Sur-approximations non régulières et terminaison pour l'analyse d'accessibilité

Vivien Pelletier

▶ **To cite this version:**

Vivien Pelletier. Sur-approximations non régulières et terminaison pour l'analyse d'accessibilité. Modélisation et simulation. Université d'Orléans, 2017. Français. NNT : 2017ORLE2044 . tel-01891863

HAL Id: tel-01891863
https://theses.hal.science/tel-01891863

Submitted on 10 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Centre Val de Loire UNIVERSITÉ
Spirit of the Loire Valley

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE : LIFO

Thèse présentée par :

Vivien PELLETIER

soutenue le : **23 octobre 2017**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **Informatique**

Sur-approximations non régulières et terminaison pour l'analyse d'accessibilité

THÈSE DIRIGÉE PAR :

Pierre RÉTY — Maître de conférences HDR, Univ. d'Orléans

THÈSE CO-ENCADRÉE PAR :

Yohan BOICHUT — Maître de conférences, Univ. d'Orléans

RAPPORTEURS :

Thomas GENET — Maître de conférences HDR, Univ. de Rennes 1

Pierre-Cyrille HÉAM — Professeur des univ., Univ. de Franche-Comté

EXAMINATEURS :

Sébastien LIMET — Professeur des univ., Univ. d'Orléans

Pierre-Étienne MOREAU — Professeur des univ., Univ. de Lorraine

Remerciements

Je souhaite commencer par remercier mes encadrants de thèse, Pierre Réty et Yohan Boichut. Leur investissement ainsi que leur patience, combinés à leur forte complémentarité ont fortement contribué à la qualité de ce mémoire.

Je remercie Thomas Genet et Pierre-Cyrille Héam pour avoir accepté d'être les rapporteurs de ce travail. Par leurs nombreux retours de très bonne qualité, ils ont fortement contribué à l'amélioration de ce document.

Je souhaite également remercier Sébastien Limet et Pierre-Étienne Moreau d'avoir accepté de prendre part au jury.

Je tiens à remercier tout particulièrement Anthony Perez pour le temps qu'il a accordé à la relecture de ce document.

Je remercie les membres de l'équipe LMV, de l'équipe PAMDA (dans sa version élargie), du LIFO et du pôle informatique de l'Université d'Orléans.

Je souhaite remercier particulièrement les différentes générations de jeunes docteurs et doctorants que j'ai côtoyés ces dernières années.

Je remercie mes co-bureaux pour ces nombreux moments conviviaux qui rendent le travail bien plus agréable.

Je tiens également à remercier tous les collègues du club d'électronique, ensemble nous avons fait de la vulgarisation ainsi que de la science mais de façon plus ludique. C'était très enrichissant.

Pour finir, je souhaite remercier ma famille et mes amis pour leur soutien.

Sommaire

Liste des figures

Chapitre 1

Introduction

1.1 Contexte

L'informatique est la science de l'automatisation du traitement de l'information. Cette science est issue en partie des mathématiques. Alors que les mathématiques sont l'étude de différents objets et la mise au point d'opérations sur ces derniers, l'informatique étudie le regroupement et l'interaction d'opérations successives sur de multiples objets. L'utilisation de séquences d'opérations ou encore les opérations récursives permettent la création d'algorithmes. En utilisant des algorithmes, des systèmes complexes sont conçus pour réaliser tout type de tâche. Un système complexe est un ensemble constitué d'un grand nombre d'entités en interaction qui empêchent l'observateur de prévoir sa rétroaction, son comportement ou évolution par le calcul. Un exemple de système complexe connu est le projet METEOR, le système d'exploitation automatique de la ligne 14 du métro de Paris. Dans cette section, le mot **système** représente entre autres des programmes informatiques, des protocoles ou encore des circuits logiques.

La conception de tels systèmes constitue un domaine à part entière, le génie logiciel. La notion de génie logiciel est due à l'informaticienne et mathématicienne Margaret Hamilton, la conceptrice du système embarqué du Programme Apollo. Il regroupe « l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi » (arrêté ministériel du 30 décembre 1983 relatif à l'enrichissement du vocabulaire de l'informatique [Journal officiel du 19 février 1984]). Une des difficultés majeures du génie logiciel est de s'assurer que le système est conforme, c'est-à-dire qu'il fait ce que l'on attend de lui, et fiable. Une première approche est le **test**. Un test consiste à utiliser le système sur une entrée spécifique et à comparer le résultat obtenu avec celui souhaité. Si l'on souhaite garantir la totalité du fonctionnement du système, il est nécessaire de tester l'ensemble des entrées possibles. Cependant, cette approche est rapidement limitée par le fait que l'ensemble des entrées possibles peut être très grand voire infini. Le domaine de la **vérification formelle** étudie et apporte des solutions à cette problématique.

La vérification formelle consiste à prouver des propriétés sur un système à l'aide de méthodes formelles. Les méthodes formelles peuvent être divisées en trois catégories :

— l'analyse statique par interprétation abstraite
— la vérification déductive
— la vérification de modèles.

Analyse statique par interprétation abstraite Cette méthode consiste à étudier le comportement d'un système sans l'exécuter. Elle est basée sur l'interprétation abstraite qui est une théorie d'approximation de la sémantique du système fondée sur les fonctions monotones [21]. Il est généralement impossible de pouvoir analyser exhaustivement un programme. Des **abstractions** sont donc utilisées pour transformer des problèmes vers des problèmes suffisamment simples pour être analysés. La difficulté est d'abstraire suffisamment un problème pour le rendre décidable tout en conservant assez d'informations pour vérifier les propriétés souhaitées. Cette technique est par exemple utilisée par la méthode B [1]. La méthode B a notamment été utilisée pour mettre au point le système d'exploitation automatique de la ligne 14 du métro de Paris.

Vérification déductive Dans ce cas, l'idée est d'associer au système des théorèmes représentant le comportement attendu de celui-ci. On part des informations qui sont garanties en entrée du système, appelées préconditions et on essaie de vérifier qu'on obtient bien les propriétés souhaitées à la sortie du système, appelées postconditions. Il est souvent impossible de raisonner sur l'intégralité du système d'un coup. Grâce à la logique de séparation [50], le système est découpé en éléments de plus petite taille, qui sont à leur tour découpés en plus petits éléments jusqu'à obtenir des éléments pouvant être raisonnablement prouvés. Une fois le système correctement découpé et l'ensemble des préconditions et des postconditions défini, des logiques mathématiques sont utilisées afin d'essayer de prouver qu'avec les préconditions, on obtient les postconditions. Ces techniques se basent souvent sur la logique de Hoare [34]. Cette phase est souvent effectuée avec des assistants de preuves comme Coq ou Isabelle par exemple. Cette méthode permet de vérifier la plupart des systèmes, mais demande en contre-partie un temps considérable.

Vérification de modèles Cette méthode analyse quant à elle de manière exhaustive le comportement du système. Un de ses pionniers est Amir Pnueli, qui a reçu le prix Turing pour ses travaux [49]. Avec ces méthodes, il faut représenter de manière astucieuse le comportement du système. Cette représentation constitue notre modèle. On utilise par exemple des logiques temporelles telles que LTL [1] ou encore CTL [2]. Une fois la modélisation effectuée, on cherche par exemple s'il existe dans le modèle un élément ne respectant pas les propriétés désirées.

Chaque catégorie de méthodes formelles ayant des avantages et des limitations, la plupart des travaux proposés pour la vérification formelle n'hésite pas à les combiner. De plus, les frontières entres ces méthodes sont poreuses et il n'est pas toujours évident de placer une technique dans une catégorie ou dans une autre.

Nos travaux combinent la vérification de modèles et l'interprétation abstraite. Nous cherchons à déterminer si le système peut se retrouver dans certaines **configurations**

1. Linear Temporal Logic
2. Computation Tree Logic

indésirables. Prenons comme exemple un programme qui trie des listes d'entiers. Une configuration indésirable serait d'avoir en sortie une liste non triée.

Une configuration peut être représentée de plusieurs manières. On peut utiliser des mots, c'est-à-dire une succession de symboles ou encore des termes. Un terme est une structure arborescente constituée de symboles possédant un nombre fixe de fils. Ce nombre est appelé arité.

Dans cette thèse, nous représentons une configuration par un terme. Un ensemble de configurations correspond à un ensemble de termes appelé **langage formel d'arbres**. Les modèles que nous étudions sont constitués de deux langages formels. Un langage représente l'ensemble des configurations initiales et le second représente l'ensemble des configurations indésirables.

L'ensemble des configurations initiales n'est pas suffisant pour vérifier le comportement du système. Il nous faut également modéliser la dynamique du système. Il existe plusieurs manières de modéliser la dynamique d'un système. Certains travaux utilisent des programmes logiques par exemple. Nous utiliserons des **systèmes de réécriture**.

Un système de réécriture est un ensemble de **règles**. Chaque règle permet d'obtenir une nouvelle configuration à partir d'une configuration d'origine. Notre modèle est donc constitué de trois éléments :

— deux langages formels de termes représentant l'ensemble des configurations initiales et l'ensemble des configurations indésirables
— un système de réécriture représentant la dynamique du système.

Ils sont respectivement noté $I, \mathcal{B}ad$ et R.

Une fois notre modèle défini, il faut déterminer si une configuration indésirable est accessible. Il existe deux approches différentes pour répondre à ce problème. L'approche **en avant** et l'approche **en arrière**.

L'approche en avant consiste à calculer l'ensemble des configurations dans lesquelles notre système peut être, appelé ensemble des configurations accessibles ou **ensemble des descendants**. Pour calculer cet ensemble, on applique le système de réécriture, R sur l'ensemble des configurations initiales I. On obtient l'ensemble des configurations du système à l'étape 1 noté $R(I)$. En appliquant à nouveau R sur $R(I) \cup I$, on obtient l'ensemble des configurations à l'étape suivante noté $R^2(I)$ [3]. En répétant cette opération jusqu'à ce que l'on ne génère plus de nouvelles configurations, on obtient l'ensemble des descendants noté $R^*(I)$.

Une fois l'ensemble des descendants calculé, on calcule l'intersection entre l'ensemble des descendants et l'ensemble des configurations indésirables. Si cette intersection est vide, le système ne peut pas se retrouver dans une configuration indésirable. Sinon, il peut se retrouver dans les configurations indésirables présentes dans cette intersection.

L'approche en arrière part de l'ensemble des configurations indésirables. Pour chaque configuration de cet ensemble, elle essaie d'obtenir un élément de I en appliquant le système de réécriture à l'envers. Si aucune des configurations de $\mathcal{B}ad$ ne peut avoir comme origine une configuration de I alors le système ne peut pas se retrouver dans une configuration indésirable. Sinon, on sait que des configurations indésirables sont accessibles.

3. Par souci de simplicité, nous supposons dans cette Section que $R^i(I) \subseteq R^{i+1}(I)$.

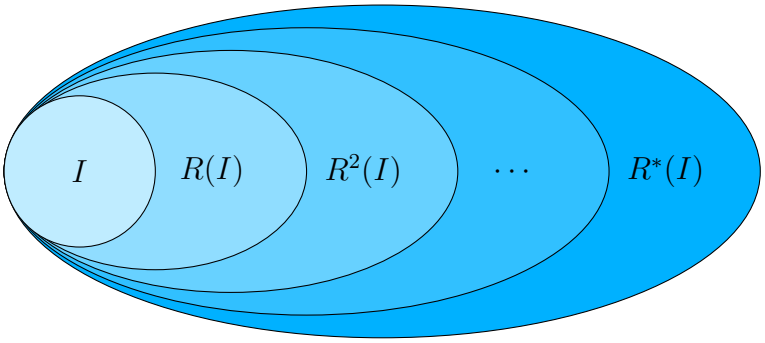
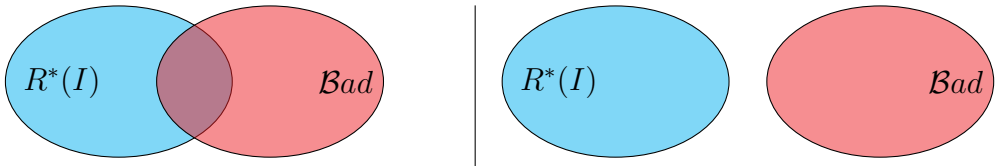


FIGURE 1.1 – Calcul de l'ensemble des descendants.



Il existe une configuration indésirable accessible

Aucune configuration indésirable n'est accessible

FIGURE 1.2 – Intersection de $R^*(I)$ et $\mathcal{B}ad$.

Les travaux présentés dans cette thèse utilisent l'approche en avant.

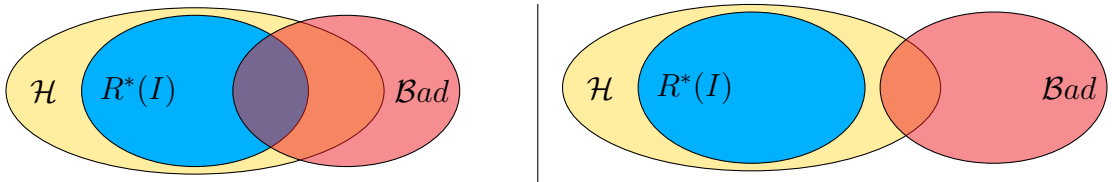
Le calcul de l'ensemble des descendants n'est pas réalisable dans le cas général. En effet, ce problème peut être réduit au problème de l'arrêt uniforme des machines de Turing, connu pour être indécidable [53].

Dans le cas où l'ensemble des descendants est fini, on peut le représenter en énumérant chaque descendant. Un système de réécriture terminant génère un nombre fini de descendants à partir d'un langage fini. Cependant, le problème de la terminaison d'un système de réécriture est indécidable dans le cas général. Ce problème a beaucoup été étudié pour les systèmes de réécriture sans stratégie. Toutefois, pour représenter le comportement de certains systèmes, une stratégie de réécriture peut être utilisée. Une stratégie de réécriture ajoute des restrictions sur la position où une règle peut être appliquée. Il est alors intéressant de transformer un système de réécriture avec stratégie en un système équivalent sans stratégie, dans la mesure où cela permet l'utilisation des techniques d'analyse de terminaison des systèmes de réécriture sans stratégie.

Si l'ensemble des descendants est infini, cela ne rend pas forcément son calcul impossible. En effet, il est possible de représenter certains ensembles infinis de termes à l'aide de structures finies. On peut utiliser par exemple des automates ou des grammaires.

Même si on ne sait comment représenter l'ensemble des descendants par une de ces structures finies, il est possible d'aborder le problème d'accessibilité à l'aide de procédures incomplètes. On n'essaie alors plus de calculer l'ensemble exact des descendants mais une sur-approximation de celui-ci, notée $\mathcal{H}$, au risque d'introduire des faux-positifs. Il sera alors possible de prouver qu'aucune configuration indésirable n'est accessible. En contrepartie, s'il l'on trouve une configuration indésirable, il ne sera pas forcément possible

de déterminer si c'est un faux-positif, c'est-à-dire si cette configuration est présente dans la sur-approximation mais pas dans l'ensemble exact des descendants.



Il existe une configuration indésirable réellement accessible

Aucune configuration indésirable n'est accessible, c'est un faux-positif

FIGURE 1.3 – Introduction des faux-positifs avec l'approximation de $R^*(I)$.

On comprend alors que la précision de la sur-approximation est déterminante pour prouver qu'aucune configuration indésirable n'est accessible. Cette précision dépend de plusieurs facteurs. Pour affiner une sur-approximation, il existe par exemple l'approche CEGAR[4] qui allie approche en avant et en arrière. À partir d'un terme présent dans la sur-approximation de l'ensemble des descendants et dans l'ensemble des configurations indésirables, cette technique essaie de retrouver l'origine de ce terme. S'il est issu d'une étape de sur-approximation, alors on modifie la sur-approximation afin de tenter de ne plus générer ce terme.

