Utilisation des langages d'arbres pour la modélisation et la vérification des systèmes à états infinis

Pierre Pillot

Utilisation des langages d'arbres pour la modélisation et la vérification des systèmes à états infinis

THÈSE

présentée

pour l'obtention du grade de

Docteur de l'Université d'Orléans

Discipline Informatique

par

Pierre Pillot

Membres du jury :

Olga Kouchnarenko	Professeur - Université de Besançon	
Sébastien Limet	Professeur - Université d'Orléans	
Sophie Tison	Professeur - Université de Lille	Rapporteur
Gernot Salzer	Professeur - Université Technique de Wien	Rapporteur
Pierre Réty	Maître de conférences, HdR - Université d'Orléans	Directeur de thèse

Remerciements

Table des matières

Chapitre 1

Introduction

Ce document présente les travaux que j'ai effectués au sein du projet PRV (Parallélisme Réalité virtuelle et Vérification des systèmes) du Laboratoire Fondamentale d'Informatique d'Orléans lors de ma thèse en Informatique. Cette thèse porte sur l'axe Vérification, a été dirigé par Pierre Réty, et est le fruit de l'encadrement de Sébastien Limet.

La modélisation et la vérification de systèmes distribués sont devenus de enjeux très importants dans le domaine du logiciel du fait du développement des technologies liées à l'Internet et aux réseaux haut débit. En effet, l'arrivée de tels réseaux a permis l'émergence de nouvelles architectures matérielles et logicielles pour la conception de systèmes d'information et de communication. Cette évolution rapide a eu pour effet de complexifier largement la conception, la mise au point et la vérification des applications informatiques du fait d'une distribution des traitements et de l'information sur des réseaux le plus souvent ouverts à des agents éventuellement malveillants. Il est donc primordial de pouvoir vérifier un certain nombres de propriétés de ces systèmes concernant aussi bien son bon fonctionnement que la sécurité et la confidentialité des informations échangées.

Par exemple afin de sécuriser les communications entre les points du réseaux des règles d'échanges (ou protocole) basées sur la cryptographie ont été créées. La cryptographie est basée sur un algorithme de chiffrement de message qui empêche en théorie toute personne qui n'a pas la clef de le déchiffrer. Malgrès cela des personnes étrangères à l'échange peuvent écouter, intercepter ou modifier les messages, ce qui dans certains cas peut lui permettre d'accéder à tout ou partie du message. Ainsi on peut vouloir qu'une partie du message reste inconnue de tierces peronnes, c'est la propriété de secret. Ou bien on peut vouloir être sûr que la personne avec laquelle on a communiqué est bien celle qu'elle prétend, c'est la propriété d'authenfication. Ou encore qu'aucune tierce personne ne puisse découvrir l'identitée d'un ou plusieurs des protagonistes du protocole, c'est la propriété d'anonymat.

La vérification de systèmes distribués se base généralement sur une spécification du système à l'aide d'outils formels tels les algèbres de processus ou le π-calcul d'une part, et d'autre part d'un langage d'expression des propriétés attendues du système qui utilise souvent des logiques prenant en compte le facteur temps (logique linéaire, temporelle ou transitionelle par exemple). Pour vérifier qu'un système a bien les propriétés attendues, il est le plus souvent nécessaire de recourir à des formalismes à base d'automates à états finis ou infinis ou à des langages d'arbres. La qualité des propriétés qui peuvent être vérifiées sur le système dépendent du pouvoir d'expression des automates ou langages utilisés. Mais l'augmentation de l'expressivité de tels outils doit éviter deux écueils très importants : l'indécidabilité des propriétés ou l'augmentation de la complexité théorique des algorithmes de vérification qui peuvent les rendre inutilisables en pratique.

Dans cette thèse nous nous intéressons aux modélisations qui se basent sur des termes. Les termes et les n-uplets de termes (aussi appelés n-uplets d'arbres) sont la structure de données de base dans de nombreux domaines de la logique et de l'informatique. Ils sont pour la vérification formelle, la déduction automatique, la programmation logique, ce que les entiers et les réels sont au calcul numérique. Généralement dans ces domaines nous n'avons pas à manipuler uniquement un terme ou un n-uplet mais plutôt des ensembles potentiellement infinis de ces objets comme par exemple dans la description de modèle dans la déduction automatique, l'approximation de calculs, ou la détection de boucles infinies.

Dans le domaine de la vérification formelle, les sytèmes de réécritures (ensemble de règles permettant d'écrire un terme en un autre terme) sont utilisés. Ce formalisme peut par exemple modéliser le comportement d'un programme ou d'un protocole. Parmis les problèmes de vérification où les systèmes de réécritures sont utilisées on trouve par exemple le problème d'atteignabilité (est-ce qu'on peut obtenir un terme t' à partir d'un terme t après l'application successives de règles d'un sytème de réécriture R) que l'on notera $t \rightarrow_R^* t'$ si il est satisfait. Ce problème permet d'étudier l'évolution du système modélisé. Un autre problème de vérification impliquant les systèmes de réécriture est celui de la R-unifiabilité (pour les termes t et t', y a t'il une application σ permettant d'instancier des variables par des termes telle que $t\sigma \rightarrow_R^* u \,{}_R^*\!\leftarrow t'\sigma$?). Par exemple si on a un système de réécriture R modélisant l'addition dans $\mathbb{N}$, on peut vouloir résoudre $x + 4 =_R 8$ qui nous donne une unique solution pour x : 4. On peut également vouloir vérifier des propriétés par exemple $\forall x, \forall y, ((x+2 =_R z \vee z+2 =_R y) \Rightarrow (x+4 =_R y))$. De manière générale à cet effet on a besoin de définir les ensembles de n-uplets qui satisfont chaque équation (par exemple pour l'équation $x + 2 =_R z$ c'est l'ensemble des couples d'entiers dont la différence est de 2) et de les comparer par la suite.

Pour des raisons pratiques et théoriques, on s'est intéressé à la représentation finie d'ensembles infinis afin de manipuler les représentations finis à

la place des ensembles infinis. Les problèmes typiques à résoudre sont

- comment construire une représentation finie d'un problème initial soit statiquement par des transformations syntaxiques, ou dynamiquement en fournissant par exemple des mécanismes de détections de boucles pendant un calcul, et
- comment exécuter les opérations comme l'union ou l'intersection sur des représentations finies et comment tester des propriétés telles que l'appartenance ou le vide.

La principale difficulté est de trouver un bon équilibre entre l'expressivité du formalisme choisi et la complexité des opérations définies dessus.

Un exemple significatif sont les automates d'arbres qui définissent la classe des langages d'arbres réguliers (arbre ici signifie terme sans variable). Ils sont clos par des opérations standards comme l'union, l'intersection, et le complément, à la fois l'appartenance et le vide sont des problèmes décidables, et la complexité de ces opérations[1] est faible. Par conséquent les automates d'arbres sont largement utilisés et ont trouvé des applications dans tous les domaines mentionnés au dessus [28, 73]. L'inconvénient des langages d'arbres réguliers est leur faible expressivité, ce qui a amené à faire des extensions comme les automates d'arbres avec contraintes [32], les automates d'ensembles d'arbres [45], et les relations régulières [28, 81].

Ces dernières sont un point central de ce mémoire. Dans [63] Limet et Salzer montrent qu'elles peuvent se modéliser par des programmes logiques particuliers, les cs-programmes réguliers. Ils ont également définis des outils à base de transformation de programmes logiques afin de pouvoir effectuer un certain nombre d'opérations ensemblistes dessus. Par ailleurs dans [64] ils ont définit une procédure générale de transformation de système de réécriture en programme logique qui permet un transfert de résultats entre les deux formalismes.

En se basant sur ces deux résultats, et en complétant les techniques de transformations de programmes concernant les cs-programmes réguliers nous avons pu dans [60] étudier la résolution de formules du premier ordre d'une théorie dont l'unique prédicat concerne la joignabilité modulo un système de réécriture d'une classe particulière. Nous nous sommes ensuite intéressés dans [61] aux formules de joignabilité du second ordre, les variables du second ordre étant interprétées comme des relations. Nous avons définis un algorithme générique qui décide de la satisfiabilité des formules positives du second ordre quand un algorithme de représentation finie des solutions des formules du premier ordre est connu. En se servant des résultats précédant nous avons appliqué cette technique à des formules particulières. Le résultat est alors un programme logique qui représente une instance des variables du second ordre. En définissant une transformation qui traduit l'instance en système de réécriture conditionnel, on peut alors synthétiser automatique-

[1] mise à part celle du complément

ment un programme qui définit une relation à partir de sa spécification.

Les outils à base de transformations de programmes logiques de [63] ne servent pas qu'à effectuer des opérations sur les cs-programmes réguliers. En effet ils utilisent ces outils pour transformer tout programme logique et requêtes sur ces programmes logiques en cs-programmes (potentiellement infinis) équivalent, les cs-programmes finis ayant la bonne propriété d'avoir un test du vide décidable (c'est à dire que l'on peut savoir si oui ou non le programme peut donner un résultat). Nous avons modifié cet outil afin qu'il puisse toujours nous donner en résultat un cs-programme fini, mais en contre partie on a perdu l'équivalence. Toutefois notre cs-programme fini est une sur-approximation de la requête sur le programme logique d'origine (c'est à dire que tout les résultats de la requête sur le programme logique d'origine sont égalements résultats du cs-programme fini obtenu). Ce qui signifie que lorsque l'on ne détecte aucun résultat sur notre cs-programme on est sûr que la requête sur le programme logique d'origine ne donne aucun résultat. Finallement on applique notre outil à la vérification de protocoles cryptographique, le programme logique d'origine étant une modélisation du réseau comprenant l'échange de messages par le protocole et une modification de ces messages par l'intrus, la requête représentant une propriétée rendue fausse par l'intrus.

Le chapitre 2 présentera les notions de bases sur les termes les langages dont on aura besoin pour appréhender les travaux. Le chapitre 3 présentera des notions quant à la programmation logique et des liens avec le langage qu'il peut exprimer. Le chapitre 4 présentera les notions de réécriture nécessaires à la compréhension de la suite du mémoire. Ensuite le chapitre 5 présentera des résultats sur la manipulation des langages réguliers sous forme de programmes logiques. Ensuite le chapitre 6 présentera les formules et la théorie du premier ordre que nous avons pu résoudre. Le chapitre suivant exposera son extension au second ordre. Le chapitre 8 exposera les résultats en cours concernant la sur-approximation de requête. Finalement dans le dernier chapitre je conclurai ce mémoire et exposerai des perspectives.

Chapitre 2

Notions de base

2.1 Introduction

Ce chapitre a pour objectif de présenter l'essentiel des notions de bases utiles à la compréhension de ce rapport et du travail effectué au cours de ma thèse. Ainsi je présenterai les notions de termes, de langages d'arbres, d'opérations sur ces langages en donnant l'exemple de quelques outils.

2.2 Arbres et termes

Un terme est une structure de donnée de base en informatique qui permet de modéliser ce que l'on souhaite manipuler, des nombres, des opérations ou d'autre concepts plus abstraits.

Définition 2.2.1. *Une* signature Σ *est un ensemble fini de* symboles de fonction*, où chaque $f \in \Sigma$ est associé à un entier n non négatif, l'*arité *de f. Pour $n \geq 0$, on note l'ensemble de tous les éléments n-aire de Σ par $\Sigma^{(n)}$. Les éléments de $\Sigma^{(0)}$ sont également appelés des* symboles constantes*.*

Exemple 2.2.2. *Si on veut seulement faire des additions en base 2 on aura par exemple comme signature $\Sigma = \{+_{/2}, 0_{/1}, 1_{/1}, \bot_{/0}\}$ où $+$ est d'arité 2, 0 et 1 sont d'arités 1 et $\bot$ est une constante exprimant la fin d'un nombre. Les nombres se lisant de gauche à droite $0(1(\bot))$ exprime le nombre 2.*

A partir de la signature et des variables (voir ci-dessous) on définit l'ensemble des termes sur lesquels on travaille.

Définition 2.2.3. *Soit Σ une signature et X un ensemble récursivement énumérable de* variables *tel que $\Sigma \cap X = \emptyset$. L'ensemble $T(\Sigma, X)$ de tous les* Σ-termes *sur X est inductivement défini comme : $X \subseteq T(\Sigma, X)$ (c'est à dire chaque variable est un terme), $\forall n \geq 0, \forall f \in \Sigma^{(n)}, \forall t_1, \ldots, t_n \in T(\Sigma, X), f(t_1, \ldots, t_n) \in T(\Sigma, X)$ (i.e les applications des symboles de fonctions aux termes donnent des termes).*

Exemple 2.2.4. *Σ est la même que dans l'exemple précédent et $X = \{x, y, z, t\}$ on a :*
$+(x, +(0(\perp), 1(1(z)))) \in T(\Sigma, X)$
$+(x, +(2(\perp), 0)) \notin T(\Sigma, X)$ *car* $2 \notin T(\Sigma, X)$ *et ici* 0 *n'a pas d'arguments alors que* 0 *est d'arité 1.*

On va maintenant définir un ensemble d'outils et de définitions sur les termes qui va nous permettre d'une part de caractériser ces termes et de permettre par la suite de mieux les manipuler.

Définition 2.2.5. *Soit Σ une signature, X un ensemble de variables disjoint de Σ, et $s, t \in T(\Sigma, X)$. L'ensemble de* positions *d'un terme s est un ensemble $Pos(s)$ de chaînes de caractères sur l'alphabet des entiers positifs,*

défini comme suit :
Si $s = x \in X$,alors $Pos(s) \stackrel{def}{=} \{\varepsilon\}$, ε désignant la chaîne de caractères vide.
Si $s = f(t_1, \ldots, t_n)$, alors $Pos(s) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{i.p | p \in Pos(s_i)\}$.
Pour $p \in Pos(s)$, le sous-terme de s à la position p, *noté par $s|_p$, est défini par induction sur la longueur de p :*
$s|_\varepsilon = s$,
$f(s_1, \ldots, s_n)|_{iq} = s_i|_q$.
Par $Var(s)$ on note l'ensemble des variables apparaissant dans s,
$Var(s) = \{x \in X | \exists p \in Pos(s), s|_p = x\}$.
On appelle $p \in Pos(s)$ une position de variable *si $t|_p$ est une variable.*
On définit $VarPos(t) = \{\, p \mid p$ est une position de variable de $t \,\}$ comme l'ensemble des positions de variables dans t.
L'ensemble $\Sigma Pos(t) \subseteq Pos(t)$ est l'ensemble des positions non variables de t, c'est à dire $\Sigma Pos = Pos(t) \setminus VarPos(t)$.

Exemple 2.2.6. *Avec Σ et X étant identiques à l'exemple précédent, on a pour le terme : $s = +(x, +(0(\perp), 1(1(z)))) \in T(\Sigma, X)$,*
$Pos(s) = \{\varepsilon, 1, 2, 2.1, 2.1.1, 2.2, 2, 2.1, 2.2.1.1\}$, $Var(s) = \{x, z\}$, $s|_{2.2.1} = 1(z)$

Définition 2.2.7. *Soit Σ une signature, et X un ensemble de variables disjoints de Σ et un terme $t \in T(\Sigma, X)$. Soit $p \in Pos(t)$ et $p = i_1.i_2.\ldots.i_n$, n est alors la longueur de p notée $|p|$ et la profondeur de t est $|t| = max(|p||p \in Pos(t))$.*

Exemple 2.2.8. *En prenant le terme s de l'exemple précédant $|s| = 5$.*

Définition 2.2.9. *Soit Σ une signature, et X un ensemble de variables disjointes de Σ et des termes $t_1, t_2, t_3 \in T(\Sigma, X)$. $t_1 = t_2[t_3]_p$ dénote le terme tel que $\forall q \in Pos(t_2)$, $q \prec p$ $q \neq p$ $t_1|_q = t_2|_q$ et $t_1|_p = t_3$.*

Exemple 2.2.10. *Soit le terme s et le terme $t_3 = +(0(\perp), y)$, on a $t_1 = s[t_3]_2 = +(x, +(0(\perp), y)$. On a remplacé le sous-terme de s à la position 2 par t_3.*

Définition 2.2.11. *Soit Σ une signature, et X un ensemble de variables disjoints de Σ. Un terme $t \in T(\Sigma, X)$ est dit linéaire si et seulement si $\forall x \in Var(t)\ \exists! p \in Pos(t)$ tel que $t|_p = x$.*

Exemple 2.2.12. *Σ et X étant les mêmes que définis précédemment on a :*
$+(x, +(0(\perp), 1(1(z)))) \in T(\Sigma, X)$ est linéaire
$+(x, +(0(x), 1(1(z)))) \in T(\Sigma, X)$ n'est pas linéaire car x est à deux positions différentes

Définition 2.2.13. *Un contexte C d'un terme t est un terme linéaire tel que $C[t_1, \ldots, t_n] = t$ et les variables de C sont remplacées par les termes $t_1, \ldots, t_n$ de gauche à droite.*

Exemple 2.2.14. *Soit le terme $t = +(0(y), +(0(\perp), 1(1(1(\perp)))))$ on a le s précédant qui est un contexte de t car :*
$s[0(y), 1(\perp)] = +(0(y), +(0(\perp), 1(1(1(\perp)))))$.

Définition 2.2.15. *Soit Σ une signature, et X un ensemble de variables disjoints de Σ. Un terme $t \in T(\Sigma, X)$ est dit clos si et seulement si $Var(t) = \emptyset$. L'ensemble de tous les termes clos sur Σ est noté par $T(\Sigma, \emptyset)$ ou $T(\Sigma)$.*

Exemple 2.2.16. *Σ et X étant les mêmes que définis précédemment on a :*
$+(\perp, +(0(\perp), 1(1(\perp)))) \in T(\Sigma, X)$ est clos
$+(\perp, +(0(\perp), 1(1(z)))) \in T(\Sigma, X)$ n'est pas clos car
$Var(+(\perp, +(0(\perp), 1(1(z))))) = \{z\}$.

Définition 2.2.17. *Soit Σ une signature et X un ensemble infini dénombrable de variables. Une $T(\Sigma, X)$-*substitution *(ou simplement substitution si la signature est clair selon le contexte) est une application $\sigma : X \to T(\Sigma, X)$ telle que $x\sigma \neq x$ pour un nombre fini de x. L'ensemble de toutes les $T(\Sigma, X)$-substitutions va être noté par $Sub(T(\Sigma, X))$ ou simplement Sub. On étend σ de $T(\Sigma, X) \to T(\Sigma, X)$ tel que $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.*

Exemple 2.2.18. *Σ et X étant les mêmes que définis précédemment on a par exemple :*
$\sigma : x \mapsto +(\perp, 0(x))$.
$+(x, x)\sigma = +(+(\perp, 0(x)), +(\perp, 0(x)))$.

Définition 2.2.19. *Soit Σ une signature, et X un ensemble de variables disjointes de Σ. On parle de renommage d'un terme t par une substitution σ si et seulement si $t\sigma\sigma^{-1} = t$ et $Var(t\sigma) \cap Var(t) = \emptyset$ Un terme s est une* instance *d'un autre terme t s'il existe une substitution σ telle que $s = \sigma(t)$.*

Exemple 2.2.20. *Soit le terme s défini précédemment par $+(x, +(0(\perp), 1(1(z))))$, un rennomage de s peut être $+(y, +(0(\perp), 1(1(u))))$, les deux termes étant identiques à l'exception des ensembles de variables qui sont disjoints. Le terme $t = +(0(y), +(0(\perp), 1(1(1(\perp)))))$ est une instance de s car on a σ définit par $\sigma : x \mapsto O(y)$ et $\sigma y :\mapsto 1(\perp)$ tel que $t = \sigma s$*

Définition 2.2.21. *L'*unification *de deux termes s et t consiste à trouver une substitution σ appelée* unificateur*, telle que $\sigma(s) = \sigma(t)$, s et t sont alors dits* unifiables*. Si deux termes s et t sont unifiables il existe une substitution σ appelée unificateur le plus général (mgu) unique à un renommage près tel que pour tout unificateur θ de s et t il existe τ tel que $\theta = \tau \circ \sigma$.*

Exemple 2.2.22. *Soit les termes $+(x, 0(y))$ et $+(0(z), 0(y))$, un unificateur de ces deux termes est σ tel que $\sigma(x) = 0(z)$. Soit les termes $+(x, 0(y))$ et $+(0(z), 1(y))$, ils n'ont pas d'unificateur car les symboles de fonction à la position 2 sont différents, l'opération de substitution permettant seulement de remplacer les variables aux positions qu'elles occupent par des termes à la même position.*

2.3 Langages d'arbres et de n-uplet d'arbres

De manière simplifiée on pourra définir le language comme ce qui est produit en résultat par un processus. Afin de connaitre ou de vérifier ce que produit un processus on a besoin tests. Malheureusement en général on a une infinité de productions différentes qu'on ne peut prédire, c'est le problème de la décidabilité-indécidabilité que l'on va voir en section 2.3.1. On définira ensuite dans la section 2.3.2 des propriétés de langages qui sont désirés qui permettent de contrôler le processus. Enfin dans les sections 2.3.3 et 2.3.4 on définira différents processus de production qui vont satisfaires certains de ces tests et propriétés.

2.3.1 Décidabilité

Cette section a pour but unique de donner les définitions de notions de décidabilité que nous utiliserons par la suite.

Une des grandes préoccupations de l'informatique théorique est de classer les différents problèmes qui lui sont donnés selon leur degré de difficulté (c'est le calcul de complexité) et selon leur limite théorique (c'est le problème de la

décision). Le premier critère essaie de calculer une approximation du temps ou de l'espace mémoire nécessaire pour résoudre un problème donné. Le second, s'attache à déterminer de manière théorique si oui ou non il existe un algorithme permettant de résoudre un problème donné et, dans le cas où il existe, si cet algorithme termine ou non. Ce rapport ne donnera pas de nouvel élément concernant le premier critère, par contre il arrivera qu'on se confronte au problème de décision de problème. On peut trouver tous les outils de base concernant les problèmes de décision dans le livre [50].

Définition 2.3.1. *Un problème est dit* décidable *s'il existe un algorithme qui pour n'importe quelle instance du problème, termine en répondant oui ou non. Dans les autre cas il est dit* indécidable

Le *test du vide* est décidable si on peut dire qu'il existe un terme clos (ou n-uplet de termes selon le cas) qui appartient au langage, ou qu'il n'en existe aucun. Il est indécidable sinon.

Le *test d'appartenance* est décidable si pour tout terme clos t (ou n-uplet de termes selon le cas) on peut dire s'il appartient ou non au langage. Il est indécidable sinon.

2.3.2 Propriétés de clôture

Une classe est *close* par une opération particulière lorsque l'opération appliquée à n'importe quel(s) objet(s) de cette classe donne un objet de la même classe. Une *propriété de clotûre* d'une classe de language est le fait qu'une classe est close par une opération particulière.

Définition 2.3.2. *Soit $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages de n-uplets d'arbres sur la signature Σ d'arité respectives n_1 et n_2, soit π une permutation sur $[1, n_1]$.*

- *L'*intersection *de deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ avec $n_1 = n_2 = n$ notée $\mathcal{L}_1 \cap \mathcal{L}_2$ est l'ensemble $\{ (t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in \mathcal{L}_1$ et $(t_1, \ldots, t_n) \in \mathcal{L}_2 \}$,*
- *L'*union *de deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ avec $n_1 = n_2 = n$ notée $\mathcal{L}_1 \cup \mathcal{L}_2$ est l'ensemble $\{ (t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in \mathcal{L}_1$ ou $(t_1, \ldots, t_n) \in \mathcal{L}_2 \}$,*
- *Le* complément *d'un langage $\mathcal{L}_1$ d'arité n notée $\overline{\mathcal{L}_1}$ est l'ensemble $\{ (t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \notin \mathcal{L}_1 \}$,*
- *La* restriction*de $\mathcal{L}_1$, notée $\mathcal{L}_1\_i$ 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 :*
  $\mathcal{L}_1\_i_1, \ldots, i_K = (\mathcal{L}_1\_i_1)\_j_2, \ldots, j_k$ où $j_l = i_l$ si $j_l < i_1$ et $j_l = i_l - 1$ si $j_l \geq i_1$,
- *La* projection *de $\mathcal{L}_1$ sur l'ensemble des composants $\{i_1, \ldots, i_k\} \subseteq 1, \ldots, n_1$, notée $\mathcal{P}_{i_1,\ldots,i_k}(\mathcal{L}_1) = \mathcal{L}_1\_\{j \in \{i_1, \ldots, n_1\}$ $\{i_1, \ldots, i_k\}\}$,*

- *La* cylindrification *de* $\mathcal{L}_1$ *sur le composant* $i (0 \leq i \leq n_1$*, notée* $C_i(\mathcal{L}_1)$*, est le langage d'arité* n_1+1 $\{ (s_1, \ldots, s_{n_1+1}) \mid \exists (t_1, \ldots, t_{n_1}) \in \mathcal{L}_1$ *et* $s_j = t_j$ *pour* $1 \leq j < i$ *et* $s_j = t_{j-1}$ *pour* $i \leq j \leq n_1 + 1 \}$*. 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 :*
  $C_{i_1,\ldots,i_K}(\mathcal{L}_1) = C_{j_2,\ldots,j_k}(C_{i_1}(\lambda_1))$ *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* $\sigma(\mathcal{L}_1)$*, 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* $\mathcal{L}_1 \bowtie_{i,j} \mathcal{L}_2$ *est le langage d'arité* $n_1 + n_2 - 1$ *:*
  $\{ (s_1, \ldots, s_{n_1}, t_1, \ldots, \ldots, t_{j-1}, t_{j+1}, \ldots, tn_2) \mid (s_1, \ldots, s_{n_1}) \in \mathcal{L}_1$ *et* $(t_1, \ldots, t_{j-1}, s_i, t_{j+1}, \ldots, t_{n_2}) \in \mathcal{L}_2 \}$*.*

2.3.3 Régularité

Dans cette section on définit des processus qui vont nous permettre d'avoir un important contrôle sur la production de ceux-ci. En effet ces processus sont stables par union, intersection, complément. Le test du vide et le test d'appartenance étant, quant à eux, décidable. Plus de détails sur les automates peuvent être trouvés dans [28], qui décrit également les relations régulières, mais sous le nom de relations reconnaissables.

Définition 2.3.3. *Un* automate fini d'arbres *sur* $\mathcal{F}$ *est un tuple* $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ *où* Q *est un ensemble d'états (unaires),* $Q_f \subseteq Q$ *est un ensemble d'états finaux, et* Δ *est un ensemble de règles de transition du type suivant :*
$f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow q(f(x_1, \ldots, x_n))$,
où $n \geq 0$, $f \in \mathcal{F}_n, q, q_1, \ldots, q_n \in Q, x_1, \ldots, x_n \in X$.

Exemple 2.3.4. $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$, $q, q_1, q_2, q_3 \in Q$, $Q_f = \{q_3\}$ *et* δ *est :*
$f(q_1(x_1), q_3(x_2)) \rightarrow q(f(x_1, x_2))$,
$g(q_2(x_1), q_3(x_2)) \rightarrow q(g(x_1, x_2))$,

Définition 2.3.5. *Soit* $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ *un automate d'arbre fini sur* $\mathcal{F}$*. La* relation de transition $\rightarrow_{\mathcal{A}}$ *est défini par :*

$$t, t' \in T(\mathcal{F} \cup Q) \stackrel{\text{def}}{=} \begin{cases} \exists C \in \mathcal{C}(\mathcal{F} \cup Q), \exists u_1, \ldots, u_n \in T(\mathcal{F}), \\ \exists f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow q(f(x_1, \ldots, x_n)) \in \Delta, \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))], \\ t' = C[q(f(u_1, \ldots, u_n))] \end{cases}$$

La clôture transitive et réflexive de $\rightarrow_{\mathcal{A}}$ est notée par $\rightarrow_{\mathcal{A}}^{*}$.

Une manière d'étendre les automates finis aux n-uplets d'arbres est de représenter un n-uplet par un arbre qui représente l'ensemble du n-uplet en "mélangeant" tout les termes de ce n-uplet.

Définition 2.3.6. *Soit $F' = (F \cup \{\perp\})^n$, où $\perp$ est un nouveau symbole. Soit k l'arité maximale des symboles de fonctions dans F. En posant à 0 l'arité de $\perp$, les arités des symboles dans F' sont définis par $a(f_1, \ldots, f_n) = max(a(f_1), \ldots, a(f_n))$. Le codage de deux termes $f(t_1, \ldots, t_n), g(u_1, \ldots, u_m) \in T(F)$ est défini par induction :*

si $n \geq m$
$[f(t_1, \ldots, t_n), g(u_1, \ldots, u_m)] =^{def} fg([t_1, u_1], \ldots, [t_m, u_m], [t_{m+1}, \perp], \ldots, [t_n, \perp])$

et si $m \geq n$.
$[f(t_1, \ldots, t_n), g(u_1, \ldots, u_m)] =^{def} fg([t_1, u_1], \ldots, [t_n, u_n], [\perp, u_{n+1}], \ldots, [\perp, u_m])$

Plus généralement le codage de n termes $f_1(t_1^1, \ldots, t_1^{k_1}), \ldots, f_n(t_n^1, \ldots, t_n^{k_n})$ est défini par
$f_1 \ldots f_n([t_1^1, \ldots, t_n^1], \ldots, [t_1^m, \ldots, t_n^m])$
où m est l'arité maximale de $f_1, \ldots, f_n \in F$ et t_i^j est, par convention, $\perp$ quand $j > k_i$.

Exemple 2.3.7. *soit $f(x_1, s(0))$ et $g(s(0), x_2)$, on a*
$[f(x_1, s(0)), g(s(1), x_2)] = fg([x_1, s(1)], [s(0), x_2]) = fg(x_1 s[\perp, 1], s x_2[0, \perp]) = fg(x_1 s(\perp 1), s x_2(0 \perp))$

Définition 2.3.8. *Rec est l'ensemble des relations régulières qui sont de la forme $R \subseteq T(F)^n$ tel que $\{[t_1, \ldots, t_n] | (t_1, \ldots, t_n) \in R\}$ est reconnu par un automate fini d'arbres sur l'alphabet $F' = (F \cup \{\perp\})^n$.*

L'intérêt des relations régulières c'est leurs propriétés qui permettent une manipulation facile. Les relations régulières sont ainsi stables par intersection, union, complément, produit cartésien, etc. L'inconvénient est que les automates et relations régulières ont une expressivité et un formalisme contraignant. D'autres formalismes dont le but est d'étendre l'expressivité des relations régulières ont été étudiés. On citera par exemple [37, 86, 92]. La section suivante en préseentera une autre : les systèmes de contraintes qui ont étés introduits dans [47] comme une version simplifiée des grammaires d'arbres synchronisés de [62].

2.3.4 Langages de n-uplets d'arbres

Soit Σ un ensemble fini de symboles avec arité, et soit T_Σ l'ensemble de tous les termes clos sur Σ. Les sous-ensembles de ${T_\Sigma}^n$ sont appelés *n-uplets de langages d'arbres n-aire sur Σ* (ou langage de n-tuple).

Un *template* est un terme clos qui en plus peut contenir des entiers positifs à la place de constantes. Formellement, si ω est l'ensemble des entiers positifs, alors $T_{\Sigma\cup\omega}$ est l'ensemble des templates sur Σ. Les entiers sont appelés *indices* et sont utilisés pour faire référence à un composant particulier du langage de n-uplets. Nous notons les substitutions comme des ensembles, c'est à dire, $A\{s_1 \mapsto t_1, \ldots, s_l \mapsto t_l\}$ est obtenu à partir de A en remplaçant simultanément toutes les occurrences de s_i par t_i. L'arité d'un objet O est noté par $\mathrm{ar}(O)$, où O peut être une fonction ou un symbole de prédicat, une variable de langage, ou une expression de langage.

Soit A un langage n-uplet, et soit $\Box$ un k-uplet de templates avec l comme le plus grand indice apparaissant dans $\Box$, ou 0 si il n'y en a aucun.

Les opérations *construction* et *filtrage* sont définis par :

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

On notera que l'opérateur de construction remplace les indices dans $\Box$ qui excèdent l'arité de A par un terme clos arbitrairement choisi.

Exemple 2.3.9. *Soit* $\Sigma = \{a/0, f/1, g/2\}$, $A = \{(a, f(a)), (f(a), a)\}$, *et soit* $\Box = [g(1,3), 2]$ *une paire de templates. on obtient :*

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

On définit ici des définitions inductives avec des relations de sous ensembles sur des langages d'expression. Soit $\mathcal{X}$ un ensemble de variables de langage. L'ensemble des *expressions de langage* est définie par la grammaire

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

pour les variables de langage A et les n-uplets de template $\Box$.

Une interprétation λ est une application associant un langage de n-uplet avec chaque variable de langage n-aire. Elle s'étend naturellement à une intérprétation pour les expressions de langage : $\lambda(\{()\}) = \{()\}$, $\lambda(e_1 \times e_2) = \lambda(e_1) \times \lambda(e_2)$, $\lambda(\Box \circ e) = \Box \circ \lambda(e)$, et $\lambda(e/\Box) = \lambda(e)/\Box$. Une interprétation, λ, est plus petite ou égale à une autre, λ', si $\lambda(A) \subseteq \lambda'(A)$ pout tout $A \in \mathcal{X}$.

Une *définition inductive* est un ensemble de contraintes de la forme $A \supseteq e$ tel que les aritées de A et e [1] coïncident. Le langage défini par une définition inductive D est donné par la plus petite interprétation λ telle que $\lambda(A) \supseteq$

[1] L'arité d'une expression de langage est le nombre de composants dans le n-uplet du langage défini par celui-ci ; il peut être déterminé directement car l'arité d'une variable de langage est donnée.

$\lambda(e)$ pour toutes les contraintes ; il est défini de manière unique et peut être obtenu comme le plus petit point fixe des contraintes. On note la plus petite interprétation $\mathcal{L}_{\mathcal{D}}$, où juste par $\mathcal{L}$ si la définition inductive est claire selon le contexte.

Dans [47] les auteurs ont introduits les systèmes de contraintes comme version simplifiée d'un formalisme traitant de langage de n-uplet d'arbres dont le test du vide est décidable. Ils ont montrés que tous les systèmes de contraintes peuvent être normalisées sous la forme $A \supseteq \Box \circ (A_1 \times \cdots \times A_k)$ avec $A, A_1, \ldots, A_k$ variables de langage. Pour les systèmes de contraintes il est pratique d'écrire les indices comme couple d'entiers, $i.j$, faisant référence au j-ème composant de A_i ; les indices l excédant l'arité de $A_1 \times \cdots \times A_k$, noté m, sont écrit comme $(k+l-m).1$. Une contrainte $A \supseteq \Box \circ (A_1 \times \cdots \times A_k)$ est dite

- *linéaire* si et seulement si aucun indice n'apparaît deux fois dans $\Box$;
- *horizontale* si et seulement si pour n'importe quels indices $i.j$ et $i'.j'$ dans $\Box$, $i = i'$ implique que $i.j$ et $i'.j'$ apparaissent dans $\Box$ à une position de la même profondeur ;
- *pseudo-régulier* si et seulement si pour une application $\pi \colon \omega \mapsto \omega$, chaque composant de $\Box$ est de la forme $f(i_1.j_1, \ldots, i_{\mathrm{ar}(f)}.j_{\mathrm{ar}(f)})$, où $\pi(i_l) = l$ pour $l = 1, \ldots, \mathrm{ar}(f)$.
- *régulier* si et seulement si c'est pseudo-régulier et linéaire.

Un système de contraintes est appelé linéaire (horizontal, régulier, pseudo-régulier) si et seulement si toutes les contraintes ont les propriétés respectives. On remarquera aisément que la régularité implique l'horizontalité et, par définition, la linearité. Syntaxiquement, la pseudo-régularité est plus libre que la régularité en admettant la duplication de certains indices. On notera également que les systèmes de contraintes réguliers encodent les relations régulières.

Exemple 2.3.10. *La contrainte*

$$X \quad \supseteq \quad [f(1.1, f(2.1, 2.1)), f(1.2, 2.2)] \circ (X \times X)$$

est ni horizontale (indice 2.1 *apparaît à la profondeur 2, tandis que l'indice* 2.2 *apparaît à la profondeur 1) ni linéaire (l'indice* 2.1 *apparaît deux fois). La contrainte*

$$X \quad \supseteq \quad [f(1.1, 2.1), f(1.1, 2.2)] \circ (X \times X)$$

est pseudo-régulière mais non régulière car l'indice 1.1 *apparaît deux fois à la position 1 de deux composants. On remarquera que la définition ne requiert pas que les indices dupliqués apparaissent dans tous les composants.*

Les systèmes de contraintes bien qu'ayant un formalisme plus puissant que les automates, sont toujours compliqués à utiliser. L'idée d'utiliser des programmes logiques qui est un outil dont le formalisme et l'utilisation sont bien connus, s'impose alors.

Chapitre 3

Programmation Logique

Dans mes travaux de thèse je vais me servir des programmes logiques afin de calculer des opérations sur les langages de n-uplet d'arbres. C'est un formalisme bien connu et intuitif. Le fondement de la programmation logique que l'on va utiliser est la théorie des clauses de Horn. On se restreint à ces clauses car elles satisfont l'objectif de considérer des formules comme des programmes et la construction de leur preuve comme l'exécution de ces programmes. Une première propriété utilisée dans ce chapitre c'est que la programmation logique en clause de Horn est calculatoirement complète [5, 98]. En d'autre termes, toute relation entre entrée et sortie qui peut être définie dans un autre formalisme peut l'être aussi par une théorie de Horn. On s'intéresse à cette propriété car elle permet d'envisager un transfert de résultats d'un formalisme à celui des clauses de Horn, on le verra pour les langages définis précédemment. Par ailleurs on s'intéressera également a des transformations de programmes par équivalence logique. Celles-ci permettent notament d'obtenir des classes sur lesquels on sait effectuer des opérations à partir de classes sur lesquels les techniques usuelles pour calculer les opérations ne donnent pas de bons résultats.

La section 3.1 définira la syntaxe et la terminologie des programmes logiques que l'on utilise. Ensuite on verra dans la section 3.2 que les langages réguliers sont définis par une classe de programmes logiques particuliers. On en profitera pour montrer des extensions syntaxiques des automates sous la forme de programmes logiques. Ensuite on verra un technique permettant le transfert de résultats des systèmes de contraintes aux clauses de Horn, ce qui a amené à définir une sous-classe de programme logique dont le test du vide est décidable : les cs-programmes. Enfin on verra une transformation basée sur les techniques de pliage \ dépliage [84] permettant de transformer certains programmes logiques en cs-programmes.

3.1 Programmation Logique

On suppose que notre programme logique est écrit en utilisant les symboles d'une signature fixée Σ.

Définition 3.1.1. *Soit $\Sigma = \{\mathcal{F}, P_r, X\}$ un langage fixé où $\mathcal{F}$ est l'ensemble des symboles de fonction, P_r est l'ensemble des prédicats, et X est l'ensemble des variables. La signature est constituée d'un ensemble de prédicat, un ensemble de termes et un ensemble de variables.*

Un atome *est une formule de la forme : $p(t_1, \ldots, t_n)$ où $p \in P_r$ n-aire, et $t_i \in T(F, X)$.*

Un but *est une* séquence *finie, éventuellement vide, d'atomes.*

Une clause *est une formule de la forme : $H \leftarrow \mathcal{B}$, où la* tête *H est un atome et le* corps *$\mathcal{B}$ est un but (éventuellement vide dans ce cas une clause est un fait). Une variable sera dite* existentielle *si il elle apparaît dans le corps de la clause mais pas dans la tête.*

Dans le cadre ce cette thèse on va considérer seulement des *programmes logiques* comme une *séquence* finie de clauses.

Exemple 3.1.2. *Soit $\Sigma = \{F, P_r, X\}$ avec $F = \{f, g, 0, 1\}$, $P_r = \{P, E\}$ et $X = \{x, y, z\}$, f, g des fonctions d'arité 2, 0 et 1 des constantes.*
$P(f(g(0,1),x),1)$ est un atome. $P(x,y)E(0,x)$ est un but, $E(x,y) \leftarrow P(x,y)$, $E(0,x)$ est une clause. $\mathcal{P} = \{\{E(x,y) \leftarrow P(y,x)\}, \{P(0,1)\}\}$ est un programme logique

L'univers de Herbrand et la base de Herbrand sont construit à partir de Σ, indépendamment des programmes. Cette supposition qui est quelque fois faite dans la théorie des programmes logiques , dans notre cas est motivé par l'utilité d'avoir un univers de Herbrand constant lors des transformations de programmes.

Définition 3.1.3. *Le* domaine de Herbrand *D_p d'un programme logique $\mathcal{P}$ est $\mathcal{T}(F)$.*
La base de Herbrand *, est $B_p = \{P(t_1, \ldots, t_n) | t_i \in \mathcal{T}(F)$ et $P \in P_r$ d'arité $n\}$.*
*On utilise la notion d'*interprétation de Herbrand*, qui sont des sous-ensemble de la base de Herbrand, pour spécifier la sémantique des programmes logique.*
Un modèle de Herbrand *d'un programme logique est une interprétation qui satisfait toutes les clauses d'un programme logique.*
Le plus petit modèle d'un programme $\mathcal{P}$ est noté par $\mathcal{M}(\mathcal{P})$. L'ensemble des n-uplets de termes pour lequel un prédicat $\mathcal{P}$ est vraie, est défini par $\mathcal{M}(\mathcal{P})|_P = \{ (t_1, \ldots, t_n) \mid P(t_1, \ldots, t_n) \in \mathcal{M}(\mathcal{P}) \}$

Exemple 3.1.4. *Le domaine de Herbrand de l'exemple précédent est*
$D_P = \{0, 1, f(0,0), g(0,0), \ldots, f(t_1,t_2), g(t_3,t_4), \ldots\}$ où t_1, t_2, t_3, t_4 sont des

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

FIG. 3.1 – Unification

termes.
La base de Herbrand de l'exemple précédent est :
$\{E(0,0),\ldots,E(d_1,d_2),P(d_3,d_4),\ldots\}$ *où* $d_1,d_2,d_3,d_4 \in D_P$ *sont clos.*
Une interprétation de Herbrand de l'exemple précédent est :
$\{E(0,1),P(1,f(1,1))\}$. *Un modèle de Herbrand pour l'exemple précédent est :* $\{P(0,1),E(1,0)\}$. *On notera alors* $\mathcal{P} \models \{P(0,1),E(1,0)\}$

Le mécanisme opérationnel de la programmation logique est la *résolution*. Ce mécanisme se base sur l'algorithme de l'unification de termes. Cet algorithme, inventé par A.Robinson [88] a pour but de trouver l'unificateur le plus général de deux termes. Il existe de nombreuses manières de le présenter, la figure 3.1 utilise une approche inspirée de [10]. Un problèùme d'unification est un ensemble d'équation $\{s_1 =^? t_1,\ldots,s_n =^? t_n\}$, 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 $x =^? t$avec $x \notin Var(t)$, et x n'apparaît dans aucune autre équation de E. Dans ce cas l'unificateur le plus général est $\mu = \{\, x \mapsto t \mid x =^? t \in E \,\}$. Notons que nous inversons les équations $t =^? x$ de manière à ce que les variables apparaissent toujours à gauche des équations.

