \{x_1 \mapsto 1, x_2 \mapsto 2, \ldots\}$. On définit la contrainte correspondant à une clause de Horn donnée par

$$\begin{aligned}&\text{constraint}(\,P(s_1,s_2,\ldots) \leftarrow P_1(t_{11},t_{12},\ldots), P_2(t_{21},t_{22},\ldots),\ldots\,) \stackrel{\text{def}}{=} \\ &\quad A_P \supseteq [s_1\sigma, s_2\sigma,\ldots]\circ\big((A_{P_1}\times A_{P_2}\times\cdots)\,/\,[t_{11}\sigma,t_{12}\sigma,\ldots,t_{21}\sigma,t_{22}\sigma,\ldots]\big)\end{aligned}$$

et la définition inductive correspondant au programme logique $\mathcal{P}$ par

$$\text{indef}(\mathcal{P}) \stackrel{\text{def}}{=} \bigcup_{C\in\mathcal{P}}\{\text{constraint}(C)\} \ .$$

La proposition suivante établit que les définitions inductives et les programmes logiques définissent les mêmes langages de n-uplets d'arbres et que les transformations données précédemment préservent l'équivalence des langages générés.

Proposition 3.2.5. *Soient $\mathcal{D}$ une définition inductive et $\mathcal{P}$ un programme logique.*
(a) $\mathcal{L}_{\mathcal{D}}(A) = \mathcal{M}(\text{horn}(\mathcal{D}))|_{P_A}$ pour toute variable de langage A.
(b) $\mathcal{M}(\mathcal{P})|_P = \mathcal{L}_{\text{indef}(\mathcal{P})}(A_P)$ pour tout symbole de prédicat P.

On peut déduire de cette proposition que les langages de n-uplets d'arbres définissables par des définitions inductives sont exactement les langages récursivement énumérables.

Définition 3.2.6. Une clause $H \leftarrow B_1,\ldots,B_k$ est une *cs-clause* si son corps est linéaire et ne contient pas de symbole de fonction.

Un programme logique est un *cs-programme* ssi toutes ses clauses sont des cs-clauses.

On peut facilement montrer la correspondance des cs-programmes avec les systèmes de contraintes de [40] définis dans la section précédente.

3.3 Les programmes logiques

Nous avons donc défini un classe de programmes logiques, les cs-programmes qui permettent de définir la même classe de langages que les systèmes de contraintes de [40] qui eux-mêmes sont équivalents au grammaires synchronisées de [55]. Le principal avantage d'utiliser le formalisme de la programmation logique pour représenter les langages de n-uplets d'arbres par rapport aux deux autres présentées, est qu'il est très largement répendu dans la communauté de la recherche en informatique. Un second avantage, nous le verrons dans la suite de ce document, est que c'est un formalisme de haut-niveau qui permet d'assouplir les restrictions syntaxiques des programmes logiques permettant de reconnaître un classe de langages donnée et par conséquent de faciliter l'usage des langages de n-uplets d'arbres dans les différents domaines où ils sont utiles.

Par contre, nous n'avons pour l'instant pas défini de moyen pour calculer les opérations sur les langages, ni pour décider les tests d'appartenance ou du vide. C'est l'objet de la section suivante qui présente des règles de transformation de programmes logiques basées sur les techniques de pliages/dépliages [77]. Nous verrons que ces règles permettent de transformer un programme logique en un cs-programme équivalent. Mais avant cela nous allons montrer la clôture de langages décrits par des cs-programmes par des opérations simples.

3.3.1 Quelques résultats de clôture

Théorème 3.3.1. *Les langages synchronisés sont clos par union, restriction, projection, cylindrification, permutation et produit cartésien.*

Preuve. Soient P_1 et P_2 deux prédicats de même arité définis par des cs-programmes et $\mathcal{L}_1$ et $\mathcal{L}_2$ les deux langages décrits par ces prédicats.

- $\mathcal{L}_1 \cup \mathcal{L}_2$ est décrit par le prédicats définis par l'union des cs-programmes définissant P_1 et P_2 à laquelle on ajoute les deux clauses suivantes $P_\cup(\vec{x}) \leftarrow P_1(\vec{x})$ et $P_\cup(\vec{x}) \leftarrow P_2(\vec{x})$.
- $\mathcal{L}_1 \backslash_i$ est défini par le programme définissant P_1 auquel on ajoute la clause $P_\backslash(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \leftarrow P_1(x_1 \ldots, x_n)$.
- $\mathcal{P}_{i_1,\ldots,i_k}(\mathcal{L}_1)$ est défini par le programme définissant P_1 auquel on ajoute la clause $P_\mathcal{P}(x_{i_1}, \ldots, x_{i_k}) \leftarrow P_1(x_1 \ldots, x_n)$.
- $C_i(\mathcal{L}_1)$ est défini par le programme définissant P_1 auquel on ajoute la clause $P_C(x_1, \ldots, x_i, y, x_{i+1}, \ldots, x_n) \leftarrow P_1(x_1 \ldots, x_n)$ avec y différent de $x_1, \ldots, x_n$.
- $\sigma(L_1)$ est défini par le programme définissant P_1 auquel on ajoute la clause $P_\sigma(x_{\sigma(1)}, \ldots, x_{\sigma(n)}) \leftarrow P_1(x_1 \ldots, x_n)$.
- Si P_3 est un prédicat défini par un cs-programme et $\mathcal{L}_3$ le langage qu'il décrit, $\mathcal{L}_1 \times \mathcal{L}_3$ et défini par le programme composé de l'union des deux cs-programmes définissant P_1 et P_3 à laquelle on ajoute la clause $P_\times(\vec{x}, \vec{y}) \leftarrow P_1(\vec{x}), P_3(\vec{y})$.

On peut remarquer que le programme définissant chacune des opérations est bien un cs-programme puisqu'il est composé de cs-programmes auxquels on ajoute une ou deux cs-clauses.

3.3.2 Transformer un programme logique en un cs-programme

Cette section présente des règles de transformations, *Definition Unfolding*, *Clause Unfolding* et *Definition Introduction*, qui transforment un programme logique en un ensemble de cs-clauses équivalent ; ce sont des instances des règles étudiées dans le domaine de la transformation de programmes logiques [77].

Définition 3.3.2. Une *définition* est une équivalence $H \leftrightarrow \mathcal{B}$ où H est un atome plat et $\mathcal{B}$ un ensemble d'atomes. H est appelée la *tête* de la définition et $\mathcal{B}$ son *corps*. Les variables qui apparaissent dans le corps de la définition mais pas dans la tête sont appelées *variables existentielles*.

Soient $\mathcal{P}$ un programme logique et $\mathcal{D}$ un ensemble de définition. On dit que $\mathcal{D}$ est *compatible avec* $\mathcal{P}$ si pour toute définition $P(\vec{x}) \leftrightarrow \mathcal{B}$ soit P n'apparaît ni dans $\mathcal{B}$ ni dans $\mathcal{P} \cup \mathcal{D} \setminus \{P(\vec{x}) \leftrightarrow \mathcal{B}\}$ soit $\mathcal{B} = P(\vec{x})$.

Les règles transforment des états $\langle \mathcal{P}, \mathcal{D}_{\text{new}}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}} \rangle$ où $\mathcal{P}$ est un programme logique qui ne change pas au cours du processus, $\mathcal{D}_{\text{new}}$ sont les définitions pas encore dépliées, $\mathcal{D}_{\text{done}}$ sont les définitions déjà traitées mais qui peuvent encore être utilisées pour simplifier des clauses, $\mathcal{C}_{\text{new}}$ sont les clauses générées à partir des définitions par dépliage et $\mathcal{C}_{\text{out}}$ sont les clauses du programme résultat.

On note $S \Rightarrow S'$ le fait que S' est un état obtenu à partir de l'état S par l'application de l'une des trois règles *Definition Unfolding*, *Clause Unfolding* ou *Definition Introduction* (leur description est donnée ci-après). La clôture réflexive transitive de $\Rightarrow$ est notée $\stackrel{*}{\Rightarrow}$. Un *état initial* est de la forme $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle$ où $\mathcal{D}$ est compatible avec $\mathcal{P}$; un *état final* est de la forme $\langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$. $\mathcal{P}$ et $\mathcal{D}$ sont appelés *entrées* de la dérivation, $\mathcal{P}'$ sa *sortie*. Une dérivation est dite *complète* si son dernier état est un état final.

Definition Unfolding. Cette règle consiste à prendre une définition non encore traitée, sélectionner un ou plusieurs atomes de son corps suivant une règle de sélection et les déplier avec toutes les clauses possibles du programme d'entrée. Formellement :

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}} \,\dot\cup\, \{L \leftrightarrow \mathcal{R} \dot\cup \{A_1, \ldots, A_k\}\},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}} \cup \{L \leftrightarrow \mathcal{R} \cup \{A_1, \ldots, A_k\}\},\, \mathcal{C}_{\text{new}} \cup \mathcal{C},\, \mathcal{C}_{\text{out}} \rangle}$$

où $\mathcal{C}$ est l'ensemble de toutes les clauses $(L \leftarrow \mathcal{R} \cup \mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k)\mu$ telles $H_i \leftarrow \mathcal{B}_i$ est une clause de $\mathcal{P}$ pour $i = 1, \ldots, k$, et telles que l'unificateur le plus général μ de $(A_1, \ldots, A_k)$ et $(H_1, \ldots, H_k)$ existe. Notons que les clauses de $\mathcal{P}$ doivent être renommées de telle manière qu'elles n'aient de variables communes ni entre elles ni avec $L \leftrightarrow \mathcal{R} \cup \{A_1, \ldots, A_k\}$.

Clause Unfolding. Cette règle est très similaire à la précédente sauf qu'elle déplie une clause de $\mathcal{C}_{\text{new}}$ au lieu d'une définition de $\mathcal{D}_{\text{new}}$. Formellement :

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}} \,\dot{\cup}\, \{L \leftarrow \mathcal{R} \dot{\cup} \{A_1, \ldots, A_k\}\},\, \mathcal{C}_{\text{out}} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}} \cup \mathcal{C},\, \mathcal{C}_{\text{out}} \rangle}$$

où $\mathcal{C}$ est l'ensemble de toutes les clauses $(L \leftarrow \mathcal{R} \cup \mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k)\mu$ telles $H_i \leftarrow \mathcal{B}_i$ est une clause de $\mathcal{P}$ pour $i = 1, \ldots, k$, et telles que l'unificateur le plus général μ de $(A_1, \ldots, A_k)$ et $(H_1, \ldots, H_k)$ existe. Notons que les clauses de $\mathcal{P}$ doivent être renommées de telle manière qu'elles n'aient de variables communes ni entre elles ni avec $L \leftrightarrow \mathcal{R} \cup \{A_1, \ldots, A_k\}$.

Définition 3.3.3. Soient $H \leftarrow \mathcal{B}$ une clause, $\mathcal{B}'$un sous-ensemble de $\mathcal{B}$ tel que $Var(\mathcal{B}') \cap Var(\mathcal{B} - \mathcal{B}') = \emptyset$, et $L \leftrightarrow \mathcal{R}$ une définition. La définition $L \leftrightarrow \mathcal{R}$ *coïncide avec* $\mathcal{B}'$, s'il existe un renommage de variables η tel que $\mathcal{R}\eta = \mathcal{B}'$ et $Var(\mathcal{B}') \setminus Var(H) = Var(\mathcal{R}\eta) \setminus Var(L\eta)$ (c'est à dire que les variables existentielles de la définition $L \leftrightarrow \mathcal{R}$ coïncident à celles de la clause). On dit alors que la définition *coïncide avec* $\mathcal{B}'$ *via* η.

Exemple 3.3.4. Soient $P(f(x, z)) \leftarrow A(x, y), B(y), C(z)$ une clause et

$$\begin{array}{rclcl} D_1 & = & Q & \leftrightarrow & A(u, v), B(v) \\ D_2 & = & Q(u) & \leftrightarrow & A(u, v), B(v) \\ D_3 & = & Q(u, v) & \leftrightarrow & A(u, v), B(v) \end{array}$$

des définitions. Prenons l'ensemble d'atomes $\mathcal{B}' = A(x, y), B(y)$ du corps de la clause. $\mathcal{B}'$ n'a aucune variable en commun avec le reste du corps de la clause et contient une variable existentielle (y n'apparaît pas dans la tête). Le renommage de variables $\eta = \{u \mapsto x,\ v \mapsto y\}$ rend les corps des trois définitions égaux à $\mathcal{B}'$, mais seule la définition D_2 coïncide avec $\mathcal{B}'$, car la seule variable existentielle de D_2 v se transforme en y par η. Dans la définition D_1 toutes les variables sont existentielles, alors que D_3 ne contient aucune variable existentielle.

Definition Introduction. Cette règle consiste à prendre une clause non encore traitée, décomposer son corps en composants minimaux sans variables communes et remplacer chaque composant qui n'est pas réduit à un seul atome linéaire plat par un atome plat qui est soit la tête d'une définition déjà traitée (si le corps de cette définition coïncide avec le composant considéré) soit construit à partir d'un nouveau symbole de prédicat et des variables du composant. Dans ce cas on introduit une nouvelle définition associant le nouveau symbole de prédicat avec le composant. Formellement :

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}} \,\dot{\cup}\, \{H \leftarrow \mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_k\},\, \mathcal{C}_{\text{out}} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}} \cup \mathcal{D},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \cup \{H \leftarrow L_1, \ldots, L_k\} \rangle}$$

où $\mathcal{B}_1, \ldots, \mathcal{B}_k$ est la composition maximale de $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ en sous ensembles non vides sans variables communes,

$$L_i = \begin{cases} L\eta & \text{si } \mathcal{D}_{\text{done}} \text{ contient une définition } L \leftrightarrow \mathcal{R} \text{ coïncidant avec } \mathcal{B}_i \text{ via } \eta \\ P(x_1, \ldots, x_n) & \text{sinon, où } P \text{ est un nouveau symbole de prédicat} \\ & \text{et } \{x_1, \ldots, x_n\} = Var(\mathcal{B}_i) \cap Var(H). \end{cases}$$

Pour $1 \leq i \leq k$, et $\mathcal{D}$ étant l'ensemble de toutes les nouvelles définitions $P(x_1, \ldots, x_n) \leftrightarrow \mathcal{B}_i$.

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$R(x,z) \leftrightarrow$ $M(x,y,z), E(y)$			U_M
	$R(x,x) \leftarrow E(a)$ $R(s(x),z) \leftarrow$ $M(x,y,z), E(s(y))$		D
$E' \leftrightarrow E(a)$	$R(s(x),z) \leftarrow$ $M(x,y,z), E(s(y))$	$R(x,x) \leftarrow E'$	U_E
	$E' \leftarrow$ $R(s(x),z) \leftarrow$ $M(x,y,z), E(s(y))$		D
	$R(s(x),z) \leftarrow$ $M(x,y,z), E(s(y))$	$E' \leftarrow$	D
$R'(x,z) \leftrightarrow$ $M(x,y,z), E(s(y))$		$R(s(x),z) \leftarrow$ $R'(x,z)$	U_E
	$R'(x,z) \leftarrow$ $M(x,s(y),z), E(y)$		D
$R''(x,z) \leftrightarrow$ $M(x,s(y),z), E(y)$		$R'(x,z) \leftarrow$ $R''(x,z)$	U_M
	$R''(s(x),z) \leftarrow$ $M(x,z), E(y)$		D
		$R''(s(x),z) \leftarrow$ $R(x,z)$	

FIG. 3.3 – Transformation en un cs-programme du programme de l'exemple 3.3.5

Exemple 3.3.5. Soit $\mathcal{P}$ le cs-programme

$$\begin{array}{ll} M(x,a,x) \leftarrow & E(a) \leftarrow \\ M(s(x),s(y),z) \leftarrow M(x,y,z) & E(s(s(x))) \leftarrow E(x) \end{array}$$

et $\mathcal{D}$ l'ensemble constitué de la seule définition $\{R(x,z) \leftrightarrow M(x,y,z), E(y)\}$.

Le prédicat M définit la soustraction, E la parité et R définit tous les couples d'entiers qui ont une différence paire. Remarquons que cette définition n'est pas une cs-clause car son corps n'est pas linéaire.

La figure 3.3 décrit une dérivation complète avec comme entrées $\mathcal{P}$ et $\mathcal{D}$. Nous avons omis $\mathcal{D}_{\text{done}}$ car il consiste simplement en l'ensemble des définitions apparaissant dans $\mathcal{D}_{\text{new}}$. De plus, on ne reporte qu'une seule fois les définitions et les clauses apparaissant dans les colonnes $\mathcal{D}_{\text{new}}$ et $\mathcal{C}_{\text{out}}$ lors de l'étape où elles apparaissent. Seule Definition Unfolding est utilisée (et pas Clause Unfolding), elle sélectionne l'atome de profondeur maximale le plus à gauche. La troisième colonne, $\mathcal{C}_{\text{out}}$, indique la liste des cs-clauses générées. La dernière colonne indique quelle règle a été utilisée : U_P signifie Definition Unfolding avec l'atome de profondeur maximale le plus à gauche avec les clause définissant le prédicat P, et D signifie Definition Introduction appliquée sur la première clause de $\mathcal{C}_{\text{new}}$.

Théorème 3.3.6 (Correction). *Soit $\mathcal{P}$ un programme logique et $\mathcal{D}$ un ensemble de définitions compatibles avec $\mathcal{P}$. Si $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle \stackrel{*}{\Rightarrow} \langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, alors $\mathcal{P}'$ est un cs-program vérifiant $\mathcal{M}(\mathcal{P}')|_P = \mathcal{M}(\mathcal{P} \cup \mathcal{D})|_P$ pour tout symbole de prédicat P définit par $\mathcal{D}$.*

Preuve. $\mathcal{P}'$ est un cs-programme par construction : les ensembles d'atomes des corps qui partagent des variables ou qui ne sont pas plats sont remplacés par un seul atome plat. L'équivalence sémantique des programmes $\mathcal{P} \cup \mathcal{D}$ et $\mathcal{P}'$ par rapport aux symboles de prédicats définis par $\mathcal{D}$ provient de la correction des règles de dépliages et d'introduction de définition en général (voir par exemple [77]).

Opération	Définition
La restriction $\mathcal{L}_1\backslash_i$	$P_{\backslash_i}(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_n) \leftrightarrow P_1(x_1,\ldots,x_n)$
La projection $\mathcal{P}_{i_1,\ldots,i_k}(\mathcal{L}_1)$	$P_{\mathcal{P}_{i_1,\ldots,i_k}}(x_{i_1},\ldots x_{i_k}) \leftrightarrow P_1(x_1,\ldots,x_n)$
La cylindrification $C_i(\mathcal{L}_1)$	$P_{C_i}(x_1,\ldots,x_i,y,x_{i+1},\ldots,x_n) \leftrightarrow P(x_1,\ldots,x_n)$
La permutation suivant σ $\sigma(L_1)$	$P_\sigma(x_{sigma(1)},\ldots,x_{\sigma(n)}) \leftrightarrow P_1(x_1,\ldots,x_n)$
Le produit cartésien $\mathcal{L}_1\times\mathcal{L}_2$	$P\times(\vec{x},\vec{y}) \leftrightarrow P_1(\vec{x}),P_2(\vec{y})$
La jointure $\mathcal{L}_1 \bowtie_{i,j} \mathcal{L}_2$	$P_\bowtie(\vec{x},y_1,\ldots,y_{j-1},y_{j+1},\ldots y_m) \leftrightarrow P_1(\vec{x}),P_2(\vec{y})$
L'intersection $\mathcal{L}_1\cap\mathcal{L}_2$	$P_\cap(\vec{x}) \leftrightarrow P_1(\vec{x}),P_2(\vec{x})$

FIG. 3.4 – Opérations sur le langage exprimées en logique

Remarque 3.3.7. Definition Introduction implique de tester si $\mathcal{D}_{\text{done}}$ contient des définitions ayant un corps égal à un renommage près aux composants $\mathcal{B}_i$. Ce test équivaut à vérifier si deux graphes sont isomorphes qui est un problème NP mais dont on conjecture qu'il n'est ni NP-complet ni n'est contenu dans P.

Remarque 3.3.8. Pour chaque état obtenu par dérivation à partir de l'état initial, $\mathcal{D}_{\text{new}}\cup\mathcal{D}_{\text{done}}$ est compatible avec $\mathcal{P}\cup\mathcal{C}_{\text{new}}$. Cela signifie en particulier que les nouveaux symboles de prédicat introduits au cours de la dérivation n'apparaissent dans aucun corps de définitions ou de clauses de $\mathcal{D}_{\text{new}}$, $\mathcal{D}_{\text{done}}$ et $\mathcal{C}_{\text{new}}$.

Remarque 3.3.9. Chaque clause de $\mathcal{C}_{\text{new}}$ est sans variable existentielle, en effet pour dans la règle Definition Introduction, on ne retient pour chaque composant que les variables qui apparaissent dans la tête de la clause traitée pour fabriquer les nouvelles définitions.

Remarque 3.3.10. Les cs-programmes générés par les $\Rightarrow$-dérivations peuvent contenir des clauses inproductives c'est à dire des clauses qui ne contribuent pas aux modèles minimaux. De telles clauses peuvent s'éliminer grâce à la procédure suivante. En partant des faits, marquer chacune des clauses $P(\vec{s}) \leftarrow Q_1(\vec{t_1}),\ldots,Q_k(\vec{t_k})$ comme productive et son symbole de prédicat P comme non vide si tous les Q_i ont été marqués comme non vides avant. En répétant ce processus jusqu'à ce qu'aucune clause et aucun symbole de prédicat ne soient marqués. Les clauses non marquées peuvent être retirées du programme sans changer sa sémantique. Cette élimination des clauses improductives est suffisant pour tester le vide du modèle minimal : l'ensemble de clauses est vide après l'élimination des clauses improductives si et seulement si le modèle minimal de Herbrand est vide. On peut en déduire la décidabilité du test du vide des systèmes de contraintes.

Voici maintenant comment on peut utiliser ces règles de transformation pour atteindre le but fixé, c'est à dire avoir un mécanisme opérationnel pour manipuler les langages de n-uplets d'arbres représentés par des programmes logiques.

Tout d'abord on peut tenter de transformer un programme logique quelconque $\mathcal{P}$ en un cs-programme équivalent. Pour cela il suffit de mettre en entrées le programme $\mathcal{P}$ et l'ensemble des tautologies $P(\vec{x}){\leftrightarrow}P(\vec{x})$ telles que P apparaît dans $\mathcal{P}$. La sortie de toute dérivation complète est un cs-programme équivelant à $\mathcal{P}$ sur tous ses prédicats.

Ensuite, si on considère un cs-programme $\mathcal{P}$ définissant les prédicats P_1 et P_2 définissant les langages $\mathcal{L}_1$ et $\mathcal{L}_2$, on peut tester l'appartenance d'un n-uplet $(t_1,\ldots,t_n)$ au langage $\mathcal{L}_1$ en donnant en entrées $\mathcal{P}$ et la définition $Appartient \leftrightarrow P_1(t_1,\ldots,t_n)$. Si le résultat d'une dérivation complète partant de ces entrées donne un programme vide (après élimination des clauses improductives) le n-uplet n'appartient pas à $\mathcal{L}_1$ sinon il lui appartient. On peut généraliser le test d'appartenance à des n-uplets non clos. La signification du test devient alors, existe-t-il une instance close de ce n-uplet qui appartient au langage $\mathcal{L}_1$?

La figure 3.4 donne pour chaque opération sur les langages de n-uplets la définition à donner en entrée à $\Rightarrow$ pour obtenir le cs-programme qui définit le langage résultat.

L'union et le complément sont traités un peu différemment. L'union peut s'exprimer en ajoutant simplement les deux clauses suivantes au programme initial : $P_\cup(\vec{x}) \leftarrow P_1(\vec{x})$ et $P_\cup(\vec{x}) \leftarrow P_2(\vec{x})$. Le complément lui nécessite le calcul du programme complété dans l'esprit de [18].

Bien sûr pour que ces opérations soient effectivement calculées par $\Rightarrow$ en donnant un cs-programme définissant le langage résultat, il faut être capable de déterminer une stratégie garantissant la terminaison du processus. Etant donné que les langages synchronisés ne sont pas clos par intersection ou jointure, ce ne sera pas toujours possible de définir une telle stratégie. Dans les sections suivantes nous définissons des classes de programmes logiques et d'ensembles de définitions pour lesquelles on peut garantir la terminaison de $\Rightarrow$.

3.3.3 Les jointures régulières

Dans cette section, nous présentons une classes de problèmes de jointures entre un langage représenté par un cs-programme et des langages réguliers d'arbres. Ce résultat nous permettra de définir un test d'appartenance assez intéressant car il permet de tester l'appartenance de n-uplets de termes clos mais aussi de tester s'il existe une instance close d'un n-uplet de termes qui appartient au langage représenté par un cs-programme. Ce même résultat nous permettra même de représenter le langage des instances closes du n-uplets qui appartiennent au langage testé.

Avant de présenter ce résultat nous avons besoin de définir un sous-classe de cs-programmes qui correspond aux langages réguliers d'arbres (c'est à dire de mono-uplets d'arbres).

Définition 3.3.11. Un programme logique est dit *monadique* s'il ne contient que des symboles de prédicat unaires. Un prédicat défini par un cs-programme linéaire monadique est dit *linéaire monadique.*

Le lemme suivant fait le lien entre les langages réguliers d'arbres et les cs-programmes linéaires monadiques.

Lemme 3.3.12. *Un ensemble de termes clos S est un langage régulier d'arbres ssi il existe un cs-programme linéaire monadique $\mathcal{P}$ tel que S est décrit par un prédicat P de $\mathcal{P}$.*

Preuve. Si S est un langage régulier d'arbres, il peut être généré par une grammaire régulière $\langle NT, \Sigma, I, \Delta\rangle$, où les productions de Δ sont des éléments de $NT \times \mathcal{T}(\Sigma, NT)$. On peut transformer cette grammaire en un programme logique en représentant chaque non-terminal par un symbole de prédicat unaire et les productions $A \Rightarrow C[X_1, \ldots, X_n]$ par les clauses $A(C[x_1, \ldots, x_n]) \leftarrow X_1(x_1), \ldots, X_n(x_n)$. Ces clauses sont des cs-clauses linéraires monadiques.

Inversement, Si $\mathcal{P}$ est un cs-programme linéaire monadique, on peut représenter chaque symbole de prédicat par un non-terminal et chaque clause $P(C[x_1, \ldots, x_n]) \leftarrow \mathcal{B}$ peut être transformée en la production $P \Rightarrow C[P_1, \ldots, P_n]$, où P_i est le symbole de prédicat correspondant à l'atome $P_i(x_i)$ si $P_i(x_i)$ appartient à $\mathcal{B}$, ou $P_i = P^*$ sinon (P^* est un non-terminal qui génère tous les termes de $\mathcal{T}(\Sigma)$).

Dans les deux cas une induction sur la longueur des dérivations montre que les deux formalismes décrivent le même langage.

Corollaire 3.3.13. *Chaque cs-programme linéaire monadique a un équivalent pour lequel chaque tête de clause est de profondeur exactement* 1.

Preuve. Chaque grammaire régulière a une grammaire régulière normalisée équivalente [21], c'est à dire que dans cette dernière toutes les productions sont de la forme $X \Rightarrow f(X_1, \ldots, X_n)$, où $X, X_1, \ldots, X_n$ sont des non-terminaux. Lorsqu'on transforme cette production en clause de Horn on obtient une clause dont la tête est de profondeur exactement 1.

Définition 3.3.14. Soit $\mathcal{P}$ un cs-programme. $P(\vec{x}) \leftrightarrow P'(\vec{t}) \cup \mathcal{B}_{reg}$ est appelée *définition de jointures régulières* compatible avec $\mathcal{P}$ si

- $P(\vec{x})$ est linéaire
- $P'(\vec{t})$ est linéaire
- les prédicats utilisés dans $\mathcal{B}_{reg}$ sont linéaires monadiques.

$P'(\vec{t})$ est appelé l'atome *clé*.

Nous montrons qu'une $\Rightarrow$-dérivation prenant en entrée un cs-programme et une définition de jointures régulières produit un cs-programme à condition d'utiliser une stratégie particulière pour appliquer les règles. Cette stratégie n'est pas la plus simple pour garantir la terminaison de la $\Rightarrow$-dérivation mais nous le verrons plus tard elle permet de préserver en un certain sens la régularité de certains arguments des prédicats dans le programme résultat. Voici donc la stratégie d'application des règles.

Une *stratégie préservant la liberté* pour les $\Rightarrow$-dérivations à partir d'une définition de jointure régulière consiste à restreindre l'application des trois règles de la manière suivante :

1. Appliquer Definition Introduction seulement sur les clauses $H \leftarrow \mathcal{B}$ de $\mathcal{C}_{\text{new}}$ telles que tous les atomes linéaires monadiques sont de profondeur 0.
2. L'atome sélectionné pour Definition Unfolding est en priorité un atome linéaire monadique de profondeur strictement supérieure à 0, s'il n'y en a pas alors l'atome clé est sélectionné.
3. L'atome sélectionné pour Clause Unfolding est un atome linéaire monadique de profondeur strictement supérieure à 0.

Exemple 3.3.15. Soient le cs-programme suivant :

$$\begin{array}{rcl} A(0,0,0) & \leftarrow & \\ A(s(c(x,y_1)),s(c(x,y_2)),c(x_3,y_3)) & \leftarrow & A(x,y_1,x_3),B(y_2,y_3) \\ B(s(x),s(y)) & \leftarrow & C(x),C(y) \\ C(0) & \leftarrow & \\ C(c(x,y)) & \leftarrow & C(x),C(y) \end{array}$$

et la définition de jointures régulières

$$P(z_1,z_2,z_3) \leftarrow A(s(z_1),s(z_2),z_3),C(z_1),C(z_2)$$

L'atome clé est $A(s(z_1),s(z_2),z_3)$. Cet atome est choisi par Definition Unfolding car toutes les atomes linéaires monadiques sont de profondeur 0. Dans la clause résultante

$$P(c(x,y_1),c(x,y_2),c(x_3,y_3)) \leftarrow A(x,y_1,x_3),B(y_2,y_3),C(c(x,y_1)),C(c(x,y_2))$$

$C(c(x,y_1))$ et $C(c(x,y_2))$ qui sont des atomes linéaires monadiques ont une profondeur de 1, par conséquent Definition Introduction ne peut pas encore s'appliquer. Clause Unfolding s'applique donc deux fois ce qui diminue la hauteur des atomes linéaires monadiques en donnant la clause

$$P(c(x,y_1),c(x,y_2),c(x_3,y_3)) \leftarrow A(x,y_1,x_3),B(y_2,y_3),C(x),C(y_1),C(y_2)$$

Maintenant, Definition Introduction peut s'appliquer.

Théorème 3.3.16. *Soit $\mathcal{P}$ un cs-programme et $P(\vec{x}) \leftarrow P'(\vec{t}),\mathcal{B}_{reg}$ une définition de jointures régulières compatible avec $\mathcal{P}$. Toute $\Rightarrow$-derivation partant de $P(\vec{x}) \leftarrow P'(\vec{t}),\mathcal{B}_{reg}$ en utilisant la stratégie préservant la liberté termine.*

Preuve. Soit $\mathcal{P}$ un cs-programme et $P(\vec{x}) \leftarrow P'(\vec{t}),\mathcal{B}_{reg}$ une définition de jointures régulières compatible avec $\mathcal{P}$. Sans perte de généralité, on suppose que les prédicats de $\mathcal{B}_{reg}$ sont définis par des programme dont la tête de ses clauses sont de profondeur exactement 1 (voir Corollaire 3.3.13).