L'expressivité de la structure finie utilisée pour représenter la sur-approximation des descendants est également déterminante. Initialement, les techniques de sur-approximation des descendants utilisaient des langages réguliers, qui ne tolèrent pas de dépendances verticales ou horizontales dans les termes. Depuis peu, des techniques utilisant des langages algébriques et/ou synchronisés ont été développées [36]. Les langages algébriques et les langages synchronisés incluent tous deux les langages réguliers mais ne sont pas comparables entre eux. Les langages algébriques supportent des dépendances en profondeur et certaines dépendances en largeur. Ils sont souvent reconnus à l'aide de grammaires. Les langages synchronisés eux sont souvent reconnus par des programmes logiques. Ils ne supportent pas les dépendances en profondeur mais supportent des dépendances en largeur plus variées.

1.2 Contributions

Nous présentons trois contributions, dont deux ont fait l'objet de publications [2, 15].

Le premier apport propose d'améliorer la technique de calcul d'une sur-approximation de l'ensemble des descendants à partir d'un langage synchronisé [9]. Ces langages synchronisés sont reconnus par des CS-programmes. Les CS-programmes sont des programmes logiques avec des restrictions supplémentaires. La technique [9] a fait l'objet d'un Erratum [10], restreignant les systèmes de réécriture utilisables à des systèmes linéaires.

4. CEGAR pour Counter Example Guided Abstraction Refinement.

Nous proposons une version modifiée de cet algorithme permettant l'utilisation de systèmes de réécriture non linéaires droits et qui calculent au moins l'ensemble des descendants *innermost*.

D'autre part, nous remarquons que les systèmes de réécriture non linéaires droits génèrent des clauses qui peuvent empêcher certains descendants d'être reconnus. Nous proposons alors un processus permettant de transformer ces clauses en un ensemble de clauses qui garantit d'obtenir a minima tous les descendants au prix d'une sur-approximation supplémentaire. En utilisant ce processus avec la méthode de complétion initialement proposée, nous pouvons obtenir tous les descendants (pas seulement les descendants *innermost*) avec des systèmes de réécriture non linéaires droits.

Le second apport de cette thèse est une amélioration de la technique de transformation de programmes logiques quelconques en CS-programmes. La technique initialement proposée [41, 42] est un semi-algorithme, c'est-à-dire un algorithme qui n'a pas la garantie de s'arrêter sur toutes les entrées.

En ajoutant une règle d'inférence, la **généralisation**, nous rendons le processus terminant. Les programmes logiques quelconques ayant une puissance d'expression plus importante que les CS-programmes, la nouvelle technique proposée peut engendrer une sur-approximation du plus petit modèle de Herbrand.

Nous avons également défini une propriété de cohérence entre les différentes applications de règles d'inférence afin de garantir la préservation de certaines propriétés tout au long du processus.

Le dernier résultat est une technique de transformation d'un système de réécriture avec stratégie en un système de réécriture sans stratégie. Cela permet d'analyser la terminaison de systèmes de réécriture avec stratégie en utilisant des techniques développées pour les systèmes de réécriture sans stratégie. Ce résultat améliore également les travaux [29], en générant moins de règles.

1.3 Plan

Cette thèse est composée de six chapitres.

Après le chapitre d'introduction, le second chapitre présente les notions utilisées dans cette thèse. Il commence par une présentation des langages de mots et de termes et les opérations associées. La seconde section traite de la réécriture.

Le chapitre 3 présente le premier apport. Ces travaux ont abouti à la publication [15]. Nous proposons des améliorations sur une méthode de calcul des descendants utilisant des approximations non régulières. Après un état de l'art, nous présentons en détail les travaux [9] sur lesquels sont basés nos résultats. Nous proposons dans la section suivante une version modifiée de cette méthode de complétion pour reconnaître l'ensemble des descendants *innermost*. La dernière section propose d'ajouter un traitement dans la méthode de complétion pour permettre l'utilisation de systèmes de réécriture non linéaires droits.

Le chapitre 4 propose un algorithme terminant permettant de transformer n'importe quel programme logique en un CS-programme au prix d'une sur-approximation. Il s'agit d'une amélioration des travaux présentés par Limet et Salzer [41, 42]. La première section présente en détail le semi-algorithme de transformation de programmes logiques en CS-programme. La section suivante introduit la généralisation, qui est une règle d'inférence permettant de rendre l'algorithme terminant. Ce chapitre se conclut sur la présentation d'un exemple.

Le chapitre 5 propose la transformation de systèmes de réécriture avec stratégie en systèmes de réécriture sans stratégie. Après un état de l'art, nous présentons la technique de transformation. Les deux sections suivantes donnent respectivement la preuve que cette technique est correcte et qu'elle préserve la terminaison. La dernière section montre comment notre résultat améliore les travaux de Giesl et Middeldorp [29]. Ces résultats ont conduit à la publication [2].

Cette thèse se termine par une conclusion qui propose différentes perspectives de recherche qu'ouvrent les contributions présentées dans ce document.

Chapitre 2

Préliminaires

Dans ce chapitre, nous plaçons le contexte dans lequel nos travaux se situent. Plus précisément, nous donnons les différentes définitions utilisées dans cette thèse. Nous commençons par introduire les langages formels sur lesquels reposent l'ensemble de nos travaux ainsi que les programmes logiques qui interviennent dans les chapitres 3 et 4. Enfin, dans une seconde section, nous introduisons la réécriture qui est utilisée dans les chapitres 3 et 5.

2.1 Les langages formels

Les langages formels sont des ensembles de mots (voir section 2.1.1) ou de termes. Ils sont souvent utilisés en informatique pour représenter des ensembles d'informations. À partir de ces ensembles d'informations, il est possible de modéliser des systèmes et notamment les états dans lesquels ils peuvent être ou ils ne doivent pas être [16, 19]. Après avoir introduit les définitions de base, nous abordons le cas particulier des langages de mots qui peuvent être considérés comme un sous-ensemble des langages de termes. Dans la partie suivante (section 2.1.2), nous présentons les langages réguliers de termes, suivis d'une section sur les langages algébriques de termes (2.1.3). La section 2.1.4 présente la programmation logique et les langages synchronisés de tuples d'arbres générés à partir de programmes logiques construits avec des clauses de Horn respectant certaines contraintes. Cette section s'achève par la présentation des langages synchronisés algébriques de tuples d'arbres qui reconnaissent les langages synchronisés de tuples d'arbres et une partie des langages algébriques d'arbres, ainsi que des langages combinant des contraintes issues de ces deux classes de langages.

Les travaux de cette thèse utilisent les termes qui sont construits à partir d'un ensemble de symboles, appelé alphabet. Ces derniers possèdent chacun un nombre fixe de paramètres, appelé arité d'un symbole.

Notation 2.1.1 (Arité d'un symbole)
Le symbole s d'arité n, avec $n \in \mathbb{N}$, est noté $s^{\backslash n}$.
L'opérateur arity(s) donne l'arité du symbole s.
*Un symbole d'arité 0 est appelé **constante**.*

Définition 2.1.2 (Alphabets)
*Un **alphabet** Σ est un ensemble fini* [1] *de symboles d'arité fixe.*
On note $\Sigma^{\backslash n} \subseteq \Sigma$ l'ensemble des symboles d'arité n présents dans Σ.

Remarque 2.1.3
L'ensemble $\Sigma^{\backslash 0}$ contient l'ensemble des constantes de Σ.

On illustre les deux définitions données ci-dessus par cet exemple.

Exemple 2.1.4 (Arité d'un symbole et Alphabets)
L'ensemble $\Sigma = \{a^{\backslash 0}, b^{\backslash 0}, f^{\backslash 2}\}$ est un alphabet.
L'ensemble $\Sigma^{\backslash 2}$ des symboles d'arité 2 est $\{f^{\backslash 2}\}$.
Les symboles a et b sont des constantes, et $arity(f) = 2$.

Les termes sont construits à partir d'un alphabet et d'un ensemble de variables. Le couple formé par ces deux ensembles s'appelle une signature.

Définition 2.1.5 (Signature)
*Soient Σ un alphabet et $\mathcal{X}$ un ensemble de variables disjoints. Le couple $(\Sigma, \mathcal{X})$ est appelé **signature**.*

Définition 2.1.6 (Termes)
*Soit $(\Sigma, \mathcal{X})$ une signature. Un **terme** défini sur $(\Sigma, \mathcal{X})$ est construit inductivement à partir des règles suivantes :*
— *$x \in \mathcal{X}$ est un terme,*
— *$c \in \Sigma^{\backslash 0}$ est un terme,*
— *Soient $f \in \Sigma^{\backslash n}$ et $t_1, \ldots, t_n$ des termes, $f(t_1, \ldots, t_n)$ est un terme.*
L'ensemble des termes pouvant être générés à partir de la signature $(\Sigma, \mathcal{X})$ est noté $T(\Sigma, \mathcal{X})$.

On définit un opérateur permettant de récupérer l'ensemble des variables présentes dans un terme.

Notation 2.1.7 (Ensemble de variables)
Soit $t \in T(\Sigma, \mathcal{X})$ un terme. L'ensemble des variables présentes dans t est $Var(t) \subseteq \mathcal{X}$.

Un terme clos est un terme qui ne contient pas de variables.

Définition 2.1.8 (Termes clos)
*Soit $(\Sigma, \mathcal{X})$ une signature. Un terme t est dit **clos** si $Var(t) = \emptyset$.*
L'ensemble des termes clos, qui ne dépend pas de $\mathcal{X}$, est $T(\Sigma, \emptyset)$, noté $T(\Sigma)$.

Les termes étant définis par induction, ils sont souvent composés de termes. Les termes qui composent un terme sont appelés des sous-termes.

Définition 2.1.9 (Sous-termes)
*Soient $(\Sigma, \mathcal{X})$ une signature et $f^{\backslash n} \in \Sigma$. Soient $t_1, \ldots, t_n \in T(\Sigma, \mathcal{X})$ et $t \in T(\Sigma, \mathcal{X})$. L'ensemble des **sous-termes** de t est défini récursivement par :*
— *t est un sous-terme de t,*

1. Cela n'est pas tout le temps le cas. Mais dans le cadre des travaux présentés dans cette thèse nous ne ferons pas appel à des alphabets infinis.

— $t_1, \ldots, t_n$ *sont des sous-termes de* t *avec* $t = f(t_1, \ldots, t_n)$,
— *les sous-termes de* $t_1, \ldots, t_n$ *sont des sous-termes de* t *avec* $t = f(t_1, \ldots, t_n)$.

Remarque 2.1.10
L'ensemble des sous-termes de t privé de t est appelé **sous-termes stricts** de t.

Pour manipuler des termes, il peut être utile d'identifier chacun de ses sous-termes. Dans ses travaux, Saul Gorn [31] propose de donner une adresse à chaque nœud d'un arbre. Nous associons à chaque sous-terme une position représentée par un vecteur d'entiers naturels.

Notation 2.1.11 (Concaténation d'un élément et d'un vecteur)
L'opérateur $\cdot$ *représente la* ***concaténation*** *entre un élément et un vecteur.*

Définition 2.1.12 (Positions des termes)
Une ***position*** *est un vecteur d'éléments de* $\mathbb{N}$. *La position représentée par le vecteur vide est notée* ϵ *et appelée racine. Soit* $(\Sigma, \mathcal{X})$ *une signature. L'ensemble des positions d'un terme* $t \in T(\Sigma, \mathcal{X})$, *noté* $Pos(t)$, *est défini par induction :*
— *si* $t \in \Sigma^{\backslash 0}$ *ou* $t \in \mathcal{X}$ *alors* $Pos(t) = \{\epsilon\}$,
— *sinon si* $t = f(t_1, \ldots, t_n)$ *avec* $f \in \Sigma^{\backslash n}$ *et* $t_1, \ldots, t_n \in T(\Sigma, \mathcal{X})$ *alors*

$$Pos(t) = \{\epsilon\} \cup \big(\bigcup_{i=1}^{n} \{i\} \diamond Pos(t_i)\big)$$

où $\diamond$ *dénote la concaténation de l'élément* i *avec chaque vecteur de* $Pos(t_i)$.

Remarque 2.1.13
Pour une position autre que la racine, on omet ϵ à la fin du vecteur.

Notation 2.1.14
Soit $t \in T(\Sigma, \mathcal{X})$. *On note par :*
— $t(p) \in \Sigma \cup \mathcal{X}$ *le symbole ou la variable apparaissant à la position* p *dans le terme* t.
— $t_{|p} \in T(\Sigma, \mathcal{X})$ *le sous-terme présent à la position* p *dans* t.
— $Pos^{Var}(t) \subseteq Pos(t)$ *l'ensemble des positions des variables d'un terme, c'est-à-dire,* $\forall p \in Pos(t)$, $p \in Pos^{Var}(t) \Leftrightarrow t(p) \in Var(t)$.
— $Pos^{NonVar}(t)$ *l'ensemble* $Pos(t) \setminus Pos^{Var}(t)$.

Une même variable peut apparaître à plusieurs positions dans un terme. Les termes qui ne sont pas dans ce cas sont dits linéaires.

Définition 2.1.15 (Termes linéaires)
Un terme t *est* ***linéaire*** *si pour tout* $x \in Var(t)$, x *n'apparaît qu'une seule fois dans* t.

Ci-dessous, nous donnons des exemples pour illustrer les définitions que nous venons d'introduire.

Exemple 2.1.16 (Termes)
Soient $\Sigma = \{a^{\backslash 0}, h^{\backslash 1}, f^{\backslash 2}\}$ et $\mathcal{X} = \{x, y\}$. Le terme $t = f(h(f(x, a)), f(x, h(a)))$ est un terme de $T(\Sigma, \mathcal{X})$. On peut également le représenter sous forme d'arbre (figure 2.1). De plus, notons que :

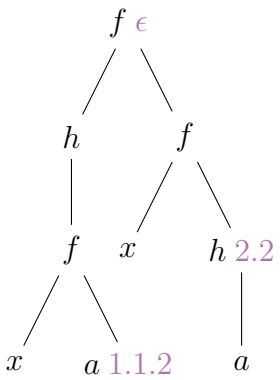


FIGURE 2.1 – Représentation sous forme d'arbre du terme $f(h(f(x,a)),f(x,h(a)))$. Certaines positions apparaissent à coté du symbole.

— t n'est pas linéaire car la variable x apparaît deux fois
— $Var(t) = \{x\}$, donc t n'est pas clos
— $Pos^{Var}(t) = \{1 \cdot 1 \cdot 1, 2 \cdot 1\}$
— $t(2 \cdot 2) = h$ et $t_{|2\cdot 2} = h(a)$ qui est un terme clos
— $t_{|1} = h(f(x,a))$ est un terme linéaire.

Nous allons manipuler les termes tout au long de cette thèse. La première opération consiste à remplacer dans un terme un symbole par un autre. On peut également remplacer un sous terme par un autre.

Notation 2.1.17 (Remplacement)
Soient $(\Sigma, \mathcal{X})$ une signature et $t, t' \in T(\Sigma, \mathcal{X})$ des termes. Soient $p \in Pos(t)$ une position dans le terme t, $n = arity(t(p))$ et $h \in \Sigma^{\backslash n}$ un symbole.
— *remplacement de symbole : $t\{h\}_p$ est obtenu en remplaçant $t(p)$ par h [2] à la position p.*
— *remplacement de sous terme : $t[t']_p$ est obtenu en remplaçant $t_{|p}$ par t' dans t à la position p.*

Exemple 2.1.18 (Remplacement)
Soient $\Sigma = \{f^{\backslash 3}, g^{\backslash 1}, h^{\backslash 1}, a^{\backslash 0}\}$ un $\mathcal{X} = \{x, y, z\}$ un ensemble de variables. Soient $t = f(x, g(a), z)$ et $t' = h(z)$ des termes. On a :
— $t\{h\}_2 = f(x, h(a), z)$
— $t[t']_2 = f(x, h(z), z)$
— $t[t']_{2.1} = f(x, g(h(z)), z)$.