3.2 Des automates aux programmes logiques

La traduction des automates d'arbres en clauses de Horn est relativement simple, et permettra ainsi de se fixer les idées dans un seul formalisme.

En considérant chaque état q de l'automate comme un symbole de prédicat d'arité 1 P_q, les automates d'arbres peuvent être transformées en clauses de Horns de telle sorte que le langage reconnu par l'état q est exactement l'interprétation de P_q dans le plus petit modèle de Herbrand de l'ensemble des clauses ainsi générées.

Les règles d'inférences suivantes permettent de transformer un automate en programme logique.

$$\frac{f(q_1, \ldots, q_n) \to q'}{P_{q'}(f(x_1, \ldots, x_n)) \leftarrow P_{q_1}(x_1) \ldots P_{q_n}(x_n)}$$

Le langage accepté par l'automate est l'union des interprétations de P_q pour $q \in Q_f$ dans le plus petit modèle des clauses de Herbrand. On appellera ce type de clause les *clauses d'automate*.

Maintenant on va définir différentes extensions syntaxique des automates sous formes de clauses de Horns.

Définition 3.2.1. *$H \leftarrow \mathcal{B}$ est dite* clause alternante *si H est linéaire d'arité 1 et tout les atomes de $\mathcal{B}$ sont d'arité 1 et sans symboles de fonction. Un programme logique est alternant si toutes ses clauses le sont.*

Exemple 3.2.2. *$A(f(x, y)) \leftarrow A(x), A(y), B(x)$ est une clause alternante.*
$A(f(x, x)) \leftarrow A(x), A(y), B(x)$ n'est pas une clause alternante car x est répété deux fois dans la tête de clause.

Définition 3.2.3. *$H \leftarrow \mathcal{B}$ est dite une* clause à deux voies *si H et tous les atomes de $\mathcal{B}$ sont linéaires et soit*

- *H est sans symbole de fonction et $\mathcal{B}$ ne contient qu'un atome linéaire, soit*
- *$H \leftarrow \mathcal{B}$ est une clause d'automate.*

Un programme logique est dit à deux voies si toutes ses clauses le sont.

Exemple 3.2.4. *$A(y) \leftarrow A(f(x, y))$. est une clause a deux voies.*

Définition 3.2.5. *$H \leftarrow \mathcal{B}$ est dite une* clause à deux voies alternantes *si H et tous les atomes de $\mathcal{B}$ sont linéaires et soit*

- *H est sans symbole de fonction et $\mathcal{B}$ ne contient qu'un atome linéaire, soit*
- *$H \leftarrow \mathcal{B}$ est sans symbole de fonction et ne contient qu'une seule variable, soit*
- *H est linéaire et B est linéaire et sans symbole de fonctions.*

Un programme logique est dit à deux voies alternants si toutes ses clauses le sont.

Exemple 3.2.6. *$A(x) \leftarrow A(x)B(x)C(x)$ est une clause à deux voies alternantes.*
$A(f(x, f(y, z))) \leftarrow A(x)B(y)$ est une clause à deux voies alternantes.
$A(x) \leftarrow A(f(f(x, y), z)$ est une clause à deux voies alternantes

Lemme 3.2.7. *Tout programme logique a deux voies alternante est équivalent à un programme logique d'automate.*

La preuve de ce lemme est évidente en prennant le lemme correpondant sur les alternating two-way automata de [99].

On a vu que des extensions syntaxiques sont possibles tout en conservant les bonnes propriétés de clôtures et de décidabilité par équivalence logique. On peut aussi améliorer l'expressivité avec des contraintes sur l'application des transitions comme dans [14, 15] tout en garde de bonnes propriétés. Toutefois cette extension n'est pas prouvée équivalente aux programmes d'automates. Un autre exemple plus récent qui traite de n-uplets de termes provenant de [81] étendant les résultats de [42], et trouvant sa raison d'être dans son lien au π-calcul est présenté ci-après.

Dans ces travaux on trouve des classes de programmes logiques différents. On verra ici celles de [81] traitant de relations et ayant une contrainte syntaxique réduite. Ils définissent initiallement des relations fortements reconnaissables par 3 types de clauses linéaires sous une forme normalisée. Elles sont définies de la manière suivante :

- $P(b) \leftarrow$;
- $P(f(x_1, \ldots, x_k)) \leftarrow Q_1(x_1), \ldots, Q_k(x_k)$;
- $P(x_1, \ldots, x_k) \leftarrow Q_1(x_1), \ldots, Q_k(x_k)$.

La particularité de la dernière clause par rapport aux précédentes est qu'elle permet de définir une relation. Les relations fortements reconnaissables correspondent en fait à un n-uplet d'automate sans synchronisation. Leur but est d'obtenir des relations fortements reconnaissables sous la forme de clause normalisées à partir de n'importe quel programme logique dont les clauses sont du type :
$H \leftarrow \mathcal{B}$ où H est linéaire et si x et $y \in Var(H)$ tels que x et y soient dans la même composante de la décomposition de $\mathcal{B}$ en composantes sans variables commune, alors x et y sont sous le même symbole de fonction dans H. Pour atteindre leur but ils se basent sur une transformation dont l'idée est la suppression de lien inutiles entre les variables. Par exemple la clause $P(f(x, y), z) \leftarrow P(x, y), Q(z)$ est équivalente à :
$P(z', z) \leftarrow P'(z'), Q(z)$
$P'(f(x, y)) \leftarrow P(x, y)$
Grâce notament à cette transformation ils arrivent à atteindre leur but en gardant l'équivalence entre le programme initial et le programme sous forme normalisé.

Dans la section suivante on va s'intéresser à la traduction de langages en clauses de Horn qui traite de n-uplets de termes et qui permettent également une synchronisation entre les différents uplets de la relation.

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

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

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m),\, \mathcal{G}\rangle \quad f \rightsquigarrow \langle (t_1, \ldots, t_n),\, \mathcal{H}\rangle}{(e \times f) \rightsquigarrow \langle (s_1, \ldots, s_m, t_1, \ldots, t_n),\, \mathcal{G} \cup \mathcal{H}\rangle} \quad \text{si } \mathcal{G} \text{ et } \mathcal{H} \text{ ne partagent pas de variables.}$$

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m),\, \mathcal{G}\rangle}{(\Box \circ e) \rightsquigarrow \langle \Box\{1 \mapsto s_1, \ldots, m \mapsto s_m, m{+}1 \mapsto x_1, m{+}2 \mapsto x_2, \ldots\},\, \mathcal{G}\rangle}$$

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m),\, \mathcal{G}\rangle}{(e/\Box) \rightsquigarrow \langle (x_1, \ldots, x_l)\mu,\, \mathcal{G}\mu\rangle} \quad \begin{array}{l}\text{si } \mu, \text{ le m.g.u. de } (s_1, \ldots, s_m) \text{ et} \\ \Box\{1 \mapsto x_1, \ldots, l \mapsto x_l\}, \text{ existe ; } l \text{ indique} \\ \text{l'index maximal apparaissant dans } \Box \text{ ou } 0 \\ \text{si il n'y en a aucun.}\end{array}$$

FIG. 3.2 – Conversion d'expressions de langage en clauses logiques

3.3 Définitions inductives vs. Logique de Horn

Nous définissons ici une traduction d'expressions de langages en clauses de Horn équivalentes sémantiquement et vice versa provenant de [63]. On sera alors capable de discuter des propriétés de clôture et problèmes de décidabilité des systèmes de contraintes dans un style purement clausal pour pouvoir utiliser les résultats des domaines de transformations de programmes logiques et les preuves de théorèmes clausal.

Pour n'importe quelle variable de langage A d'arité n, soit P_A représentant un symbole de prédicat n-aire uniquement associé à A. La clause correspondante à une contrainte est définie par

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

La relation $\rightsquigarrow$ est spécifiée dans la Figure 3.2. Pour une définition inductive $\mathcal{D}$, i.e., 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 3.3.1. *Soit* $\Sigma = \{a/0, s/1\}$. *Les contraintes*

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

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

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

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

FIG. 3.3 – Conversion des expressions de l'exemple 3.3.1 en clauses logique

définissent les langages

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

on obtient les clauses de Horn pour la définition inductive :

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

La transformation du langage d'expressions en logique de Horn est donnée par la figure 3.3.

Pour chaque symbole de prédicat P d'arité n, soit A_P une variable de langage n-aire uniquement associée à P, et soit σ une substitution fixée remplaçant chaque variable du premier-ordre par un unique entier positif, i.e., $\sigma = \{x_1 \mapsto 1, x_2 \mapsto 2, \ldots\}$. on défini les contraintes correspondantes à une clause de programme par

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

et la définition d'induction définie par un programme logique, $\mathcal{P}$, comme

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

Les langages définis par une définition inductive $\mathcal{D}$ sont les plus petits points fixes des contraintes. Le plus petit point fixe solution de la variable $\mathcal{A}$, c'est à dire le langage associé à $\mathcal{A}$ est noté $\text{Ł}_D(\mathcal{A})$. La proposition suivante montre que les définitions inductives et les programmes logiques sont essentiellement les mêmes et les transformations ci-dessus préservent l'équivalence avec les langages de n-uplets générés.

Proposition 3.3.2. *Soit $\mathcal{D}$ une définition inductive et $\mathcal{P}$ un programme logique.*

(a) $\mathcal{L}_{\mathcal{D}}(A) = \mathcal{M}(\text{horn}(\mathcal{D}))|_{P_A}$ *pour chaque variable A du langage.*

(b) $\mathcal{M}(\mathcal{P})|_P = \mathcal{L}_{\text{indef}(\mathcal{P})}(A_P)$ *pour chaque symbole de prédicat P.*

Comme corollaire on obtient que les langages de n-uplets définissable par des définitions inductives sont exactement les langages de n-uplets récursivement énumérables.

Mais les langages de n-uplets récursivement énumérables n'ont beaucoup de bonnes propriétés notement le test du vide qui n'est pas décidable. Le langage est trop expressif, les auteurs ont donc restreints leurs études à des sous-classes qui ont un test du vide décidable. Une clause de programme $H \leftarrow B_1, \ldots, B_k$ est une *cs-clause* si $B_1, \ldots, B_k$ est linéaire et ne contient aucun symbole de fonction, i.e., si tous les arguments de B_i sont des variables n'apparaissant nul par ailleurs dans le corps . Une cs-clause est appelée

- *linéaire* si et seulement si sa tête est linéaire.
- *horizontale* si et seulement si pour deux variables apparaissant dans la tête qui sont arguments du même atome dans le corps apparaissent à la même profondeur dans la tête.
- *régulière* si et seulement si la tête est linéaire et pour une application $\pi\colon \omega \mapsto \omega$, chaque argument de la tête est de la forme $f(x_1, \ldots, x_{\text{ar}(f)})$, où si x_i apparaît dans l'atome du corps B_j tel que $\pi(j) = i$.
- *pseudo-régulière* si et seulement si pour des applications $\pi\colon \omega \mapsto \omega$, et $\pi'\colon V \mapsto \omega$ chaque argument de la tête est de la forme $f(x_1, \ldots, x_{\text{ar}(f)})$, où si x_i apparaît dans l'atome du corps B_j tel que $\pi(j) = i$, $\pi'(x_i) = i$ autrement.

Exemple 3.3.3.
$A(f(f(x,y),x)) \leftarrow B(a(x),x)$ *n'est pas une cs-clause car d'une part le corps n'est pas linéaire et d'autre part il contient un symbole de fonction a.*
$A(f(f(x,y),x)) \leftarrow B(x,y)$ *est une cs-clause mais qui n'est pas linéaire car la variable x apparaît deux fois au niveau de la tête de clause.*
$A(f(f(x,y),z)) \leftarrow B(x,z)$ *est une cs-clause linéaire mais elle n'est pas horizontale car x et z qui aparaissent dans le même atome du corps mais n'apparaissent jamais à la même hauteur dans la tête de clause.*
$A(f(x,y),f(y,x)) \leftarrow B(x,y)$ *est une cs-clause horizontale mais qui n'est pas pseudo-régulière car x apparaît à des positions différentes (1 et 2) dans des arguments différents de la tête de clauses.*
$A(f(x,y),f(x,z)) \leftarrow B(y,z)$ *est une cs-clause pseudo-régulière mais qui n'est pas régulière car la tête n'est pas linéaire.*
$A(f(x,y),f(z,u)) \leftarrow B(x,z)$ *est une cs-clause régulière.*

Un programme logique est un *cs-programme* si et seulement si toutes ces clauses sont des cs-clauses. Il est linéaire (horizontal, régulier, pseudo-régulier) si et seulement si toutes ses clauses le sont.

La proposition suivante montre que les sous classes de définitions inductives correspondent à leur contreparties en clauses logiques.

Proposition 3.3.4.

(a) Si $\mathcal{D}$ est un système de contraintes (linéaire, horizontal, régulier, ou pseudo-régulier) , alors horn($\mathcal{D}$) *est un cs-programme (linéaire, horizontal, régulier, ou pseudo-régulier).*

(b) Si $\mathcal{P}$ est un cs-programme (linéaire, horizontal, régulier, ou pseudo-régulier) alors indef($\mathcal{P}$) *est un système de contraintes (linéaire, horizontal, régulier, ou pseudo-régulier).*

3.4 Transformation de programmes logiques en cs-programmes

Dans cette section on voit une technique de transformation de programmes [64] qui se base sur des techniques bien connues de transformations de programmes [85, 94].

Une *définition* est une clause qui est une équivalence $H \leftrightarrow \mathcal{B}$ où H est un atome linéaire tel que ni H ni $\mathcal{B}$ ne contiennent de symbole de fonction. Un ensemble de définitions S est dit *compatible* avec un programme logique $\mathcal{P}$ si les symboles des têtes de prédicats des définitions sont toutes différentes et n'apparaissent pas dans $\mathcal{P}$.

La procédure prend comme entrée, un programme logique $\mathcal{P}$ et un ensemble de définitions I compatible avec $\mathcal{P}$. Quand elle termine, elle produit un cs-programme $\mathcal{P}'$ tel que pour chaque définition $p(\vec{X}) \leftrightarrow \mathcal{B}$ de I, $\mathcal{P}' \models p(\vec{X})\sigma$ si et seulement si il existe σ' s.t. $\sigma'\backslash_{\vec{X}} = \sigma$ et $\mathcal{P} \models \mathcal{B}\sigma'$.

Cette procédure est définie au moyen de deux règles qui transforment les 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 le programme logique d'entrée, $\mathcal{D}_{\text{done}}$ sont les définitions introduites par le processus et utilisées pour simplifier les clauses, $\mathcal{C}_{\text{new}}$ contient les définitions et les clauses à traiter, et $\mathcal{C}_{\text{out}}$ sont les cs-clauses générées par le processus.

On écrit $S \Rightarrow^U S'$ si S' est un état obtenu à partir de l'état S en appliquant *Dépliage* et $S \Rightarrow^D S'$ si il est obtenu par l'application d' *Introduction de Définition* définis plus bas. $\Rightarrow$ représente soit $\Rightarrow^U$ ou $\Rightarrow^D$. Une séquence $S_1 \Rightarrow S_2 \cdots \Rightarrow S_n$ est appelée une $\Rightarrow$-dérivation, elle peut être notée par $S_1 \stackrel{*}{\Rightarrow} S_n$.

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 les entrées de la dérivation, $\mathcal{P}'$ est sa sortie. Une dérivation est *complète* si son dernier état est final. Par la suite $\leftarrow ?$ représente soit $\leftarrow$ soit $\leftrightarrow$.

DÉPLIAGE.

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

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 que $H_i \leftarrow \mathcal{B}_i$ est une clause dans $\mathcal{P}$ pour $i = 1, \ldots, k$, et telles que l'unificateur le plus général μ de $\{A\}$ et $(H_1, \ldots, H_k)$ existe.

INTRODUCTION DE DÉFINITIONS. Prendre une clause non encore traitée, décomposer son corps en composants sans variables communes, et remplacer chaque composant qui n'est pas encore un simple atome linéaire sans symbole de fonction par un atome qui est soit dans l'ensemble des anciennes définitions s'il existe, sinon il est construit à partir d'un nouveau symbole de prédicat avec toutes les variables qu'il contient.

Pour chaque construction, on introduit une nouvelle définition associant le nouveau symbole de prédicat avec le composant remplacé. Formellement :

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

où $\mathcal{B}_1, \ldots, \mathcal{B}_k$ est une décomposition maximale de $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ en sous-ensemble sans variables communes,

$$L_i = \begin{cases} L\eta^{-1} & \text{si } (L \leftrightarrow \mathcal{B}_i)\eta \in \mathcal{D}_{\text{done}} \text{ pour un renommage} \\ & \text{de variable } \eta \\ & \text{et } \mathit{Var}(L\eta) = \mathit{Var}(\mathcal{B}_i\eta) \\ P_i(x_1, \ldots, x_n) & \text{autrement, } \{x_1, \ldots, x_n\} \text{ l'ensemble des vars. de } \mathcal{B}_i. \end{cases}$$

pour $1 \leq i \leq k$ et le nouveau symbole de prédicat P_i, et où $\mathcal{D}$ est l'ensemble de tout $L_i \leftrightarrow \mathcal{B}_i$ tel que L_i contient un nouveau symbole de prédicat.

La présentation des règles différent légèrement de celles habituellement présentées comme dans [84], Dépliage correspond au unfolding de règles et Introduction de Définition est une composition de Definition et Folding. Cette présentation est motivée par un but, celui de produire un cs-programme à partir de l'entrée, qui peut être vue commme une application spécialisée de la tupling strategy de [84].

Exemple 3.4.1. *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 soit $\mathcal{D} = \{R(x, y, z) \leftarrow M(x, y, z), E(y)\}$. Le prédicat M définit la soustraction, E la parité, et R définit tous les couples de nombres qui ont une différence paire. On notera que cette définition n'est pas une cs-clause, car les atomes du corps partagent des variables.

La figure 3.4 donne une dérivation complète avec comme entrée $\mathcal{P}$ et $\mathcal{D}$. On omet $\mathcal{D}_{\text{done}}$ car il contient seulement les définitions de $\mathcal{D}_{\text{new}}$. De plus,

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

FIG. 3.4 – Transformation du programme d'Example 3.4.1 en un cs-programme

on liste les définitions et clauses dans les colonnes $\mathcal{D}_{\text{new}}$ et $\mathcal{C}_{\text{out}}$ seulement dans l'étape qui les ajoute. Le dépliage sélectionne toujours l'atome le plus à gauche de ceux qui ont le terme de profondeur maximale. La troisième colonne, $\mathcal{C}_{\text{out}}$, liste les cs-clauses générées. La dernière colonne donne la règle appliquée : U_P signifie dépliage de l'atome le plus à gauche de profondeur maximale avec les clauses pour prédicat P, et D signifie l'introduction de définition appliquée à la première clause de $\mathcal{C}_{\text{new}}$.

Théorème 3.4.1.1 (Correction)**.** *Soit $\mathcal{P}$ un programme logique, $\mathcal{D}$ un ensemble de définitions compatible 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-programme avec la propriété que $\mathcal{M}(\mathcal{P}')|_P = \mathcal{M}(\mathcal{P} \cup \mathcal{D})|_P$ pour tous les symboles de prédicats P définis par $\mathcal{D}$.*

Comme attendu la transformation ne termine pas toujours. Toutefois on définit maintenant deux classes de programmes logiques qui ont un équivalent qui est un cs-programme. Ces deux classes permettent d'introduire des symboles de fonction dans le corps des clauses mais elles doivent rester linéaire.

Définition 3.4.2. *Une clause est* quasi-cs, *si le corps est linéaire et pour chaque variable qui apparait à la fois dans le corps et la tête, la profondeur de ses occurrences dans le corps est plus petite ou égale à la profondeur de toutes les occurences dans la tête. Un programme est quasi-cs si toutes ses clauses le sont.*

Exemple 3.4.3. *La clause* $P(f(s(x), y), g(x)) \leftarrow P(s(y), s(x))$ *est quasi-cs car les variables dans le corps,* x *et* y*, apparaissent à la profondeur* 1 *dans le corps et à la profondeur* 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.*

Théorème 3.4.3.1. *Soit* $\mathcal{P}$ *un quasi-cs programme, et soit* $\mathcal{D}_\mathcal{P}$ *l'ensemble de toutes les tautologies* $P(\vec{x}) \leftarrow P(\vec{x})$ *telles que* P *apparaît dans* $\mathcal{P}$*. Chaque* $\Rightarrow$*-dérivation avec comme entrée* $\mathcal{P}$ *et* $\mathcal{D}_\mathcal{P}$ *est fini.*

Par conséquent, chaque programme quasi-cs peut être effectivement transformé en un cs-programme équivalent. La preuve de ce théorème est donnée dans [63].

Une autre classe dont la transformation en cs-programme est finie est la classe des programmes instance-based.

Définition 3.4.4. *Une clause* $H \leftarrow \mathcal{B}$ *d'un programme logique* $\mathcal{P}$ *est* instance-based *(est une ib-clause) si* $\mathcal{B}$ *est linéaire et pour chaque paire d'atome* H', A *où* H' *est la tête d'une clause de* $\mathcal{P}$ *proprement renommée et* A *est un atome dans* $\mathcal{B}$ *tel que* H' *et* A *s'unifient,* H' *est une instance de* A*. Un programme est instance-based (est un ib-programme) si il contient seulement des ib-clauses.*

Exemple 3.4.5. *Considérons le programme*

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

La première clause est une ib-clause car la seule tête s'unifiant avec un atome du corps, $P(g(s(x)), f(g(y), z))$*, est une instance de l'atome du corps. La seconde clause n'est pas une ib-clause car la tête* $P(f(x, y), g(x))$ *s'unifie avec l'atome du corps* $P(f(s(y), x), z)$ *mais n'est pas une instance de celui-ci.*

Théorème 3.4.5.1. *Soit* $\mathcal{P}$ *un ib-programme, et soit* $\mathcal{D}_\mathcal{P}$ *l'ensemble de toutes les tautologies* $P(\vec{x}) \leftarrow P(\vec{x})$ *telles que* P *apparaît dans* $\mathcal{P}$*. Chaque* $\Rightarrow$*-dérivation avec comme entrée* $\mathcal{P}$ *et* $\mathcal{D}_\mathcal{P}$ *est fini.*

3.5 Propriétés des relations pseudo-régulières

Cette section présente des propriétés des relations pseudo-régulières qui ont été prouvées avant mon travail en thèse. La décidabilité des tests d'ap-

partenances, du vide ainsi que la clotûre par intersection des relations pseudo-régulières ont été montrées dans [63] en utilisant les transformations de programmes logique résumée plus haut.

Des propriétés des relations pseudo-régulières ont pu être prouvées grâce au résultat suivant [63].

Définition 3.5.1. *Une définition* $H \leftrightarrow B_1, \ldots, \mathcal{B}_k$ *des une* définition de jointure généralisée *si elle ne contient aucun symbole de fonction et* $Var(H) \subseteq Var(B_1, \ldots, B_k)$.

Théorème 3.5.1.1. *[63] Soit* $\mathcal{P}$ *un programme PR, et soit* D_p *une définition de jointure généralisée. N'importe quelle dérivation complète avec comme entrée* $\mathcal{P}$ *et* $\{D_p\}$ *qui déplie à chaque étape de dépliage tous les atomes silmutanément est finie et sa sortie est un programme PR.*

Corollaire 3.5.2. *[63] Les n-uplets de languages spécifiés par des programmes PR sont clos par intersections et jointure. Un programme PR représentant le résultat de l'opération peut être calculé via la* $\Rightarrow$*-dérivation.*

Exemple 3.5.3. *Considérons le programme PR* $\mathcal{P}$ *suitant,*
$\mathcal{P} = \{P_g(s(x), s(x)) \leftarrow\}$. *La définition de jointure généralisée* $I(x, y) \leftarrow P_g(x, y), Id_2(x, y)$ *définit l'intersection des relations définies respectivement par* P_g *et* Id_2. *La* $\Rightarrow$*-dérivation avec pour entrée un programme* $\mathcal{P}$ *et la définition de jointure précédente peut être résumée comme suit* $\mathcal{D}_{\text{done}}$ *est omis et la dernière colonne indique quelle règle a été appliquée :*

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	*rule*
$I(x, y) \leftarrow P_g(x, y),$ $Id_2(x, y)$			U
	$I(s(x), s(x)) \leftarrow Id_2(x, x)$		D
$I'(x) \leftarrow Id_2(x, x)$		$I(s(x), s(x)) \leftarrow I'(x)$	U
	$I'(0) \leftarrow$ $I'(s(x)) \leftarrow Id_2(x, x)$ $I'(c(x, y)) \leftarrow Id_2(x, x),$ $Id_2(y, y)$		D
		$I'(0) \leftarrow$ $I'(s(x)) \leftarrow I'(x)$ $I'(c(x, y)) \leftarrow I'(x), I'(y)$	

La sortie est le programme composé des clauses de $\mathcal{C}_{\text{out}}$.

3.6 Conclusion

L'étroite connection entre d'un côté les ensembles de contraintes et les automates d'arbres, et de l'autre côté les programmes logiques est assez naturelle et est utilisé fréquement dans un sens ou dans l'autre (voir par exemple [43, 90] ou [28]). Ici on s'est essentiellement concentré sur le traitement de langages de n-uplet d'arbres.

Toutefois dans ce chapitre les techniques présentées pour effectuer les tests ou le calcul des opérations ensemblistes sur les relations ne paraissent pas être les plus générales possibles. En effet d'autres classes de langages d'arbres comme par exemple [37] 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 jointure régulières existent. Le formalisme pour décrire ces grammaires est assez complexe et il serait donc intéressant d'étudier la classe des 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 mêmes opérations sur ces langages ou bien s'inspirer des méthodes utilisées par les auteurs de [83] qui ont montrés ces résultats sur une classe intermédiaire.

Chapitre 4

Réécriture

La réécriture est un formalisme permettant d'exprimer de manière très naturelle des fonctions ou des programmes aux moyens de règles de réécriture (elles permettent en fait de transformer un terme en un autre terme). Un système de réécriture (TRS pour Term rewriting System en anglais) est un ensemble de règles de réécriture. C'est une forme de modélisation majeure et fondamentale aux domaines comme la vérification de preuve, la vérification de système, ou la programmation logique fonctionnelle. Les propriétés les plus désirées dépendent de la capacité à calculer $R^*(E)$ (l'ensemble des termes atteignables à partir des éléments de E) au moyen des languages d'arbres qui ont un test du vide décidable et sont stables par intersection . Des auteurs comme [48, 91, 87] ont étudiés des classes de TRS qui préservent effectivement la reconnaissance, i.e. $R^*(E)$ est régulier si E est régulier.

La reconnaissance est habituellement préservée en encodant les dérivations de TRS comme des automates d'arbres et en exploitant les propriétées des classes de TRS respectant certaines contraintes. Une autre méthode encode d'abord la relation de réécriture au moyen de n-uplets de langage d'arbre [62, 70], i.e. ils calculent un langage d'arbre $\{ (t, t') \mid t \rightarrow_R^* t' \}$, et puis ils obtiennent la reconnaissance par projection.

Dans ce chapitre après avoir donné les définitions formelles on présentera l'encodage présenté dans l'article [64], qui présente une transformation de système de réecriture en clauses de Horn applicable à une certaine forme de réécriture. La transformation proposée a l'avantage de préserver des propriétés structurales essentielles de TRS. Ainsi cette transformation fait naturellement correspondre certaines classes de TRS à certaines classes de programme logique. Cette propriété permet ainsi de transposer des résultats d'un formalisme à l'autre.

4.1 Réécriture et réécriture conditionnelle

Dans cette section on va définir ce qu'est une règle de réécriture ainsi qu'un ensemble de règles de réécriture. Pour des détails concernant la réécriture on pourra consulter [9].

Définition 4.1.1. *Soit Σ une signature et X un ensemble fini dénombrable de variables disjointes de Σ. Une* règle de réécriture *est un couple $(s,t) \in T(\Sigma,X) \times T(\Sigma,X)$ avec s n'appartenant pas à X. Les règles de réécriture seront écrites $s \to t$. On appellera* s le coté gauche (lhs) et t le côté droit (rhs) *d'une règle de réécriture $s \to t$. Un* système de réécriture *(TRS) est un ensemble fini de règles de réécriture.*

Exemple 4.1.2. *Σ et X étant les mêmes que définis précédemment on a : $0(x)+0(y) \to 0(x+y)$ qui est une règle de réécriture avec le lhs qui est $0(x)+0(y)$ et le rhs qui est $0(x+y)$.*

Définition 4.1.3. *Soit R un TRS. La* relation de réduction $\to_R \subseteq T(\Sigma,X) \times T(\Sigma,V)$ *est définie par*
$s \to_R t$ si et seulement si $\exists (l,r) \in R, p \in Pos(s), \sigma \in Sub(T(\Sigma,X))$ tels que $s|_p = \sigma(l)$ et $t = s[\sigma(r)]_p$. 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* irréductible.

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 que $s \to^* s_1$ et $s \to^* s_2$, il existe un terme t tel que $s_1 \to^* t$ et $s_2 \to^* t$. 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 *étape de données* si σ telle que $s|_u = l\sigma$ est une *substitution de données* (substitution où le domaine d'arrivé est construit sur $\mathcal{T}(\mathcal{C},Var)$). Une dérivation de réécriture est une *dérivation de données* si toutes ses étapes sont des étapes de données. Les termes s et t sont dits *joignables de données* si il existe une substitution de données σ et un terme de données s' tels que $s\sigma \to_R^* s'$ et $t\sigma \to_R^* s'$ par des dérivations de données. Lorsque R est confluent on dit que s et t sont *R-unifiables de données* et que σ est un *R-unificateur de données*.

Exemple 4.1.4. *On présente ici le TRS de l'addition pour les entiers positfs représenté en nombre binaire.* $+$ *est le symbole pour l'addition et* $\oplus$ *le symbole pour l'addition avec retenue. Les règles avec* n_1 *ou* n_2 *représentent plusieurs règles où* n_1 *et* n_2 *peuvent être remplacées par* 0 *ou* 1*.*

$\bot + n_1(y) \to n_1(y)$ $\quad$ $n_1(y) + \bot \to n_1(y)$

$0(x) + 0(y) \to 0(x + y)$ $\quad$ $1(x) + 1(y) \to 0(x \oplus y)$

$n_1(x) + n_2(y) \to 1(x + y)$ *if* $n_1 \neq n_2$ $\quad$ $\bot + \bot \to \bot$

$s(\bot) \to 1(\bot)$ $\quad$ $s(0(x)) \to 1(x)$

$s(1(x)) \to 0(s(x))$

$\bot \oplus 0(y) \to 1(y)$ $\quad$ $\bot \oplus 1(y) \to 0(s(y))$

$0(y) \oplus \bot \to 1(y)$ $\quad$ $1(y) \oplus \bot \to 0(s(y))$

$0(x) \oplus 0(y) \to 1(x + y)$ $\quad$ $1(x) \oplus 1(y) \to 1(x \oplus y)$

$n_1(x) \oplus n_2(y) \to 0(x \oplus y)$ *if* $n_1 \neq n_2$ $\quad$ $\bot \oplus \bot \to 1(\bot)$

On a $0(1(\bot))+1(1(\bot)) \to_R^* 1(0(1(\bot)))$ *car on a la dérivation de données suivante :*

$0(1(\bot)) + 1(1(\bot)) \to_R 1(1(\bot) + 1(\bot)) \to_R 1(0(\bot \oplus \bot)) \to_R 1(0(1(\bot)))$.

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. On pourra remarquer dans l'exemple précédent que les différents symboles n'ont pas le même rôle $+, \oplus$ et s expriment des relations sur les données qui sont composées de $0, 1$ et $\bot$. C'est un type de TRS particulier qui sera définit ci-après.

Définition 4.1.5. *Soit* R *un TRS et* t *un terme. On dit que la position* $p \in \Sigma Pos(t)$ *est une* position de redex potentiel*, si il existe* $t\sigma$*, une instance close de* t *telle que* $t\sigma \to^* t'$ *sans jamais réécrire à des occurences plus petite 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 concrétement la notion de redex potentiel nous utiliserons des sur-approximations de cet ensemble. De manière générique on appelera ensemble de *position de redex possible*, noté $PRedPos_R(t)$, un ensemble tel que $Pot_R(t) \subseteq PRedPos_R(t) \subseteq \Sigma Pos(t)$.

Définition 4.1.6. *Soit* R *un TRS et* t *un terme de* $PRedPos(t)$ *un ensemble de positions de redex possibles 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* $Pos(C) = \{\, v \mid u.v \in Pos(t),$ $\nexists v'$ *tel que* $u < u.v' < u.v$ *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 maxiamle à laquelle x apparaît dans un redex possible de t. Le contexte C qui ne contient aucun redex possible et qui est tel que* $t = C[t_1, \ldots, t_n]$ *où* u_i *pour* $1 \leq i \leq n$ *est une position de redex possible, est appelé la* partie irréductible *de t et se note* $Irr_R(t)$. *Si u et v sont deux positions de redex possibles telles que* $u < v$ *alors v est dite* imbriquée

Exemple 4.1.7. *Soit 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 .

Définition 4.1.8. *Un* système de réécriture conditionnel *CTRS (pour conditionnal term rewriting système en anglais), est un ensemble fini de règles de réécritures de la forme* $l \to r \leftarrow \mathcal{B}$ *où $\mathcal{B}$ est une conjonction finie de conditions qui doivent être vérifiées avant la réécriture. Il y a une étape de réécriture,* $t \to_R s$ *si et seulement si il existe une rêgle conditionnelle* $l \to r \leftarrow \mathcal{B}$ *dans R, une position non-variable u dans t, et une substitution σ, telle que* $t|_u = l\sigma$ *et* $s = t[u \leftarrow r\sigma]$ *et $\mathcal{B}\sigma$ est vraie. La notion d'étape de donnée s'étend naturellement au cas conditionnel.*

On va maintenant avoir des définitions essentielles pour ma thèse car elles définissent de manière général les CTRS et dans quel cadre j'ai travaillé.

Définition 4.1.9. *Un* CTRS de jointures *est un CTRS où les conditions sont des paires de termes* $s \downarrow_R t$ *qui sont vérifiées pour une substitution σ si $s\sigma$ et $t\sigma$ sont R-joignable.*
Un CTRS basé sur les constructeurs *est un CTRS où le lhs des règles de réécritures sont de la forme* $f(t_1, \ldots, t_n)$ *où f est un symbole de fonction défini et chaque* t_i $(1 \leq i \leq n)$ *est un terme de donnée, l'exemple 4.1.4 en est un.*

Dans le contexte des CTRS basé sur les constructeurs, une solution de données *d'une équation de joignabilité* $s \downarrow^?_R t$ *est une substitution de donnée σ telle que* $s\sigma \to^*_R u$ *et* $t\sigma \to^*_R u$ *où u est un terme de données et toutes les étapes de réécritures sont des étapes de données.*
Pour un CTRS R et une fonction définie f, on note par $\tilde{f}$ la relation de donnée $\tilde{f} = \{ (t_1, \ldots, t_n, t) \in T^{n+1}(\mathcal{C}) \mid f(t_1, \ldots, t_n) \to^*_R t$ *avec seulement des étapes de données* $\}$.

Un terme t sera dit forme normale s'il ne contient aucune partie réductible. Un CTRS R sera dit confluent si pour chaque terme t il existe une unique forme normale t' tel que $t \to^*_R t'$.

Remarque 4.1.9.1. L'utilisation de la joignabilité de donnée est essentielle. On va le montrer en servant de l'Exemple 4.1.4, comme ce système est connu pour être confluent on va remplacer le $\downarrow_R^?$ par $=_R$. En effet sans cette distinction la formule $\exists x((0(\bot)+1(\bot) =_R y) \vee (\not(0(\bot)+1(\bot) =_R x)) \vee x =_R y)$ aurait des solutions positives. La seule solution pour y selon l'équation $(0(\bot) + 1(\bot) =_R y$ étant $1(\bot)$ mais par exemple la solution $x = 1(\bot + \bot)$ satisfait $(\not(0(\bot) + 1(\bot) =_R x))$, et pour ces deux valeurs $x =_R y$ est vraie.

4.2 La réécriture basique

On comprendra aisément que pour les systèmes de réécriture non confluent la stratégie de réécriture (choix de la partie du terme qui est réécrite) est cruciale, en effet selon les stratégies les termes obtenus peuvent être différents. Dans cette section on parlera d'une stratégie de réécriture particulière, car c'est celle sur laquelle se base l'encodage des relations de réécriture en programme logique de [64]. La réécriture basique est une stratégie de réécriture qui 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 distinction est faite a priori en séparant les symboles de données et les symboles de fonctions définie. La réécriture basique peut être vus comme un cas particulier de la surréduction basique [51]. Dans les dérivations de réécriture basique, les parties de termes qui ont été considérés 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 réécrire par la suite dans la dérivation. Une telle stratégie n'est pas complète mais elle l'est pour la celle des systèmes linéaires droits.

Définition 4.2.1. *Soient R un système de réécriture, s,t des termes et P, Q des ensembles de positions telles que $P \subseteq Pos(s)$ et $Q \subseteq Pos(t)$. On définit la réécriture sur les paires (s, P) de la manière suivante :*
$(s, P) \overset{bas}{\rightarrow}_{[u,l\rightarrow r]} (t, Q)$ *ssi* $s \rightarrow_{[u,l\rightarrow r]} t$
et $Q = \{ v \in P \mid u \not\leq v \} \cup u.\Sigma Pos(r)$
et $u \in P$
$(s, P) \overset{bas}{\rightarrow}_{[u,l\rightarrow r]} (t, Q)$ *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 appellé l'ensemble des* positions basiques *de s.*

Exemple 4.2.2. *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. 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 réécrire par la suite.

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

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

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

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

FIG. 4.1 – Convertion des règles de réécriture en clauses logique

Théorème 4.2.2.1. *[64] Soit R un système de réécriture linéaire droit alors $\to_R^*$ et $\overset{bas}{\to}$ définissent la même relation*

Dans mes travaux de thèse je me placerais uniquement dans le cas de la réécriture basée sur les constructeurs. Le lemme suivant est donc particulièrement important pour transposer les résultats de la transformation présentée dans la prochaine section.

Lemme 4.2.3. *Soit R un système de réécriture basée sur les constructeurs, toute dérivation de données $s \to_R^* t$ est une dérivation basique.*

4.3 Transformation d'un TRS en clauses de Horns

La figure 4.1 spécifie les règles pour transformer les termes et règles de réécriture en clauses de Horn. Pour un TRS R, soit $\mathcal{LP}(R)$ le programme logique contenant les clauses obtenues par application de la quatrième règle à toutes les règles de réécriture dans R. Pour un soucis de simplicité, on notera par x_u les nouvelles variables introduites dans la troisième règle pour le sous-terme $f(s_1, \ldots, s_n)$ à l'occurrence u d'un rhs s et A_u l'atome produit par cette règle.

Exemple 4.3.1. *Les règles de réécriture et clauses suivantes spécifient la*

multiplication et l'addition.

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

Définition 4.3.2. *Soit R un TRS on a $\overset{bas}{\to}^*$ qui est équivalent à $\to^*$ si tout $lhs \in R$ est linéaire.*

Soit $\mathcal{P}_Id$ l'ensemble des clauses $\{ P_f(\vec{x}, f(\vec{x})) \leftarrow | \ f \in \mathcal{F} \}$

Théorème 4.3.2.1. *[64] Soit R un TRS, s un terme tel que $s \leadsto \langle s', \mathcal{G} \rangle$. $s \overset{bas}{\to}^* t$ si et seulement si $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{G}\mu$ et $t = s'\mu$.*

Malheureusement la transformation de n'importe quel TRS ne mène pas à un cs-programme. Ceci est principalement due à la non linéarité des clauses ainsi qu'à leur profondeur ≥ 1. Cependant on présentera ci-dessous deux classes de TRS ib -TRS et quasi-cs-TRS définies danss [64] se transformant respectivement en ib-cs-programme et quasi-cs-programme.

Un quasi-cs-TRS est un TRS avec les propriétés suivantes :
- il est linéaire droit
- aucun rhs ne contient de redex potentiel imbriqué
- Pour $l \to r \in R$, x une variable d'un redex potentiel rp de r on a soit $x \notin Var(l)$ ou $|p|$ tel que $rp|_p = x$ est inférieur ou égale à la profondeur minimale à laquelle x apparait dans l.

Un ib-TRS est un TRS avec les propriétés suivantes :
- il est linéaire droit
- chaque rhs ne contient aucun redex potentiel imbriqué
- si C est un redex potentiel d'un rhs l un lhs alors l est une instance de C

4.4 Conclusion

Les résultats de la section précédente ont été utilisés pour des travaux concernant la préservation de la régularité. Grâce à des propriétés de jointure de langage régulier uet des langages définit par la transformation des quasi-TRS ou ib en cs-programme les auteurs de [64] ont montrés la préservation de la régularité pour les langages réguliers où chaque terme de ce langage contient au plus k redex potentiel imbriqués. Ce résultat reste toutefois incomparable à un certain nombre parmis lesquels [48, 78, 91] utilisant des techniques propres à la réécriture.

Le domaine de la réécriture est très riche. Je ne m'y étends pas plus en détails ici car pour y apporter des éléments nouveaux je me suis basé pour le moment uniquement sur la technique présentée par la figure 4.1.

Chapitre 5

Equivalences sémantiques et extensions syntaxiques des programmes logiques pseudo-réguliers

5.1 Introduction et rappel

Nous sommes intéressés par des langages de représentation finie dont le test du vide est décidable et qui sont clos par de nombreuses opérations ensemblistes. Malheureusement la plupart des langages de n-uplets d'arbres ne sont pas clos sous toutes les opérations ensemblistes. Une des classes satisfaisant les conditions de notre intérêt parmis les plus expressives sont les relations régulières [28]. Dans le chapitre 3 on a vu que les relations pseudo-régulières sont des languages de n-uplets d'arbres qui autorisent un certain type de duplications entre les composants du n-uplet des relations régulières. Dans [63], les tests du vide et d'appartenance aussi bien que les opérations d'intersection, projection, union et jointure des relations pseudo-régulières ont été décrites en termes de transformations de programmes logiques (les stabilité des relations pseudo-régulières par ces opérations ont ainsi été montrés), mais aucun résultat sur le complément des relations pseudo-régulière n'est donné. Dans ce chapitre, on complète cette lacune en donnant un algorithme qui calcule le complément des relations pseudo-régulières.