Soit τ la profondeur maximale d'un atome apparaissant dans $\mathcal{P}$ et $P(\vec{x}) \leftarrow P'(\vec{t}),\mathcal{B}_{reg}$. On montre qu'une $\Rightarrow$-dérivation utilisant une stratégie préservant la liberté génère des définitions de jointures régulières dont les atomes clés sont de profondeur inférieure à τ dans $\mathcal{D}_{\text{new}}$ et les atomes linéaires monadiques sont de profondeur inférieure à $2 \times \tau$ et enfin que les atomes des corps de clauses de $\mathcal{C}_{\text{new}}$ qui ne sont pas linéaires monadiques, ne partagent jamais de variables entre eux et sont linéaires.

Cette propriété est vraie pour l'état initial. Considérons une dérivation utilisant une stratégie préservant la liberté $S_0 \Rightarrow^* S_n$ où S_n vérifie l'hypothèse d'induction.

Si la règle appliquée à la prochaine étape est Definition Unfolding sur $P(\vec{x}) \leftarrow P'(\vec{t}),\mathcal{B}_{reg}$, on a deux cas :

1. $\mathcal{B}_{reg}$ contient au moins un atome, $Q(f(\vec{t}))$, de profondeur supérieure à un. Definition Unfolding est donc appliquée sur $Q(f(\vec{t}))$ utilisant une clause de la forme $Q(f(\vec{x})) \leftarrow \mathcal{B}$. L'unificateur le plus général μ de $Q(f(\vec{t}))$ et $Q(f(\vec{x}))$ n'instancie aucune variable de $Q(f(\vec{t}))$, par conséquent $(P'(\vec{t}), \mathcal{B}_{reg})\mu = P'(\vec{t}), \mathcal{B}_{reg}$ d'une part et $Depth(Q(f(\vec{x}))\mu) = Depth(Q(f(\vec{t})))$ qui est strictement supérieur à $Depth(\mathcal{B}\mu)$ d'autre part donc la propriété sur la profondeur des atomes est préservée. De plus la nouvelle clause introduite dans $\mathcal{C}_{\text{new}}$ contient un seul atome non linéaire monadique et est linéaire donc il vérifie la propriété d'induction.
2. Si tous les atomes de $\mathcal{B}_{reg}$ ont une profondeur de 0, Definition Unfolding est appliquée sur l'atome clé $P'(\vec{t})$ utilisant la clause $P'(\vec{t'}) \leftarrow \mathcal{B}$. Comme $P'(\vec{t})$ est linéaire, d'après le lemme 2.2.3, les variables de $P'(\vec{t'})$ sont instanciées par des termes de profondeur inférieure à $\tau_{P'(\vec{t})} - \tau_{x,P'(\vec{t})}$ donc inférieure à τ ce qui signifie que $\mathcal{B}'\mu$ à une profondeur inférieure à τ. Par ailleurs, le corollaire 2.2.5 nous indique que $\tau_{x\mu} \leq \max(\tau_{P'(\vec{t})}, \tau_{P'(\vec{t'})} + \Delta Depth(P'(\vec{t})))$ donc $\tau_{x\mu}$ est inférieur à $2 \times \tau$ par conséquent $\mathcal{B}_{reg}\mu$ ont une profondeur inférieure à $2 \times \tau$. Comme $P'(\vec{t})$ est linéaire $x\mu$ est linéaire pour $x \in Var(P'(\vec{t'}))$ et $x\mu$ ne partage pas de variables avec $y\mu$ ($y \neq x$ et $y \in Var(P'(\vec{t'}))$). $\mathcal{B}'$ étant linéaire $\mathcal{B}'\mu$ est aussi linéaire, donc les atomes qui ne sont pas linéaires monadiques ne partagent pas de variables entre eux et sont linéaires. Par conséquent la nouvelle clause introduite dans $\mathcal{C}_{\text{new}}$ vérifie la propriété d'induction.

Si la prochaine étape est Clause Unfolding appliquée à $H \leftarrow A, \mathcal{B}$ de $\mathcal{C}_{\text{new}}$, où A est linéaire monadique de profondeur supérieure à 0. Si la clause de $\mathcal{P}$ utilisée pour cette étape est $H' \leftarrow \mathcal{B}'$ et μ est l'unificateur le plus général de A et H, alors la clause résultat est $H \leftarrow \mathcal{B}'\mu, \mathcal{B}$. Comme dans le premier cas de Definition Unfolding, on peut voir que les atomes de $\mathcal{B}\mu$ ne changent pas de profondeur et que la profondeur des atomes de $\mathcal{B}'\mu$ est strictement inférieure à $Depth(A)$, donc cette règle ne peut pas être appliquée infiniment. Finalement les atomes non linéaires monadiques de $\mathcal{B}'\mu, \mathcal{B}$ sont tous dans $\mathcal{B}$ donc ils restent linéaires et ne partagent pas de variables entre eux.

Si la prochaine étape est Definition Introduction, utilisée sur la clause $H \leftarrow \mathcal{B}$ de $\mathcal{C}_{\text{new}}$. D'après la stratégie préservant la liberté tous les atomes linéaires monadiques ont une profondeur de 0, d'après l'hypothèse d'induction, tous les atomes non monadiques sont linéaires et ne partagent pas de variables entre eux. Soit $\mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_n$ la décomposition maximale de $\mathcal{B}$ en sous-ensembles non vides sans variables communes. Chaque $\mathcal{B}_i$ contient au plus un atome qui n'est pas linéaire monadique, donc c'est le corps d'une définition de jointures régulières. Donc les nouvelles définitions introduites dans $\mathcal{D}_{\text{new}}$ sont des définitions de jointures régulières dont les atomes ont une profondeur inférieure à τ.

Ce résultat nous permet de montrer la décidabilité du test d'appartenance pour les langages représentés par des cs-programes. Il permet aussi d'élargir le résultat [40, 55] en permettant de tester si au moins instance close d'un n-plet d'arbres appartient au langage.

Corollaire 3.3.17. *Soit $(t_1, \ldots, t_n)$ un n-uplet de termes et $\mathcal{L}$ un langage synchronisé, il est décidable de savoir si une instance close de $(t_1, \ldots, t_n)$ appartient à $\mathcal{L}$.*

Preuve. $\mathcal{L}$ peut être représenté par un cs-programme dont le prédicat P reconnaît le langage $\mathcal{L}$. La définition $Test \leftrightarrow P(t_1, \ldots, t_n)$ est une définition de jointures régulières compatible avec $\mathcal{P}$, donc si le résultat d'une $\Rightarrow$-dérivation qui respecte la stratégie de préservation de la liberté est un programme vide après élimination des clauses improductives (voir remarque 3.3.10) alors aucune instance close de $(t_1, \ldots, t_n)$ n'appartient à $\mathcal{L}$ dans le cas contraire le test est vrai. Notons que si $(t_1, \ldots, t_n)$ est clos alors le test est un un test d'appartenance classique.

Nous allons maintenant voir l'intérêt de la stratégie de préservation de la liberté. Pour cela nous définissons d'abord la notion composant libre pour un prédicat dans un programme logique. Cette notion permet de détecter des composants de langages synchronisés qui sont réguliers et qui en plus sont générés comme un langage régulier dans le programme logique.

Définition 3.3.18. Une variable est *linéaire* dans une clause, si elle apparaît au plus une fois dans la tête et au plus une fois dans le corps de la clause. Une position d'argument est une paire

(P, i) où P est un symbole prédicat et i est un entier entre 1 et l'arité de P. Pour un programme $\mathcal{P}$, l'ensemble l'ensemble de ses positions d'argument non-libre est défini inductivement comme suit :

- Si $\mathcal{P}$ contient une clause $P(s_1, \ldots, s_m) \leftarrow \mathcal{B}$ telle que l'argument s_i contient une variable non-linéaire, alors la position d'argument (P, i) est non-libre.
- Si $\mathcal{P}$ contient une clause $P(s_1, \ldots, s_m) \leftarrow \mathcal{B}$ telle que l'argument s_i partage des variables avec deux arguments différents du même atome alors la position d'argument (P, i) est non-libre.
- Si $\mathcal{P}$ contient une clause $P(s_1, \ldots, s_m) \leftarrow Q(t_1, \ldots, t_n), \mathcal{B}$ telle que l'argument s_i partage une variable avec l'argument t_j du corps et que la position d'argument (Q, j) est non-libre, alors la position d'argument (P, i) est non-libre.

Une position d'argument (P, i) est libre dans un programme $\mathcal{P}$, si ce n'est pas une position non-libre de $\mathcal{P}$; dans ce cas on dit que le i^{eme} argument de P est libre dans $\mathcal{P}$.

Exemple 3.3.19. Soit $\mathcal{P}$ le programme de l'exemple 3.3.5. $(M, 1)$ et $(M, 3)$ sont des positions d'arguments non-libres à cause de la première clause définissant M (la variable x est non-linéaire). Ces positions sont les seules à être non libres donc on en déduit que le second argument de M et le seule argument E sont libres dans $\mathcal{P}$.

On montre que les arguments libres génèrent des langages réguliers de termes. A cette fin, on définit un cs-programme linéaire monadique $Reg(\mathcal{P})$ qui est équivalent au cs-programme $\mathcal{P}$ sur ses arguments libres. Pour chaque position d'argument (P, i) on définit P_i un nouveau symbole de prédicat unaire. $Reg(\mathcal{P})$ est constitué des clauses

$$P_i(s_i) \quad \leftarrow \quad \{\, Q_j(t_j) \mid Q(t_1, \ldots, t_n) \in \mathcal{B},\ 1 \leq j \leq n,\ Var(s_i) \cap Var(t_j) \neq \emptyset \,\}$$

pour chaque position d'argument libre (P, i) et chaque clause $P(s_1, \ldots, s_m) \leftarrow \mathcal{B}$ de $\mathcal{P}$ telle que $\mathcal{P} \models \mathcal{B}\theta$ pour une substitution clause θ.

La condition $\mathcal{P} \models \mathcal{B}\theta$ assure que le corps de la clause peut être satisfait pour au moins une de ses instances. En d'autres termes elle assure que la clause est productive et contribue au modèle minimal. Elle est nécessaire pour maintenir l'équivalence entre $\mathcal{P}$ et $Reg(\mathcal{P})$: sans cette condition une clause improductive dans $\mathcal{P}$ pourrait générer des clauses productives dans $Reg(\mathcal{P})$ ce qui aurait pour conséquence que le modèle minimal de ce programme soit plus grand que celui de $\mathcal{P}$. Comme on l'a vu dans la remarque 3.3.10 l'identification des clauses improductives est décidable pour les cs-programmes.

Pour caractériser la relation entre $\mathcal{P}$ et $Reg(\mathcal{P})$ on introduit un opérateur qui projette un ensemble d'atomes clos $\mathcal{A}$ sur ses argument libres :

$$Reg_{\mathcal{P}}(\mathcal{A}) \quad = \quad \{\, P_i(t_i) \mid P(t_1, \ldots, t_m) \in \mathcal{A} \text{ et } (P, i) \text{ est une position d'argument libre} \,\}$$

Lemme 3.3.20. *Soit $\mathcal{P}$ un cs-programme. $Reg(\mathcal{P})$ est un cs-programme linéaire monadique, et la sémantique de $\mathcal{P}$ et $Reg(\mathcal{P})$ coïncide sur les positions d'argument libres, formellement on a $\mathcal{M}_{Reg(\mathcal{P})} = Reg_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$.*

Preuve. Il est évident que $Reg(\mathcal{P})$ est un cs-programme linéaire monadique car toutes les clauses de $\mathcal{P}$ sont des cs-clauses, les nouvelles clauses sont donc aussi des cs-clauses. Par construction tous les symboles de prédicat de $Reg(\mathcal{P})$ sont monadiques. La linéarité des têtes de clause de $Reg(\mathcal{P})$ provient de la linéarité des arguments libres dans la tête des clauses de $\mathcal{P}$. Pour montrer l'équivalence sémantique des deux programmes on fait une induction sur l'opérateur T.

$Reg_{\mathcal{P}}(T^n_{\mathcal{P}}(\emptyset)) \subseteq \mathcal{M}_{Reg(\mathcal{P})}$ *pour tout n :* On montre que si A est un atome clos appartenant à $Reg_{\mathcal{P}}(T^{n+1}_{\mathcal{P}}(\emptyset))$, alors il appartient aussi à $\mathcal{M}_{Reg(\mathcal{P})}$. $A \in Reg_{\mathcal{P}}(T^{n+1}_{\mathcal{P}}(\emptyset))$ ssi il est de la forme $P_i(t_i)$, il y a un atome $P(t_1, \ldots, t_m)$ de $T^{n+1}_{\mathcal{P}}(\emptyset)$, et$(P, i)$ est une position d'argument libre de $\mathcal{P}$.

$P(t_1, \ldots, t_m) \in T^{n+1}_{\mathcal{P}}(\emptyset)$ ssi $\mathcal{P}$ contient une clause $P(s_1, \ldots, s_m) \leftarrow \mathcal{B}$ et il existe une substitution close σ telle que $P(s_1, \ldots, s_m)\sigma = P(t_1, \ldots, t_m)$ et $\mathcal{B}\sigma \subseteq T^n_{\mathcal{P}}(\emptyset)$.

Comme (P, i) est libre, $Reg(\mathcal{P})$ contient une clause $\mathcal{P}_i(s_i) \leftarrow \mathcal{B}'$ où $\mathcal{B}'$ est consistué d'atomes monadiques provenant de positions d'argument libre de $\mathcal{B}$. On a alors $\mathcal{B}'\sigma \subseteq Reg_{\mathcal{P}}(\mathcal{B}\sigma) \subseteq$

$Reg_{\mathcal{P}}(T^n_{\mathcal{P}}(\emptyset)) \subseteq \mathcal{M}_{Reg(\mathcal{P})}$, où la dernière étape utilise l'hypothèse d'induction. $\mathcal{B}'\sigma \subseteq \mathcal{M}_{Reg(\mathcal{P})}$ implique $P_i(s_i)\sigma \in \mathcal{M}_{Reg(\mathcal{P})}$. Comme $P_i(s_i)\sigma = P_i(t_i) = A$ on conclut que A appartient à $\mathcal{M}_{Reg(\mathcal{P})}$.

$T^n_{Reg(\mathcal{P})}(\emptyset) \subseteq Reg_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$ pour tout n : On montre que si A est un atome clos de $T^{n+1}_{Reg(\mathcal{P})}(\emptyset)$, alors il apprtient aussi à $Reg_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$. $A \in T^{n+1}_{Reg(\mathcal{P})}(\emptyset)$ ssi il existe une clause $P_i(s_i) \leftarrow \mathcal{B}'$ de $Reg(\mathcal{P})$ telle que pour une substitution σ, $A = P_i(s_i)\sigma$ et $\mathcal{B}'\sigma \subseteq T^n_{Reg(\mathcal{P})}(\emptyset)$. Par définition de *Reg*, le programme $\mathcal{P}$ contient une clause $P(\ldots, s_i, \ldots) \leftarrow \mathcal{B}$ où $\mathcal{B}$ contient un atome $Q(\vec{x})$ pour chaque $Q_j(x_j) \in \mathcal{B}'$ ainsi que d'éventuels autres atomes qui ne partagent pas de variables avec s_i. De plus, il existe une substitution θ telle que $\mathcal{B}\theta \subseteq \mathcal{M}_{\mathcal{P}}$.

Par hypothèse d'induction, $T^n_{Reg(\mathcal{P})}(\emptyset) \subseteq Reg_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$, donc $\mathcal{B}'\sigma \subseteq Reg_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$. Par définition de $Reg_{\mathcal{P}}$, $\mathcal{M}_{\mathcal{P}}$ contient un atome clos $Q(\vec{x})\tau$ pour chaque $Q_j(t_j) \in \mathcal{B}'\sigma$ tel que $x_j\tau = t_j$. On peut noter que grâce à la linéarité de $\mathcal{B}$, τ est simplement une extension de σ et peut donc être la même pour chaque $Q(\vec{x})$.

Maintenant, considérons la substitution $\tau\theta$. Pour les atomes $Q(\vec{x})$, $Q(\vec{x})\tau\theta = Q(\vec{x})\tau$ appartient à $\mathcal{M}_{\mathcal{P}}$ du fait de la discussion précédente. Pour tous les autres atomes $B \in \mathcal{B}$ qui ne partagent pas de variables avec s_i, on a $B\tau\theta = B\theta$, qui est dans $\mathcal{M}_{\mathcal{P}}$ par définition de *Reg*. Par conséquent, $P(\ldots, s_i, \ldots)\tau\theta \in \mathcal{M}_{\mathcal{P}}$, d'où $P(s_i)\tau\theta \in Reg_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$. Comme $P(s_i)\tau\theta = P(s_i)\sigma\theta = P(s_i)\sigma = A$ on en déduit que A appartient à $Reg_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$.

Théorème 3.3.21. *Soit $\mathcal{P}$ un cs-programme, et $P(\vec{x}) \leftarrow P'(\vec{t}), \mathcal{B}_{reg}$ une définition de jointures régulières compatible avec $\mathcal{P}$. Soit $x \in \vec{x}$ telle que x est le i^{eme} composant de $\vec{t}$ et le i^{eme} argument de P' est libre dans $\mathcal{P}$ et $x \notin Var(\mathcal{B}_{reg})$. Soit $\mathcal{P}'$ la sortie d'une $\Rightarrow$-dérivation utilisant la stratégie de préservation de la liberté. L'argument où x apparaît dans $P(\vec{x})$ est un argument libre de P dans $\mathcal{P}'$.*

Preuve. On prouve par induction sur la longueur de la $\Rightarrow$-dérivation que pour toutes les définitions $P(\vec{x}) \leftarrow P'(\vec{t}), \mathcal{B}_{reg}$ de $\mathcal{D}_{\text{done}}$, et pour toute position i de P telle que $P(\vec{x})|_i = x$, $x \notin Var(\mathcal{B}_{reg})$ et x apparaît à la profondeur 0 dans $P'(\vec{t})$ à une position libre de P', le i^{eme} argument de P est libre dans $\mathcal{C}_{\text{out}}$.

Cette propriété est évidemment vraie quand la longueur de la dérivation est 0 car $\mathcal{D}_{\text{done}}$ et $\mathcal{C}_{\text{out}}$ sont vides dans un état initial.

Considérons une dérivation $S_0 \Rightarrow^* S_n$.

Si la prochaine étape est Definition Unfolding. Soit $P(\vec{x}) \leftarrow P'(\vec{t}), \mathcal{B}_{reg} \in \mathcal{D}_{\text{new}}$, on sait que cette définition est une définition de jointures régulières compatible avec $\mathcal{P}$. Supposons que le i^{eme} argument de P' est libre dans $\mathcal{P}$, et que x est le terme à la position i de $P'(\vec{t})$ et qu'elle n'apparaît pas dans $\mathcal{B}_{reg}$. Par définition des jointures régulières, $P'(\vec{t})$ est linéaire donc x n'apparaît pas dans les autres arguments de $P'(\vec{t})$. Comme $P(\vec{x}) \leftarrow P'(\vec{t}), \mathcal{B}_{reg} \in \mathcal{D}_{\text{new}}$, on sait que P n'apparaît dans aucune tête de clause de $\mathcal{C}_{\text{out}}$, par conséquent tous les arguments de P sont libres dans $\mathcal{C}_{\text{out}}$.

De plus, si Definition Unfolding est appliquée sur un atome A de $\mathcal{B}_{reg}$ en utilisant la clause $H \leftarrow \mathcal{B}$ et μ l'unificateur le plus général de A et H, alors μ n'instancie aucune variable de $P'(\vec{t}), \mathcal{B}_{reg}$. La nouvelle clause introduite dans $\mathcal{C}_{\text{new}}$ est $P(\vec{x}) \leftarrow P'(\vec{t}) \cup (\mathcal{B}_{reg} \setminus \{A\}) \dot\cup \mathcal{B}\mu$ où $\mathcal{B}\mu$ est un ensemble d'atomes linéaires monadiques.

Si l'atome clé est déplié avec la clause $H \leftarrow \mathcal{B}$ en utilisant le mgu μ. La nouvelle clause introduite dans $\mathcal{C}_{\text{new}}$ est $P(\vec{x}) \leftarrow \mathcal{B}\mu \cup \mathcal{B}_{reg}\mu$. On sait que le i^{eme} argument t de H est linéaire et ne partage aucune variable avec les autres arguments de H donc μ est telle que $\mu = \mu' \dot\cup \{x \mapsto t\}$. Par conséquent les variables de $x\mu'$ qui appartiennent à $Var(\mathcal{B}'\mu)$, n'apparaissent qu'une fois dans $\mathcal{B}'\mu$ à la profondeur 0 car les variables de t' ne sont pas instanciées. De plus si $x\mu'$ (c'est à dire t') partage des variables avec un atome A' de $\mathcal{B}'\mu$ alors elles apparaissent toutes dans le même argument libre de A' à la profondeur 0 (ils ne partagent donc qu'une seule variable).

Si la prochaine étape est Clause Unfolding. Tout d'abord notons que $\mathcal{D}_{\text{done}}$ et $\mathcal{C}_{\text{out}}$ ne sont pas modifiés par cette règle donc la propriété d'induction est vérifiée. Soit $P(\vec{t}) \leftarrow A, \mathcal{B}$ la clause de $\mathcal{C}_{\text{new}}$ qui est dépliée où A est l'atome choisi, soit $H' \leftarrow \mathcal{B}'$ la clause de $\mathcal{P}$ utilisée et μ le mgu de A et H'. Soit $P(\vec{x}) \leftarrow P'(\vec{t'}), \mathcal{B}_{reg} \in \mathcal{D}_{\text{new}}$ la définition dont est issue $P(\vec{t}) \leftarrow A, \mathcal{B}$ et σ une application telle que $P(\vec{t}) = P(\vec{x})\sigma$. On suppose que si x est le i^{eme} argument de $P'(\vec{t'})$ et que (P', i) est libre dans $\mathcal{P}$ et x n'apparaît pas dans $\mathcal{B}_{reg}$, alors toutes les variables de $x\sigma$ apparaissent une seule fois dans $A, \mathcal{B}$ et pour tout $A' \in A, \mathcal{B}$, $Var(x\sigma) \cap Var(A') \neq \emptyset$ implique que l'intersection se réduit à une seule variable qui apparaît à la profondeur 0 dans un argument libre de A'. Remarquons que cette propriété est vraie pour les clauses introduites dans $\mathcal{C}_{\text{new}}$ par Definition Unfolding (voir ci-dessus). On sait que A est un atome linéaire monadique de profondeur supérieure à 0 donc les autres atomes de $\mathcal{B}$ ne sont pas instanciés par σ. Par conséquent les variables de $x\sigma$ n'apparaissent qu'une seule fois, à la profondeur 0 dans le corps de la clause résultante et si $x\mu$ partage des variables avec un atome du corps, il n'en partage qu'une seule qui est à la profondeur 0 dans un argument libre de cet atome.

Enfin si la prochaine étape est Definition Introduction. Soit $P(\vec{t}) \leftarrow \mathcal{B}$ la clause de $\mathcal{C}_{\text{new}}$ utilisée par cette règle. Soit $P(\vec{x}) \leftarrow P'(\vec{t'}), \mathcal{B}_{reg} \in \mathcal{D}_{\text{new}}$ la définition d'où provient $P(\vec{t}) \leftarrow A, \mathcal{B}$ et σ la substitution telle que $P(\vec{t}) = P(\vec{x})\sigma$. D'après les propriétés des règles de dépliage, on sait que si x est le i^{eme} argument de $P'(\vec{t'})$ et (P', i) est libre dans $\mathcal{P}$ et x n'apparaît pas dans $\mathcal{B}_{reg}$, alors toutes les variables de $x\sigma$ n'apparaissent qu'une seule fois dans $A, \mathcal{B}$ et pour tout $A' \in A, \mathcal{B}$, $Var(x\sigma) \cap Var(A') \neq \emptyset$ implique que l'intersection se réduit à une seule variable qui apparaît à la profondeur 0 dans un argument libre de A'. Soit $\mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_n$ la décomposition maximale de $\mathcal{B}$ en sous-ensembles disjoints, non-vides et sans variables communes. Les variables de $x\sigma$ sont toutes à la profondeur 0 dans le corps des définitions introduites dans $\mathcal{D}_{\text{new}}$. De plus la nouvelle clause $P(\vec{x})\sigma \leftarrow L_1 \dot{\cup} \cdots \dot{\cup} L_n$ est telle que les variables de $x\sigma$ sont linéaires dans cette clause et $x\sigma$ partage au plus une variable avec chaque $L_i = P_i(\vec{y_i})$, $1 \leq i \leq n$. Si P_i est une nouveau symbole de prédicat alors tous ses arguments sont libres dans $\mathcal{C}_{\text{out}}$, de plus les variables communes apparaissent à la profondeur 0 de l'atome clé de $\mathcal{B}_i$. Sinon, il existe une définition $P_i(\vec{y_i}) \leftarrow \mathcal{B}'_i \in \mathcal{D}_{\text{done}}$ telle que $\mathcal{B}'_i$ est égal à $\mathcal{B}_i$ à un renommage des variables près (soit η ce renommage). Donc les variables communes avec $x\mu$ apparaissent à la profondeur 0 dans un argument libre d'un atome de $\mathcal{B}'_i\eta$, et dans un argument libre de P_i dans $\mathcal{C}_{\text{out}}$.

Les résultats présentés dans cette section nous permettent de retrouver les propriétés des langages synchronisés montrés dans [40, 55] c'est à dire la décidabilité des tests du vide et d'appartenance, la clôture par union et par jointure avec un langage régulier. On peut remarquer que l'utilisation du formalisme de la programmation logique permet d'exprimer plus simplement des problèmes de jointures qu'avec des formalismes à base de grammaires ou de contraintes. En effet, dans l'exemple 3.3.5, $R(x, z) \leftrightarrow M(x, y, z), E(y)$ est une définition de jointure régulière, mais la définition suivante $R(x, z) \leftrightarrow M(x, y, z), E(y), E(s(x))$ qui associe chaque nombre pair avec le résultat de l'addition de celui-ci avec un nombre impair, l'est aussi. Et enfin le test suivant permet de vérifier que l'addition d'un nombre pair avec un nombre impair n'est jamais un nombre paire $T \leftrightarrow M(x, y, z), E(y), E(s(x)), E(s(z))$. Pour chaque problème, nous avons une seule définition de jointures alors qu'avec des formalismes plus basiques il aurait fallut résoudre le problème de manière incrémentale en calculant à chaque étape une seule jointure.

3.4 Deux classes de programmes logiques particulières

La classe des cs-programmes permet de définir les langages synchronisés mais sa syntaxe reste très restrictive, puisque le corps des clauses doit être linéaire et sans symboles de fonction. Dans cette section nous définissons deux classes de programmes qui ne sont pas des cs-programmes mais qui ont un équivalent qui lui est un cs-programme. Ces deux classes de programmes ne lèvent pas la restriction sur la linéarité mais elles permettent d'introduire des symboles de fonction dans le corps des clauses sous certaines conditions. La première classe se base sur la différence de hauteur des variables entre le corps et la tête de la clause alors que le second se base sur la manière dont s'unifie un atome d'un corps de clause avec les têtes de clauses du programme. Pour ces deux classes on trouve les cs-programmes équivalents par l'application de $\Rightarrow$ sans utiliser Clause Unfolding. Pour cela, nous montrons que la terminaison des $\Rightarrow$-dérivations qui choisissent toujours l'ensemble des atomes de profondeur maximales pour l'application de Definition Unfolding.

Une dérivation est finie si à partir d'un certain point plus aucune définition n'est ajoutée dans $\mathcal{D}_{\text{new}}$. Pour montrer cette propriété pour tous les programmes d'une classe donnée, il suffit de montrer que les sous-ensembles sans variables communes $\mathcal{B}_i$ de la règle Definition Introduction satisfont deux conditions :

– le nombre d'atomes de $\mathcal{B}_i$ est borné ;
– la profondeur maximale des atomes de $\mathcal{B}_i$ est aussi bornée.

Dans ce cas, on ne peut construire qu'un nombre fini de $\mathcal{B}_i$ et par conséquant on ne peut avoir qu'un nombre fini de définitions potentielles. D'après la remarque 3.3.8 les $\mathcal{B}_i$ sont construits sur la signature originale du programme en entrée et donc le nombre de symboles de prédicats n'augmente pas.

3.4.1 Les programmes quasi-cs

Définition 3.4.1. Une clause est *quasi-cs*, si son corps est linéaire et si chaque variable qui apparaît à la fois dans le corps et dans la tête de la clause est telle que sa profondeur dans le corps est inférieure ou égale à la plus petite profondeur où elle apparaît dans la tête. Un programme logique est quasi-cs si toutes ses clauses le sont.

Exemple 3.4.2. La clause $P(f(s(x), y), g(x)) \leftarrow P(s(y), s(x))$ est quasi-cs car les variables du corps, x et y, apparaissent à la profondeur 1 dans le corps et aux profondeurs respectives 1 et 2 dans la tête.

La clause $P(f(s(x), y), x) \leftarrow P(s(y), s(x))$ n'est pas quasi-cs car x apparaît à la profondeur 0 dans la tête mais à la profondeur 1 dans le corps.

Chaque programme quasi-cs peut se transformer en un cs-programme équivalent.

Théorème 3.4.3. *Soit $\mathcal{P}$ un programme quasi-cs, et soit $\mathcal{D}_{\mathcal{P}}$ l'ensemble de toutes les tautologies $P(\vec{x}){\leftarrow}P(\vec{x})$ telles que P apparaît dans $\mathcal{P}$. Il existe une $\Rightarrow$-dérivation complète avec comme entrées $\mathcal{P}$ et $\mathcal{D}_{\mathcal{P}}$ qui est finie.*

Preuve. Soit τ la profondeur maximale d'un atome de $\mathcal{P}$. On montre qu'en utilisant uniquement Definition Unfolding et Definition Introduction on a :

– les corps des clauses de $\mathcal{C}_{\text{new}}$ sont linéaires et ont une profondeur bornée par τ et
– les corps des définitions de $\mathcal{D}_{\text{new}}$ sont constitués d'un seul atome de profondeur bornée par τ.

Ceci implique qu'on ne peut générer qu'un nombre fini de définitions différentes à un renommage des variables près et donc les dérivations complètes sont finies. Remarquons que les définitions initiales de $\mathcal{D}_{\text{new}}$, c'est à dire celles de $\mathcal{D}_{\mathcal{P}}$, sont de cette forme.

Definition Unfolding. Considérons une définition $L \leftrightarrow A$ de $\mathcal{D}_{\text{new}}$ et une clause quasi-cs $H \leftarrow \mathcal{B}$ de $\mathcal{P}$ telles que A et H sont unifiables par le mgu μ. La clause ajoutée à $\mathcal{C}_{\text{new}}$ est $(L \leftarrow \mathcal{B})\mu = L\mu \leftarrow \mathcal{B}\mu = L\mu_A \leftarrow \mathcal{B}\mu_H$, où $\mu_A = \mu|_{Var(A)}$ et $\mu_H = \mu|_{Var(H)}$. Les atomes H et A sont sans variables communes (les définitions et les clauses sont renommées avant le dépliage) et A est linéaire, par conséquent, d'après le lemme 2.2.3, $ran(\mu_H)$ est un ensemble linéaire de termes dont les variable sont parmis celles de A et donc n'apparaissent pas dans $\mathcal{B}$. Comme $\mathcal{B}$ est linéaire par définition, $\mathcal{B}\mu_H$ est linéaire aussi.