Définition 2.1.19 (Contexte)
*Soient $(\Sigma, \mathcal{X})$ une signature et $\Box \notin \Sigma \cup \mathcal{X}$ un symbole d'arité 0 appelé **marque**. On appelle **contexte** sur $(\Sigma, \mathcal{X})$ tout terme de $T(\Sigma \cup \{\Box\}, \mathcal{X})$ dans lequel $\Box$ apparait exactement une fois. Soient C un contexte et t un terme. On note $C[t]$ le terme obtenu en remplaçant $\Box$ par t dans C.*

La substitution est l'une des opérations essentielles à la manipulation des termes. Elle consiste à remplacer toutes les occurrences d'une variable par un terme.

2. Il faut que $arity(h) = arity(t(p))$

Définition 2.1.20 (Substitution)
*Soient Σ un alphabet et les ensembles de variables $\mathcal{X}, \mathcal{X}'$ tel que $\mathcal{X} \subseteq \mathcal{X}'$. Une **substitution** est une application $\sigma : \mathcal{X} \to T(\Sigma, \mathcal{X}')$.*
Une substitution peut s'appliquer sur un terme. Soient $t \in T(\Sigma, \mathcal{X}')$ un terme et σ une substitution. L'application de σ sur t notée σ_t est définie par induction :

- *Si t est une constante, c'est-à-dire $t \in \Sigma^{\backslash 0}$ alors $\sigma_t = t$.*
- *Si $t \in \mathcal{X}$ alors $\sigma_t = \sigma(t)$.*
- *Si $t \in \mathcal{X}' \setminus \mathcal{X}$ alors $\sigma_t = t$.*
- *Si $t = f(t_1, \ldots, t_n)$ avec $f \in \Sigma^{\backslash n}$ et $t_1, \ldots, t_n \in T(\Sigma, \mathcal{X}')$ alors $\sigma_t = f(\sigma_{t_1}, \ldots, \sigma_{t_n})$.*

Notation 2.1.21
Par souci de simplicité (mais par abus de notation), nous utiliserons $\sigma(t)$ pour dénoter σ_t. De plus, pour indiquer que le terme x doit être substitué par le terme a, nous utiliserons x/a. Enfin, soient σ, σ' des substitutions et x une variable. On note $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$.

Exemple 2.1.22 (Substitution)
Soient $\Sigma = \{f^{\backslash 2}, g^{\backslash 1}, a^{\backslash 0}\}$ et l'ensemble de variables $\mathcal{X} = \{x, y, z\}$. Soient $t = f(g(x), f(a, f(x, y))$ un terme et la substitution $\sigma = \{x/g(z), y/a\}$. On obtient alors $\sigma(t) = f(g(g(z)), f(a, f(g(z), a)))$.

Définition 2.1.23 (Matching)
*Soient $(\Sigma, \mathcal{X})$ une signature et $t, t' \in T(\Sigma, \mathcal{X})$ deux termes. Le terme t **matche** t' s'il existe une substitution σ telle que $\sigma(t) = t'$.*

Exemple 2.1.24 (Matching)
Soient $\Sigma = \{f^{\backslash 2}, g^{\backslash 1}, a^{\backslash 0}\}$ un alphabet et $\mathcal{X} = \{x, y, z\}$ un ensemble de variables. Soient $t = f(g(x), y)$ et $t' = f(g(a), z) \in T(\Sigma, \mathcal{X})$. Dans la substituion $\sigma = \{x/a, y/z\}$ le terme t matche t' car $\sigma(t) = t'$. L'inverse n'est cependant pas vérifié.

L'unification permet aussi de rendre deux termes égaux à l'aide de substitutions. Mais contrairement au matching, elle permet d'instancier les variables des deux termes.

Définition 2.1.25 (Unification)
*Soient $(\Sigma, \mathcal{X})$ une signature et $t, t' \in T(\Sigma, \mathcal{X})$ deux termes. Les termes t et t' sont **unifiables** s'il existe une substitution σ appelée unifieur telle que $\sigma(t) = \sigma(t')$.*

Définition 2.1.26 (Unifieur le plus général)
Soient $(\Sigma, \mathcal{X})$ une signature et $t, t' \in T(\Sigma, \mathcal{X})$ deux termes. Un unifieur σ de t et t' tel que :

$$\forall \alpha,\ \alpha(t) = \alpha(t') \Rightarrow \exists\ \lambda, \alpha = \lambda \circ \sigma$$

*est appelé **unifieur le plus général** et noté mgu[3]. Un tel unifieur est unique à renommage des variables près.*

Exemple 2.1.27 (Unification)
Soient $\Sigma = \{f^{\backslash 3}, g^{\backslash 1}, a^{\backslash 0}\}$ un alphabet et $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$ un ensemble de variables. Soient $t = f(g(x_1), x_1, x_2)$ et $t' = f(x_3, a, g(x_4)) \in T(\Sigma, \mathcal{X})$ deux termes. On a :

- $\alpha = \{x_1/a, x_3/g(x1), x_2/g(a), x_4/x_1\}$ est un unifieur car $\alpha(t) = \alpha(t') = f(g(a), a, g(a))$.
- $\sigma = \{x_1/a, x_3/g(x_1), x_2/g(x_4)\}$ est un mgu. En effet on a $\sigma(t) = \sigma(t') = f(g(a), a, g(x_4))$ et par exemple $\lambda = \{x_4/a\}$ est telle que $\alpha = \sigma \circ \lambda$.

3. Most General Unifier

— α n'est pas un mgu. On a par exemple σ qui respecte $\sigma(t) = \sigma(t')$ mais il n'existe aucune substituion λ telle que $\sigma = \lambda \circ \alpha$.
— t ne matche pas t' et t' ne matche pas t.

Maintenant que nous avons correctement défini les termes, nous pouvons introduire la notion de langage formel.

Définition 2.1.28 (Langages formels de termes)
Un langage formel de termes est un ensemble de termes. On notera $\mathcal{L}(O)$ le langage généré par le générateur O.

2.1.1 Langages formels de mots et générateurs associés

Comme dit précédemment, les langages de mots peuvent être considérés comme un sous-ensemble des langages de termes. Ils sont notamment utilisés dans le chapitre 5 pour reconnaître les positions où il est permis de réécrire. De plus, les outils de génération ainsi que la hiérarchisation de langages formels sont plus simples dans le cas particulier des langages de mots.

Définition 2.1.29 (Mots)
Soit Σ un alphabet constitué uniquement de symboles d'arité 1.Soit Φ une constante appelée symbole de fin de mot.
*L'ensemble des **mots** définis sur l'alphabet Σ est l'ensemble des termes $T(\Sigma \cup \{\Phi\})$ noté $\mathcal{W}(\Sigma)$.*

Notation 2.1.30
On notera souvent les mots en omettant le symbole de fin de mot.

On donne ci-dessous un exemple de langage de mots.

Exemple 2.1.31
Soit $\Sigma = \{a^{\backslash 1}, b^{\backslash 1}, c^{\backslash 1}\}$ un alphabet et Φ le symbole de fin de mot. $a(b(a(c(a(\Phi)))))$ est un mot formé à partir de cet alphabet. Il peut aussi être noté $abaca$. $\{a, aa, ba, ca, bac, cab, baba, caca, abaca, acaba\}$ est un langage de mots construit à partir de l'alphabet Σ.

Remarque 2.1.32
Vu que cette section traite uniquement des mots, l'arité des symboles de l'alphabet sera toujours de 1 et donc ne sera plus indiquée.

Les langages de mots pouvant être infinis, l'énumération des éléments de ceux-ci ne constitue pas une méthode suffisante pour les définir. Il existe donc plusieurs outils pour reconnaître un langage comme les expressions rationnelles, les grammaires ou encore les automates. La plupart de ces outils ne peuvent pas reconnaître l'ensemble des langages formels. Afin d'identifier facilement quels ensembles de langages peuvent être reconnus par certains outils, Noam Chompsky propose une hiérarchisation des classes de langages de mots [17].

Langages réguliers ou rationnels La classe la plus restrictive est appelée langage de type 3 par Chompsky, mais plus communément langages réguliers ou langages rationnels. Cette classe regroupe les langages stables par opérations rationnelles.

Définition 2.1.33 (Opérations rationnelles)
*Soient A et B deux langages. Il existe trois **opérations rationnelles** :*

- *la **concaténation** de A et B est le langage AB qui est constitué des mots xy avec $x \in A$ et $y \in B$.*
- *l'**union** de A et B est le langage noté $A \cup B$ défini par $x \in A \cup B \Leftrightarrow x \in A \vee x \in B$.*
- *L'**étoile de Kleene** d'un langage A est le plus petit langage A^* qui contient le mot vide ϵ, le langage A et qui est clos par concaténation.*

On illustre l'utilisation de ces opérateurs sur des langages finis de mots.

Exemple 2.1.34 (Opérations rationnelles)
Soient les langages $A = \{a, ab\}$ et $B = \{c, cd\}$. La concaténation $AB = \{ac, acd, abc, abcd\}$, l'union $A \cup B = \{a, ab, c, cd\}$ et l'étoile de Kleene $A^* = \{\epsilon, a, ab, aa, aab, aba, abab, \ldots\}$.

Les langages rationnels possèdent également l'avantage d'être stables par intersection, c'est-à-dire que l'intersection de deux langages rationnels est un langage rationnel. De plus, l'inclusion ($L \subseteq L'$), le test du vide ($L = \emptyset$), l'universalité ($L = \Sigma^*$) et l'appartenance ($w \in L$) sont décidables (avec Σ un alphabet, L, L' des langages réguliers et w un mot).

Les expressions rationnelles permettent de décrire des langages rationnels de façon plutôt lisible.

Définition 2.1.35 (Expressions rationnelles, [38])
*Soient ϵ le mot vide, Σ un alphabet, et $Q = \{^*, ^+\}$ un ensemble de quantifieurs. On note $()$ l'opérateur parenthèses et $|$ l'opérateur ou logique. Les **expressions rationnelles** sont définies par induction selon les règles suivantes :*

- *soit ω une expression rationnelle constituée uniquement de symboles de Σ, alors cette expression reconnaît le mot ω.*
- *$(r_1|r_2)$ est une expression rationnelle qui reconnaît les mots reconnus par r_1 ou par r_2.*
- *$(r_1)(r_2)$ est une expression rationnelle qui reconnaît les mots xy avec x un mot reconnu par r_1 et y un mot reconnu par r_2.*
- *$(r_1)^+$ est une expression rationnelle qui reconnaît les mots $\omega_1\omega_2\ldots$ avec $\omega_1, \omega_2, \ldots$ des mots reconnus par r_1.*
- *$(r_1)^*$ est équivalent à $(\epsilon|(r_1)^+)$.*
- *(r_1) est équivalent à r_1.*

avec r_1 et r_2 deux expressions rationnelles.

Remarque 2.1.36
On peut omettre les parenthèses suivant la convention suivante. Les quantifieurs sont les plus prioritaires suivis de l'opérateur de concaténation puis de l'opérateur $|$. La concaténation et l'opérateur $|$ sont associatifs par la gauche.

Exemple 2.1.37 (Expressions rationnelles)
Soit l'alphabet $\Sigma = \{a, b, c\}$, le langage reconnu par l'expression $a^+(b^*(c|a))(b^*(c|a))$ contient par exemple les mots acc, $ababa$, $aabcba$ et $abbbbbbcbc$.

Remarque 2.1.38
Dans cette thèse, on se permettra d'utiliser la même syntaxe que les expressions rationnelles pour exprimer des comptages [4] dans des termes sur des symboles unaires ou des mots. Pour ce faire, on ajoute le quantifieur ^{k} avec $k \in \mathbb{N}$ défini comme $(r_1)^k$ est équivalent à $\underbrace{(r_1)(r_1)\ldots}_{\text{k fois}}$.

Les grammaires formelles sont un autre outil pour reconnaître des langages de mots.

Définition 2.1.39 (Grammaires formelles)
*Une **grammaire** est définie par un quadruplet $\mathcal{G} = (N, \Sigma, P, S)$ où :*
- *Σ est l'alphabet à partir duquel les mots seront construits.*
- *N est l'ensemble des symboles non terminaux, qui est disjoint de Σ.*
- *P est un ensemble de règles de production de la forme*

$$(\Sigma \cup N)^* \times N \times (\Sigma \cup N)^* \to (\Sigma \cup N)^*$$

- *$S \in N$ le symbole non terminal initial.*

Pour générer un mot à partir d'une grammaire, on utilise les règles de production pour réduire les non terminaux (dérivation). On commence avec le mot constitué uniquement du symbole S. Un mot est accepté uniquement s'il ne contient plus aucun symbole non terminal.

Les grammaires formelles sans restrictions peuvent reconnaître des langages qui ne sont pas rationnels. En ajoutant certaines limitations à celles-ci, on obtient des grammaires qui reconnaissent exclusivement les langages réguliers. Ces grammaires s'appellent grammaires rationnelles ou régulières. Avant de définir ces dernières, nous avons besoin d'introduire la notion de linéarité d'une règle de production de grammaire.

Définition 2.1.40 (Grammaires linéaires droites ou linéaires gauches)
Soit $\mathcal{G} = (N, \Sigma, P, S)$, une grammaire.

- *$\mathcal{G}$ est **linéaire gauche** si P est un ensemble de règles dont les membres droits sont de la forme $(N \cup \{\epsilon\})(\Sigma \cup \{\epsilon\})$.*
- *$\mathcal{G}$ est **linéaire droit** si P est un ensemble de règles dont les membres droits sont de la forme $(\Sigma \cup \{\epsilon\})(N \cup \{\epsilon\})$.*

Définition 2.1.41 (Grammaires régulières)
Une grammaire $\mathcal{G} = (N, \Sigma, P, S)$ est régulière si elle respecte les contraintes suivantes :
- *$\forall\, p \in P$, le membre gauche de p est un non terminal.*
- *$\mathcal{G}$ est soit linéaire gauche, soit linéaire droit.*

L'exemple 2.1.42 représente le langage de l'exemple 2.1.37 avec une grammaire régulière linéaire droite.

Exemple 2.1.42 (Grammaires régulières)
Soit $\mathcal{G} = (N, \Sigma, P, S)$ une grammaire définie par :
- $N = \{S, A, B_1, B_2, C\}$,
- $\Sigma = \{a, b, c\}$,

4. La mise en place de comptages nous fait sortir de la classe des langages réguliers.

— $P = \left\{ \begin{array}{llll} S \to A, & A \to aA, & A \to aB_1, & B_1 \to bB_1, \\ B_1 \to C, & C \to aB_2, & C \to cB_2, & B_2 \to bB_2, \\ B_2 \to a, & B_2 \to c & & \end{array} \right\}$

On peut obtenir le mot $aabcba$ avec la dérivation suivante : $S \to A \to aA \to aaA \to aaB_1 \to aabB_1 \to aabC \to aabcB_2 \to aabcbB_2 \to aabcba$

Les automates finis permettent également de reconnaître les langages rationnels.

Définition 2.1.43 (Automates finis non-déterministes)
Un automate est un quintuplet de la forme $\mathcal{A} = (\Sigma, Q, q_0, \Delta, F)$ où :

- *Σ est l'alphabet à partir duquel les mots seront construits,*
- *Q est l'ensemble des états de l'automate,*
- *$q_0 \in Q$ est l'état initial de l'automate,*
- *Δ est l'ensemble des triplets représentant une transition de la forme $Q \times \Sigma \times Q$,*
- *$F \subseteq Q$ est l'ensemble des états acceptants.*

Pour reconnaître un mot, on part de l'état initial avec le mot que l'on cherche à reconnaître. Ensuite, on utilise une des transitions qui part de l'état actuel et consomme le symbole en début de mot puis change l'état. On répète cette étape tant qu'il reste des symboles dans le mot. Une fois tous les symboles consommés, si on est dans un état final, le mot est alors reconnu.