Un effet de bord de la définition de cet algorithme est la preuve que les relations pseudo-régulières et régulières définissent la même classe de langage. Malgrès cette équivalence, les relations pseudo-régulières (plus précisement les programmes logiques pseudo-réguliers) sont d'un grand intérêt car ils autorisent des formes de non linéarité qui compactent la définition des relations régulières

Par la suite on va assouplir encore les restrictions syntaxiques des relations pseudo-régulières par le biais principalement de la transformation $\Rightarrow$ définie dans le chapitre 3 qui va nous permettre de transformer un programme logique non pseudo-régulier en un programme logique pseudo-régulier équivalent. Ici le but de l'extension des classes par équivalence est d'affaiblir les restrictions syntaxiques des classes de programmes codant les pseudo-réguliers afin de faciliter par la suite l'utilisation de ces outils dans des domaines comme la réécriture pour élargir les classes des sytèmes de réécriture traités par exemple (voir chapitre 6 et 7). C'est à dire que l'on va définir des classes de programmes logiques et dans le chapitre 6 de CTRS qui par leur syntaxe définissent des relations régulières sans en avoir l'apparence. On va pour cela affaiblir encore les restrictions syntaxiques des langages réguliers, en autorisant des termes plus complexes dans les têtes de clause (voir section 5.2.2), et en autorisant de la non linéarité (section 5.2.1) et des symboles de fonctions dans le corps des clauses (section 5.3). Enfin dans la section 5.4 on verra une limite théorique à l'assouplissement des restrictions syntaxiques. Dans ce chapitre seules les sections 5.3 et 5.4 n'ont pas parus dans [60].

5.2 Représentation des languages de n-uplets d'arbres

On se focalisera initiallement sur des sous-classes de cs-programmes, respectivement les programmes réguliers et pseudo-réguliers vues dans le chapitre 3. On introduit ici de nouvelles défintions de classe de programme logique que l'on va étudier dans ce chapitre.

Définition 5.2.1. *Soit $H \leftarrow \mathcal{B}$ une clause telle que $\mathcal{B}$ ne contienne aucun symbole de fonction.*

- *$H \leftarrow \mathcal{B}$ est appellé* pseudo-régulier-like *(PR-like en abrégé) si et seulement si chaque argument de H est de la forme $f(x_1, \ldots, x_{\mathrm{ar}(f)})$ où f est un symbole de fonction et les x_is sont des variables deux à deux distinctes et il existe une application $\pi\colon Var \mapsto \mathbb{N}$ telle que $\pi(x_l) = l$ pour tout $l = 1, \ldots, \mathrm{ar}(f)$ et $\pi(x) = \pi(y)$ pour toutes les variables x et y apparaissant dans le même atome du corps.*
- *$H \leftarrow \mathcal{B}$ est appelé* pseudo-régulier *(PR en abrégé) si et seulement si c'est PR-like et $\mathcal{B}$ est linéaire.*
- *$H \leftarrow \mathcal{B}$ est appellé* régulier *(R en abrégé) si et seulement si c'est PR et H est linéaire*
- *$H \leftarrow \mathcal{B}$ est appelé* pseudo-régulier partagé *(PR-partagé en abrégé) si et seulement si c'est PR-like et ne contient aucune variable existentielle*
- *$H \leftarrow \mathcal{B}$ est appelé* régulier partagé *(R-partagé en abrégé) si et seulement c'est PR-partagé et H est linéaire.*

Un programme est PR-like, PR, R, PR-partagé or R-partagé si toutes ses clauses sont du type correspondant.

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

Exemple 5.2.2. *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 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 non plus 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 ce n'est ni PR car le corps est linéaire et ni PR-partagé car* z *est une variable existentielle.*

Le programme logique suivant est PR. Notons que les deux premières clauses sont R car leur tête sont linéaires

$P_f(s(x_1), s(x_2), s(x_3)) \leftarrow Q(x_1, x_2, x_3)$ $\qquad$ $P_f(0,0,0) \leftarrow$

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

$P_g(s(x), s(x)) \leftarrow$ $\qquad$ $Id_2(s(x), s(y)) \leftarrow Id_2(x, y)$ $\qquad$ $Id_2(0,0) \leftarrow$

$Id_2(c(x_1, y_1), c(x_2, y_2)) \leftarrow Id_2(x_1, x_2), Id_2(y_1, y_2)$

Pour une raison concernant l'élimination de variables existentielles on va présenter ici la nouvelle règle d'introduction de définition qui va remplacer celle définie section 3.4

INTRODUCTION DE DÉFINITIONS. Prendre une clause non encore traitée, décomposer son corps en composants sans variables communes, et remplacer chaque composant qui n'est pas encore un simple atome linéaire sans symbole de fonction par un atome qui est soit dans l'ensemble des anciennes définitions s'il existe, sinon il est construit à partir d'un nouveau symbole de prédicat avec les variables qu'il contient.

Pour chaque échec, on introduit une nouvelle définition associant le nouveau symbole de prédicat avec le composant remplacé. Formellement :

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

où $\mathcal{B}_1, \ldots, \mathcal{B}_k$ est une décomposition maximale de $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ en sous-ensemble sans variables communes,

$$L_i = \begin{cases} L\eta^{-1} & \text{si } (L{\leftrightarrow}\mathcal{B}_i)\eta \in \mathcal{D}_{\text{done}} \text{ pour un renommage} \\ & \text{de variable } \eta \\ & \text{et } Var(\mathcal{B}_i\eta) \cap Var(H\eta) = Var(L\eta) \\ P_i(x_1, \ldots, x_n) & \text{autrement, } \{x_1, \ldots, x_n\} = Var(\mathcal{B}_i) \cap Var(H) \\ & \text{l'ensemble des vars. de } \mathcal{B}_i. \end{cases}$$

pour $1 \leq i \leq k$ et le nouveau symbole de prédicat P_i, et où $\mathcal{D}$ est l'ensemble de tout $L_i{\leftrightarrow}\mathcal{B}_i$ tel que L_i contient un nouveau symbole de prédicat.

Le seul changement de cette règle par rapport à la précédente du même nom est que dans les nouvelles définitions les variables dans le prédicat doivent obligatoirement appartenir également à la tête de clause . Ce seul changement nous permettra de garder les mêmes théorèmes et les mêmes preuves que pour la transformation précédente.

Propriété 5.2.3. *Soit $\mathcal{P}$ un programme PR, et soit D_p une définition de jointure généralisée. N'importe quelle dérivation complète avec comme entrée $\mathcal{P}$ et $\{D_p\}$ qui déplie simultanément tous les atomes à chaque étape de dépliage est finie et sa sortie est un programme PR sans variable existentielle.*

Démonstration. Le théorème 3.5.1.1 implique que la dérivation complète est finie et que la sortie est un programme PR. L'étape d'introduction de définition impose que la sortie est constituée des clauses de la forme $H \leftarrow L_1, \ldots, L_k$ telle que L_i contient uniquement des variables de H. Donc aucune variable existentielle n'est introduite. □

5.2.1 Programmes pseudo-réguliers partagés

Dans cette section, on montre commment calculer le complément d'un langage pseudo-régulier en utilisant des techniques de transformations de programmes logiques. On utilisera pour ce faire la classe des programmes PR-partagé. Un programme de cette classe peut être transformé en un programme PR fini. De plus la tête des clauses de ce programme en entrée sont toutes linéaires. Le programme résultant est alors un programme R. Ce résultat permet de prouver l'équivalence entre la classe des langages générés par des relations pseudo-régulières et la classe des langages générés par les relations régulières.

Théorème 5.2.3.1. *Soit $\mathcal{P}$ un programme PR-partagé, et soit $\mathcal{D}_p$ l'ensemble de toutes les tautologies $P(x) \leftarrow P(x)$ telles que P apparait dans $\mathcal{P}$. Toute $\Rightarrow$-dérivation avec comme entrée $\mathcal{P}$ et $\mathcal{D}_\mathcal{P}$ est finie et sa sortie est un programme PR. Si le programme $\mathcal{P}$ en sortie est R-partagé alors le résultat est un programme R.*

Démonstration. La partie de la preuve concernant la production de PR-clauses est similaire à celle concernant la stabilité par intersection des languages pseudo-régulier de [63]. La preuve de terminaison est quant à elle basée sur le fait qu'on ne peut construire qu'un nombre fini de définitions de jointure généralisée dans $\mathcal{D}_{\text{new}}$ quand le nombre de variables dupliquées et le nombre maximal d'occurences d'une variable sont bornées pour l'ensemble de chaque définition de jointure construite. Dans notre cas cela provient directement de la limitation de l'arité des prédicats en tête de clause des définitions de jointure généralisés sans variable existentielle. Cette aritée étant bornée par la plus grande arité des prédicats du programme logique en entrée.

Dépliage. Soit $P(x_1, \ldots, x_n) \leftarrow A_1, \ldots, A_k$ une définition de jointure généralisée. Déplier cette définition en utilisant les clauses $H_i \leftarrow B_{i,1}, B_{i,2}, \ldots$ pour $1 \leq i \leq k$ donne la clause $P(x_1, \ldots, x_n)\mu \leftarrow \mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_m$ où μ est le plus petit unificateur général de $(A_1, \ldots, A_k)$ et $(H_1, \ldots, H_k)$ et $\mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_m$ est la décomposition du corps de la clause en composants sans variables communes.

Premièrement observons attentivement le problème d'unification impliquée à chaque étape de dépliage. Chaque H_i est de la forme $P_i(f_{i1}(\vec{y}_{i1}), f_{i2}(\vec{y}_{i2}), \ldots)$ où chaque $f_{ij}(\vec{y}_{ij})$ est un terme linéaire et chaque A_i est de la forme $P_i(\vec{x}_i)$. De plus, toutes les clauses étant renommées pour l'unification, $f_{ij}(\vec{y}_{ij})$ et $f_{i'j'}(\vec{y}_{i'j'})$ ne partagent aucune variable pour $i \neq i'$. Par conséquent le calcul de μ revient à unifier simultanément tous les couples $(x_{ij}, f_{ij}(\vec{y}_{ij}))$ où chaque $f_{ij}(\vec{y}_{ij})$ est un terme linéaire. Ce problème d'unification peut être résolu comme suit : Premièrement, pour chaque $x \in dup(A_1, \ldots, A_k)$, on a le problème d'unification $mgu_x(\{x_{ij} = f_{ij}(\vec{y}_{ij}) | x_{ij} = x\})$ où $dup(A_1, \ldots, A_k)$ dénote l'ensemble des variables apparaissant au moins deux fois dans $A_1, \ldots, A_k$. La décomposition de ces problèmes donne l'ensemble d'égalités $Eq_x = \{\vec{y}_{ij}|_l = \vec{y}_{i'j'}|_l | x_{ij} = x_{i'j'}\}$. Cela revient à calculer la solution du problème d'unification suivant $\cup_{x \in dup(A_1,\ldots,A_k)} Eq_x$. On connait maintenant que deux variables $\vec{y}_{ij}|_l$ et $\vec{y}_{i'j'}|_{l'}$ sont égales seulement si $i = i'$. Alors de la définition pseudo-régulière on a $\pi(\vec{y}_{ij}|_l) = \pi(\vec{y}_{i'j'}|_{l'}) = l = l'$. Par conséquent si $\vec{y}_{i_1 j_1}|_{l_1} = y_{i'_1 j'_1}|_{l_1}$ et $\vec{y}_{i_2 j_2}|_{l_2} = y_{i'_2 j'_2}|_{l_2}$ de $\cup_{x \in dup(A_1,\ldots,A_k)} Eq_x$ ont une variable en commun, $l_1 = l_2$.

A partir de là, on peut en déduire que si $x\mu$ est une variable pour $x \in Var(H_1 \ldots H_k)$, alors aucun symbole de fonction n'apparait dans $\mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_m$. On peut également voir que pour chaque variable $x \in Var(H_1, \ldots, H_k)$, $\pi(x) = \pi(x\mu)$ et $x\mu$ apparaissent à la position $\pi(x\mu)$ dans les arguments de $H_1\mu, \ldots, H_k\mu$. Ceci induit que toutes les variables de chaque $B_{ij}\mu$ qui apparait dans $H_i\mu$ ont la même image par π. Etant donné que l'on a $(A_1\mu, \ldots, A_k\mu) = (H_1\mu, \ldots, H_k\mu)$ et $Var(P(x_1, \ldots, x_n)) \subseteq Var(A_1, \ldots, A_k)$, on en déduit que $Var(P(x_1, \ldots, x_n)\mu) \subseteq Var(H_1\mu, \ldots, H_k\mu)$. Par conséquent toutes les va-

riables de chaque atome d'un composant $\mathcal{B}_l$ de $P(x_1,\ldots,x_n)\mu \leftarrow \mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_m$ qui apparaissent dans $P(x_1,\ldots,x_n)\mu$ ont la même image par π et cette valeur est la position dans les arguments de $P(x_1,\ldots,x_n)\mu$ où ils apparaissent. De plus les clauses du programme logique étant PR-partagées et $P(x_1,\ldots,x_n) \leftarrow A_1,\ldots,A_k$ étant une définition de jointure généralisée, elles ne contiennent aucune variable existentielle. Par conséquent, $P(x_1,\ldots,x_n) \leftarrow \mathcal{B}_1 \dot{\cup},\ldots,\dot{\cup},\mathcal{B}_m$ ne contient également aucune variable existentielle.

Introduction de définition Les nouvelles clauses ajoutées à $\mathcal{C}_{\text{out}}$ sont de la forme $P(x_1,\ldots,x_n)\mu \leftarrow L_1,\ldots,L_m$ où les variables de chaque L_i sont celles du composant correspondant $\mathcal{B}_i$ au niveau de l'étape de dépliage. Donc si une variable apparaît à la position p d'un argument de $P(x_1,\ldots,x_n)\mu$ elle a p comme image par π, et toutes les variables de chaque L_i ont la même image par π alors c'est une clause pseudo-regulière. De plus toutes les variables de chaque L_i appartiennent à $P(x_1,\ldots,x_n)\mu$ car $Var((\mathcal{B}_1,\ldots,\mathcal{B}_m)) \subseteq Var((H_1,\ldots,H_k))$. De la propriété pseudo-régulière, on a que chaque argument de $P(x_1,\ldots,x_n)\mu$ peut contenir au maximum une variable d'un L_i particulier, l'arité des prédicats L_i est donc bornée à n. Comme le nombre de symbole de prédicat dans $\mathcal{P}$ est fini et borné, le nombre d'ensemble d'atomes distincts (à rennomage de variables près) construits à partir de ces deux ensembles finis est fini.

□

Ce résultat a plusieurs conséquences intéressantes.

Corollaire 5.2.4. *Tout programme R-partagé (resp. PR-partagé) $\mathcal{P}$ peut être transformé en un programme R (resp. PR) fini et équivalent.*

Si l'ensemble d'entrée de définitions de jointure est l'ensemble des tautologies $P(\vec{x})\leftarrow P(\vec{x})$ telles que P apparait dans $\mathcal{P}$, alors la $\Rightarrow$-dérivation produit un programme R (or PR) qui définit le même pédicat que $\mathcal{P}$.

Corollaire 5.2.5. *Tout programme PR peut être transformé en un programme R fini et équivalent.*

Démonstration. Sans perte de généralité par la propriété 5.2.3 on peut considérer que les programmes PR ne contiennent aucune clause contenant des variables existentielles. Observons d'abord que toute clause PR $H \leftarrow \mathcal{B}$ est équivalente à la clause R-partagée $H' \leftarrow \mathcal{B}' \dot{\cup} Id$ où

- H' est un atome linéaire tel que $H = H'\sigma$ où σ est une substitution de Var à Var, $\mathcal{B} = \mathcal{B}'\sigma$ 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 des programmes R et définissent l'ensemble des i-uplet de termes identiques.

$H' \leftarrow \mathcal{B}' \dot{\cup} Id$ est R-partagé car il 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 apparaissant à la même position dans les arguments de H'. □

L'équivalence des classes de langages générés par les relations régulières et pseudo-régulières ne décroissent pas l'intérêt des programmes PR car ils sont syntaxiquement moins restrictifs que les programme R, comme les chapitres 6 et 7 le montreront.

Maintenant on va utiliser les programmes PR-partagé pour donner un algorithme qui va calculer le complément des languages de n-uplets d'arbres définis par un prédicat P d'un PR programme $\mathcal{P}$ utilisant la transformation $\Rightarrow$. Cet algorithme nous sera notement utile pour calculer les solutions des formules du chapitre 6 grâce aux transformations de programmes logiques. Il consiste en deux étapes. La première étape calcule à partir de $\mathcal{P}$ un nouveau programme R-partagé $\overline{\mathcal{P}}$ qui définit le complément de P. La seconde étape utilise le corollaire 5.2.4 pour obtenir un programme régulier équivalent à $\overline{\mathcal{P}}$.

Soit P un pédicat défini par le programme PR $\mathcal{P}$. Sans perte de généralité on peut considérer que $\mathcal{P}$ n'a ni clauses improductives (i.e. clauses qui ne contribuent pas au plus petit modèle de Herbrand), ni clauses avec des variables existentielles. Ensuite $\mathcal{P}$ est transformé en programme R-partagé $\mathcal{P}'$ par la technique utilisé dans la preuve du corollaire 5.2.5.

Pour chaque pédicat 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 $Q(f_1(\vec{x}_1), \ldots, f_n(\vec{x}_n))$ comme $\{Q(f_1(\vec{x}_1), \ldots, f_n(\vec{x}_n)) \leftarrow \mathcal{B} \in \mathcal{P}'\}$. On pourra remarquer que les têtes des clauses de $\mathcal{P}'$ étant linéaires toutes les têtes de clauses de $AC(Q, \vec{f})$ sont égales à renommage de variables près. Dans la suite on les considérera comme égale. Pour un atome $A = P(\vec{t})$, $\overline{A}$ dénotera $\overline{P}(\vec{t})$.

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

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

Exemple 5.2.6. *Considérons le programme logique composé uniquement des trois clauses suivantes* $P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(x_1, x_2)$, $P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(y_1, y_2)$ *et* $P(0, 0) \leftarrow$

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

Lemme 5.2.7. *Soit $\mathcal{P}$ un programme R-partagé sans aucune clause improductive. Le programme $\overline{\mathcal{P}}$ est l'ensemble des clauses $\bigcup_{P\in\mathcal{P},\vec{f}\in\mathcal{C}^n} \overline{AC}(P,\vec{f})$ et telle que $\forall P \in \mathcal{P}, \forall (t_1,\ldots,t_n) \in T^n_{\mathcal{C}}, \mathcal{P} \models P(t_1,\ldots,t_n)$ si et seulement si $\overline{\mathcal{P}} \models \overline{P}(t_1,\ldots,t_n)$.*

Démonstration. On montre que $\overline{\mathcal{P}}$ est obtenu par $\mathcal{P}$ en utilisant juste des équivalences logiques. Notons $H \leftarrow$ par $H \leftarrow vrai$. $\mathcal{P}$ est équivalent à $\{ H \leftarrow faux \bigvee_{H\leftarrow\mathcal{B}\in\mathcal{P}} \mathcal{B} \mid H = P(f_1(\vec{x}_1),\ldots,f_n(\vec{x}_n)), P \in \mathcal{P}, (f_1,\ldots,f_n) \in \mathcal{C}^n \}$ i.e. pour chaque atome possible $H = P(f_1(\vec{x}_1),\ldots,f_n(\vec{x}_n))$, on collecte l'ensemble des corps de clauses qui ont pour tête H. Ces corps sont connectés par l'opérateur logique ou. On remarquera que les têtes de clauses étant linéaires le symbole $\leftarrow$ ici est une équivalence $\leftrightarrow$. La négation de ces formules est $\{\overline{H} \leftrightarrow true \bigwedge_{H\leftarrow\mathcal{B}\in\mathcal{P}} \neg\mathcal{B}\}$. Si un $\mathcal{B}$ est la constante *vraie* alors $\overline{H}$ est équivalente à $faux$, et par conséquent aucune clause n'a pour tête $\overline{H}$ dans $\overline{\mathcal{P}}$. Si la définition est réduite à $H \leftarrow faux$ signifiant qu'aucune des instances clauses de H ne sont dans le modèle de $\mathcal{P}$ alors $\overline{H} \leftarrow$ est dans $\overline{\mathcal{P}}$, signifiant que toute les instances closes de $\overline{H}$ sont dans le modèle de $\overline{\mathcal{P}}$. Finallement, dans les autre cas, $true \bigwedge_{H\leftarrow\mathcal{B}\in\mathcal{P}} \neg\mathcal{B}$ est réduit a une disjonction de conjonction d'atomes négatifs. Chaque conjonction est de la forme $\neg A_1 \wedge \cdots \wedge \neg A_k$ où chaque A_i appartient à un différent $\mathcal{B}_i$. Ces définitions produisent les clauses $\{ \overline{Q}(f_1(\vec{x}_1),\ldots,f_n(\vec{x}_n)) \leftarrow \overline{A_1},\ldots,\overline{A_k} \mid AC(Q,(f_1,\ldots,f_n)) = \{H \leftarrow \mathcal{B}_1,\ldots,H \leftarrow \mathcal{B}_k\}$ and $\forall i \in [1,k] A_i \in \mathcal{B}_i \}$ de $\overline{\mathcal{P}}$. □

Lemme 5.2.8. *Soit $\mathcal{P}$ un programme R-partagé sans aucune clauses improductives. Le programme $\overline{\mathcal{P}}$ est un programme R-partagé.*

La preuve de ce lemme est évidente car aucune substitution à part des renomages de variables ont été utilisées pour définir $\overline{\mathcal{P}}$. Les applications utilisées pour vérifier la régularité partagée de $\mathcal{P}$ sont donc préservées dans $\overline{\mathcal{P}}$.

Théorème 5.2.8.1. *Les relations définies par des programmes-logiques pseudo-reguliers sont closes par complément.*

Démonstration. Soit $\mathcal{L}$ une relation pseudo-régulière définie par le programme $\mathcal{P}$. Premièrement calculons un programme régulier équivalent $\mathcal{P}'$ sans aucune clause improductive ni de variables existentielles dans le corps des clauses. $\overline{\mathcal{P}'}$ définit le complément de $\mathcal{L}$ dans $\mathcal{T}^N_{\mathcal{C}}$ si $\mathcal{L}$ est d'arité N. □

Pour deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ on a $\mathcal{L}_1 \setminus \mathcal{L}_2 = \overline{\mathcal{L}_2} \cap \mathcal{L}_1$ par conséquent on a le corollaire suivant.

Corollaire 5.2.9. *Soit $\mathcal{P}_1$ et $\mathcal{P}_2$ deux programmes PRs, $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages definis par les prédicats $P_{\mathcal{L}_1}$ de $\mathcal{P}_1$ et $P_{\mathcal{L}_2}$ de $\mathcal{P}_2$. Le programme*

qui représente le langage $\mathcal{L}_1 \setminus \mathcal{L}_2$ est calculé en deux phases. Premièrement calculer le programme $\overline{\mathcal{P}_2}$ et ensuite calculer $\mathcal{P}_1 \cap \overline{\mathcal{P}_2}$ a partir de $\mathcal{P}_1 \cup \overline{\mathcal{P}_2}$ et la définition $P_{\mathcal{L}_1 \setminus \mathcal{L}_2}(\vec{x}) \leftarrow P_{\mathcal{L}_1}(\vec{x}), P_{\overline{\mathcal{L}_2}}(\vec{x})$ en utilisant la $\Rightarrow$-dérivation.

On remarquera que l'entrée du second point est une définition de jointure généralisée.

5.2.2 Programmes logiques non-Greibach

Même si les programmes PR sont moins restrictifs que les réguliers, leur syntaxe reste toujours très stricte. Cette section a pour but de définir une classe de programmes logiques qui peuvent être transformés en PR. Notre but est ici d'autoriser plus d'un symbole de fonction dans la tête des clauses.

Définition 5.2.10. *Une clause de Horn est dite* non-Greibach *(NG en abrégé) si au moins un argument de la tête est de profondeur au moins un. Soit $H \leftarrow \mathcal{B}$ une clause telle que $\mathcal{B}$ ne contienne aucun symbole de fonction.*

- *$H \leftarrow \mathcal{B}$ est dite* NGPR-like *si et seulement si aucun des arguments de H ne sont des variables et il existe une application π: $Var \mapsto \mathbb{N}^+$ tel que $\pi(x) = u$ alors toutes les occurences de x dans les arguments de H sont à la position u et $\pi(x) = \pi(y)$ pour toutes les variables x et y apparaissant dans le même atome du corps.*
- *$H \leftarrow \mathcal{B}$ est dite* NGPR *si et seulement si c'est NGPR-like et $\mathcal{B}$ est linéaire.*
- *$H \leftarrow \mathcal{B}$ est dite* NGR *si et seulement si c'est NGPR avec H linéaire*
- *$H \leftarrow \mathcal{B}$ is dite* NG-PR-partagé *si et seulement si c'est NGPR-like avec aucune variable existentielle.*
- *$H \leftarrow \mathcal{B}$ is dite* NG-R-partagé *si et seulement si c'est NG-PR-partagé et H est linéaire.*

Un programme est NGPR-like, NGPR, R, NG-PR-partagé or NG-R-partagé si toutes ses 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 occurence 1.1 dans les arguments de la tête. (La tête étant linéaire c'est plus précisement NGR).

Lemme 5.2.11. *Toute NG-xxx clause $H \leftarrow B$ a un ensemble équivalent de xxx clauses (xxx étant soit PR ou PR-partagé ou R ou R-partagé).*

Démonstration. 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 des atomes dont les variables apparaissent à la profondeur un dans les arguments de la tête et chaque $\mathcal{B}_i$ est l'ensemble des atomes n'apparaissant pas dans $\mathcal{B}$ et dont les variables ont une image par π de la forme $i.u$.

Considérant 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 [1] autrement. 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 des clauses $\{P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n})) \leftarrow \mathcal{B}, P_1(\vec{y_1}), \ldots, P_k(\vec{y_k})\} \cup_{i \in [1,k]} P_i(\vec{y_i})\sigma \leftarrow \mathcal{B}_i$ où les P_is sont de nouveaux symboles de prédicats et $\vec{y_i}$ est le vecteur de variables de $\vec{x_1}, \ldots, \vec{x_n}$ tel que leur images par $\pi\sigma$ dans $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$ est de la forme $i.p$ et $p \neq \varepsilon$.

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 un tête de profondeur strictement inférieur à la tête de $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$. Par conséquent en itérant ce processus, on obtient un ensemble de clauses PR-like. On peut remarquer que

- Aucune variable existentielle n'est introduite par le processus alors 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-partagé, les clauses résultantes sont toutes NGPR-partagées
- Les nouvelles variables introduites par le processus sont toutes deux à deux différentes, par conséquent 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 alors les clauses résultantes sont PR et en plus si $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$ est linéaire alors les têtes des nouvelles clauses sont aussi linéaires.

□

Exemple 5.2.12. *Considérons la clause NGPR-partagée*

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

L'ensemble de clauses PR-partagées équivalent est

$$P_f(s(x), s(y), s(z)) \leftarrow P_1(x, y, z)$$
$$P_1(c(x_1, y_1), c(x_1, y_2), c(x_2, y_3)) \leftarrow P_f(y_1, y_1, y_3), P_g(x_1, x_2)$$

5.3 Programmes logiques Pseudo-régulier avec symboles de fonctions dans le corps

Cette section a le même but que la précédante à la différence près que la restriction syntaxique que l'on souhaite assouplir est celle concernant la présence de symboles de fonctions dans le corps des clauses. Le premier intérêt est de savoir que c'est possible, les inconvénients sont que ces extensions sont toujours restrictives, pas forcément facile a appréhender, et

[1] i.e. les nouvelles variables introduites sont deux à deux distinctes et également différentes de celles de $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$.

dans un cas nécessite l'introduction d'une nouvelle transformation de programmes. On montrera notament dans cette section que l'on peut assouplir les restrictions syntaxiques des clauses NGPR et NGPR-partagées en autorisant des termes clos dans le corps des clauses tout en restant dans la même classe de langage.

Comme présenté dans le chapitre 3 on sait qu'il existe notamment deux classes de programmes logiques ib-cs et quasi-cs tels que la tranformation appliquée à ces programmes logique donne pour résultat un cs-programme. Ici la démarche est la même, on a voulu chercher les classes de programmes logiques qui pouvaient par cette même transformation donner un programme logique pseudo-régulier. On a donc cherché les conditions nécessaires sur la forme des clauses et du programme logique permetant l'obtention de clauses pseudo-régulières. Les définitions suivantes permettront d'expliciter ces conditions.

Au premier abord on pourrait penser qu'il suffit de définir des clauses contraintes seulement par les positions des variables. Or le programme suivant nous montre que ce n'est pas si simple.

$H(s(s(s(x))), s(s(s(y))), s(s(s(z)))) \leftarrow B(g(f(x,y)), z).$

$B(g(x), g(y)) \leftarrow B(x,y).$

$B(0,0) \leftarrow$

L'étape de dépliage appliquée à la première clause donne :
$H(s(s(s(x))), s(s(s(y))), s(s(s(g(z))))) \leftarrow B(f(x,y), z)$, or x et y sont à la position 1.1.1 dans les arguments de la tête tandis que z est à la position 1.1.1.1 dans la tête. L'utilisation de la transformation nécessite donc de controler attentivement la position des variables les unes par rapport aux autres.

5.3.1 Programmes réguliers compatibles

Définition 5.3.1. *Deux variables x et $y \in B(t_1, \ldots, t_n)$ tel que $x \in t_i$ et $y \in t_j$ seront dites de positions imbriquées de différence q (où q est une position) si $t_i|_p = x$ et $t_j|_{p.q} = y$ avec $q \neq \varepsilon$. Une clause $H \leftarrow B_1 \ldots B_n$ avec $B_1 \ldots B_n$ linéaire sera dite pré-régulière compatible si chaque argument de H est de la forme $f(x_1, \ldots, x_{\mathrm{ar}(f)})$ où f est un symbole de fonction et il existe une application π: $Var \mapsto \mathbb{N}^+$ telle que $\pi(x) = u$ pour x à la position u d'un argument de la tête et $\pi(x) = \pi(y)$ pour toutes les variables x et y apparaissant dans le même atome du corps.*

Une clause $H \leftarrow B_1 \ldots B_n$ sera dite régulière compatible si elle est pré-régulière compatible et si pour tout unificateur μ de H' tête de clause de $\mathcal{P}$ et B_i on a :

– *si x et y deux variables de position imbriquées de différence q de B_i, si $u \in PosVar(y\mu)$ alors $q.u \notin PosVar(x\mu)$.*
– $\forall z \in Var(B_i)\ |z\mu| \geq 1$.

Un programme logique $\mathcal{P}$ est dit (pré-)régulier compatible si toutes ses clauses sont (pré-)régulières compatibles.

Exemple 5.3.2. *Soit l'atome suivant : $H(f(x, g(y)), h(g(z), f(t, 0)))$, dans l'atome précédent x et z sont de positions imbriquées de différence 1. Soit $\mathcal{P}$ le programme logique constitué des deux clauses suivantes :*

$$H(f(x_1, s(y_1)), g(x_2, s(y_2))) \leftarrow H(x_1, g(x_2, z)).$$

$$H(0, 1).$$

$\mathcal{P}$ est pré-régulier compatible mais n'est pas régulier compatible car si σ est l'unificateur du corps avec la tête on a d'une part $|\sigma x_2| = 0$ et d'autre part x_1 et x_2 de fifférence 1 et $PosVar(\sigma(x_1)) = \{1, 2.1\}$ et $PosVar(\sigma(x_2)) = \{\varepsilon\}$
Soit $\mathcal{P}$ le programme logique constitué des deux clauses suivantes :

$$H(f(x_1, s(y_1)), g(x_2, s(s(z)))) \leftarrow H(x_1, g(z_2, x_2)).$$

$$H(0, 1).$$

$\mathcal{P}$ est régulier compatible car les variables x_1 et x_2 d'une part sont de positions imbriquées de différence 2 et lorsque le corps s'unifie avec la tête on a $PosVar(\sigma(x_1)) = \{1, 2.1\}$ et $PosVar(\sigma(x_2)) = \{1.1\}$ et d'autre part ils apparaissent à la même position 1 dans les deux arguments de la tête.

Lemme 5.3.3. *Soit $\mathcal{P}$ un programme régulier compatible, et soit $\mathcal{D}_\mathcal{P}$ l'ensemble de toutes les tautologies $P(\vec{x})\leftarrow P(\vec{x})$ telles que P apparaît dans $\mathcal{P}$. N'importe quelle $\Rightarrow$-dérivation avec comme entrée $\mathcal{P}$ et $\mathcal{D}_\mathcal{P}$ donne pour résultat un cs-programme régulier.*

Afin de simplifier la démonstration de la preuve de ce lemme on introduit une nouvelle sorte de clause qui va être la classe présente dans les nouvelles définitions.

Définition 5.3.4. *Les définitions régulières compatibles sont de la forme : $H \leftarrow B_1$ où H est linéaire sans symbole de fonction et si $x, y \in B_i$ alors x et y sont réguliers compatibles et si $x, y \in H$ et i, j entiers naturels et u position alors $H|_{i.u} = x$ et $H|_{j.u} = y$.*

Démonstration. On va montrer que toutes les définitions que l'on va obtenir sont des définitions régulières compatibles et que toutes les clauses obtenues dans $\mathcal{C}_{\text{new}}$ sont NGR-compatibles. On notera que les tautologies sont de la forme voulue, et on montrera que l'étape de dépliage appliquée à une définition régulière compatible donne pour résultat des clauses régulières

avec des nouvelles définitions régulières compatibles. On remarquera d'abord que la décomposition en componsantes sans variables commune nous donne des nouvelles définitions dont le corps ne contient qu'un atome. Les variables de ces atomes étant à la même positions au niveau de la tête la nouvelle clause dans $\mathcal{C}_{\text{out}}$ est NGR.

Dépliage. Soit $L \leftarrow B_i$ une clause dans $\mathcal{D}_{new}$ à traiter. Soient x et y deux variables de B_i, et σ l'unificateur le plus général de B_i et H' tête de la clause $H' \leftarrow A_1 \dots A_n$ de $\mathcal{P}$. Par définition des clauses compatibles on a $\sigma(x) = t_x$ et $t_x \notin Var$, x est remplacée par t_x dans la tête de la clause créée au niveau de $\mathcal{C}_{\text{out}}$.

On distinguera maintenant trois cas : x et y sont à la même position dans des arguments différents de B_i ou bien ils sont de positions imbriquées, ou dans les autres cas. Les positions des variables $x_1 \in Var(\sigma(x))$ et $y_1 \in Var(\sigma(y))$ nous donnant les positions des variables dans μL

- x et y sont à la même position p dans des arguments différents de B_i. Dans ce cas si x_1 et y_1 sont à la même position $p.r$ dans H' alors ils sont à la même position r dans $\sigma(x)$ et $\sigma(y)$ respectivement.
- x et y sont de positions imbriquées de différence q. Dans ce cas x_1 et y_1 ne peuvent pas avoir la même position dans H' : la définition du programme régulier compatible contraignant que pour tout couple de variables dans $(\sigma(x), \sigma(y))$ leurs positions dans les arguments différents soient différentes de q.
- x et y sont deux variables ni de positions imbriquées ni à la même position. Dans ce cas x_1 et y_1 ne sont pas à la même position dans les arguments de H' (car $\sigma(x)$ et $\sigma(y)$ ne sont pas à des positions imbriquées et ne sont pas non plus aux mêmes positions dans des arguments différents).

Soit une clause $(L \leftarrow A_1 \dots A_n)\sigma$ de $\mathcal{C}_{\text{new}}$, comme $|z\sigma| \geq 1$ on a pour x et $y \in Var(A_i\sigma) \cap Var(L\sigma)$ $|\sigma^{-1}(x)| = x$ et $\sigma^{-1}(y) = y$ d'où x et $y \in A_i\sigma$ à la même position dans les arguments $\sigma(H')$ et donc à la même position dans $L\sigma$. Par conséquent $(L \leftarrow A_1 \dots A_n)\sigma$ est une nouvelle clause régulière compatible.

Introduction de définitions. On s'assure ici que toutes les définitions ne contiennent qu'un atome et que les atomes dans le corps des définitions sont compatibles. Soit une clause $(L \leftarrow A_1 \dots A_n)\sigma$ de $\mathcal{C}_{\text{new}}$, qui est régulière compatible. Le corps est linéaire donc les composantes sans variables communes ne contiennent qu'un seul atome. De plus chaque atome est compatible donc les définitions sont bien de la forme souhaitée.

Par induction toutes les clauses générées dans $\mathcal{C}_{\text{out}}$ sont NGR. □

La régularité est assurée mais pas la terminaison, on aura donc besoin d'autres restrictions.

5.3.2 Programmes ib-réguliers

Définition 5.3.5. *Une clause $H \leftarrow A_1 \dots A_m$ est ib-régulière dans un programme logique $\mathcal{P}$ si elle est ib-cs et régulière compatible. Un programme $\mathcal{P}$ est ib-régulier si toutes ses clauses sont ib-régulières.*

Lemme 5.3.6. *Soit $\mathcal{P}$ un programme ib-régulier, et soit $\mathcal{D}_\mathcal{P}$ l'ensemble de toutes les tautologies $P(\vec{x}) \leftarrow P(\vec{x})$ telles que P apparaît dans $\mathcal{P}$. N'importe quelle $\Rightarrow$-dérivation avec comme entrée $\mathcal{P}$ et $\mathcal{D}_\mathcal{P}$ est finie et donne pour résultat un cs-programme régulier.*

Démonstration. Un programme ib-régulier est par définition un ib programme la dérivation est donc finie. De plus comme ib-régulier est régulier compatible on a l'assurance que les clauses obtenues sont régulières. □

Définition 5.3.7. *Une clause $H \leftarrow A_1 \dots A_m$ est quasi-régulière dans un programme logique $\mathcal{P}$ si elle est quasi-cs et régulière compatible. Un programme $\mathcal{P}$ est quasi-régulier si toutes ses clauses sont quasi-régulières.*

C'est cette définition qui est utilisée pour étendre les NGPR au NGPR avec des termes clos dans le corps

Lemme 5.3.8. *Soit $\mathcal{P}$ un programme quasi-régulier, et soit $\mathcal{D}_\mathcal{P}$ l'ensemble de toutes les tautologies $P(\vec{x}) \leftarrow P(\vec{x})$ telles que P apparaît dans $\mathcal{P}$. N'importe quelle $\Rightarrow$-dérivation avec comme entrée $\mathcal{P}$ et $\mathcal{D}_\mathcal{P}$ est finie et donne pour résultat un cs-programme régulier.*

Démonstration. Un programme quasi-régulier est par définition un ib programme, la dérivation est donc finie. De plus comme quasi-régulier implique régulier compatible on a l'assurance que les clauses obtenues sont régulières. □

5.3.3 programmes NGPRC-partagés

Dans cette section on étudie le cas de l'extension des clauses NGPR-partagées, par l'addition de symboles de fonction dans le corps. Ce cas est plus complexe que le précédant car la duplication des variables dans les atomes du corps oblige à contrôler plus précisement les positions des occurences d'une même variable.

On définit maintenant une application qui permet de linéariser un atome, on en aura besoin pour définir les programmes logiques NGPRC-partagés. Cette application permet de linéariser les atomes.

Définition 5.3.9. *On définit une application $\psi : \mathcal{T}(P_r, \Sigma, \mathcal{X}) \mapsto \mathcal{T}(P_r, \Sigma, \mathcal{X})$ à rennomage de variable près tel que pour tout atome B $\psi(B) = B_l$ tel que B_l est linéaire et subsume B et soit σ l'unificateur le plus général de B et B_l alors tout $x \in Var(B_l)$ $\sigma x = y$.*

Exemple 5.3.10. $\psi(P(f(x,x,y))) = P(f(x_1,x_2,x_3))$

Définition 5.3.11. *Une clause $H \leftarrow B_1 \dots B_n$ d'un programme logique $\mathcal{P}$ sera dite pré-NGPRC-partagé une clause sans variable existentielle de $\mathcal{P}$ telle que chaque argument de H est de la forme $f(x_1,\dots,x_{\mathrm{ar}(f)})$ où f est un symbole de fonction et il existe une application $\pi\colon Var \mapsto \mathbb{N}^+$ telle que $\pi(x) = u$ pour x à la position u d'un argument de la tête et $\pi(x) = \pi(y)$ pour toutes les variables x et y apparaissant dans le même atome du corps.*

Une clause $H \leftarrow B_1 \dots B_n$ est pseudo-régulière non greibach partagé compatible (NPRC-partagé pour en abrégé) si elle est pré-NGPRC-partagée et si μ l'unificateur le plus général de H' tête de clause de $\mathcal{P}$ et $\psi(B_i)$ existe on a :

- *si x et y sont deux variables de position imbriquées de différence q de $\psi(B_i)$, et si $u \in PosVar(y\mu)$ alors $q.u \notin PosVar(x\mu)$.*
- *$\forall z \in Var(\psi(B_i))\ |z\mu| \geq 1$.*

Un programme logique $\mathcal{P}$ est dit (pré-)NGPRC-partagé si toutes ses clauses sont (pré-)NGPRC-partagées.

Exemple 5.3.12. *Soit $\mathcal{P}$ le programme logique constitué des deux clauses suivantes :*

$$H(f(x_1,s(y_1)),g(x_2,s(y_2))) \leftarrow H(x_1,g(x_1,x_2)).$$

$$H(0,1).$$

$\mathcal{P}$ est pré-NGPRC-partagé mais il n'est pas NGPRC-partagé car on a $\psi(H(x_1,g(x_1,x_2))) = H(y_1,g(y_3,y_2))$ et y_1 et y_3 sont de positions imbriquées de différence 1 et lorsque σ est l'unificateur le plus général de $H(y_1,g(y_3,y_2)))$ et $H(f(x_1,s(y_1)),g(x_2,s(y_2)))$ on a $PosVar(\sigma(y_1)) = \{1,2.1\}$ et $PosVar(\sigma(x_2)) = \{\varepsilon\}$. Soit $\mathcal{P}'$ le programme logique constitué des deux clauses suivantes :

$$H(f(x_1,s(y_1)),g(x_2,s(s(z)))) \leftarrow H(x_1,g(0,x_1)).$$

$$H(0,1).$$

$\mathcal{P}'$ est NGPRC-partagé.

Maintenant on a besoin de définir une nouvelle transformation de programme afin de pouvoir transformer les programmes logiques NGPRC-partagés en programmes NGPR-partagé. Cette transformation s'inspire fortement celle définie au niveau du chapitre 3.