Il reste à montrer que la profondeur de $\mathcal{B}\mu_H$ est bornée par τ. Soit x une variable apparaissant à la fois dans la tête H et dans le corps $\mathcal{B}$ de la clause. Par la définition des clauses quasi-cs on a $\tau_{x,\mathcal{B}} \leq \tau_{x,H}$ et donc $\tau_{x,\mathcal{B}} + \tau_{x\mu_H} \leq \tau_{x,H} + \tau_{x\mu_H}$, d'après le lemme 2.2.3 on a $\tau_{x\mu_H} \leq \tau_A - \tau_{x,H}$ donc $\tau_{x,H} + \tau_{x\mu_H} \leq \tau_A$, et $\tau_A \leq \tau$ par l'hypothèse d'induction. En combinant ces inégalités on obtient $\tau_{x,\mathcal{B}} + \tau_{x\mu_H} \leq \tau_A$. Par conséquent, la profondeur de $\mathcal{B}\mu_H$ est la profondeur maximale des atomes de $\mathcal{B}$, ainsi que les valeurs $\tau_{x,\mathcal{B}} + \tau_{x\mu}$ de chaque variable x partagée par la tête et le corps de la clauses, sont aussi bornés par τ.

Definition introduction. Le corps de toutes les clauses $H \leftarrow \mathcal{B}$ de $\mathcal{C}_{\text{new}}$ est linéaire donc leur décomposition maximale en sous-ensembles sans variables communes donnent des singletons. Comme la profondeur de $\mathcal{B}$ est bornée par τ, les nouvelles définitions ajoutées dans $\mathcal{D}_{\text{new}}$ sont de la forme $L \leftrightarrow A$ où A est linéaire et de profondeur au plus τ.

3.4.2 Les programmes ib

Définition 3.4.4. Une clause $H \leftarrow \mathcal{B}$ d'un programme logique $\mathcal{P}$ est *basée sur les instances* (on dit alors que c'est une clause *ib* pour instance-based) si $\mathcal{B}$ est linéaire et pour toute paire d'atomes H', A où H' est la tête d'une clause de $\mathcal{P}$ renommée proprement et A un atome de $\mathcal{B}$ telle que H' et A s'unifient, H' est une instance de A. Une programme est basé sur les instances (est un programme ib) s'il est constitué uniquement de clauses ib.

Exemple 3.4.5. Considérons le programme

$$\begin{array}{rcl} P(f(x,y),g(x)) & \leftarrow & P(g(s(x)),y) \\ P(g(s(x)),f(g(y),z)) & \leftarrow & P(f(s(y),x),z) \end{array}$$

La première clause est basée sur les instances car la seule tête de clause s'unifiant avec l'atome du corps est $P(g(s(x)),f(g(y),z))$ et c'est une instance de l'atome du corps. La seconde clause n'est pas une clause ib car la tête de clause $P(f(x,y),g(x))$ s'unifie avec l'atome du corps $P(f(s(y),x),z)$ et ce n'est pas une instance de celui-ci.

Théorème 3.4.6. *Soit $\mathcal{P}$ un programme ib, et soit $\mathcal{D}_{\mathcal{P}}$ l'ensemble de toutes les tautologies $P(\vec{x}) \leftarrow P(\vec{x})$ telles que P apparaît dans $\mathcal{P}$. Toute $\Rightarrow$-dérivation ayant en entrée $\mathcal{P}$ et $\mathcal{D}_{\mathcal{P}}$ est finie.*

Preuve. Nous montrons que toutes les définitions de $\mathcal{D}_{\text{new}}$ sont de la forme $L \leftrightarrow A$, où A est un atome linéaire qui subsume toutes les têtes de clause qui s'unifient avec lui. Ceci implique qu'on ne peut générer qu'un nombre fini de définitions différentes à un renommage des variables près et donc toute dérivation est finie. Remarquons que les définitions de $\mathcal{D}_{\mathcal{P}}$ sont de cette forme.

Definition Unfolding. Considérons une définition $L \leftrightarrow A$ de $\mathcal{D}_{\text{new}}$ et une clause ib $H \leftarrow \mathcal{B}$ de $\mathcal{P}$ telles que A et H sont unifiables par le mgu μ. Comme A subsume H on a $H\mu = H$ et $\mathcal{B}\mu = \mathcal{B}$. Par conséquent $\mathcal{B}\mu_H$, le corps de la nouvelle clause $(L \leftarrow \mathcal{B})\mu = L\mu_A \leftarrow \mathcal{B}\mu_H$, est composé d'atomes linéaires sont variables communes. Par conséquent la décomposition de $\mathcal{B}$ donne uniquement des singletons. De plus, comme $H \leftarrow \mathcal{B}$ est une clause ib chaque atome de $\mathcal{B}$ subsume toutes les têtes de clauses de $\mathcal{P}$ qui s'unifient avec lui.

Definition introduction. Le corps de toutes les clauses $H \leftarrow \mathcal{B}$ de $\mathcal{C}_{\text{new}}$ sont linéaires donc leur décomposition maximale en sous-ensembles sans variables communes donnent des singletons. Donc les nouvelles définitions ajoutées dans $\mathcal{D}_{\text{new}}$ sont de la forme $L \leftrightarrow A$ où A est linéaire et subsume toutes les têtes de clauses de $\mathcal{P}$ qui s'unifient avec lui.

On peut remarquer qu'un programme peut être à la fois quasi-cs et ib mais qu'il existe des programmes quasi-cs qui ne sont pas ib comme par exemple :

$$P(s(s(s(x)))) \leftarrow P(s(s(x))) \qquad P(s(x)) \leftarrow P(x)$$

et vice versa, il existe des programmes qui sont ib mais pas quasi-cs

$$P(x) \leftarrow Q(s(x))$$

Nous verrons dans le chapitre 4 que ces deux classes de programmes vont nous permettre de définir assez facilement des classes de système de réécriture dont les propriétés pourront être déduites de celles des cs-programmes.

3.5 Les programmes pseudo-réguliers

Nous avons défini la classe des cs-programmes et démontré quelques propriétés sur les langages synchronisés reconnus par ces cs-programmes. Malheureusement, comme il est démontré dans [55], les langages synchronisés ne sont pas clos par toutes les opérations de la définition 2.5.1. Ils ne sont notamment pas clos par intersection ou par complément. Dans cette section nous étudions une sous-classe des cs-programmes qui est closes par toutes les opérations ensemblistes et qui permet de reconnaître la classe des relations régulières. Les programmes de cette classe sont appelés *pseudo-réguliers* car ils sont une extension des cs-programmes réguliers qui correspondent aux relations régulières de [21] mais cette extension ne permet pas d'accroître l'expressivité des programmes obtenus comme nous le verrons par la suite. La définition suivante introduit une série de classes de programmes autour des programmes pseudo-réguliers qui nous seront utiles par la suite.

Définition 3.5.1. Soit $H \leftarrow \mathcal{B}$ une clause telle que $\mathcal{B}$ ne contient pas de symboles de fonction.

- $H \leftarrow \mathcal{B}$ est dite *quasi pseudo-régulière* (PR-like en abbrégé ssi chacun des argument de H est de la forme $f(x_1, \ldots, x_{\text{ar}(f)})$ où f est un symbole de fonction et les x_i sont des variables distinctes les unes des autres et qu'il existe une application $\pi\colon \mathit{Var} \mapsto I\!N$ telle que $\pi(x_l) = l$ pour tout $l = 1, \ldots, \text{ar}(f)$ et $\pi(x) = \pi(y)$ pour toutes variables x et y apparaissant dans le même atome du corps.
- $H \leftarrow \mathcal{B}$ est dite *pseudo-régulière* (PR en abbrégé) ssi elle est PR-like et $\mathcal{B}$ est linéaire.
- $H \leftarrow \mathcal{B}$ est appelée *régulière* (R en abbrégé) ssi elle est PR et H est linéaire.
- $H \leftarrow \mathcal{B}$ est dite *pseudo-régulière partagée* (SPR en abbrégé) ssi elle est PR-like et ne contient pas de variables existentielle.
- $H \leftarrow \mathcal{B}$ est dite *régulière partagée* (SR en abbrégé) ssi elle est SPR et H est linéaire.

Un programme est PR-like, PR, R, SPR ou SR si toutes ses clauses sont du type correspondant.

La condition de pseudo-régularité impose que les variables puissent être partionnées en groupes disjoints de telle façon que des variables apparaissant dans le même atome du corps appartiennent au même groupe et qu'il y ait une correspondance 1-1 entre les positions des arguments de la tête et ces groupes.

Exemple 3.5.2. La clause $P(c(y_1, x_1), c(x_1, y_2)) \leftarrow P_1(x_1), P_2(y_1, y_2)$ n'est pas PR-like car x_1 est à la position 2 dans le premier argument de la tête de la clause et à la position 1 dans le second argument.

La clause $P(c(x_1, y_1), c(x_1, y_2)) \leftarrow P_1(y_1), P_2(x_1, y_2)$ n'est pas PR-like car x_1 et y_2 apparaissent dans le même atome du corps mais n'apparaissent pas à la même position dans les arguments de la tête de clause.

La clause $P(c(x_1, y_1), c(x_1, y_2)) \leftarrow P_2(y_1, z), P_2(y_2, y_1)$ est PR-like mais elle n'est ni PR car le corps n'est pas linéaire, ni SPR car z est une variable existentielle.

Le programme logique suivant est PR. Notons que les clauses définissant Id_2 sont régulières car leur tête est linéaire

$$
\begin{array}{ll}
P_f(s(x_1), s(x_2), s(x_3)) \leftarrow Q(x_1, x_2, x_3) & Id_2(c(x_1, y_1), c(x_2, y_2)) \leftarrow Id_2(x_1, x_2), \\
 & \qquad\qquad\qquad\qquad\qquad\quad Id_2(y_1, y_2) \\
P_f(0, 0, 0) \leftarrow & Id_2(s(x), s(y)) \leftarrow Id_2(x, y) \\
P_g(s(x), s(x)) & Id_2(0, 0) \leftarrow \\
\multicolumn{2}{l}{Q(c(x_1, y_1), c(x_1, y_2), c(x_2, y_3)) \leftarrow P_g(x_1, x_2) P_f(y_1, y_2, y_3)}
\end{array}
$$

Exemple 3.5.3. Le programme logique suivant est pseudo-régulier. Remarquons que les clauses qui définissent Id_2 sont régulières car leurs têtes sont linéaires

$$
\begin{array}{ll}
P_f(s(x_1), s(x_2), s(x_3)) \leftarrow Q(x_1, x_2, x_3) & Id_2(c(x_1, y_1), c(x_2, y_2)) \leftarrow Id_2(x_1, x_2), \\
 & \qquad\qquad\qquad\qquad\qquad\quad Id_2(y_1, y_2) \\
P_f(0, 0, 0) \leftarrow & Id_2(s(x), s(y)) \leftarrow Id_2(x, y) \\
P_g(s(x), s(x)) & Id_2(0, 0) \leftarrow \\
\multicolumn{2}{l}{Q(c(x_1, y_1), c(x_1, y_2), c(x_2, y_3)) \leftarrow P_g(x_1, x_2) P_f(y_1, y_2, y_3)}
\end{array}
$$

La décidabilité des tests du vide et d'appartenance pour les langages décrits par des programmes pseudo-régulier provient du fait que ces programmes sont des cs-programmes. Pour montrer la clôture des langages décrits par des cs-programmes par les opérations ensemblistes et celles spécifiques aux relations nous avons besoin de deux résultats principaux qui s'appuient sur les $\Rightarrow$-dérivations. Le premier concerne la clôture des cs-programmes par une jointure généralisée et montre que les programmes SPR peuvent se transformer en programmes pseudo-réguliers. La classe des programmes SPR est utile aussi pour montrer que les programmes pseudo-réguliers et le programmes réguliers ont le même pouvoir d'expression en terme de langages reconnus.

3.5.1 La jointure généralisée des programmes pseudo-réguliers

Définition 3.5.4. Une définition $H \leftrightarrow B_1, \ldots, B_k$ est une *définition de jointure généralisée* si elle ne contient aucun symbole de fonction et $Var(H) \subseteq Var(B_1, \ldots, B_k)$.

Théorème 3.5.5. *Soient $\mathcal{P}$ un programme pseudo-régulier et D une définition de jointure généralisée compatible avec $\mathcal{P}$. Toute $\Rightarrow$-dérivation complète avec $\mathcal{P}$ et $\{D\}$ comme entrées, qui, à chaque étape de dépliage utilise Definition Unfolding sur tous les atomes simultanément, est finie et sa sortie est une programme pseudo-régulier.*

La preuve repose sur des arguments similaires à ceux utilisés pour les théorèmes 3.4.3 et 3.4.6. Pour cela nous définissons une mesure de complexité pour les définitions qui tient compte du nombre de duplications des variables. Pour un objet O (O étant un atome ou une liste d'atomes), $\mathrm{nocc}(x, O)$ est le nombre d'occurrences de x dans O et $\mathrm{nocc}(O) = \sum_{x \in Var(O)} \mathrm{nocc}(x, O)$ le nombre total d'occurrences variables dans O. De plus, $\mathrm{nvar}(O) = |Var(O)|$ est le nombre de variables différentes de O. La complexité d'une variable d'un objet est le nombre de ses occurrences dupliquées : $\chi(x, O) = \mathrm{nocc}(x, O) - 1$ pour un ensemble $\mathcal{X} \subseteq Var(O)$ $\chi(\mathcal{X}, O) = \sum_{x \in \mathcal{X}} \chi(x, O)$. La complexité d'un objet est la somme des occurrences dupliquées de ses variables, plus exactement :

$$\chi(O) = \sum_{x \in Var(O)} \chi(x, O) = \mathrm{nocc}(O) - \mathrm{nvar}(O)$$

La complexité d'une définition ou d'une clause est la complexité de son corps. La complexité d'une variable non dupliquée est zéro et n'est pas ajoutée à la complexité de l'objet. La complexité d'un ensemble linéaire d'atomes est aussi zéro. La complexité augmente soit quand le nombre d'occurrences d'une certaine variable augmente soit quand le nombre de variables dupliquées augmente. Voici quelques propriétés de χ.

Lemme 3.5.6. *Si $\mathcal{X} \subseteq Var(O)$ alors $\chi(O, \mathcal{X}) \leq \chi(O)$.*

C'est évident car $\chi(O)$ et $\chi(O, \mathcal{X})$ sont des sommes dont les termes sont des entiers positifs tels que les termes de la seconde somme apparaissent tous dans la première.

Lemme 3.5.7. *Si $\mathcal{A}$ est un ensemble d'atomes qui ne peut pas se décomposer en sous-ensembles sans variables communes alors il contient au plus $\chi(\mathcal{A}) + 1$ atomes.*

Preuve. Cette propriété se montrer par une simple induction sur la valeur de $\chi(\mathcal{A})$.

- $\chi(\mathcal{A}) = 0$, cela signifique que $\mathcal{A}$ est un ensemble linéaire d'atomes. Comme $\mathcal{A}$ ne peut pas se décomposer en sous-ensembles sans variables communes, $\mathcal{A}$ se réduit à un singleton.
- Supposons que tout ensemble d'atomes $\mathcal{A}$ qui ne peut pas se décomposer en sous-ensembles sans variables communes tels que $\chi(\mathcal{A}) \leq n$ contient au plus $\chi(\mathcal{A}) + 1$ atomes. Considérons l'ensemble indécomposable d'atomes $\{A\} \cup \mathcal{A}$, A partage au moins une variable avec l'un des atomes de $\mathcal{A}$, donc au moins une variable de $Var(\mathcal{A})$ a plus d'occurrences dans $\{A\} \cup \mathcal{A}$ que dans $\mathcal{A}$, par conséquent $\chi(\{A\} \cup \mathcal{A}) > \chi(\mathcal{A})$. Supposons que $\mathcal{A}$ que la décomposition maximale de $\mathcal{A}$ en sous-ensembles non vides sans variables communes donne $\mathcal{A}_1 \dot\cup \cdots \dot\cup \mathcal{A}_k$ avec $\chi(\mathcal{A}_i) = k_i$ D'après l'hypothèse d'induction $\mathcal{A}_i$ contient au plus $k_i + 1$ atomes. Comme $\{A\} \cup \mathcal{A}$ est indécomposable A partage au moins une variable avec chacun des $\mathcal{A}_i$ et cette variable est différente pour chacun des $\mathcal{A}_i$ puisqu'ils n'ont pas de variables communes, par conséquent, $\chi(\{A\} \cup \mathcal{A}) \leq k + \Sigma_{1 \leq i \leq k} k_i$. De plus le nombre d'atomes de $\{A\} \cup \mathcal{A}$ est d'au plus $\Sigma_{1 \leq i \leq k}(k_i + 1) + 1$ c'est à dire $\Sigma_{1 \leq i \leq k}(k_i) + k + 1 \leq \chi((\{A\} \cup \mathcal{A}) + 1$.

Preuve. (du théorème 3.5.5) On montre que

- le dépliage d'une définition de jointure généralisée de $\mathcal{D}_{\text{new}}$ ajoute à $\mathcal{C}_{\text{new}}$ des clauses qui sont pseudo-régulières en considérant les sous-ensembles sans variables communes du corps, S_j, au lieu des atomes qui satisfont trois conditions : (a) S_j ne contient aucun symbole de fonction, (b) la complexité de S_j n'excède pas la complexité de la définition dépliée et (c) toutes les variables de S_j sont associées à la position dans les arguments de la tête, c'est à dire, $\pi(x) = \pi(y)$ pour tous $x, y \in \mathit{Var}(S_j)$.
- l'application de Definition Introduction sur une clause satiasfaisant les trois conditions ci-dessus produit une clause pseudo-régulière dans $\mathcal{C}_{\text{out}}$ et des définitions de jointure généralisée dans $\mathcal{D}_{\text{new}}$ qui ont une complexité inférieure ou égale à celle de la définition initiale $\mathcal{D}$.

Cette dernière propriété garantie la terminaison puisqu'elle signifie que les définitions de $\mathcal{D}_{\text{new}}$ sont bornée par $\chi(\mathcal{D})$ et donc le nombre d'atomes de leur corps est borné par $\chi(D)+1$ (lemme 3.5.7) or on ne peut construire qu'un nombre fini de définitions différentes à un renommage près avec un nombre borné d'atomes.

Definition Unfolding. Soient $D = L \leftrightarrow A_1, \ldots, A_n$ une définition de jointure généralisée, et $C_i = H_i \leftarrow \mathcal{B}_i$, pour $1 \leq i \leq n$, des clauses de $\mathcal{P}$ telles que l'unificateur le plus général simultané de $(A_1, \ldots, A_n)$ et $(H_1, \ldots, H_n)$ existe. On note π_i l'application de $\mathit{Var} \to I\!N$ qui permet de montrer la pseudo-régularité de C_i et $\pi = \pi_1 \dot\cup \cdots \dot\cup \pi_n$ (les clauses n'ont pas de variables communes car elles sont renommées avant l'unification).

Nous montrons que $x\mu$ est un terme linéaire de profondeur 1 pour $x \in \mathit{Var}(A_1, \ldots, A_n)$ et que pour $x \in \mathit{Var}(H_1, \ldots, H_n)$, $x\mu$ est une variable telle que $\pi(x) = \pi(x\mu)$.

L'unificateur le plus général μ est la solution du problème $\Pi = \{\, A_i \stackrel{?}{=} H_i \mid 1 \leq i \leq n \,\}$ qui se réduit à $\{\, y_{ij} \stackrel{?}{=} f_{ij}(\vec{z}_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq \mathit{Ar}(P_i) \,\}$ si $A_i = P_i(\cdots, y_{ij}, \cdots)$ et $H_i = P_i(\cdots, f_{ij}(\vec{z}_{ij}), \cdots)$. Soit $x_1, \ldots, x_l$ l'énumération des variables de D, on a évidemment $\{x_1, \ldots, x_l\} = \{\, y_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq \mathit{Ar}(P_i) \,\}$. On reformule le problème d'unification de la manière suivante $\bigcup_{1 \leq k \leq l} \Pi_k$ où $\Pi_k = \{\, x_k \stackrel{?}{=} f_k(\vec{z}_{ij}) \mid y_{ij} = x_k \,\}$. Notons que $y_{ij} = y_{i'j'}$ implique $f_{ij} = f_{i'j'}$ car l'unificateur existe par hypothèse. Π_k contient autant d'équations que x_k a d'occurrences dans le corps de D. Si ce nombre est supérieur à un, alors Π_k n'est pas encore résolu et nous devons unifier les termes $\{\, f_k(\vec{z}_{ij}) \mid y_{ij} = x_k \,\}$.

Comme les clauses sont renommées avant l'unification, on sait que $z_{ij}|_m \neq z_{i'j'}|_{m'}$ si $i \neq i'$. Comme chaque clause est pseudo-régulière on sait que $z_{ij}|_m \neq z_{i'j'}|_{m'}$ si $m \neq m'$. Par conséquent si $z_{ij}|_m = z_{i'j'}|_{m'}$ alors $i = i'$ et $m = m'$.

Par conséquent, la décomposition des problèmes d'unification précédents donne des équations du type $z_{ij}|_m \stackrel{?}{=} z_{i'j'}|_m$. Notons E_k, l'ensemble des équations provenant du problème Π_k et E_k^m le sous ensemble de E_k qui implique des variables ayant m pour image par π. Enfin $E = E_1 \cup \cdots \cup E_l$.

Décomposons E en sous-ensembles non vides d'équations sans variables communes. Toutes les variables d'un même composant seront instanciées par la même variable par le mgu μ. Si deux équations $z_{i_1 j_1}|_{m_1} \stackrel{?}{=} z_{i'_1 j'_1}|_{m_1}$ et $z_{i_2 j_2}|_{m_2} \stackrel{?}{=} z_{i'_2 j'_2}|_{m_2}$ appartiennent au même composant alors $m_1 = m_2$. Ce qui signifie que μ est tel que si $z_{ij}|_m\mu = z_{i'j'}|_{m'}\mu$ alors $m = m'$.

On en déduit que pour tout $x \in \mathit{Var}(H_1, \ldots, H_n)$ $x\mu$ est une variable de $\mathit{Var}(H_1, \ldots, H_n)$ telle que $\pi(x) = \pi(x\mu)$. D'autre part, les variables $x_1, \ldots, x_l$ de D sont instanciées par des termes de la forme $f(\vec{y})$ où $\pi(\vec{y}|_m) = m$ donc des termes linéaires.

Maintenant, $\mathcal{B}_1 \ldots, \mathcal{B}_n$ est un ensemble d'atomes linéaires sans variables communes par conséquent si A et B sont deux atomes de $\mathcal{B}_1 \ldots, \mathcal{B}_n$ tels que $A\mu$ partage des variables avec $B\mu$ alors $\exists z_{ij}|_m \in \mathit{Var}(A)$ et $\exists z_{i'j'}|_m \in \mathit{Var}(B)$ en d'autres termes leurs variables ont la même image par π. Donc deux atomes A, B de $\mathcal{B}_1\mu \ldots, \mathcal{B}_n\mu$ qui se retrouvent dans le même composant de la décomposition en sous-ensembles non vides sans variables communes de $\mathcal{B}_1\mu \ldots, \mathcal{B}_n\mu$, ont leurs variables qui ont la même image par π.

De plus $x_k\mu = f_k(\vec{x}_k)$ pour $x_k \in \mathit{Var}(D)$ est tel que $\vec{x}_k|_m = \vec{z_{ij}}|_m$ et donc les variables de $L\mu$ qui apparaissent à l'occurrence m dans un argument de L ont pour image m par π.

Il reste maintenant à montrer que la complexité de chacun des composants S de $\mathcal{B}_1\mu \ldots, \mathcal{B}_n\mu$ lors de la décomposition en sous-ensembles non-vides sans variables communes, est au plus égale à la complexité de D. On a vu que toutes les variables de tous les atomes de S ont la même image par π par conséquent la complexité de S dépend du nombre d'occurrences dupliquées de variables ayant la même image par π disons m.

On sait que $H_1, \ldots, H_n$ est un ensemble d'atomes sans variables communes et que $\mathcal{B}_1, \ldots, \mathcal{B}_n$ a en plus la propriété d'être linéaire donc $\chi(\mathcal{B}_1, \ldots, \mathcal{B}_n) = 0$. Comme $x\mu$ est une variable pour $x \in Var(H_1, \ldots, H_n)$, $\text{nocc}((\mathcal{B}_1\mu, \ldots, \mathcal{B}_n\mu), y) = |\{ x \mid x\mu = y \}|$ pour $y \in Var(\mathcal{B}_1\mu, \ldots, \mathcal{B}_n\mu)$.

Soit $E' = \{ E^m_k \mid Var(S) \cap Var(S) \neq \emptyset \}$. $E_1, \ldots, E_{l'}$ une énumeration des éléments de E' et $\mathcal{E}_1, \ldots, \mathcal{E}_{l''}$ la décomposition maximale de E' en sous ensembles non vides sans variables communes. On peut remarquer que $\text{nvar}(E_i) \leq \text{nocc}(x_j)$ si $E_i = E^m_j$. Nous allons montrer que $\text{nvar}(\mathcal{E}_i) \leq \Sigma_{E_j \in \mathcal{E}_i}(\text{nvar}(E_j)) - |\mathcal{E}_i| + 1$. Pour cela, on considère que $\mathcal{E}_i = F_1, \ldots, F_{l_i}$ avec $Var(F_j) \cap Var(F_{j+1}) \neq \emptyset$ pour $1 \leq j < l_j$ ce qui est toujours possible car $\mathcal{E}_i$ ne peut pas se décomposer en sous ensembles sans variables communes. $\text{nvar}(F_1) = \text{nvar}(F_1) - 1 + 1$. Maintenant considérons que $\text{nvar}(F_1 \cup \cdots \cup F_k) \leq \Sigma_{1 \leq j \leq k}(\text{nvar}(F_j)) - k + 1$. $\text{nvar}(F_1 \cup \cdots \cup F_k \cup F_{k+1}) \leq \text{nvar}(F_1 \cup \cdots \cup F_k) + \text{nvar}(F_{k+1}) - 1$ car $Var(F_1 \cup \cdots \cup F_k) \cap Var(F_{k+1}) \neq \emptyset$ donc $\text{nvar}(F_1 \cup \cdots \cup F_k \cup F_{k+1}) \leq \Sigma_{1 \leq j \leq k}(\text{nvar}(F_j)) - k + 1 + \text{nvar}(F_{j+1}) - 1 = \Sigma_{1 \leq j \leq k+1}(\text{nvar}(F_j)) - (k+1) + 1$ donc $\text{nvar}(\mathcal{E}_i) \leq \Sigma_{E_j \in \mathcal{E}_i}(\text{nvar}(E_j)) - |\mathcal{E}_i| + 1$. $\text{nvar}(S) = l'' + l'''$ où $l''' = |Var(S) \setminus Var(E')|$. $\text{nocc}(S) = \Sigma_{1 \leq i \leq l''} \text{nvar}(\mathcal{E}_i) + l'''$ $\chi(S) = \Sigma_{1 \leq i \leq l''} \text{nvar}(\mathcal{E}_i) + l''' - l'' - l''' = \Sigma_{1 \leq i \leq l''} \text{nvar}(\mathcal{E}_i) - l'' \leq \Sigma_{1 \leq i \leq l''}(\Sigma_{E_j \in \mathcal{E}_i}(\text{nvar}(E_j)) - |\mathcal{E}_i| + 1) = \Sigma_{1 \leq i \leq l''}(\Sigma_{E_j \in \mathcal{E}_i}(\text{nocc}(A_1, \ldots, A_n, x_j)) - |\mathcal{E}_i| + 1)$ or l'union des x_j est inclus dans $Var(A_1, \ldots, A_n)$ et la somme des $|\mathcal{E}_i|$ vaut l' c'est à dire le nombre de x_j donc $\Sigma_{1 \leq i \leq l''}(\Sigma_{E_j \in \mathcal{E}_i}(\text{nocc}(A_1, \ldots, A_n, x_j)) - |\mathcal{E}_i| + 1) = \Sigma_{1 \leq i \leq l'}(\text{nocc}(A_1, \ldots, A_n, x_i)) - l'$ donc d'après le lemme 3.5.6 ce nombre est inférieur ou égal à $\text{nocc}(A_1, \ldots, A_n) - \text{nvar}(A_1, \ldots, A_n)$ et donc à $\chi(A_1, \ldots, A_n) = \chi(D)$.

Definition Introduction. Comme vu ci-dessus, les clauses de $\mathcal{C}_{\text{new}}$ sont de la forme $P(\cdots f_k(\vec{z}_k) \cdots) \leftarrow \cdots \cup S_j \cup \cdots$, où les S_j ne contiennent pas de symboles de fonction et $\pi(x) = j$ pour toutes les variables x de S_j. De plus, la complexité de S_j est bornée par la complexité de la définition dépliée D, c'est à dire $\chi(S_j) \leq \chi(D)$.

Definition Introduction décompose chaque S_j en sous-ensembles sans variables communes et les remplace par des atomes linéaires. La clause obtenue qui est ajoutée à $\mathcal{C}_{\text{out}}$, est pseudo-régulière car le corps ne contient plus de variables dupliquées. Les corps des définitions ajoutées à $\mathcal{D}_{\text{new}}$, sont les sous-ensembles sans variables communes S_j qui ont une complexité inférieure ou égale à $\chi(D)$. Par conséquent, par induction, on peut conclure que la complexité des définitions est bornée par $\chi(\mathcal{D})$, la complexité de la définition initiale.

Théorème 3.5.8. *Les langages définis par des programmes pseudo-réguliers sont clos par union, intersection, restriction, projection, cylindrification, permutation et produit cartésien.*

Preuve. Pour cela il suffit de remarquer que toutes les opérations de la figure 3.4 sont toutes des définitions de jointures généralisées. Pour l'union de $\mathcal{L}_1$ et $\mathcal{L}_2$ définis respectivement par les prédicats P_1 de $\mathcal{P}_1$ et P_2 de $\mathcal{P}_2$, il suffit de calculer les deux programmes $\mathcal{P}'_1$ et $\mathcal{P}'_2$ à partir des définitions $P_\cup(\vec{x}) \leftarrow P_1(\vec{x})$ et $P_\cup(\vec{x}) \leftarrow P_2(\vec{x})$ respectivement, puis de prendre comme résultat l'union de $\mathcal{P}'_1$ et $\mathcal{P}'_2$.

Remarque 3.5.9. Pour un cs-programme régulier et une définition de jointure simple, le théorème 3.5.5 garantit simplement que le résultat de la dérivation sera un programme pseudo-régulier. En regardant attentivement la preuve on peut voir que la dérivation préserve la régularité car la linéarité de tous les atomes se propage dans les clauses de $\mathcal{C}_{\text{out}}$. Donc le résultat de la dérivation est un cs-programme régulier.