L'exemple 2.1.44 représente le langage de l'exemple 2.1.37 avec un automate fini déterministe.

Exemple 2.1.44 (Automates finis)
Soit $\mathcal{A} = (\Sigma, Q, q_0, \Delta, F)$ un automate fini où :

- $\Sigma = \{a, b, c\}$
- $Q = \{q_0, q_1, q_2, q_f\}$,
- $\Delta = \left\{ \begin{array}{lll} (q_0, a, q_0) & (q_0, a, q_1) & (q_1, b, q_1) \\ (q_1, a, q_2) & (q_1, c, q_2) & (q_2, b, q_2) \\ (q_2, a, q_f) & (q_2, c, q_f) & \end{array} \right\}$
- $F = \{q_f\}$.

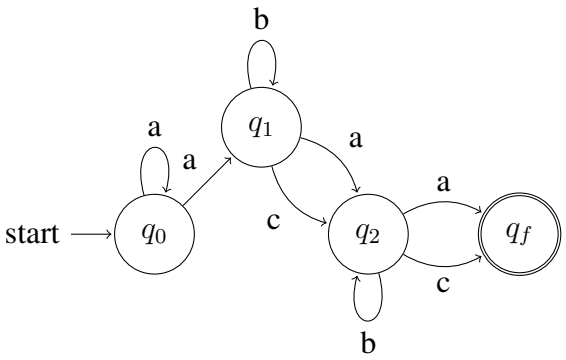


FIGURE 2.2 – Automate reconnaissant le langage $a^+(b^*(c|a))^2$.

Langages algébriques Chomsky définit ensuite les langages de type 2 aussi appelés langages algébriques. Cette classe inclut les langages rationnels, mais contient également d'autres langages. Contrairement aux langages rationnels, ils permettent les comptages, par exemple pour reconnaître le langage des mots $a^n b^n$. En revanche, cette classe n'est plus stable par certains opérateurs, par exemple l'intersection entre deux langages algébriques n'est pas forcement un langage algébrique. L'inclusion ($L \subseteq L'$), la disjonction ($L \cap L' = \emptyset$) et l'universalité ($L = \Sigma^*$) ne sont pas décidables pour les langages algébriques (avec Σ un alphabet, L, L' des langages algébriques). Par contre, l'appartenance et le test du vide le sont. L'intersection entre un langage rationnel et un langage algébrique est un langage algébrique.
Les langages algébriques peuvent être reconnus par des grammaires algébriques.

Définition 2.1.45 (Grammaires algébriques)
Une grammaire $\mathcal{G} = (N, \Sigma, P, S)$ est algébrique si chaque membre gauche des règles de production est composé d'un symbole non terminal. En d'autres termes, les règles de production doivent être de la forme $N \to (N \cup \Sigma)^*$.

On présente un exemple de grammaire algébrique qui reconnaît le langage $a^n b^n$.

Exemple 2.1.46 (Grammaires algébriques)
La grammaire $\mathcal{G} = (\{S\}, \{a, b\}, P, S)$ avec $P = \{S \to aSb; S \to \epsilon\}$ reconnaît le langage $a^n b^n$. Le mot $aaabbb$ peut par exemple être obtenu avec la dérivation $S \to aSb \to aaSbb \to aaaSbbb \to aaabbb$.

Une amélioration des automates finis non-déterministes permet de reconnaître les langages algébriques. En effet, en ajoutant une mémoire représentée par une pile, nous pouvons compter le nombre de fois que certaines transitions ont été effectuées et ainsi reconnaître par exemple le langage $a^n b^n$. Pour ce faire, chaque transition empile ou dépile des symboles dans la mémoire ou ne modifie pas la pile.

Définition 2.1.47 (Automates à pile non-déterministes)
*Un **automate à pile non-déterministe** est un septuplet $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, q_0, Z, F)$ avec :*
- *Q l'ensemble des états de l'automate,*
- *Σ l'alphabet des symboles à partir duquel les mots seront construits,*
- *Γ l'alphabet des symboles de pile,*
- *Δ l'ensemble des transitions de la forme $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$,*
- *q_0 l'état initial de l'automate,*
- *$Z \in \Gamma$ le symbole initial de la pile,*
- *$F \subseteq Q$ l'ensemble des états finaux.*

Le fonctionnement des automates à pile non-déterministes est basé sur celui des automates finis non-déterministes avec la gestion de la pile en plus. On commence avec une pile contenant le symbole Z. Une transition ne peut être appliquée que si le symbole en haut de la pile est le même que celui du membre gauche de la règle de transition. À chaque transition, on dépile le symbole de Γ présent dans le membre gauche de la règle de transition et on empile les symboles de Γ présents dans le membre droit, s'il y en a. Un mot est reconnu quand l'automate est sur un état final.

On présente en exemple un automate à pile reconnaissant le langage algébrique $a^n b^n$.

Exemple 2.1.48 (Automates à pile non-déterministes)
$\mathcal{A} = \{q_0, q_1, q_f\}, \{a, b\}, \{Z, \alpha\}, \Delta, q_0, Z, \{q_f\})$ avec

$$\Delta = \left\{ \begin{array}{lll} q_0, a, Z \to q_0, \alpha Z & q_0, a, \alpha \to q_0, \alpha\alpha & q_0, \epsilon, \alpha \to q_1, \alpha \\ q_0, \epsilon, Z \to q_1, Z & q_1, b, \alpha \to q_1, \epsilon & q_1, \epsilon, Z \to q_f, Z \end{array} \right\}$$

La figure 2.3 représente le même automate.

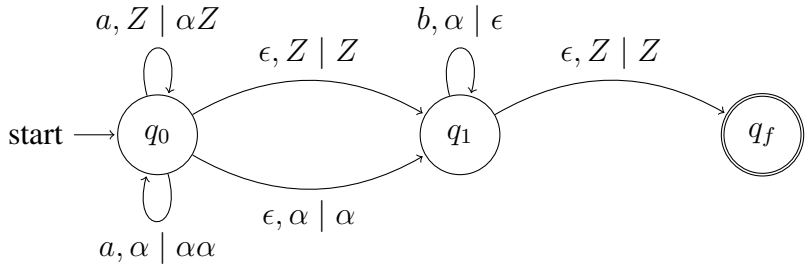


FIGURE 2.3 – Automate à pile non-déterministe reconnaissant le langage $a^n b^n$.

Langages contextuels Chomsky définit ensuite les langages de type 1, aussi appelés langages contextuels. Cette classe de langages inclut les langages algébriques. On peut reconnaître des langages de cette classe à l'aide de grammaires contextuelles.

Définition 2.1.49 (Grammaires contextuelles)
*Soient $\mathcal{G} = (N, \Sigma, P, S)$ une grammaire, A un non terminal et $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ avec $\gamma \neq \epsilon$. La grammaire $\mathcal{G}$ est dite **contextuelle** si toutes ses règles de production sont de la forme $\alpha A \beta \to \alpha\gamma\beta$.*

Langages récursivement énumérables La dernière classe de langages définis par Chomsky, les langages de type 0 sont aussi appelés les langages récursivement énumérables. Ces langages sont reconnus par les grammaires formelles sans restrictions, comme elles ont été introduites dans la définition 2.1.39. Cette classe de langages inclut les langages de type 1. On peut reconnaître les langages de type 0 également à l'aide d'une machine de Turing.

Définition 2.1.50 (Machines de Turing)
Une machine de Turing un septuplet $(Q, \Gamma, b, \Sigma, q_0, \Delta, F)$ où :

- *Q est un ensemble fini d'états.*
- *Γ est l'alphabet des symboles de la bande (mémoire).*
- *$b \in \Gamma$ est un symbole particulier (dit blanc).*
- *Σ est l'alphabet des symboles en entrée ($\Sigma \subseteq \Gamma$ privé de $\{b\}$).*
- *$q_0 \in Q$ est l'état initial.*
- *Δ est l'ensemble des relations de transitions qui sont de la forme $Q \times \Gamma \to Q \times \Gamma \times \{\hookleftarrow, \hookrightarrow\}$.*
- *$F \subseteq Q$ est l'ensemble des états acceptants (ou finaux, terminaux).*

Pour utiliser une machine de Turing, on commence dans l'état q_0 avec en memoire un mot de $\mathcal{W}(\Sigma)$ et la tête est positionnée sur le premier symbole de ce mot. On choisit ensuite une transition compatible, c'est-à-dire qui a dans son membre gauche le même état que l'état actuel de la machine et le même symbole de Γ que celui présent au niveau

de la tête. On applique la transition en remplaçant le symbole de Γ présent au niveau de la tête par celui du membre droit de la règle de transition. On passe ensuite à la case adjacente de la bande en fonction de la direction indiquée dans la règle de transition et on met la machine dans l'état présent dans le membre droit de cette même règle. On effectue les transitions les unes après les autres. Le mot initialement présent sur la bande est reconnu s'il est possible d'arriver dans un des états acceptants.

On peut par exemple modéliser le langage de l'ensemble des formules logiques vraies construites à partir des littéraux $\top$ (vrai), $\bot$ (faux) et de l'opérateur $\wedge$ (et logique).

Exemple 2.1.51
Soit la machine de Turing $\mathcal{M} = (Q, \Gamma, b, \Sigma, q_v, \Delta, \{q_{acc}\})$ avec :
- $Q = \{q_v, q_{op}, q_e, q_{acc}\}$,
- $\Gamma = \{\top, \bot, \wedge, \text{`` ''}\}$,
- $b = \text{`` ''}$,
- $\Sigma = \{\top, \bot, \wedge\}$,
- $\Delta = \left\{ \begin{array}{ll} (q_v, \top) \to (q_{op}, \text{`` ''}, \hookrightarrow); & (q_v, \bot) \to (q_e, \text{`` ''}, \hookrightarrow); \\ (q_{op}, \wedge) \to (q_v, \text{`` ''}, \hookrightarrow); & (q_{op}, \text{`` ''}) \to (q_{acc}, \top, \hookrightarrow) \end{array} \right\}$

En utilisant cette machine sur le mot d'entrée $\top \wedge \top$, on passe dans l'état q_{op} avec le mot $\wedge\top$ sur la bande, puis $(q_v, \top)$, $(q_{op}, \text{`` ''})$ et enfin $(q_{acc}, \text{`` ''})$. L'état q_{acc} étant acceptant, $\top \wedge \top$ est reconnu par $\mathcal{M}$.

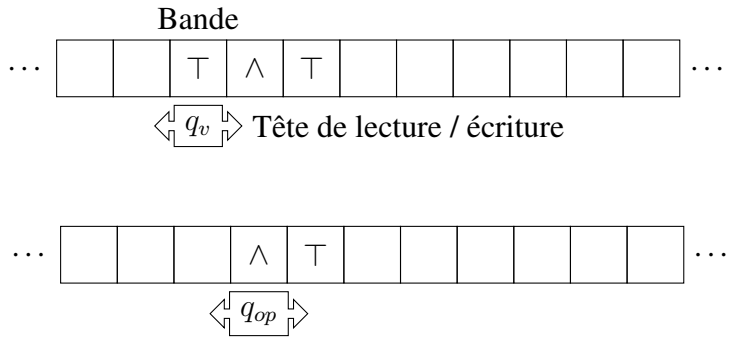


FIGURE 2.4 – Illustration de la machine de Turing de l'exemple 2.1.51

Les langages contextuels (de type 1) peuvent être reconnus par des automates linéairement bornés, qui sont des machines de Turing à bande finie.

2.1.2 Langages d'arbres réguliers

La notion de classe de langages existe également pour les langages de termes. Les langages de termes réguliers peuvent être reconnus par des grammaires d'arbres régulières. Commençons par introduire la notion de grammaires d'arbres.

Définition 2.1.52 (Grammaires d'arbres)
Soient le quadruplet $\mathcal{G} = (N, \Sigma, P, S)$ et $\mathcal{X}$ un ensemble de variables où :

— *N est l'ensemble des symboles non terminaux. Chaque symbole non terminal possède une arité (qui peut être supérieure à 0).*
— *Σ est l'alphabet.*
— *P est l'ensemble des règles de production de la forme $T(\Sigma \cup N \cup \mathcal{X}) \to T(\Sigma \cup N \cup \mathcal{X})$.*
— *S est le non terminal initial.*

*Le quadruplet $\mathcal{G}$ est appelé **grammaire d'arbres**.*

En contraignant les grammaires d'arbres, on peut obtenir des grammaires qui reconnaissent uniquement les langages réguliers d'arbres.

Définition 2.1.53 (Grammaires d'arbres régulières)
*Une grammaire $\mathcal{G} = (N, \Sigma, P, S)$ est **régulière** si :*
— *N est un ensemble de non terminaux d'arité 0 et*
— *chaque règle de production de P est de la forme $A \to T(\Sigma \cup N)$ avec $A \in N$.*

On peut par exemple générer le langage qui représente l'ensemble des listes d'entiers naturels pairs. Pour représenter les naturels, nous allons utiliser les entiers de Peano.

Définition 2.1.54 (Entiers de Peano)
L'ensemble des entiers naturels est défini comme le langage reconnu par la grammaire :

$$\mathbb{N}_P = (\{S_{Nat}\}, \{s^{\backslash 1}, 0^{\backslash 0}\}, \{S_{Nat} \to s(S_{Nat}); S_{Nat} \to 0\}, S_{Nat})$$

Le terme 0 représente l'entier 0, le terme $s(0)$ représente l'entier 1 et l'entier $s(x)$ avec x un terme reconnu par $\mathbb{N}_P$ est le successeur de l'entier x.

On peut ainsi représenter n'importe quel entier naturel en utilisant uniquement deux symboles différents ($s^{\backslash 1}$ et $0^{\backslash 0}$).

Exemple 2.1.55 (Entier de Peano)
Le nombre 3 est représenté par le terme $s(s(s(0)))$.

On peut donc maintenant à l'aide des entiers de Peano et des grammaires d'arbres régulières reconnaître l'ensemble des listes contenant des entiers pairs.

Exemple 2.1.56 (Grammaires d'arbres régulières)
Soit $\mathcal{G} = (\{S, S_{even}\}, \{c^{\backslash 2}, nil^{\backslash 0}, s^{\backslash 1}, 0^{\backslash 0}\}, P, S)$ avec :

$$P = \left\{ \begin{array}{ll} S \to c(S_{even}, S)\,; & S \to nil\,; \\ S_{even} \to s(s(S_{even}))\,; & S_{even} \to 0 \end{array} \right\}$$

Le terme $c(s(s(s(s(0)))), c(0, nil))$ fait partie du langage $\mathcal{L}(\mathcal{G})$ et représente la liste $[4, 0]$.

On peut également reconnaître les langages d'arbres réguliers à l'aide d'automates [20], notamment en utilisant des automates finis d'arbres ascendants non-déterministes.

Définition 2.1.57 (Automates finis d'arbres ascendants non-déterministes)
Soit $\mathcal{X}$ un ensemble de variables. Un automate fini d'arbres ascendant non-déterministe est un quadruplet $\mathcal{A} = (Q, \Sigma, \Delta, Q_f)$ tel que :
— *Q est l'ensemble des symboles d'états de l'automate. Ces symboles sont d'arité 1.*
— *Σ est l'alphabet.*

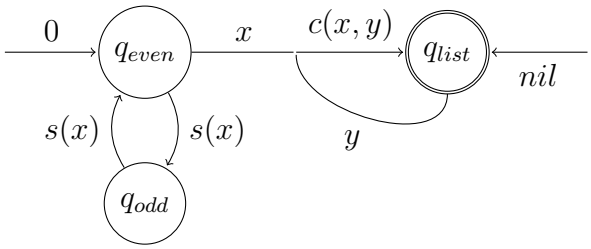


FIGURE 2.5 – Automate fini d'arbre ascendant non deterministe.