L'idée est de profiter d'une part que les variables peuvent être dupliquées dans un même atome du corps et d'autre part que les variables peuvent être dans différents atomes. Ainsi on va définir une nouvelle règle d'introduction de définition où chaque atome du corps de clause va former une

nouvelle composante. De plus pour tous ces atomes, chaque occurence de chaque variable sera prise en compte pour définir le nouveau prédicat. La transformation consiste en 2 règles, *dépliage* (celle de la $\Rightarrow$-dérivation) et *introduction de définition atomique*, qui transforment les programmes logiques NGPR-partagé compatible en NPGPR-partagé.

Par commodité on conservera les états de la transformation $\Rightarrow$-dérivation qui ont le même rôle.

On écrit $S \Rightarrow_p S'$ si S' est un état obtenu d'un état S par une application de la règle de *dépliage* ou de la règle *introduction atomique de définition* (voir plus bas la description). La cloture réflexive et transitive de $\Rightarrow_p$ est notée par $\overset{*}{\Rightarrow}_p$. Un *état initial* est de la forme $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle$ où $\mathcal{D}$ est linéaire 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 appelées les entrées d'une dérivation, $\mathcal{P}'$ est la sortie. Une dérivation est appelée *complète* si son dernier état est un état final.

INTRODUCTION DE DÉFINITIONS ATOMIQUES. Prendre une clause non encore traitée et remplacer chaque atome qui n'est pas encore sans symbole de fonction par un atome qui est soit dans l'ensemble des anciennes définitions s'il existe, sinon il est construit à partir d'un nouveau symbole de prédicat avec les variables qu'il contient.

Pour chaque construction, on introduit une nouvelle définition associant le nouveau symbole de prédicat avec le composant remplacé. Formellement :

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}} \dot{\cup} \{H \leftarrow B_1 \dot{\cup} \cdots \dot{\cup} B_k\},\, \mathcal{C}_{\text{out}} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}} \cup \mathcal{D},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \cup \{H \leftarrow \{L_1, \ldots, L_k\}\} \rangle}$$

$$L_i = \begin{cases} L & \text{si } \{L \leftarrow B_i\}\eta^{-1} \in \mathcal{D}_{\text{done}} \text{ avec } \{L \leftarrow B_i\}\eta^{-1} \\ & \text{linéaire et } L \text{ sans symbole de fonction pour une} \\ & \text{substitution de variables } \eta \\ P_i(\vec{x}) & \text{autrement, } \vec{x} \text{ variables de } B_i. \end{cases}$$

pour $1 \leq i \leq k$ et le nouveau symbole de prédicat P_i, et où $\mathcal{D}$ est l'ensemble de tout $L_i \leftarrow B_i$ tel que L_i contient un nouveau symbole de prédicat.

Exemple 5.3.13. *Soit $\mathcal{P}$ le programme logique NGPRC-partagé*

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

et soit l'ensemble de définition initiale $\mathcal{D} = \{R(x) \leftarrow A(x, x)\}$. Le prédicat E définit la la parité, et A définit tous les couples de nombres distincts de différence paire. R récupère tous couples de nombres identiques définis par A. On remarquera d'abord que l'ensemble de définitions initial n'est pas

linéaire, mais dans ce cas la procédure termine. On notera également que le programme n'est pas NGPR-partagé car un corps de clause contient des symboles de fonctions.

La figure 3.4 donne une dérivation complète comme entrée $\mathcal{P}$ et $\mathcal{D}$. On omet $\mathcal{D}_{\text{done}}$ car il contient seulement les définitions de $\mathcal{D}_{\text{new}}$. De plus, on liste les définitions et clauses dans les colonnes $\mathcal{C}_{\text{new}}$ et $\mathcal{C}_{\text{out}}$ seulement dans l'étape qui les ajoute. Le dépliage sélectionne toujours l'atome le plus à gauche de ceux qui ont le terme de profondeur maximale. La troisième colonne, $\mathcal{C}_{\text{out}}$, liste les clauses NGPRC-partagées générées. La dernière colonne donne la règle appliquée : D signifie dépliage pour la première clause de $\mathcal{D}_{\text{new}}$, et A signifie l'introduction atomique appliquée à la première clause de $\mathcal{C}_{\text{new}}$.

Théorème 5.3.13.1. *[Correction] Soit $\mathcal{P}$ un programme logique, $\mathcal{D}$ un ensemble de définitions compatibles avec $\mathcal{P}$. Si $\langle \mathcal{P}, \mathcal{P}, \emptyset, \emptyset, \emptyset \rangle \stackrel{*}{\Rightarrow} \langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, alors $\mathcal{P}'$ est un programme logique tel que $\mathcal{M}(\mathcal{P}')|_P = \mathcal{M}(\mathcal{P} \cup \mathcal{D})|_P$ pour tous les symboles de prédicats P définis par $\mathcal{P}$.*

L'équivalence sémantique des programmes $\mathcal{P} \cup \mathcal{D}$ et $\mathcal{P}'$ par rapport aux symboles de prédicat introduits par $\mathcal{D}$ provient de la correction des règles de dépliage et d'introduction de définition en général.

Lemme 5.3.14. *Soit $\mathcal{P}$ un programme logique NGPRC-partagé, et soit $\mathcal{D}_{\mathcal{P}}$ l'ensemble de toutes les tautologies $P(\vec{x}) \leftarrow P(\vec{x})$ telles que P apparaît dans $\mathcal{P}$. N'importe quelle $\Rightarrow_p$-dérivation avec comme entrée $\mathcal{P}$ et $\mathcal{D}_{\mathcal{P}}$ donne pour résultat un programme NGPR-partagé.*

Démonstration. La preuve est très similaire à la preuve du lemme 5.3.3 en notant que le cas de la variable dupliquée dans un même atome du corps est résolu car les définitions atomiques sont linéaires. On observera également qu'aucune variable existentielle n'est introduite au niveau des nouvelles clauses créees. On va montrer que toutes les définitions que l'on va obtenir sont des définitions régulières compatibles et que toutes les clauses obtenues dans $\mathcal{C}_{\text{new}}$ sont NGPRC-partagées. On notera que les tautologies sont de la forme voulue, et on montrera que l'étape de dépliage appliquée à une définition régulière compatible donne pour résultat des clauses NGPR-partagées. On remarquera d'abord que l'étape introduction atomique ne donne que des définitions linéaires et dont le corps ne contient qu'un atome. Les variables de $L_i\eta$ étant à la même position au niveau de la tête (car ce sont les variables de B_i) la nouvelle clause dans $\mathcal{C}_{\text{out}}$ est NGPR-partagée.
*Dépliage.*Soit $L \leftarrow B_i$ une clause dans $\mathcal{D}_{new}$ à traiter. Soient x et y deux variables de B_i, et σ l'unificateur le plus général de B_i et H' tête de la clause $H' \leftarrow A_1 \dots A_n$ de $\mathcal{P}$. Par définition des clauses compatibles on a $\sigma(x) = t_x$ et $t_x \notin Var$, x est remplacée par t_x dans la tête de la clause créée au niveau de $\mathcal{C}_{\text{out}}$. On distinguera maintenant trois cas : x et y à la même position dans des arguments différents de B_i ou a des positions imbriquées, ou dans

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$R(x) \leftarrow A(x,x)$			D
	$R(s(x)) \leftarrow A(x,x)$ $E(x)E(x)$ $R(s(x)) \leftarrow A(x,x)$ $E(s(x))E(s(x))$		A
$L_1(x,y) \leftarrow A(x,y)$	$R(s(x)) \leftarrow A(x,x)$ $E(s(x))E(s(x))$	$R(s(x)) \leftarrow L_1(x,x)$ $E(x)E(x)$	A
$L_1(x,y) \leftarrow A(x,y)$ $L_2(x) \leftarrow E(s(x))$		$R(s(x)) \leftarrow L_1(x,x)$ $L_2(x)L_2(x)$	D
$L_2(x) \leftarrow E(s(x))$	$L_1(s(x),a) \leftarrow$ $L_1(a,s(x)) \leftarrow$ $L_1(s(x),s(y)) \leftarrow A(x,y)$ $E(x)E(y)$ $L_1(s(x),s(y)) \leftarrow A(x,x)$ $E(s(x))E(s(y))$		A
$L_2(x) \leftarrow E(s(x))$	$L_1(a,s(x)) \leftarrow$ $L_1(s(x),s(y)) \leftarrow A(x,y)$ $E(x)E(y)$ $L_1(s(x),s(y)) \leftarrow A(x,x)$ $E(s(x))E(s(y))$	$L_1(s(x),a) \leftarrow$	A
$L_2(x) \leftarrow E(s(x))$	$L_1(s(x),s(y)) \leftarrow A(x,y)$ $E(x)E(y)$ $L_1(s(x),s(y)) \leftarrow A(x,x)$ $E(s(x))E(s(y))$	$L_1(a,s(x)) \leftarrow$	A
$L_2(x) \leftarrow E(s(x))$	$L_1(s(x),s(y)) \leftarrow A(x,x)$ $E(s(x))E(s(y))$	$L_1(s(x),s(y)) \leftarrow A(x,y)$ $E(x)E(y)$	A
$L_2(x) \leftarrow E(s(x))$		$L_1(s(x),s(y)) \leftarrow A(x,y)$ $L_2(x)L_2(y)$	D
	$L_2(s(x)) \leftarrow E(x)$		A
		$L_2(s(x)) \leftarrow E(x)$	

FIG. 5.1 – Transformation du programme de l'exemple 5.3.13 en un programme NGPR-partagé

les autres cas. Les positions des variables $x_1 \in Var(\sigma(x))$ et $y_1 \in Var(\sigma(y))$ nous donnant les positions des variables dans μL

- x et y sont à la même position p dans des arguments différents de B_i. Dans ce cas si $x_1 \in Var(\sigma(x))$ et $y_1 \in Var(\sigma(y))$ sont à la même position $p.r$ dans H' alors ils sont à la même position r dans $\sigma(x)$ et $\sigma(y)$ respectivement.
- x et y sont de positions imbriquées de différence q alors x_1 et y_1 ne peuvent pas avoir la même position dans H' la définition du programme régulier compatible contraignant que pour tout couple de variables dans $(\sigma(x), \sigma(y))$ leurs positions dans les arguments différents soient différentes de q.
- x et y deux variables ni de positions imbriquées et ni à la même position. Dans ce cas si $x_1 \in Var(\sigma(x))$ et $y_1 \in Var(\sigma(y))$ alors x_1 et y_1 ne sont pas à la même position dans les arguments de H' (car $\sigma(x)$ et $\sigma(y)$ ne sont pas à des positions imbriquées et ne sont pas non plus aux même positions dans des arguments différents).

Au niveau de la clause construite $(L \leftarrow A_1 \dots A_n)\sigma$ dans $\mathcal{C}_{\text{new}}$, comme pour tout z de B_i $|z\sigma| \geq 1$ on a pour x et $y \in Var(A_i)$, $|\sigma(x)| = 0$ et $|\sigma(y)| = 0$ (les variables de A_i appartenant à H') d'où σx et $\sigma y \in A_i\sigma$ qui sont à la même position dans les arguments $\sigma(H')$ et donc à la même position dans $L\sigma$. Par conséquent $(L \leftarrow A_1 \dots A_n)\sigma$ est une nouvelle clause NGPRC-partagée. Par induction toutes les clauses générées dans $\mathcal{C}_{\text{out}}$ sont NGPR-partagée.

Introduction de définitions. Par définition toutes les définitions ne contiennent qu'un atome dans le corps. Il nous reste a voir que les atomes dans le corps des définitions sont compatibles. Soit une clause $(L \leftarrow A_1 \dots A_n)\sigma$ de $\mathcal{C}_{\text{new}}$, qui est NGPRC-partagée, régulière compatible. Comme l'atome dans le corps de la définition est une transformation de $A_i\sigma$ en un atome linéaire correspondant le corps est linéaire. Donc les blocs d'atome de variables communes ne contiennent qu'un seul atome. De plus chaque atome est compatible donc par induction les définitions sont bien de la forme souhaitée.

□

La régularité est assurée mais pas la terminaison, on aura donc besoin d'autres restrictions.

Pour cela on va s'inspirer des résultats présentés dans le chapitre 3 concernant les programmes logiques qui se transforment en cs-programmes.

Définition 5.3.15. *Une clause $H \leftarrow A_1 \dots A_m$ est quasi-NGPRC-partagée si elle est NGPRC-partagée et que chaque variable qui apparaît à la fois dans la tête et le corps de 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 $\mathcal{P}$ est quasi-NGPRC-partagé si toutes ses clauses le sont.*

On remarquera que cette définition inclus l'extension des NGPR-partagés en autorisant des termes clos au niveau des arguments d'atome corps. De telles clauses, c'est à dire les clauses quasi-NGPRC-partagé, où toutes les variables du corps sont arguments d'atomes du corps seront appellées par abus de langage RP-closes.

Exemple 5.3.16. $H(f(x,y_1),f(x,y_2)) \leftarrow H(x,f(0,f(0,0)))H(y_2,y_1)$ *est une*
clause quasi-NGPRC-partagée. De plus comme toutes les variables dans le corps sont arguments d'atomes c'est une clause RP-close.

Chaque programme quasi-NGPRC-partagé peut se transformer en un programme NGPR-partagé.

Lemme 5.3.17. *Soit $\mathcal{P}$ un programme quasi-NGPRC-partagé, 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 $\Rightarrow_p$-dérivation complète avec comme entrée $\mathcal{P}$ et $\mathcal{D}_\mathcal{P}$ qui est finie.*

Démonstration. Soit τ la profondeur maximale d'un atome de $\mathcal{P}$ on montre qu'en utilisant uniquement Dépliage et Introduction de définition atomique on a :

- Les corps des clauses de $\mathcal{C}_{\text{new}}$ sont partagée compatibles et ont une profondeur bornée par τ et
- Les corps de définitions de $\mathcal{D}_{\text{new}}$ sont constitués d'un seul atome bornée par τ

Ceci implique qu'on ne peut générer qu'un nombre fini de définitions différentes à un rennomage 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.

Dépliage. Considérons une définition $L \leftrightarrow A$ de $\mathcal{D}_{\text{new}}$ et une clause quasi-NGPRC-partagée $H \leftarrow \mathcal{B}$ de $\mathcal{P}$ telles que A et H sont unifiables par le mgu μ. Soit $x \in Var(H)$ telle que $|x\mu| > 0$ ce qui par définition 5.3.11 de la compatibilité ne peut arriver que lorsque $x\mu$ est un terme clos (car $\mathcal{B}$ ne contient pas de variables existentielles). Par propriété de l'unification par l'unifcateur le plus général $|H'\mu| = |A\mu| \leq \tau$. De plus toujours par propriétés de l'unification et par la définition 5.3.15 (qui empêche qu'une variable de A soit unifiée à une variable de H) à rennomage de variable près les seules variables de $\mathcal{B}\mu$ qui sont instanciées sont celles où $|x\mu| > 0$. Par définition 3.4.2 les variables y de $\mathcal{B}$ sont à une hauteur inférieure ou égale à celle au niveau de la tête de de clause d'où pour tout $B_i \in \mathcal{B}$ $|B_i\mu| \leq \tau$.

Introduction atomique. L'introduction atomique n'introduit que des nouvelles définitions dont chaque corps n'est qu'un atome linéaire. Les nouvelles définitions dans $\mathcal{D}_{\text{new}}$ sont toutes de la forme $L(\vec{x}) \leftarrow B\eta^{-1}$ avec pour toutes les variables x de B $x\eta = y$ et $y \in Var$, d'où $|B\eta^{-1}| \leq \tau$.

Par induction seul un nombre fini de nouvelles définitions peuvent être générées. □

5.4 A propos des programmes logique PR-like

Au début de ce chapitre on a vu l'équivalence sémantique des programmes logique R avec les programmes logiques PR, R-shared, PR-shared etc. Le but de cette section est d'étudier le cas général des programmes logiques PR-like. A cet effet je définirai les programmes horizontaux-like (qui subsument les programmes PR-like) et je montrerai par une procédure de transformation de programme l'équivalence sémantique des programmes logiques définis (ce sont les programmes logique généraux que l'on utilise dans ce mémoire) avec les programmes logiques horizontal-like. Ce résultat me permettra de transformer un codage du problème de correpondance de Post sous forme de programme logique en un programme PR-like, ce qui démontre la non équivalence des classes de langages générés par les programmes logique R et PR-like.

Définition 5.4.1. *Une clause $H \leftarrow \mathcal{B}$ est dite horizontale-like si $\mathcal{B}$ est sans symbole de fonction et tout les arguments de H sont de la forme $f(\vec{x})$ Un programme logique sera dit horizontal-like si toutes ses clauses le sont.*

Exemple 5.4.2. *$H(f(x,y),x) \leftarrow$ n'est pas horizontale-like car le deuxième argument de la tête est une variable. $H(f(x,y),g(z,x)) \leftarrow H(z,x)H(y,z)$ est une clause horizontale-like.*

5.4.1 Equivalence sémantique des programme logiques définis et des programmes horizontaux-like

Cette sous section présente 2 règles, *Génération* et *introduction de faits*, qui transforment les programmes logiques en programmes horizontal-like équivalents. Je montrerai que le processus de transformation termine.

Les règles de transformations ont pour états $\langle \mathcal{P}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}}, \mathcal{G}_{\text{new}}, \mathcal{G}_{\text{done}} \rangle$ où $\mathcal{P}$ est un programme logique qui reste inchangé, $\mathcal{C}_{\text{new}}$ sont les clauses non encore applatie, $\mathcal{C}_{\text{out}}$ sont les clauses déjà traitées mais encore utilisées pour la simplification de clauses, $\mathcal{G}_{\text{new}}$ sont les clauses générées à partir des définitions par dépliage, et $\mathcal{G}_{\text{done}}$ sont les clauses horizontales-like déjà générées. Un ensemble de définitions, $\mathcal{D}$, est *compatible avec* $\mathcal{P}$, si tous les symboles de prédicats apparaissant dans la tête des définitions apparaissent exactement dans une tête et nul part ailleurs dans $\mathcal{D}$ et $\mathcal{P}$; la seule exception sont les définitions tautologique de la forme $P(\vec{x}) \leftarrow P(\vec{x})$ où P peut apparaître sans restrictions dans $\mathcal{D}$ et $\mathcal{P}$. Les symboles de prédicats dans la tête de $\mathcal{D}$ sont appelés les *symboles de prédicats définis par* $\mathcal{D}$. Les définitions tautologiques sont utiles pour déclencher les transformations des

clauses définissant P sans avoir à introduire de prédicat. L'alternative serait de remplacer la tautologie par $P'(\vec{x}) \leftarrow P(\vec{x})$ pour un nouveau symbole de prédicat P'.

On écrit $S \Rightarrow_{hl} S'$ si S' est un état obtenu d'un état S par une application de la règle de *génération* ou de la règle *introduction de définition* (voir plus bas leur description). La clôture réflexive et transitive de $\Rightarrow_{hl}$ est notée par $\stackrel{*}{\Rightarrow}_{hl}$. Un *état initial* est de la forme $\langle \mathcal{P}, \emptyset, \emptyset, \mathcal{D}, \emptyset \rangle$ où $\mathcal{D}$ est compatible avec $\mathcal{P}$; un *état final* est de la forme $\langle \mathcal{P}, \emptyset, \mathcal{P}', \emptyset, \mathcal{D}' \rangle$. $\mathcal{P}$ et $\mathcal{D}$ sont appelées les entrées d'une dérivation, $\mathcal{P}'$ est la sortie. Une dérivation est appelée *complète* si son dernier état est un état final. Pout tout $f \in \Sigma$ on rajoute au programme initial les clauses linéaires $All(f(\vec{x}))$..

GÉNÉRATION. Prendre une définition non encore traitée, ajouter les atomes unaire sans symbole de fonction de dépliages $All(x)$ dans le corps si x est à la position ε d'un argument , et les déplier avec toutes les clauses correspondantes du programme initial. Formellement :

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

où $\{A_1, \ldots, A_k\}$ est l'ensemble des atomes $All(x)$ tels que $\exists i, L|_i = x \in Var$. $\mathcal{C}$ est l'ensemble de toutes les clauses $(L \leftarrow \mathcal{R})\mu$, telles que H_i est une tête de clause dans $\mathcal{P}$ pour $i = 1, \ldots, k$, si l'unificateur le plus général μ de $(A_1, \ldots, A_k)$ et $(H_1, \ldots, H_k)$ existe. Sinon $\mathcal{C} = setL \leftarrow \mathcal{R}$. On notera que les clauses de $\mathcal{P}$ sont renommées de telles sorte qu'elles ne partagent de variables ente elles ni avec $L \leftarrow \mathcal{R} \cup \{A_1, \ldots, A_k\}$.

INTRODUCTION DE FAITS. Prendre une clause non encore traitée, démembrer la tête et le corps en remplaçant chaque terme non variable t_i à la hauteur 1 dans les arguments de la tête et à hauteur 0 dans les différents atomes du corps par une nouvelle variable x_i et introduire pour chacun un atome qui est soit dans l'ensemble des anciennes définitions si il existe et sinon il est construit à partir d'un nouveau symbole de prédicat et des variables x_i et $\vec{y} \in Var(t_i)$.

Pour chaque construction on introduit un nouveau fait associant le terme t_i remplacé et les variables contenues dans t_i.

Formellement :

$$\frac{\langle \mathcal{P},\, \mathcal{G}_{\text{new}},\, \mathcal{G}_{\text{done}},\, \mathcal{C}_{\text{new}} \,\dot\cup\, \{(H \leftarrow B_1, \ldots, B_k)\mu\},\, \mathcal{C}_{\text{out}} \rangle}{\langle \mathcal{P},\, \mathcal{G}_{\text{new}} \cup \mathcal{D},\, \mathcal{G}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \cup \{H \leftarrow B_1, \ldots, B_k, L_1, \ldots, L_m\} \rangle}$$

où $H \leftarrow B_1, \ldots, B_k, L_1, \ldots, L_m$ est une clause horizontale-like telle que si $B_i|_j = u_{ij}$ ou si $H|_{i.j} = u_{ij}$, et $u_{ij}\mu = t_{ij}$ non variable alors il existe L_l tel que $Var(t_{ij}) \cup \{u_{ij}\} = Var(L_l)$ et $|Var(t_j) \cup \{u_{ij}\}| = |Var(L_l)|$.

$$L_i = \begin{cases} L\eta & \text{si } (L\mu \leftarrow) \in \mathcal{D}_{\text{done}} \text{ pour un renomage. de variable } \eta \\ P_i j(u_{ij}, x_1, \ldots, x_n) & \text{autrement, } \{x_1, \ldots, x_n\} \text{ les variables de } t_i j. \end{cases}$$

pour $1 \leq i \leq k$ et les nouveaux symboles de prédicats $P_i j$, et où $\mathcal{D}$ est l'ensemble de toutes les clauses $L_i\mu \leftarrow$ telles que L_i contient un nouveau symbole de prédicat.

Exemple 5.4.3. *On reprend le cs-programme $\mathcal{P}$ de l'exemple 3.4.1.*

$$\begin{array}{ll} Plus(0, x, x) \leftarrow & E(a) \leftarrow \\ Plus(s(x), y, s(z)) \leftarrow Plus(x, y, z) & E(s(s(x))) \leftarrow E(x) \end{array}$$

et soit $\mathcal{D} = \{R(x, y, z) \leftarrow M(x, y, z), E(y)\}$.

La figure 5.2 donne une dérivation complète comme entrées $\mathcal{P}$ et $\mathcal{D}$. On omet $\mathcal{D}_{\text{done}}$ car il contient seulement les définitions de $\mathcal{D}_{\text{new}}$. De plus, on liste les définitions et clauses dans les colonnes $\mathcal{G}_{\text{new}}$ et $\mathcal{C}_{\text{out}}$ seulement dans l'étape qui les ajoute. La troisième colonne, $\mathcal{C}_{\text{out}}$, liste les clauses horizontales-like générées. La dernière colonne donne la règle appliquée : G signifie génération pour la première clause de $\mathcal{G}_{\text{new}}$, et F signifie l'introduction de faits appliquée à la première clause de $\mathcal{C}_{\text{new}}$.

Théorème 5.4.3.1. *Soit $\mathcal{P}$ un programme logique, $\mathcal{D}$ un ensemble de définitions compatibles avec $\mathcal{P}$. Si $\langle \mathcal{P}, \mathcal{P}, \emptyset, \emptyset, \emptyset \rangle \stackrel{*}{\Rightarrow}_h l\langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, alors $\mathcal{P}'$ est un programme logique horizontale-like fini tel que $\mathcal{M}(\mathcal{P}')|_P = \mathcal{M}(\mathcal{P} \cup \mathcal{D})|_P$ pour tous les symboles de prédicats P définis par $\mathcal{P}$.*

Démonstration. Cette preuve se fera en trois temps. On prouvera d'abord que les clauses générées sont du type voulues. Ensuite on regardera l'équivalence sémantique des programmes. Enfin on donnera un argument de terminaison.

$\mathcal{P}'$ est un programme horizontale-like par construction.
On doit avoir les variables des têtes de clauses de $\mathcal{C}_{\text{new}}$ à une profondeur au moins 1 dans les arguments car l'étape d'introduction d'introduction de fait oblige que les clauses de $Cnew$ soient des instances de clauses horizontales-like.
Génération. Tous les arguments des têtes de clauses dans $\mathcal{C}_{\text{new}}$ sont non variables. En effet on effectue pour toutes les variables x aux positions ε dans les arguments des têtes de clause un dépliage des $All(x)$. Soit σ l'unificateur le plus général de $All(x)$ avec une tête de clause de $\mathcal{P}$ alors $|\sigma x| = 1$.
Introduction de faits. Par définition toutes les clauses dans $\mathcal{C}_{\text{out}}$ sont des clauses horizontales-like. Par induction toutes les clauses de $\mathcal{C}_{\text{out}}$ sont horizontales-like.

$\mathcal{G}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$Plus(0,x,x) \leftarrow$ $Plus(s(x),y,s(z)) \leftarrow$ $\quad Plus(x,y,z)$			G
$Plus(s(x),y,s(z)) \leftarrow$ $\quad Plus(x,y,z)$	$Plus(0,0,0) \leftarrow$ $Plus(0,s(x),s(x)) \leftarrow$		F
$Plus(s(x),y,s(z)) \leftarrow$ $\quad Plus(x,y,z)$	$Plus(0,s(x),s(x)) \leftarrow$	$Plus(0,0,0) \leftarrow$	F
$Plus(s(x),y,s(z)) \leftarrow$ $\quad Plus(x,y,z)$		$Plus(0,s(x),s(x)) \leftarrow$	G
	$Plus(s(x),0,s(z)) \leftarrow$ $\quad Plus(x,0,z)$ $Plus(s(x),s(y),s(z)) \leftarrow$ $\quad Plus(x,s(y),z)$		F
$L(0) \leftarrow$	$Plus(s(x),s(y),s(z)) \leftarrow$ $\quad Plus(x,s(y),z)$	$Plus(s(x),0,s(z)) \leftarrow$ $\quad Plus(x,y,z)L_0(y)$	F
$L_0(0) \leftarrow$ $L_1(s(y),y) \leftarrow$		$Plus(s(x),s(y),s(z)) \leftarrow$ $\quad Plus(x,u,z),L_1(u,y)$	G
$L_1(s(y),y) \leftarrow$	$L_0(0) \leftarrow$		F
$L_1(s(y),y) \leftarrow$		$L_0(0) \leftarrow$	G
	$L_1(s(0),0) \leftarrow$ $L_1(s(s(y)),s(y)) \leftarrow$		F
	$L_1(s(s(y)),s(y)) \leftarrow$	$L_1(s(x),0) \leftarrow$ $\quad L_0(x).$	F
		$L_1(s(x),s(y)) \leftarrow$ $\quad L_1(x,y).$	

FIG. 5.2 – Transformation du programme d'Exemple 5.4.3 en un programme horizontal-like

L'équivalence sémantique des programmes $\mathcal{P}$ et $\mathcal{P}'$.
Génération Premièrement l'introduction des atomes $All(x)$ dans le corps des clauses conserve la sémantique des programmes. Deuxièmement par correction des règles de dépliage la sémantique est équivalente .
Introduction de faits Concernant l'équivalence sémantique de l'étape introduction de faits il suffit d'effectuer un étape de dépliage de la clause nouvellement introduite dans $H \leftarrow B_1, \ldots, \mathcal{B}_k, L_1, \ldots, L_m$ en dépliant par rapport aux $L_1, \ldots, L_m$ des têtes de clauses $H_1 \ldots H_k$ dans $\mathcal{G}_{\text{new}}$ pour obtenir la clause initiale de $\mathcal{C}_{\text{new}}$ $(H \leftarrow B_1, \ldots, \mathcal{B}_k)\mu$ (car l'unificateur le plus général des L_i et H_i est μ) . Par correction des règles de dépliages la sémantique est alors équivalente.

On s'occupe maintenant de l'argument de terminaison de la procédure. Le nombre de clauses du programme $\mathcal{P}'$ dépend du nombre de clauses qui ont été introduite dans $\mathcal{C}_{\text{new}}$, celles-ci contiennent initiallement $\mathcal{P}$ et ensuite consistent en des faits dont la hauteur maximale τ d'un argument est celle d'un argument d'un atome du programme originel $\mathcal{P}$ plus un. Ceci à cause de l'étape de génération au départ qui augmente potentiellement la hauteur des atomes du corps de 1 au maximum.
Génération. Augmente la hauteur des atomes d'au plus 1. En effet les têtes de clauses avec lesquels on procède l'unification sont de profondeurs 1. On remarquera encore par la définition de l'étape de génération que lorsque la clause dans $\mathcal{G}_{\text{new}}$ est un fait, les clauses obtenues dans $\mathcal{C}_{\text{new}}$ sont égalements des faits.
Introduction de faits. On va montrer par induction que la hauteur maximale d'un argument dans une nouvelle définition est $\tau + 1$. On ne considère maintenant que les nouvelles définitions dans $\mathcal{G}_{\text{new}}$. Par définition de l'étape d'introduction de définition on a que toutes ces nouvelles définitions sont des faits. Les nouvelles définitions dans $\mathcal{D}_{\text{new}}$ sont des faits $H \leftarrow$ où aucun argument de H n'est variable. Les nouveaux atomes dans $\mathcal{G}_{\text{new}}$ sont donc des faits dont les arguments de la tête ont une profondeur au plus τ, donc des têtes de clauses dont la hauteur maximale est $\tau + 1$. Par induction la hauteur maximale d'un argument d'une clause dans $\mathcal{G}_{\text{new}}$ est τ. Le nombre de faits dont la hauteur est bornée étant fini par induction on a un nombre fini de clauses dans $\mathcal{P}'$ □

5.4.2 Test du vide indécidable pour les programmes logique PR-like

Le but initial a été de prouver la non-équivalence sémantique des programmes logique R et des programmes logiques PR-like. On montre plus que cela, on montre l'indécidabilité du test du vide pour les programmes PR-like, pour cela on va modéliser toute instance du *problème de correspondance de Post* réputé indécidable sous la forme d'un programme logique PR-like.

Définition 5.4.4. *Le problème de correspondance de Post [36]*
Soient A et C deux alphabets disjoints finis. Soient ϕ et ϕ' deux morhismes de A^* dans C^*. Le problème de correspondance de Post pour ϕ et ϕ' consiste à trouver un mot non vide $\alpha \in A^+$ tel que $\phi(\alpha) = \phi'(\alpha)$

Théorème 5.4.4.1. *Il n'existe pas d'algorithme uniforme pour résoudre le problème de correspondance de Post. Le problème reste indécidable lorsque ϕ et ϕ' sont injectives.*

Nous allons utiliser les conventions suivantes pour exprimer le problème de correspondance de Post dans le cadre de la programmation logique. Etant donné $A = \{a_1, \ldots, a_n\}$ et $C = \{c_1, \ldots, c_n\}$ deux alphabets finis disjoints, à chaque lettre $a_i \in A$ (resp. $c_i \in C$) est associé le symbole d'arité 1 a_i (resp. c_i). Le symbole $\perp$ représente la fin de mot. Par exemple mot $\alpha\beta\gamma$ sera représenté par le terme $\alpha(\beta(\gamma(\perp)))$. Par convention si l'on pose $\omega = \alpha\beta\gamma$, la notation $\omega(\perp) = \alpha(\beta(\gamma(\perp)))$. Pour représenter les préfixes, nous allons utiliser des termes non clos. Par exemple le préfixe $\alpha\beta\gamma$ sera représenté par le terme $\alpha(\beta(\gamma(x)))$ où x est une variable.

Afin de prouver la non équivalence sémantique des programmes logique R et des programmes logiques PR-like, je vais transformer ce problème en un programme logique PR-like.

Pour tout $a_i \in A$ on définit les clauses $Post(\omega_i(x), \omega'_i(y)) \leftarrow Post(x, y)$ et $Post(\omega_i(x), \omega'_i(y)) \leftarrow$ où $\omega_i = \phi(a_i)$ et $\omega'_i = \phi'(a_i)$.

Le problème de correspondance de Post se modélise alors par la clause $PCP \leftarrow Post(x, y), Id(x, y)$ avec les clauses $\{Id(f(x), f(y)) \leftarrow Id(x, y)\} \cup \{Id(\perp, \perp)\}$ si $f \in C$.

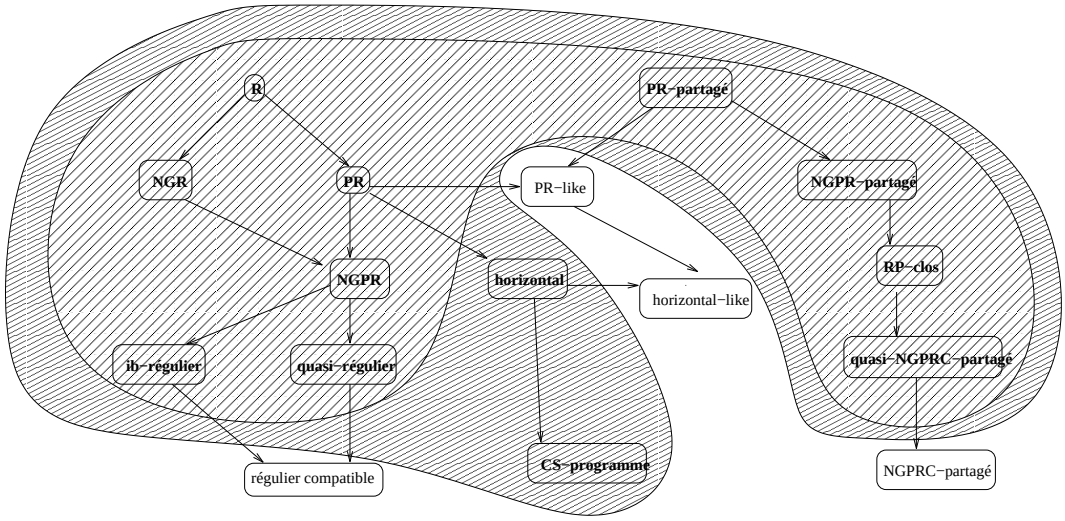
L'ensemble des clauses $\{Id(f(x), f(y)) \leftarrow Id(x, y)\} \cup \{Id(\perp, \perp)\}$ si $f \in C$ forme un programme logique régulier.

L'ensemble des clauses
$\{PCP \leftarrow Post(x, y), Id(x, y)\} \cup \{Post(\omega_i(x), \omega'_i(y)) \leftarrow Post(x, y)\} \cup \{Post \leftarrow Post(x, y), Id(x, y)\} \cup \{Id(f(x), f(y)) \leftarrow Id(x, y)\} \cup \{Id(\perp, \perp)\}$ peut d'après le Théorème 5.4.3.1 être transformé en un programme horizontale-like équivalent et plus précisément PR-like car les symboles de fonctions sont au plus d'arité 1.

,

5.5 Conclusion

Dans ce chapitre j'ai défini plusieurs classes de programmes logiques qui reconnaissent les relations régulières par différents assouplissements syntaxiques tout en montrant une limite théorique. Ces différentes classes sont présentées dans la figure 5.3.



signifie la classe des programmes logiques A est plus souple syntaxiquement que la classe des programmes logiques B

la partie hachurée la plus interne représente les langages de n−uplet réguliers, et l'externe les langages synchronisés.

FIG. 5.3 – Diagramme d'inclusion syntaxique et d'expressivté

Toutefois il serait sans doute possible d'autoriser des extensions aux clauses de 3.2 en autorisant des clauses de la formes $P(\vec{x}) \leftarrow \mathcal{B}$ où $\mathcal{B}$ est sans symbole de fonction, ou encore des clauses de la forme $P(\vec{x}) \leftarrow Q(\vec{t})$ en contraignant les variables de $\vec{t}$, tout en restant un programme exprimant une relation régulière. De manière similaire on pourrait utiliser des techniques différentes de celles utilisées dans ce chapitre comme celle montrées section 3.2 afin d'assouplir encore les restrictions syntaxiques.

Dans ce chapitre j'ai également représenté un algorithme qui calcule le complément d'une relation régulière au moyen de programme logique. Mais d'autres propriétés peuvent être désirées, comme par exemple le fait d'être clos par clôture transitives. C'est toutefois une propriété très forte et qui demanderait des restrictions supplémentaires. C'est le cas de la classe de de langage des ground tree transducers [33]. On pourrait alors étudier 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 de cs-programmes réguliers. On pourrait peut être ainsi étendre les résultats obtenus dans [33] ou dans un autre domaine ceux sur la réécriture préfixe sur les chaînes [57].

Chapitre 6

Résolution des formules pseudo-régulières du premier ordre

6.1 Introduction

Une formule du premier ordre peut être vue vue comme un ensemble d'équations dont les termes sont du premiers ordres et les variables sont quantifiées universellement ou existentiellement. Elles sont étudiées depuis longtemps. Dans le cas où les variables ne peuvent prendre qu'un nombre fini de valeur elles sont décidables et le problème consiste alors à trouver la résolution la plus rapide pour certains types de problèmes. Par contre leur résolutions dans le domaine infini qui nous intéresse est de manière générale indécidable.

On s'intéresse ici à trouver une représentation finie décidable des solutions des formules du premier ordre où le seul prédicat est la R-joignabilité $\downarrow_R^?$ où R est un CTRS. La R-unification (qui est équivalent à la joignabilité lorsque le système de réécriture est confluent et les termes de l'équation ne partagent pas de variables) est utilisée par exemple dans des langages de programmation logico-fonctionnels comme Cury [7] ou Babel [56] ou pour faire de la vérification de protocoles cryptographiques [52, 75, 76]. Résoudre une simple équation $s \downarrow_R^? t$ est connue pour être indécidable sans de fortes restrictions sur le TRS [39, 62]. Des résultats positifs ont été montrés en restreignant le TRS [71] pour obtenir un problème équationnel fini (i.e. assurer que l'ensemble des solutions minimales est fini). Par exemple les théories shallow ou standard [29, 82] imposent que la profondeur de toutes les variables au niveau du TRS (non-orienté) est un (en fait les propriétés des théories standards affaiblissent un peu cette restriction en autorisant les variables à une profondeur plus grande que un quand elle ne sont pas sous un symbole de fonction définie). Pour les théories infinies (les théories où

l'ensemble des solutions d'une équation $s \downarrow^?_R t$ peut être infinis) plusieurs résultats ont été donnés pour décider de l'existence de solutions en utilisant le problème d'atteignabilité (e.g. [87, 93, 77]) mais très peu d'entre eux [62, 66] donnent une représentation finie des solutions et encore moins vont au délà de la résolution d'une simple équation.

Afin d'atteindre notre but, nous avons d'abord à résoudre le problème de la représentation d'un ensemble potentiellement infini de solutions d'une équation $s \downarrow^?_R t$. L'outil le plus commun pour une telle représentation est celle des tree languages recognizers [28] tels que les automates d'arbres ou grammaire d'arbre. Dans notre contexte, un langage d'arbre est l'ensemble des termes clos représenté par un programme logique comme dans par exemple [63, 81].

Pour les raisons de bonnes propriétés de clôture notre choix s'est porté sur les relations régulières. La représentation finie des ensembles infinis de solutions étant choisie, on donne un algorithme pour calculer une représentation des solutions de l'équation $s \downarrow^?_R t$. On utilisera pour cela une extension de la méthode décrite dans la section 4.3, où un système de réécriture est transformé en un programme logique qui code la clôture transitive de la relation de réécriture $\rightarrow^*_R$, afin de pouvoir traiter également des CTRS. Grâce à cette transformation, on peut définir une classe de CTRS dont le programme logique correspondant définit des relations régulières, on se basera sur les résultats du chapitre 5. Ensuite pour calculer les solutions des formules où $\downarrow^?_R$ est le seul prédicat quand R appartient à cette classe on utilise la clôture des relations régulières sous les opérations ensemblistes.

6.2 Transformation d'un CTRS en programme logique

On étend la technique présentée section 4.3 qui encode la relation de réécriture par un programme logique afin de traiter des CTRS. Cette transformation a pour but d'obtenir des programmes logiques qui préservent aussi bien que possible les propriétés syntaxiques des CTRS. Le programme logique obtenu encode la relation de réécriture avec des étapes de données. Par mesure de clarté les notations utilisées concernant cette nouvelle transformation seront les même que pour celle concernant les TRS décrite section 4.3.

La Table 6.1 spécifie les règles qui transforment les termes et les systèmes de réécritures conditionels en clauses de Horn elle étend la Table 4.1 en autorisant les conditions dans les règles. P_{id} est un prédicat pseudo-régulier qui définit l'égalité entre les termes de données. Son ensemble de clauses est noté par $\mathcal{P}_{id}$ et est

$$\mathcal{P}_{id} = \{ P_{id}(c(x_1, \ldots, x_n), c(y_1, \ldots, y_n)) \leftarrow P_{id}(x_1, y_1), \ldots, P_{id}(x_n, y_n) \mid c \in \mathcal{C} \}$$

Pour un CTRS R, soit $\mathcal{LP}(R)$ le programme logique consistant de $\mathcal{P}_{id}$ et

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

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \dots s_n \rightsquigarrow \langle t_n, \mathcal{G}_n \rangle}{f(s_1, \dots, s_n) \rightsquigarrow \langle f(t_1, \dots, t_n), \bigcup_i \mathcal{G}_i \rangle} \quad \text{si } f \in \mathcal{C}$$

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \dots s_n \rightsquigarrow \langle t_n, \mathcal{G}_n \rangle}{f(s_1, \dots, s_n) \rightsquigarrow \langle x, \bigcup_i \mathcal{G}_i \bigcup \{P_f(t_1, \dots, t_n, x)\} \rangle} \quad \text{si } f \in \mathcal{F}$$

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \qquad s_2 \rightsquigarrow \langle t_2, \mathcal{G}_2 \rangle}{s_1 \downarrow_R s_2 \rightsquigarrow \langle \varepsilon, \mathcal{G}_1 \cup \mathcal{G}_2 \cup P_{id}(t_1, t_2) \rangle}$$

$$\frac{s \rightsquigarrow \langle t, \mathcal{G} \rangle \quad c_1 \rightsquigarrow \langle \varepsilon, \mathcal{G}_1 \rangle \dots c_k \rightsquigarrow \langle \varepsilon, \mathcal{G}_k \rangle}{f(s_1, \dots, s_n) \to s \leftarrow c_1 \dots c_k \rightsquigarrow P_f(s_1, \dots, s_n, t) \leftarrow \mathcal{G} \cup \mathcal{G}_1 \cup \dots \cup G_k}$$

Notons que les variables x introduites par la troisième règle sont de nouvelles variables.