Exemple 3.5.10. Soient le langage des couples (t, t) tels que

$$t = c(s^{n_1}(0), c(s^{n_2}(0), \cdots c(s^{n_k}(0), 0) \cdots))$$

et le langage des couples (t', t') tels que

$$t' = c(c(\cdots c(0, s^{n_k}(0)) \cdots, s^{n_2}(0)), s^{n_1}(0))$$

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$Q(x,y) \leftrightarrow$ $P(x,y), P'(x,y)$			U
	$Q(0,0) \leftarrow$ $Q(c(x,y),c(x,y)) \leftarrow$ $S(x), P(y,y),$ $S(y), P'(x,x)$		D
	$Q(c(x,y),c(x,y)) \leftarrow$ $S(x), P(y,y),$ $S(y), P'(x,x)$	$Q(0,0) \leftarrow$	D
$Q'(x) \leftrightarrow$ $S(x), P'(x,x)$ $Q''(y) \leftrightarrow$ $S(y), P(y,y)$		$Q(c(x,y),c(x,y)) \leftarrow$ $Q'(x), Q''(y)$	U
$Q''(y) \leftrightarrow$ $S(y), P(y,y)$	$Q'(0) \leftarrow$		D
$Q''(y) \leftrightarrow$ $S(y), P(y,y)$		$Q'(0) \leftarrow$	U
	$Q''(0) \leftarrow$		D
		$Q''(0) \leftarrow$	

FIG. 3.5 – Calcul de l'intersection de l'exemple 3.5.10

pour $k, n_1, \ldots, n_k \geq 0$.

Ces deux langages sont décrits respectivement par les prédicats P et P' du programme pseudo-régulier suivant

$$\begin{array}{rclrcl} S(0) & \leftarrow & & S(s(x)) & \leftarrow & S(x) \\ P(0,0) & \leftarrow & & P(c(x,y),c(x,z)) & \leftarrow & S(x), P(y,z) \\ P'(0,0) & \leftarrow & & P'(c(y,x),c(z,x)) & \leftarrow & S(x), P(y,z) \end{array}$$

L'intersection de ces deux langages est définies par

$$Q(x,y) \quad \leftrightarrow \quad P(x,y), P'(x,y)$$

qui est une définition de jointure généralisée. La figure 3.5 montre une dérivation complète partant de cette définition. Le résultat est le programme pseudo-régulier suivant

$$\begin{array}{rclrl} Q(0,0) & \leftarrow & & Q'(0) & \leftarrow \\ Q(c(x,y),c(x,y)) & \leftarrow & Q'(x), Q''(y) & Q''(0) & \leftarrow \end{array}$$

qui définit le langage $\{(0,0),\ (c(0,0),c(0,0))\}$.

On peut noter que les jointures généralisées permettent d'exprimer en une seule définition une série d'opérations sur les langages de n-uplets d'arbres.

3.5.2 Le complément des programmes pseudo-réguliers

Dans cette section nous montrons comment calculer la complément d'un langage décrit par un programme pseudo-régulier en utilisant les techniques de transformations de programmes logiques. Pour cela nous utilisons la classe des programmes SPR. Un programme de cette classe peut se transformer en un programme pseudo-régulier. De plus si les têtes des clauses du programme en entrée sont toutes linéaires alors le résultat est un cs-programme régulier. Ce résultat nous permettra de montrer que les programmes pseudo-réguliers et les cs-programmes réguliers ont le même pouvoir d'expression c'est a dire qu'ils définissent les relations régulières.

Théorème 3.5.11. *Soient $\mathcal{P}$ un programme SPR et $\mathcal{D}_p$ l'ensemble de toutes les tautologies $P(x) \leftarrow P(x)$ telles que P apparaît dans $\mathcal{P}$. Toute $\Rightarrow$-dérivation prenant en entrées $\mathcal{P}$ et $\mathcal{D}_{\mathcal{P}}$*

est finie et sa sortie est un programme PR. Si le programme en entrée est SR alors le résultat est un cs-programme régulier.

Preuve. La preuve est très similaire à la preuve du théorème 3.5.5 la seule différence concerne la borne de χ pour les définitions apparaissant dans la dérivation. En effet le corps des clauses n'étant plus linéaire, on ne peut plus garantir que χ n'augmente pas lors du dépliage. Par contre, nous avons une hypothèse supplémentaire l'absence de variables existentielles. Cette hypothèse va nous permettre de borner le nombre de variables apparaissant dans le corps des définitions par $MaxAr$ où $MaxAr = Max(\{ Ar(P) \mid P \in \mathcal{P} \}$

Soient $P(x_1, \ldots, x_n) \leftrightarrow A_1, \ldots, A_k$ une définition de jointure généralisée sans variables existentielles avec $n \leq MaxAr$, $H_i \leftarrow \mathcal{B}_i$ $(1 \leq i \leq k)$ les clauses utilisées pour le dépliage et μ l'unificateur le plus général de $(A_1, \ldots, A_k)$ et $(H_1, \ldots, H_k)$. Par la définition de SPR on a $Var(\mathcal{B}_i) \subseteq Var(H_i)$ par conséquent $Var(\mathcal{B}_i\mu) \subseteq Var(H_i\mu)$ donc $P(x_1, \ldots, x_n)\mu \leftrightarrow \mathcal{B}_1\mu, \ldots, \mathcal{B}_k\mu$ est sans variables existentielles. Comme dans la preuve du théorème 3.5.5, on montre que $\pi(x\mu) = \pi(x)$ pour toute variable x de $(H_1, \ldots, H_k)$ donc pour chaque composant S de la décomposition de $\mathcal{B}_1\mu, \ldots, \mathcal{B}_k\mu$ en sous ensemble non vide sans variables communes, le nombre de variable est limité à n donc les atomes de S sont de la forme suivante $Q(\vec{x})$ où Q est un élément d'un ensemble fini et les éléments de $\vec{x}$ sont pris dans un ensemble fini, ce qui signifie qu'un nombre fini de tels atomes peut être construit, donc il existe qu'un nombre fini de composants S possibles.

Corollaire 3.5.12. *Tout programme SR (resp. SPR) $\mathcal{P}$ peut être transformé en un programme R (resp PR) équivalent.*

Un conséquence très intéressante du théorème 3.5.11 est qu'il permet de montrer que les cs-programmes réguliers et les programmes PR ont le même pouvoir d'expression.

Corollaire 3.5.13. *Tout programme PR peut être transformé en un cs-programme régulier équivalent.*

Preuve. Sans perte de généralité on peut considérer que le programme PR est sans variables existentielles, si ce n'est pas le cas, on le transforme en un programme équivalent sans variables existentielles (remarque 3.3.9).

Toute clause PR $H \leftarrow \mathcal{B}$ est équivalent à la clause SR $H' \leftarrow \mathcal{B}' \dot\cup Id$ où

- H' est un atome linéaire tel que $H = H'\sigma$, σ étant une substitution de Var dans Var telle que $\mathcal{B} = \mathcal{B}'\sigma$ et
- $Id = \{Id_n(x_1, \ldots, x_n) | n > 1$ et $\{x_1, \ldots, x_n\}$ est l'ensemble des variables qui ont la même image par $\sigma\}$ où chaque Id_i sont des prédicats définis par un cs-programme régulier et décrivent les ensembles de i-uplets de termes identiques.

$H' \leftarrow \mathcal{B}' \dot\cup Id$ est SR car elle n'a aucune variable existentielle, H' est linéaire et $\forall x, y \in Var(A)$ avec $A \in \mathcal{B}' \dot\cup Id$, x et y apparaissent à la même position dans les arguments de H'. Ce programme SR peut se transformer en un cs-programme régulier équivalent.

L'équivalence sémantique entre les programmes PR et R diminue pas l'intérêt des programmes PR car ils sont syntaxiquement moins restrictifs que les réguliers comme nous le verrons dans la chapitre suivant de ce document.

Nous allons maintenant utiliser les programmes SPR pour donner un algorithme de calcul du complément d'un langage de n-uplets d'arbres décrit par un prédicat P d'un programme PR $\mathcal{P}$ qui utilise la transformation $\Rightarrow$. Cet algorithme est constitué de deux étapes. La première étape produit à partir de $\mathcal{P}$ un nouveau programme SR $\overline{\mathcal{P}}$ qui définit le complément de P. La seconde étape utilise le corollaire 3.5.12 pour obtenir un cs-programme régulier équivalent à $\overline{\mathcal{P}}$.

Soit P un prédicat définit par le programme PR $\mathcal{P}$. Sans perte de généralité on peut considérer que $\mathcal{P}$ ne contient ni clauses improductives (voir remarque 3.3.10, ni variables existentielles (voir remarque 3.3.9). $\mathcal{P}$ est équivalent au programme SR $\mathcal{P}'$ obtenue par la technique utilisée dans la preuve du corollaire 3.5.13.

Pour chaque prédicat Q d'arité n de $\mathcal{P}'$ et chaque n-uplet $(f_1, \ldots, f_n)$ de $\mathcal{C}^n$, on définit l'ensemble $AC(Q, (f_1, \ldots, f_n))$ de toutes les clauses de $\mathcal{P}$ dont la tête est $Q(f_1(\vec{x}_1), \ldots, f_n(\vec{x}_n))$ par

$\{Q(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n)) \leftarrow \mathcal{B} \in \mathcal{P}'\}$. Notons que comme les têtes de clauses sont linéaires dans $\mathcal{P}'$, toutes les têtes de clause de $AC(Q, \vec{f})$ sont égales à un renommage près. Dans la suite nous les considèrerons comme égales. Pour un atome $A = P(\vec{t})$, $\overline{A}$ est une notation pour $\overline{P}(\vec{t})$.

Soient $\vec{f} = f_1, \dots f_n$, $\vec{x}_i$ $(1 \leq i \leq n)$ des vecteurs différents deux à deux de variables distintes et $AC(Q, \vec{f}) = \{H \leftarrow \mathcal{B}_1, \dots, H \leftarrow \mathcal{B}_k\}$. On définit

$$\overline{AC}(Q, \vec{f}) = \begin{cases} \emptyset & \text{si } H \leftarrow \in AC(Q, \vec{f}) \\ Q(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n)) \leftarrow & \text{si } AC(Q, \vec{f}) = \emptyset \\ \quad \{\, \overline{H} \leftarrow \overline{A_1}, \dots, \overline{A_k} \mid \forall i \in [1,k] A_i \in \mathcal{B}_i \,\} & \text{sinon} \end{cases}$$

Exemple 3.5.14. Considérons le programme logique constitué des trois clauses suivantes :

$$\begin{array}{l} P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(x_1, x_2) \\ P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(y_1, y_2) \\ P(0, 0) \leftarrow \end{array}$$

$$\begin{array}{l} AC(P, (0, c)) = \emptyset \\ AC(P, (c, 0)) = \emptyset \\ AC(P, (0, 0)) = \{P(0, 0) \leftarrow\} \\ \ AC(P, (c, c)) = \{ \\ \quad P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(x_1, x_2), \\ \quad P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(y_1, y_2)\} \end{array}
\quad\Bigg\|\quad
\begin{array}{l} \overline{AC}(P, (0, c)) = \{\overline{P}(0, c(x, y)) \leftarrow\} \\ \overline{AC}(P, (c, 0)) = \{\overline{P}(c(x, y), 0) \leftarrow\} \\ \overline{AC}(P, (0, 0)) = \emptyset \\ \ \overline{AC}(P, (c, c)) = \{ \quad \overline{P}(c(x_1, y_1), c(x_2, y_2)) \leftarrow \\ \qquad\qquad\qquad\qquad \overline{P}(x_1, x_2), \overline{P}(y_1, y_2)\} \end{array}$$

Lemme 3.5.15. *Soit $\mathcal{P}$ un programme SPR sans clauses improductives, ni variables existentielles. Le programme $\overline{\mathcal{P}}$ constitué de l'ensemble de clauses $\bigcup_{P \in \mathcal{P}, \vec{f} \in \mathcal{F}^n} \overline{AC}(P, \vec{f})$ est tel que $\forall P \in \mathcal{P}, \forall (t_1, \dots, t_n) \in T^n_{\mathcal{F}}, \mathcal{P} \models P(t_1, \dots, t_n)$ si et seulement si $\overline{\mathcal{P}} \models \overline{P}(t_1, \dots, t_n)$.*

Preuve. On montre que $\overline{\mathcal{P}}$ est obtenu à partir de $\mathcal{P}$ en utilisant simplement des équivalences logiques. Dans cette preuve on note $H \leftarrow$ par $H \leftarrow vrai$. $\mathcal{P}$ est équivalent à $\{\, H \leftrightarrow false \bigvee_{H \leftarrow \mathcal{B} \in \mathcal{P}} \mathcal{B} \mid H = P(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n)), P \in \mathcal{P}, (f_1, \dots, f_n) \in \mathcal{F}^n \,\}$ c'est à dire pour tous les atomes possibles de la forme $H = P(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n))$, on regroupe l'ensembles des corps des clauses qui ont pour tête H. Ces corps sont connectés par l'opérateur logique *ou*. La négation de ces formules est $\{\overline{H} \leftrightarrow true \bigwedge_{H \leftarrow \mathcal{B} \in \mathcal{P}} \neg \mathcal{B}\}$. Si un des corps de clause est la constante $vrai$ $\overline{H}$ est équivalent à faux, par conséquent aucune clause ayant pour tête $\overline{H}$ ne se trouve dans $\overline{\mathcal{P}}$. Si la définition se réduit à $H \leftrightarrow false$ signifiant qu'aucune instance close de H se trouve dans le plus petit modèle de $\mathcal{P}$ alors $\overline{H} \leftarrow$ est dans $\overline{\mathcal{P}}$, ce qui signifie que toutes les instances closes de $\overline{H}$ sont dans le plus petit modèle de $\overline{\mathcal{P}}$. Finalement, dans les autres cas, $true \bigwedge_{H \leftarrow \mathcal{B} \in \mathcal{P}} \neg \mathcal{B}$ est réduit en une disjonction de conjonctions d'atomes négatifs. Chaque conjonction est de la forme $\neg A_1 \wedge \dots \wedge \neg A_k$ où chaque A_i appartient à un $\mathcal{B}_i$ différent. Ces définitions produisent les clauses $\{\, \overline{Q}(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n)) \leftarrow \overline{A_1}, \dots, \overline{A_k} \mid AC(Q, (f_1, \dots, f_n)) = \{H \leftarrow \mathcal{B}_1, \dots, H \leftarrow \mathcal{B}_k\}$ et $\forall i \in [1,k] A_i \in \mathcal{B}_i \,\}$ de $\overline{\mathcal{P}}$.

Lemme 3.5.16. *Soit $\mathcal{P}$ un SPR sans clauses improductives ni variables existentielles. Le programme $\overline{\mathcal{P}}$ est SPR.*

La preuve de ce lemme est évidente car aucune substitution (à part des renommages de variables) n'est appliquée pour définir $\overline{\mathcal{P}}$ donc les application π utilisées pour montrer la peudo-régularité dans $\mathcal{P}$ sont préservées dans $\overline{\mathcal{P}}$.

On peut donc montrer la clôture par complément de la manière suivante

Théorème 3.5.17. *Soient $\mathcal{P}_1$ et $\mathcal{P}_2$ deux programmes PR, $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages décrits par les prédicats $P_{\mathcal{L}_1}$ de $\mathcal{P}_1$ et $P_{\mathcal{L}_2}$ de $\mathcal{P}_2$. Le programme décrivant $\mathcal{L}_1 \setminus \mathcal{L}_2$ est calculé en deux étapes. On calcule d'abord $\overline{\mathcal{P}_2}$ puis on calcule $\mathcal{P}_1 \cap \overline{\mathcal{P}_2}$ à partir de $\mathcal{P}_1 \cup \overline{\mathcal{P}_2}$ et de la définition $P_{\mathcal{L}_1 \setminus \mathcal{L}_2}(\vec{x}) \leftrightarrow P_{\mathcal{L}_1}(\vec{x}), P_{\overline{\mathcal{L}_2}}(\vec{x})$ en utilisant $\Rightarrow$.*

3.5.3 Les programmes logiques Non-Greibach

Même si les programmes PR sont syntaxiquement moins restrictifs que les programmes R, leur syntaxe reste très stricte. Le but de cette section est de permettre d'affaiblir un peu les restrictions syntaxiques en permettant notamment d'avoir des têtes de clauses de hauteur supérieure à 1.

Définition 3.5.18. Une clause est dite *non-Greibach* (NG en abbrégé) si au moins un des arguments de sa tête a une hauteur supérieure à un. Soit $H \leftarrow \mathcal{B}$ une clause telle que $\mathcal{B}$ ne contient pas de symboles de fonction.

- $H \leftarrow \mathcal{B}$ est dite *NGPR-like* ssi aucun des arguments de H n'est une variable et il existe une application $\pi\colon Var \mapsto \mathbb{N}^+$ telle que si $\pi(x) = u$ alors toutes les occurrences de x dans les arguments de H apparaissent à la position u et $\pi(x) = \pi(y)$ pour toutes variables x et y apparaissant dans le même atome du corps.
- $H \leftarrow \mathcal{B}$ est dite *NGPR* ssi elle est NGPR-like et $\mathcal{B}$ est linéaire.
- $H \leftarrow \mathcal{B}$ est dite *NGR* ssi elle est NGPR et H est linéaire.
- $H \leftarrow \mathcal{B}$ est dite *NG-SPR* ssi elle est NGPR-like et elle ne contient pas de variable existentielle.
- $H \leftarrow \mathcal{B}$ est dite *NG-SR* ssi elle est NG-SPR et H est linéaire.

Un programme est NGPR-like, NGPR, R, NG-SPR or NG-SR si toutes les clauses sont du type correspondant.

Par exemple, la clause $P(c(s(x), y), s(s(z))) \leftarrow P(x, z), Q(y)$ est NGPR car x et z apparaissent à la même position 1.1 dans les arguments de la tête. Plus précisément elle est NGR car la tête est linéaire.

Lemme 3.5.19. *Toute clause NG-xxx $H \leftarrow B$ est équivalente à un ensemble de clauses xxx (xxx étant soit PR, soit SPR, soit R, soit SR).*

Preuve. Soit $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot\cup \ldots \dot\cup \mathcal{B}_k$ une clause NGPR-like où $\mathcal{B}$ est l'ensemble des atomes dont les variables qui apparaissent dans la tête, sont toutes à la profondeur 1 dans les arguments de la tête qui les contiennent et chaque $\mathcal{B}_i$ est l'ensemble des atomes qui n'apparaissent pas dans $\mathcal{B}$ et dont les variables ont une image par π de la forme $i.u$ ($u \neq \varepsilon$).

Considérons l'atome $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n}))$ où $\vec{x_i}|_j = \vec{t_i}|_j$ si $\vec{t_i}|_j$ est une variable et $\vec{x_i}|_j$ est une nouvelle variable [2] sinon. Soit σ telle que $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n}))\sigma = P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$

$$P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot\cup \ldots \dot\cup \mathcal{B}_k$$

est équivalent à l'ensemble de clauses

$$\{P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n})) \leftarrow \mathcal{B}, P_1(\vec{y_1}), \ldots, P_k(\vec{y_k})\} \bigcup_{i\in[1,k]} \{P_i(\vec{y_i})\sigma \leftarrow \mathcal{B}_i\}$$

où les P_i sont des nouveaux symboles de prédicat et $\vec{y_i}$ est le vecteur de toutes les variables z apparaissant à la position i dans $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n}))$ et telles que $z\sigma \neq z$.

La clause $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n})) \leftarrow \mathcal{B}, P_1(\vec{y_1}), \ldots, P_k(\vec{y_k})$ est PR-like par construction et les clauses $P_i(\vec{y_i})\sigma \leftarrow \mathcal{B}_i$ sont NGPR-like et ont des têtes de profondeurs strictement inférieures à la profondeur de $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$. Par conséquent, en itérant ce procesus on obtient un ensemble de clauses PR-like. On peut remarquer que

- le processus n'introduit aucune variable existentielle donc si $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_k$ est NG-SPR, les clauses résultantes sont toutes clauses NG-SPR
- les nouvelle variables introduites par le processus sont toutes différentes, donc si $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_k$ est NGPR les clauses résultantes le sont aussi et de plus si $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$ est linéaire les têtes des nouvelles clauses le sont aussi.

[2] c'est à dire que les nouvelles variables introduites sont différentes deux à deux et sont aussi différentes des variables de $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$.

Exemple 3.5.20. Considérons la clause NG-SPR

$$P_f(s(c(x_1, y_1)), s(c(x_1, y_2)), s(c(x_2, y_3))) \leftarrow P_f(y_1, y_1, y_3), P_g(x_1, x_2)$$

L'ensemble de clauses SPR équivalent est

$$\begin{array}{l} P_f(s(x), s(y), s(z)) \leftarrow P_1(x, y, z) \\ P_1(c(x_1, y_1), c(x_1, y_2), c(x_2, y_3)) \leftarrow P_f(y_1, y_1, y_3), P_g(x_1, x_2) \end{array}$$

3.6 Travaux liés

Le lien étroit qu'il y a entre les ensembles de contraintes d'un coté et les automates d'arbres de l'autre est très naturel. Il a été utilisé fréquemment dans la littérature (on peut citer par exemple [31] ou [21, Section 7.6]). Cependant aucun de ces articles n'exploite le point de vue de la programmation logique pour la résolution des problèmes des langages de n-uplets d'arbres. Parmi les travaux sur les contraintes ensemblistes, on peut aussi citer [36] dont l'objectif est de définir des contraintes sur des ensembles de n-uplets d'arbres.

On peut faire un parallèle avec notre travail dans les domaines de l'inférence et de la vérification de types pour les programmes logiques. Le type d'une variable de n'importe quel programme logique peut se définir par l'ensemble des valeurs que cette variable peut prendre. En opérant ainsi on obtient des types exprimés par des ensembles de termes clos du 1er ordre. Frühwirth et al. [30] ont montré que les types pour un programme logique peuvent être approximés et décrits sémantiquement par des programmes logiques ne contenant que des prédicats unaires dont les clauses sont de la forme $H(t) \leftarrow P_1(t_1), \ldots, P_n(t_n)$ où t est linéaire et chaque t_i est soit un sous-terme de t, soit un terme contenant (strictement) t, ou soit une terme sans variables communes avec t. Grâce à des techniques de dépliage, ces programmes peuvent se simplifier en cs-programmes linéaires monadiques pour lesquels on peut faire de la vérification de type par un test d'appartenance. Cette approche a été partiellement étendue à des termes d'ordre supérieur [38] et au automate AC (associatifs-commutatifs) [39].

Une différence importante entre la vérification de type en programmation logique et notre travail est l'arité des langages que nous manipulons. En effet nous nous intéressons à des langages d'arité quelconques et aux interactions entre les différents composants du n-uplets, les types, eux, sont des langages unaires. Par conséquent, les cs-programmes peuvent contenir des prédicats d'arité quelconques mais nécessitent la linéarité des corps des clauses, alors que les programmes ne contenant que des prédicats unaires nécessitent la linéarité des têtes de clause mais sont compatibles avec des corps de clauses contenant des symboles de fonction et des variables dupliquées.

Nielson et al. [75] ont étendu [30] en introduisant des prédicats non unaires pour manipuler des relations. Ils définissent ainsi les relations fortement reconnaissables par des programmes logiques contenant uniquement des clauses normalisées de la forme :

$$\begin{array}{lcl} P(f(x_1, \ldots, x_n)) & \leftarrow & Q_1(x_1), \ldots, Q_k(x_n) \\ P(x_1, \ldots, x_n) & \leftarrow & Q_1(x_1), \ldots, Q_k(x_n) \end{array}$$

où les x_i sont des variables distinctes. Comme les clauses normalisées sont quasi-cs, notre mécanisme de transformation peut transformer ces programmes en cs-programmes. On obtient en fait un cs-programme régulier car la tête et le corps de chaque clause sont linéaire. [75] définit plusieurs classes de programmes logiques qui ont des équivalents normalisés.

L'idée de base de l'algorithme de transformation de [75] est d'enlever les liens inutiles entre les variables. Par exemple la clause $P(f(x, y), z) \leftarrow P(x, y), Q(z)$ est équivalente à

$$\begin{array}{rcl} P(z', z) & \leftarrow & P'(z'), Q(z) \\ P'(f(x, y)) & \leftarrow & P'(x), Q(y) \end{array}$$

Nos règles de transformation ne sont pas capables de détecter ce genre de cas et par conséquent ne fonctionnent pas sur les classes définies dans [75]. Il semble donc très intéressant d'étudier la possibilité d'intégrer ce genre de techniques dans notre cadre afin de pouvoir étendre les classes de programmes logiques pouvant se transformer en cs-programmes ou une de leurs sous-classes.

De nombreux algorithmes ont été proposés pour décider la satisfiabilité des contraintes ensemblistes (c'est à dire pour décider le test du vide) comme par exemple [7, 16, 24, 42]. Ces algorithmes exploitent les propriétés algébriques des expressions ensemblistes pour réduire les contraintes dans une sorte de forme résolue, qui, dans le cas des articles cités, est un système de contraintes linéaires monadiques (c'est à dire sur des mono-uplets et sans variables dupliquées). Les règles de transformation décrites dans ces articles ne sont pas directement comparables avec notre technique car elles utilisent des propriétés spécifiques aux contraintes ensemblistes considérées. Il nous reste à étudier comment notre cadre pourrait traiter ces classes de problèmes. On peut noter que des algorithmes à base d'automates d'arbres [36, 37] ont été utilisés pour résoudre des contraintes ensemblistes. Ces travaux pourraient nous servir de base pour utiliser nos outils dans le but de résoudre des contraintes ensemblistes sur des ensembles de n-uplets d'arbres.

Les classes de programmes et les techniques de preuve que nous avons utilisées dans ce chapitre, ressemblent un peu à celles utilisées pour les procédures de décision de la résolution (voir par exemple [27, 28]). Dans ces articles, on montre la décidabilité de certaines classes de clauses en montrant la terminaison de certains raffinements de la résolution. Ces preuves de terminaison se basent essentiellement sur une borne pour la longueur et la profondeur des clauses comme nous le faisons. Par contre, dans ces techniques la procédure de décision augmente l'ensemble initial de clauses dans le but de dériver la clause vide alors qu'ici nous simplifions des clauses afin d'obtenir un programme d'une classe plus simple.

Du point de vue de la programmation logique, les sections 3.4 et 3.5.2 introduisent des classes de programmes logiques qui ont des cs-programmes équivalents. Proietti and Pettorossi [78] définissent une classe similaire aux quasi-cs programmes appelée programmes non-ascendants. Cette classe requiert aussi que la profondeur des occurrences d'une variable donnée soit plus grande dans la tête des clause que dans leur corps mais les clauses non-ascendantes sont complètement linéaires alors que les clauses quasi-cs peuvent avoir des têtes non-linéaires. On peut noter que la classe des programmes ib ne peut se comparer ni à la classe des programmes quasi-cs ni à celle des programmes ascendants. [78] définit aussi la classe des programmes "tree-like", qui contient la classes des cs-programmes linéaire monadiques mais cet article n'explore pas les liens entre cette classe de programmes et les langages d'arbres. De plus, notre résultat concernant la projection de cs-programmes sur leurs arguments libres (lemme 3.3.20) produit directement un cs-programme linéaire monadique.

Deux implantations des langages synchronisés ont été effectuées ([85] pour les grammaires synchronisées et [52] pour les systèmes de contraintes). Ils ont montré la difficulté à manipuler ces formalismes en pratique. G. Salzer a effectué un premier prototype d'implantation de nos règles de transformations de programmes logiques. Ce prototype permet de passer du formalisme des systèmes de contraintes vers celui de la programmation logique (et vice-versa) et surtout il implémente toutes les opérations (sauf le complément) sur les langages synchronisés contrairement aux précédents travaux. Ce prototype demande évidemment à être optimisé et complété pour pouvoir spécifier les différentes stratégies d'application des règles.

Chapitre 4

Applications des langages de n-uplets d'arbres

Dans ce chapitre, nous présentons plusieurs applications des résultats sur les langages de n-uplets d'arbres pour montrer des résultats dans les domaines de la réécriture et de la vérification de systèmes à états infinis. Nous nous intéressons plus particulièrement à deux problèmes. Le première est l'accessibilité dans les systèmes de réécriture (est-ce qu'un terme t est accessible à partir d'un terme s par réécriture ? ou en d'autres termes est-ce que $s \rightarrow_R^* t$?). En fait, nous nous intéressons à une version plus général de ce problème qui cherche l'ensemble des termes accessibles à partir d'un ensemble de termes initiaux et notamment est-ce le système de réécriture préserve la régularité c'est à dire si l'ensemble des termes initiaux est régulier est-ce que l'ensemble des termes accessibles est aussi régulier ? Le deuxième problème qui nous intéresse est la joignabilité, c'est à dire étant donnés deux termes s et t quelles sont les substitutions σ telles que $s\sigma \downarrow_R t\sigma$? Les résultats que nous allons montrer, s'appuient sur une procédure qui calcule à partir d'un système de réécriture R, un programme logique qui simule la relation $\rightarrow_R^*$ induite par R.

Ce chapitre se décompose de la manière suivante : nous présentons d'abord la procédure de transformation d'un système de réécriture en un programme logique ainsi que les propriétés de celle-ci, ensuite nous utilisons cette procédure afin d'exhiber deux classes de systèmes de réécriture qui préservent la régularité, nous montrons comment ces résultats peuvent s'appliquer dans le cadre de la vérification de systèmes infinis, enfin nous définissons une méthode générale pour résoudre la data-joignabilité en utilisant des langages d'arbres et nous appliquons cette méthode pour résoudre les formules pseudo-regulières du premier ordre.

4.1 Codage de la réécriture basique par des programmes logiques

Cette section décrit comment nous codons la relation de réécriture par un programme logique. Le but de ce codage est d'obtenir des programmes logiques pour lesquels il est possible de montrer qu'il préserve la régularité. La transformation que nous présentons ici fonctionne pour n'importe quel système de réécriture mais ne produit pas nécessairement un cs-programme.

Une des principales différences entre la réécriture de terme et la programmation logique est qu'en programmation logique on fait une distinction entre les symboles de prédicat et les symboles de fonction, c'est à dire entre les données et les opérations qu'on applique dessus. Cette distinction n'est généralement pas faite dans le cadre de la réécriture (sauf dans le cadre des systèmes de réécriture basés sur les constructeurs). Notre but étant de montrer des propriétés des systèmes de réécritures via des cs-programmes, il faut que la transformation des règles de réécriture en clauses préserve au mieux la struture des termes. Le prix à payer pour cela est que le programme logique obtenu ne code qu'une forme restreinte de la réécriture que l'on a appelée réécriture basique.

4.1.1 La réécriture basique

La réécriture basique est une stratégie de réécriture que se situe entre la réécriture générale où aucune distinction n'est faite entre les opérations et les données, et la réécriture basée sur les constructeurs où cette dinstinction est faite a priori en séparant les symboles de données et les symboles de fonction définie. La réécriture basique peut être vue comme un cas particulier de la surréduction basique [45]. Dans les dérivations de réécriture basique, les parties de termes qui ont été considérées une fois comme des données, c'est à dire qui ont été inclus dans la substitution d'une étape de réécriture, ne peuvent plus se réécriture par la suite dans la dérivation. Une telle stratégie n'est pas complète en générale mais nous montrons qu'elle l'est pour une grande classe de systèmes de réécriture, celle des systèmes linéaires droits.