— Δ *est l'ensemble des règles de transitions de la forme* $f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n))$, *avec* $f \in \Sigma^{\backslash n}$, $q, q_1, \ldots q_n \in Q$ *et* $x_1, \ldots, x_n \in \mathcal{X}$.
— $Q_f \subseteq Q$ *est l'ensemble des états acceptants.*

L'exemple 2.1.58 reconnaît le langage de l'exemple 2.1.56 à l'aide d'un automate fini d'arbres ascendant non-déterministe.

Exemple 2.1.58 (Automates finis d'arbres ascendants non-déterministes)
Soit $\mathcal{A} = (\{q_{odd}, q_{even}, q_{list}\}, \{c^{\backslash 2}, nil^{\backslash 0}, s^{\backslash 1}, 0^{\backslash 0}\}, \Delta, \{q_{list}\})$ avec :

$$\Delta = \left\{ \begin{array}{lll} 0 \to q_{even}(0)\,; & s(q_{even}(x)) \to q_{odd}(s(x))\,; & s(q_{odd}(x)) \to q_{even}(s(x))\,; \\ nil \to q_{list}(nil)\,; & c(q_{even}(x), q_{list}(y)) \to q_{list}(c(x,y)) & \end{array} \right\}$$

La figure 2.5 représente le même automate.

Reconnaître les langages d'arbres réguliers à l'aide d'automates peut être fait en partant des feuilles de l'arbre (les constantes) et en allant vers la racine, comme le font les automates d'arbres ascendant non-déterministe. Mais il est également possible de le faire de la racine aux feuilles. On parlera dans ce cas d'automates d'arbres descendants non-déterministes.

Définition 2.1.59 (Automates finis d'arbres descendants non-déterministes)
Soit $\mathcal{X}$ *un ensemble de variables. Un automate fini d'arbres descendant non-déterministe est un quintuplet* $\mathcal{A} = (Q, \Sigma, \Delta, Q_i)$ *tel que*
— Q *est l'ensemble des symboles d'états de l'automate. Ces symboles sont d'arité 1.*
— Σ *est l'alphabet.*
— Δ *est l'ensemble des règles de transitions de la forme* $q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n))$, *avec* $f \in \Sigma^{\backslash n}$, $q, q_1, \ldots q_n \in Q$ *et* $x_1, \ldots, x_n \in \mathcal{X}$.
— $Q_i \subseteq Q$ *est l'ensemble des états initiaux.*

Voici la version descendante de l'automate présenté dans l'exemple 2.1.58.

Exemple 2.1.60 (Automates finis d'arbres descendants non-déterministes)
Soit $\mathcal{A} = (\{q_{odd}, q_{even}, q_{list}\}, \{c^{\backslash 2}, nil^{\backslash 0}, s^{\backslash 1}, 0^{\backslash 0}\}, \Delta, \{q_{list}\})$ avec :

$$\Delta = \left\{ \begin{array}{lll} q_{list}(c(x,y)) \to c(q_{even}(x), q_{list}(y))\,; & q_{list}(nil) \to nil\,; & q_{odd}(s(x)) \to s(q_{even}(x))\,; \\ q_{even}(s(x)) \to s(q_{odd}(x))\,; & q_{even}(0) \to 0 & \end{array} \right\}$$

La figure 2.6 représente cet automate à l'aide d'un graphe.

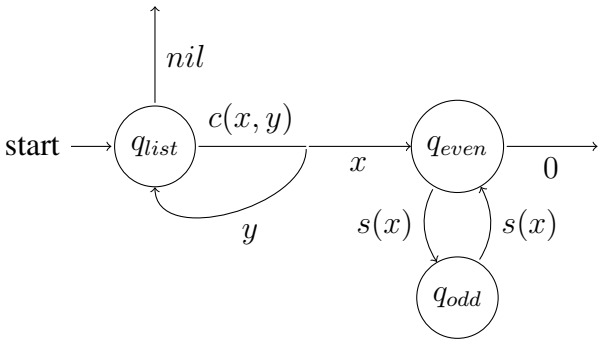


FIGURE 2.6 – Automate fini d'arbre descendant non deterministe.

Les automates d'arbres finis non-déterministes, qu'ils soient descendants ou ascendants peuvent reconnaître les langages d'arbres réguliers. Cependant, la version déterministe [5] de ces automates ne reconnaît pas les mêmes classes de langages. En effet, la version ascendante qu'elle soit déterministe ou non, reconnaît les langages réguliers d'arbres, alors que la version déterministe des automates d'arbres finis descendants ne reconnaît qu'un sous-ensemble des langages réguliers.

2.1.3 Langages algébriques d'arbres

Comme pour les langages de mots, il existe la classe des langages algébriques d'arbres. Cette classe inclut les langages réguliers d'arbres. Les langages algébriques d'arbres peuvent être reconnus en utilisant des grammaires algébriques d'arbres.

Définition 2.1.61 (Grammaires algébriques d'arbres [51])
Soit $\mathcal{X}$ un ensemble de variables. Une ***grammaire algébrique d'arbres*** *est une grammaire d'arbres $\mathcal{G} = (N, \Sigma, P, S)$ où toutes les règles de P sont de la forme $A(x_1, \ldots, x_{arity(A)}) \to T(N \cup \Sigma, \{x_1, \ldots, x_{arity(A)}\})$, avec $A \in N$ et $x_1, \ldots, x_{arity(A)} \in \mathcal{X}$.*

Cette grammaire fonctionne comme une grammaire générale d'arbre. Pour illustrer son utilisation, nous présentons un exemple qui génère l'ensemble des inéquations formées à partir des entiers naturels de Peano et de l'opérateur $\geq$.

Exemple 2.1.62 (Grammaires algébriques d'arbres)
Soit $\mathcal{X} = \{x, y\}$. Soit la grammaire algébrique d'arbres

$$\mathcal{G} = (\{S^{\backslash 0}, Geq^{\backslash 2}, Nat^{\backslash 0}\}, \{0^{\backslash 0}, s^{\backslash 1}, \geq^{\backslash 2}\}, P, S)$$

avec :

$$P = \left\{ \begin{array}{lll} S \to Geq(Nat, 0)\,; & Nat \to 0\,; & Nat \to s(Nat)\,; \\ Geq(x,y) \to Geq(s(x), s(y))\,; & Geq(x,y) \to \geq (x,y)\,; & \end{array} \right\}$$

On peut par exemple générer le terme représentant $2 \geq 1$ avec la dérivation :

$$\begin{array}{rcl} S & \to & Geq(Nat, 0) \to Geq(s(Nat), s(0)) \to Geq(s(s(Nat)), s(0)) \\ & \to & Geq(s(s(0)), s(0)) \to \geq (s(s(0)), s(0)) \end{array}$$

5. A savoir celle où l'on n'autorise pas deux transitions ayant le même membre gauche à un renommage de variables près et un membre droit différent dans Δ.

L'ordre des dérivations peut être restreint. On différencie entre autres deux restrictions opposées, une priorisant la dérivation des non terminaux au niveau des feuilles de l'arbre (IO) et l'autre au niveau de la racine (OI) [23, 24].

Définition 2.1.63 (Grammaire algébrique d'arbres avec stratégie de dérivation IO)
Soient $\mathcal{X}$ un ensemble de variables et $\mathcal{G} = (N, \Sigma, P, S)$ une grammaire algébrique d'arbres. Une dérivation d'un terme $t \in T(N \cup \Sigma, \mathcal{X})$ à la position $p \in Pos(t)$ est IO *si les sous-termes stricts de $t_{|p}$ sont de la forme $T(\Sigma, \mathcal{X})$.*

Définition 2.1.64 (Grammaires algébriques d'arbre avec stratégie de dérivation OI)
Soient $\mathcal{X}$ un ensemble de variables et $\mathcal{G} = (N, \Sigma, P, S)$ une grammaire algébrique d'arbres. Une dérivation d'un terme $t \in T(N \cup \Sigma, \mathcal{X})$ à la position $p \in Pos(t)$ est OI *s'il n'y a pas de symboles non terminaux entre la position p et la racine.*

Remarque 2.1.65
Les langages générés avec la stratégie de dérivation OI sont identiques à ceux générés en utilisant une stratégie arbitraire. Cependant, la stratégie de dérivation IO peut reconnaître des langages qui ne peuvent être reconnus par une stratégie arbitraire. La réciproque est également vraie.

Pour reconnaître les langages algébriques d'arbres, on peut également utiliser des automates d'arbres pushdown [33].

Définition 2.1.66 (Automates d'arbres pushdown)
Soient $\mathcal{X}$ et $\mathcal{X}'$ deux ensembles de variables. Un ***automate d'arbres pushdown*** *est un sextuplet $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ tel que :*

- *Q est un ensemble fini d'états d'arité 2,*
- *Σ est un alphabet fini d'arité fixe de symboles de sortie* [6]*,*
- *Γ est un alphabet fini de symboles d'arité fixe, appelé alphabet pushdown*
- *$q_0 \in Q$ est l'état initial*
- *$Z_0 \in \Gamma^{\backslash 0}$ est le symbole initial pushdown*
- *P est l'ensemble des règles pouvant avoir deux formes :*
  - *les* ***règles de lecture*** *qui sont de la forme :*

$$q(f(x_1, \ldots, x_n), \alpha(x'_1, \ldots, x'_k)) \to f(q_1(x_1, \pi_1), \ldots, q_n(x_n, \pi_n))$$

  - *les* ***ϵ-règles*** *qui sont de la forme $q(x, \alpha(x'_1, \ldots, x'_k)) \to q'(x, \pi)$*

  avec $f \in \Sigma^{\backslash n}$, $\alpha \in \Gamma^k$, $q, q', q_1, \ldots, q_n \in Q$, $x, x_1, \ldots, x_n \in \mathcal{X}$, $x'_1, \ldots, x'_k \in \mathcal{X}'$ et $\pi_1, \ldots, \pi_n \in T(\Gamma, \{x'_1, \ldots, x'_k\})$.

Les ensembles de variables $\mathcal{X}$ et $\mathcal{X}'$ acceptent des substitutions respectivement définies sur $T(\Sigma, \mathcal{X})$ et $T(\Gamma, \mathcal{X}')$. Une transition acceptante de l'automate est de la forme :

$$q(a, \alpha(x'_1, \ldots, x'_k)) \to a$$

avec $q \in Q$, $a \in \Sigma^{\backslash 0}$, $\alpha \in \Gamma^k$ et $x'_1, \ldots, x'_k \in \mathcal{X}'$. Le langage reconnu par un automate d'arbre pushdown $\mathcal{A}$ est le langage $\{t \in T(\Sigma) \mid q_0(t, Z_0) \to_A^* t\}$.

6. L'alphabet de sortie est l'alphabet utilisé par les termes reconnus

Pour illustrer l'utilisation des automates d'arbres pushdown, considérons à nouveau l'exemple 2.1.62.

Exemple 2.1.67 (Automates d'arbres pushdown)
Soient $\mathcal{X} = \{x, y\}$ et $\mathcal{X}' = \{x'\}$. Soit l'automate d'arbre pushdown $\mathcal{A} = (\{q_0, q_1, q_2\}, \{0^{\backslash 0}, s^{\backslash 1}, \geq^{\backslash 2}\}, \{Z_0^{\backslash 0}, \alpha^{\backslash 1}\}, q_0, Z_0, \Delta)$ avec :

$$\Delta = \left\{ \begin{array}{ll} q_0(x, x') \to q_0(x, \alpha(x'))\,; & q_0(\geq (x, y), x') \to \geq (q_1(x, x'), q_2(y, x'))\,; \\ q_1(s(x), \alpha(x')) \to s(q_1(x, x')) & q_1(0, Z_0) \to 0\,; \\ q_1(s(x), x') \to s(q_1(x, x'))\,; & q_2(s(x), \alpha(x')) \to s(q_2(x, x'))\,; \\ q_2(0, Z_0) \to 0 & \end{array} \right\}$$

Le terme $t = \geq (s(s(0)), s(0))$ signifiant $2 \geq 1$ peut par exemple être reconnu avec la dérivation

$$\begin{array}{rcl} q_0(t, Z_0) & \to & q_0(t, \alpha(Z_0)) \\ & \to & \geq (q_1(s(s(0)), \alpha(Z_0)), q_2(s(0), \alpha(Z_0))) \\ & \to & \geq (s(q_1(s(0), Z_0)), q_2(s(0), \alpha(Z_0))) \\ & \to & \geq (s(q_1(s(0), Z_0)), s(q_2(0, Z_0))) \\ & \to & \geq (s(s(q_1(0, Z_0))), s(q_2(0, Z_0))) \\ & \to & \geq (s(s(q_1(0, Z_0))), s(0)) \\ & \to & t \end{array}$$

Il existe également les grammaires linéaires indexées d'arbres pour reconnaître les langages algébriques d'arbres [36].

Définition 2.1.68 (Grammaires linéaires indexées d'arbres)
*Une **grammaire linéaire indexée d'arbres** est un quintuplet $\mathcal{G} = (\Sigma, N, \Gamma, P, S)$ où :*

- *Σ est un alphabet*
- *N est un ensemble fini de non terminaux d'arité zéro*
- *Γ est un alphabet fini de symboles de mots d'index*
- *P est un ensemble fini de règles de la forme $\gamma \to t$ avec $\gamma \in \langle N \times \mathcal{W}(\Gamma)\rangle$ et $t \in T(\Sigma \cup \langle N \times \mathcal{W}(\Gamma)\rangle)$.*
- *S est le non terminal initial.*

Pour effectuer une dérivation dans un terme t à la position $p \in Pos(t)$ avec la règle $\langle A, w'\rangle \to t'$, il est nécessaire que $t_{|p} = \langle A, w'w\rangle$ [7]. La dérivation donne le nouveau terme $t_{new} = t[t'']_p$ [8] avec :

$$\forall p' \in Pos(t') \; si \; t'_{|p'} = \langle B, w''\rangle \; alors \; t'' = t'[\langle B, w''w\rangle]_{p'} \; [9]$$

Notation 2.1.69
Les couples de la forme $\langle N \times \mathcal{W}(\Gamma)\rangle$ seront notés $A[\alpha\alpha\beta]$ avec $A \in N$ et $\alpha\alpha\beta \in W(\Gamma)$.

Pour illustrer les grammaires linéaires indexées d'arbres, nous allons considérons à nouveau l'exemple 2.1.62.

7. Le mot d'index du membre gauche de la règle de dérivation doit être un préfixe du mot d'index du non terminal dérivé.
8. On remplace le sous terme à la position de la dérivation par celui nouvellement obtenu.
9. Pour chaque non terminal présent dans le membre droit de la règle, on copie le mot d'index du non terminal dérivé privé de son suffixe et on le concatène à celui du non terminal courant.