TAB. 6.1 – Convertion des règles de CTRS en clauses de Horn

de l'ensemble des clauses obtenues en appliquant la cinquième règle à toutes les règles de réécriture dans R.

Remarque 6.2.0.2. Dans le cas où les conditions sont de la forme $f(u_1, \dots, u_n) \downarrow_R u$ avec $u_1, \dots, u_n, u$ construits sur $\mathcal{T}(\mathcal{C} \cup \mathit{Var})$ et f symbole de fonction défini. La transformation donne $P_f(u_1, \dots, u_n, z), P_{id}(u, z)$ qui est simplifié par $P_f(u_1, \dots, u_n, u)$.

Exemple 6.2.1. *En supposant un système de réécriture définissant la négation, et les comparaisons $\geq$ et $\leq$ les règles de réécritures conditionnelles et clauses suivantes spécifient la valeur absolue.*

$$\begin{array}{lcl} \mathit{abs}(x) \to x \leftarrow x \geq 0 & \rightsquigarrow & P_{abs}(x, x) \leftarrow P_{\geq}(x, 0) \\ \mathit{abs}(x) \to -x \leftarrow x \leq 0 & \rightsquigarrow & P_{abs}(x, y) \leftarrow P_{-}(x, y) P_{\leq}(x, 0) \end{array}$$

Les règles de la table 6.1 diffèrent légérement de celles présentées dans [64] car [64] traite de TRS géneraux tandis que l'on travaille sur les CTRS basés sur les constructeurs dans ce chapitre. Le théorème 6.2.1.1 diffère légérement du théorème 4.3.2.1 car l'ensemble de clauses $\mathcal{P}_{Id}$ est ajouté à $\mathcal{LP}(R)$ pour pouvoir être capable d'arrêter la dérivation de réécriture à n'importe quelle étape. Pour calculer les solutions de données d'une équation de joignabilité, on a besoin de calculer la dérivation de réécriture jusqu'à l'obtention d'un terme de donnée, $\mathcal{P}_{Id}$ a par conséquent été enlevé.

Le théorème suivant établit la relation entre un CTRS R basé sur les constructeurs et $\mathcal{LP}(R)$.

Théorème 6.2.1.1. *Soit R un CTRS basé sur les constructeurs , s un terme, et $s \rightsquigarrow \langle s', \mathcal{G}\rangle$. Alors on a $s \to^* t$ avec t term de données si et seulement si il y a une substitution μ telle que $t = s'\mu$ et $\mathcal{LP}(R) \models \mathcal{G}\mu$.*

Ce théorème se démontre par les deux lemmes suivants 6.2.2 et 6.2.4 pour lesquels on introduit des notions auxiliares. Pour le corps $\mathcal{G}$ d'une clause générée par les règles de transformations à partir d'une règle $l \to r$, on définit

- $O_{\mathcal{G}} = \{ u \mid A_u \in \mathcal{G} \}$
- $O^0_{\mathcal{G}} = \{ u \in O_{\mathcal{G}} \mid u \not< v \text{ pour tout } v \in O_{\mathcal{G}} \}$
- $O^{i+1}_{\mathcal{G}} = \{ u \in O_{\mathcal{G}} \setminus \bigcup_i O^i_{\mathcal{G}} \mid u \not< v \text{ for all } v \in O_{\mathcal{G}} \setminus \bigcup_i O^i_{\mathcal{G}} \}$
- $\mathit{Height}(\mathcal{G}) = \max\{ i \mid O^i_{\mathcal{G}} \neq \emptyset \}$
- $\mathcal{G}^i = \{ A_u \in \mathcal{G} \mid u \in O^i_{\mathcal{G}} \}$

Remarquons que $O_{\mathcal{G}}$ est identique à $\mathit{PRedPos}(r)$ (définition 4.1.6), $\mathit{Height}(\mathcal{G})$ est le nombre maximal de positions de redex potentiels imbriqués dans r, et $O^i_{\mathcal{G}}$ est l'ensemble des positions de redex potentiel qui sont imbriquées sous $\mathit{Height}(\mathcal{G}) - i$ redex.

Dans la prochaine définition on note $s \leftarrow s' \Leftarrow C$ si s peut être réécrit dans s' si il a besoin que toutes les conditions dans C doivent être satisfaites. La relation *cpt* qui compte le nombre nécéssaire d'étapes de réécriture est récursivement définit comme suit :

- $\mathit{cpt}(s \to t \Leftarrow \emptyset) = 1$
- $\mathit{cpt}(s \to s' \Leftarrow \{\bigcup_{s\downarrow^?_R t \in C} s\sigma \to^* ut\sigma \to^* u\} \to^* t) = 1 + \mathit{cpt}(s' \to^* t) + \mathit{cpt}(\{\bigcup_{s\downarrow^?_R t \in C} s\sigma \to^* ut\sigma \to^* u\})$.
- $\mathit{cpt}(\{s' \to^* t\} \cup r) = \mathit{cpt}(s' \to^* t) + \mathit{cpt}(r)$

Lemme 6.2.2. *Soit R un système de réécriture conditionnel basé sur les constructeurs, s un terme, et $s \rightsquigarrow \langle s', \mathcal{G}\rangle$. Si $\mathcal{LP}(R) \models \mathcal{G}\mu$ pour des substitutions μ, alors $s \to^* s'\mu$.*

Démonstration. Pour la partie conditionnelle on a à prouver que si $\mathcal{LP}(R) \models \mathcal{G}\mu$ pour une substitution μ, alors $\exists n\ \mathit{cpt}(s \to_R s'\mu) = n$.

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

$h = 0$ *:* Signifie que $s \rightsquigarrow \langle s, \emptyset\rangle$, alors s est irréducible alors $s \to^* s$ avec 0 étape.

$h > 0$ *:* 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 avec $A_u = P_f(t_1, \ldots, t_n, t)\sigma$. Cette clause appartient à $\mathcal{LP}(R)$: soit $f(s_1, \ldots, s_n) \to r \Leftarrow c$ la règle de réécriture conditionnelle qui a produit la clause. Sans perte de généralité, on peut considérer que l'atome A_u est tel qu'il n'existe pas d'atome $A_v\mu$ tel que $v > u$. Le prohain but dans la preuve est $(\mathcal{G}\setminus A_u)\mu \cup \mathcal{B}\sigma$. Soit $s'' = s[u \leftarrow \sigma r \Leftarrow \sigma c]$, $s'' \rightsquigarrow \langle s', \mathcal{G} \setminus \{ A_v \mid u \leq v \}) \cup \mathcal{B}\sigma\rangle$. Notons que σ est une substitution de donnée car ni A_u ni $P_f(t_1, \ldots, t_n, t)$ ne contiennent de symboles de fonction définie 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 $A_u.\mathcal{LP}(R) \models (\mathcal{G} \setminus \{ A_v \mid u \leq v \})\mu \cup \mathcal{B}\sigma$ avec une preuve de longueur inférieure à l, donc $s[u \leftarrow \sigma r \Leftarrow c] \rightarrow^* s'\mu \uplus \sigma$ par une dérivation de donnée. Si $u \notin Pos(s')$ on a $s' = s''$, 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 $s[u \leftarrow \sigma r \Leftarrow c\sigma] \rightarrow s'\mu$ or l'étape $s \rightarrow s[u \leftarrow \sigma r \Leftarrow c\sigma$ est une étape de donnée donc $s \rightarrow s'\mu$ par une dérivation de données et $s'\mu$ est un terme de données. □

Pour la preuve du lemme 6.2.4, on introduit la notation suivante.

Définition 6.2.3. *Pour une dérivation de données $s \rightarrow^* t$ on définit le terme $(s \overset{bas}{\rightarrow}{}^* t) \setminus u$ comme suit :*

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

De manière intuitive, $(s \overset{bas}{\rightarrow}{}^* t) \setminus u$ représente le terme obtenu en appliquant les étapes de réécriture de $s \rightarrow^* t$ à $s|_u$.

Lemme 6.2.4. *Soit R un CTRS basé sur les constructeurs, s un terme, et $s \rightsquigarrow \langle s', \mathcal{G} \rangle$. Si $s \rightarrow^* t$ avec t terme de données alors $\mathcal{LP}(R) \models \mathcal{G}\delta$ où $\delta = \{ x_v \mapsto (s \overset{bas}{\rightarrow}{}^* t) \setminus v \mid v \in PRedPos(s) \}$ et $s'\delta = t$.*

Démonstration. On fait cette preuve par induction sur le nombre n d'étapes de réécritures nécessaires. Pour la partie conditionnelle on a à prouver que si $\exists n\ cpt(s \rightarrow_R t) = n$ alors $\mathcal{LP}(R) \models \mathcal{G}\delta$ où $\delta = \{ x_v \mapsto (s \overset{bas}{\rightarrow}{}^* t) \setminus v \mid v \in PRedPos(s) \}$ et $s'\delta = t$.

$n > 0$: $s \rightarrow_{[u, l \rightarrow r \Leftarrow c, \sigma]} t' \rightarrow^m t$. Par propriété de la réécriture de données $u \in PRedPos(s)$ implique $u \in O_{\mathcal{G}}$. Soit $A_u = P_f(s_1, \ldots, s_n, x_u)$ et $P_f(s'_1, \ldots, s'_n, s'') \leftarrow \mathcal{B}$ la clause de $\mathcal{LP}(R)$ produite pour $l \rightarrow r \Leftarrow c$. On a $t' \rightsquigarrow \mathcal{G} \setminus \{ A_v \mid v > u \} \cup \{ A_v \mid v \in PRedPos(r\sigma) \}$ et $cpt(t' \rightarrow^m t) < n$ alors $\mathcal{LP}(R) \models (\mathcal{G} \setminus \{ A_v \mid v > u \} \cup \{ A_v \mid v \in PRedPos(r\sigma) \} \cup \{ C_i \mid i \in c)\delta' \}$ où $\delta' = \{ x_v \mapsto (t' \overset{bas}{\rightarrow}{}^* t) \setminus v \mid v \in PRedPos(t') \}$. Par définition, pour tout $v \not> u$, $(s \overset{bas}{\rightarrow}{}^* t) \setminus v = (t' \overset{bas}{\rightarrow}{}^* t) \setminus v$, Par l'hypothèse d'induction,pour tout $i \in c$,$\mathcal{LP}(R) \models (\{ C_i \mid i \in c)\delta' \}$. Donc pour tout $v \not> u$, $x_v\delta' = x_v\delta$ et $\mathcal{LP}(R) \models (\mathcal{G} \setminus \{ A_v \mid v > u \})\delta$.

Maintenant, on a à prouver que $\mathcal{LP}(R) \models \{ A_v \in \mathcal{G} \mid v \geq u \}\delta$. Par définition, pour tout $v > u$, $(s \overset{bas}{\rightarrow}{}^* t) \setminus v = s|_v$ alors, comme dans le cas $k = 0$, $s_i\delta = s|_{u.i}$ par conséquent $f(s_1, \ldots, s_n)\delta = l\sigma = f(s'_1, \ldots, s'_n)\sigma$. De plus $x_u\delta = (s \overset{bas}{\rightarrow}{}^* t) \setminus u = (t' \overset{bas}{\rightarrow}{}^* t) \setminus u$ et $s'' = Irr(r)$ alors $(t' \overset{bas}{\rightarrow}{}^* t) \setminus u = Irr(r)[v \leftarrow (s \overset{bas}{\rightarrow}{}^* t) \setminus u|_v, v \in PRedPos(r)]$. 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 par conséquent $\mathcal{G}\delta$ est une conséquence logique de $(\mathcal{G} \setminus \{A_u\})\delta \cup \mathcal{B}\sigma'$. $\mathcal{B}\sigma' \subseteq (\{ A_v \mid v \in PRedPos(r\sigma) \})\delta'$ et par conséquent $\mathcal{LP}(R) \cup \models \mathcal{B}\sigma'$.

Il reste à prouver que $s'\delta = t$. Par hypothèse d'induction $Irr(t')\delta' = t$. $s' = Irr(s)$ et $s \rightarrow_{[u,l\rightarrow r\Leftarrow c,\sigma]} t'$. Alors, soit $x_u \notin Var(s')$ et dans ce cas $s' = Irr(t')$ ou $x_u \in Var(s')$ et dans ce cas $Irr(t') = s'[u \leftarrow Irr(r\sigma) \Leftarrow c\sigma]$. Dans le premier cas, comme $x_v\delta = x_v\delta'$ pour tout $v \not\geq u$, on obtient $s'\delta = Irr(t')\delta' = t$. 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 $s'\delta = Irr(t')\delta'[u \leftarrow x_u\delta] = Irr(t')\delta'[u \leftarrow t|_u] = t[u \leftarrow t|_u] = t$. □

Définition 6.2.5. *Soit R un CTRS et $\mathcal{P}$ un programme logique. On dit que $\mathcal{P}$* encode *$\rightarrow_R^*$ si pour tout les termes s et t, on a la relation $s \rightarrow_R^* t$ si et seulement si pour des substitutions μ, $\mathcal{P} \models \mathcal{G}\mu$ et $t = s'\mu$ où $s \rightsquigarrow \langle s', \mathcal{G}\rangle$.*

6.3 Résolution des équations de joignabilité

Dans cette section, nous montrons comment utiliser les langages d'arbres pour résoudre la joignabilité de données. Nous présentons d'abord une méthode générale présentée dans [66], pour résoudre ce problème à l'aide des langages d'arbres. Puis nous montrons que grâce aux propriétés des programmes pseudo-réguliers, nous pouvons résoudre les formules pseudo-régulières du 1$^{\text{er}}$ ordre, c'est à dire des formules basées sur l'unique prédicat de joignabilité modulo un CTRS pseudo-régulier général.

La méthode de résolution des équations de joignabilité s'inspire de celle décrite dans [66] 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 6.3.1. *Soit R un système de réécriture basé sur les constructeurs, $s \downarrow_R^? t$ est appelée une* équation de joignabilité *ou* problème joignabilité. *$\downarrow_R^?$ est appelé* prédicat de joignabilité. *L'ensemble des solutions de données closes d'une telle équation est noté $SOL(s \downarrow_R^? t)$, et est l'ensemble $\{\, \sigma$ data-substitution close $\mid s\sigma \downarrow_R^? t\sigma \,\}$.*

Dans la suite nous allons utiliser le programme P_{Eq} qui définit l'égalité entre deux termes. On a vu au Chapitre 5 que ce prorgamme est régulier.

Définition 6.3.2. *Soit R un système de réécriture basé sur les constructeurs et $s \downarrow_R^? t$ une équation de joignabilité, $\vec{x}$ une énumération de variables de $s \downarrow_R^? t$. On définit $\mathcal{LP}(s \downarrow_R^? t) = P_{s\downarrow_R^? t}(\vec{x}) \leftrightarrow P_{Eq}(s', t'), \mathcal{G}_s, \mathcal{G}_t$ où $s \rightsquigarrow (x_s, \mathcal{G}_s)$ et $t \rightsquigarrow (x_t, \mathcal{G}_t)$.*

On montre maintenant que la clause $\mathcal{LP}(s \downarrow_R^? t)$ permet de représenter l'ensemble des solutions de $s \downarrow_R^? t$.

Lemme 6.3.3. *Soit R un CTRS basé sur les constructeurs et $s \downarrow_R^? t$ une équation de joignabilité $\mathcal{LP}(R) \cup \mathcal{LP}(s \downarrow_R^? t) \cup \mathcal{P}_{Eq} \models P_{s\downarrow_R^? t}(s_1, \ldots, s_n)$ si et seulement si $\sigma = \{\, x|_i \mapsto s|_i \mid 1 \leq i \leq n \,\} \in SOL(s \downarrow_R^? t)$.*

Démonstration. Tout d'abord, on peut remarquer que pour tout terme t tel que $t \rightsquigarrow (t', \mathcal{G})$ et toute instance de donnée close $t\sigma'$ de ce terme telle que $t\sigma' \rightsquigarrow (t'', \mathcal{G}'')$, on a $\mathcal{G} = \mathcal{G}''\sigma'$ car toutes les positions de fonction définie sont les même dans les deux termes. De plus $t'' = t\sigma'$. Soient σ une substitution de donnée close telle que $s\sigma$ et $t\sigma$ sont des instances de données closes de s et t respectivement et μ une substitution telle que $\mathcal{LP}(R) \cup \mathcal{LP}(s \downarrow^?_R t) \cup \mathcal{P}_{Eq} \models (\mathcal{P}_{Eq}(s', t'), \mathcal{G}_s, \mathcal{G}_t)\sigma\mu$ avec $s \rightsquigarrow (x_s, \mathcal{G}_s)$ et $t \rightsquigarrow (x_t, \mathcal{G}_t)$, d'après le théorème 6.2.1.1 , $\mathcal{LP}(R) \models \mathcal{G}_s\mu$ si et seulement si $s \rightarrow s'\sigma\mu$ et $\mathcal{LP}(R) \models \mathcal{G}_t\mu$ si et seulement si $t \rightarrow t'\sigma\mu$ par des dérivations de données et $s'\sigma\mu$ et $t'\sigma\mu$ sont des termes de données. Or $\mathcal{LP}(R) \cup \mathcal{LP}(s \downarrow^?_R t) \cup \mathcal{P}_{Eq} \models (\mathcal{P}_Eq(s', t'))\sigma\mu$ implique que $s'\sigma\mu = t'\sigma\mu$ donc $\mathcal{LP}(R) \cup \mathcal{LP}(s \downarrow^?_R t) \cup \mathcal{P}_{Eq} \models (\mathcal{P}_Eq(s', t'), \mathcal{G}_s, \mathcal{G}_t)\sigma\mu$ si et seulement si $s\sigma$ et $t\sigma$ joignables de données donc si $\sigma \in SOL(s \downarrow^?_R t)$. □

En regardant attentivement on s'aperçoit que $\mathcal{LP}(s \downarrow^?_R t)$ est encore plus général qu'une jointure généralisée car s et t peuvent contenir des symboles constructeurs. Au vu de l'état de nos connaissances sur le test du vide de jointure de programme, on a besoin à la fois de restrictions sur s et t et sur le CTRS d'origine. On va utiliser ici les résultats du chapitre 5 qui nous permet d'obtenir un programme logique PR fini à partir d'une jointure généralisée de programmes logiques PR. On va donc maintenant se restreindre doublement. La première restriction concerne la clôture transitive d'un CTRS où l'on veut qu'elle puisse se modéliser par une programme logique PR. La seconde restriction concerne s et t afin qu'ils n'induisent pas de constructeurs dans la jointure.

Définition 6.3.4. *Un CTRS R est un* CTRS régulier général *si $\mathcal{LP}(R)$ est équivalent à un programme pseudo-régulier $\mathcal{P}$.*

On verra à la fin de ce chapitre que cette classe n'est pas seulement théorique.

Définition 6.3.5. *Une* équation de joignabilité pseudo-régulière *est une équation $s \downarrow^?_R t$ telle que s et t sont des termes construits sur $\mathcal{T}(\mathcal{F}, Var)$ c'est à dire des termes ne contenant aucun constructeurs.*

La définition précédente semble interdire les équations du type $f(x) \downarrow^?_R 0$ où f est un symbole de fonction définit et 0 un constructeur. En fait on peut exprimer ce type d'équations en rajoutant au système de réécriture une nouvelle règle $zero \rightarrow 0$ et de remplacer l'équation par $f(x) \downarrow^?_R zero$. Plus généralement, on peut utiliser la même méthode pour tous les termes de données sans variables commune avec le reste de l'équation.

Lemme 6.3.6. *Soit $s \downarrow^?_R t$ une équation de joignabilité pseudo-régulière. $\mathcal{LP}(s \downarrow^?_R t)$ est une définition de jointure généralisée.*

Démonstration. Pour montrer ce lemme, il suffit de remarquer que pour un terme s sans constructeur $Irr(s) = \emptyset$ et 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 $s \rightsquigarrow (x_e, \mathcal{G})$ et $\mathcal{G}$ ne contient aucun symbole de fonction. □

On peut donc maintenant montrer le résultat suivant.

Théorème 6.3.6.1. *Soit R un CTRS pseudo-régulier général et $s \downarrow^?_R t$ une équation de joignabilité pseudo-régulière. $SOL(s \downarrow^?_R t)$ peut être représenté par un programme pseudo-régulier.*

Démonstration. D'après la définition 6.3.4 $\mathcal{LP}(R)$ est équivalent à un programme PR $\mathcal{P}$ Les clauses définissant P_{Eq} sont des clauses pseudo-régulières. D'après le lemme 6.3.6 $LP(s \downarrow^?_R t)$ est une définition de jointure généralisée. D'après le lemme 6.3.3, on a :

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

si et seulement si $\sigma = \{\, x|_i \mapsto s|_i \mid 1 \leq i \leq n \,\} \in SOL(s \downarrow^?_R t)$. Donc d'après le théorème 3.5.1.1 $\mathcal{P}$ et $P_{s\downarrow^?_R t}(s_1, \ldots, s_n)$ peuvent être transformés en un programme pseudo-régulier. □

6.4 Résolution des formules pseudo-régulières

Dans cette section nous montrons comment combiner les résultats sur les programmes pseudo-réguliers décrits chapitre 5 et ceux de la section précédente pour trouver les solutions des formules pseudo-régulières. Voici tout d'abord les classes de formules que nous allons considérer.

Définition 6.4.1. *Soit R un CTRS pseudo-régulier général, 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 définie comme suit :
$SOL(s \downarrow^?_R t)$ est l'ensemble des solutions de données 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 $\sigma|_{Var(e_2)} \in SOL(e_2) \,\}$
$SOL(\exists x e) = \{\, \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \exists \sigma' \in SOL(e), \sigma = \sigma'|_{Var(e)\setminus\{x\}} \,\}$
$SOL(\forall x e) = \{\, \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \forall \sigma' \in SOL(e), \sigma = \sigma'|_{Var(e)\setminus\{x\}} \,\}$

Même si les restrictions sur les équations de joignabilité sont fortes, de nombreuses propriétés intéressantes concernant les CTRS peuvent être exprimées par des formules pseudo-régulières. Parmis celles-ci on peut citer :

- 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 points 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-régulières est donnée par l'algorithme suivant :

Algorithme 1. *Soit R un CTRS pseudo-régulier général 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 $\mathcal{P}_e$ definit $SOL(e)$ 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 $\vec{x} = Var(e_1 \wedge e_2)$ définit $SOL(e_1 \vee e_2)$.*
5. *Si $\mathcal{P}_e$ définit $SOL(e)$ alors tout programme $\mathcal{P}_{\exists xe}$ équivalent à $P_e \cup \{P_{\exists xe,\vec{y}}(\vec{y}) \leftarrow P_{e,\vec{x}'}(\vec{x}')\}$ définit $SOL(\exists xe)$ si $\vec{y}$ est la projection de $\vec{x}$ sur tout ses arguments sauf celui de la variable x.*

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

Théorème 6.4.1.1. *Soit R un CTRS pseudo-regulier général et e une formule pseudo-régulière et $\mathcal{P}_e$ le programme résultant de l'application de l'algorithme 1. $\mathcal{P}_e \models P_{e,\vec{x}}(\vec{t})$ si et seulement si $\{ \vec{x}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq length(\vec{x}) \} \in SOL(e)$.*

Démonstration. Cette preuve se fait par induction sur la structure de la formule. D'après le théorème 6.3.6.1, on sait que pour chaque formule élémentaire $s \downarrow_R^? t$, $\mathcal{P}_{s\downarrow_R^? t} \models P_{s\downarrow_R^? t,\vec{x}}(\vec{t})$ si et seulement si $\{ \vec{x}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq length(\vec{x}) \} \in SOL(s \downarrow_R^? t)$. Soit e_1 et e_2 deux formules telles que $\mathcal{P}_{e_1}$ et $\mathcal{P}_{e_2}$ sont les programmes pseudo-réguliers obtenus pour représenter les solutions de ces deux formules.

- D'après le théorème 5.2.8.1, on sait que $\mathcal{P}_{\neg e_1}$ définit $SOL(\neg e_1)$ et est un programme pseudo-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 un programme PR, par conséquent d'après le théorème 3.5.1.1 $\mathcal{P}_{e_1 \wedge e_2}$ est un programme PR 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 un programme PR, par conséquent d'après le théorème 3.5.1.1 $\mathcal{P}_{e_1 \wedge e_2}$ est un programme PR qui définit les solutions 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 définit par un programme PR, par conséquent d'après le théorème 3.5.1.1 $P_{\exists x e_1,\vec{y}}$ est défini par un programme PR. 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 $SOL(e_1)|_{\{}x\}$.

□

Maintenant que l'on a défini un algorithme pour résoudre les formules pseudo-régulières du premier ordre il nous reste à déterminer des CTRS pour lesquels $\mathcal{LP}(R)$ est équivalent à un programme logique pseudo-régulier. A cet effet on va utiliser les résultats du chapitre 5 qui définit des extensions syntaxiques des programmes logiques pseudo-régulier.

Définition 6.4.2. *Un CTRS R basé sur les constructeurs est dit pseudo-régulier si toutes les règles de réécritures sont de la forme*
$f(t_1, \ldots, t_n) \rightarrow C[f_1(\vec{u}_1), \ldots, f_m(\vec{u}_m)] \leftarrow f'_1(\vec{u}'_1) \downarrow_R u'_1 \ldots f'_k(\vec{u}'_k) \downarrow_R u_k$ *où*
- *où $\vec{u}_j, u_j$ est composé de variables et de termes clos sur des constructeurs.*
- *C est un contexte de constructeurs, $f, f_1, \ldots, f_m, f'_1, \ldots, f'_k$ sont des symboles de fonctions définis*
- $\bigcup_{1 \leq i \leq m}(\vec{x}_i) \cup \bigcup_{1 \leq i \leq k}(\vec{x'_i}) \cup \bigcup_{1 \leq i \leq k}(x'_k) \subseteq Var(f(t_1, \ldots, t_n)) \cup Var(C)$
- *il existe une application π: $Var \mapsto \mathbb{N}^+$, telle que $\pi(x) = u$ implique que toutes les occurrences de x dans $t_1, \ldots, t_n$ et dans C sont à la position u,*
- *toutes les variables de $\vec{x}'_i$ ont la même image par π comme x'_i ($1 \leq i \leq k$).*
- *l'image par π de toutes les variables de $\vec{x}_i$ est u, la position de $f_i(\vec{x}_i)$ dans $C[f_1(\vec{x}_1), \ldots, f_m(\vec{x}_m)]$ ($1 \leq i \leq m$).*

On remarquera C est la partie irréducible *de $C[f_1(\vec{x}_1), \ldots, f_m(\vec{x}_m)]$*

Exemple 6.4.3.
$R = \{f(s(c(x,y)), s(c(x,z))) \rightarrow s(c(g(x), f(y,y))) \Leftarrow g(z) \downarrow_R y,\ f(s(0), s(0)) \rightarrow 0, g(s(x)) \rightarrow s(x)\}$ *est pseudo-régulière. La partie irreducible du lhs de la première règle est $s(c(\Box_1, \Box_2))$ et il ne contient que positions de redexs* 1.1 *et* 1.2 *correspondant aux redex potentiels $g(x)$ et $f(y,y)$. Notons que la Definition 6.4.2 n'interdit pas les variables dupliquées au sein d'un même redex*

potentiel.

Lemme 6.4.4. *Si R est un CTRS pseudo-regulier, $\mathcal{LP}(R)$ est un programme logique RP-clos.*

Démonstration. Chaque règle d'un CTRS pseudo-régulier est de la forme $f(t_1, \ldots, t_n) \to C[f_1(\vec{u}_1), \ldots, f_k(\vec{u}_k)]$ où C est la partie irréducible du lhs de la règle, et $f, f_1, \ldots, f_k$ sont des symboles de fonctions. C et chaques t_i, u_j ne contiennent pas de symboles de fonctions définis. Par conséquent la clause produite pour cette 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 $f_1(\vec{u}_1), \ldots, f_k(\vec{u}_k)$ sont des variables de $f(t_1, \ldots, t_n)$, la clause ne contient aucune variable existentielle. En étendant l'application π de
$f(t_1, \ldots, t_n) \to C[f_1(\vec{u}_1), \ldots, f_k(\vec{u}_k)]$ à $\pi' = \pi \cup_{1 \leq i \leq k} \{u_{v_i} \mapsto v_i\}$ on obtient que la transformation concernant les règles forment une clause RP-close.

On montre maintenant que chaque jointure conditionnelle n'ajoute seulement qu'un atome dans le corps de clause. Pour chaque $f(u_1, \ldots, u_n) \downarrow^?_R u$ de la partie conditionnelle on a un unique symbole de fonction f. Par conséquent en appliquant $\rightsquigarrow$ on créé un unique atome $P_f(u_1, \ldots, u_n, u)$ qui est ajouté au corps de la clause. Il reste à prouver que toutes les variables de chaque atome A de $\mathcal{G}$ (avec $C \rightsquigarrow \langle \varepsilon, \mathcal{G} \rangle$) ont la même image par π ce qui est par définition le cas car chaque atome A est de la forme $P_f(\vec{u}, u)$ quand il provient de la condition $f(\vec{u}) \downarrow^?_R u$. □

On remarquera que si R est un CTRS pseudo-regulier alors $\mathcal{LP}(R)$ peut se transformer en un programme logique pseudo-régulier équivalent, par conséquent les CTRS pseudo-réguliers font partie de la classe des CTRS pseudo-réguliers généraux.

6.5 Travaux liés

On peut rapprocher ce travail de ceux effectués dans le cadre des formules de la théorie de la réécriture en une étape, introduite par [6] et qui a été démontrée indécidable par Treinen [97]. Des sous-classes de cette théorie ont alors été étudiées (par exemple [72, 80]) en restreignant la classe des formules autorisées et la classe des systèmes de réécriture utilisés. La plupart de ces résultat se basent sur des techniques à base d'automate d'arbres. Etant donné qu'il est bien connu que l'unifiabilité et la joignabilité de deux termes sont indécidables sans mettre de très fortes restrictions sur le TRS (voir par exemple [39, 62]), nous avons pris le parti de restreindre dès le départ la classe des CTRS traités.

Bien que possédant de fortes restrictions les CTRS pseudo-réguliers ont un certain pouvoir d'expression, on pourra revoir l'exemple 4.1.4, ou par

exemple dans les algèbres de processus étudiées par D.Lugiez et D.Schnoebelen [70] où la clôture transitive et reflexive de leur transitions est exprimée dans [65] par un CTRS pseudo-régulier. Toutefois il serait sans doute intéressant d'utiliser la même méthode appliquée à des CTRS un peu plus généraux. Par exemple pour une simple jointure entre deux relations il suffit que les arguments qui sont joints se comportent comme un automate (c'est à dire que les variables d'un même argument de la tête soient arguments d'atomes différents au niveau des atomes d'un corps linéaire).

On pourra aisément remarquer qu'il suffit d'une légère adaptation pour appliquer directement la formule à des prédicats définissant des automates ou des relations pseudo-régulière. Dans [74] l'auteur a défini une classe d'automates de chaînes à partir de formules quantifiées de $\forall$ et $\exists$ de premiers ordres sur des propriétés de positions dans l'automate. Ces propriétés concernent des comparaisons de positions ($>$), ainsi que la présence d'un symbole particulier à une position. En définissant une relation associant une position à un symbole, une même variable représentant les symboles définissant l'automate, et une relation de comparaison sur les positions, il paraît très envisageable de représenter l'automate le plus général obtenu par ces formules. D'une manière similaire on pourrait également synthétiser les automates dans la logique temporelle de [24] qui sont définis par une formule du même type (le temps représentant la position dans l'automate). De plus comme dans le formalisme succintement décrit pour synthétiser ces automates, une même variable représente un automate, on peut alors imaginer avoir différentes variables pour différents automates et ainsi étendre cette technique pour un réseau d'automate ce qui pourrait également amener à synthétiser des automates cellulaires.

Chapitre 7

Satisfiabité de formules positives du second ordre

7.1 Introduction

Les théories du second ordre ont été largement étudiées depuis longtemps à cause de leur intérêt pratique dans plusieurs domaines de l'informatique. La classe la plus étudiée est celle de la logique monadique du second ordre. Plusieurs variantes de cette logique ont été prouvées décidables grâce à l'utilisation des techniques d'automates (voir [96] pour un aperçu). Les solutions des formules dans une telle logique sont représentées par des automates sur des chaines ou des arbres. Par exemple les logiques faibles du second ordre avec k successeurs WSkS sont résolues en utilisant les automates d'arbres finis [95] qui définissent des relations régulières. Les applications de la logique monadique du second ordre sont nombreuses, depuis la vérification de circuit [31] qui est son utilisation historique, à la vérification de programmes [54].

Comme dans le chapitre précédant, nous étudions une classe de formules basées sur le prédicat $\downarrow^?_R$. Dans ce chapitre on considérera que chaque symbole de fonction f d'arité n d'un CTRS R induit une relation d'arité $n+1$. Cette relation est une relation sur les termes de données elle est définit comme l'ensemble $\{ (t_1, \ldots, t_n, t) \mid f(t_1, \ldots, t_n) \rightarrow^* t \}$, c'est une relation entre les arguments de f et le résultat de la clôture transitive de cette relation. Dans les formules du second ordre étudiées dans ce chapitre, les variables du second ordre représentent des relations définissables par des CTRS sur les constructeurs. Par exemple, une solution de la formule $\forall x X(x) \downarrow^?_R x + x$ où x est une variable du premier ordre et X une variable du second, X est une relation (t_1, t_2) telle que $t_2 = 2 \times t_1$. Le but est ici de construire automatiquement les règles de réécritures qui définissent cette relation.

La principale contribution de ce chapitre est la définition d'un algorithme

général pour décider de la satisfiabilité des formules positives de joignabilité du second ordre quand il existe un algorithme qui produit une représentation finie des solutions des formules de joignabilité du premier ordre correspondant. La sortie de l'algorithme est l'ensemble vide quand la formule du second ordre n'est pas satisfiable et une instance particulière des variables du second ordre autrement.

L'algorithme général est ensuite utilisé pour montrer la décidabilité de la satisfiabilité des formules positives du second orde. Quand une formule est satisfiable, on obtient un programme logique qui calcule un CTRS qui définit une instance possible pour les variables du second ordre. Ce résultat peut être utilisé pour générer automatiquement un CTRS (qui peut être considéré comme un programme) à partir de la spécification du résultat attendu. L'exemple 4.1.4 est un CTRS pseudo-régulier. Il n'est cependant pas réellement conditionnel, car il n'a pas de conditions associées aux règles. Par contre l'exemple 7.1.1 expose des règles conditionnelles.

Exemple 7.1.1. *Considérons d'abord les deux règles suivantes qui définissent la soustraction élément par élément de deux listes d'entiers.*

$sl(\bot, \bot) \to \bot$

$(sl(c(x_1, y_1), c(x_2, y_2)) \to c(x_3, sl(y_1, y_2))) \leftarrow x_2 + x_3 \downarrow_R x_1$

Dans le contexte de TRS sans condition, la fonction sl nécéssiterait la définition explicite de la soustraction entre deux entiers.

$\forall x X(x) = x+x$ *est une formule positive du second ordre. Notre procédure construit automatiquement le CTRS suivant qui définit la seule instance possible pour* X*.* f_X *est le symbole représentant la solution pour* X*.*

$$\begin{array}{ll} f_X(\bot) \to \bot & f'_X(\bot) \to 1(\bot) \\ f_X(0(x)) \to 0(f_X(x)) & f'_X(0(x)) \to 1(f_X(x)) \\ f_X(1(x)) \to 0(f'_X(x)) & f'_X(1(x)) \to 1(f'_X(x)) \end{array}$$

Notons que f_X *introduit un* 0 *dans les unités pour faire un décalage des nombres ce qui correspond au double d'un entier représenté en binaire.*

La section 7.2 définit l'algèbre du second ordre que l'on considère. La section 7.3 définit la classe des formules positives du second ordre que l'on utilise. La section 7.4 décrit l'algorithme pour les décider. La section 7.5 définit la transformation de programmes logique en CTRS pseudo-régulier pour représenter les relations dans le même formalisme que celui d'origine. Finallement la section 7.6 conclu ce chapitre.

7.2 Relations et second ordre

Ce chapitre traite principalement de relations sur les termes de donnés clos. Une *relation de donnée* $\tilde{r}$ d'*arité* n est un sous-ensemble de $\mathcal{T}^{n+1}(\mathcal{C})$. Notons que ce qu'on appelle l'arité pour une relation est son nombre de

composant moins 1 car dans notre contexte, une relation $\tilde{r}$ avec n composants modèlise en fait une relation de $\mathcal{T}^{n-1}(\mathcal{C})$ à $\mathcal{T}(\mathcal{C})$. La notation $\tilde{r}(t_1, \ldots, t_{n-1})$ représente l'ensemble $\{ t_n \mid (t_1, \ldots, t_{n-1}, t_n) \in \tilde{r} \}$ de termes de données clos. L'ensemble de toutes les relations de données est récursivement énumérable, alors on associe à chaque relation de donnée d'arité n un unique *symbole de relation* d'arité n différent de $(\Sigma \cup \mathit{Var})$ et appellé le *nom* de la relation. $\mathcal{R}$ représente l'ensemble des noms des relations.

Soit $\mathcal{X}$ un ensemble de variables du second ordre avec arité, différent de $\Sigma \cup \mathit{Var} \cup \mathcal{R}$. L'ensemble des termes du second ordre construits sur la signature $\mathcal{T}(\Sigma, \mathit{Var}, \mathcal{R}, \mathcal{X})$ est le plus petit ensemble tel que

- *Var* est inclus dans l'ensemble,
- si $t_1, \ldots, t_n$ sont des termes du second ordre et f est soit un symbole d'arité n de Σ ou une variable d'arité n de $\mathcal{X}$ ou un symbole de relation d'arité n de $\mathcal{R}$, alors $f(t_1, \ldots, t_n)$ est un terme du second ordre.

Pour un terme du second ordre, $\mathit{Var}(t)$ dénote l'ensemble des variables du premier et du second ordre de t. Un terme du second ordre t est dit *clos* si $\mathit{Var}(t)$ est vide. Soit R un CTRS, t un terme clos du second ordre, alors le *modèle* de t noté par $\mathcal{M}_R(t)$ est l'ensemble des termes clos de données inductivement définis sur la structure des termes [1]

- $\{ c(t_1, \ldots, t_n) \mid \forall 1 \leq i \leq n, t_i \in \mathcal{M}_R(s_i) \}$ si $t = c(s_1, \ldots, s_n)$ et $c \in \mathcal{C}$,
- $\{ \tilde{f}(t_1, \ldots, t_n) \mid \forall 1 \leq i \leq n, t_i \in \mathcal{M}_R(s_i) \}$ si $t = f(s_1, \ldots, s_n)$ et $f \in \mathcal{F}$,
- $\{ \tilde{r}(t_1, \ldots, t_n) \mid \forall 1 \leq i \leq n, t_i \in \mathcal{M}_R(s_i) \}$ si $t = \tilde{r}(s_1, \ldots, s_n)$ et $\tilde{r} \in \mathcal{R}$.

Notons que pour un terme clos du premier ordre t, $\mathcal{M}_R(t)$ est l'ensemble des termes clos de données s tel que $t \rightarrow_R^* s$ avec une dérivation de données. C'est aisément prouvé par induction sur la hauteur du terme t. La relation définie par un symbole $f \in \mathcal{F}$ d'arité n est l'ensemble
$\{ (t_1, \ldots, t_n, \mathcal{M}_R(f(t_1, \ldots, t_n))) \mid \mathcal{M}_R(f(t_1, \ldots, t_n) \neq \emptyset \}$.

Une *substitution du second ordre* σ est une application de $\mathit{Var} \cup \mathcal{X}$ à $\mathcal{T}(\mathcal{F} \cup \mathcal{C}, \mathit{Var}) \cup \mathcal{R} \cup \mathcal{X}$ telle que $x\sigma \neq x$ seulement pour un ensemble fini de $\mathit{Var} \cup \mathcal{X}$ et

- si $x \in \mathit{Var}$, $x\sigma \in \mathcal{T}(\mathcal{F} \cup \mathcal{C}, \mathit{Var})$
- si $x \in \mathcal{X}$ et x est d'arité n, $x\sigma$ est soit un symbole de $\mathcal{F} \cup \mathcal{R}$ d'arité n ou est x lui même.

Le domaine de σ est l'ensemble des variables telles que $x\sigma \neq x$. Une substitution du second ordre est appelée *close de donnée* si pour tout $x \in \mathit{Dom}(\sigma) \cap \mathit{Var}$, $x\sigma$ est un terme clos de donnée. On étend σ à $\mathcal{T}(\Sigma, \mathit{Var}, \mathcal{R}, \mathcal{X})$ de manière homomorphique.

Exemple 7.2.1. *Considérons le CTRS de l'exemple 4.1.4, la relation* $\tilde{r} = \{(t_1, 0(t_1))\}$, *t le terme du second ordre* $X(x)+y$ *et la substitution* $\sigma = \{X \mapsto \tilde{r}, x \mapsto 1(\bot), y \mapsto 1(0(\bot))\}$. $\mathit{Var}(t) = \mathit{Dom}(\sigma) = \{X, x, y\}$. σ *est clos de*

[1] Notons que dans la définition, la notation $s(t_1, \ldots, t_n)$ pour un symbole s d'arité 0 (c'est à dire quand n égal 0) dénote s lui même.

donnée. $t\sigma = \tilde{r}(1(\bot))+1(0(\bot))$ *est un terme clos et* $\mathcal{M}_R(t\sigma) = \{0(0(1(\bot)))\}$ *puisque* $\mathcal{M}_R(x\sigma) = \{1(\bot)\}$ *alors* $\mathcal{M}_R(\tilde{r}(x\sigma)) = \{0(1(\bot))\}$ *et finalement* $\mathcal{M}_R(t\sigma)$ *est l'ensemble* $\{\, \tilde{+}(t_1, t_2) \mid t_1 \in \mathcal{M}_R(\tilde{r}(x\sigma))$ *et* $t_2 \in \mathcal{M}_R(y\sigma) \,\}$ *c'est à dire l'ensemble* $\{\tilde{+}(0(1(\bot)), 1(0(\bot)))\} = \{1(1(\bot))\}$.