Définition 4.1.1. Soient R un système de réécriture, s, t des termes et P, Q des ensemble de positions tels que $P \subseteq Pos(s)$ et $Q \subseteq Pos(t)$. On définit la réécriture sur les paires (s, P) de la manière suivante :

$$\begin{array}{lll} (s,P) \overset{bas}{\rightarrow}_{[u,l\rightarrow r]} (t,Q) & \text{ssi} & s \rightarrow_{[u,l\rightarrow r]} t \\ & \text{et} & Q = \{\, v \in P \mid u \not\leq v \,\} \cup u.\Sigma Pos(r) \\ & \text{et} & u \in P \end{array}$$

$(s, P) \overset{bas}{\rightarrow}_{[u,l\rightarrow r]} (t, Q)$ s'écrit aussi $s \overset{bas(P)}{\longrightarrow} t$ ou simplement $s \overset{bas}{\rightarrow} t$. Une telle étape s'appelle une étape P-basique, et P est appelé l'ensemble des *positions basiques* de s.

Une dérivation de réécriture $s \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$ est appelée P-basique si $s \overset{bas(P)}{\longrightarrow} t_1 \overset{bas}{\rightarrow} \cdots \overset{bas}{\rightarrow} t_n$ et se note $s \overset{bas(P)^*}{\longrightarrow} t_n$. La *réécriture basique* est la relation $\overset{bas^*}{\rightarrow} = \{\, (s,t) \mid s \overset{bas(Pos(s))^*}{\longrightarrow} t \,\}$. Pour un ensemble de termes E, $R^*_{bas}(E) = \{\, t \mid s \overset{bas^*}{\rightarrow} t \text{ avec } s \in E \,\}$.

Exemple 4.1.2. Soient le système de réécriture constitué de l'unique règle $r_1 = f(g(x)) \rightarrow x$ et le terme $f(g(f(g(0))))$. La dérivation

$$f(g(f(g(0)))) \rightarrow_{[1.1,r_1]} f(g(0)) \rightarrow_{[\varepsilon,r_1]} 0$$

est une dérivation $\{\varepsilon, 1, 1.1\}$-basique, alors que

$$f(g(f(g(0)))) \rightarrow_{[\varepsilon,r_1]} f(g(0)) \rightarrow_{[\varepsilon,r_1]} 0$$

ne l'est pas car l'ensemble des positions basiques est vide après la première étape de réécriture :

$$Q = \{\, v \in \{\varepsilon, 1, 1.1\} \mid \varepsilon \not\leq v \,\} \cup \varepsilon.\emptyset = \emptyset \cup \emptyset = \emptyset$$

L'intuition est la suivante : dans la seconde dérivation, le sous-terme $f(g(0))$ est une donnée pour la première étape de réécriture. Il est inclus dans la substitution $\{x \mapsto f(g(0))\}$ qui filtre le membre gauche de la règle avec le terme initial. Par conséquent cette partie de terme ne peut plus se réécriture par la suite.

Remarque 4.1.3. En observant la définition, on peut facilement déduire que l'ensemble des positions basiques d'un terme intermédiaire dans une dérivation P-basique dépend à la fois de P et des étapes de réécriture basique précédentes. Par conséquent si $S \subseteq P$, toute dérivation S-basique est aussi P-basique.

La réécriture basique n'est pas une stratégie complète pour un système de réécriture général mais nous montrons qu'elle l'est pour les systèmes linéaires droit. La démonstration de cette propriété nécessite la définition suivante dont le but est de tracer les parties de termes qui ont été inclus dans une substitution lors d'une étape de réécriture.

Définition 4.1.4. Soit R un système de réécriture. Pour l'étape de réécriture $s \rightarrow_{[u,l\rightarrow r,\sigma]} t$, L'ensemble $Ant_{s\rightarrow t}(v)$ est défini pour tout $v \in Pos(t)$ et vaut

- $\{v\}$ si $v \in Pos(t)$ et v est incomparable avec u

– $\emptyset$ si soit $v \leq u$ soit $v = u.u'$ et $u' \in \Sigma Pos(r)$
– $\{ u.u'.w \mid l|_{u'} = x \}$ si $v = u.u''.w$ et $r|_{u''} = x$

Pour une dérivation $t_0 \to^* t_n$, $Ant^*_{t_0 \to^* t_n}(u)$ est défini pour tout $u \in Pos(t_n)$ comme l'ensemble $\{u\}$ si $n = 0$ et $\{ w \in Ant^*_{t_0 \to^* t_{n-1}}(v) \mid v \in Ant_{t_{n-1} \to t_n}(u) \}$ sinon.

Exemple 4.1.5. Considérons le système

$$\{f(s(x), x) \to g(x), g(x) \to x\}$$

et la dérivation

$$c(x, f(s(0), 0)) \to c(x, g(0)) \to c(x, 0)$$

On a

$$\begin{aligned} Ant_{c(x,g(0)) \to c(x,0)}(2) &= \{2.1\} \\ Ant_{c(x,f(s(0),0)) \to c(x,g(0))}(2.1) &= \{2.1.1, 2.2\} \\ Ant^*_{c(x,f(s(0),0)) \to^* c(x,0)}(2) &= \{2.1.1, 2.2\} \end{aligned}$$

Notons que $Ant_{c(x,f(s(0),0)) \to c(x,g(0))}(2) = \emptyset$ et $Ant_{c(x,f(s(0),0)) \to c(x,g(0))}(1) = \{1\}$.

Théorème 4.1.6. *Si R un système de réécriture linéaire droit alors $\to^*_R$ et $\overset{bas}{\to}{}^*_R$ définissent la même relation.*

Preuve. Nous devons montrer que si $t \to^*_R t'$ alors $t \overset{bas}{\to}{}^*_R t'$ (l'inverse est évident). Nous montrons que dans une dérivation $t \to^*_R t'$ qui n'est pas basique on peut toujours commuter les étapes non-basiques de telle manière que la dérivation devient basique.

Soit $t_1 \to_{[u_1, l_1 \to r_1, \sigma_1]} t_2 \to_{[u_2, l_2 \to r_2, \sigma_2]} t_3$ telle que $Ant_{t_1 \to t_2}(u_2) \neq \emptyset$ alors

$$t_1 \to^*_{[u \in Ant_{t_1}(u_2), l_2 \to r_2, \sigma_2]} t'_2 \to_{[u_1, l_1 \to r_1, \sigma'_1]} t_3$$

Si u_1 et u_2 ne sont pas comparables, c'est évident. Si $u_1 \leq u_2$ alors on a $u_2 = u_1.u'.w$ et $u_3 = u_1.u''.w$ pour tout $u_3 \in Ant_{t_1 \to t_2}(u_2)$ où $r_1|_{u'} = x$ et $l_1|_{u''} = x$. Donc les deux étapes de réécriture peuvent s'écrire

$$t_1 \to t_1[u_1 \leftarrow r_1\sigma_1] \to t_1[u_1 \leftarrow r_1\sigma_1[u' \leftarrow r_2\sigma_2]] = t_3$$

Par conséquent, pour tout $u_3 \in Ant_{t_1 \to t_2}(u_2)$, on a $t_1|_{u_3} = l_2\sigma_2$ donc si on applique sur t_1 la règle $l_2 \to r_2$ avec comme substitution σ_2 sur toutes les occurrences de $Ant_{t_1 \to t_2}(u_2)$, on obtient le terme t'_2 tel que $t'_2 = t_1[u_3 \leftarrow r_2\sigma_2 | u_3 \in Ant_{t_1 \to t_2}(u_2)]$ par conséquent $t'_2|_{u_1} = l_1\sigma'_1$ avec $y\sigma'_1 = y\sigma_1$ si $y \neq x$ et $x\sigma'_1 = x\sigma_1[u' \leftarrow r_2\sigma_2]$ (x étant égale à $r|_{u'}$). Ce qui signifie que $t'_2 \to_{[u_1, l_1 \to r_1, \sigma'_1]} t'_2[u_1 \leftarrow r_1\sigma'_1] = t_1[u_1 \leftarrow r_1\sigma'_1] = t_1[u_1 \leftarrow r_1\sigma_1[u' \leftarrow r_2\sigma_2]] = t_3$ (remarquons que u' est unique car r_1 est linéaire droit).

Par induction, on obtient que si $t_1 \to_{[u_1, l_1 \to r_1, \sigma_1]} t'_1 \to^* t_2 \to_{[u_2, l_2 \to r_2, \sigma_2]} t_3$ avec $Ant^*_{t_1 \to^* t_2}(u_2) \neq \emptyset$ alors $t_1 \to^*_{[u \in Ant^*_{t_1 \to^* t_2}(u_2), l_2 \to r_2, \sigma_2]} t''_1 \to^* t'_2 \to_{[u_1, l_1 \to r_1, \sigma'_1]} t_3$.

Grâce à ce résultat on peut montrer que si $t \to^* t'$ alors $t \to^* t'$ par une dérivation telle que $t_i \to_{[u_i, l_i \to r_i, \sigma_i]} t_{i+1}$ et $t_j \to_{[u_j, l_j \to r_j, \sigma_j]} t_{j+1}$, $i < j$ implique que $u_i \not\leq v_j$ pour tout $v_j \in Ant^*_{t_i \to t_j}(u_j)$ ce qui assure que la dérivation est basique.

Le lemme suivant nous indique que dans le cadre de la réécriture basée sur les constructeurs, les data-dérivations sont des dérivations basiques.

Lemme 4.1.7. *Soit R un système de réécriture basé sur les constructeurs, toute data-dérivation $s \to^*_R t$ est une dérivation basique.*

Preuve. On peut le montrer par une induction sur la longueur l de la dérivation. Pour $l = 0$ la propriété est trivialement vérifiée. Si $l > 0$, on a $s \to s[u \to r\sigma] \to^*_R t$. Par hypothèse d'induction $s[u \to r\sigma] \to^*_R t$ est une dérivation basique, la première étape est aussi une étape basique en considérant $Pos(s)$ comme ensemble de positions basiques initiale, or σ est une data-substitution donc aucune étape de réécriture ne peut s'appliquer sur les occurrences apportées par σ dans la data-dérivation $s[u \to r\sigma] \to^*_R t$, donc l'ensemble de la dérivation $s \to^*_R t$ est basique.

Dans la définition de $\overset{bas}{\rightarrow}^*$, l'ensemble des positions basiques considéré est le plus grand possible, et donc il peut inclure des positions qui ne seront jamais réduites par une règle de réécriture, même après avoir réduit certains de leurs sous-termes). On peut restreindre l'ensemble des positions basiques tout en préservant la relation $\overset{bas}{\rightarrow}^*$ en utilisant la notion de redex possible (voir définition 2.3.2).

Définition 4.1.8. Soient R un système de réécriture, s, t des termes et P, Q des ensembles de positions tels que $P \subseteq Pos(s)$ et $Q \subseteq Pos(t)$. Soit $(s, P) \Rightarrow_{[u,l\rightarrow r]} (t, Q)$ défnit de la même manière que $(s, P) \rightarrow_{[u,l\rightarrow r]} (t, Q)$ dans la définition 4.1.1, sauf que Q est l'ensemble $\{ v \in P \mid u \not\leq v \} \cup u.PRedPos(r)$, c'est à dire ΣPos est remplacé par $PRedPos$. Cette relation est notée $s \overset{bas(P)}{\Longrightarrow} t$ ou $s \overset{bas}{\Rightarrow} t$. $s \overset{bas(P)}{\Longrightarrow}^* t_n$ est une abbréviation de $s \overset{bas(P)}{\Longrightarrow} t_1 \overset{bas}{\Rightarrow} \cdots \overset{bas}{\Rightarrow} t_n$.

Lemme 4.1.9. *$s \overset{bas(Pos(s))}{\longrightarrow}^* t$ ssi $s \overset{bas(PRedPos(s))}{\Longrightarrow}^* t$.*

Preuve. Soient le terme s et les ensembles de positions $P \subseteq Pos(s)$ et $Q = P \cap PRedPos(s)$. On montre que l'étape $s \rightarrow t$ est P-basique ssi elle est Q-basique, et que l'ensemble P' des positions basiques de t d'après le définition 4.1.1 et l'ensemble correspondant Q' d'après la définition 4.1.8 satisfont $Q' = P' \cap PRedPos(t)$.

Si $s \rightarrow_{[u,l\rightarrow r]} t$ est Q-basique, elle est P-basique car $Q \subseteq P$ (voir remarque 4.1.3). L'inverse est aussi vrai car si $s|_u = l\sigma$ alors $u \in PRedPos(s)$. Maintenant soient $P' = e_1 \cup e_2$ et $Q' = e_3 \cup e_4$ les ensembles de positions basiques de t d'après les définitions 4.1.1 et 4.1.8 respectivement, où $e_1 = \{ v \in Pos(s) \mid u \not\leq v \}$, $e_2 = u.\Sigma Pos(r)$, $e_3 = \{ v \in PRedPos(s) \mid u \not\leq v \}$, et $e_4 = u.PRedPos(r)$. De plus, on a $PRedPos(t) = e_3 \cup e_4$ car $t = s[u\rightarrow r\sigma]$. Comme $e_3 \subseteq e1$ et $e_4 \subseteq e_2$ on obtient $(e_3 \cup e_4) \subseteq (e_1 \cup e_2)$ et donc $(e_1 \cup e_2) \cap (e_3 \cup e_4) = e_3 \cup e_4$. Ce qui nous permet de conclure que $Q' = P' \cap PRedPos(t)$. Comme $PRedPos(t) \subseteq Pos(t)$ pour tout terme t, on obtient $PRedPos(t) \cap Pos(t) = PRedPos(t)$. Donc le lemme peut être montré par une induction sur la longueur de la dérivation.

D'après le lemme 4.1.9, on peut utiliser la définiton 4.1.8 dans la suite de ce document.

4.1.2 Transformation des systèmes de réécriture en programmes logiques

La figure 4.1 décrit les règles de transformations des termes en ensemble d'atomes et des règles de réécriture en clauses de Horn. Il faut bien remarquer que ces règles sont indépendantes de la notion de redex possible choisie. Pour un système de réécriture R, la clause obtenue en appliquant la quatrième règle de la figure 4.1 sur $l\rightarrow r$ est notée $\mathcal{LP}(l\rightarrow r)$ et $\mathcal{LP}(R)$ est l'ensemble de clauses obtenues pour les règles de R.

La nouvelle variable introduite par la troisième règle pour le sous-terme $f(s_1, \dots, s_n)$ à la position u du membre droit s de la règle de réécriture, est notée x_u et A_u est l'atome produit par cette règle.

Exemple 4.1.10. Les règles de réécriture ainsi que les clauses suivantes spécifient la multiplication et l'addition.

$$\begin{array}{lcl}
*(0, x) \rightarrow 0 & \rightsquigarrow & P_*(0, x, 0) \leftarrow \\
*(s(x), y) \rightarrow +(y, *(x, y)) & \rightsquigarrow & P_*(s(x), y, x_\varepsilon) \leftarrow P_+(y, x_2, x_\varepsilon), P_*(x, y, x_2) \\
+(0, x) \rightarrow x & \rightsquigarrow & P_+(0, x, x) \leftarrow \\
+(s(x), y) \rightarrow s(+(x, y)) & \rightsquigarrow & P_+(s(x), y, s(x_1)) \leftarrow P_+(x, y, x_1)
\end{array}$$

Soit $\mathcal{P}_{Id}$ l'ensemble de clauses $\{ P_f(\vec{x}, f(\vec{x})) \leftarrow \ \mid f \in \mathcal{F} \}$ exprimant la réfexivité de $\rightarrow^*$. Cet ensemble de clauses nous permet de pouvoir arrêter la dérivation de réécriture à n'importe quelle étape.

Théorème 4.1.11. *Soient R un système de réécriture, s un terme et $s \rightsquigarrow \langle s', \mathcal{G} \rangle$. $s \overset{bas}{\rightarrow}^* t$ ssi il existe une substitution μ telle que $t = s'\mu$ et $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{G}\mu$.*

$$v \rightsquigarrow \langle v, \emptyset \rangle \quad \text{si } v \in \mathit{Var}$$

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \quad \cdots \quad s_n \rightsquigarrow \langle t_n, \mathcal{G}_n \rangle}{f(s_1, \ldots, s_n) \rightsquigarrow \langle f(t_1, \ldots, t_n), \bigcup_i \mathcal{G}_i \rangle} \quad \text{si } \varepsilon \notin \mathit{PRedPos}_R(f(s_1, \ldots, s_n))$$

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \quad \cdots \quad s_n \rightsquigarrow \langle t_n, \mathcal{G}_n \rangle}{f(s_1, \ldots, s_n) \rightsquigarrow \langle x, \{P_f(t_1, \ldots, t_n, x)\} \cup \bigcup_i \mathcal{G}_i \rangle} \quad \text{si } \varepsilon \in \mathit{PRedPos}_R(f(s_1, \ldots, s_n))$$

$$\frac{s \rightsquigarrow \langle t, \mathcal{G} \rangle}{f(s_1, \ldots, s_n) \to s \rightsquigarrow P_f(s_1, \ldots, s_n, t) \leftarrow \mathcal{G}} \quad \text{si } f(s_1, \ldots, s_n) \to s \in R$$

FIG. 4.1 – Convertir les règles de réécriture en clauses de Horn

Le théorème 4.1.11 va être montré par l'intermédiaire des lemmes 4.1.12 et 4.1.14.

Lemme 4.1.12. *Soient R un système de réécriture, s un terme et $s \rightsquigarrow \langle s', \mathcal{G} \rangle$. Si $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{G}\mu$ pour une substitution μ, alors $s \overset{bas}{\to}^* s'\mu$.*

Preuve. $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{G}\mu$ ssi il existe une preuve dont le but initial est $\mathcal{G}\mu$. On utilise une induction sur la longueur l de la preuve.

$l = 0$: Cela signifie que $s \rightsquigarrow \langle s, \emptyset \rangle$, donc s est irréductible donc $s \overset{bas}{\to}^* s$ en 0 étape.

$l > 0$: Soit A_u l'atome choisi et $P_f(t_1, \ldots, t_n, t) \leftarrow \mathcal{B}$ la clause utilisée lors de la première étape de la preuve. Cette clause appartient soit à $\mathcal{P}_{Id}$, soit à $\mathcal{LP}(R)$. Dans le premier cas la clause est donc $P_f(x_1, \ldots, x_n, f(x_1, \ldots, x_n)) \leftarrow$ et le but résultat est $(\mathcal{G} \setminus A_u)\mu$. $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models (\mathcal{G} \setminus \{A_u\})\mu$ avec une preuve de longueur inférieure à l, donc $s \overset{bas}{\to}^* s'\mu$. Dans le second cas, soit $f(s_1, \ldots, s_n) \to r$ la règle de réécriture qui a produit cette clause. Le prochain but dans la preuve est $(\mathcal{G} \setminus A_u)\mu \cup \mathcal{B}\sigma$. Soit $s'' = s[u \leftarrow \sigma r]$, $s'' \rightsquigarrow \langle s', \mathcal{G} \setminus \{ A_v \mid u \leq v \} \cup \mathcal{B}\sigma \cup \mathcal{B}' \rangle$ où $\mathcal{B}'$ est l'ensemble des atomes apportés par σ. $(\mathcal{G} \setminus \{ A_v \mid u \leq v \})\mu \cup \mathcal{B}\sigma \subseteq (\mathcal{G} \setminus A_u)\mu \cup \mathcal{B}\sigma$ donc $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models (\mathcal{G} \setminus \{ A_v \mid u \leq v \})\mu \cup \mathcal{B}\sigma$ avec une preuve de longueur inférieure à l, et $\mathcal{P}_{Id} \models \mathcal{B}'$ donc $s[u \leftarrow \sigma r] \overset{bas}{\to}^* s'\mu \uplus \sigma = s'\mu$ et aucune règle de réécriture n'a été appliquées sur les positions de $r\sigma$ apportées par σ. Maintenant, considérons le terme $s|_u$. $s|_u \rightsquigarrow \langle x_u, \bigcup_{u \leq v, v \in \mathit{PRedPos}(s|_u)} A_v \rangle$. $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \{ A_v\mu \mid u \leq v \}$ avec une preuve de longueur inférieure à l, donc $s|_u \overset{bas}{\to}^* x_u\mu = r\sigma$. Par conséquent $s \overset{bas}{\to}^* s[u \leftarrow r\sigma] \overset{bas}{\to}^* s'\mu$. Dans la seconde partie de la dérivation aucun sous-terme apporté par σ n'a été réécrit donc toute la dérivation est basique.

Pour la preuve du lemme 4.1.14, nous introduisons les notations suivantes.

Définition 4.1.13. Pour une dérivation basique $s \overset{bas}{\to}^* t$ on définit le terme $(s \overset{bas}{\to}^* t) \setminus u$ comme suit :

- $(s \overset{bas}{\to}^* t) \setminus u = s|_u$ si la longueur de la dérivation est 0
- si la dérivation est de la forme $s \to s[v \leftarrow r\sigma] \overset{bas}{\to}^* t$, $(s \overset{bas}{\to}^* t) \setminus u = s|_u$ si $v < u$ et $(s \overset{bas}{\to}^* t) \setminus u = (s[v \leftarrow r\sigma] \overset{bas}{\to}^* t) \setminus u$ sinon.

Intuitivement, $(s \overset{bas}{\to}^* t) \setminus u$ désigne le terme obtenu en appliquant les étapes de réécriture de $s \overset{bas}{\to}^* t$ sur $s|_u$.

Lemme 4.1.14. *Soient R un système de réécriture, s un terme et $s \rightsquigarrow \langle s', \mathcal{G} \rangle$. Si $s \overset{bas}{\to}^* t$ alors $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{G}\delta$ où $\delta = \{ x_v \mapsto (s \overset{bas}{\to}^* t) \setminus v \mid v \in \mathit{PRedPos}(s) \}$ et $s'\delta = t$.*

Preuve. On fait cette preuve par induction sur la longueur k de la dérivation.

$k = 0$: On a $s = t$ et tous les $A_u \in \mathcal{G}$ sont de la forme $P_f(s'_1, \ldots, s'_n, x_u)$, où $f(s'_1, \ldots, s'_n)$ est le redex possible de s à la position u. Notons que pour toute position de redex possible $u.v$ de s il existe dans $f(s'_1, \ldots, s'_n)$ la position v et $f(s'_1, \ldots, s'_n)|_v = x_v$. Comme la longueur de la dérivation est 0, $x_u\delta = s|_u$ par définition, ce qui signifie que $s'_i\delta = s|_{u.i}$. Par conséquent $P_f(s'_1, \ldots, s'_n, x_u)\delta = P_f(s|_{u.1}, \ldots, s|_{u.n}, s|_u)$ qui est une instance de $P_f(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$ une tête de clause de $\mathcal{P}_{Id}$ (notons que $s|_u = f(s|_{u.1}, \ldots, s|_{u.n})$). On peut donc déduire que $\mathcal{P}_{Id} \models \mathcal{G}\delta$ et $s'\delta = t$ dans ce cas.

$k > 0$: $s \rightarrow_{[u,l\rightarrow r,\sigma]} t' \overset{bas}{\rightarrow}^* t$. Par la définition 4.1.8 de la réécriture basique $u \in \mathit{PRedPos}(s)$ par conséquent $u \in O_{\mathcal{G}}$. Soient $A_u = P_f(s_1, \ldots, s_n, x_u)$ et $P_f(s'_1, \ldots, s'_n, s'') \leftarrow \mathcal{B}$ la clause de $\mathcal{LP}(R)$ produite par $l \rightarrow r$. On a $t' \rightsquigarrow \mathcal{G} \setminus \{ A_v \mid v > u \} \cup \{ A_v \mid v \in \mathit{PRedPos}(r\sigma) \}$ et $t' \overset{bas}{\rightarrow}^* t$ est de longueur inférieure à k donc $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models (\mathcal{G} \setminus \{ A_v \mid v > u \} \cup \{ A_v \mid v \in \mathit{PRedPos}(r\sigma) \})\delta'$ où $\delta' = \{ x_v \mapsto (t' \overset{bas}{\rightarrow}^* t) \setminus v \mid v \in \mathit{PRedPos}(t') \}$. Par définition, pour tout $v \not> u$, $(s \overset{bas}{\rightarrow}^* t) \setminus v = (t' \overset{bas}{\rightarrow}^* t) \setminus v$, donc pour tout $v \not> u$, $x_v\delta' = x_v\delta$ et $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models (\mathcal{G} \setminus \{ A_v \mid v > u \})\delta$.

Maintenant nous devons montrer que $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \{ A_v \in \mathcal{G} \mid v \geq u \}\delta$. Par définition, pour tout $v > u$, $(s \overset{bas}{\rightarrow}^* t) \setminus v = s|_v$ donc, comme dans le cas $k = 0$, $s_i\delta = s|_{u.i}$ par conséquent $\mathcal{P}_{Id} \models \{ A_v \in G \mid v > u \}$ d'une part et $f(s_1, \ldots, s_n)\delta = l\sigma = f(s'_1, \ldots, s'_n)\sigma$ d'autre part. De plus $x_u\delta = (s \overset{bas}{\rightarrow}^* t) \setminus u = (t' \overset{bas}{\rightarrow}^* t) \setminus u$ et $s'' = \mathit{Irr}(r)$ donc $(t' \overset{bas}{\rightarrow}^* t) \setminus u = \mathit{Irr}(r)[v \leftarrow (s \overset{bas}{\rightarrow}^* t) \setminus u|_v, v \in \mathit{PRedPos}(r)]$. Soit $\sigma' = \sigma \cup \{x_v \mapsto (s \overset{bas}{\rightarrow}^* t) \setminus u|_v\}$. On a $P_f(s_1, \ldots, s_n, x_u)\delta = P_f(s'_1, \ldots, s'_n, s'')\sigma'$ et donc $\mathcal{G}\delta$ est une conséquence logique de $(\mathcal{G} \setminus \{A_u\})\delta \cup \mathcal{B}\sigma'$. $\mathcal{B}\sigma' \subseteq (\{ A_v \mid v \in \mathit{PRedPos}(r\sigma) \})\delta'$ d'où $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{B}\sigma'$.

Il reste à prouver que $s'\delta = t$. Par l'hypothèse d'induction, on sait que $\mathit{Irr}(t')\delta' = t$. $s' = \mathit{Irr}(s)$ et $s \rightarrow_{[u,l\rightarrow r,\sigma]} t'$. Donc, soit $x_u \notin \mathit{Var}(s')$ et dans ce cas $s' = \mathit{Irr}(t')$, soit $x_u \in \mathit{Var}(s')$ et dans ce cas $\mathit{Irr}(t') = s'[u \leftarrow \mathit{Irr}(r\sigma)]$. Dans le premier cas, comme $x_v\delta = x_v\delta'$ pour tout $v \not\geq u$, on obtient $s'\delta = \mathit{Irr}(t')\delta' = t$. Dans le second cas, $x_u\delta = (s \overset{bas}{\rightarrow}^* t) \setminus u = (t' \overset{bas}{\rightarrow}^* t) \setminus u = t|_u$, par conséquent $s'\delta = \mathit{Irr}(t')\delta'[u \leftarrow x_u\delta] = \mathit{Irr}(t')\delta'[u \leftarrow t|_u] = t[u \leftarrow t|_u] = t$.

Dans l'énoncé du théorème 4.1.11 est apparu l'ensemble de clauses $\mathcal{P}_{Id}$ qui est crucial pour permettre coder la réflexivité de $\rightarrow^*$. Si cet ensemble de clauses n'est pas inclus on obtient un résultat différent qui prend tout son sens dans le cadre de la réécriture basée sur les constructeurs comme nous le verrons dans la section 4.4.

Malheureusement – mais ce n'est pas une surprise – la transformation d'un système de réécriture arbitraire ne produit pas un cs-programme car en général on obtient des corps de clauses non-lineaires et avec des symboles de fonction. La non-linéarité des corps a elle même deux causes : la non-linéarité droite des règles de réécriture d'une part et l'existance de positions de redex possible imbriquées d'autre part. Par conséquent les systèmes de réécriture dont les dérivations basiques peuvent s'exprimer par un cs-programme doivent être linéaires droit, et ne pas avoir de positions de redex possible imbriquées. De plus pour éviter les symboles de fonction dans le corps des clauses il faudrait que les redex possibles des membres droits soient plats c'est à dire de la forme $f(x_1, \ldots, x_n)$ où f est une symbole de fonction et $x_1, \ldots, x_n$ sont des variables. Les définitions 4.1.15 et 4.1.16 donnent deux classes de systèmes de réécriture où la condition sur les redex possibles plats est affaiblie grâce au programmes quasi-cs et ib. Afin de bien fixer les définitions de ces deux classes nous utilisons la définition 2.3.4 comme concrétisation de la notion de position de redex possible mais on pourrait raffiner ces deux classes en raffinant la notion de position de redex possible.

Définition 4.1.15. Un système de réécriture R est dit *quasi-cs* s'il a les propriétés suivantes :
- il est linéaire droit,
- aucun membre droit ne contient de redex possibles imbriqués,
- pour toutes règles $l \rightarrow r$ de R et pour toute variable x de $\mathit{PRedVar}(r) \cap \mathit{Var}(l)$, $\mathit{PRedDepth}(x)$ est plus petit ou égal à la profondeur minimale où x apparaît dans l.

Définition 4.1.16. Un système de réécriture R est dit *à base d'instance* ou *ib* s'il a les propriétés suivantes :

– il est linéaire droit,
– aucun membre droit ne contient de redex possibles imbriqués,
– pour tout redex possible C d'un membre droit de règle et tout membre gauche l qui s'unifie avec C, l est une instance de C.

Définition 4.1.17. Soient R un système de réécriture et $\mathcal{P}$ un programme logique. On dit que $\mathcal{P}$ *encode* $\rightarrow_R^*$ si pour tous termes s et t, $s \rightarrow_R^* t$ ssi pour une substitution μ, $\mathcal{P} \models \mathcal{G}\mu$ et $t = s'\mu$ où $s \rightsquigarrow \langle s', \mathcal{G} \rangle$.

Théorème 4.1.18. *Soit R un système de réécriture quasi-cs ou ib. Il existe un cs-programme $\mathcal{P}$ qui encode $\rightarrow_R^*$.*

Preuve. Les systèmes de réécriture quasi-cs et ib sont linéaires droits donc par le théorème 4.1.6, $\rightarrow^*$ et $\stackrel{bas}{\rightarrow}^*$ sont identiques pour ces deux classes. Par le théorème 4.1.11 on sait que pour tous termes s et t, $s \rightarrow_R^* t$ ssi $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{G}\mu$ et $t = s'\mu$ où $s \rightsquigarrow \langle s', \mathcal{G} \rangle$.

Montrons maintenant que $\mathcal{LP}(R) \cup \mathcal{P}_{Id}$ peut être transformé en un cs-programme équivalent. Les clauses de $\mathcal{P}_{Id}$ sont évidemment des cs-clauses, par conséquent il suffit de montrer que le programme logique $\mathcal{LP}(R)$ est soit soit un programme quasi-cs soit un programme ib. Comme les systèmes de réécriture quasi-cs et ib sont linéaires droits et ne contiennent pas de redex possibles imbriqués dans leur membres droits, $\mathcal{LP}(R)$ ne contient que des clauses dont le corps est linéaire. De plus, si R est quasi-cs, chacune de ses règles $l \rightarrow r$ transformée en $H \leftarrow \mathcal{G}$, est telle que la profondeur d'une variable x dans une redex possible de r est inférieure ou égale à la profondeur minimale ou x apparaît dans l, donc la profondeur de x dans $\mathcal{G}$ est inférieure ou égale à la profondeur minimale où x apparaît dans H. Si R est ib, le fait que chaque redex possible des membres droits de règle de réécriture subsume tous les membres gauches qui s'unifient avec lui, assure que chaque atome apparaissant dans un corps de $\mathcal{LP}(R)$ subsume toutes les têtes de clauses qui s'unifient avec lui. Finalement d'après les théorèmes 3.4.3 et 3.4.6, on sait que dans les deux cas $\mathcal{LP}(R) \cup \mathcal{P}_{Id}$ peuvent se transformer en un cs-programme équivalent.