Exemple 2.1.70
Soit $\mathcal{G} = (\{0^{\backslash 0}, s^{\backslash 1}, \geq^{\backslash 2}\}, \{S, Geq, A, B\}, \{\alpha, \beta\}, P, S)$ une grammaire linéaire indexée d'arbres avec :

$$P = \left\{ \begin{array}{lll} S \to Geq[\beta]\,; & Geq \to Geq[\alpha]\,; & Geq \to \geq (A, B)\,; \\ A[\alpha] \to s(A)\,; & A[\beta] \to 0\,; & A \to s(A)\,; \\ B[\alpha] \to s(B)\,; & B[\beta] \to 0 & \end{array} \right\}$$

Le terme représentant $2 \geq 1$ peut par exemple être généré avec la dérivation :

$$\begin{array}{rcl} S[\,] & \to & Geq[\beta] \\ & \to & Geq[\alpha\beta] \\ & \to & \geq (A[\alpha\beta], B[\alpha\beta]) \\ & \to & \geq (s(A[\beta]), B[\alpha\beta]) \\ & \to & \geq (s(A[\beta]), s(B[\beta])) \\ & \to & \geq (s(s(A[\beta])), s(B[\beta])) \\ & \to & \geq (s(s(A[\beta])), s(0)) \\ & \to & \geq (s(s(0)), s(0)) \end{array}$$

2.1.4 Langages synchronisés de tuples d'arbres et Programmes logiques

Les grammaires algébriques d'arbres avec stratégie arbitraire, les automates d'arbres pushdown et les grammaires linéaires indexées d'arbres ne peuvent pas générer des langages de la forme $\{f(t, t)\}$ avec $f \in \Sigma^{\backslash 2}$ et $t \in T(\Sigma)$. Cependant, les grammaires IO mais également des langages synchronisés de tuples d'arbres en sont capables.
Pour reconnaître des langages synchronisés de tuples d'arbres, nous allons utiliser des programmes logiques. Les programmes logiques ayant la puissance des machines de Turing, ils peuvent reconnaître les langages récursivement énumérables. Les CS-programmes [32, 39, 40] sont des programmes logiques avec des restrictions sur la forme des clauses. Ces restrictions permettent notamment que le test d'appartenance, le test du vide et l'intersection entre un langage synchronisé de tuples d'arbres et un langage régulier d'arbres soient décidables.
Un CS-programme est construit sur un alphabet et un ensemble de prédicats d'arité fixe, ainsi qu'un ensemble de variables. Des atomes sont alors construits à partir de ces trois ensembles.

Définition 2.1.71 (Atomes)
Soient $(\Sigma, \mathcal{X})$ une signature. Soient $t_1, \ldots, t_n \in T(\Sigma, \mathcal{X})$ des termes et P un prédicat d'arité n. On dit que $P(t_1, \ldots, t_n)$ est un ***atome****.*

Notation 2.1.72
On utilisera la lettre A pour les atomes. Soient i un entier positif et $A = P(t_1, \ldots, t_n)$ un atome, on note $A|_i$ le terme t_i. L'opérateur $Pred$ donne le prédicat de l'atome, $Pred(A) = P$.

Un atome A est clos si pour tout i le terme $A|_i$ est clos. Les opérateurs *arity* et Var sont naturellement étendus sur les atomes. De même, l'opérateur Var est étendu sur les ensembles d'atomes de sorte que, pour $B = \{A_1, \ldots, A_n\}$ on ait $Var(B) = \bigcup_{\forall A \in B} Var(A)$.

Définition 2.1.73 (Atomes linéaires)
*Un atome A est **linéaire** si $\forall x \in Var(A)$, x n'apparaît qu'une seule fois dans A.*

Exemple 2.1.74 (Atomes)
Soient l'alphabet $\Sigma = \{a^{\backslash 0}, f^{\backslash 2}, g^{\backslash 1}\}$, le prédicat $P^{\backslash 3}$ et l'ensemble de variables $\mathcal{X} = \{x, y, z\}$.
- $A = P(f(x, g(a)), x, f(a, y))$ est un atome
- $A|_3 = f(a, y)$
- $Pred(A) = P$
- $Var(A) = \{x, y\}$ et
- A n'est pas linéaire car x apparaît deux fois dans A.

Grâce aux atomes, on peut définir les clauses de Horn. Dans cette thèse nous considérerons uniquement les clauses de Horn strictes et exprimées par une implication.

Définition 2.1.75 (Clauses de Horn)
*Soient $H, A_1, \ldots, A_n$ des atomes et $B = A_1 \wedge A_2 \wedge \ldots \wedge A_n$. Une **clause de Horn** est une expression du type $H \leftarrow B$.*
Les clauses de Horn sont supposées disjointes deux à deux[10]*. Le membre gauche de la clause est appelé **tête** de clause et le membre droit **corps**. Les clauses ayant comme corps l'ensemble vide d'atomes sont appelées des **faits**.*

Notation 2.1.76
Dans la suite de cette thèse, nous noterons en général $B = A_1 \wedge A_2 \wedge \ldots \wedge A_n$ sous la forme $B = A_1, \ldots, A_n$ plutôt qu'en utilisant le symbole de conjonction $\wedge$. Par abus de notation, nous notons de manière indifférenciée la conjonction $A_1, \ldots, A_n$ et l'ensemble $\{A_1, \ldots, A_n\}$. L'opérateur $B|_i$ représente ainsi A_i.

Pour définir les CS-clauses, il nous faut d'abord définir quelques propriétés sur les ensembles d'atomes.

Définition 2.1.77 (Linéaire et plat)
Soit $B = A_1, \ldots, A_n$. On dit que :
- *B est **plat** si pour tout $A \in B$ et pour tout $1 \leq i \leq arity(Pred(A))$, $A|_i$ est une variable.*
- *B est **linéaire** si pour tout $x \in Var(B)$, x n'apparaît qu'une seule fois dans B*[11]*.*

Remarque 2.1.78
L'ensemble vide d'atomes $\emptyset$ est linéaire et plat.

Nous pouvons maintenant définir les CS-clauses.

Définition 2.1.79 (CS-clauses)
*La clause de Horn $H \leftarrow B$ est une **CS-clause** si B est plat et linéaire.*

Définition 2.1.80
Un programme logique est un ensemble de clauses de Horn.

10. Elles ne partagent pas de variables. Le même nom de variables sera parfois utilisé dans des clauses différentes, mais ces variables seront cependant indépendantes.

11. La notion de linéarité d'un ensemble d'atome implique que chaque atome de cet ensemble soit linéaire et qu'ils ne partagent pas de variables entre eux.

Définition 2.1.81 (CS-programmes)
Un CS*-programme est un programme logique composé de* CS*-clauses.*

Les CS-clauses peuvent être séparées en plusieurs catégories.

Définition 2.1.82
Soient $(\Sigma, \mathcal{X})$ une signature et $t_1, \dots t_n \in T(\Sigma, \mathcal{X})$. La CS*-clauses $P(t_1, \dots, t_n) \leftarrow B$ est :*
- vide*, si $t_1, \dots, t_n$ sont des variables.*
- normalisée*, si pour tout entier $i \in [1, n]$, t_i est soit une variable soit un terme de la forme $f(x_1, \dots, x_k)$ avec $f \in \Sigma^{\backslash k}$ et $x_1, \dots, x_k \in \mathcal{X}$.*
- synchronisante*, si B est composé d'un seul atome.*
- copiante*, si $P(t_1, \dots, t_n)$ n'est pas linéaire.*
- préservante*, si $Var(P(t_1, \dots, t_n)) \subseteq Var(B)$.*

Un CS-programme est normalisé si toutes ses clauses sont normalisées.

Exemple 2.1.83 (CS-clauses)
Soient l'alphabet $\Sigma = \{a^{\backslash 0}, f^{\backslash 2}, g^{\backslash 1}\}$, l'ensemble de prédicats $\{P^{\backslash 3}, Q_1^{\backslash 1}, Q_2^{\backslash 2}\}$ et l'ensemble de variables $\mathcal{X} = \{x, y, z\}$.
- La clause $P(x, g(y), a) \leftarrow Q_1(x), Q_2(x, y)$ n'est pas une CS-clause car le corps de la clause n'est pas linéaire, la variable x apparaissant deux fois.
- $Q_2(x, y) \leftarrow$ est un fait non copiant et vide.
- $P(f(x, y), x, y) \leftarrow Q_2(x, y)$ est une CS-clause normalisée, copiante, synchronisante et préservante.

Définition 2.1.84 (Dérivations)
Soient $Prog$ un programme logique et $G = A_1, \dots, A_n$ et $G' = A'_1, \dots, A'_n$ des ensembles d'atomes.
- ***Dérivation*** *$G \rightsquigarrow_{H \leftarrow B, [\sigma]} G'$:*
  G se dérive en G' par une étape de résolution si $Prog$ contient une clause $H \leftarrow B$ et un atome $A = G|_i$ tels qu'il existe un mgu σ avec $\sigma(A) = \sigma(H)$ et $G' = \sigma(G)[\sigma(B)]_i$.
- ***Dérivation faible*** *$G \rightarrow_{H \leftarrow B, [\sigma]} G'$:*
  G se réécrit en G' si $Prog$ contient une clause $H \leftarrow B$ et un atome $A = G|_i$ tels qu'il existe σ issue du matching de A et H [12] et $G' = G[\sigma(B)]_i$.

Notation 2.1.85
La clôture transitive et réflexive de $\rightsquigarrow$ est dénotée par $\rightsquigarrow^*$, et la clôture transitive et réflexive de $\rightarrow$ par $\rightarrow^*$. Quand le contexte sera clair et pour simplifier les notations, on omettra la référence à la substitution ou à la clause.

Exemple 2.1.86 (Dérivations)
Soient $Prog = \{P(x_1, g(x_2)) \leftarrow P'(x_1, x_2).\ P(f(x_1), x_2) \leftarrow P''(x_1, x_2).\}$ un programme logique et $G = P(f(x), y)$. On a :

- $P(f(x), y)) \rightsquigarrow_{[P(x_1, g(x_2)) \leftarrow P'(x_1, x_2), \sigma]} P'(f(x), x_2)$ avec $\sigma = \{x_1/f(x), y/g(x_2)\}$ par la clause $P(x_1, g(x_2)) \leftarrow P'(x_1, x_2)$
- D'autre part, $P(f(x), y)) \rightarrow_\alpha P''(x, y)$ avec $\alpha = \{x_1/x, x_2/y\}$ par la clause $P(f(x_1), x_2) \leftarrow P''(x_1, x_2)$.

12. Rappelons que $A = \sigma(H)$

Théorème 2.1.87 (Modèle de Herbrand)**.** *Soient A un atome clos et $Prog$ un programme logique. L'atome A appartient au **modèle de Herbrand** si et seulement si $A \rightsquigarrow^*_{Prog} \emptyset$. On note $A \in Mod(Prog)$.*

Définition 2.1.88 (Langage d'un programme logique)
*Soit $Prog$ un programme logique défini sur un alphabet Σ. Le **langage** de $P \in Pred(Prog)$, noté $\mathcal{L}_{Prog}(P)$, est l'ensemble des tuples de termes clos $\overrightarrow{t} \in T(\Sigma)$ tel que $P(\overrightarrow{t}) \in Mod(Prog)$. Par abus de notation, on note $\mathcal{L}_{Prog}(P)$ par $\mathcal{L}(P)$. Le langage de $Prog$ noté $\mathcal{L}(Prog)$ est l'union des langages reconnus par ses prédicats, c'est-à-dire, $\mathcal{L}(Prog) = \bigcup_{P \in Prog} \mathcal{L}(P)$.*

Exemple 2.1.89 (Langage d'un programme logique)
Soit $\Sigma = \{f^{\backslash 2}, g^{\backslash 1}, a^{\backslash 0}, b^{\backslash 0}\}$. Soient $A = P(f(g(a), g(b)))$ et $A' = P(f(g(a), b))$ deux atomes clos. Dans le programme logique $Prog = \{P(f(x, y)) \leftarrow Q(x, y).\ Q(g(x), g(y)) \leftarrow Q(x, y).\ Q(a, b).\}$, on a :

— $A \in Mod(Prog)$
— $A' \notin Mod(Prog)$
— $\mathcal{L}(P) = \{f(g^n(a), g^n(b)) | n \geq 0\}$

La classe des langages reconnus par les CS-programmes inclut les langages réguliers d'arbres. Cependant, certains langages algébriques d'arbres ne peuvent être reconnus par les CS-programmes et la réciproque est également vraie.

Exemple 2.1.90 (Langages synchronisés de tuples d'arbres non algébrique)
Soient $\Sigma = \{f^{\backslash 2}, a^{\backslash 0}\}$ un alphabet et $Prog = \{P(f(x, x)) \leftarrow P(x).\ P(a, a).\}$ un programme logique. Le langage reconnu par $Prog$ est le langage des arbres binaires parfaits [13] construit sur l'alphabet Σ.

Les langages algébriques d'arbres ne supportent des dépendances en largeur que sur des termes constitués de symboles d'arité inférieure ou égale à 1. Ce langage peut dont être reconnu par un CS-programme, mais pas par une grammaire algébrique d'arbre.

De plus, les programmes logiques sont particulièrement pratiques pour représenter certains langages de programmation ou encore des opérations sur les entiers de Peano. Le langage des listes triées de taille bornée d'entiers naturels peut par exemple être exprimé comme suit.

Exemple 2.1.91 (Langages de l'ensemble des listes triées de taille 3 d'entiers naturels)
Soient $\Sigma = \{c^{\backslash 2}, nil^{\backslash 0}, s^{\backslash 1}, 0^{\backslash 0}\}$ et :

$$Prog = \left\{ \begin{array}{ll} P_1(c(x, y), z) \leftarrow P_2(x, y, z). & P_2(x, c(y, z)) \leftarrow P_3(x, y, z). \\ P_3(x, y, c(z, nil)) \leftarrow P_4(x, y, z). & P_4(x, y, s(z)) \leftarrow P_4(x, y, z). \\ P_4(x, y, z) \leftarrow P_5(x, y, z). & P_5(x, s(y), s(z)) \leftarrow P_5(x, y, z). \\ P_5(x, y, z) \leftarrow P_6(x, y, z). & P_6(s(x), s(y), s(z)) \leftarrow P_6(x, y, z). \\ P_6(0, 0, 0). & \end{array} \right\}$$

Alors $\mathcal{L}(P_1) = \{c(s^n(0), c(s^{n+m}(0), c(s^{n+m+k}(0), nil))) \mid n \geq 0, m \geq 0, k \geq 0\}$.

13. Un arbre binaire parfait est un arbre binaire où toutes les feuilles ont la même distance par rapport à la racine.

Comme dit précédemment, le test du vide est décidable pour les langages synchronisés d'arbres. Un atome de la forme $P(x_1, \ldots, x_{arity(P)})$ avec $x_1, \ldots, x_{arity(P)}$ des variables différentes et libres est appelé un atome libre. Un atome libre et un second atome ayant le même symbole de prédicat sont unifiables. On en déduit qu'un atome libre peut être dérivé par toutes les CS-clauses ayant le même symbole de prédicat dans leur atome de tête. Quand on dérive un atome libre, on obtient une conjonction linéaire d'atomes libres. On peut alors à nouveau dériver ces atomes avec toutes les clauses qui ont le même symbole de prédicat dans leur tête.

Un tuple de termes $\overrightarrow{t}$ est reconnu par un prédicat P s'il est possible de dériver l'atome $P(\overrightarrow{t})$ en la conjonction vide. Si le langage du prédicat n'est pas vide, il est forcement possible de dériver l'atome libre construit avec ce symbole de prédicat en la conjonction vide. Pour réduire le nombre d'atomes présents dans la conjonction, il faut faire une dérivation avec un fait.

On peut alors voir ce problème comme un problème d'accessibilité. On crée un hypergraphe orienté $\mathcal{G}$. Chaque symbole de prédicat constitue un sommet de $\mathcal{G}$. Chaque CS-clause est un hyperarc constitué de plusieurs queues et d'une pointe. La pointe de l'arc est incidente au sommet représentant le symbole de prédicat de l'atome de la tête de la CS-clause. Chaque symbole de prédicat présent dans le corps de la CS-clause est représenté par une queue d'arc incidente au sommet représentant ce symbole de prédicat.

On commence par marquer les sommets représentant les symboles de prédicat présents dans les faits. Ensuite, pour chaque arc, si l'ensemble des sommets incidents aux queues sont marqués, on marque le sommet incident à la pointe. On recommence jusqu'à ce qu'un parcours complet des arcs ne permette pas d'ajouter une nouvelle marque.

Nous proposons l'algorithme 1 pour construire l'ensemble des prédicats issu d'un CS-programme qui génèrent un langage non vide.

L'intersection entre un langage synchronisé et un langage régulier d'arbres est également décidable. Un langage régulier de termes peut être reconnu par un CS-programme n'ayant que des prédicats d'arité 1. Nous montrons à l'aide d'un exemple comment créer un CS-programme qui reconnaît le même langage qu'un automate d'arbre descendant.