Notons que le modèle du terme clos $c(\bot, \bot)+0(\bot)$ *est vide car le premier argument de* $\tilde{+}$ *n'a jamais en tête le symbole de constructeur* c.

Un CTRS R sera dit correspondant à une relation r si il existe $f \in \mathcal{F}$ tel que $f(t_1, \ldots, t_n) \to_R^* t$ pour tout $(t_1, \ldots, t_n, t) \in r$.

7.3 Equations de joignabilité et formules

Définition 7.3.1. *Soit R un CTRS. Une* équation de joignabilité du second ordre $s \downarrow_R^? t$ *est une équation telle que s et t sont des termes du second ordre. Une* équation de joignabilité du premier ordre *est une équation $s \downarrow_R^? t$ telle que ni s ni t ne contiennent de variable du second ordre.*

Définition 7.3.2. *Soit R un CTRS, et $s \downarrow_R^? t$ une équation de joignabilité du second ordre. Une substitution close de données σ est une solution de $s \downarrow_R^? t$ si et seulement si $s\sigma$ et $t\sigma$ sont des termes clos et $\mathcal{M}_R(s\sigma) \cap \mathcal{M}_R(t\sigma) \neq \emptyset$.*

Soit R un CTRS, les formules de R-joignabilité du second ordre sont définies par la grammaire suivante :

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

où $s \downarrow_R^? t$ est une équation de joignabilité du second ordre et x est une variable du premier ordre.

Définition 7.3.3. *Une formule de R-joignabilité qui ne contient aucune variable du second ordre est appellée une* formule de R-joignabilité du premier ordre *. Une formule de R-joignabilité où les variable du second ordre n'apparaissent pas sous une négation est appellée* formule positive de R-joignabilité du second ordre.

L'ensemble des solutions d'une formule de R-joignabilité est définie comme suit

- $SOL(s \downarrow_R^? t) = \{\, \sigma \mid Dom(\sigma) = Var(s \downarrow_R^? t), \mathcal{M}_R(s\sigma) \cap \mathcal{M}_R(t\sigma) \neq \emptyset \,\}$
- $SOL(\neg e) = \{\, \sigma \mid Dom(\sigma) = Var(e), \sigma \notin SOL(e) \,\}$
- $SOL(e_1 \wedge e_2) = \{\, \sigma \mid Dom(\sigma) = Var(e_1 \wedge e_2), \sigma|_{Var(e_1)} \in SOL(e_1), \sigma|_{Var(e_2)} \in SOL(e_2) \,\}$
- $SOL(e_1 \vee e_2) = \{\, \sigma \mid Dom(\sigma) = Var(e_1 \vee e_2), \sigma|_{Var(e_1)} \in SOL(e_1)$ ou $\sigma|_{Var(e_2)} \in SOL(e_2) \,\}$
- $SOL(\exists x e) = \{\, \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \exists \sigma' \in SOL(e), \sigma = \sigma'|_{Var(e)\setminus\{x\}} \,\}$

– $SOL(\forall xe) = \{\, \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \forall t \in \mathcal{T}(\mathcal{C}), \sigma \cup \{x \mapsto t\} \in SOL(e), \sigma = \sigma'|_{Var(e)\setminus\{x\}} \,\}$

Exemple 7.3.4. *Considérons une fois de plus les CTRS de l'exemple 4.1.4 et la formule de joignabilité du second ordre suivante* $\forall x \exists y\, \neg(y + y \downarrow^?_R x) \vee X(x) \downarrow^?_R true$.

Cette formule est positive. Une solution de cette formule instancie la variable X *à la relation* $\tilde{e} = \{\, (n, true) \mid n \text{ est un nombre paire} \,\}$. *Notons que toute relation* $\tilde{e'}$ *telle que* $\tilde{e'} \supseteq \tilde{e}$ *est également une solution de la formule car c'est une implication et non une équivalence.*

L'ensemble des solutions pour les variables du second ordre peut être infini comme dans l'exemple 7.3.4, et dans le cas général ces solutions peuvent être incomparables. Cela arrive lorsqu'une même variable du second ordre apparaît des deux cotés du connecteur $\vee$ comme dans la formule $\forall x \exists y\, \neg(y + y \downarrow^?_R x) \vee (X(x) \downarrow^?_R true \vee X(x) \downarrow^?_R 0)$. $\tilde{e}$ de l'exemple 7.3.4 est une solution aussi bien que $\tilde{e''} = \{\, (n, 0) \mid n \text{ est un nombre pair} \,\}$ mais $\tilde{e}$ et $\tilde{e''}$ ne peuvent être comparées. Par conséquent résoudre des équations de joignabilité du second ordre dans le cas général reviendrait à représenter de manière finie des ensembles infinis de relations, en d'autre termes des ensembles infinis d'ensembles infinis. Les cas où une telle représentation peut être obtenue sont extrêmements rares et nécéssiteraient de très fortes restrictions sur le CTRS. C'est pourquoi on se restreint aux formules positives du second ordre d'une part, et d'autre part, nous décrivons un algorithme de décision qui retourne une solution si la formule est satisfiable et l'ensemble vide sinon.

7.4 Décision des Formules Positives du Second Ordre

L'algorithme repose sur les idées suivantes : une formule positive du second ordre est satisfiale si et seulement si sa partie du premier ordre est satisfiable. L'algorithme a donc pour but de séparer la partie du premier ordre et les équations du second ordre, ensuite de résoudre la partie du premier ordre de la formule puis de calculer une instance pour les variables du second ordre à partir des solutions de la partie du premier ordre. L'algorithme consiste en deux étapes principales. La première étape aplatit les équations de la formule. La seconde transforme le résultat en sa forme normale disjonctive prenexe.

Les règles de la table 7.1 sont utilisées pour transformer n'importe quelle équation de joignabilité du second ordre à une formule équivalente dont les équations sont de la forme $f(\vec{x}) \downarrow^?_R y$ où $f \in \mathcal{F} \cup \mathcal{C} \cup \mathcal{X}$ et $\vec{x}, y$ sont des variables du premier ordre. De telles équations sont appelées *équations de joignabilité*. Une formule contenant seulement des équations aplaties est appelée une *formule aplatie*.

$$\frac{\top}{\delta(y) = \langle x, x \downarrow^?_R y\rangle} \quad \text{si } x, y \in \mathit{Var} \text{ et } x \neq y$$

$$\frac{\delta(t_1) = \langle x_1, e_1\rangle \quad \cdots \quad \delta(t_n) = \langle x_n, e_n\rangle}{\delta(f(t_1, \ldots, t_n)) = \langle x, \exists x_1 \ldots x_n (f(x_1, \ldots, x_n) \downarrow^?_R x \wedge \bigwedge_{1 \leq i \leq n} e_i)\rangle}$$

si $f \in \mathcal{F} \cup \mathcal{X}$ et $x \notin \bigcup_{1 \leq i \leq n} \mathit{Var}(e_i)$ et les x_is sont différents deux à deux

$$\frac{\delta(s) = \langle x_s, e_s\rangle \qquad \delta(t) = \langle x_t, e_t\rangle}{\delta(s \downarrow^?_R t) = \exists x_s, x_t \; x_s \downarrow^?_R x_t \wedge e_s \wedge e_t} \quad \text{si } x_s, x_t \in \mathit{Var} \text{ et } x_s \neq x_t$$

TAB. 7.1 – Applatir les équations du second ordre

Notons que pour un terme du second ordre t, $\delta(t)$ est unique à renomage des variables du premier ordre, par conséquent la condition concernant la deuxième règle sur les x_is n'est pas une restriction. De plus chaque e_i contient le x_i correspondant donc x est différent de tous les x_is.

δ peut être étendu pour aplatir les formules du second ordre de la manière suivante $\delta(\neg e) = \neg\delta(e)$, $\delta(e_1 \wedge e_2) = \delta(e_1) \wedge \delta(e_2)$, $\delta(e_1 \vee e_2) = \delta(e_1) \vee \delta(e_2)$, $\delta(\exists x\, e) = \exists x\, \delta(e)$ et $\delta(\forall x\, e) = \forall x\, \delta(e)$.

Lemme 7.4.1. *Soit R un CTRS et s un terme du second ordre tel que $\delta(s) = \langle x, e\rangle$ et σ une substitution close de donnée $s\sigma$ est un terme clos. Si $t \in \mathcal{M}(s\sigma)$ alors $\sigma \cup \{x \mapsto t\} \in SOL(e)$.*

Démonstration. La preuve est faite par induction sur la profondeur de s. Si s est une variable alors e est l'équation $x \downarrow^?_R s$. Soit σ une substitution close de données dont le domaine ne contient pas x et telle que $s\sigma$ est un terme de données. $\{s\sigma\} = \mathcal{M}(s\sigma)$ car $s\sigma \in \mathcal{T}(\mathcal{C})$. Par conséquent $\sigma \cup \{x \mapsto s\sigma\}$ est une solution de $x \downarrow^?_R s$. Supposons que s est de la forme $f(s_1, \ldots, s_n)$, $\delta(s_i) = \langle x_i, e_i\rangle$. $\delta(s)$ est alors $\langle x, e\rangle$ où $e = \exists x_1 \ldots x_n (f(x_1, \ldots, x_n) \downarrow^?_R x \wedge \bigwedge_{1 \leq i \leq n} e_i)$. Soit $t \in \mathcal{M}(s)$ et $\sigma_i = \sigma|_{\mathit{Var}(s_i)}$. Par définition il existe $t_i \in \mathcal{M}(s_i\sigma_i)$ et $t \in \mathcal{M}(f(t_1, \ldots, t_n))$. Par hypothèse d'induction $\sigma_i \cup \{x_i \mapsto t_i\} \in SOL(e_i)$. Soit $\sigma' = \bigcup 1 \leq i \leq n(\sigma_i \cup \{x_i \mapsto t_i\})$. $f(x_1, \ldots, x_n)\sigma' = f(t_1, \ldots, t_n)$ par conséquent $\sigma' \cup \{x \mapsto t\}$ est une solution de $f(x_1, \ldots, x_n) \downarrow^?_R x$, alors $\sigma' \cup \{x \mapsto t\}$ est une solution de $f(x_1, \ldots, x_n) \downarrow^?_R x \wedge \bigwedge_{1 \leq i \leq n} e_i$. $\sigma'|_{Dom(\sigma') \setminus \{x_1, \ldots, x_n\}} = \sigma$ d'où $\sigma \cup \{x \mapsto t\} \in SOL(e)$. □

Lemme 7.4.2. *Soit R un CTRS et F une formule de joignabilité du second ordre. On a $SOL(F) = SOL(\delta(F))$.*

Démonstration. Considérons une formule contenant une seule équation $s \downarrow^?_R t$. $\delta(s \downarrow^?_R t) = \langle \exists x_s, x_t, s \downarrow^?_R x_s \wedge t \downarrow^?_R x_t \wedge x_s \downarrow^?_R x_t\rangle$ avec $\delta(s) = \langle x_s, s \downarrow^?_R x_s\rangle$ et $\delta(t) = \langle x_t, t \downarrow^?_R x_t\rangle$. Si $\sigma \in SOL(s \downarrow^?_R t)$, alors il existe un terme u tel

que $u \in \mathcal{M}(s\sigma)$ et $u \in \mathcal{M}(t\sigma)$, du lemme 7.4.1 on sait que $\sigma \cup \{x_s \mapsto s\} \in SOL(s \downarrow_R^? x_s)$ et $\sigma \cup \{x_t \mapsto t\} \in SOL(t \downarrow_R^? x_t)$ donc $\sigma \cup \{x_s \mapsto s, x_t \mapsto t\} \in SOL(s \downarrow_R^? x_s \wedge t \downarrow_R^? x_t \wedge x_s \downarrow_R^? x_t)$ ce qui implique que $\sigma \in SOL(\delta(e))$.

Maintenant si $\sigma \in SOL(\delta(e))$ alors il existe $\sigma' = \sigma \cup \{x_s \mapsto u_s, x_t \mapsto u_t\}$ qui est une solution de $s \downarrow_R^? x_s \wedge t \downarrow_R^? x_t \wedge x_s \downarrow_R^? x_t$. On a $u_s \in \mathcal{M}(s\sigma')$ et $u_t \in \mathcal{M}(t\sigma')$. Comme ni s ni t ne contient x_s et x_t, on a $s\sigma' = s\sigma$ et $t\sigma' = t\sigma$. De plus $x_s\sigma' \downarrow_R^? x_t\sigma'$, alors nous avons $u_s = u_t$, de là on en conclut que $u_s \in \mathcal{M}(s\sigma) \cap \mathcal{M}(t\sigma)$ d'où $\sigma \in SOL(s \downarrow_R^? t)$.

Par une induction sur la structure de la formule F on prouve que $SOL(F) = SOL(\delta(F))$.

□

On étudie maintenant les propriétés des formules non quantifiées conjonctives aplaties car l'algorithme travaille sur la forme disjonctive normale prenexe.

Considérons une formule conjonctive F, de la forme $fo(\vec{y}) \wedge X_1(\vec{x}_1) \downarrow_R^? x_1 \wedge \cdots \wedge X_m(\vec{x}_m) \downarrow_R^? x_m$ où $fo(\vec{y})$ est une conjonction d'équations du premier ordre et $\vec{y}$ le vecteur des variables apparaissant dans ces équations.

Dans le but de simplifier les notations, nous supposons que l'ensemble des solutions des formules du premier odre $fo(\vec{y})$ est représenté par un programme logique $\mathcal{P}_{fo(\vec{y})}$ qui définit le prédicat $P_{fo(\vec{y})}$ d'arité $|\vec{y}|$ telle que $\mathcal{P}_{fo(\vec{y})} \models P_{fo(\vec{y})}(\vec{t})$ si et seulement si la substitution $\{\, \vec{y}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq |\vec{y}| \,\} \in SOL(fo(\vec{y}))$. Soit $SO(F)$ l'ensemble suivant de clauses $\{\, P_{X_i}(\vec{x}_i, x_i) \leftarrow P_{fo}(\vec{y}) \mid 1 \leq i \leq m \,\}$ et $\sigma_{SO(F)} = \{X_i \mapsto \tilde{r}_{X_i}^F\}$ où $\tilde{r}_{X_i}^F = \{\, (\vec{t}, t) \mid SO(F) \cup \mathcal{P}_{fo(\vec{y})} \models P_{X_i}(\vec{t}, t) \,\}$.

Lemme 7.4.3. *Soit F une formule positive conjonctive aplatie du second ordre. F est satisfiable si et seulement si $F\sigma_{SO(F)}$ est satisfiable.*

Démonstration. Il est évident que si $F\sigma_{SO(F)}$ est satisfiable alors F est satisfiable. Maintenant supposons que F est satisfiable. Cela veut dire que $fo(\vec{y})$ est satisfiable, par conséquent le modèle de $P_{fo}(\vec{y})$ n'est pas vide. Donc $SO(F)$ instancie chaque variable du second ordre de F par une relation non vide, $F\sigma_{SO(F)}$ est donc satisfiable car les variables du second order n'apparaissent que dans les équations triviales $X(\vec{x}) \downarrow_R^? x$. □

Le lemme suivant aide à caractériser les instances des variables du second ordre que l'on calcule. Il est utilisé pour prouver que notre algorithme donne une solution si et seulement si la formule entière est satisfiable. Ce résultat est également important lorsque l'on veut synthétiser un CTRS à partir d'une formule positive du second ordre.

Lemme 7.4.4. *Soit $F = fo(\vec{y}) \wedge X_1(\vec{x}_1) \downarrow_R^? x_1 \wedge \cdots \wedge X_m(\vec{x}_m) \downarrow_R^? x_m$ une formule conjonctive. Alors $\sigma_{SO(F)}$ est la plus petite solution de la formule du second ordre suivante $\forall \vec{z} fo(\vec{y}) \Rightarrow X_1(\vec{x}_1) \downarrow_R^? x_1 \wedge \cdots \wedge X_m(\vec{x}_m) \downarrow_R^? x_m$ où $\vec{z}$ est un vecteur composé de l'union des variables de $\vec{y}$, de $\vec{x}_i$ et x_i $(1 \leq i \leq m)$.*

Démonstration. La preuve de ce lemme est évidente car $\sigma_{SO(F)}$ est calculée à partir de l'ensemble de clauses $P_{X_i}(\vec{x}_i, x_i) \leftarrow P_{fo}(\vec{y})$ $(1 \leq i \leq n)$. □

On est maintenant prêt à décrire l'algorithme qui décide la satisfiabilité d'une formule positive du second ordre et donne une instance des variables du second ordre et l'ensemble des solutions correspondantes pour les variables du premier ordre quand la formule est satisfiable. Cet algorithme repose sur l'existence d'un algorithme qui résoud les formules de joignabilité du premier ordre et donne une représentation finie de leurs solutions.

Algorithme 2. *Soit $\mathcal{A}$ un algorithme pour résoudre les équations de joignabilité de premier ordre, R un CTRS et F une formule positive du second-ordre.*

1. *Calculer $\delta(F)$ la forme aplatie de F*
2. *Calculer F' équivalent à $\delta(F)$ et dans une forme normale disjonctive prenexe. Soit $C_1, \ldots, C_n$ les facteurs conjonctifs de F'.*
3. *Pour chaque C_i $(1 \leq i \leq n)$, utiliser $\mathcal{A}$ pour résoudre la partie du premier ordre de C_i à partir de laquelle $\sigma_{SO(C_i)}$ est déduit.*
4. *Soit $\sigma_{SO(F')} = \{ X_j \mapsto \tilde{r'}_{X_j} \mid X_j \in Var(F'), \tilde{r'}_{X_j} = \bigcup_{1 \leq i \leq n} \tilde{r}^{C_i}_{X_j} \}$. Utiliser $\mathcal{A}$ pour résoudre $F'\sigma_{SO(F')}$ en considérant les variables du second ordre comme des symboles de fonctions définis.*
5. *Si $SOL(F'\sigma_{SO(F')}) \neq \emptyset$ retourne $\sigma_{SO(F')}$ et les solutions de $F'\sigma_{SO(F')}$ sinon retourne $\emptyset$.*

Théorème 7.4.4.1. *Soit $\mathcal{A}$ un algorithme pour résoudre les équations de joignabilité du premier ordre, R un CTRS et F une formule positive du second ordre. F est satisfiable si et seulement si la sortie de l'Algorithme 2 n'est pas vide.*

Démonstration. Le lemme 7.4.2, nous dit que $SOL(F) = SOL(\delta(F))$ et par conséquent $SOL(F) = SOL(F')$.

Soit $C_1, \ldots, C_n$ les facteurs conjonctifs de F'. Chaque C_i est de la forme $fo_i(\vec{y}_i) \wedge so_i(\vec{x}_i, \vec{X}_i)$ où $fo_i(\vec{y}_i)$ est une conjonction d'équation de joignabilité du premier ordre dont les variables sont celles de $\vec{y}_i$ et $so_i(\vec{x}_i, \vec{X}_i)$ est une conjonction d'équations de joignabilitées aplaties du second ordre de la forme $X(\vec{x}) \downarrow^?_R x$ où $X \in X_i$ et les variables de $\vec{x}$ et x apparaissent dans $\vec{x}_i$.

L'algorithme $\mathcal{A}$ peut calculer une représentation finie des solutions de $fo_i(\vec{y}_i)$ qui donne les définitions de $\sigma_{SO(C_i)}$. Le Lemme 7.4.4, nous informe que $\sigma_{SO(C_i)}$ est le plus petit modèle de la formule $\forall \vec{z_i}, fo_i(\vec{y}_i) \Rightarrow so_i(\vec{x}_i, \vec{X}_i)$. Par définition de $\sigma_{SO(F')}$, on a $X\sigma_{SO(F')} \supseteq X\sigma_{SO(C_i)}$ pour n'importe quelle variable de F' et n'importe quel C_i $1 \leq i \leq n$. Donc, $\sigma_{SO(F')}$ est un modèle des formules $\forall \vec{z_i}, fo_i(\vec{y}_i) \Rightarrow so_i(\vec{x}_i, \vec{X}_i)$ ce qui signifie que $\forall \vec{z_i}, fo_i(\vec{y}_i) \Rightarrow so_i(\vec{x}_i, \vec{X}_i)\sigma_{SO(F')}$ et de manière évidente
$\forall \vec{z_i}, fo_i(\vec{y}_i) \Rightarrow fo_i(\vec{y}_i) \wedge so_i(\vec{x}_i, \vec{X}_i)\sigma_{SO(F')}$.

Alors on peut en déduire que n'importe quel modèle de $fo_1(\vec{y}_1) \vee \ldots fo_n(\vec{y}_n)$ est un modèle de $(fo_1(\vec{y}_1) \wedge so_1(\vec{x}_1, \vec{X}_1)\sigma_{SO(F')}) \vee \cdots \vee (fo_n(\vec{y}_n) \wedge so_n(\vec{x}_n, \vec{X}_n)\sigma_{SO(F')})$. Supposons que $F' = Q\, C_1 \vee \cdots \vee C_n$ où Q représente les quantifications de la formule. Si F est satisfiable, F' est également satisfiable de même que $Q\, fo_1(\vec{y}_1) \vee \ldots fo_n(\vec{y}_n)$. A partir de la remarque ci-dessus on peut déduire que $Q\, (fo_1(\vec{y}_1) \wedge so_1(\vec{x}_1, \vec{X}_1)\sigma_{SO(F')}) \vee \cdots \vee (fo_n(\vec{y}_n) \wedge so_n(\vec{x}_n, \vec{X}_n)\sigma_{SO(F')})$ est également satisfiable et par conséquent l'algorithme $\mathcal{A}$ calcule un ensemble de solutions non vide.

D'autre part, si $\mathcal{A}$ calcule un ensemble non vide de solutions pour $F'\sigma_{SO(F')}$, cela veut dire que $F'\sigma_{SO(F')}$ est satisfiable, donc F' et F sont également satisfiables.

□

Exemple 7.4.5. *Considérons une fois de plus la formule de l'exemple 7.3.4, c'est à dire* $\forall x \exists y\, \neg(y + y \downarrow^?_R x) \vee X(x) \downarrow^?_R true$.

La formule aplatie de cette formule est $\forall x \exists y\, \neg(y + y \downarrow^?_R x) \vee \exists z\, X(x) \downarrow^?_R z \wedge true \downarrow^?_R z$. *Sa forme disjonctive prenexe est* $\forall x \exists y \exists z\, \neg(y + y \downarrow^?_R x) \vee X(x) \downarrow^?_R z \wedge true \downarrow^?_R z$.

On a les facteurs conjonctifs $C_1 = \neg(y + y \downarrow^?_R x)$ *et* $C_2 = X(x) \downarrow^?_R z \wedge true \downarrow^?_R z$. *Les solutions de* C_1 *sont les ensembles* $\{\, (y \mapsto n, x \mapsto m) \mid 2 \times n \neq m \,\}$ *et l'unique solution de* $true \downarrow^?_R z$ *est* $\{z \mapsto true\}$ *qui peut être représentée par le programme logique composée par l'unique clause* $P(true) \leftarrow$. *Les variables du second ordre de chaque facteur étant claculé à partir des solutions du premier ordre de ces facteurs on a* $SO_X^{C_2} = \{P_X(x, z) \leftarrow P(z)\}$ *ce qui signifie que* $\tilde{r}_X^{C_2} = \{\, t, true \mid t \in \mathcal{T}(\mathcal{C}) \,\}$. *La sortie de l'algorithme est alors* $X \mapsto \tilde{r}_X^{C_2}$ *car toutes les variables du premier ordre sont quantifiées.*

Cette solution n'est pas la plus petite mais l'instance pour X *garantie que chaque modèle de* $true \downarrow^?_R z$ *sont des modèles de* $X(x) \downarrow^?_R z$ *qui est nécéssaire pour la correction de l'algorithme.*

Pour synthétiser la fonction paire, on peut utiliser la propriété établie par le Lemme 7.4.4 et mettre la formule $\forall x \forall y\, y + y \downarrow^?_R x \wedge X(x) \downarrow^?_R true$ *en entrée de l'algorithme. De plus ce lemme établit que l'instance calculé pour* X *par l'algorithme est la plus petite solution de* $\forall x, \forall y y + y \downarrow^?_R x \Rightarrow X(x) \downarrow^?_R true$.

7.5 Synthèse de programme

On a montré au chapitre 6 qu'il existe un algorithme $\mathcal{A}$ qui produit un programme logique quasi-NGPRC-partagé quand la formule est satisfiable. Il serait intéressant d'être capable de donner le CTRS correspondant tel que l'instance des variables du second ordre soit exprimée dans le même formalisme que la relation définie par les symboles de fonctions du CTRS en entrée.

Cette transformation peut être utilisée pour synthétiser des programmes à partir de formules de positives du second ordre.

Définition 7.5.1. *Soit un atome A sans symbole de fonction de la forme $P(x_1, \ldots, x_{n-1}, x_n)$ alors $term(A) = f_P(x_1, \ldots, x_{n-1})$ et $equat(A) = f_P(x_1, \ldots, x_{n-1}) \downarrow_R x_n$. On étend equat à un ensemble d'atomes aplatis $\mathcal{G}$ de manière naturelle c'est à dire $equat(\mathcal{G}) = \{ equat(A) \mid A \in \mathcal{G} \}$.*

Exemple 7.5.2. *Soit l'atome $P(x_1, x_2, x_3)$ alors $term(P(x_1, x_2, x_3)) = f_P(x_1, x_2)$ et $equat(P(x_1, x_2, x_3)) = f_P(x_1, x_2) \downarrow^?_R x_2$.*

Soit $\mathcal{G}$ l'ensemble d'atomes $\{B(x_1, x_2), C(x_2, x_1)\}$, alors $equat(\mathcal{G}) = \{f_B(x_1) \downarrow^?_R x_2, f_C(x_2) \downarrow^?_R x_1\}$.

Définition 7.5.3. *Soit $\mathcal{C} = P(t_1, \ldots, t_{n-1}, C[x_1, \ldots, x_m]) \leftarrow \mathcal{G}, \mathcal{G}'$ une clause RP-close où $\mathcal{G} = \{ P(\vec{x}, x) \mid x \in \{x_1, \ldots, x_m\}$ et x apparaît une fois dans $\mathcal{G}, \mathcal{G}' \}$*

$\mathcal{RR}(\mathcal{C}) = f_P(t_1, \ldots, t_{n-1}) \rightarrow C[x_1, \ldots, x_m]\sigma \leftarrow equat(\mathcal{G}')$ où $\sigma = \{ x \mapsto term(P(\vec{x}, x)) \mid P(\vec{x}, x) \in \mathcal{G} \}$.

Pour une programme logique RP-clos $\mathcal{P}$, soit $\mathcal{RR}(\mathcal{P})$ représentant le CTRS consistant des règles $\{ \mathcal{RR}(H \leftarrow \mathcal{G}) \mid H \leftarrow \mathcal{G} \in \mathcal{P} \}$.

Par exemple, $P_f(s(x_1, y), s(x_2, y), s(z_1, z_2)) \leftarrow P_f(x_2, x_1, z_1), P_f(x_1, x_1, x_2)$ est transformé en $f(s(x_1, y), s(x_2, y)) \rightarrow s(f(x_2, x_1), z_2) \leftarrow f(x_1, x_1) \downarrow_R x_2$.

Lemme 7.5.4. *Si $\mathcal{P}$ est un programme logique RP-clos alors $\mathcal{RR}(\mathcal{P})$ est un CTRS pseudo-régulier.*

La preuve de ce lemme est essentiellement faite par la correspondance entre les atomes du corps d'une clause RP-close et les symboles de fonction dans les rhs des règles conditionnelles correspondantes.

Démonstration. Considérons une clause $\mathcal{C} = H \leftarrow \mathcal{G}, \mathcal{G}'$ d'un programme RP-clos $\mathcal{P}$ où $H = P(t_1, \ldots, t_{n-1}, C[x_1, \ldots, x_m])$, $\mathcal{G}$ est l'ensemble d'atomes $P(\vec{u}, x)$ tels que $x \in \{x_1, \ldots, x_m\}$ et x n'apparaît qu'une seule fois dans $\mathcal{G}, \mathcal{G}'$. La règle de réécriture correspondante est $f_P(t_1, \ldots, t_{n-1}) \rightarrow C[x_1, \ldots, x_m]\sigma \leftarrow equat(\mathcal{G}')$ où $\sigma = \{ x \mapsto term(P(\vec{u}, x)) \mid P(\vec{u}, x) \in \mathcal{G} \}$. A partir de la définition des RP-clos on sait que, on sait que

- Aucune variable existentielle n'apparaît dans $\mathcal{C}$ ce qui signifie que $Var(\mathcal{G}, \mathcal{G}') \subseteq Var(H)$ par conséquent $\bigcup_{1 \leq i \leq m} Var(x_i\sigma) \cup Var(equat(\mathcal{G}') \subseteq Var(f_P(t_1, \ldots, t_{n-1}) \cup Var(C)$.
- Il existe une application π de Var à $\mathbb{N}^+$ tel que si $\pi(x) = p$ alors toutes les occurences de x sont à la position p dans $t_1, \ldots, t_n$ et C
- Toute les variables d'un atome donné A de $\mathcal{G}, \mathcal{G}'$ ont la même image par π par conséquent toutes les variables de chaque équation de $equat(\mathcal{G}')$ ont la même image par π. De plus, toutes les variables de $x\sigma$ ont la même image que x par π i.e. si $x \in \{x_1, \ldots, x_n\}$, la position de $x\sigma$ dans $C[x_1, \ldots, x_m]\sigma$.

Alors on peut en conclure que $\mathcal{RR}(\mathcal{C})$ est une règle conditionnelle pseudo-régulière. □

Lemme 7.5.5. *Si $\mathcal{P}$ est un programme logique RP-clos alors $\mathcal{LP}(\mathcal{RR}(\mathcal{P})) = \mathcal{P}$ modulo le renomage des variables introduites dans les symboles de prédicats auxiliaires.*

Démonstration. Dans cette preuve on considère que P_{f_P} est une notation pour le symbole de prédicat P. Soit $\mathcal{P}$ un programme logique RP-clos et $\mathcal{C}$ la clause de $\mathcal{P}$. $\mathcal{C}$ est de la forme $P(t_1, \ldots, t_{n-1}, C[x_1, \ldots, x_m]) \leftarrow \mathcal{G}, \mathcal{G}'$.

$\mathcal{RR}(\mathcal{C}) = f_P(t_1, \ldots, t_{n-1}) \rightarrow C[x_1, \ldots, x_m]\sigma \leftarrow equat(\mathcal{G}')$ où $\sigma = \{\, x \mapsto term(P(\vec{u}, x)) \mid P(\vec{u}, x) \in \mathcal{G} \,\}$. Pour chaque atome $P(\vec{u}, u)$ de $\mathcal{G}'$, $equat(P(\vec{u}, u)) = f_P(\vec{u}) \downarrow_R u$. De la remarque 6.2.0.2, chaque équation $f_P(\vec{u}) \downarrow_R u$ est transformée en $P_{f_P}(\vec{u}, u)$ (i.e $P(\vec{u}, u)$).

Chaque atome $P(\vec{u}, x)$ de $\mathcal{G}$ est transformée dans le terme $f_P(\vec{u})$. A partir de la table 6.1 $f_P(\vec{u}) \rightsquigarrow \langle x, P_{f_P}(\vec{u}, x)\rangle$, par conséquent $C[x_1, \ldots, x_m]\sigma \rightsquigarrow \langle C[x_1, \ldots, x_m], \mathcal{G}\rangle$. D'où $f_P(t_1, \ldots, t_{n-1}) \rightarrow C[x_1, \ldots, x_m]\sigma \leftarrow equat(\mathcal{G}') \rightsquigarrow \mathcal{C}$. □

Théorème 7.5.5.1. *Soit $\mathcal{P}$ un programme logique RP-clos et $P(\vec{t}, t)$ un atome clos. $\mathcal{P} \models P(\vec{t}, t)$ si et seulement si $f_P(\vec{t}) \rightarrow^* t$.*

Démonstration. D'après le lemme 7.5.5, on sait que $\mathcal{LP}(\mathcal{RR}(\mathcal{P})) = \mathcal{P}$. Le terme $f_P(\vec{t})$ est transformé par $\rightsquigarrow$ en $\langle x, P(\vec{t}, x)\rangle$ car tous les termes de $\vec{t}$ sont clos de données et on considère P_{f_P} comme une notation pour P. D'après le Théorème 6.2.1.1 on sait que $f_P(\vec{t}) \rightarrow^* t$ si et seulement si t est un terme clos de données, $\mathcal{LP}(\mathcal{RR}(\mathcal{P})) \models P(\vec{t}, x)\mu$ et $t = x\mu$, en d'autres mots si et seulement si $\mathcal{P} \models P(\vec{t}, t)$. □

7.6 Travaux reliés

Dans ce chapitre, on a décrit un algorithme pour décider de la satisfiabilité de formules de joignabilité du second ordre quand il existe un algorithme qui calcule une représentation décidable des formules du premier ordre correspondant. Cet algorithme a été appliqué à la classe des formules pseudo-régulières du second ordre. Quand la formule est satisfiable, l'algorithme exprime des instances des variables du second ordre par un CTRS. Ce résultat donne un mécanisme pour synthétiser des CTRS qui peuvent être considérés comme des programmes logiques fonctionnels.

Les théories du second ordre avec des variables du second ordre apparaissant dans les termes ont été étudiées dans le contexte de l'unification du second ordre (voir [58, 80] par exemple). Nos équations de joignabilité sont des équations d'unifiabilité quand le CTRS est confluent par conséquent résoudre des équations pseudo-régulières du second ordre demande de résoudre l'unification du second order modulo un CTRS.

La synthèse de programme à partir d'une spécification a déjà été étudiée. Dans le contexte des programmes fonctionnels par exemple [35, 3] utilise les systèmes de réécritures pour donner un modèle de calcul pour la programmation fonctionnelle. Les spécifications de la fonction à synthétiser est l'ensemble des équations qui peuvent être vues comme des formules conjonctives positives. La logique du second ordre a été utilisée dans [21] pour la spécification dans le but de synthétiser des programmes logiques mais des heuristiques sont utilisées et le résultat est partiellement correct tandis que notre méthode est exacte (i.e. on obtient une instance correcte pour les variables du second ordre). Dans la plus part des cas la synthèse de programmes (fonctionels ou non) utilise des méthodes d'induction avec des méthodes déductives pour trouver des résultats partiellement corrects. En conséquence de telles méthodes génerent des programmes plus généraux que les nôtres. Dans notre travail, de telles solutions partielles pourraient être générées en utilisant des approximations durant le calcul des opérations sur les relations régulières.

Chapitre 8

Approximations pour des tests d'inconsistance

8.1 Introduction

Dans ce chapitre je présente des travaux qui n'ont pas encore été publiés, et qui représentent ma dernière année de recherche de ma thèse. Jusqu'à présent on s'était concentré sur des méthodes exactes afin de calculer des opérations sur les langages de n-uplet d'arbres, on s'intéresse maintenant à des méthodes approximatives. Les approximations sont utilisées par exemple dans le but de simplifier un calcul, de permettre au calcul de terminer ou encore de donner un intervalle de confiance. Le problème est alors de donner une approximation suffisement fine et adaptée au problème pour qu'elle se révèle efficace.

De manière générale deux sortes d'approximations sont utilisées en vérification. La première est la sous-approximation. Sous approximer un programme P permet d'assurer que le modèle de la sous-approximation appartient au modèle de P. La seconde est la sur-approximation qui permet d'assurer que le modèle de la sur-approximation d'un programme P contient le modèle de P. Ces méthodes se révèlent particulièrement pertinentes lorsqu'elles s'appliquent à la vérification de protocoles cryptographiques. En effet les sous-approximations permettent de trouver des attaques sur les protocoles [8], mais ne pas trouver d'attaque ne garantit pas qu'il y en ait aucune. Les sur-approximations permettent de prouver des propriétés telles que la sureté ou l'authentification [8, 44, 12] en détectant le vide de l'approximation, mais lorsqu'on a pas le vide on n'a pas forcément une attaque.

L'idée de l'utilisation de programmes logiques pour représenter les protocoles de sécurité n'st pas nouvelle, les protocoles peuvent être analysés symboliquement avec des prouveurs généraux [34, 100] ou avec des algo-

rithmes et outils spécifiques [30, 12, 11]. On propose dans ce chapitre un test d'inconsistance de requêtes sur les programmes logiques définis, basé sur les techniques de transformations de programmes logiques vues en section 3.4. La première approximation proposée s'inspire de celle originellement introduite dans [65]. L'idée de base provient de la création d'un nombre fini de cs-clauses en bornant les possibilités d'acroissement infini au niveau de la hauteur des termes dans les clauses et du nombre d'atomes dans une même composante de la décomposition maximale en composante sans variables communes.

On se base sur les cs-programmes car le test du vide de leur modèle peut être décidé en temps linéaire. Afin de tester l'inconsistance on définit un cs-programme qui sur-approxime les solutions du problème initial remplaçant les sous-termes hors des bornes par des nouvelles variables. Mais cette première approximation n'est pas adaptée aux protocoles cryptographiques. Par contre en utilisant des prédicats d'approximations associées à ces variables on montre que notre technique peut être appliquée avec succès à la vérification de protocoles cryptographiques et en particulier au protocole de Needham-Schroder(-Lowe) avec un nombre non borné de sessions.
.

Dans la section 8.2 on voit une nouvelle technique de transformation de programmes qui nous permet de déplier successivement une même clause. La section 8.3 montre la première sur-approximation ainsi qu'une optimisation. La section 8.4 montre une technique de sur-approximation que l'on appliquera avec succès à l'exemple de Needham-Schroder(-Lowe) décrit en section 8.5.

8.2 Transformation de base

Dans ce chapitre on va utiliser une variante de la $\Rightarrow$-dérivation. A partir de cette variante on va définir des méthodes d'approximations. Cette procédure à l'avantage de laisser la possibilité de déplier une même clause plusieurs fois de suite. Dans ce chapitre on présente les clauses sous la forme prolog car c'est celle-ci que l'on a utilisé pour l'implantation. La différence est que les variables sont en lettre capitales et les noms de prédicats sont en minuscules.

Cette procédure prend comme entrée, un programme logique $\mathcal{P}$ et un ensemble de définitions I compatible avec $\mathcal{P}$. Quand elle termine, elle produit un cs-programme $\mathcal{P}'$ telle que pour toute définition $p(\vec{X}) \leftrightarrow \mathcal{B}$ de I, $\mathcal{P}' \models p(\vec{X})\sigma$ si et seulement si il existe σ' tel que $\sigma'\backslash_{\vec{X}} = \sigma$ et $\mathcal{P} \models \mathcal{B}\sigma'$.

Cette procédure est définie au moyen de deux règles (qui ont le même rôle que pour la $\Rightarrow$-dérivation) qui transforment l'état $\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}} \rangle$ où $\mathcal{P}$ est l'entrée du programme logique, $\mathcal{D}_{\text{done}}$ sont les définitions introduites

par le processus et utilisées pour simplifier les clauses, $\mathcal{C}_{\text{new}}$ contient les définitions et les clauses à traiter, et $\mathcal{C}_{\text{out}}$ sont les clauses générées jusque là.

On écrit $S \Rightarrow_s^U S'$ si S' est un état obtenu à partir de l'état S par l'application *Dépliage* et $S \Rightarrow_s^D S'$ si il est obtenu par l'application d'*Introduction de définition* définis plus bas. $\Rightarrow_s$ signifie soit $\Rightarrow_s^U$ ou soit $\Rightarrow_s^D$. Une séquence $S_1 \Rightarrow_s S_2 \cdots \Rightarrow_s S_n$ est appellée une $\Rightarrow_s$-dérivation, il peut être abrégé en $S_1 \stackrel{*}{\Rightarrow}_s S_n$.

Un *état initial* est de la forme $\langle \mathcal{P}, \mathcal{D}, \mathcal{D}, \emptyset \rangle$ où $\mathcal{D}$ est compatible avec $\mathcal{P}$. Un *état final* est de la forme $\langle \mathcal{P}, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$. $\mathcal{P}$ et $\mathcal{D}$ sont appellés les entrées de la dérivation, $\mathcal{P}'$ est sa sortie. Une dérivation est *complète* si son dernier état est final. Dans la suite $\leftarrow ?$ représente soit $\leftarrow$ soit $\leftrightarrow$.

DÉPLIAGE.

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

où $\mathcal{C}$ est l'ensemble de toutes les clauses $(L \leftarrow \mathcal{R} \cup \mathcal{B})\mu$ telles que $H \leftarrow \mathcal{B}$ est une clause dans $\mathcal{P}$, et telle que l'unificateur le plus général μ de A et H existe.

Remarquons que soit une clause ou une définition peut être dépliée pour produire un ensemble de clause.

DEFINITION INTRODUCTION.

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

où $\mathcal{B}_1, \ldots, \mathcal{B}_k$ est une décomposition maximale de $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ en composantes sans variables communes,

$$L_i = \begin{cases} L\eta^{-1} & \text{si } L \leftrightarrow \mathcal{B}_i)\eta \in \mathcal{D}_{\text{done}} \text{ pour un rennomage de} \\ & \text{var } \eta \\ & \text{et } \mathit{Var}(H) \cap \mathit{Var}(\mathcal{B}_i) = \mathit{Var}(L\eta) \cap \mathit{Var}(\mathcal{B}_i\eta) \\ P_i(x_1, \ldots, x_n) & \text{autrement, } \{x_1, \ldots, x_n\} \text{ les vars. de } \mathcal{B}_i. \end{cases}$$

pour $1 \leq i \leq k$ et un nouveau symbole de prédicat P_i, et où $\mathcal{D}$ est l'ensemble de tous les $L_i \leftrightarrow \mathcal{B}_i$ tels que L_i contient un nouveau symbole de prédicat.

Exemple 8.2.1. *Soit $\mathcal{P}$ le programme qui définit les relations d'addition et inférieur.*

$$\begin{array}{ll} \mathit{add}(0, X, X) \leftarrow & \mathit{add}(s(X), Y, s(Z)) \leftarrow \mathit{add}(X, Y, Z) \\ \mathit{inf}(0, X) \leftarrow & \mathit{inf}(s(X), s(Y)) \leftarrow \mathit{inf}(X, Y) \end{array}$$

et le but $g = \mathit{res}(X, Z) \leftrightarrow \mathit{inf}(X, Z), \mathit{add}(X, Y, Z)$. La figure ci-dessus résume une $\Rightarrow$-dérivation avec $\mathcal{P}$ et g comme entrée.