4.2 Préservation de la régularité

Dans cette section, nous montrons comment utiliser les résultats obtenus sur les cs-programmes pour ainsi que la transformation définie dans la section précédente pour montrer de nouveaux résultats sur la préservation de la régularité dans les systèmes de réécriture. Plus précisément nous montrons que les systèmes de réécriture quasi-cs et ib préservent la régularité à condition que l'ensemble initial possède en plus la propriété de ne contenir qu'un nombre borné de redex possibles imbriqués[1]. En effet, dans [80] il est démontré que si on considère le système de réécriture $R = \{f(g(x)) \rightarrow g(f(x))\}$, qui est à la fois quasi-cs et ib, et le langage régulier $E = (fg)^*0$ le langage $R^*(E)$ n'est pas régulier. $R^*(E)$ est l'ensemble des termes qui contiennent autant de f que de g et en particulier l'ensemble des termes irréductibles obtenus à partir de E est le langage $g^n f^n 0$ n'est pas un langage régulier. Or ce langage peut être obtenu de la manière suivante $R^*(E) \setminus Red(R)$ où $Red(R)$ est l'ensemble des termes reductibles par R. On peut montrer que $Red(R)$ est un langage régulier car R est linéaire droit (voir [21]) donc d'après la clôture de langages réguliers par les opérations ensemblistes si $R^*(E) \setminus Red(R)$ n'est pas régulier alors que $Red(R)$ est régulier alors $R^*(E)$ n'est pas régulier.

On peut remarquer que les termes appartenant à E contiennent un nombre non-borné de redex possibles imbriqués. On va montrer que si E est restreint à un langage régulier dont les éléments contiennent un nombre borné de redex imbriqués alors on peut préserver la régularité.

Définition 4.2.1. Un langage E est dit *à imbrications de redex possibles bornées* s'il existe un entier k tel que chaque terme de E ne contient au plus k redex possibles imbriqués.

Théorème 4.2.2. *Si E est un langage régulier à imbrications de redex possibles bornées et R est un système de réécriture qui est soit quasi-cs soit ib alors $R^*(E)$ est une langage régulier d'arbres.*

[1] Notons que dans cette section, nous utilisons la notion de redex possible de la définition 2.3.4.

Dans la suite de cette section, nous allons prouver ce théorème et montrer qu'il est décidable de déterminer si un langage d'arbres régulier est à imbrications de redex possibles bornées en considérant la définition 2.3.4 pour les redex possibles.

L'idée principale de la preuve est de réduire le calcul de $R^*(E)$ à un ensemble de clauses de jointure dans le cas où R est un système de réécriture soit quasi-cs, soit ib et E est un langage régulier à imbrications de redex possibles bornées. On sait déjà que le programme logique $\mathcal{P} = \mathcal{LP}(R)$ qui encode $\rightarrow_R^*$ est un cs-programme, donc il ne reste plus qu'à montrer que le dernier argument de chaque prédicat de $\mathcal{P}$ est libre dans $\mathcal{P}$ et que E peut s'exprimer en termes de jointures stratifiées où la jointure de la couche la plus "profonde" est une définition de jointure régulière et où les jointures des couches supérieures deviennent à leur tour des définitions de jointures régulières quand les jointures du dessous ont été résolues.

Nous commençons par montrer que le dernier argument de chaque prédicat de $\mathcal{LP}(R)$ est libre dans $\mathcal{LP}(R)$ quand R est soit quasi-cs soit ib. En fait, cette propriété n'est vraie pour la transformation présentée dans la figure 4.1. Un règle de réécriture comme $+(x, 0) \rightarrow x$ est transformée en la clause $P_+(x, 0, x) \leftarrow$. Il est évident que le troisième argument de P_+ n'est pas libre car x apparaît deux fois dans la tête de la clause. Cette situation se produit chaque fois qu'une règle $l \rightarrow r$ est telle que $\mathit{Var}(\mathit{Irr}_R(r)) \cap \mathit{Var}(l) \neq \emptyset$. Pour remédier à ce problème, nous modifions légèrement le programme $\mathcal{LP}(R)$.

Définition 4.2.3. Soit $\mathcal{P}_{Eq}$ l'ensemble de clauses

$$\{ P_{Eq}(f(x_1, \ldots, x_n), f(x'_1, \ldots, x'_n)) \leftarrow P_{Eq}(x_1, x'_1), \ldots, P_{Eq}(x_n, x'_n) \mid f \in \mathcal{F} \}$$

où P_{Eq} est un nouveau symbole de prédicat qui n'apparaît pas dans $\mathcal{LP}(R)$.

Il est évident que P_{Eq} défini l'égalité sur les termes clos : si s et t sont deux termes clos, on a $\mathcal{P}_{Eq} \models P_{Eq}(s, t)$ ssi $s = t$. De plus, on peut remarquer que ce programme est un cs-programme régulier.

Soient $f(\vec{s}) \rightarrow r$ un règle de réécriture et $\mathcal{LP}(f(\vec{s}) \rightarrow r) = P_f(\vec{s}, s) \leftarrow \mathcal{B}$ la clause correspondante. La clause modifiée se définit ainsi

$$\mathcal{LP}'(f(\vec{s}) \rightarrow r) \quad = \quad P_f(\vec{s}, s\sigma_r) \leftarrow \mathcal{B}_r \cup \mathcal{B}$$

où $\sigma_r = \{ x \mapsto x_r \mid x \in \mathit{Var}(\vec{s}) \cap \mathit{Var}(r) \}$ est une substitution qui renomme chaque variable x qui est partagée par les deux côtés de la règle, en la nouvelle variable x_r, et où $\mathcal{B}_r = \{ P_{Eq}(x, x_r) \mid x \in \mathit{Var}(\vec{s}) \cap \mathit{Var}(r) \}$ est un ensemble d'atomes assurant l'égalité de x et x_r. Le programme modifié est définit par

$$\mathcal{LP}'(R) \quad = \quad \{ \mathcal{LP}'(l \rightarrow r) \mid l \rightarrow r \in R \} \cup \mathcal{P}_{Eq} \ .$$

Notons que $x \in \mathit{Var}(\mathit{Irr}(r))$ implique $x \notin \mathit{Var}(\mathcal{B})$, puisque la règle $l \rightarrow r$ est soit quasi-cs soit ib et donc linéaire droite.

Par exemple, pour la règle $+(x, 0) \rightarrow x$ on obtient $\sigma_r = \{x \mapsto x_r\}$ et $\mathcal{B}_r = \{P_{Eq}(x, x_r)\}$. La clause modifiée est donc $P(x, 0, x_r) \leftarrow P_{Eq}(x, x_r)$.

Lemme 4.2.4. *Soit R un système de réécriture soit quasi-cs soit ib. $\mathcal{LP}'(R)$ et $\mathcal{LP}(R)$ sont équivalents et $\mathcal{LP}'(R)$ est soit quasi-cs soit ib.*

Preuve. Observons d'abord que les clauses de $\mathcal{P}_{Eq}$ sont à la fois quasi-cs et ib puisque leur corps ne contient pas de symboles de fonction. Considérons la clause $P_f(s_1, \ldots, s_n, s\sigma_r) \leftarrow \mathcal{B}_r \cup \mathcal{B}$ de $\mathcal{LP}'(R)$ obtenue à partir de la clause $P_f(s_1, \ldots, s_n, s) \leftarrow \mathcal{B}$ de $\mathcal{LP}(R)$. Les atomes de $\mathcal{B}_r$ sont tous de profondeur 0, par conséquent si $P_f(s_1, \ldots, s_n, s) \leftarrow \mathcal{B}$ est une clause quasi-cs (resp ib) alors $P_f(s_1, \ldots, s_n, s\sigma_r) \leftarrow \mathcal{B}_r \cup \mathcal{B}$ est aussi quasi-cs (resp ib). Il est évident que $\mathcal{P}_{Eq} \models P_{Eq}(t_1, t_2)$ ssi $t_1 = t_2$. On en déduit que la sémantique de P_f dans $\mathcal{LP}(R)$ et la sémantique de P_f dans $\mathcal{LP}'(R)$ sont les mêmes du point de vue du plus petit modèle de Herbrand.

Lemme 4.2.5. *Soit R un système de réécriture soit quasi-cs soit ib. Le dernier composant de chaque prédicat de $\mathcal{LP}'(R)$ est libre dans $\mathcal{LP}'(R)$.*

Preuve. On peut déjà remarquer que les deux arguments des prédicats de $\mathcal{P}_{Eq}$ sont libres dans $\mathcal{P}_{Eq}$ car toutes les clause de $\mathcal{P}_{Eq}$ sont linéaires. Maintenant, considérons $f(s_1,\ldots,s_n) \to r \in R$ et $\mathcal{LP}'(f(s_1,\ldots,s_n) \to r) = P_f(s_1,\ldots,s_n,s\sigma') \leftarrow \mathcal{B}_r \cup \mathcal{B}$. On peut observer que les variables de $s\sigma'$ sont linéaires dans $P_f(s_1,\ldots,s_n,s\sigma') \leftarrow \mathcal{B}_r \cup \mathcal{B}$ car r est linéaire (donc $s\sigma'$ est aussi linéaire) et tous les redex possibles de r sont représentés par des variables différentes dans $s\sigma'$. De plus chaque variable de $s\sigma'$ apparaît une seule fois dans $\mathcal{B}_r \cup \mathcal{B}$ et toujours dans le dernier argument des atomes. Par conséquent, $s\sigma'$ partage au plus une variable avec chaque atome de $\mathcal{B}_r \cup \mathcal{B}$ et la variable partagée apparaît toujours dans le dernier argument de cet atome. Donc, le dernier argument de chaque prédicat de $\mathcal{LP}'(R)$ n'est pas non-libre dans $\mathcal{LP}'(R)$, donc il est libre dans $\mathcal{LP}'(R)$.

Le lemme 3.3.12 nous dit que tout langage régulier d'arbres peut être décrit par un cs-programme linéaire monadique. Comme les langages réguliers à imbrications de redex possibles bornées appartiennent à une sous-classe des langages réguliers, on peut aussi les décrire par des cs-programmes linéaires monadiques. L'objectif des lemmes suivants est de montrer qu'il est toujours possible de repérer les clauses d'un cs-programme linéaire monadique qui introduisent un nouveau redex possible. L'algorithme donné dans la figure 4.2 permet de le faire effectivement.

Définition 4.2.6. Soient R un système de réécriture, E un langage régulier d'arbres décrit par le cs-prédicat linéaire monadique P du programme $\mathcal{P}$ et $C = P(f(x_1,\ldots,x_n)) \leftarrow P_1(x_1),\ldots,P_n(x_n)$ une clause de $\mathcal{P}$. La clause C est dite *introduisant un redex possible (PRI en abbrégé)* s'il existe des termes $t_1,\ldots,t_n$ tels que $\mathcal{P} \models P_1(t_1),\ldots,P_n(t_n)$ et $f(t_1,\ldots,t_n)$ contient un redex possible en tête. La clause C est dite *unifiable gauche en tête (LTU en abbrégé)* s'il existe des termes $t_1,\ldots,t_n$ tels que $\mathcal{P} \models P_1(t_1),\ldots,P_n(t_n)$ et $f(t_1,\ldots,t_n)$ s'unifie avec au moins un membre gauche d'un règle de R.

Il est facile de voir que LTU implique PRI mais l'inverse est faux. Par exemple, prenons le système de réécriture

$$R = \{f(s(x)) \to x, g(x) \to s(x)\}$$

et $\mathcal{P}$ le programme

$$\{P(f(x)) \leftarrow P'(x), P'(g(x)) \leftarrow P'(x), P'(0) \leftarrow\}$$

La première clause est PRI car $\mathcal{P} \models P'(g(0))$ et $f(\Box)$ est un redex possible d'après la définition 2.3.4, alors qu'elle n'est pas LTU. Mais notons que $f(g(0)) \to f(s(0)) \to 0$ donc le terme $f(g(0))$, qui est dans le langage décrit par P, peut se réécrire à la position ε après avoir réécrit $g(0)$, donc LTU serait une notion trop restrictive.

Lemme 4.2.7. *Soient R un système de réécriture et E un langage régulier d'arbre décrit par le cs-prédicat linéaire monadique P du programme $\mathcal{P}$. Pour une clause $H \leftarrow \mathcal{B} \in \mathcal{P}$, il est possible de décider si $H \leftarrow \mathcal{B}$ est LTU.*

Preuve. Sans perte de généralité, on peut supposer que les têtes de clauses de $\mathcal{P}$ sont toutes de hauteur exactement 1 (voir le corollaire 3.3.13). Considérons le programme logique $\{ P_{lhs}(l) \leftarrow | \, l \to r \in R \}$. Ce programme est évidemment un cs-programme. Le prédicat P_{lhs} décrit l'ensemble des instances closes des membres gauches des règles de réécriture de R. Soit

$$C = Q(f(x_1,\ldots,x_n)) \leftarrow Q_1(x_1),\ldots Q_n(x_n)$$

une clause de $\mathcal{P}$. La définition

$$Test_C(x_1,\ldots,x_n) \leftrightarrow P_{lhs}(f(x_1,\ldots,x_n)), P(x_1),\ldots,P(x_n)$$

est une définition de jointure régulière compatible avec $\mathcal{P} \cup \mathcal{P}_{lhs}$, donc elle peut être transformée en un cs-programme dont le test du vide est décidable. De plus, nous avons $\mathcal{P} \cup \mathcal{P}_{lhs} \models Test_C(t_1,\ldots,t_n)$ ssi $\mathcal{P} \models Q(f(t_1,\ldots,t_n))$ et $f(t_1,\ldots,t_n)$ est une instance clause du membre gauche d'une règle de R, donc C est LTU.

$$\frac{\langle \mathcal{P}_{in} \cup \{P(f(\vec{x})) \leftarrow \mathcal{B}\}, \mathcal{P}_{out}, \mathcal{P}_{pri}, \mathcal{P}_{aux}\rangle}{\langle \mathcal{P}_{in} \cup \mathcal{P}_{out}, \emptyset, \mathcal{P}_{pri} \cup \{P(f(\vec{x})) \leftarrow \mathcal{B}\}, \mathcal{P}_{aux} \cup \{P(x) \leftarrow\}\rangle} \quad \begin{array}{c} P(f(\vec{x})) \leftarrow \mathcal{B} \text{ est} \\ \text{LTU dans } \mathcal{P}_{all} \end{array}$$

$$\frac{\langle \mathcal{P}_{in} \cup \{P(f(\vec{x})) \leftarrow \mathcal{B}\}, \mathcal{P}_{out}, \mathcal{P}_{pri}, \mathcal{P}_{aux}\rangle}{\langle \mathcal{P}_{in}, \mathcal{P}_{out} \cup \{P(f(\vec{x})) \leftarrow \mathcal{B}\}, \mathcal{P}_{pri}, \mathcal{P}_{aux}\rangle} P(f(\vec{x})) \leftarrow \mathcal{B} \text{ n'est pas LTU dans } \mathcal{P}_{all}$$

où $P_{all} = \mathcal{P}_{in} \cup \mathcal{P}_{out} \cup \mathcal{P}_{pri} \cup \mathcal{P}_{aux}$

FIG. 4.2 – Détecter les clauses PRI

Pour un système de réécriture R, l'algorithme pour repérer les clauses PRI d'un cs-programme linéaire monadique $\mathcal{P}$ décrivant le langage régulier E, est donné par les deux règles de la figure 4.2. Les règles transforment des états de la forme $\langle \mathcal{P}_{in}, \mathcal{P}_{out}, \mathcal{P}_{pri}, \mathcal{P}_{aux}\rangle$ où $\mathcal{P}_{in}$ est l'ensemble des clauses qui ne sont pas détectées comme PRI et pas encore traitées, $\mathcal{P}_{out}$ est l'ensemble des clauses qui ne sont pas détectées comme étant PRI et déjà traitées, $\mathcal{P}_{pri}$ est l'ensemble des clauses PRI et $\mathcal{P}_{aux}$ est un ensemble de clauses qui permet de détecter les clauses PRI en testant simplement si elle est LTU. L'état$\langle \mathcal{P}, \emptyset, \emptyset, \emptyset\rangle$ est appelé *état initial* et les états de la forme $\langle \emptyset, \mathcal{P}_{out}, \mathcal{P}_{pri}, \mathcal{P}_{aux}\rangle$ sont appelés *états finaux*. Une dérivation est une séquence $S_0, S_1, \ldots, S_n$ d'états telle que S_0 est un état initial et S_{i+1} est obtenu à partir de S_i $(0 \leq i \leq n-1)$ en appliquant une des deux règles de la figure 4.2. Une dérivation est complète si S_n est une état final.

Lemme 4.2.8. *Soient R un système de réécriture, $\mathcal{P}$ un cs-programme linéaire monadique $\mathcal{P}$, toute dérivation qui utilise les règles de la figure 4.2 tant que c'est possible est finie et atteint un état final.*

Preuve. On peut remarquer que $\mathcal{P}_{in} \cup \mathcal{P}_{out} \cup \mathcal{P}_{pri} = \mathcal{P}$ et dans toute dérivation $S_0, S_1, \ldots, S_n$ $S_i \neq S_j$ si $i \neq j$ $(i, j \in [0, n])$. Donc il est impossible de construire une dérivation infinie. De plus, si $\mathcal{P}_{in}$ n'est pas vide alors une des deux règles s'applique car leurs conditions d'application sont la négation l'une de l'autre donc le dernier état d'une dérivation que l'on ne peut pas poursuivre est un état final.

Lemme 4.2.9. *Soient R un système de réécriture et $\mathcal{P}$ un cs-prédicat linéaire monadique. Toute dérivation complète des règles décrites dans la figure 4.2 donnant l'état $\langle \emptyset, \mathcal{P}_{out}, \mathcal{P}_{pri}, \mathcal{P}_{aux}\rangle$ est telle que $\mathcal{P}_{pri}$ est l'ensemble des clauses PRI de $\mathcal{P}$ et $\mathcal{P}_{out}$ est l'ensemble des clauses non PRI $\mathcal{P}$.*

Preuve. Il est évident que toutes les clauses de $\mathcal{P}_{pri}$ sont PRI puisque LTU implique PRI. Montrons que $\mathcal{P}_{out}$ ne contient aucune clause PRI de $\mathcal{P}$. Pour un terme clos t, $nn(t)$ indique le nombre d'imbrications de redex possibles. Pour une clause $C = P(f(x_1, \ldots, x_n)) \leftarrow \mathcal{B}$ de $\mathcal{P}$, $mnn(C) = Min(\{\, nn(f(t_1, \ldots, t_n)) \mid \mathcal{P} \models P(f(t_1 \ldots t_n))\,\})$ (si C n'a pas de modèle on pose $mnn(C) = 0$). Nous montrons que toutes les clauses PRI sont repérées par induction sur le mnn des clauses.

Soit $C = P(f(x_1, \ldots, x_n)) \leftarrow \mathcal{B}$ une clause de $\mathcal{P}$ telle que $mnn(C) = 0$. Cela signifie qu'il existe $t_1, \ldots, t_n$ tels que $f(t_1, \ldots, t_n)$ ne contient aucun redex possible et $\mathcal{P} \models P(f(t_1, \ldots, t_n))$ donc soit $f(t_1, \ldots, t_n)$ s'unifie avec un membre gauche de règle et donc C est mis dans l'ensemble $\mathcal{P}_{pri}$ soit elle ne s'unifie avec aucun membre gauche et dans ce cas la clause n'est pas PRI.

Suposons que toutes les clauses PRI C de $\mathcal{P}$ telles que $mnn(C) < k$ sont dans $\mathcal{P}_{pri}$ et montrons que $C' = P(f(x_1, \ldots, x_n)) \leftarrow \mathcal{B}$ telle que C' est une clause PRI de $\mathcal{P}$ et $mnn(C') = k$ est mise dans $\mathcal{P}_{pri}$. Comme C' est PRI et $mnn(C') = n$ il existe $f(t_1, \ldots, t_n)$ tel que $nn(f(t_1, \ldots, t_n)) = k$, $\mathcal{P} \models P(f(t_1, \ldots, t_n))$ et $f(t_1, \ldots, t_n) = D[s_1, \ldots, s_m]$ où D est un redex possible, pour $1 \leq i \leq m$ ε est une position de redex possible de s_i donc $nn(s_i) < k$. Par conséquent, il existe des clauses $C_1, \ldots C_m$ de $\mathcal{P}$ telles que $mnn(C_i) < k$, la tête de chaque C_i est de la forme $P_i(f_i(\vec{x}_i))$ et $\mathcal{P} \models P_i(s_i)$ $(1 \leq i \leq m)$, donc $\{C_1, \ldots C_m\} \subseteq \mathcal{P}_{pri}$ et $\{P_1(x_1) \leftarrow, \ldots, P_m(x_m) \leftarrow\} \subseteq P_{aux}$. D'où $\mathcal{P}_{all} \models P(D[x_1, \ldots, x_m])$ de plus $D[x_1, \ldots, x_m]$ s'unifie avec avec un membre gauche car c'est un redex possible donc C est ajouté à $\mathcal{P}_{pri}$.

Lemme 4.2.10. *Soient R un système de réécriture et E un langage régulier. On peut décider si E est une langage régulier à imbrications de redex possibles bornées.*

Preuve. Comme E est un langage régulier d'arbres, d'après le lemme 3.3.12 on sait qu'il existe un cs-programme linéaire monadique $\mathcal{P}$ définissant le prédicat P qui décrit E. Sans perte de généralité, on peut considérer que $\mathcal{P}$ ne contient aucune clause improductive (c'est à dit qui ne contribuent au modèle minimal) et toutes les têtes de ses clauses sont de profondeur exactement 1. D'après le lemme 4.2.9, on peut partager les clauses de $\mathcal{P}$ en deux ensembles $\mathcal{P}_{out}$ et $\mathcal{P}_{pri}$ où les clauses PRI sont dans $\mathcal{P}_{pri}$. Si E n'est pas à imbrications de redex possibles bornées alors au moins une des clauses de $\mathcal{P}_{pri}$ peut être utilisée un nombre non borné de fois dans une preuve. Comme $\mathcal{P}$ est un cs-programme la profondeur des atomes des corps de clause est 0 donc étant donné un tel atome $Q(x)$ il s'unifie toujours avec les toutes les têtes de clauses $Q(f(\vec{x}))$, donc une clause PRI peut être utilisée un nombre non borné de fois dans une preuve ssi il existe un cycle dans le graphe donc les sommets sont les atomes de $\mathcal{P}$ et dont l'ensemble des arcs est le suivant $\{\langle Q(t), Q'(t')\rangle | Q(t) \leftarrow \mathcal{B}, Q'(t') \in \mathcal{P}$ où $Q = Q'\}$.

Lemme 4.2.11. *Soient R un système de réécriture et E un langage régulier d'arbres à imbrications de redex possibles bornées décrit par le prédicat P du programme $\mathcal{P} \cup \mathcal{P}_{pri}$ où $\mathcal{P}_{pri}$ est l'ensemble des clauses PRI du programme. $\mathcal{P}_{pri}$ peut être stratifié de la manière suivante :*
- *$\mathcal{P}^0_{pri} = \{Q(f(\vec{x})) \leftarrow \mathcal{B}$ telles que la définition de chaque prédicate Q' tel que $Q'(x) \in \mathcal{B}$, ne contient aucune clause PRI $\}$.*
- *$\mathcal{P}^i_{pri} = \{Q(f(\vec{x})) \leftarrow \mathcal{B} \notin \bigcup_{j<i} \mathcal{P}^j_{pri}$ telles que la définition de chaque prédicat Q' tel que $Q'(x) \in \mathcal{B}$ ne contient aucune clause PRI à part celles de $\bigcup_{j<i} L_j\}$.*

Preuve. Supposons que le lemme soit faux. Cela signifie qu'il existe au moins une clause PRI $Q(f(\vec{x})) \leftarrow \mathcal{B}$ telle que la définition d'au moins un prédicat apparaissant dans $\mathcal{B}$ contient $Q(f(\vec{x})) \leftarrow \mathcal{B}$ elle même. Par conséquent le redex possible introduit par cette clause pourrait être trouvé autant de fois que l'on veut dans un terme de E ce qui contredit le fait que E est un langage régulier d'arbres à imbrications de redex possibles bornées.

Preuve. (du théorème 4.2.2) Soient R un système de réécriture qui est soit quasi-cs soit ib et E un langage régulier d'arbres à imbrications de redex possibles bornées.

D'après le lemme 4.2.9, on sait que E peut être décrit par un cs-predicat P linéaire mondique P défini par le programme $\mathcal{P} \cup \mathcal{P}_{pri}$ où toutes les clauses PRI sont dans $\mathcal{P}_{pri}$. D'après le lemme 4.2.11, on sait que $\mathcal{P}_{pri}$ peut être stratifié. Calculer $R^*(E)$ revient à remplacer les redex possibles de E par un de leurs descendants via R. On peut coder cela en remplaçant les clauses PRI $Q(f(x_1, \ldots, x_n)) \leftarrow \mathcal{B}$ de $\mathcal{P}_{pri}$ par la clause $Q(y) \leftarrow P_f(x_1, \ldots, x_n, y), \mathcal{B}$. Appelons $\mathcal{P}'_{pri}$ ce nouvel ensemble de clauses et définissons $\mathcal{P}'^i_{pri} = \{\, Q(y) \leftrightarrow P_f(x_1, \ldots, x_n, y), \mathcal{B} \mid Q(f(x_1, \ldots, x_n)) \leftarrow \mathcal{B} \in \mathcal{P}^i_{pri} \,\}$.

Comme R est soit quasi-cs soit ib, d'après le théorème 4.1.18 on sait qu'il existe un cs-programme $cs(R)$ qui encode $\rightarrow^*_R$. Considérons la procédure suivante :

$\mathcal{P}_0 = \mathcal{P}$, $Done = \emptyset$, $k = 1$
tant que $Done \neq \mathcal{P}'_{pri}$ faire
  Soit $i = \min\{\, j \mid \mathcal{P}'^j_{pri} \setminus Done \neq \emptyset \,\}$
  Soit $C \in \mathcal{P}'^i_{pri} \setminus Done$
  Soit $\mathcal{P}''$ le cs-programme calculé à partir de C et $\mathcal{P}_{k-1} \cup cs(R)$
  $\mathcal{P}_k = Reg(\mathcal{P}'')$
  $k = k + 1$
  $Done = Done \cup \{C\}$
fin tant que

$\mathcal{P}_0$ est un cs-programme linéaire monadique car $\mathcal{P}$ en est un. Donc la première itération est un problème de jointure régulière, c'est à dire qu'une définition de jointure régulière compatible avec $\mathcal{P}$ doit être transformée en un cs-programme. Supposons qu'à l'étape k, C est une définition

$$X \to_\Delta t \quad \text{si } X \to t \in \Delta$$

$$\frac{s \to_\Delta s'}{s \mid t \to_\Delta s' \mid t} \qquad \frac{t \to_\Delta t'}{s \mid t \to_\Delta s \mid t'}$$

$$\frac{s \to_\Delta s'}{s \cdot t \to_\Delta s' \cdot t} \qquad \frac{t \to_\Delta t'}{s \cdot t \to_\Delta s \cdot t'} \quad \text{si } s \text{ est fini}$$

FIG. 4.3 – Sémantique de l'algèbre de processus

de jointure régulière compatible avec $P_k \cup cs(R)$. On sait d'après le lemme 4.2.5 que le dernier argument de chaque prédicat de $cs(R)$ est libre dans $cs(R)$. D'après le théorème 3.3.21 on sait que le dernier argument de chaque prédicat de $\mathcal{P}''$ est libre dans $\mathcal{P}''$, par conséquent $Reg(\mathcal{P}'') = \mathcal{P}_{k+1}$ est un cs-programme linéaire monadique.

Par conséquent, lorsque l'algorithme se termine le programme obtenu $\mathcal{P}_n$ est un cs-programme linéaire monadique qui décrit le langage $R^*(E)$. Donc $R^*(E)$ est un langage régulier d'arbres.

4.3 Exemples d'applications à la vérification

Les résultats présentés dans la section précédente peuvent trouver des applications dans le domaine de la vérification comme l'ont déjà montré de nombreux auteurs [33, 51, 89]. Cela nécessite de les adapter aux besoins spécifiques de la vérification. Dans cette section, nous esquissons deux développements possibles pour appliquer nos résultats au problème d'accessibilité. Ce problème, bien connu en vérification consiste à vérifier si certains états du systèmes sont accessibles à partir d'un ensemble d'états initiaux, en d'autres termes, est-ce que le système peut évoluer vers l'un de ces état. T. Genet et F. Klay, par exemple, utilise cette technique pour montrer des propriété de sûreté du protocol à clé publique de Needham-Shrœder [33]. En vérification, les états peuvent être modelisés par des termes, les transitions du système deviennent alors des règles de réécriture. Le problème d'accessibilité revient donc à se poser la question suivante : peut-on atteindre par réécriture certains termes à partir d'un ensemble de termes initiaux E ? c'est à dire est-ce que certains termes appartiennent à $R^*(E)$? Par exemple, pour montrer une propriété de sûreté du système, il suffit de montrer que l'ensemble des états non-sûrs du système ne sont pas atteignables à partir des états initiaux.

Le premier développement que nous présentons, concerne les algèbres de processus étudiés par D. Lugiez et P. Schnoebelen [67]. Dans leur approche, on utilise trois symboles de fonction : "0" qui exprime la fin d'un processus, "·" qui exprime la composition séquentielle et "|" la composition parallèle. On utilise aussi un ensemble fini de *variables de processus* noté PV. Les *processus* sont des termes de la signature $\mathcal{T}(\{0, \cdot, |\}, PV)$. Le comportement du système est défini par un ensemble fini Δ de *transitions* de la forme $X \to t$ où X est une variable de processus et t un processus. Un terme t est *fini* si $t \in \mathcal{T}(\{0, \cdot, |\})$, c'est à dire, s'il ne contient aucune variable de processus. La sémantique de la relation $\to_\Delta$ est définie dans la figure 4.3.

Pour résoudre le problème d'accessibilité dans ces algèbres de processus avec notre méthode, nous devons coder la clôture réflexive, transitive de la relation $\to_\Delta$ par un système de réécriture. Nous allons considérer les symboles de $\{0, \cdot, |\}$ et de PV comme des symboles de fonction et nous ajoutons deux nouveaux symboles à la signature : $Next*$ et $Finish$. $Next*$ est une fonction définie telle que $Next*(x)$ réduit le processus x en l'un de ses descendants et $Finish(x)$ réduit x en l'un de ses descendants terminés. Le système de réécriture $\mathcal{R}(\Delta)$ de la figure 4.4 calcule la clôture réflexive, transitive de $\to_\Delta$.