Exemple 2.1.92
Reprenons l'automate d'arbres descendant donné dans l'exemple 2.1.60.
Soit $\mathcal{A} = (\{q_{odd}, q_{even}, q_{list}\}, \{c^{\backslash 2}, nil^{\backslash 0}, s^{\backslash 1}, 0^{\backslash 0}\}, \Delta, \{q_{list}\})$ avec

$$\Delta = \left\{ \begin{array}{lll} q_{list}(c(x,y)) \to c(q_{even}(x), q_{list}(y)); & q_{list}(nil) \to nil; & q_{odd}(s(x)) \to s(q_{even}(x)); \\ q_{even}(s(x)) \to s(q_{odd}(x)); & q_{even}(0) \to 0 & \end{array} \right\}$$

Pour représenter cet automate avec le CS-programme Reg, nous allons utiliser un symbole de prédicat pour chaque état de celui-ci. Nous aurons donc $Pred(Reg) = \{Q_{odd}^{\backslash 1}, Q_{even}^{\backslash 1}, Q_{list}^{\backslash 1}\}$. Enfin, nous allons créer une CS-clause par transition :

$$Reg = \left\{ \begin{array}{lll} Q_{list}(c(x,y)) \leftarrow Q_{even}(x), Q_{list}(y). & Q_{list}(nil). & Q_{odd}(s(x)) \leftarrow Q_{even}(x). \\ Q_{even}(s(x)) \leftarrow Q_{odd}(x). & Q_{even}(0). & \end{array} \right\}$$

Nous avons $\mathcal{L}_{Reg}(Q_{list}) = \mathcal{L}(\mathcal{A})$.

L'intersection entre un langage synchronisé d'arbres reconnu par le prédicat P du

```
1  marks ← ∅ // contiendra l'ensemble des prédicats marqués.
2  newMarks ← vrai
3  while newMarks == vrai do
4      newMarks ← faux
5      foreach C ∈ Prog do
6          fullBodyMarked ← vrai
7          foreach A ∈ Body(C) do
8              if Pred(A) ∉ marks then
9                  fullBodyMarked ← faux
10             end
11         end
12         if fullBodyMarked == vrai then
13             marks ← marks ∪{Pred(Head(C))}
14             Prog ← Prog \ {C}
15             newMarks ← vrai
16         end
17     end
18 end
19 Les prédicats présents dans marks génèrent un langage non vide.
20 Les prédicats non présents dans marks génère le langage vide.
```

Algorithme 1 : `emptyness(Prog)`

CS-programme $Prog$ noté $\mathcal{L}_{Prog}(P)$ et le langage régulier reconnu par le prédicat R du CS-programme Reg est reconnue par le programme logique (et non CS) $Prog' = Prog \cup Reg \cup \{I(x) \leftarrow P(x), R(x)\}$ [14].

On peut ensuite utiliser la technique de transformation exacte de programmes logiques en CS-programmes issue des travaux [41] et présentée dans la section 4.1. Dans leurs travaux, les auteurs définissent la notion de *regular join* (Définition 5) et prouvent que leur algorithme de transformation termine toujours avec un *regular join* en entrée. Le programme logique $Prog'$ étant un *regular join*, on obtient alors le CS-programme qui reconnaît l'intersection des langages $\mathcal{L}_{Prog}(P)$ et $\mathcal{L}_{Reg}(R)$.

2.1.5 Langages synchronisés algébriques de tuples d'arbres

Les SCF-programmes [15] [11, 46, 47] sont des programmes logiques avec des modes.

Définition 2.1.93 (Programmes logiques avec modes)
*Un **programme logique avec des modes** est un programme logique tel qu'on associe à chaque prédicat P un tuple de arity(P) modes, qui peuvent être de deux types différents : entrée ou sortie.*

Notation 2.1.94
Afin de différencier les arguments d'entrée et de sortie des prédicats, les arguments de sortie s

14. On a besoin d'avoir $Pred(Prog)$ disjoint de $Pred(Reg)$ et que le prédicat I soit un nouveau prédicat.
15. Synchronized Context Free

seront notés $\widehat{s}$.
Soit P un prédicat, $ar^{in}(P)$ est le nombre d'arguments d'entrée de P et $ar^{out}(P)$ est le nombre d'arguments de sortie, tel que arity$(P) = ar^{in}(P) + ar^{out}(P)$.
Soit $B = A_1, \ldots, A_n$.

- *$In(B)$ est le tuple des termes qui sont en arguments d'entrée des atomes de B ;*
- *$ar^{in}(B) = \sum_{i=1}^{n} ar^{in}(Pred(A_i))$ et*
- *$Var^{in}(B) = \bigcup_{i=1}^{n}(Var(A_i))$.*
- *$Out(B)$, $ar^{out}(B)$ et $Var^{out}(B)$ sont définis de la même manière pour les arguments de sortie.*

Notation 2.1.95
On notera $P^{\backslash o:i}$ le prédicat P tel que $ar^{out}(P) = o$ et $ar^{in}(P) = i$.

Exemple 2.1.96
Soient $P^{\backslash 2:1}$ et $Q^{\backslash 1:2}$ des prédicats et $t_1, \ldots, t_6 \in T(\Sigma, \mathcal{X})$. Soit $B = P(\widehat{t_1}, \widehat{t_2}, t_3), Q(\widehat{t_4}, t_5, t_6)$. Alors $Out(B) = (t_1, t_2, t_4)$ et $In(B) = (t_3, t_5, t_6)$.

Définition 2.1.97 (Boucles de dépendance)
*Soit $B = A_1, \ldots, A_n$. On a $A_j \succ A_k$ [16] s'il existe une variable x dans l'ensemble $Var^{in}(A_j) \cap Var^{out}(A_k)$. En d'autres termes, l'entrée de A_j dépend de la sortie de A_k. On dit que B a une **boucle de dépendance** s'il existe $A_j \in B$ tel que $A_j \succ^+ A_j$ [17].*

Exemple 2.1.98 (Programmes logiques avec modes)
Soient $(\{s^{\backslash 1}\}, \{x, y\})$ une signature, et $Q^{\backslash 1:1}$ et $R^{\backslash 1:1}$ des prédicats. Dans ce cas, $Q(\widehat{x}, s(y)), R(\widehat{y}, s(x))$ a une boucle de dépendance. En effet, on a $Q(\widehat{x}, s(y)) \succ R(\widehat{y}, s(x)) \succ Q(\widehat{x}, s(y))$.

Définition 2.1.99 (SCF-clauses)
Une SCF-clause $H \leftarrow B$ est une clause de Horn avec mode telle que :
- *$In(B) \cdot Out(H)$ [18] est un tuple linéaire composé uniquement de variables,*
- *B ne possède pas de boucle de dépendance.*

Définition 2.1.100 (SCF-programmes)
Un SCF-programme est un programme logique composé de SCF-clauses.

Exemple 2.1.101
Soient $(\Sigma, \mathcal{X})$ une signature avec $\Sigma = \{s^{\backslash 1}\}$ et $\mathcal{X} = \{x, y\}$, et $Prog = \{P(\widehat{x}, y) \leftarrow P(\widehat{s(x)}, y)\}$ avec $P^{\backslash 1:1}$ un prédicat. Le programme logique $Prog$ n'est pas un SCF-programme car $In(H) \cdot Out(B)$ n'est pas un tuple de variables.
Soit $Prog' = \{P'(\widehat{s(x)}, y) \leftarrow P'(\widehat{x}, s(y))\}$ avec $P'^{\backslash 1:1}$ un prédicat. Le programme logique $Prog'$ est un SCF-programme car $In(H) \cdot Out(B) = \{y, x\}$ est un tuple linéaire de variables et il n'y a pas de boucle de dépendance dans le corps de la clause.

16. Avec éventuellement $j = k$.
17. Rappelons que $\succ^+$ est la fermeture transitive de $\succ$
18. Dans ce cas, $\cdot$ est la concaténation de tuple.

Définition 2.1.102 (Langage reconnu par un SCF-programme)
Soit $Prog$ un SCF-programme. Pour un prédicat P avec $ar^{in}(P) = 0$, le langage de tuples d'arbres généré par P est $\mathcal{L}_{Prog}(P) = \{\overrightarrow{t} \in T(\Sigma)^{ar^{out}(P)} | P(\overrightarrow{t}) \in Mod(Prog)\}$. Le langage $\mathcal{L}_{Prog}$ est un langage synchronisé algébrique de tuples d'arbres (S-CF langage).

Remarque 2.1.103
Un SCF-programme dont les prédicats n'ont pas d'arguments d'entrée est un CS-programme.

Exemple 2.1.104
Soit $(\Sigma, \mathcal{X})$ une signature avec $\Sigma = \{c^{\backslash 2}, f^{\backslash 1}, g^{\backslash 1}, h^{\backslash 1}, i^{\backslash 1}, a^{\backslash 0}, b^{\backslash 0}\}$ et $\mathcal{X} = \{x, y, x', y'\}$. Soient $S^{\backslash 1:0}$ et $P^{\backslash 2:2}$ deux prédicats. Soit $Prog$ le SCF-programme suivant :

$$Prog = \left\{ \begin{array}{ll} S(\widehat{c(x,y)}) \leftarrow P(\widehat{x}, \widehat{y}, a, b). & P(\widehat{x}, \widehat{y}, x, y). \\ P(\widehat{f(x)}, \widehat{g(y)}, x', y') \leftarrow P(\widehat{x}, \widehat{y}, h(x'), i(y')). & \end{array} \right\}$$

Le langage généré par S est $\mathcal{L}_{Prog}(S) = \{c(f^n(h^n(a)), g^n(i^n(b))) \mid n \in \mathbb{N}\}$. Ce langage n'est ni un langage synchronisé de tuples [19], ni un langage algébrique.

2.2 Réécriture

La réécriture permet d'obtenir d'autres termes à l'aide de règles de réécriture. Le pouvoir d'expression de la réécriture est similaire à celui des machines de Turing. Les systèmes de réécriture sont souvent utilisés pour modéliser un système et représentent la dynamique de ce dernier [6, 26].

Nous allons commencer par définir les systèmes de réécriture et les différentes propriétés que nous allons utiliser. Nous allons ensuite présenter quelques outils pour décider de la terminaison des systèmes de réécriture. Ces outils seront utilisés dans le chapitre 5. Enfin, nous allons introduire la notion de stratégies de réécriture. Cette notion sera employée dans les chapitres 3 et 5.

Définition 2.2.1 (Règles de réécriture)
*Soit $(\Sigma, \mathcal{X})$ une signature. Une **règle de réécriture** $l \to r$ est un couple $r, l \in T(\Sigma, \mathcal{X})$, où l n'est pas une variable et $Var(r) \subseteq Var(l)$.*

Un terme peut être réécrit s'il existe un de ses sous termes qui match avec le membre gauche d'une règle de réécriture.

Définition 2.2.2 (Réécriture d'un terme)
*Soient $(\Sigma, \mathcal{X})$ une signature et $t \in T(\Sigma, \mathcal{X})$ un terme. Un terme t est réducible par une règle $l \to r$ s'il existe une position $p \in Pos(t)$ et une substitution σ telles que $t_{|p} = \sigma(l)$. On obtient alors le nouveau terme $t' = t[\sigma(r)]_p$, une opération notée $t \to^p_{l \to r} t'$. Une étape de réécriture est appelée une **réduction**.*

Notation 2.2.3
Par souci de simplicité et quand le contexte sera clair, on omettra la règle ou la position de l'opérateur $\to^{position}_{rgle}$. Ainsi, la clôture réflexive transitive de $\to$ par $\to^*$ est dénotée.

19. Il y a une dépendance verticale.

Définition 2.2.4 (Systèmes de réécriture (TRS))
Un système de réécriture, noté TRS [20] *est un ensemble de règles de réécriture deux-à-deux indépendantes* [21]. *Soient* $(\Sigma, \mathcal{X})$ *une signature,* $t, t' \in T(\Sigma, \mathcal{X})$ *des termes et* R *un* TRS. *On note* $t \to_R t'$ *s'il existe une règle* $l \to r \in R$ *telle que* $t \to_{l\to r} t'$. *Si* $t \to_R^* t'$, *on dit que* t' *est un* ***descendant*** *de* t.

Notation 2.2.5
Dans la suite de cette thèse, la lettre R *sera souvent utilisée pour désigner un* TRS.

Définition 2.2.6 (Linéarité des TRS)
Une règle de réécriture $l \to r$ *est dite :*

- ***linéaire droite*** *si pour toute variable* $x \in Var(r)$, x *n'apparaît qu'une fois dans* r.
- ***linéaire gauche*** *si pour toute variable* $x \in Var(l)$, x *n'apparaît qu'une fois dans* l.

Une règle de réécriture est linéaire si elle à la fois est linéaire droite et gauche.

Remarque 2.2.7
Un TRS est linéaire, linéaire droit, linéaire gauche si toutes ses règles sont linéaires, linéaires droites, linéaires gauches, respectivement.

Définition 2.2.8 (Règles collapsing)
Une règle collapsing est une règle qui a comme membre droit une variable.

Exemple 2.2.9 (TRS)
Soit $(\Sigma, \mathcal{X})$ une signature avec $\Sigma = \{c^{\setminus 2}, s^{\setminus 1}, 0^{\setminus 0}\}$ et $\mathcal{X} = \{x, y\}$. Alors :

- $x \to s(x)$ n'est pas une règle de réécriture car son membre gauche est une variable.
- $c(x, x) \to x$ est une règle de réécriture collapsing, non linéaire gauche et linéaire droite.

- $R = \{c(0, x) \to x;\ c(s(x), y) \to c(x, s(y))\}$ est un système de réécriture linéaire qui représente le calcul de l'addition. Soit le terme $t = c(s(s(0), s(0)))$ qui représente $2+1$. On a $t \to_{c(s(x),y)\to c(x,s(y))} c(s(0), s(s(0))) \to_{c(s(x),y)\to c(x,s(y))} c(0, s(s(s(0)))) \to_{c(0,x)\to x} s(s(s(0)))$. Ce qui confirme que $2 + 1 = 3$.

2.2.1 Terminaison des systèmes de réécriture

La terminaison des systèmes de réécriture a été fortement étudiée [22, 30, 37, 48, 58]. En sachant qu'un système de réécriture est terminant, on peut par exemple calculer exactement l'ensemble des descendants à partir d'un ensemble fini de termes. La terminaison des systèmes de réécriture est cependant indécidable dans le cas général. En effet, l'arrêt uniforme des machines de Turing [53], qui est connu comme un problème indécidable est réductible en la terminaison des systèmes de réécriture. Plusieurs outils ont été mis en place pour permettre de décider de la terminaison dans des cas qui n'étaient pas encore couverts. Nous allons aborder deux outils : l'élimination par typage [58] et le multiset path ordering [22, 48].

Définition 2.2.10 (Terminaison des systèmes de réécriture)
Un TRS *est terminant s'il n'existe pas de chemin de réduction infini.*

20. Term Rewriting System.
21. Elles ne partagent pas de variables. Le même nom de variables sera parfois utilisé dans des clauses différentes, mais ces variables seront cependant indépendantes.

Elimination par typage [58] En définissant une fonction de typage sur les termes qui présente les propriétés adaptées, nous allons réduire l'ensemble des termes sur lesquels l'étude de la terminaison est nécessaire. Ces travaux ont été introduits par H. Zantema [58].

Pour ce faire, il faut commencer par définir un profil pour les symboles et les variables.