$\mathcal{D}_{\text{done}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$res(X,Z) \leftrightarrow$ $\quad inf(X,Z),$ $\quad add(X,Y,Z)$	$res(X,Z) \leftrightarrow$ $\quad inf(X,Z),$ $\quad add(X,Y,Z)$		
	$res(0,0) \leftarrow inf(0,0)$ $res(s(X),s(Z)) \leftarrow$ $\quad inf(s(X),s(Z)),$ $\quad add(X,Y,Z)$		U
	$res(0,0) \leftarrow$ $res(s(X),s(Z)) \leftarrow$ $\quad inf(s(X),s(Z)),$ $\quad add(X,Y,Z)$		U
	$res(s(X),s(Z)) \leftarrow$ $\quad inf(s(X),s(Z)),$ $\quad add(X,Y,Z)$	$res(0,0) \leftarrow$	D
$res'(X,Z) \leftrightarrow$ $\quad inf(s(X),s(Z)),$ $\quad add(X,Y,Z)$	$res'(X,Z) \leftrightarrow$ $\quad inf(s(X),s(Z)),$ $\quad add(X,Y,Z)$	$res(s(X),s(Z)) \leftarrow$ $\quad res'(X,Z)$	D
	$res'(X,Z) \leftarrow$ $\quad inf(X,Z), add(X,Y,Z)$		U
		$res'(X,Z) \leftarrow res(X,Z)$	D

FIG. 8.1 – Une $\Rightarrow$-dérivation

Théorème 8.2.1.1. *Soit $\mathcal{P}$ un programme logique et $\mathcal{D}$ l'ensemble des définitions compatibles avec $\mathcal{P}$. Si $\langle \mathcal{P}, \mathcal{D}, \mathcal{D}, \emptyset \rangle \stackrel{*}{\Rightarrow}_s \langle \mathcal{P}, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, alors $\mathcal{P}'$ est un cs-programme qui a la propriété que $\mathcal{M}(\mathcal{P}')|_p = \mathcal{M}(\mathcal{P} \cup \mathcal{D})|_p$ pour tout les symboles de prédicats p définis par $\mathcal{D}$.*

Démonstration. $\mathcal{P}'$ est un cs-programme par construction : les sous-ensembles des atomes du corps partageant des variables ou contenant des symboles de fonctions sont remplacés par un simple atome linéaire sans symbole de fonction. L'équivalence sémantique des programmes logiques $\mathcal{P} \cup \mathcal{D}$ et $\mathcal{P}'$ est obtenu grâce à des résultats généraux sur les transformations de programmes logique (voir par exemple [1]).

Soit l'état final $\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \emptyset, \mathcal{C}_{\text{out}} \rangle$ obtenu à partir de l'entrée $\mathcal{P}$, $\mathcal{D}$. Par construction $\mathcal{P}$ et $\mathcal{C}_{\text{out}}$ n'ont aucun symbole de prédicat en commun, alors l'invariant nous informe que $\mathcal{M}(\mathcal{P} \cup \mathcal{D}_{\text{done}}) = \mathcal{M}(\mathcal{C}_{\text{out}})$. Les définitions de $\mathcal{D}_{\text{done}}$ sont telles que le symbole de prédicat de la tête est l'un de $\mathcal{C}_{\text{out}}$ tandis que ceux apparaissant dans le corps proviennent de $\mathcal{P}$, de plus les prédicats des têtes sont deux à deux distincts, alors la sémantique des définitions de $\mathcal{D}_{\text{done}}$ ne sont pas inter-dépendante. Comme $\mathcal{D} \subseteq \mathcal{D}_{\text{done}}$, on a que $\mathcal{M}(\mathcal{P} \cup \mathcal{D}_{\text{done}})_p = \mathcal{M}(\mathcal{P} \cup \mathcal{D})_p$ pour chaque p défini par $\mathcal{D}$, donc $\mathcal{M}(\mathcal{P} \cup \mathcal{D})_p = \mathcal{M}(\mathcal{C}_{\text{out}})_p$.

□

Parmi les propriétés intéressantes des cs-programmes, on se focalise ici sur la suivante.

Propriété 8.2.2. *Il est décidable de vérifier si un prédicat d'un cs-programme a un modèle vide.*

Démonstration. Comme une cs-clause $H \leftarrow \mathcal{B}$ possède un corps linéaire sans symbole de fonction, n'importe quel atome $p(\vec{X})$ de $\mathcal{B}$ s'unifie avec une tête de clause de $\mathcal{P}$ de la forme $p(\vec{t})$. De plus le mgu n'instancie pas toute les variables de $\mathcal{B} \setminus \{p(\vec{X})\}$. Par conséquent, le prédicat d'un cs-programme $\mathcal{P}$ a un modèle non vide ssi il appartient à l'ensemble $Mark(\mathcal{P})$ récursivement défini comme suit :

- $p \in Mark(\mathcal{P})$ si il existe un fait $p(\vec{t}) \leftarrow$ dans $\mathcal{P}$
- $p \in Mark(\mathcal{P})$ si il existe une clause $p(\vec{t}) \leftarrow \mathcal{B}$ dans $\mathcal{B}$ et tout les prédicats de $\mathcal{B}$ sont dans $Mark(\mathcal{P})$

tous les prédicats qui ne sont pas dans $Mark$, ont un modèle vide. □

8.3 Approximation simple

Une première technique d'approximation que l'on peu utiliser, remplace de manière basique des sous-termes dans le corps de la clause de l'ensemble $\mathcal{C}_{\text{new}}$ par des variables existentielles dans le but d'atteindre une clause qui subsume l'originale. Ceci garantit l'obtention d'une sur-approximation.

Définition 8.3.1. *Soit $H \leftarrow \mathcal{B}$ et $H \leftarrow B'$ deux clauses de Horn. $H \leftarrow B'$ est dite une* approximation *de $H \leftarrow \mathcal{B}$ si il existe une substitution σ telle que $\mathcal{B} = \mathcal{B}'\sigma$.*

Par définition de l'approximation, le lemme suivant est vérifié de manière évidente.

Lemme 8.3.2. *Soit C et C' des clauses de Horn telles que C' est une approximation de C. $\mathcal{M}(C) \subseteq \mathcal{M}(C')$*

Les deux raisons pour lesquelles les $\Rightarrow$-dérivations ne terminent pas en général sont la croissance non bornée de la hauteur des atomes dans le corps de clauses de $\mathcal{C}_{\text{new}}$ d'un côté, et l'augmentation du nombre d'atomes partageants les variables de l'autre. Cette dernière raison est due à l'augmentation de soit du nombre de variables dupliquées soit du nombre maximal d'occurences d'une variable (ou les deux). Afin de définir un processus d'approximation qui termine toujours, il est suffisant de garantir que ces paramètres sont bornés dans les clauses d'approximations dans $\mathcal{C}_{\text{new}}$.

Définition 8.3.3. *Soit d et n deux entiers positifs, une* (d, n)-approximation *d'une clause C est une clause C' telle que les atomes du corps sont à une profondeur inférieure à d et les composants de la décomposition maximale du corps en composantes sans variables communes contient au plus n atomes.*

(D, N)-APPROXIMATION.

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

où $\mathcal{B}_1, \ldots, \mathcal{B}_k$ est une décomposition maximale de $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ en composantes sans variables communes, $\exists i$ tel que $\mathcal{B}_i$ est soit plus haut que d ou il contient plus de n atomes et $H \leftarrow \mathcal{B}'$ est une (d, n)-approximation de $H \leftarrow \mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_k$.

On écrit $S \Rightarrow^{(d,n)} S'$ si S' est un état obtenu à partir de l'état S en appliquant les règles (d, n)-Approximation. Remarquons que (d, n)-Approximation peut être appliqué seulement sur des clauses qui excèdent les bornes. On étend la notation $\Rightarrow_s$, $\stackrel{*}{\Rightarrow}_s$ et $\Rightarrow_s$-dérivation pour inclure $\Rightarrow_s^{(d,n)}$ étapes.

Une (d, n)-*approximation super-étape* ou (d, n)-*étape* en abrégé, est une $\Rightarrow_s$-dérivation $S \Rightarrow_s^* S'$ consistant en trois phases :

- Phase 1 : Un nombre fini d'étapes de Dépliage à partir de S et donnant l'état S_U
- Phase 2 : L'application tant qu'elle est possible de la (d, n)-Approximation démarrant de S_U et produisant $S_{(d,n)}$ (c'est à dire une (d, n)-Approximation ne peut être appliquée sur $S_{(d,n)}$)
- Phase 3 : L'application de la règle d'Introduction de Définition tant que possible en partant de $S_{(d,n)}$ et donnant S' (C'est à dire la règle d'Introduction de Définition ne peut être appliquée sur S').

Exemple 8.3.4. *Considérons le programme*
$dble(0, 0) \leftarrow$
$dble(s(X), s(s(Y))) \leftarrow dble(X, Y)$
et le but $ans(X) \leftarrow dble(X, X)$. *Il n'y a pas de* $\Rightarrow$*-dérivation qui atteint un état final. Toutefois le cs-programme* $ans(0) \leftarrow$ *défini la réponse.*

Exemple 8.3.5. *Considérons l'exemple 8.3.4. La figure 8.2 résume une* $\Rightarrow_s$*-dérivation en utilisant* $(1, 1)$*-étape. Dans ce cas précis cette approximation donne le résultat exact car* ans_1, ans_2 *et* ans_3 *sont inconsistants mais il est évident que ce n'est pas le cas général.*

Théorème 8.3.5.1. *N'importe quelle* $\Rightarrow_s$*-dérivation consistant en* (d, n)*-étapes est finie.*

Démonstration. Chaque (d, n)-étape est finie car la phase 1 est un nombre fini d'applications de Dépliage. La phase 2 est également finie car il n'est pas possible d'appliquer encore de (d, n)-Approximation sur une clause approximée. Finalement, la phase 3 est également finie car chaque application de cette règle élimine une clause dans $\mathcal{C}_{\text{new}}$.

Maintenant il reste à prouver qu'un nombre fini de définitions peut être générée à partir d'un but initial. Comme les corps des définitions sont

$\mathcal{D}_{\text{done}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	
$ans(X) \leftrightarrow$ $double(X, X)$	$ans(X) \leftrightarrow double(X, X)$		
	$ans(0) \leftarrow$ $ans(s(s(X))) \leftarrow double(s(X), X)$		U
	$ans(s(s(X))) \leftarrow double(s(X), X)$	$ans(0) \leftarrow$	D
$ans_1(X) \leftrightarrow$ $double(s(X), X)$	$ans_1(X) \leftrightarrow double(s(X), X)$	$ans(s(s(X))) \leftarrow$ $ans_1(X)$	D
	$ans_1(s(s(X))) \leftarrow double(s(s(X)), X)$		U
	$ans_1(s(s(X))) \leftarrow double(s(Y), X)$		A
$ans_2(X, Y) \leftrightarrow$ $double(s(Y), X)$	$ans_2(X, Y) \leftrightarrow double(s(Y), X)$	$ans_1(s(s(X))) \leftarrow$ $ans_2(Y, X)$	D
	$ans_2(s(s(X)), Y) \leftarrow double(Y, X)$		U
$ans_3(Y, X) \leftrightarrow$ $double(Y, X)$	$ans_3(Y, X) \leftrightarrow double(Y, X)$	$ans_2(s(s(X)), Y) \leftrightarrow$ $ans_3(Y, X)$	D
	$ans_3(s(Y), s(s(X))) \leftrightarrow double(Y, X)$		U
		$ans_3(s(Y), s(s(X))) \leftrightarrow$ $ans_3(Y, X)$	D

FIG. 8.2 – Une approximation

bornées en profondeur par d et en longueur par n, il y a seulement un nombre fini de corps différents (à rennomage de variable près). □

Théorème 8.3.5.2. *Soit $\mathcal{P}$ un programme logique et $\mathcal{D}$ un ensemble de définitions compatible avec $\mathcal{P}$. Si $\langle \mathcal{P}, \mathcal{D}, \mathcal{D}, \emptyset \rangle \stackrel{*}{\Rightarrow} \langle \mathcal{P}, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, alors $\mathcal{P}'$ est un cs-programme avec la propriété que $\mathcal{M}(\mathcal{P} \cup \mathcal{D})|_p \subseteq \mathcal{M}(\mathcal{P}')|_p$ pour tous les symboles de prédicats p définis par $\mathcal{D}$.*

Démonstration. Il est suffisant de prouver que les règles préservent la propriété suivante : $\mathcal{M}(\mathcal{P} \cup \mathcal{D}_{\text{done}}) \subseteq \mathcal{M}(\mathcal{P} \cup \mathcal{C}_{\text{new}} \cup \mathcal{C}_{\text{out}})$ qui est déjà prouvée pour Dépliage et Introduction de Définition. La (d, n)-Approximation transforme l'état $\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}} \rangle$ en $\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}'_{\text{new}}, \mathcal{C}_{\text{out}} \rangle$ où $\mathcal{C}'_{\text{new}}$ est obtenu à partir de $\mathcal{C}_{\text{new}}$ en remplaçant une de ses clauses par une plus générale, par conséquent $\mathcal{M}(\mathcal{P} \cup \mathcal{C}_{\text{new}} \cup \mathcal{C}_{\text{out}}) \subseteq \mathcal{M}(\mathcal{P} \cup \mathcal{C}'_{\text{new}} \cup \mathcal{C}_{\text{out}})$. □

Dans la plupart des applications, les sur-approximations elles mêmes n'ont pas beaucoup d'intérêt. Par contre la décision de leur vacuité peut être porteuse d'information car si l'approximation est vide alors le problème original est inconsistant. Par ailleurs des problèmes comme ceux étudiées dans la vérification, ont souvent un grande complexité combinatoire, ainsi que leur approximations. Le calcul de leurs modèles au moyen de cs-programme (ou d'automate d'arbre comme dans [40]), mène a une représentation qui explose en espace.

On propose ici une méthode pour vérifier le vide sur une sur-approximation qui évite le calcul de l'approximation entière et alors réduit une large part du calcul. Cette optimisation repose sur les propriétés des cs-clauses utilisées dans la preuve de la propriétée 8.2.2. Pendant le calcul de l'approximation, d'une part on marque les prédicats de $\mathcal{C}_{\text{out}}$ qui sont détéctées comme possédant un modèle non-vide et d'autre part on élimine à partir de $\mathcal{C}_{\text{new}}$ les clauses et les définitions dont la tête se compose d'un prédicat marqué.

Un nouvel ensemble est ajouté aux états de $\Rightarrow$-dérivation $\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}}, \mathcal{P}_{\text{mark}} \rangle$ où $\mathcal{P}_{\text{mark}}$ collecte les symboles de prédicats marqués. Un état initial est maintenant $\langle \mathcal{P}, \mathcal{D}, \mathcal{D}, \emptyset, \emptyset \rangle$ où $\mathcal{D}$ est un ensemble de définitions compatibles avec le programme logique $\mathcal{P}$. Les trois premières règles restent inchangées. Deux nouvelles règles, Marquage et Réduction, sont introduites, la première marque les prédicats, et la seconde élimine des clauses et des définitions de $\mathcal{C}_{\text{new}}$.

MARQUAGE.

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \cup \{p(\vec{t}) \leftarrow p_1(\vec{t_1}), \ldots, p_n(\vec{t_n})\},\, \mathcal{P}_{\text{mark}} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}},\, \mathcal{P}_{\text{mark}} \cup \{p\} \rangle}$$

si $\{p_1, \ldots, p_n\} \subseteq \mathcal{P}_{\text{mark}}$

RÉDUCTION.

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}} \,\dot{\cup}\, \{p(\vec{t}) \leftarrow \mathcal{B},\, \mathcal{C}_{\text{out}},\, \mathcal{P}_{\text{mark}}\} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}},\, \mathcal{P}_{\text{mark}} \rangle}$$

si $p \in \mathcal{P}_{\text{mark}}$

On écrit $S \hookrightarrow S'$ si S' est un état obtenu à partir de l'état S en appliquant l'une des 5 règles présentées dans ce chapitre. Une séquence $S_1 \hookrightarrow S_2 \ldots \hookrightarrow S_n$ est appelée une $\hookrightarrow$-dérivation, il peut être abrégé en $S_1 \overset{*}{\hookrightarrow} S_n$. Un état initial est de la forme $\langle \mathcal{P}, \mathcal{D}, \mathcal{D}, \emptyset, \emptyset \rangle$ où D est un ensemble de définitions compatible avec $\mathcal{P}$ et un état final est de la forme $\langle \mathcal{P}, \mathcal{D}, \emptyset, \mathcal{C}_{\text{out}}, \mathcal{P}_{\text{mark}} \rangle$ sur lesquels n'importe quelle étape de Marquage n'a aucun effet.

Exemple 8.3.6. *Considérons de nouveau l'exemple 8.3.4. La figure 8.3 montre comment Marquage et Réduction raccourcissent la $\Rightarrow$-dérivation de la figure 8.2 quand un fait pour ans est ajouté à $\mathcal{C}_{\text{out}}$.*

Pour cet exemple particulier, aucune approximation n'a été faite ce qui assure que ans(0) est un élément réel du modèle du but initial. En géneral, ce n'est pas le cas, et le non-vide du but approximé ne signifie pas que le problème initial est consistant.

Le théorème suivant permet d'assurer que le vide du modèle de l'approximation précédente garanti le vide du modèle du problème initial.

$\mathcal{D}_{\text{done}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	$\mathcal{P}_{\text{mark}}$	
$ans(X) \leftrightarrow$ $double(X,X)$	$ans(X) \leftrightarrow double(X,X)$			
	$ans(0) \leftarrow$ $ans(s(s(X))) \leftarrow double(s(X),X)$			U
	$ans(s(s(X))) \leftarrow double(s(X),X)$	$ans(0) \leftarrow$		D
$ans(s(s(X))) \leftarrow$ $double(s(X),X)$		ans	M	
			ans	P

FIG. 8.3 – Test du vide

Théorème 8.3.6.1. *Soit $\mathcal{P}$ un programme logique et $\mathcal{D}$ un ensemble de définitions compatible avec $\mathcal{P}$. Si $\langle \mathcal{P}, \mathcal{D}, \mathcal{D}, \emptyset, \emptyset\rangle \overset{*}{\hookrightarrow} \langle \mathcal{P}, \mathcal{D}', \emptyset, \mathcal{P}', \mathcal{P}_{\text{mark}}\rangle$, alors $\mathcal{P}'$ est un cs-programme qui certifie que si $\mathcal{M}(\mathcal{P}')|_p = \emptyset$ alors $\mathcal{M}(\mathcal{P} \cup \mathcal{D})|_p = \emptyset$ pour tout les symboles de prédicats p définis par $\mathcal{D}$.*

Démonstration. N'importe quelle $\hookrightarrow$-dérivation $S_0 \hookrightarrow S_1 \hookrightarrow \ldots \hookrightarrow S_n$ correspond à une $\Rightarrow_s$-dérivation $S'_0 \Rightarrow_s^? S'_1 \Rightarrow_s^? \cdots \Rightarrow_s^? S'_n$ où $\Rightarrow_s^?$ signifie 0 ou 1 $\Rightarrow_s$-étape et si $S_i = \langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}}, \mathcal{P}_{\text{mark}}\rangle$ alors $S'_i = \langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}} \cup \mathcal{C}_{mark}, \mathcal{C}_{\text{out}}\rangle$ où $\mathcal{C}_{mark}$ est l'ensemble des clauses et définitions éliminées par Réduction.

Il est facile de voir que pour un état $S_n = \langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}}, \mathcal{P}_{\text{mark}}\rangle$ d'une $\hookrightarrow$-dérivation démarrant par un état initial S, $p \in \mathcal{P}_{\text{mark}}$ seulement si $\mathcal{M}(\mathcal{C}_{\text{out}})_p \neq \emptyset$. Si S_n est un état final (c'est à dire $\mathcal{C}_{\text{new}} = \emptyset$), il correspond à l'état $S'_n = \langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}} \cup \mathcal{C}_{mark}, \mathcal{C}_{\text{out}}\rangle$ d'une $\Rightarrow_s$-dérivation. Par construction, toutes les définitions et clauses, $p(\vec{t}) \leftarrow ?\mathcal{B}$, de $\mathcal{C}_{mark}$ sont telles que $p \in \mathcal{P}_{\text{mark}}$. Considérons un symbole de prédicat q apparaissant dans la tête d'une définition de $\mathcal{D}_{\text{done}}$ mais pas dans $\mathcal{P}_{\text{mark}}$, car S_n est un état final, Marquage n'a pas d'effet dessus, alors $\mathcal{M}(\mathcal{C}_{\text{out}})_q = \emptyset$. q n'apparaît ni dans C_{mark} ni dans $\mathcal{P}$ par conséquent $\mathcal{M}(\mathcal{P} \cup \mathcal{C}_{mark} \cup \mathcal{C}_{\text{out}})_q = \emptyset$, car $\mathcal{M}(\mathcal{P} \cup \mathcal{C}_{mark} \cup \mathcal{C}_{\text{out}})$ est un sur ensemble de $\mathcal{M}(\mathcal{P} \cup \mathcal{D})$ où $\mathcal{D}$ est l'ensemble initial des définitions, $\mathcal{M}(\mathcal{P} \cup \mathcal{D})|_q = \emptyset$. □

Malheureusement la technique d'approximation utilisée peut s'avérer inutile même sur des cas simples comme l'illustre l'exemple 8.3.7. C'est pour cette raison que nous allons introduire de nouvelles règles d'approximation dans la section suivante.

Exemple 8.3.7. *Considérons le programme*
$gr(0) \leftarrow$
$gr(X) \leftarrow gr(s(X))$
$gr(s(X)) \leftarrow gr(X)$

et le but ans $\leftarrow gr(1)$. *Soif* f *un symbole de fonction* f^n *représente* $n-1$ *fois l'application de* f *sur lui même. Soit* n *le paramètre concernant la hauteur pour l'approximation on peut construire un but* $gr(s^n(1))$ *après* n *applications successives de la règle de dépliage du but par la seconde clause. En approximant on obtient alors le but* $gr(s^n(X))$. *Lorsque l'on déplie cette clause à* n *reprises par rapport à la troisième clause on a le but* $gr(X)$ *qui s'unifie avec* $g(0)$ *et l'approximation nous trouve alors un faux modèle.*

Dans la section suivante on verra un technique d'approximation permettant de résoudre ce problème.

8.4 Approximation complexe

Maintenant on a besoin de définir une procédure de calcul de sur-approximation qui nous permet de traiter le cas de l'exemple précédant avec succès. A cet effet on s'est inspiré de l'idée de [44] définissant des fonctions d'approximations régulières liées à des positions particulières de TRS. Transposé à notre cas cela nous donne la définition d'un nombre fini de prédicats d'approximations nous permettant de limiter en hauteur les atomes des corps des clauses.

Cette sur-approximation va être définie dans la suite de cette section, mais avant on a besoin de d'autres définitions. On définit d'abord l'ensemble de toutes les clauses de Horn $\mathcal{H}$ et l'ensemble de tous les programmes logiques Pl. La prochaine définition va nous permettre de caractériser les clauses sur lesquels on va pouvoir effectuer ou non les différentes règles de la prochaine approximation.

Définition 8.4.1. *On définit l'application* $Type : \mathcal{H}, Pl \mapsto \{1,2\}$ *de la manière suivante :*

- $Type(H \leftarrow \mathcal{B}, \mathcal{P}) = 1$ *si tous les atomes de* $\mathcal{B}$ *dont les prédicats sont têtes de clauses de* $\mathcal{P}$ *sont sans symbole de fonction. De telles clauses seront dites du type 1 si* $\mathcal{P}$ *est clair dans le contexte .*
- $Type(H \leftarrow \mathcal{B}, \mathcal{P}) = 2$ *sinon. De telles clauses seront dites être du type 2 si* $\mathcal{P}$ *est clair dans le contexte.*

Exemple 8.4.2. $Type((p(0) \leftarrow A_0(x), A_1(f(y))), \{A_0(0) \leftarrow\}) = 1$ *car* A_0 *est le seul prédicat tête de clause de* $\{A_0(0) \leftarrow\}$ *et il n'apparaît que dans un atome sans symbole de fonction.*
$Type((p(0) \leftarrow A_0(x), A_1(y), A_O(O)), \{A_0(0) \leftarrow\}) = 2$ *car* A_0 *est le seul prédicat tête de clause de* $\{A_0(0) \leftarrow\}$ *et il apparaît dans un atome du corps* $A_0(0)$ *avec un symbole de fonction.*

Les deux prochaines définitions introduisent des applications qui vont permettre de donner des bornes pour les clauses que l'on va générer dans la prochaine approximation.

Définition 8.4.3. *Soit $H \leftarrow \mathcal{B}$ une clause de Horn, on définit les applications $Compos(H \leftarrow \mathcal{B}) = m$ et $Hauteur(H \leftarrow \mathcal{B}) = n$ avec m le nombre d'atomes maximal dans une même composante de la décomposition en composante sans variables communes de $\mathcal{B}$, et n la hauteur maximal d'un atome de $\mathcal{B}$.*

Définition 8.4.4. *Une application $Cond : \mathcal{H} \mapsto \{vrai, faux\}$ est une* condition d'approximation *s'il existe deux entiers n et m tels que pour toute clause de Horn $H \leftarrow \mathcal{B}$ telle que $Hauteur(H \leftarrow \mathcal{B}) > n$ ou $Compos(H \leftarrow \mathcal{B}) > m$ alors $Cond(H \leftarrow \mathcal{B}) = vrai$. Dans ce cas n et m seront dites être les* bornes vertical (et horizontal respectivement) *de la condition d'approximation.*

La prochaine définition définit une stratégie de sur-approximation qui s'applique forcément lorsque des bornes ont été atteintes et qui permet d'assurer que l'on produit effectivement une sur-approximation.

Définition 8.4.5. *Soit une clause $p(\vec{t}) \leftarrow \mathcal{B}$ dont les prédicats sont définis par un programme logique $\mathcal{P}$ et soit une condition d'approximation $Cond$ tels que $Cond(p(\vec{t}) \leftarrow \mathcal{B}) = vrai$ de borne vertical n et de borne horizontal m et $\mathcal{A}$ un ensemble fini de prédicat. Une application $Strat$ telle que $Strat(p(\vec{t}) \leftarrow \mathcal{B}, \mathcal{A}) = (\{p(\vec{t'}) \leftarrow \mathcal{B}'\}, S)$ est une* stratégie de sur-approximation *si tout $A \leftarrow A' \in S$ est une cs-clause dont les prédicats sont dans $\mathcal{A}$, et si $\mathcal{M}(\mathcal{P} \cup \{p(\vec{t}) \leftarrow B\})_P \subseteq \mathcal{M}(\mathcal{P} \cup \{p(\vec{t'} \leftarrow \mathcal{B}'\} \cup S)_P$ et tout $C \in \{p(\vec{t'}) \leftarrow \mathcal{B}'\} \cup S$ on a $Hauteur(C) \leq n$ et $Compos(C) \leq m$.*

On remarquera que les approximations définies dans la section précédante sont un cas particulier de stratégie de sur-approximation, où l'ensemble de prédicat $\mathcal{A}$ est vide.

Les règles de transformations ont pour états $\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}}, \mathcal{A}_{\text{new}}, \mathcal{A} \rangle$ $\mathcal{P}$, $\mathcal{D}_{\text{done}}$, $\mathcal{C}_{\text{new}}$ et $\mathcal{C}_{\text{out}}$ ont le même rôle que dans la transformation définie section 8.2 et $\mathcal{A}_{\text{new}}$ est l'ensemble des clauses d'approximations générées, et $\mathcal{A}$ est un ensemble fini de noms de prédicat différents de ceux apparaissant dans $\mathcal{P}$.

On écrit $S \Rightarrow_{Strat,\mathcal{A}} S'$ si S' est un état obtenu d'un état S par une application de la règle de *dépliage approx*, de la règle *introduction d'approximations* (voir plus bas la description) ou de la règle *introduction de définitions* avec la stratégie d'approximation *Strat* qui quand elle sera appliquée aura toujours $\mathcal{A}$ comme second argument. La clôture réflexive et transitive de $\Rightarrow_{Strat,\mathcal{A}}$ est notée par $\stackrel{*}{\Rightarrow}_{Strat,\mathcal{A}}$. Un *état initial* est de la forme $\langle \mathcal{P}, \emptyset, \mathcal{D}, \emptyset, \emptyset, \mathcal{A} \rangle$ où $\mathcal{D}$ est linéaire compatible avec $\mathcal{P}$; un *état potentiellement*

final est de la forme $\langle \mathcal{P}, \mathcal{D}', \emptyset, \mathcal{P}', \mathcal{A}', \mathcal{A} \rangle$. $\mathcal{P}$ et $\mathcal{D}$ sont appelées les entrées d'une dérivation, $\mathcal{P}'$ est la sortie. Un état S est *final* si c'est un état potentiellement final et tout S' tel que $S \Rightarrow_{Strat,\mathcal{A}} S'$ on a $S' = S$. Une dérivation est appelée *complète* si son dernier état est final.

DÉPLIAGE APPROX.

$$\frac{\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}} \dot{\cup} \{L \leftarrow \mathcal{R} \dot{\cup} A\}, \mathcal{C}_{\text{out}}, \mathcal{A}_{\text{new}}, \mathcal{A} \rangle}{\langle \mathcal{P}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}} \cup \mathcal{C} \cup \mathcal{D}_1, \mathcal{C}_{\text{out}}, \mathcal{A}_{\text{new}}, \mathcal{A} \rangle,}$$

où $\mathcal{C}$ est l'ensemble de toutes les clauses $(L \leftarrow \mathcal{R} \cup B_1)\mu$ telles que $H_i \leftarrow \mathcal{B}_i$ est une clause d'un programme $\mathcal{D}$ définit ci-dessous, pour $i = 1, \ldots, k$, telles que l'unificateur le plus général μ de $\{A\}$ et H_1 existe, et soit :

- $Type(\{L \leftarrow \mathcal{R} \dot{\cup} A\}, \mathcal{A}_{\text{new}}) = 1$ et $Cond(\{L \leftarrow \mathcal{R} \dot{\cup} A\}) = faux$ alors $\mathcal{D} = \mathcal{P}$ et $\mathcal{D}_1 = \emptyset$.
- $Type(\{L \leftarrow \mathcal{R} \dot{\cup} A\}, \mathcal{A}_{\text{new}}) = 2$ et A est un atome avec symbole de fonction dont le prédicat apparaît en tête d'une clause de $\mathcal{A}_{\text{new}}$ et alors $\mathcal{D} = \mathcal{P}$ et $\mathcal{D}_1 = \{L \leftarrow \mathcal{R} \dot{\cup} A\}$

INTRODUCTION DE DÉFINITIONS APPROXIMATIVES. Prendre une clause C non encore traitée si $Cond(C) = vrai$ alors appliquer la stratégie d'approximation $Strat$ sur C et appliquer Introduction de définition sur le premier argument du résultat et introduire les clauses du second argument du résultat dans les nouvelles définitions d'approximations, sinon appliquer introduction de définition.

Formellement :

$$\frac{\langle \mathcal{P},\ \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}} \dot{\cup} \{H \leftarrow \mathcal{B}\}, \mathcal{C}_{\text{out}}, \mathcal{A}_{\text{new}}, \mathcal{A} \rangle}{\langle \mathcal{P},\ \mathcal{D}_{\text{done}} \cup \mathcal{D},\ \mathcal{C}_{\text{new}} \cup \mathcal{D},\ \mathcal{C}_{\text{out}} \cup \{H \leftarrow \{L_1, \ldots, L_k\}\},\ \mathcal{A}_{\text{new}} \cup \mathcal{D}_2,\ \mathcal{A} \rangle}$$

$Type(H \leftarrow \mathcal{B}, \mathcal{A}_{\text{new}}) = 1$ et $Cond(H \leftarrow \mathcal{B}) = vrai$, alors $Strat(H \leftarrow \mathcal{B}, \mathcal{A}) = (\{H \leftarrow \mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_k\}, \mathcal{D}_2)$ et $\mathcal{B}_1, \ldots, \mathcal{B}_k$ est une décomposition maximale de $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ en composantes sans variables communes.

$$L_i = \begin{cases} L & \text{si } \{L \leftarrow \mathcal{B}_i\}\eta^{-1} \in \mathcal{D}_{\text{done}} \text{ avec } \{L \leftarrow \mathcal{B}_i\}\eta^{-1} \text{ linéaire et} \\ & L \text{ sans symbole de fonction pour une substitution de} \\ \text{variables } \eta & \\ P_i(\vec{x}) & \text{autrement, } \vec{x} \text{ variables de } \mathcal{B}_i. \end{cases}$$

pour $1 \leq i \leq k$ et le nouveau symbole de prédicat P_i, et où $\mathcal{D}$ est l'ensemble de tout $L_i \leftarrow \mathcal{B}_i$ tel que L_i contient un nouveau symbole de prédicat.

INTRODUCTION DE DÉFINITIONS. Prendre une clause C non encore traitée si $Cond(C) = faux$ alors on applique définition d'introduction.

Formellement :

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

$Type(H \leftarrow \mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_k, \mathcal{A}_{\text{new}}) = 1$ et $Cond(H \leftarrow \mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_k) = faux$, et $\mathcal{B}_1, \ldots, \mathcal{B}_k$ est une décomposition maximale de $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ en composantes sans variables communes

$$L_i = \begin{cases} L\eta^{-1} & \text{si } (L \leftrightarrow \mathcal{B}_i)\eta \in \mathcal{D}_{\text{done}} \text{ pour un rennomage de var } \eta \\ & \text{and } Var(H) \cap Var(\mathcal{B}_i) = Var(L\eta) \cap Var(\mathcal{B}_i\eta) \\ P_i(x_1, \ldots, x_n) & \text{autrement, } \{x_1, \ldots, x_n\} \text{ les vars. de } \mathcal{B}_i. \end{cases}$$

pour $1 \leq i \leq k$ et un nouveau symbole de prédicat P_i, et où $\mathcal{D}$ est l'ensemble de tous les $L_i \leftrightarrow \mathcal{B}_i$ tels que L_i contient un nouveau symbole de prédicat.

On remarquera que l'on n'introduit de nouvelles définition que lorsque les clauses sont du type 1.

Exemple 8.4.6. *Considérons encore l'exemple 8.3.7. La figure 8.4 résume une $\Rightarrow_{Strat}$-dérivation où $\mathcal{A} = \{P\}$ est l'ensemble fini des prédicats d'approximations (ici il ne contient qu'un élément) la condition d'approximation donne vrai si on a un symbole de fonction à une profondeur d'au moins* 1 *dans un atome du corps et $Strat(H \leftarrow \mathcal{B}\sigma, \mathcal{A}) = (\{H \leftarrow \mathcal{B}', \vec{P}_i(x)\}, \{\, p(t\sigma) \leftarrow \vec{p}(Y) \mid p(Y) \in \mathcal{B}\sigma$ et $Y \in Var(t\sigma)\,\} \cup \{\, p(t) \leftarrow \mid$ si t est à la profondeur* 1 *d'un argument de $\mathcal{B}\sigma$ et $\nexists y$ tel que $p(Y) \in \mathcal{B}\sigma et Y \in Var(t\sigma)\,\})$. Il arrive que cette approximation donne le résultat exact car ans, L_1 et L_2 sont inconsistants mais il est évident que ce n'est pas le cas général.*

Théorème 8.4.6.1. *Soit $\mathcal{P}$ un programme logique, $\mathcal{D}$ un ensemble de définitions compatible avec $\mathcal{P}$. Si $\langle \mathcal{P}, \mathcal{D}, \mathcal{D}, \emptyset, \emptyset, \mathcal{A}\rangle \stackrel{*}{\Rightarrow}_{Strat,\mathcal{A}} \langle \mathcal{P}, \mathcal{D}', \mathcal{D}_2, \mathcal{P}', \mathcal{A}_{\text{new}}, \mathcal{A}\rangle$, alors $\mathcal{P}'$ est un cs-programme avec la propriété que $\mathcal{M}(\mathcal{P} \cup \mathcal{D})|_p \subseteq \mathcal{M}(\mathcal{P}')|_p$ pour tout les symboles de prédicats p définis par $\mathcal{D}$.*

Démonstration. D'abord on remarquera que le modèle de $\mathcal{A}_{\text{new}}$ est croissant car on ne fait qu'y rajouter des clauses. On distinguera ici pour un soucis de détail $\mathcal{A}^d_{\text{new}}$ les clauses de $\mathcal{A}_{\text{new}}$ avant l'application d'une règle et $\mathcal{A}^f_{\text{new}}$ les clauses de $\mathcal{A}_{\text{new}}$ après l'application d'une règle. Il est suffisant de prouver que les règles préservent la propriété suivante : $\mathcal{M}(\mathcal{P} \cup \mathcal{D}_{\text{done}} \cup \mathcal{A}^d_{\text{new}}) \subseteq \mathcal{M}(\mathcal{P} \cup \mathcal{C}_{\text{new}} \cup \mathcal{C}_{\text{out}} \cup \mathcal{A}^f_{\text{new}})$ qui est déjà prouvée pour Dépliage et Introduction de Définition. En considérant $\mathcal{A}^d_{\text{new}} \cup \mathcal{P}$ comme un même programme on a Dépliage approx qui devient un cas particulier de la règle Dépliage. On remarquera également que pour les clauses du type 1 l'étape d'Introduction

$\mathcal{D}_{\text{done}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	$\mathcal{A}_{\text{new}}$	
$ans \leftarrow$ $\quad gr(1)$	$ans \leftarrow gr(1)$			
	$ans \leftarrow gr(s(1))$			U
$l_1(X) \leftarrow$ $\quad gr(s(X))p(X)$		$ans \leftarrow l_1(X)$	$p(1) \leftarrow$	U
	$l_1(X) \leftarrow$ $\quad gr(X)p(X)$ $l_1(s(X)) \leftarrow$ $\quad gr(s(s(X)))p(s(X))$			D
$l_2(X) \leftarrow$ $\quad gr(X)p(X)$	$l_1(s(X)) \leftarrow$ $\quad gr(s(s(X)))p(s(X))$	$l_1(s(X)) \leftarrow L_2(X)$		D
	$l_2(0) \leftarrow 0)$ $l_2(X) \leftarrow g(s(X)),$ $\quad p(X)$ $l_2(s(X)) \leftarrow g(X),$ $\quad p(s(X))$ $l_1(s(X)) \leftarrow$ $\quad gr(s(s(X)))p(s(X))$			D
	$l_2(0) \leftarrow p(0)$ $l_2(s(X)) \leftarrow g(X),$ $\quad p(s(X))$ $l_1(s(X)) \leftarrow$ $\quad gr(s(s(X)))p(s(X))$	$l_2(X) \leftarrow l_1(X)$		D

FIG. 8.4 – Une approximation complexe

d'approximation consiste en une étape d'Introduction de Définition. Enfin soit une clause $H \leftarrow \mathcal{B}$ du type 2 l'Introduction d'approximation effectue l'étape d'introduction de définition sur un ensemble de clauses dont le modèle est par définition de la stratégie d'approximation inclus dans celui de $C \cup \mathcal{P} \cup \mathcal{A}^d_{\text{new}} \cup \mathcal{D}_{\text{done}}$. □

On remarquera initiallement que les clauses dans $\mathcal{A}_{\text{new}}$ sont des cs-clauses, et qu'en l'état actuel on ne peut pas garantir la terminaison du processus, notament on ne garanti pas que le dépliage successif des clauses du type 2 termine. Afin de garantir cette terminaison on va introduire une nouvelle stratégie.

Définition 8.4.7. *Une stratégie de sur-approximation Strat sera dite* terminante *si pour tout clause $H \leftarrow \mathcal{B}$ dont le résultat de la condition d'approximation appliqué à C est vrai si $Strat(H \leftarrow \mathcal{B}, \mathcal{A}) = (\mathcal{P}, \mathcal{P}')$ tel que $\mathcal{P}'$ est un programme logique composé uniquement de clauses d'automates (voir section 3.2), et $\mathcal{A}$ un ensemble fini de noms de prédicats.*

Afin de garantir la terminaison du processus on a également besoin d'une stratégie concernant le choix des clauses dans $\mathcal{C}_{\text{new}}$. Le soucis vient du fait que $\mathcal{C}_{\text{new}}$ contient toujours une clause de type 2 dès que celle-ci a été générée. Il convient donc de ne pas choisir toujours la même clause. On a besoin de généraliser ce mécanisme pour l'obtention d'un état final.

Théorème 8.4.7.1. *N'importe quelle $\Rightarrow_{Strat,\mathcal{A}}$-dérivation mène à un état final si Strat est une stratégie de sur-approximation terminante et si l'étape Dépliage approx sélectionne en priorité une clause du type 1 dans $\mathcal{C}_{\text{new}}$, et si il n'existe pas de clause dans $\mathcal{C}_{\text{new}}$ l'étape Dépliage approx séléctionne une à une toutes les clauses de $\mathcal{C}_{\text{new}}$.*

Démonstration. Remarquons d'abord que pour toutes les clauses introduites dans $\mathcal{A}_{\text{new}}$ on a seulement des prédicats qui sont dans $\mathcal{A}$. De plus ce sont des cs-programmes ayant une hauteur de tête finie. Le nombre de clauses possible générés dans $\mathcal{A}_{\text{new}}$ est donc fini. Les clauses du type 2 sont seulements concernés par l'étape Dépliage approx. Comme on a une stratégie de sur-approximation terminante, on va déplier par rapport à des clauses d'automates qui nous donne un nombre fini de dépliage. Les clauses dans $\mathcal{C}_{\text{new}}$ ont une hauteur bornée car d'une part on ne peut déplier les clauses du type 1 si la condition d'approximation est vraie. D'autre part le dépliage des clauses du type 2 n'engendre pas d'augmentation de la hauteur maximale d'un atome dans la clause (car on ne déplie que par rapport à des clauses d'automates). D'autre part les prédicats dans le corps de $\mathcal{C}_{\text{new}}$ sont exclusivement ceux de $\mathcal{A}$ ou de $\mathcal{P}$. On aura donc un nombre fini de corps différents (à rennomage de variable près) dans l'ensemble des clauses de $\mathcal{C}_{\text{new}}$. Par conséquent le nombre de nouvelles définitions dans $\mathcal{D}_{\text{done}}$ est également borné.

On remarquera que seul les clauses du type 1 sont éliminées de $\mathcal{C}_{\text{new}}$. Finalement pour l'étape de dépliage la séléction de toutes les clauses de type 2 permet d'avoir la certitude de pouvoir créer de nouvelles clauses dans $\mathcal{C}_{\text{new}}$ si c'est possible. Comme la procédure ne boucle pas avant d'atteindre un état final et que le nombre d'état dans les différentes piles est fini, le processus atteint un état final.

□

8.4.1 Un mot sur la trace du modèle

Lorsque le modèle est non vide il peut être intéressant de savoir pourquoi et de voir si une sur-approximation a été introduite ou non pour trouver ce modèle. C'est le but de la trace.