On peut observer que le symbole $Finish$ ne peut pas être éliminé si le processus x n'a pas de descendant terminé. Ce système de réécriture est à la fois quasi-cs et ib car il est linéaire droit et les seules fonctions définies sont $Next*$ et $Finish$. Un des principaux résultats de [67] est le suivant : étant donné un ensemble régulier de processus E (c'est à dire un langage régulier d'arbres de la signature $\{0, \cdot, |\} \cup PV$), l'ensemble des descendants de E par la relation $\to_\Delta$

$Next{*}(x) \to x$	$Next{*}(X) \to Next{*}(t)$ où $X \to t \in \Delta$
$Next{*}(x \cdot y) \to Next{*}(x) \cdot y$	$Next{*}(x \mid y) \to Next{*}(x) \mid Next{*}(y)$
$Next{*}(x \cdot y) \to Finish(x) \cdot Next{*}(y)$	
$Finish(0) \to 0$	$Finish(X) \to Finish(t)$ où $X \to t \in \Delta$
$Finish(x \cdot y) \to Finish(x) \cdot Finish(y)$	$Finish(x \mid y) \to Finish(x) \mid Finish(y)$

FIG. 4.4 – Le système de réécriture $\mathcal{R}(\Delta)$.

est un ensemble régulier d'arbres. On peut obtenir le même résultat en considérant l'ensemble $E' = \{\, Next{*}(t) \mid t \in E \,\}$. E' est une ensemble régulier d'arbres qui est en plus à imbrications de redex possibles bornées puisque chaque terme de E' ne contient qu'une seule occurrence de symbole de fonction définie : *Next∗*. Donc d'après le théorème 4.2.2, $\mathcal{R}^*(E')$ est un langage régulier d'arbres.

D. Lugiez et P. Schnoebelen montrent aussi que l'ensemble des prédesseurs par le relation $\to_\Delta$, d'un ensemble régulier de processus et aussi un langage régulier. Pour obtenir ce résultat dans notre cadre, il suffit de remarquer que la transformation du système de réécriture en un cs-programme produit des prédicats dont les deux arguments sont libres.

Cette utilisation de nos résultats aux algèbres de processus est simplement esquissée. Elle doit être développée plus en avant pour étendre les résultats de [67]. Le principal désaventage des algèbres de processus de [67] est leur absence de communication inter-processus. Les symboles de fonction *Next∗* et *Finish* de $\mathcal{R}(\Delta)$ sont d'arité un alors que nos résultats peuvent manipuler des symboles de fonction d'arité quelconque. On peut facilement imaginer que l'on pourrait exprimer certaines formes restreintes de communications entre processus en utilisant des fonctions d'arité supérieure à un.

Le deuxième développement possible concenrne les méthodes de vérification qui se basent sur la réécriture de termes comme par exemple [33, 34, 89]. Elles fonctionnent généralement de la manière suivante. On code le comportement du système à modéliser par un système de réécriture R dans lequel une étape de réécriture représente une transition d'un état du système vers un autre état. Ensuite, on décrit l'ensemble des états non-sûrs du système par un langage régulier d'arbres U. Enfin, on décrit l'ensemble des états initiaux aussi par un langage régulier d'arbres. La méthode de vérification consiste alors à calculer $R^*(E)$ et à vérifier que cet ensemble a une intersection vide avec U. Comme $R^*(E)$ n'est pas un langage régulier en général comme on l'a vu section 4.1, on doit sur-approximer $R^*(E)$ par un langage régulier d'arbres $O(R^*(E))$ comme cela la sûreté du système sera montrer si $O(R^*(E)) \cap U$ est vide. Dans le cas d'une intersection non-vide, il restera à vérifier si cela est dû à une défaillance du système ou simplement un problème lié à l'approximation.

Par conséquent la clé de ces méthodes réside dans la manière dont les approximations sont faites. Comme notre méthode pour calculer $R^*(E)$ est une méthode exacte, il nous faut trouver des moyens de sur-approximer $R^*(E)$ pour les systèmes de réécriture qui ne sont ni quasi-cs ni ib. On peut le faire en transformant le système de réécriture R en un nouveau R', qui appartient à l'une des deux classes que l'on sait traiter et pour lequel $R'^*(E) \subseteq R^*(E)$. L'idée est de transformer chaque règle de réécriture qui n'est pas ib (resp quasi-cs) en une nouvelle qui l'est, en remplaçant certains sous-termes par des variables. Ce type d'approximation est souvent utilisée (voir par exemple [8] dans le contexte de la réécriture ou [78] dans celui de la programmation logique). Notre approximation se base sur le lemme suivant.

Lemme 4.3.1. *Soient* $R_1 = R \cup \{l \to r\}$ *et* $R_2 = R \cup \{l \to r[u \leftarrow x]\}$ *des systèmes de réécriture où* $x \notin Var(r)$*. Si* E *est un langage régulier d'arbres alors* $R_1^*(E) \subseteq R_2^*(E)$*.*

Preuve. Il suffit de remarquer que l'ensemble S_1 des instances closes de r est inclus dans l'ensemble S_2 des instances closes de $r[u \leftarrow x]$. Par conséquent on a $R_1^*(S_1) \subseteq R_2^*(S_2)$.

La transformation de base pour approximer un système de réécriture est donc la suivante.

$$\text{Approximate.} \quad \frac{R \cup \{l \to r\}}{R \cup \{l \to r[u \leftarrow x]\}} \quad \text{où } x \notin Var(r)$$

Cette règle est appliquée dans quatre situations différentes définies par les conditions suivantes :

(unnest) $u \in PRedPos(r)$ et $v < u$ pour $v \in PRedPos(r)$.

(linearise) $r|_u$ est une variable apparaissant plusieurs fois dans r.

(quasi-cs) $r|_u$ est une variable de $PRedVar(r) \cap Var(l)$ dans la profondeur minimale dans l est strictement inférieure à $PRedDepth(r|_u)$.

(ib) $r|_u$ n'est pas une variable, $u = v.w$ pour $v \in PRedPos(r)$, et il existe une règle $l' \to r' \in R$ telle que $r|_v$ et l' s'unifient et $w \notin \Sigma Pos(l')$.

Théorème 4.3.2. *Un système de réécriture R' obtenu à partir d'un système R en utilisant la stratégie suivante*

1. *appliquer la règle* Approximate *avec la condition (unnest) tant que c'est possible,*
2. *appliquer la règle* Approximate *avec la condition (linearise) tant que c'est possible,*
3. *appliquer la règle* Approximate *avec la condition quasi-cs (resp ib) tant que c'est possible,*

est un système de réécriture quasi-cs (resp ib) qui satisfait $R^*(E) \subseteq R'^*(E)$.

Preuve. On peut déjà observer que *Approximate* ne peut pas s'appliquer indéfiniment avec l'une des quatre conditions car chaque application de cette règle supprime une position dans un membre droit sur laquelle elle peut s'appliquer. Donc, pour un membre droit d'un système de réécriture r si la condition (unnest) n'est pas vérifiée alors r ne contient aucun redex possible imbriqué ; si la condition (linearise) n'est pas vérifiée alors r est linéaire ; si la condition (quasi-cs) n'est pas vérifiée alors r statisfait la dernière condition de la définition 4.1.15 ; et si la condition (ib) n'est pas vérifiée alors r satisfait la dernière condition de la définition 4.1.16.

Donc, tout système R' obtenu à partir d'un système de réécriture R en utilisant la stratégie décrite ci-dessus est quasi-cs (resp ib). De plus, d'après le lemme 4.3.1, on a $R^*(E) \subseteq R'^*(E)$.

Etant donnés un système de réécriture R codant le comportement d'un système, un ensemble E de termes initiaux exprimé par un langage régulier d'arbres à imbrications de redex possibles bornées et un ensemble U d'états non sûrs exprimé par un langage régulier d'arbres, on peut utiliser le théorème 4.3.2 pour approximer R par un système R', ensuite on calcule $R'^*(E)$ en se basant sur le théorème 4.2.2, enfin on vérifie la sûreté du système en testant $R'^*(E) \cap U = \emptyset$.

4.4 Data-joignabilité

Dans cette section, nous montrons comment utiliser les langages d'arbres pour résoudre la data-joignabilité. Nous présentons d'abord une méthode générale présentée dans [65], pour résoudre ce problème à l'aide des langages d'arbres, puis nous montrons que grâce aux proriétés des programmes pseudo-réguliers, nous pouvons résoudre les formules pseudo-régulières du 1^{er} ordre, c'est à dire des formules basées sur l'unique prédicat de joingnabilité modulo système de réécriture pseudo-régulier. Notons que dans cette section nous utilisons la notion de redex possible adaptée aux systèmes de réécriture basés sur les constructeurs, c'est à dire celle de la définition 2.3.5.

4.4.1 Résolution des équations de joingnabilité

La méthode de résolution des équations de joignabilité s'inspire de celle décrite dans [65] en l'adaptant à la représentation des langages de n-uplets d'arbres par des programmes logiques.

Tout d'abord nous définissons le type d'équations que nous allons manipuler.

Définition 4.4.1. Soit R un système de réécriture basé sur les constructeurs, $s \downarrow_R^? t$ est appelée une *équation de joignabilité* ou *problème de joignabilité*. $\downarrow_R^?$ est appelé *prédicat de joignabilité*. L'ensemble des data-solutions closes d'une telle équation est noté $SOL(s \downarrow_R^? t)$, et est l'ensemble $\{\sigma$ data-substitution close telles que $s\sigma \downarrow_R t\sigma\}$.

Dans la suite, nous allons utiliser le programme $\mathcal{P}_{Eq}$ de la définition 4.2.3, en le restreignant aux symboles de constructeurs.

Définition 4.4.2. Soient R un système de réécriture basé sur les constructeurs et $s \downarrow_R^? t$ une équation de joignabilité, $\vec{x}$ une énumération des variables de $s \downarrow_R^? t$. On définit $\mathcal{LP}(s \downarrow_R^? t) = P_{s\downarrow_R^? t}(\vec{x}) \leftrightarrow P_{Eq}(s', t'), G_s, G_t$ où $s \rightsquigarrow (s', G_s)$ et $t \rightsquigarrow (t', G_t)$ (voir figure 4.1).

Pour pouvoir démontrer que la clause $\mathcal{LP}(s \downarrow_R^? t)$ nous permet de représenter l'ensemble des solutions de $s \downarrow_R^? t$ il nous faut reprendre le théorème 4.1.11 dans le cadre de la réécriture basée sur les constructeurs et des data-dérivations. Pour cela nous enlevons simplement l'ensemble de clauses $\mathcal{P}_{Id}$ on obtient alors le théorème suivant.

Théorème 4.4.3. *Soient R un système de réécriture basé sur les constructeurs, s un terme et $s \rightsquigarrow \langle s', \mathcal{G}\rangle$. $\mathcal{LP}(R) \models \mathcal{G}\mu$ pour une data-substitution μ, ssi $s \overset{bas}{\rightarrow}^* s'\mu$ par une data-dérivation et $s'\mu$ est un data-terme.*

Ce théorème se démontre par les deux lemmes suivants, on rappelle que d'après le lemme 4.1.7 toute data-dérivation est basique.

Lemme 4.4.4. *Soient R un système de réécriture basé sur les constructeurs, s un terme et $s \rightsquigarrow \langle s', \mathcal{G}\rangle$. Si $\mathcal{LP}(R) \models \mathcal{G}\mu$ pour une data-substitution μ, alors $s \overset{bas}{\rightarrow}^* s'\mu$ par une data-dérivation et $s'\mu$ est un data-terme.*

Preuve. $\mathcal{LP}(R) \models \mathcal{G}\mu$ ssi il existe une preuve dont le but initial est $\mathcal{G}\mu$. On utilise une induction sur la longueur l de la preuve.

$l = 0$ *:* Cela signifie que $s \rightsquigarrow \langle s, \emptyset\rangle$, donc s est un data-terme.

$l > 0$ *:* Soit $A_u\mu$ l'atome choisi et $P_f(t_1, \ldots, t_n, t) \leftarrow \mathcal{B}$ la clause utilisée lors de la première étape de la preuve avec $A_u = P_f(t_1, \ldots, t_n, t)\sigma$. Cette clause appartient à $\mathcal{LP}(R)$; soit $f(s_1, \ldots, s_n) \rightarrow r$ la règle de réécriture qui a produit cette clause. Sans perte de généralité, on peut considérer que l'atome A_u est tel qu'il n'existe pas d'atome $A_v\mu$ dans $\mathcal{G}\mu$ tel que $v > u$. Le prochain but dans la preuve est $(\mathcal{G} \setminus \{A_u\})\mu \cup \mathcal{B}\sigma$. $s[u \leftarrow r\sigma] \rightsquigarrow \langle s'', \mathcal{G} \setminus \{A_u\}\rangle \cup \mathcal{B}\sigma$. Notons que σ est une data-substitution car ni A_u ni $P_f((t_1, \ldots, t_n, t)$ ne contiennent de symboles de fonction définie (voir définition 2.3.5) donc σ n'apporte pas de redex possible. D'autre part, $s'' = s'$ si $u \notin Pos(s')$ sinon $s'' = s'\{x_u \mapsto t\}$ car t est la partie irréductible de r et dans ce cas $x_u\mu = t\sigma$ car x_u est le dernier argument de A_u. $\mathcal{LP}(R) \models (\mathcal{G} \setminus \{A_u\})\mu \cup \mathcal{B}\sigma$ avec une preuve de longueur inférieure à l donc $s[u \leftarrow \sigma r] \overset{bas}{\rightarrow}^* s''\mu \uplus \sigma$ par une data-dérivation. Si $u \notin Pos(s')$ $s' = s''$ donc $s''\mu \uplus \sigma = s'\mu$ sinon $s''\mu \uplus \sigma = (s'\{x_u \mapsto t\})\mu \uplus \sigma = s'\mu[u \leftarrow t\sigma] = s'\mu$ donc $s[u \leftarrow \sigma r] \overset{bas}{\rightarrow}^* s'\mu$ or l'étape $s \overset{bas}{\rightarrow} s[u \leftarrow \sigma r]$ est une data-étape donc $s \overset{bas}{\rightarrow}^* s'\mu$ par une data-dérivation et $s'\mu$ est un data-terme.

Lemme 4.4.5. *Soient R un système de réécriture basé sur les constructeurs, s un terme et $s \rightsquigarrow \langle s', \mathcal{G}\rangle$. Si $s \overset{bas}{\rightarrow}^* s'\mu$ par une data-dérivation où $s'\mu$ est un data-terme alors $\mathcal{LP}(R) \models \mathcal{G}\mu$.*

Preuve. On montre ce lemme par induction sur la longueur l de la data-dérivation.

$l = 0$ *:* Donc $s = s'\mu$ est il est sans redex possible donc $\mathcal{G} = \emptyset$ par conséquent $\mathcal{LP}(R) \models \mathcal{G}\mu$.

$l > 0$ *:* Considérons que $s \overset{bas}{\rightarrow} s[u \leftarrow r\sigma] \overset{bas}{\rightarrow}^* s'\mu$ avec $s|_u = l\sigma$ où $l \rightarrow r \in R$. Posons $\mathcal{LP}(l \rightarrow r) = P_f(s_1, \ldots, s_n, t) \leftarrow \mathcal{B}$ et $s[u \leftarrow r\sigma] \rightsquigarrow \langle s'', G''\rangle$. On a $s[u \leftarrow r\sigma] \overset{bas}{\rightarrow}^* s''\mu''$ avec une data-derivation de longueur $l - 1$ et $s''\mu'' = s'\mu$. Par conséquent soit $u \notin Pos(s')$ et dans ce cas $\mu = \mu''$ soit $u \in Pos(s')$ et dans ce cas $s'' = s'x_u \mapsto t$ car t est la partie irréductible de r et donc $x\mu'' = x\mu$ pour toute variable de s' différente de x_u, et $x_u\mu = t\mu''$. Comme σ est une data-substitution et R un TRS basé sur les constructeurs, le seul symbole de fonction défini de $l\sigma$ et son symbole de tête par conséquent $\mathcal{G}$ ne contient aucun atome A_v tel que $v > u$. Par conséquent $G'' = G \setminus \{A_u\} \cup \mathcal{B}\sigma$. Par l'hypothèse d'induction on sait que $\mathcal{LP}(R) \models \mathcal{G}''\mu''$ donc $\mathcal{LP}(R) \models (G \setminus \{A_u\} \cup \mathcal{B}\sigma)\mu''$ donc $\mathcal{LP}(R) \models (G \setminus \{A_u\})\mu \cup \mathcal{B}\sigma\mu''$ par conséquent $\mathcal{LP}(R) \models (G \setminus \{A_u\})\mu$ d'une part et $\mathcal{LP}(R) \models \mathcal{B}\sigma\mu''$ d'autre part donc $\mathcal{LP}(R) \models P_f(s_1, \ldots, s_n, t)\sigma\mu'' = A_u\mu$ ce qui termine la preuve.

Lemme 4.4.6. *Soient R un système de réécriture basé sur les constructeurs et $s \downarrow_R^? t$ une équation de joignabilité, $\mathcal{LP}(R) \cup \mathcal{LP}(s \downarrow_R^? t) \cup \mathcal{P}_{Eq} \models P_{s\downarrow_R^? t}(s_1, \ldots, s_n)$ ssi $\sigma = \{ x_i \mapsto s_i \mid i \in [1, n] \} \in SOL(s \downarrow_R^? t)$.*

Preuve. Tout d'abord, on peut remarquer que pour tout terme t tel que $t \rightsquigarrow \langle t', \mathcal{G}\rangle$ et toute data-instance close $t\sigma$ de ce terme telle que $t\sigma \rightsquigarrow \langle t'', \mathcal{G}''\rangle$, on a $\mathcal{G} = \mathcal{G}''\sigma$ car les positions de fonction définie sont les mêmes dans les deux termes. De plus $t'' = t\sigma$. Soient σ une data-substitution close telle que $s\sigma$ et $t\sigma$ sont des data-instances closes de s et t respectivement et μ une substitution telle que $\mathcal{LP}(R) \cup \mathcal{P}_{Eq} \models (P_{Eq}(s', t'), \mathcal{G}_s, \mathcal{G}_t)\sigma\mu$ avec $s \rightsquigarrow \langle s', \mathcal{G}_s\rangle$ et $t \rightsquigarrow \langle t', \mathcal{G}_t\rangle$, d'après le théorème 4.4.3, $\mathcal{LP}(R) \models \mathcal{G}_s\sigma\mu$ ssi $s \overset{bas}{\rightarrow}^* s'\sigma\mu$ et $\mathcal{LP}(R) \models \mathcal{G}_t\sigma\mu$ ssi $t \overset{bas}{\rightarrow}^* t'\sigma\mu$ par des data-dérivations et $s'\sigma\mu$ et $t'\sigma\mu$ sont des data-termes. Or $\mathcal{LP}(R) \cup \mathcal{P}_{Eq} \models (P_{Eq}(s', t'))\sigma\mu$ implique que $s'\sigma\mu = t'\sigma\mu$ donc $\mathcal{LP}(R) \cup \mathcal{P}_{Eq} \models (P_{Eq}(s', t'), \mathcal{G}_s, \mathcal{G}_t)\sigma\mu$ ssi $s\sigma$ et $t\sigma$ sont data-joignables donc ssi $\sigma \in SOL(s \downarrow_R^? t)$.

Le résultat du lemme 4.4.6 est trop général pour nos besoins puisque même si le système de réécriture est un système quasi-cs ou ib la clause $\mathcal{LP}(s \downarrow_R^? t)$ a un corps sans propriétés particulières ce qui signifie qu'en général, on ne pourra pas calculer de cs-programme représentant l'ensemble des solutions de $s \downarrow_R^? t$. Nous imposons donc des restrictions pour obtenir une classe de problèmes que l'on peut résoudre avec les cs-programmes. Nous nous intéressons plus particulièrement aux programmes pseudo-réguliers pour lesquels on sait calculer toutes les opérations sur les relations.

Définition 4.4.7. Un système de réécriture basé sur les constructeurs R est dit *pseudo-régulier* si pour toute règle de réécriture $f(t_1, \ldots, t_n) \rightarrow r \in R$ on a :
- aucun des t_i $(1 \leq i \leq n)$ n'est une variable,
- il existe une application π: $Var \mapsto \mathbb{N}^+$, telle que $\pi(x) = u$ implique que toutes les occurrences de x dans chacun des $t_1, \ldots, t_n$ et dans $Irr(r)$ sont à la position u,
- chaque redex possible C à la position u de r est de la forme $f(x_1, \ldots, x_n)$ où $\forall 1 \leq i \leq n$ $x_i \in Var(f(t_1, \ldots, t_n))$ et $\pi(x_i) = u$.

Exemple 4.4.8.

$$R = \{f(s(c(x, y)), s(c(x, z))) \rightarrow p(c(g(x), f(y, y))), f(s(0), s(0)) \rightarrow 0, g(s(x)) \rightarrow s(x)\}$$

est un système de réécriture pseudo-régulier. En effet, la partie irréductible du membre droit de la première règle est $p(c(\square_1, \square_2))$, ce membre droit contient donc deux redex possibles $g(x)$ et $f(y, y)$. Les variables de ces redex possibles apparaissent bien toujours à la même occurrence dans les arguments du membre gauche de la règle. Le seconde règle est trivialement pseudo-régulière puisqu'elle est close. Enfin la dernière règle l'est aussi car son membre droit n'a pas de redex-possible et la variable x apparaît à la même occurrence dans ce membre droit et dans l'argument du membre gauche (c'est à dire $s(x)$). Notons que la définition 4.4.7 n'interdit pas les variables dupliquées à l'intérieur d'un même redex possible.

Le lemme suivant nous permet de connaître la classe de programmes logiques dans laquelle se trouve $\mathcal{LP}(R)$.

Lemme 4.4.9. *Soit R un système de réécriture pseudo-régulier, $\mathcal{LP}(R)$ est un programme logique NG-SPR.*

Preuve. Chaque règle du système de réécriture pseudo-régulier est de la forme

$$f(t_1, \ldots, t_n) \rightarrow C[f_1(\vec{x}_1), \ldots, f_k(\vec{x}_k)]$$

où C est la partie irréductible du membre droit de la règle, et $f, f_1, \ldots, f_k$ sont des symboles de fonction définie. C et chacun des t_i sont des data-termes. Par conséquent la clause produite pour une telle règle est

$$P_f(t_1, \ldots, t_n, C[x_{u_1} \ldots, x_{u_k}]) \leftarrow P_{f_1}(\vec{x}_1, x_{u_1}), \ldots, P_{f_k}(\vec{x}_k, x_{u_k})$$

Comme toutes les variables de $f_1(\vec{x}_1), \ldots, f_k(\vec{x}_k)$ sont des variables de $f(t_1, \ldots, t_n)$, la clause ne contient pas de variable existentielle. En étendant l'application π de la règle pseudo-régulière $f(t_1, \ldots, t_n) \rightarrow C[f_1(\vec{x}_1), \ldots, f_k(\vec{x}_k)]$ à $\pi' = \pi \cup_{1 \leq i \leq k} \{x_{u_i} \mapsto u_i\}$ on obtient le fait que la clause est NG-SPR.

Maintenant il nous faut aussi donner des conditions suffisantes sur les équations de joignabilité afin d'obtenir des clauses qui correspondent à une classe de problèmes que l'on sait résoudre. Plus précisément, on souhaite obtenir à partir de l'équation de jointure, une définition de jointure généralisée (définition 3.5.4), c'est à dire une clause dont le corps ne contient aucun symbole de fonction.

Définition 4.4.10. Une *équation de joignabilité pseudo-régulière* est une équation $s \downarrow_R^? t$ telle que s et t sont des termes construits sur $T(\mathcal{D}, Var)$, c'est à dire des termes ne contenant aucun constructeur.

La définition précédente semble interdire des équations du type $f(x) \downarrow_R^? 0$ où f est une fonction définie et 0 un constructeur. En fait on peut exprimer ce type d'équation en rajoutant au système de réécriture une nouvelle règle $zero \to 0$ et de remplacer l'équation par $f(x) \downarrow_R^? zero$. Plus généralement, on peut utiliser la même méthode pour tous les sous-termes de l'équation qui sont des data-termes sans variables communes avec le reste de l'équation.

Lemme 4.4.11. *Soit $s \downarrow_R^? t$ une équation de joignabilité pseudo-régulière. $\mathcal{LP}(s \downarrow_R^? t)$ est une définition de jointure généralisée.*

Preuve. Pour montrer ce lemme, il suffit de remarquer que pour un terme s sans constructeur $Irr(s) = \square$ est que chacun des redex possibles de ce terme est de la forme $f(x_1, \ldots, x_n)$ où x_i ($1 \leq i \leq n$) est soit une variable soit une variable de contexte. Par conséquent $s \rightsquigarrow \langle x_\varepsilon, \mathcal{G}\rangle$ et $\mathcal{G}$ ne contient aucun symbole de fonction.

On peut donc maintenant montrer le résultat suivant.

Théorème 4.4.12. *Soient R un système de réécriture pseudo-régulier et $s \downarrow_R^? t$ une équation de joignabilité pseudo-régulière. $SOL(s \downarrow_R^? t)$ peut être représentée par un programme pseudo-régulier.*

Preuve. D'après le lemme 4.4.9 $\mathcal{LP}(R)$ est une programme NG-SPR, les clauses du programme définissant $\mathcal{P}_{Eq}$ sont des clauses pseudo-régulières. D'après le lemme 4.4.11 $\mathcal{LP}(s \downarrow_R^? t)$ est une définition de jointure généralisée. D'après le lemme 4.4.6, on a

$$\mathcal{LP}(R) \cup \mathcal{LP}(s \downarrow_R^? t) \cup \mathcal{P}_{Eq} \models P_{s\downarrow_R^? t}(s_1, \ldots, s_n)$$

ssi $\sigma = \{\, x_i \mapsto s_i \mid i \in [1, n] \,\} \in SOL(s \downarrow_R^? t)$. Or d'après le lemme 3.5.19 et le corrolaire 3.5.12 le programme $\mathcal{LP}(R) \cup \mathcal{P}_{Eq}$ peut être transformée en un programme pseudo-regulier équivalent noté $\mathcal{P}$, donc d'après le théorème 3.5.5, $\mathcal{P}$ et $P_{s\downarrow_R^? t}(s_1, \ldots, s_n)$ peuvent-être transformés en un programme pseudo-régulier.

4.4.2 Résolution des formules pseudo-régulières

Dans cette section, nous montrons comment combiner les résultats sur les programmes pseudo-réguliers décrits section 3.5 et ceux de la section précédente pour trouver les solutions des formules pseudo-régulières. Voici tout d'abord, la classe des formules que nous allons considérer.

Définition 4.4.13. Soit R une système de réécriture pseudo-régulier, les *R-formules pseudo-régulières* sont définies par la grammaire suivante :

$$e ::= s \downarrow_R^? t | \neg e | e \vee e | e \wedge e | \exists x e | \forall x e$$

où $s \downarrow_R^? t$ est une équation de joignabilité pseudo-régulière.

Le R sera ommis lorsqu'il pourra être déduit du contexte. L'ensemble des solutions d'une telle formule est défini comme suit :

- $SOL(s \downarrow_R^? t)$ est l'ensemble des data-solutions de $s \downarrow_R^? t$
- $SOL(\neg e) = \{\, \sigma \mid \sigma \notin SOL(e) \,\}$

- $SOL(e_1 \wedge e_2) = \{ \sigma \mid Dom(\sigma) = Var(e_1 \wedge e2), \sigma|_{Var(e_1)} \in SOL(e_1), \sigma|_{Var(e_2)} \in SOL(e_2) \}$
- $SOL(e_1 \vee e_2) = \{ \sigma \mid Dom(\sigma) = Var(e_1 \vee e_2), \sigma|_{Var(e_1)} \in SOL(e_1)$ ou $\sigma|_{Var(e_2)} \in SOL(e_2) \}$
- $SOL(\exists x e) = \{ \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \exists \sigma' \in SOL(e), \sigma = \sigma'|_{Var(e)\setminus\{x\}} \}$
- $SOL(\forall x e) = \{ \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \forall \sigma' \in SOL(e), \sigma = \sigma'|_{Var(e)\setminus\{x\}} \}$

Malgré les restrictions imposées sur les équations de joignabilité, beaucoup de propriétés intéressantes peuvent être exprimées par des formules pseudo-régulières dont voici quelques exemples.

- La confluence $\forall x, y \neg(f(x, y) \downarrow_R^? z \wedge f(x, y) \downarrow_R^? z') \vee z \downarrow_R^? z'$.
- La commutativité $\forall x, y f(x, y) \downarrow_R^? f(y, x)$.
- L'associativité $\forall x, y, z f(x, f(y, z)) \downarrow_R^? f(f(x, y), z)$.
- L'idempotence $\forall x g(g(x)) \downarrow_R^? x$.
- La recherche des point fixes d'une fonction $g(x) \downarrow_R^? x$.
- La recherche des éléments neutres $\forall x f(x, y) \downarrow_R^? x$.

La résolution des formules pseudo-régulière est donnée par l'algorithme suivant :

Algorithme 4.4.14. *Soientt R un système de réécriture pseudo-régulier et e une R-formule pseudo-régulière.*

1. *Résoudre chaque formule élémentaire $s \downarrow_R^? t$ en calculant $\mathcal{P}_{s\downarrow_R^? t}$*
2. *Si $\mathcal{P}_e$ définit $SOL(e)$ alors $\overline{\mathcal{P}_e} = \mathcal{P}_{\neg e}$ définit $SOL(\neg e)$.*
3. *Si $\mathcal{P}_{e_1}$ et $\mathcal{P}_{e_2}$ définissent respectivement $SOL(e_1)$ et $SOL(e_2)$ alors tout programme $\mathcal{P}_{e_1 \wedge e_2}$ équivalent à $\mathcal{P}_{e_1} \cup \mathcal{P}_{e_2} \cup \{P_{e_1\wedge e_2, \vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1), P_{e_2,\vec{x}_2}(\vec{x}_2)\}$ où $\vec{x} = Var(e_1 \wedge e_2)$ définit $SOL(e_1 \wedge e_2)$*
4. *Si $\mathcal{P}_{e_1}$ et $\mathcal{P}_{e_2}$ définissent respectivement $SOL(e_1)$ et $SOL(e_2)$ alors tout programme $\mathcal{P}_{e_1 \vee e_2}$ équivalent à $\mathcal{P}_{e_1} \cup \mathcal{P}_{e_2} \cup \{P_{e_1\vee e_2, \vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1), P_{e_1\vee e_2,\vec{x}}(\vec{x}) \leftarrow P_{e_2,\vec{x}_2}(\vec{x}_2)\}$ où $\vec{x} = Var(e_1 \vee e_2)$, définit $SOL(e_1 \vee e_2)$.*
5. *Si $\mathcal{P}_e$ définit $SOL(e)$ alors tout programme $\mathcal{P}_{\exists x e}$ équivalent à $\mathcal{P}_e \cup \{P_{\exists x e, \vec{y}}(\vec{y}) \leftarrow P_{e,\vec{x}'}(\vec{x}')\}$ définit $SOL(\exists x e)$ si $\vec{y}$ est la projection de $\vec{x}'$ sur tous ses arguments sauf celui définissant la variable x.*

Pour résoudre les formules du type $\forall x e$, on utilise l'équivalence bien connu $\forall x e \equiv \neg\exists\neg e$.

Théorème 4.4.15. *Soient R un système de réécriture pseudo-régulier et e une R-formule pseudo-régulière et $\mathcal{P}_e$ le programme résultant de l'application de l'algorithme 4.4.14. $\mathcal{P}_e \models P_{e,\vec{x}}(\vec{t})$ ssi $\{ \vec{x}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq length(\vec{x}) \} \in SOL(e)$.*

Preuve. Cette preuve se fait par induction sur la structure de la formule. D'après le théorème 4.4.12, on sait que pour chaque formule élémentaire $s \downarrow_R^? t$, $\mathcal{P}_{s\downarrow_R^? t} \models P_{s\downarrow_R^? t, \vec{x}}(\vec{t})$ ssi $\{ \vec{x}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq length(\vec{x}) \} \in SOL(s \downarrow_R^? t)$. Soient e_1 et e_2 deux formules telles que $\mathcal{P}_{e_1}$ et $\mathcal{P}_{e_2}$ sont les programmes obtenus pour représenter les solutions de ces deux formules.

- D'après le théorème 3.5.17, on sait que $\mathcal{P}_{\neg e_1}$ définit les solution de $SOL(\neg e_1)$ et est un cs-programme régulier.
- $P_{e_1\wedge e_2,\vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1), P_{e_2,\vec{x}_2}(\vec{x}_2)$ est une définition de jointure généralisée car $P_{e_1,\vec{x}_1}$ et $P_{e_2,\vec{x}_2}$ sont définis par des programmes pseudo-réguliers, par conséquent, d'après le théorème 3.5.5 $\mathcal{P}_{e_1\wedge e_2}$ est une programme pseudo-régulier qui définit les solutions de $e_1 \wedge e_2$.
- $P_{e_1\vee e_2,\vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1)$ et $P_{e_1\vee e_2,\vec{x}}(\vec{x}) \leftarrow P_{e_2,\vec{x}_2}(\vec{x}_2)$ sont des définitions de jointures généralisées car $P_{e_1,\vec{x}_1}$ et $P_{e_2,\vec{x}_2}$ sont définis par des programme pseudo-réguliers, par conséquent d'après le théorème 3.5.5 $\mathcal{P}_{e_1\vee e_2}$ est une programme pseudo-régulier définissant les solution de $e_1 \vee e_2$.
- $P_{\exists x e_1,\vec{y}}(\vec{y}) \leftarrow P_{e_1,\vec{x}'}(\vec{x}')$ est une définition de jointure généralisée car $P_{e_1,\vec{x}'}(\vec{x}')$ est défini par un programme pseudo-régulier, donc d'après le théorème 3.5.5 $P_{\exists x e_1,\vec{y}}$ est défini par un programme pseudo-régulier. De plus, $P_{\exists x e_1,\vec{y}}(\vec{y})$ définit les solutions de $\exists x e_1$ car $P_{e_1,\vec{x}'}(\vec{x}')$ définit $SOL(e_1)$ et par conséquent $P_{\exists x e_1,\vec{y}}(\vec{y})$ définit $SOL(e_1)|_{\{x\}}$.

4.5 Travaux liés

Nos résultats sur la préservation de la régularité, nous ont permis de définir les systèmes de réécriture quasi-cs et à base d'instances. Pierre Réty [80] a montré un résultat de préservation de la régularité pour les systèmes de réécriture basés sur les constructeurs. La classe étudiée dans [80] se trouve dans l'intersection des systèmes de réécriture basés sur les constructeurs et des systèmes à base d'instances. Il faut de plus que les membres droits des règles de réécriture ne contiennent pas de symboles de fonction définie imbriqués. Le langage d'entrée est lui restreint à des data-instances closes de termes linéaires ce qui signifie que notre résultat généralise celui de Réty. En effet, si on considère le terme $f(s(g(x))$ où s et 0 sont des constructeurs, ses data-instances closes ne contiennent que deux redex possibles et donc entrent dans notre cadre. Par contre, on peut aussi calculer l'ensemble des descendants du langage régulier $s^*(f(s^*(g(x))))$ où f et g peuvent apparaître à n'importe quelle profondeur dans les termes du langage. Un autre exemple qui montre que nous étendons le résultat de Réty, est le suivant $\{f(s(x)) \rightarrow f(f(p(x)))\}$. Ce système ne satisfait pas la condition sur les symboles de fonction définie imbriqués alors qu'il est à base d'instances car $f(f(p(x)))$ ne contient pas de redex possibles imbriqués (au sens de la définition 2.3.4). Par contre, les systèmes de réécriture quasi-cs ne peuvent pas se comparer avec la classe étudiée dans [80], c'est à dire qu'aucune ne subsume l'autre.

La classe des systèmes de réécriture linéaire droit et à chemin de superposition fini de [88] admet des redex possibles imbriqués ce qui est interdit dans notre contexte, mais elle n'inclus pas des règles comme $f(s(x)) \rightarrow f(x)$. Ce genre de règles ne pose pas de problèmes dans notre cadre puisqu'elles sont linéaires droites, et donc la réécriture basique coïncide avec la réécriture générale. D'autres résultats de préservation de la régularité, parmi lesquels [41, 73, 86], ne sont pas comparables aux nôtres. Ils utilisent souvent des techniques propres à la réécriture, il faudrait étudier dans quelle mesure ces résultats pourraient être démontrés dans notre cadre avec pour but de les étendre.

Mayr [70] a montré que le problème d'accessibilité pour les systèmes de réécriture dits de processus est décidable en utilisant pour cela un algorithme ad-hoc. Le problème que se pose Mayr est le suivant : étant donnés deux termes, est-ce qu'un des deux termes peut être obtenu à partir de l'autre en utilisant les transitions du système de réécriture de processus ? De récents travaux [12] ont étudié le problème des ensembles de descendants dans les systèmes de réécriture de processus utilisant des techniques d'approximation à base d'automates. Ces problèmes peuvent être résolus lorsque le système de réécriture préserve la régularité. Cependant nos résultats ne peuvent pas s'appliquer directement dans ce contexte puisque la réécriture de processus se place dans le cadre de la réécriture modulo AC (associativité, commutativité) ce que nous n'avons pas encore étudié. Dans [81], Réty et Vuotto ont montré des résultats de préservation de la régularité lorsqu'on applique les règles de réécriture avec une certaine stratégie. Pour montrer des résultats similaires dans notre cadre il faudrait modifier les règles de la figure 4.1 afin de prendre en compte la stratégie de réécriture utilisée.

En ce qui concerne la résolution des formules pseudo-régulières du premier ordre, on peut rapprocher ce travail à ceux effectués dans le cadre des formules de la théorie de la réécriture en une étape, introduite par [14] et qui a été démontrée indécidable par Treinen [91]. Des sous-classes de cette théorie ont alors été étudiées (par exemple [69, 74]) en restreignant la classe des formules autorisées et la classe de systèmes de réécriture utilisés. La plupart de ces résultats s'appuient sur des techniques à base d'automates d'arbres. Etant donné qu'il bien connu que l'unifiabilité et la joingnabilité de deux termes sont indécidables sans mettre de très fortes restrictions sur le système de réécriture ou sur l'équation à résoudre (voir par exemple [26, 55]), nous avons pris le parti de restreindre tout de suite la classe de systèmes de réécriture que nous allions traiter. Dans le domaine de l'unification équationnelle (qui est équivalent à la joingnabilité lorsque le système de réécriture est confluent) de nombreux travaux ont recherché des résultats positifs pour la décidabilité de la satisfiabilité de l'unification. Certains résultats consistent à restreindre le système de réécriture [22, 68, 76] afin d'obtenir un problème finitaire (c'est à dire qui assure que l'ensemble minimal des solutions est fini). D'autres résultats positifs ont été montrés dans le cadre des théories infinitaires (des théories où l'ensemble des solutions est potentiellement infini), de nombreux résultats de décidabilité de l'existence de solutions ont été montré, certain en utilisant l'accessibilité comme par exemple [80, 88]) mais très peu offrent la possibilité de représenter de manière finie l'ensemble des solutions [55, 65] et encore moins vont au delà de la résolution d'une seule équation. D'après les résultats obtenus dans [55], nous devrions pouvoir d'étendre le théorème 3.3.16 sur les jointures régulières afin de faire des jointures sur un composant de deux cs-programmes si l'argument de jointure est libre dans l'un des deux programmes. Avec un tel résultat nous pourrions représenter l'ensemble des solutions des équations de joignabilité pour les systèmes de réécriture quasi-cs et ib.

Chapitre 5

Conclusion et perspectives

Dans ce mémoire, nous avons présenté des outils basés sur la programmation logique pour manipuler les langages de n-uplets d'arbres et nous avons montré comment utiliser ces outils pour résoudre un certain nombre de problèmes liés à la réécriture et à la vérification. Ces travaux posent de nouvelles questions et ouvrent à certain nombre de perspectives de recherches intéressantes.

Jusqu'à présent, nous avons donné des conditions suffisantes pour que la procédure de transformation de programmes logiques termine mais nous n'avons pas étudié avec précision la complexité de la procédure. Il serait très intéressant en effet de comparer la complexité de notre technique par rapport à celle des algorithmes classiques sur les automates d'arbres pour le cas des programmes pseudo-réguliers par exemple. Cela nous permettrait de savoir dans quelle mesure le gain en facilité d'expression apportée par la programmation logique altère les performances attendues des algorithmes de traitement de ces langages.

D'autre part certaines sous-classes des langages synchronisés n'ont pas encore été étudiés sous l'angle de la programmation logique. On peut citer les relations faiblement régulières de [59] qui définissent une sur-classe stricte des relations régulières, qui sont closes par intersection (contrairement au langages synchronisés) et qui gardent un test du vide décidable. Il serait donc intéressant de savoir s'il existe une stratégie pour l'application des règles de notre transformation qui permette de terminer dans ce cas. Une autre classe de langages intéressante est celle des ground tree transducers [23]. En effet cette sous-classe des relations régulières a la particularité d'être close par clôture transitive ce qui pourrait nous permettre de connaître des conditions suffisantes pour qu'un programme contenant des clauses du type $R*(x,y) \leftarrow R(x,z)R*(z,y)$ ait un équivalent dans la classe des cs-programmes (ou une de ses sous-classes). On pourrait peut-être ainsi étendre les résultats obtenus dans [23] ou ceux sur la récriture préfixe sur les chaînes [50].

D'autres part il existe des classes de langages d'arbres qui n'ont pas été étudiées dans ce document comme par exemple [25] qui définit des grammaires d'hypergraphes dont l'expressivité va au-delà des langages synchronisés mais qui gardent des tests du vide et d'appartenance décidables et qui sont closes par jointures régulières. Le formalisme pour décrire ces grammaires est assez complexe et il serait donc intéressant d'étudier la classe de programmes logiques qui leur correspond. Cette classe est sans aucun doute possible une sur-classe des cs-programmes et par conséquent il faudrait modifier les algorithmes de transformation de programmes pour effectuer les opérations sur ces langages.

Un certain nombre d'auteurs [2, 17, 20, 43, 84] ont proposé des formalismes à base de schématisation de termes pour représenter des ensembles de termes. Ces formalismes se basent sur des répétitions motifs, contrôlées par des compteurs, dans les termes. Les classes étudiées ont de bonnes propriétés du point de vue des opérations ensemblistes et leur pouvoir d'expression est très intéressant. Malheureusement ce type de langages n'inclut pas les langages réguliers et s'insère difficilement dans notre cadre du fait de la possibilité de dupliquer de manière non bornée les variables. Il serait cependant intéressant d'essayer d'améliorer l'expressivité des cs-programmes en ajoutant des contraintes numériques dans les clauses afin de pouvoir exprimer des comptages à l'intérieur du langage. Le formalisme de la programmation logique avec contraintes [47] permet d'envisager une extension assez naturelle des cs-programmes dans cette voie.

Dans ce mémoire, nous avons principalement étudié des méthodes exactes pour résoudre les différents problèmes que nous nous sommes posés sur les langages et leurs applications. Ces méthodes exactes nécessitent des restrictions syntaxiques sur les objets que nous manipulons afin de ne pas se heuter à des cas indécidables de nos problèmes. Comme nous l'avons vu dans le chapitre précédent, une alternative est d'utiliser une approximation décidable du problème afin de tendre vers la solution ou de la trouver dans certains cas. Nous avons montré dans le chapitre précédent une méthode permettant de faire des approximations a priori du problème étudié. Dans le domaine de la préservation de la régularité, des travaux ont été menés [34, 35] pour effectuer l'approximation en cours de calcul ce qui a l'avantage de pouvoir mieux maîtriser cette approximation et de la raffiner en fonction des besoins. Dans notre cadre, on pourrait imaginer d'introduire ce genre de techniques dans le processus de transformation de programmes logiques, par exemple en remplaçant les occurrences dupliquées de certaines variables par de nouvelles variables afin d'éviter de faire grandir de manière non bornée les sous-ensembles d'atomes sans variables communes et donc de garantir la terminaison du processus.

Dans le chapitre 3, nous avons décrit des classes de programmes logiques qui peuvent se transformer en programmes pseudo-réguliers mais nous avons aussi remarqué que les classes de programmes définies dans [75] qui ont un équivalent régulier, ne se transforment pas en cs-programme régulier par notre technique. On peut aussi remarquer que les programmes pseudo-réguliers n'autorisent pas les clauses de la forme $H(\vec{x}) \leftarrow \mathcal{B}$ où $\mathcal{B}$ est linéaire et sans symboles de fonction (des transitions vides en quelques sortes). Il serait donc intéressant d'étudier comment traiter ces deux cas dans notre cadre. Les programmes pseudo-réguliers servent à résoudre les formules pseudo-régulières du premier ordre, mais on peut se demander dans quelle mesure on pourrait introduire des variables du second ordre dans ces formules. En effet, il serait intéressant d'être capable de résoudre des équations du type $\forall x f(X(x)) \downarrow_R^? x$ car si on est en plus capable de transformer un programme logique en un système de réécriture, nous aurions un mécanisme de construction automatique de fonctions à partir de leurs spécifications. Pour l'exemple ci-dessus, cela signifirait construire automatiquement la fonction inverse de f.

Dans le chapitre précédent, nous avons esquissé des pistes pour l'application de nos outils dans le domaine de la vérification en utilisant des modélisations à base de réécriture. Pour mettre en œuvre réellement ces techniques, il reste encore du travail au niveau de l'implantation de nos algorithmes. Il faudrait notamment améliorer et optimiser et compléter l'application développée en swi-Prolog par G. Salzer[1]. Il serait sans doutes intéressant d'effectuer une implantation avec des outils à base de logique de réécriture comme MAUDE [19] ou ELAN [11, 48], ce qui nous permettrait d'implanter plus facilement les différentes stratégies d'application des règles de transformation. Il faudrait aussi implanter les procédures de transcription des systèmes de réécriture en programmes logiques ainsi que les mécanismes de résolution des problèmes que nous avons évoqués dans le chapitre précédent, c'est à dire le calcul des ensembles de descendants et la résolution des formules pseudo-régulières. Ce travail a été commencé par P. Pillot. Au niveau théorique, il faudrait intégrer des techniques d'approximations citées ci-dessus afin de pouvoir traiter des problèmes généraux. Il serait aussi intéressant d'arriver à représenter les solutions d'une équation de joingnabilité dans le cadre des systèmes de réécriture quasi-cs et ib. En effet grâce à un tel résultat, dans le cadre d'un problème de sûreté, on serait en mesure d'indiquer quels sont les termes initiaux qui provoquent le problème de sûreté et donc en cas d'approximation, on pourrait essayer de savoir si le problème détecté provient d'un réel défaut du système ou si c'est l'approximation qui était trop grossière. Un autre point à étudier serait la possibilité de travailler dans le cadre de la réécriture modulo l'associativité, commutativité par exemple car dans de nombreux cas [33, 46, 51] ces propriétés sont utiles en vérification. Un objectif possible, pourrait être de construire un outil de vérification de protocoles cryptographiques que l'on pourrait intégrer dans des applications comme AVISPA [4].

Les liens entre nos travaux et les contraintes ensemblistes demandent à être approfondis. En effet nos procédures de calcul et de décision sur les ensembles devraient pouvoir apporter de nouveaux résultats dans ce domaine. De plus, certaines techniques de vérification, ([15] par exemple),

[1]Disponible à l'URL `www.logic.at/css`

ou d'analyse de programmes ([75] par exemple) utilisent les contraintes ensemblistes comme formalisme.

Dans le cadre de collaborations au sein de l'équipe Vérification, Parallèlisme et Sécurité, j'ai eu l'occasion de participer au projet RNTL GeoBench au sein duquel j'ai plus particulièrement travaillé à la conception d'un environnement de programmation appelé FlowVr [1]. FlowVr permet de développer et de déployer des applications sur une grappe d'ordinateurs. Il permet aussi de coupler des codes parallèles éthérogènes grâce à sa modularité et à ses interfaces de communication. La structure des applications FlowVr sous forme de modules déployables se prête bien à une modélisation en termes d'arbres finis. Une des problématiques du calcul sur grappe de PC est la prévision de performance, c'est à dire connaître une estimation de la rapidité de l'application en fonction des caractéristiques de la machine cible et éventuellement la possibilité d'aider le programmeur à déployer son application pour obtenir des performances optimales. Il serait donc intéressant d'étudier dans quelle mesure les outils que nous avons présentés peuvent aider à résoudre ce type de problèmes.

Enfin, l'exploitation de documents hypertextes comme les documents XML par exemple, utilise beaucoup de techniques à base d'automates d'arbres. Dans ce cadre les arbres sont le plus souvent différents de ceux que nous avons présentés dans ce mémoire car ils se construisent sur une signature potentiellement infinie et avec des symboles sans arité. Dans [13], le lien entre les automates d'arbres à arité non fixée et les automates classiques est étudié, il serait donc intéressant d'étudier la possibilité de définir des langages de n-uplets sur ce type d'arbres et de déterminer si les résultats que nous avons obtenus peuvent se transcrire dans ce cadre. Il pourrait être utile par exemple dans le cadre des bases de données XML pour évaluer des requêtes ou vérifier des contraintes d'intégrité.

Bibliographie

[1] J. Allard, V. Gouranton, L. Lecointre, E. Melin S. Limet, B. Raffin, and S. Robert. "flowvr : a middleware for large scale virtual reality applications". In *"proc. of the international conference EUROPAR 2004*, volume 3149 of *LNCS*, pages 497–505. Springer Verlag, 2004.

[2] Ali Amaniss, Miki Hermann, and Denis Lugiez. Set operations for recurrent term schematizations. In *TAPSOFT'97 : Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 333–344. Springer, 1997.

[3] S. Antoy, M. Hanus, B. Massey, and F. Steiner. An implementation of narrowing strategies. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'01)*, pages 207–217. ACM Press, 2001.

[4] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, United Kingdom, 1998.

[6] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[7] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Eighth Annual IEEE Symposium on Logic in Computer Science, Montreal (Canada)*, pages 75–83. IEEE Computer Society Press, 1993.

[8] D. Bert, R. Echahed, and B. Østvold. Abstract Rewriting. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *proceedings 3rd Internationnal Workshop on Static Analysis WSA'93*, volume 724 of *LNCS*, pages 178–192. Springer, 1993.

[9] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In A. Finkel and M. Jantzen, editors, *Ninth Anual Symbosium on Theoretical Aspects of Computer Science*, volume 577 of *LNCS*, pages 161–171. Springer Verlag, 1992.

[10] Bruno Bogaert, Franck Seynhaeve, and Sophie Tison. The recognizability problem for tree automata with comparisons between brothers. In *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Proceedings*, volume 1578 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 1999.

[11] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of elan. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[12] Ahmed Bouajjani and Tayssir Touili. On computing reachability sets of process rewrite systems. In *Term Rewriting and Applications, 16th International Conference, RTA 2005*, volume 3467 of *Lecture Notes in Computer Science*, pages 484–499. Springer, 2005.

[13] Julien Carme, Joachim Niehren, and Marc Tommasi. Querying unranked trees with stepwise tree automata. In *Rewriting Techniques and Applications, 15th International Conference RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 2004.

[14] Anne-Cécile Caron, Jean-Luc Coquidé, and Max Dauchet. Encompassment properties and automata with constraints. In *5th International Conference on Rewriting Techniques and Applications*, volume 960 of *LNCS*, pages 328–342. Springer Verlag, 1993.

[15] W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In *Proc. TACAS '98*, volume 1384 of *LNCS*, pages 358–395. Springer, 1998.

[16] Witold Charatonik and Andreas Podelski. Set constraints with intersection. *Information and Computation*, 179(2) :151–385, 2002.

[17] H. Chen and J. Hsiang. Recurrent domains : Their unification and application to logic programming. *Information and Computation*, 122 :45–69, 1995.

[18] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[19] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude : Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[20] H. Comon. On unification of terms with integer exponents. *Mathematical Sytems Theory*, 28(1) :67–88, 1995.

[21] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications (TATA)*. `http://www.grappa.univ-lille3.fr/tata`, 1997.

[22] H. Comon, M. Haberstrau, and J.-P. Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Information and Computation*, 111(1) :154–191, 1994.

[23] Max Dauchet, Sophie Tison, Thierry Heuillard, and Pierre Lescanne. Decidability of the confluence of ground term rewriting systems. In *Proceedings of the Symposium on Logic in Computer Science*, pages 353–359. IEEE Computer Society, 1987.

[24] Philippe Devienne, Jean-Marc Talbot, and Sophie Tison. Co-definite set constraints with membership expressions. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 25–39. MIT Press, 1998.

[25] Joost Engelfriet and Linda Heyker. Context-free hypergraph grammars have the same term-generating power as attribute grammars. *Acta Inf.*, 29(2) :161–210, 1992.

[26] Heinz Faßbender and Sebastian Maneth. A strict border for the decidability of e-unification for recursive functions. *Journal of Functional and Logic Programming*, 1998(4), 1998.

[27] C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 25, pages 1791–1849. Elsevier Science, 2001.

[28] C. G. Fermüller and G. Salzer. Ordered paramodulation and resolution as decision procedure. In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LPAR'93)*, LNCS 698 (LNAI), pages 122–133. Springer Verlag, 1993.

[29] C.G. Fermüller and A. Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 2(6) :173–203, 1996.

[30] T. W. Frühwirth, E. Y. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science*, pages 300–309, 1991.

[31] J. P. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *4th International Symposium, PADL 2002*, volume 2257 of *LNCS*, pages 243–261. Springer, 2002.

[32] John P. Gallagher and Julio C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher-Order and Symbolic Computation*, 14(2-3) :143–172, 2001.

[33] T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *Proceedings 17th International Conference on Automated Deduction*, volume 1831 of *LNAI*, pages 151–165. Springer Verlag, 2000.

[34] T. Genet and V. Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. In *Proc. of the 8th conference LPAR*, volume 2250 of *LNCS*, pages 695–706. Springer, 2001.

[35] Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Rewriting Techniques and Applications, 9th International Conference, RTA-98*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

[36] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proceedings of the* 34^{th} *Symp. on Foundations of Computer Science*, pages 372–380, 1993.

[37] R. Gilleron, S. Tison, and M. Tommasi. Set constraints and automata. *Information and Computation*, 1(149) :1–41, 1999.

[38] J. Goubault-Larrecq. Higher-order positive set constraints. In *Proc. 16th Int. Workshop Computer Science Logic (CSL'2002), Edinburgh, Scotland*, volume 2471 of *LNCS*, pages 473–489. Springer, 2002.

[39] J. Goubault-Larrecq and K. N. Verma. Alternating two-way AC-tree automata. Research Report LSV-02-11, Lab. Specification and Verification, ENS de Cachan, Cachan, France, September 2002. 21 pages.

[40] V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. In *Proceedings of FoSSaCs*, volume 2030 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2001.

[41] P. Gyenizse and S. Vágvölgyi. Linear generalized semi-monadic rewrite systems effectively preserve recognizability. *Theoretical Computer Science*, 194 :87–122, 1998.

[42] N.C. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proc. of the 17th ACM Symp. on Principles of Programming Languages*, pages 197–209. ACM press, 1990.

[43] M. Hermann and R. Galbavý. Unification of infinite sets of terms schematized by primal grammars. *Theoretical Computer Science*, 176(1-2) :111–158, 1997.

[44] Miki Hermann and Gernot Salzer. On the word, subsumption, and complement problem for recurrent term schematizations. In *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98*, volume 1450 of *Lecture Notes in Computer Science*, pages 257–266. Springer, 1998.

[45] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *LNCS*, pages 318–334. Springer, 1980.

[46] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000*, volume 1955 of *Lecture Notes in Computer Science*, pages 131–160. Springer, 2000.

[47] Joxan Jaffar and Michael J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 19/20 :503–581, 1994.

[48] Claude Kirchner and Hélène Kirchner. Rule-based programming and proving : The elan experience outcomes. In *9th Asian Computing Science Conference*, volume 3321 of *Lecture Notes in Computer Science*, pages 363–379. Springer, 2004.

[49] Herbert Kuchen, Rita Loogen, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. The functional logic language BABEL and its implementation on a graph machine. *New Generation Computing*, 14(4) :391–427, 1996.

[50] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *Computer Aided Verification, 14th International Conference, CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2002.

[51] Pascal Lafourcade, Denis Lugiez, and Ralf Treinen. Intruder deduction for AC-like equational theories with homomorphisms. In Jürgen Giesl, editor, *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *Lecture Notes in Computer Science*, pages 308–322. Springer, April 2005.

[52] Benoit Lecland and Pierre Réty. Ttsli : An implementation of tree-tuple synchronized languages. In *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 376–379. Springer, 2002.

[53] A. Leitsch. Decision procedures and model building, or how to improve logical information in automated deduction. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *LNCS*, pages 62–79. Springer, 2000.

[54] S. Limet and P. Pillot. Solving first order formulae of pseudo-regular theory. In *International Colloquium on Theoretical Aspects of Computing (ICTAC05*, volume 3722 of *LNCS*, pages 110–124, 2005.

[55] S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars. *Discrete Mathematics and Theoretical Computer Science*, 1 :69–98, 1997.

[56] S. Limet and P. Rety. E-Unification by Means of Tree Tuple Synchronized Grammars. In *proc of the 6th Colloquium on Trees in Algebra and Programming*, number 1214 in Lecture Notes in Computer Science, Lille, France, 1997. Springer-Verlag.

[57] S. Limet and P. Rety. Solving Disequations modulo some Class of Rewrite System. In T. Nipkow, editor, *Proceedings of 9th Conference on Rewriting Techniques and Applications*, volume 1379 of *LNCS*, pages 121–135, Tsukuba, Japan, 1998. Springer Verlag.

[58] S. Limet and P. Rety. A New Result about the Decidability of the Existential One-step Rewriting Theory. In *Proceedings of 10th Conference on Rewriting Techniques and Applications, Trento (Italy)*, volume 1631 of *LNCS*, pages 118–132. Springer Verlag, 1999.

[59] S. Limet, P. Réty, and H. Seidl. Weakly Regular Relations and Applications. In *Proceedings of 12th Conference on Rewriting Techniques and Applications, Utrecht (the Netherlands)*, volume 2051 of *LNCS*, pages 185–200. Springer Verlag, 2001.

[60] S. Limet and G. Salzer. Manipulating tree tuple languages by transforming logic programs. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[61] S. Limet and G. Salzer. Proving properties of term rewrite systems via logic programs. In *Proc. 15th Int. Conf. on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *LNCS*, pages 170–184. Springer, 2004.

[62] S. Limet and G. Salzer. Basic rewriting via logic programming, with an application to the reachability problem. *Journal of Automata, Languages and Combinatorics*, 2005 to appear.

[63] S. Limet and G. Salzer. Tree tuple languages from the logic programming point of view. *Journal of Automated Reasoning*, 2005 to appear.

[64] S. Limet and F. Saubion. On Partial Validation of Logic Programs. In *Proceedings of the 6th internationnal Conference on Algebraic Methodology and Software Technology AMAST'97*, LNCS, Sidney, Australia, 1997. Springer Verlag.

[65] S. Limet and F. Saubion. A general framework for R-unification. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *proc of the Conf. on Principle of Declarative Programming (PLILP-ALP)*, volume 1490 of *LNCS*, pages 266–281. Springer, 1998.

[66] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[67] D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 274(1-2) :89–115, 2002.

[68] Christopher Lynch and Barbara Morawska. Faster basic syntactic mutation with sorts for some separable equational theories. In *Term Rewriting and Applications, 16th International Conference, RTA 2005*, volume 3467 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2005.

[69] Jerzy Marcinkowski. Undecidability of the first order theory of one-step right ground rewriting. In *8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 241–253. Springer Verlag, 1997.

[70] Richard Mayr. Process rewrite systems. *Information and Computation*, 156(1-2) :264–286, 2000.

[71] Catherine Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1) :5–36, 1992.

[72] José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Electr. Notes Theor. Comput. Sci.*, 117 :153–182, 2005.

[73] Takashi Nagaya and Yoshihito Toyama. Decidability for left-linear growing term rewriting systems. *Inf. Comput.*, 178(2) :499–514, 2002.

[74] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. On equality up-to constraints over finite trees, context unification and one-step rewriting. In *14th International Conference on Automated Deduction*, volume 1249 of *LNAI*, pages 34–48. Springer Verlag, 1997.

[75] F. Nielson, H. Riis Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations and spi. In *Proc. SAS'02*, volume 2477 of *LNCS*, pages 20–35. Springer Verlag, 2002.

[76] Robert Nieuwenhuis. Decidability and complexity analysis by basic paramodulation. *Information and Computation.*, 147(1) :1–21, 1998.

[77] A. Pettorossi and M. Proietti. Transformation of logic programs. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.

[78] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1) :89–124, 1995.

[79] J.-C. Raoult. Rational tree relations. *Bulletin of the Belgian Mathematical Society* *www.ulb.ac.be/assoc/bms/bms.bull.html*, 4(1) :149–176, 1997.

[80] P. Réty. Regular sets of descendants for constructor-based rewrite systems. In *Proc. of the 6th conference LPAR*, number 1705 in LNAI. Springer, 1999.

[81] Pierre Réty and Julie Vuotto. Regular sets of descendants by some rewrite strategies. In *Rewriting Techniques and Applications, 13th International Conference, RTA 2002*, volume 2378 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2002.

[82] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[83] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of formal languages, vol. 3 : beyond words*. Springer-Verlag, 1997.

[84] G. Salzer. The unification of infinite sets of terms and its applications. In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *LNCS*, pages 409–420. Springer, 1992.

[85] Frédéric Saubion and Igor Stéphan. A unified framework to compute over tree synchronized grammars and primal grammars. *Discrete Mathematics and Theoretical Computer Science*, 5 :227–262, 2002.

[86] H. Seki, T. Takai, Y. Fujinaka, and Y. Kaji. Layered transducing term rewriting system and its recognizability preserving property. In S. Tison, editor, *Proc. of 13th Conference RTA*, volume 2378 of *LNCS*. Springer, 2002.

[87] F. Seynhaeve, S. Tison, Tommasi M., and R. Treinen. Grid structures and undecidable constraint theories. *Theoretical Computer Science*, 258(1/2) :453–490, May 2001.

[88] T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *Proc. 11th Int. Conf. on Rewriting Techniques and Applications (RTA'00)*, volume 1833 of *LNCS*, pages 270–273. Springer, 2000.

[89] Toshinori Takai. A verification technique using term rewriting systems and abstract interpretation. In V. van Oostrom, editor, *Proc. 15th Int. Conf. on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *LNCS*, pages 119–133. Springer, 2004.

[90] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S. Tärnlund, editor, *Proc. 2nd Int. Logic Programming Conf. (ICLP)*, pages 127–138. University of Uppsala, Sweden, 1984.

[91] Ralf Treinen. The first-order theory of linear one-step rewriting is undecidable. *Theoretical Computer Science*, 208(1–2) :179–190, 1998.