A cet effet nous avons cherché le plus petit chemin pour trouver un modèle dans $\mathcal{C}_{\text{out}}$ et pour chacune de ces clauses pour chaque atome dans $\mathcal{C}_{\text{out}}$ nous avons récupéré les dépliages des règles qui nous ont donné ce résultat. Malheureusement je n'ai pas encore eu le temps de décrire ce processus de manière formelle. Toutefois j'ai pu implémenter cette technique et un exemple de trace est disponible en annexe pour un problème exposé dans la section suivante.

8.5 Application à la vérification de protocoles cryptographiques

Le but de cette section est de montrer que l'on peu utiliser avec succès notre méthode de sur-approximation sur un problème complexe. On a choisi pour cela de modéliser un protocole cryptograpique à l'aide d'un programme logique. On présente ici l'étude du cas du protocole à clef publique de Needham-Schroeder (NSPK, pour Needham-Schroeder Public Key protocol en anglais) [79]. Plus précisement, on utilise d'abord ici la version initiale, et ensuite la version corrigée du protocole sans les clef de serveur [69] car nous considérons dès le départ que chaque agent possède la *clef publique* de tout les autres agent.

On a choisi ce protocole principalement pour les deux raisons suivantes. Premièrement c'est un protocole qui est utilisé et qui peut être rapidement compris. La deuxième raison c'est la relative difficulté de la vérification de ce protocle car la version bugguée était considérée comme sûrmakee à partir de [79] jusqu'à [68].

Notre modélisation est fortement inspirée de [44] où les règles sont représentées comme des règles de réécriture, le réseau comme un automate avec de la réécriture et l'approximation est finallement intersectée avec un automate représentant les états interdits dans le but de prouver la secu-

rité. La principale différence est que l'on utilise juste des clauses de Horn pour tout représenter comme le fait [13]. Remarquons que si le système est considéré comme non-sur notre méthode et formalisme révèle automatiquement une attaque possible comme dans [16, 2]. Un exemple de trace d'attaque est exposé dans l'annexe A.

8.5.1 Le protocole à clef publique de Needham-Schroeder

Le protocole NSPK a pour but d'établir l'authentification mutuelle entre un *initiateur* A et un *répondeur* B. C'est à dire qu'à la fin du protocole, A veut être sûre qu'il a bien communiquer avec B et inversement. Le protocole utilise des *clef publique de cryptage*. Ce qui signifie que l'on considère que chaque agent a possède une clef publique K_a. Les nonces N_A et N_B sont supposés être nouveaux. Les clef sont assymétriques, c'est à dire que seul le propriétaire de la clef est capable de décrypter un message crypé avec sa clef. Dans ce protocole on considère que n'importe qui peut encrypter un message avec n'importe quelle clef. Premièrement regardons la version initiale du protocole.

1. $A \Rightarrow B : \{N_A, A\}_{K_B}$
2. $B \Rightarrow A : \{N_A, N_B\}_{K_A}$
3. $A \Rightarrow B : \{N_B\}_{K_B}$

A la première étape, A essaye d'initier une communication avec B. A crée un nonce N_A qu'il envoie à B dans un message contenant également son identité, le tout encryté avec la clef publique de $B : K_B$. Dans la seconde étape, B envoie à A un message contenant le même nonce N_A et un nouveau nonce N_B, le tout encrypté cette fois avec la clef publique de A. Finallement, dans la troisième étape, A renvoit à B le nonce N_B qu'il a reçu encrypté avec la clef publique de B.

On voit ci-après la description des trois étapes de la version correcte du protocole. La seule différence avec la précédente concerne la seconde règle où l'identité de l'envoyeur B est encrypté avec les nonces N_A et N_B.

1. $A \Rightarrow B : \{N_A, A\}_{K_B}$
2. $B \Rightarrow A : \{N_A, N_B, B\}_{K_A}$
3. $A \Rightarrow B : \{N_B\}_{K_B}$

Comme notre propos n'est pas ici de montrer comment représenter un protocole cryptographique on ne va pas expliquer en détail le code.

8.5.2 Codage du protocole

.

Dans cette section on présente un modélisation du protocole NSPK par un programme logique. Premièrement on présente la signature $\mathcal{F}$ et les

termes de $\mathcal{T}(\mathcal{F})$ agent, clef, message et ainsi de suite. Chaque agent est labellé par un unique identifiant. Le terme $m(x, y, z)$ représente un message, y l'envoyeur affiché, z le réceptionneur et finallement c le contenu du message. On peut remarquer que l'on ne distingue pas le véritable receveur du receptionneur affiché, car l'on considère que l'intrus peut intercepter chaque message. Le terme $pk(a)$ signifie la clef publique de a. Le terme $nonce(a, b)$ représente un nouveau message envoyé par a et reçu par b. Le terme $enc(k, \vec{n}, v)$ représente l'encryptage d'un vecteur $\vec{n}$ de composition de nonce, clef, agent et identification avec la clef publique k, et v est seulement là dans un but de vérification et désigne celui qui a effectivement encrypter le nonce.

On a un nombre non borné d'agents mais nous distinguons spécialement deux d'entres eux a et b les autres étant représentés par des entiers.

$$\begin{array}{ll} agt(a). & agt(b). \\ agt(X) : -int(X). & int(0). \\ int(s(X)) : -int(X). & \end{array}$$

Maintenant pour chaque étape du protocole on donne la clause correspondante. Le codage est assez direct à partir de la spécification. Voyons d'abord la version non bugguée du protocole.

1. $A \Rightarrow B : \{N_A, A\}_{K_B}$

$$p(m(agt(A), agt(B), enc(key(agt(B), agt(A), nonce(A, B), A)))) : - \\ agt(A), agt(B).$$

2. $B \Rightarrow A : \{N_A, N_B, B\}_{K_A}$

$$p(m(agt(B), agt(Y), enc(key(agt(Y)), U, nonce(B, Y), B))) : - \\ p(m(A, agt(B), enc(key(agt(B)), agt(Y), U, C))).$$

3. $A \Rightarrow B : \{N_B\}_{K_B}$

$$p(m(agt(A), B, enc(key(B), U, A))) : - \\ p(m(B, agt(A), enc(key(agt(A)), nonce(A, N1), U, C)))$$

Maintenant voyons la version corrigé du protocole NSPK. Les différences sont dans la seconde et troisième règle où l'identité du supposé encrypteur est encrypté.

1. $A \Rightarrow B : \{N_A, A\}_{K_B}$

$$p(m(agt(A), agt(A), agt(B), enc(pk(agt(B)), agt(A), nonce(agt(A), agt(B))))) : - \\ agt(A), agt(B).$$

2. $B \Rightarrow A : \{N_A, N_B, B\}_{K_A}$

$$p(m(agt(B), agt(Y), enc(key(agt(Y)), agt(B), U, nonce(B, Y), B))) : - \\ p(m(A, agt(B), enc(key(agt(B)), agt(Y), U, C)))$$

3. $A \Rightarrow B : \{N_B\}_{K_B}$

$$p(m(agt(A), Y, enc(key(Y), U, A))) : -$$
$$\quad p(m(B, agt(A), enc(key(agt(A)), Y, nonce(A, N1), U, C)))$$

On remarquera que comme dans [13] le réseau est modélisé de manière interne au processus, c'est à dire que l'ensemble des messages du réseau correspond au modèle de p.

8.5.3 Capacités de l'intrus

Dans cette section on présente les capacités de l'intrus à partir des messages du réseau. Une intrusion consiste en l'émission d'un nouveau message à partir d'un message du réseau.

L'intrus est capable de construire ses messages à partir de la liste des messages déjà émis, peut envoyer à n'importe quel agent et prétendre être n'importe quel agent. On considère ici que tous les agents différents de a ou b peuvent aider l'intrus. A partir d'un message du réseau l'intrus peut :

- réenvoyer un message du réseau à n'importe qui en se faisant passer pour n'importe qui ;
- décrypter et encrypter un message si il est encrypté avec la clef de tous les agents distincts de a ou b ;
- encrypter la partie correspondant au contenu du message avec n'importe quelle clef comme si il était n'importe qui.

Ci-dessous, le code correspondant pour la version non-corrigée du protocole. Pour la version corrigée on rajoute des nouvelles clauses avec au bon endroit un nouvel argument.

$$p(m(agt(Z), agt(T), M1)) : -$$
$$\quad p(m(B, C, M1)), agent(Z), agent(T).$$
$$p(m(agt(Z1), agt(T1), M)) : -$$
$$\quad p(m(Y, Z, enc(key(agt(T)), M, K))), agti(T), agt(Z1), agt(T1).$$
$$p(m(agt(Z1), agt(T1), M)) : -$$
$$\quad p(m(Y, Z, enc(key(agt(T)), M, M1, K))), agti(T), agt(Z1), agent(T1).$$
$$p(m(agt(Z1), agt(T1), M1)) : -$$
$$\quad p(m(Y, Z, enc(key(agt(T)), M, M1, K))), agti(T), agt(Z1), agt(T1).$$
$$p(m(agt(Z1), agt(T1), enc(key(agt(X)), M, K))) : -$$
$$\quad agt(X), p(m(U, V, M)), agt(Z1), agt(T1), agti(K).$$
$$p(m(agt(Z1), agt(T1), enc(key(agt(X)), agt(E), M, K))) : -$$
$$\quad agent(X), agent(E), p(m(B, C, M)), agent(Z1), agent(T1), agti(K).$$
$$p(m(agt(Z1), agt(T1), enc(key(agt(X)), M1, M2, K))) : -$$
$$\quad agt(X), p(m(B1, C1, M1)), p(m(B2, C2, M2)), agt(Z1), agt(T1), agti(K).$$

8.5.4 But, Authentification et Confidentialité

Dans cette section on présente les états interdits. Concernant la confidentialité on ne souhaite pas que l'intrus puisse envoyer les nonces $nonce(a, b)$ et $nonce(b, a)$ sur le réseau.

$$forbid(m(I, Y, Z, nonce(a, b))) : -agti(I).$$
$$forbid(m(I, Y, Z, nonce(b, a))) : -agti(I).$$

Concernant l'authentification on se concentre sur le message reçu à la fin des étapes 2 et 3 du protocole. Dans le premier cas A reçoit un message de B contenant un nonce $nonce(A, N1)$ et il veut que $N1$ qui a envoyé le message soit le même que celui qui a encrypté le nonce c'est à dire Z. Le deuxième cas fonctionne de manière similaire. Ici c'est pour la version corrigée du protocole. pour la version non-corrigée, on supprime le Y.

$$forbid(m(X, B, A, enc(pk(A), Y, nonce(A, N1), U, Z))) : -aut(A, N1, Z).$$
$$forbid(m(A, A, B, enc(pk(B), nonce(B, N), Z))) : -aut(B, N, Z).$$

$aut(A, B, X) : -isab(A), isab(B), diff(B, X).$ $\quad isab(agt(a)).$
$isab(agt(b)).$ $\quad diff(agt(a), agt(b)).$
$diff(agt(b), agt(a)).$ $\quad diff(A, agt(X)) : -isab(A), agti(X).$

Contrairement à [44] qui calcule d'abord une sur-approximation avant d'effectuer l'intersection, on va effectuer ici directement la sur-approximation sur l'intersection des états interdits et du modèle du prédicat représentant le réseau.
$: -p(R), forbid(R).$

8.5.5 Stratégie et prédicats d'approximations pour NSPK

Dans cette section on a présenté jusqu'ici le code permettant de modéliser le problème de vérification lié au protocole cryptographique NSPK. Dans la section 8.4 on a présenté la technique basée sur la transformation de programmes que l'on utilise pour pouvoir prouver les propriétés de confidentialité et d'authentification du protocole NSPK. Ici on s'attachera donc à montrer la manière dont on utilise les prédicats d'approximations pour résoudre efficacement le problème exposé dans cette section.

La stratégie que l'on va utiliser ici est spécifique au problème. En effet on va garder la structure des messages, pour cela on va introduire un prédicat d'approximation dans le corps des clauses dès que l'on aura deux symboles identiques l'un en dessous de l'autre[1]. Par exemple lorsque l'on aura $p(m(X, Y, enc(Z, T, enc(X, Y, Z))))$ on va introduire un nouveau prédicat

[1] non nécessairement directement

d'approximation pour avoir $p(m(X, Y, enc(Z, T, U))), approx(U)$ et la nouvelle clause $approx(enc(X, Y, Z)) \leftarrow$. La condition d'approximation liée à cette stratégie possède une borne verticale égale au nombre de symbole de constructeur dans Σ car un même symbole ne peut se répéter à deux reprises. En pratique dans cet exemple il n'y a pas besoin de fixer une borne horizontale car le nombre d'atomes dans une décomposition maximale en composante sans variable commune est naturellement bornée.

On aurait pu introduire des prédicats d'approximation correspondant aux différentes positions dans les atomes où l'on fait une approximation, mais dans la pratique on rencontre un autre problème. Celui-ci est lié au fait que le prédicat d'approximation va confondre à un momment les termes $enc(key(agt(0), X, Y, agt(Z))$ et $enc(agt(a), X, nonce(a, b), agt(Z))$ qui contient un message que l'on veut être confidentiel. Le problème survient lors du dépliage du prédicat d'approximation contenant ces données, en effet on va pouvoir décrypter le premier terme et avoir une variable Y au lieu du sous terme $nonce(a, b)$ qui devrait toujours rester encrypté. Il y a sans doute plusieurs manières d'éviter l'apparition de ce problème. Celle que l'on a choisies est de donner le même nom de prédicat *nonceab* pour toutes les fois où le sous-terme que l'on va sur-approximer contient soit $nonce(a, b)$ soit $nonce(b, a)$. Afin de pouvoir garder le lien avec ce prédicat, un autre nom de prédicat *approx* est donné lorsque le sous-terme à sur-approximer contient une variable dans un atome dont le prédicat est *nonceab*. Cette méthode permet dans l'implantation actuelle de prouver la confidentialité en 1 seconde et de donner la trace d'une attaque pour la confidentialité présentée en annexe en 54 secondes.

8.6 Conclusion

Dans ce chapitre on a vu commment on pouvait utiliser les cs-programmes afin de détecter le vide à partir de requêtes. On a appliqué cette méthode avec succès dans le cadre des questions de sûreté et d'authentification de protocoles cryptographiques, mais il reste encore de nombreux tests à faire, ainsi que de nombreuses optimisations. Par exemple on pourraît inverser le marquage c'est à dire marquer des prédicats dont on sait qu'ils ont déjà un modèle afin d'éviter des calculs intutils.

Malgré tout l'utilisation des prédicats d'approximations paraît prometteuse, en effet ceux-ci permettent de stocker les données comme une sorte de multiensemble de telles manière qu'elles pourraient permettre de traiter de propriétés algébriques comme le font les auteurs de [17] avec les fonctions d'approximations. De plus on peut espérer restreindre les paramètres de la stratégie de sur-approximation terminante et faire des prédicats d'approximations de n-uplet d'arbres ce qui nous permettrait d'être encore plus précis en synchronisant les approximations.

Notre outil a une vocation initiale d'être très générale. Il a le même but que [22] de détecter des requêtes insatisfiables sur des programmes logique. Ils utilisent une méthode d'abduction et traitent en grande majorité des programmes quasi-cs ou ib. La détection de requête insatisfiable sur des programmes logique est également un des buts de [13] qui utilise un méthode basée sur le chainage avant pour une application exclusive au domaine de la vérification de protocole cryptographique. C'est un mécanisme très rapide mais très particulier ainsi il est incapable de détecter le vide pour le but de l'exemple 8.6.1.

Exemple 8.6.1. *Considérons le programme*
$dble(0,0) \leftarrow$
$dble(s(X), s(s(Y))) \leftarrow dble(X,Y)$
et le but $ans(X) \leftarrow dble(s(X), s(X))$.

Un premier avantage de notre outil est qu'il sur-approxime plus finement car il définit une sur-approximation non régulière, contrairement à [18, 11, 38] dont la sur-approximation est toujours régulière. Une contre partie directe c'est que cette finesse supplémentaire engendre un nombre d'états plus élevé. Un deuxième avantage est qu'il est plus pertinent pour traiter des relations notament lorsqu'il y a synchronisation. En revanche il est beaucoup moins efficace que [38] pour traiter l'atteignabilité car il a besoin actuellement de garder toute la relation. Toutefois il serait sans doute intéressant d'étudier la possibilité d'utiliser les cs-programmes pour sur-approximer le calcul de descendants.

Chapitre 9

Conclusions et perspectives

Dans ce mémoire j'ai présenté des outils basés sur la programmation logique pour manipuler les langages de n-uplets d'arbres régulier et non régulier. Grâce aux bonnes propriétés des langages de n-uplets d'arbres réguliers nous avons pu montrer comment utiliser ces outils dans le domaine de la réécriture pour résoudre une théorie du premier ordre modulo le prédicat $\downarrow_R^?$. A partir de ces mêmes méthodes on a pu définir une méthode générale pour pouvoir déterminer la satisfiabilité des formules du second ordre lorsqu'un algorithme pour donner les solutions de la partie du premier ordre correspondant existe.

Pour mettre en oeuvre les techniques concernant la résolution de la théorie pseudo-régulière du premier ordre, il reste encore du travail au niveau de l'implémantation de nos algorithmes. J'ai implanté la transformation de la section 3.4 mais il faut encore améliorer optimiser et compléter (par l'implantation du calcul du complément et l'algorithme de résolution des formules) l'application développée en swi-prolog par G.salzer[1], il serait sans doute intéressant d'effectuer une implantation avec des outils à base de logique de réécriture comme MAUDE [26] ou ELAN [20], ce qui nous permettrait d'implanter plus facilement les différentes stratégies d'application des règles de transformation.

Jusqu'à présent, nous avons donné des conditions pour que les procédures terminent mais nous n'avons pas étudié avec précision la complexité de ces procédures. Dans le cas de la résolution des formules pseudo-régulières la complexité est au moins celle de la résolution de formules diophantiennes. Ils pourrait être intéressant d'utiliser les techniques de [19, 55] afin de réduire notre complexité.

D'autre part certaines sous-classes des langages synchronisés n'ont pas encore été étudiées sous l'angle de la programmation logique, on peut citer les relations faiblement régulière de [67] qui définissent une sur-classe stricte des relations régulières, qui sont closes par intersection (contrairement au

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

langages synchronisés) et qui gardent un test du vide décidable. Par ailleurs il existe des classe de TRS dont des résultats sont obtenus à partir de leur relation avec les automates ou les automates avec contraintes comme dans [46], il serait sans doute intéressant d'étudier la possibilité d'assouplir les restrictions syntaxique de ces TRS par des techniques similaires à celles utilisées dans ce mémoire.

En examinant attentivement la méthode de résolution des formules du second ordre, on s'aperçoit qu'il serait possible d'autoriser des règles du type $f(\vec{x}) \to t_f$ où $t_f \in \mathcal{T}(\mathcal{F}, Var)$ dans les CTRS pseudo-régulier généraux, ce qui permettrait d'autoriser des redex imbriqués.

Un certain nombre d'auteurs [4, 25, 27, 49, 89] ont proposé des formalismes à base de schématisation de termes pour représenter des ensembles de termes. Ces formalismes se basent sur des répétitions motifs, contrôlés 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 des variables. Il serait cependant intéressant d'améliorer l'expressivité des cs-programmes en ajoutant des contraintes numériques dans les clauses afin de pouvoir exprimer des comptages a l'intérieur du langage. Le formalisme de la programmation logique avec contrainte [53] permet d'envisager une extension assez naturelle des cs-programmes dans cette voie. Dans ce mémoire, j'ai principalement montré des méthodes exactes pour résoudre les différents problèmes posés sur les langages et leurs applications. Ces méthodes exactes nécessitent des restrictions syntaxiques sur les objets manipulés afin de ne pas se heurter à des cas indécidables des problèmes.

Dans [41] les auteurs proposent de vérifier rapidement certaines propriétés de protocoles de logique temporelle à l'aide d'approximation, de transformation de programmes (comprenant le complément) ainsi qu'un solveur pour les contraintes numériques. Cette méthode est très adaptée pour résoudre leur problème, cependant il serait sans doute intéressant de s'en inspirer afin de pouvoir résoudre des requêtes plus générales, il pourrait être intéressant par exemple de pouvoir intégrer une technique de complétion dans notre outil d'approximations. L'outil developpé par Blanchet est pour le momment plus rapide que le notre pour des raisons d'implantation d'heuristique et de méthodes adaptées aux protocoles cryptographique. Toutefois il serait sans doute judicieux d'intégrer certaines de ces méthodes. Dans le sens inverse il serait intéressant d'introduire les prédicats d'approximations dans l'outil développé par Blanchet pour que son outil puisse traiter des cas plus généraux que les protocoles cryptographiques qu'il résoud actuellement. Il serait également intéressant à terme d'étudier la possibilité d'insérer notre programme dans AVISPA.

Enfin, l'exploitation de documents hypertextes comme les documents

XML par exemple, utilise beacoup 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 infini et avec des symboles sans arité. Dans [23], par exemple le lien entre les automates à arité non fixé et les automates classique est étudié, il serait donc intéressant d'étudier la possibilité de définir des langages de n-uplet sur ce type d'arbres et de déterminer si les résultats obtenus peuvent se transcrire dans ce cadre. Pour cela on peut raisonnablement imaginer étendre les formules sur les positions de [74] sur les chaines afin de résoudre la transposition de ces formules aux formules sur les positions dans les arbres d'arité non-fixé comme celles présentées dans [59], où imaginer encore d'autres extensions. Il pourraît être utile par exemple dans le cadre des bases de données XML pour évaluer des requêtes ou vérifier des contraintes d'intégrité.

Annexe A

Trace d'une attaque

Cette annexe présente la trace obtenue pour le problème de confidentialité du protocole cryptographique de Needham-Schroeder non corrigé.

La trace que l'on exhibe est celle obtenue en parcourant le plus petit arbre de preuve obtenu avec les clauses de $\mathcal{C}_{\text{out}}$. Elle représente l'échange de messages donné par la figure A.1

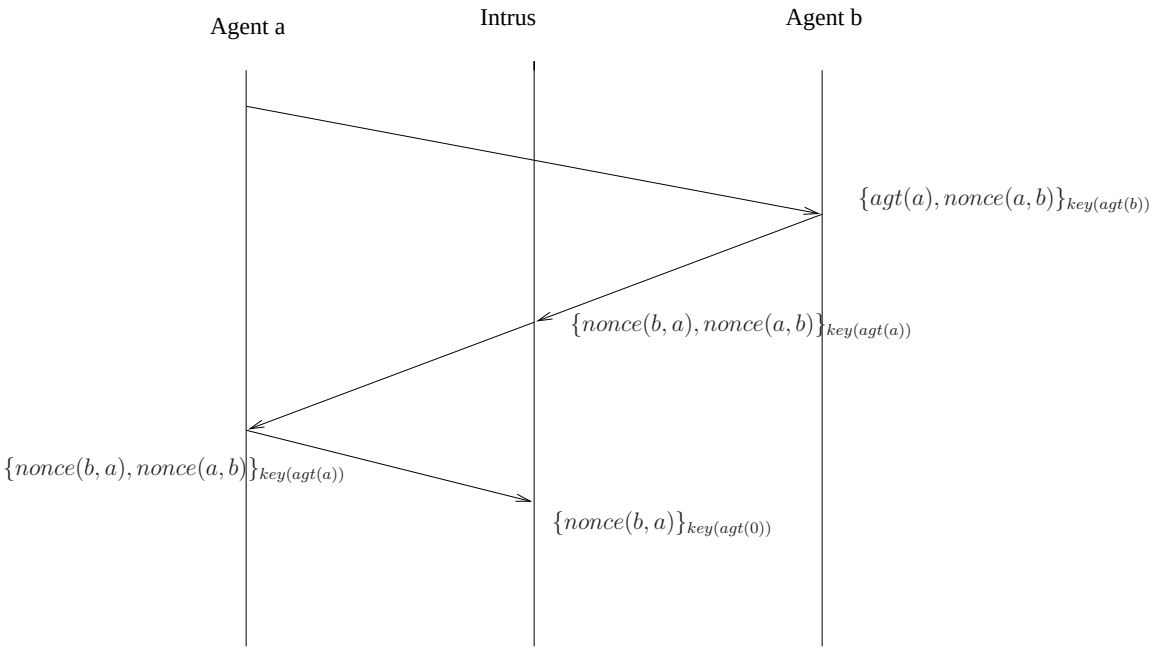


Fig. A.1 – Messages de la trace

Le résultat exposé est pratiquement celui obtenu par notre implémentation, actuellement le nom des variables est toujours celui donné par le système (par exemple $G1773$), la transformation plus lisible ci-desous n'a

pas encore été implémentée par manque de temps. Le programme d'origine et la méthode utilisée sont explicités dans les sections 8.4 et 8.5. On présentera ci-desous les clauses deux par deux. La première représente la clause dans $\mathcal{C}_{\text{out}}$ et la seconde son impact sur le dépliage des clauses. Ci dessous une variable portant le même nom dans un même couple de clause représente la même variable.

$$\begin{aligned} &but1 : -p1(X, Y) \\ &but : -p(m(agt(X), agt(Y), nonce(b, a))) \end{aligned}$$

Le but à atteindre est $p_1(X, Y)$, c'est à dire un message dont $nonce(a, b)$ n'est pas encrypté.

$$\begin{aligned} &p1(X, Y) : -agtip1(U, Z, T, V), agt1(X), agt1(Y) \\ &p(m(agt(X), agt(Y), nonce(b, a))) : - \\ &\quad p(m(Z, T, enc(key(agt(U)), nonce(b, a), V))), agti(U), agt(X), agt(Y) \end{aligned}$$

Le but est atteint après que l'intrus ait décrypté un message codé avec sa clef.

$$\begin{aligned} &agtip1(X, Y, Z, T) : -entierp1(X, Y, Z, T) \\ &agti(X) : -entier(X) \end{aligned}$$

On a simplement déplié par rapport à $agti$.

$$\begin{aligned} &agt1(b). \\ &agt(b). \end{aligned}$$

On a seulement besoin de considérer que $agt1(X)$ va correspondre à l'agent b.

$$\begin{aligned} &entierp1(0, X, Y, Z) : -p3(X, Y, Z) \\ &entier(0). \end{aligned}$$

On a simplement déplié par rapport à $entier$. On est maintenant assuré que l'on a bien codé avec la clef de l'intrus.

$$\begin{aligned} &p3(agt(X), agt(0), agt(X)) : -p4(X, Y, Z) \\ &p(m(agt(X), agt(0), enc(key(agt(0)), nonce(b, a), agt(X)))) : - \\ &\quad p(m(agt(0), agt(X), enc(key(agt(X)), nonce(X, Y), nonce(b, a), Z))) \end{aligned}$$

Maintenant on sait que X a envoyé $nonce(a, b)$ crypté avec la clef de 0 car l'intrus lui a envoyé un message encrypté avec sa clef avec un nonce que X a reconnu car c'est X qui l'a envoyé.

$$\begin{aligned} &p4(X, Y, Z) : -agtp1(X, U, T, Y, Z), agt2 \\ &p(m(agt(0), agt(X), enc(key(agt(X)), nonce(X, Y), nonce(b, a), Z))) : - \\ &\quad p(m(U, T, enc(key(agt(X)), nonce(X, Y), nonce(b, a), Z))), agt(0), agt(X) \end{aligned}$$

L'intrus a récupéré la partie encryptée d'un message qu'il a simplement envoyé à X.

$$agtp1(a, X, Y, Z, T) : -p5(X, Y, Z, T)$$
$$agt(a).$$

On sait maintenant que l'agent X est l'agent a.

$$agt2 : -entier4$$
$$agt(0) : -entier(0)$$

On vérifie juste que $agt(0)$ est bien consistant.

$$entier4.$$
$$entier(0).$$

Et c'est bien le cas.

$$p5(agt(b), agt(a), X, agt(b)) : -p6(Y, X, Z)$$
$$p(m(agt(b), agt(a), enc(key(agt(a)), nonce(a, X), nonce(b, a), agt(b)))) : -$$
$$p(m(agt(Y), agt(b), enc(key(agt(b)), agt(a), nonce(a, X), Z)))$$

Le message que l'intrus a transformé est un message que b envoie à a en réponse à un message d'initialisation de Y vers b avec encryptant $nonce(a, X)$.

$$p6(a, b, agt(a)) : -agt3, agt4$$
$$p(m(agt(a), agt(b), enc(key(agt(b)), agt(a), nonce(a, b), agt(a)))) : -agt(a), agt(b)$$

Le Y de l'initialisation c'est le a et le X c'est b. C'est donc l'initialisation du protocole, avec a envoyant $nonce(a, b)$ à b encrypté avec la clef de b

$$agt3.$$
$$agt(a).$$

On s'assure que $agt(a)$ est consistant.

$$agt4.$$
$$agt(b).$$

On s'assure que $agt(b)$ est consistant.

On remarquera qu'aucune sur-approximation n'a été introduite, et par conséquent cette trace correspond à une vraie attaque. On notera que ce n'est pas l'attaque détectée par Lowe, car on a pas imposé dans notre mondélisation que chaque individu devait effectuer obligatoirement toutes les étapes du protocole.

Bibliographie

[1] *Transformation of Logic Programs*, chapter 5, pages 697–787. Oxford University Press, 1998.

[2] Xavier Allamigeon and Bruno Blanchet. Reconstruction of Attacks against Cryptographic Protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005. IEEE Computer Society.

[3] M. Alpuente, D. Ballis, F. J. Correa, and M. Falaschi. Automated Correction of Functional Logic Programs. In P. Degano, editor, *Proc. of the European Symp. on Programming, ESOP 2003*, volume 2618 of *LNCS*, pages 54–68. Springer, 2003.

[4] Ali Amaniss, Miki Hermann, and Denis Lugiez. Set operations for recurrent term schematizations. In *TAPSOFT*, pages 333–344, 1997.

[5] H. Andréka and I. Németi. The generalised completeness of horn predicate-logic as a programming language, dai. Technical report, University of Edinburg, 1976.

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

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

[8] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Moedersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The Avispa tool for the automated validation of internet security protocols and applications. In *CAV 2005, 17th Int. Conf. on Computer Aided Verification*, volume 3576 of *LNCS*, pages 281–285, Edinburgh, Scotland, UK, July 2005. Springer-Verlag.

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

[10] F. Baader and W. Snyder. *Handbook of Automated Reasoning*, volume 1, chapter 8, pages 445–532. Elsevier Science, 2001.

[11] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *In IEEE Symposium on Security and Privacy*, pages 86–100. IEEE Computer Society, 2004.

[12] B. Blanchet and A. Podelski. Verification of cryptographic protocols : Tagging enforces termination. In Ed. A. Gordon, editor, *In Foundations of Software Science and Computation Structures (FoSSaCS 03)*, volume 2620 of *Lecture Notes in Computer Science*, pages 136–152. Springer- Verlag, 2003.

[13] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[14] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *Ninth Annual Symposium on Theorical Aspects of Computer Science*, volume 724 of *LNCS*, pages 161–171. Springer Verlag, 1992.

[15] F. Bogaert, B. Seynhaeve and S. Tison. The recognazability problem for tree automata with comparisons between brothers. In *Fondation of Software Science and Computation Structure, Second International Conference, (FoSSaCS'99)*, volume 1578 of *LNCS*, pages 150–164. Springer Verlag, 1999.

[16] Y. Boichut and T. Genet. Feasible Trace Reconstruction for Rewriting Approximations. In *RTA*, volume 4098 of *LNCS*, pages 123–135. Springer, 2006.

[17] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Handling algebraic properties in automatic analysis of security protocols. In *ICTAC 2006, Int. Colloquium on Theoretical Aspects of Computing*, volume 4281 of *LNCS*, pages 153–167, Tunis, Tunisia, November 2006. Springer.

[18] Y. Boichut, P.-C. Héam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay technique to automatically verify security protocols. In *Proc. Int. Ws. on Automated Verification of Infinite-State Systems (AVIS'2004), joint to ETAPS'04*, pages 1–11, Barcelona, Spain, April 2004. The final version will be published in EN in Theoretical Computer Science, Elsevier.

[19] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata : An overview. In *ICLP*, pages 1–19, 2002.

[20] C. Kirchner H. Moreau P.E Ringeissen C. Borovansky, P. Kirchner. An overview of elan. *Electr. Notes in Comput. Sci.*, 15, 1998.

[21] P. Bostrom, H. Idestam-Alquist. Induction of logic programs by example-guided unfolding. *Journal of Logic Programming*, 40 :159–183, 1999.

[22] Maurice Bruynooghe, Henk Vandecasteele, D. Andre de Waal, and Marc Denecker. Detecting unsolvable queries for definite logic programs. *Lecture Notes in Computer Science*, 1490 :118– ? ?, 1998.

[23] 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.

[24] A.N. Chebotarev. Construction of an automaton from a formula of the monadic first order theory of natural numbers. *Cybernetics and System Analysis*, 37(4) :540–550, 2001.

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

[26] 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.

[27] H. Common. On unification of terms with integer exponent. *Mathematical System Theory*, 28(1) :67–88, 1995.

[28] 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.

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

[30] H. Comon-Lundh and V Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In Ed. R. Nieuwenhuis, editor, *In 14th Int. Conf. Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 148–164. Springer-Verlag, 2003.

[31] D. A. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 31–41. Springer Verlag, 1995.

[32] M. Dauchet and S. Tison. Structural complexity of classes of tree languages. In M. Nivat and A. Podelski, editors, *Tree Automata and Languages*, pages 327–353. North-Holland, Amsterdam, 1992.

[33] Max Dauchet, Sophie Tison, Thierry Heuillard, and Pierre Lescanne. Decidability of the confluence of ground term rewriting systems. In *LICS*, pages 353–359, 1987.

[34] J. Denker, G. Meseguer and C. Talcott. Protocol specification and analysis in maude. In N. Heintze and Eds J. Wing, editors, *In Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

[35] N. Dershowitz and E. Pinchover. Inductive synthesis of equationnal programs. In *Proc. of the Eighth National Conference on Artificial Intelligence*, pages 234–239. AAAI press, 1990.

[36] E.L.Post. A variant of recursively unsolvable problem. *Bull. Am. Math. Soc.*, 52 :264–268, 1946.

[37] J. ENgelfriet and L. Heyker. Context-free hypergraph grammars have the same term-generating power as attribute grammar. *Acta inf.*, 29(2) :161–210, 1992.

[38] Thomas Genet et Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33(3-4) :341–383, 2004.

[39] 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.

[40] G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning*, 33 (3-4) :341–383, 2004.

[41] F. Fioravanti, A. Pettorossi, and M. Proietti. Automatic proofs of protocols via program transformation. In A. Skowron A. Szczuka M. (Eds.) Dunin-Keplicz, B. Jankowski, editor, *Monitoring, Security, and Rescue Techniques in Multiagent Systems, Advances in Soft Computing Series*, Springer, pages 99–116, 2005.

[42] 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.

[43] J. P. Gallagher and G. Puebla. Abstract interpretation over nondeterministic 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.

[44] 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.

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

[46] Guillem Godoy and Sophie Tison. On the normalization and unique normalization properties of term rewrite systems. In *CADE*, pages 247–262, 2007.

[47] V. Gouranton, P. Réty, and H. Seidl. Synchronized tree languages revisited and new applications. In *Proceedings of 6th Conference on FoSSaCS, Genova (Italy)*, volume 2030 of *LNCS*, pages 214–229. Springer, 2001.

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

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

[50] J.E Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[51] J.M Hulot. Canonical forms and unification. In *Proceedings 5th International Conference on Automated Deduction*, volume 87, pages 318–334. Springer, 1980.

[52] M. Jacquemard, F. Rusinowitch and L. Vigneron. Compiling and verifying security protocols. In *Logic for Programming, Artificial Intelligence, and Reasoning, 7th International Conference on Logic for Programming, LPAR 2000*, volume 2000 of *LNCS*, pages 131–160. Springer, 2006.

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

[54] J. L. Jensen, M. E. Jorgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 226–236, 1997.

[55] Felix Klaedtke. On the automata size for presburger arithmetic. In Harald Ganzinger, editor, *Proceedings of the Nineteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2004*, pages 110–119. IEEE Computer Society Press, July 2004.

[56] R. Moreno-Savaro J.J. Kuchen, H. Loogen and M. Rodriguez-Artalejo. The functionnal logic language babel and its implementation on a graph machine. *New Generation Computing.*, 14(4) :391–427, 1996.

[57] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Model checking linear properties of prefix-recognizable systems. In *CAV*, pages 371–385, 2002.

[58] L. Levy and M. Villaret. Linear second-order unification and context unification with tree-regular constraints. In *Proc. of the 11th Int. Conf.*

on Rewriting Techniques and Applications, RTA'00, volume 1833 of *LNCS*, pages 156–171. Springer, 2000.

[59] Leonid Libkin. Logics for unranked trees : An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[60] S. Limet and P. Pillot. Solving first order formulae of pseudo-regular theory. In *Proc. 2th Int. Conf. on Thoerical Aspect of Computing (ICTAC'05)*, volume 3091 of *LNCS*, pages 110–124. Springer, 2005.

[61] S. Limet and P. Pillot. Deciding satisfiability of positive second order joinability formulae. In *Proc. 13th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'06)*, volume 4246 of *LNAI*, pages 15–29. Springer, 2006.

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

[63] 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. Accepted for publication in Journal of Automated Reasoning.

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

[65] S. Limet and G. Salzer. Tree tuple languages from the logic programming point of view. *Journal of Automated Reasoning*, 37(4) :323–349, November 2006.

[66] 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.

[67] Sébastien Limet, Pierre Réty, and Helmut Seidl. Weakly regular relations and applications. In *12th International Conference RTA*, volume 2051 of *LNCS*, pages 185–200. Springer Verlag, 2001.

[68] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3) :131–133, 1995.

[69] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

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

[71] C. Lynch and B. Morawska. Faster basic syntactic mutation with sorts for some separable equational theories. In *16th International Conference RTA*, volume 30467 of *LNCS*, pages 241–263. Springer Verlag, 2005.

[72] J. Marcinkowski. Undecidability of the first order theory of one-step right ground rewriting. In *8th International Conference RTA*, volume 1232 of *LNCS*, pages 241–253. Springer Verlag, 1997.

[73] R. Matzinger. *Computational Representations of Models in First-Order Logic*. Dissertation, Technische Universität Wien, Austria, 2000.

[74] R. McNaughton and S. Papert. Counter-free automata. *MIT Press*, 1971.

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

[76] J. Meseguer and P. Thati. Symbolic reachabilty analysis using narrowing and its application to verification of cryptographic protocols. *Electr. Notes Theor. Comput. Sci.*, 117 :153–182, 2005.

[77] Ichiro Mitsuhashi, Michio Oyamaguchi, and Florent Jacquemard. The confluence problem for flat trss. In *AISC*, pages 68–81, 2006.

[78] T. Nagaya and Y. Toyama. Decidability for left lineatr growing term rewriting systems. *Inf. Comput.*, 178 (2) :499–514, 2002.

[79] R. M. (Roger Michael) Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), 1978.

[80] 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.

[81] F. Nielson, H. Riis Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations and spi. In *Proc. SAS'02*, number 2477 in Lecture Notes in Computer Science, pages 20–35. Springer-Verlag, 2002.

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

[83] J. Chen P. R(é)ty, J. Chabin. R-unification thanks to synchronized-contextfree tree languages. In *International Workshop on Unification (UNIF)*, April 2005.

[84] 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.

[85] 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.

[86] 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.

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

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

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] J. Thatcher and J. Wright. Generalized finite tree automata theory with an application to a descision problem of second-order logic. *Mathematical System Theory*, 2(1) :57–81, 1968.

[96] W. Thomas. *Handbook of Formal Language*, volume 3, chapter 7, pages 389–455. Springer Verlag, 1997.

[97] Ralf Treinen. The first-order theory of linear one-step rewriting is undecidable. *Theoretical Computer Science*, 208(1-2) :149–177, 1998.

[98] S-A. Tärnlund. Horn clauses computability. *BIT*, 17 :215–226, 1977.

[99] Moshe Y. Vardi. Reasoning about the past with two-way automata. In *ICALP '98 : Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 628–641, London, UK, 1998. Springer-Verlag.

[100] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In Ed. H. Ganzinger, editor, *In 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–328. Springer-Verlag, 1999.

Utilisation des langages d'arbres pour la modélisation et la vérification des systèmes à états infinis.

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. Nous avons choisi la programmation logique comme formalisme pour décrire les langages de n-uplets d'arbre (c.à.d. les relations) et les techniques de transformations de programmes pour calculer les opérations sur ceux-ci. Dans un premier temps on étudie une classe de relations closes par la plupart des opérations ensemblistes, la classe des relations pseudo-régulières. Grâce à un lien entre programmes logiques et systèmes de réécriture, nous définissons des classes de systèmes de réécriture conditionnelle dont la clôture transitive est une relation pseudo-régulière. On applique ce résultat pour donner une classe décidable de formules du premier ordre basées sur le prédicat $\downarrow_R^?$ où R est un système de réécriture conditionnelle pseudo-régulier. Ensuite on étend ce résultat au second ordre, les variables du second ordre étant interprétées comme des relations. A partir d'un algorithme général original décidant de la satisfiabilité de formules du second ordre sous certaines conditions, nous représentons une instance des variables du second ordre en système de réécriture conditionnelle. On montre que ce travail peut permettre la synthèse automatique de programme.

Dans un dernier temps, nous utilisons des sur-approximations pour des tests d'inconsistances. A cet effet, nous utilisons une classe de programmes logiques non réguliers dont le test du vide est décidable pour effectuer la sur-approximation. Nous appliquons finallement cette méthode à la vérification de protocoles cryptographiques.

Mots clés : Réécriture conditionnelle, Programmes logiques, relations régulières, second ordre, approximations.

Using tree languages for modelise and verify infinite states systems

Abstract This document presents different tools to represent and manipulate infinit sets of tree tuples called tree tuple languages. We choose logic programming as formalism to describe tree tuple languages (i.e relations) and logic program transformation techniques for computing operations on them. In a first step we study a class of relation closed under usual sets operations, the class of pseudo-regular relations. Thanks to a link between logic programs and rewrite systems, we define classes of conditionnal term rewriting system the transitive closure of which is a pseudo-regular relation. We apply this result to give a decidable class of first order formulae based on the joinability predicate $\downarrow_R^?$ where R is a pseudo-regular term rewrite system. Then we extend this result to the second order, the second order variables are interpreted as relations. From an original generic algorithm that decides the satisfiability of positive second order formulae under some conditions, we represent one particular instance in a conditionnal term rewriting system. We show that this work can be used to automatically synthetize a program.

In a last step, we use over approximations to test inconsistances tests. For this we use a non regular logic programs which have a decidable emptiness test to make the over-approximation. We finally apply this method to verify cryptographic protocols.

Keywords : Conditionnal Rewriting, logic programs, regular relations, second order, approximations.