Définition 2.2.11 (Profil des éléments d'une signature)
*Soit $(\Sigma, \mathcal{X})$ une signature et $\mathcal{S}$ un ensemble de types. Un **profil** est une fonction de $\Sigma \cup \mathcal{X} \to \cup_{n\geq 0}\mathcal{S}^n$ vérifiant que pour tout $f \in \Sigma$, $profil(f) \in \mathcal{S}^{arity(f)+1}$ et pour tout $x \in \mathcal{X}$, $profil(x) \in \mathcal{S}^1$.*

Remarque 2.2.12
Pour un symbole $f \in \Sigma^{\backslash n}$ avec $n \geq 1$, les n premiers éléments du tuple $profil(f)$ correspondent aux types des arguments de f, et le dernier au type de f. De plus, pour une constante $a \in \Sigma^{\backslash 0}$ ou une variable $x \in \mathcal{X}$, $profil(a)$ correspond au type de a ou de x, respectivement. Enfin, par défintion, la fonction $profil$ est unique pour chaque élément de la signature.

Exemple 2.2.13 (Profil des éléments d'une signature)
Soient $(\Sigma, \mathcal{X})$ une signature et $\mathcal{S}$ un ensemble de types. Soient $f \in \Sigma^{\backslash n}$ et $s, s_1, \ldots, s_n \in \mathcal{S}$. On peut définir le profil de f par $profil(f) = (s_1, \ldots, s_n, s)$.

En utilisant la fonction $profil$, on peut maintenant associer un type à un terme.

Définition 2.2.14 (Typage de termes)
*Soient $(\Sigma, \mathcal{X})$ une signature, $\mathcal{S}$ un ensemble de types et $profil$ la fonction de profil de $(\Sigma, \mathcal{X})$. Le **typage de terme** est une fonction $sort : T(\Sigma, \mathcal{X}) \to \mathcal{S}$ définie récursivement par :*
- $\forall\, a \in \Sigma^{\backslash 0},\ sort(a) = profil(a)$,
- $\forall\, x \in \mathcal{X},\ sort(x) = profil(x)$,
- $\forall\, t = f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{X})$ *avec* $f \in \Sigma^{\backslash n}$, *si* $profil(f) = (sort(t_1), \ldots, sort(t_n), s)$ *alors* $sort(t) = s$.

La fonction $sort$ peut ne pas être défini pour certains termes. Ainsi, la fonction de typage permet de distinguer deux ensembles de termes, à savoir ceux qui sont bien typés et les autres.

Définition 2.2.15 (Termes bien typés)
*Soient $(\Sigma, \mathcal{X})$ une signature $t \in T(\Sigma, \mathcal{X})$ un terme et $sort$ une fonction de typage. Le terme t est **bien typé** si et seulement si $sort(t)$ est défini.*

Exemple 2.2.16 (Typage de termes)
Soient $\Sigma = \{+^{\backslash 2}, s^{\backslash 1}, 0^{\backslash 0}, 1^{\backslash 0}\}$, $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$ et $\mathcal{S} = \{s_1, s_2\}$. On définit la fonction $profil$ par :
- $profil(x_i) = s_1$ avec $i \in \{1, 2, 3\}$,
- $profil(x_4) = s_2$,
- $profil(s) = (s_1, s_1)$,
- $profil(+) = (s_1, s_2, s_2)$.

On obtient ainsi pour la fonction $sort$:
- $sort(s(x_1)) = s_1$ vu que $(sort(x_1), s_1) = (s_1, s_1) = profil(s)$,

— $sort(+(x_2, x_4)) = s_2$ vu que $(sort(x_2), sort(x_4), s_2) = (s_1, s_2, s_2) = profil(+)$,
— $sort(+(s(x_1), +(x_2, x_4))) = s_2$ vu que $(sort(s(x_1)), sort(+(x_2, x_4)), s_2) = (s_1, s_2, s_2) = profil(+)$.

Ainsi :

— $+(s(x_1), +(x_2, x_4))$ est bien typé.

— $+(+(s(x_1), x_4), x_2)$ n'est pas bien typé vu que $(sort(s(x_1)), sort(x_4), s_2) = (s_1, s_2, s_2) = profil(s)$ et $(sort(+(s(x_1), x_4)), sort(x_2), s_2) = (s_2, s_1, s_2) \neq profil(+)$.

Le typage de termes peut s'étendre au typage de règles de réécriture.

Définition 2.2.17 (Typage de règles de réécriture)
Soit $sort$ une fonction de typage. Elle est étendue aux règles de réécriture de la façon suivante : $sort(l \to r)$ est défini si et seulement si $sort(l) = sort(r)$ [22] *et, dans ce cas, $sort(l \to r) = sort(l)$. $sort(l \to r) = sort(l)$ si et seulement si l, r sont bien typés et $sort(l) = sort(r)$.*

Définition 2.2.18 (Systèmes de réécriture $\mathcal{S}$-sorted)
Soit R un système de réécriture, $\mathcal{S}$ un ensemble de types et $sort$ une fonction de typage. Le système de réécriture R est $\mathcal{S}$-sorted si pour toute règle $l \to r \in R$, $sort(l \to r)$ est défini.

Dans les travaux de H. Zantema [58], le théorème 21 nous permet à partir d'un système de réécriture $\mathcal{S}$-sorted et sans règles collapsing de prouver la terminaison en observant uniquement les termes bien typés.

Théorème 2.2.19 (Préservation de la terminaison pour les termes bien typés)**.** *Soit R un système de réécriture $\mathcal{S}$-sorted sans règles collapsing. Le système de réécriture R est terminant sur l'ensemble des termes si et seulement si R est terminant sur les termes bien typés.*

Multiset Path Ordering (MPO) [22, 48] À l'aide d'un préordre [23] sur les symboles, une relation sur les termes est définie. Afin de garantir que le système de réécriture est terminant, il suffit d'avoir pour chaque règle de réécriture le membre gauche qui est un successeur du membre droit. Nous avons besoin de donner quelques définitions intermédiaires.

Définition 2.2.20 (Multiensembles)
Un multiensemble est un ensemble dans lequel chaque élément peut apparaître plusieurs fois. On notera $\{\!\{\ldots\}\!\}$ un multiensemble.

Définition 2.2.21 (Extension d'une relation binaire sur les multiensembles)
Soit $\mathcal{R}$ une relation binaire sur un ensemble E.
On définit la relation $\mathcal{R}_0^{mult}$ sur $Mult(E)$ [24] *par :*

$$\begin{aligned} \forall\, t_1, \ldots t_n &\in E,\ \{\!\{t_1, \ldots, t_n\}\!\}\ \mathcal{R}_0^{mult} A \text{ si et seulement si} \\ \exists\, i &\in \{1, \ldots, n\} \text{ et } t'_1, \ldots, t'_k \text{ tels que } t_i\ \mathcal{R}\ t'_1, \ldots, t_i\ \mathcal{R}\ t'_k \text{ et} \\ A &= \{\!\{t_1, \ldots, t_{i-1}, t_{i+1}, \ \ldots, \ t_n\}\!\} \cup \{\!\{t'_1, \ \ldots, t'_k\}\!\} \end{aligned}$$

On définit l'extension multiensemble $\mathcal{R}^{mult}$ de $\mathcal{R}$ par $\mathcal{R}^{mult} = (\mathcal{R}_0^{mult})^+$.

22. Notons que dans ce cas, l et r sont bien typés.
23. Un préordre est une relation réflexive et transitive.
24. Mult(E) est l'ensemble des multiensembles des parties de E.

Définition 2.2.22 (Relation de précédence)
Soit $(\Sigma, \mathcal{X})$ une signature. Soit $\succeq$ un préordre défini sur Σ. On décompose $\succeq$ en deux relations :
— *$\sim$, l'équivalence associée à $\succeq$ ($f \sim g \Leftrightarrow f \succeq g$ et $g \succeq f$ avec $f, g \in \Sigma$),*
— *$\succ$, l'ordre strict associée à $\succeq$ ($f \succ g \Leftrightarrow f \succeq g$ et $f \not\sim g$ avec $f, g \in \Sigma$).*
La relation $\succeq$ est une relation de précédence si $\succ$ est noéthérien [25].

Définition 2.2.23 (Multiset Path Ordering)
Soient $(\Sigma, \mathcal{X})$ une signature et $\succeq$ une relation de précédence. Soient $t, t' \in T(\Sigma, \mathcal{X})$ des termes tels que $t = f(t_1, \ldots, t_{arity(f)})$ et $t' = g(t'_1, \ldots, t'_{arity(g)})$. On a $t \succ_{mpo} t'$ si et seulement si :

— *$\exists i$ tel que $t_i \succeq_{mpo} t'$, ou*
— *$f \succ g$ et $\forall i, t \succ_{mpo} t'_i$ ou*
— *$f \sim g$ et $\{\!\{t_1, \ldots, t_{arity(f)}\}\!\} \succ_{mpo}^{mult} \{\!\{t'_1, \ldots, t'_{arity(g)}\}\!\}$*

Théorème 2.2.24 ([22, 48]). *Soient $(\Sigma, \mathcal{X})$ une signature et $\succeq$ une relation de précédence Σ. Un système de réécriture R est terminant si pour toute règle $l \to r \in R$ on a $l \succ_{mpo} r$.*

2.2.2 Stratégies de réécriture

Un système de réécriture peut être appliqué en fonction d'une stratégie. Un système de réécriture sans stratégie, ou plus exactement avec une stratégie arbitraire, permet de réécrire un terme t à partir du moment où le membre gauche d'une règle de réécriture matche avec un des sous-termes de t. Une stratégie ajoute des conditions supplémentaires à la réécriture d'un terme. Les stratégies de réécriture permettent notamment de reproduire le comportement de certains systèmes. Nous allons aborder quatre stratégies différentes : la stratégie *innermost*, qui sera utilisée dans le chapitre 3 et les stratégies *context-sensitive* (CSTRS) et *prefix-constrained* (pCTRS) qui sont employées dans le chapitre 5 ainsi que les stratégies programmables.

Innermost La stratégie de réécriture *innermost* peut être considérée comme similaire à la stratégie d'évaluation d'appel par valeur de certains langages de programmation comme OCaml, Java ou C. La stratégie de réécriture *innermost* priorise la réécriture en profondeur dans le terme.

Définition 2.2.25 (Stratégie *innermost*)
Soient $(\Sigma, \mathcal{X})$ une signature et R un système de réécriture. Soient $t, t' \in T(\Sigma, \mathcal{X})$ des termes. Pour $p \in Pos(t)$, $t \to_R^p t'$ avec la stratégie innermost si pour toute position $p' \in Pos(t_{|p})$ et $p' \neq \epsilon, (t_{|p})_{|p'}$ n'est pas réductible. On note alors $t \to_{R_{in}}^p t'$.

Exemple 2.2.26
Soient $(\Sigma, \mathcal{X})$ une signature avec $\Sigma = \{f^{\backslash 1}, g^{\backslash 1}\}$ et $\mathcal{X} = \{x\}$. Soit le système de réécriture $R = \{f(x) \to g(x),\ g(x) \to f(x)\}$. Le terme $t = f(g(x))$ peut se réécrire en $g(g(x))$ mais cette étape de réécriture n'est pas *innermost* vu que $t_{|1} = g(x)$ peut également être réécrit. La seule étape de réécriture *innermost* de t est $f(g(x)) \to_{R_{in}} f(f(x))$.

25. Une relation noéthérienne $\mathcal{R}$ définie sur E est une relation qui ne contient pas de suite infinie $t_1, \ldots, t_n, \ldots \in E$ tel que $t_1 \mathcal{R} t_2 \wedge t_2 \mathcal{R} t_3 \ldots \wedge t_{n-1} \mathcal{R} t_n \wedge t_n \mathcal{R} t_{n+1} \wedge \ldots$.

Les stratégies programmables [7, 18] proposent un langage pour définir des stratégies de réécriture. Le terme sur lequel est appliqué une stratégie programmable le **sujet**. L'application d'une stratégie programmable sur un sujet peut soit ne pas terminer, échouer ou retourner un unique résultat. Le langage permettant de définir des stratégies est composé de trois types d'opérateurs : les opérateurs élémentaires, les opérateurs de contrôle de combinaison et les opérateurs de combinaison transversale.

Opérations élémentaires On considère une règle de réécriture comme une opération élémentaire des stratégies programmables.

Définition 2.2.27 (Opérations élémentaires des stratégies programmables)
*Soient $(\Sigma, \mathcal{X})$ une signature et $l \to r$ une règle de réécriture définie sur $(\Sigma, \mathcal{X})$. Il existe trois **opérateurs élémentaires pour les stratégies programmables** :*

- Identity *: opération qui réussit toujours et retourne le sujet inchangé.*
- Fail *: opération qui échoue toujours.*
- *$l \to r$: opération qui tente de réécrire le sujet à la racine. S'il n'existe pas de substitution σ telle que $\sigma(l)$ est égale au sujet alors l'opération échoue, sinon elle réussit et retourne $\sigma(r)$.*

Opérations de contrôlé de combinaison Les opérations de contrôle de combinaison permettent de combiner plusieurs stratégies.

Définition 2.2.28 (Opérations de contrôle de combinaison)
*Soient $(\Sigma, \mathcal{X})$ une signature et $\mathcal{X}_S$ un ensemble de variables de stratégies tel que $\mathcal{X}_s \cap \mathcal{X} = \emptyset$. Soient S_1 et S_2 des stratégies programmables et $X \in \mathcal{X}_s$. Il existe trois **opérateurs de contrôle de combinaison des stratégies programmables** :*

- *$S_1 \leftarrow S_2$: opération de choix. Si la stratégie S_1 échoue elle applique la stratégie S_2.*
- *$S_1; S_2$: opération de séquence. Elle applique en premier la stratégie S_1 sur le sujet, puis la stratégie S_2 sur le résultat issu de l'application de S_1 sur le sujet. Si une des deux stratégies échoue elle échoue.*
- *$\mu X \cdot S_1$: attache la variable de stratégie X à la stratégie S_1.*
- *X : référence la variable de stratégie X.*

Remarque 2.2.29
Les deux derniers opérateurs sont utilisés pour encoder la récursivité. Le fait d'attacher une variable de stratégie à une stratégie permet de réévaluer cette stratégie au moment où la variable sera référencée.

Opérations de combinaison transversale Les opérations de combinaison transversale permettent de modifier la position courante à laquelle les stratégies s'appliquent.

Définition 2.2.30 (Opérations de combinaison transversale)
Soient $(\Sigma, \mathcal{X})$ une signature et S une stratégie programmable. Il existe deux opérateurs de combinaison transversale pour les stratégies programmables :

- *$One(S)$: opération qui réussit si la stratégie S réussit pour **l'un des sous-termes stricts** du sujet. Cette opération échoue toujours si le sujet est une constante.*
- *$All(S)$: opération qui réussit si la stratégie S réussit pour **tous les sous-termes stricts** du sujet. Cette opération réussit toujours si le sujet est une constante.*

Grâce à ces différents opérateurs, il est possible de programmer de nombreuses stratégies différentes, parmi lesquelles la stratégie *innermost*.

Exemple 2.2.31 (Stratégies programmables)
On définit d'abord la stratégie $Try(S)$ qui essaie d'appliquer la stratégie S sur le sujet et applique la stratégie *Identity* en cas d'échec :

$$Try(S) = S \leftarrow\!\!\!\rho\ Identity$$

On peut alors définir la stratégie $Innermost$:

$$Innermost(S) = \mu X \cdot (All(X); Try(S; X))$$

L'appel à l'opérateur de combinaison transversale $All(X)$ permet de descendre aux feuilles du terme. L'appel à la stratégie Try nous permet d'essayer de réécrire sans que cela ne fasse échouer la stratégie. Enfin, la séquence $S; X$ permet de redescendre dans les feuilles à chaque fois que la règle de réécriture représentée ici par S réussit à s'appliquer.

Utilisons notre stratégie $Innermost$ avec la règle de réécriture $S = f(f(y)) \to f(g(y))$ et le sujet $f^3(a)$.

— On commence avec $Innermost(S) = \mu X \cdot (All(X); Try(S; X))$ et $f^3(a)$. L'opérateur $All(X)$ nous fait appliquer $\mu X \cdot (All(X); Try(S; X))$ sur les sous-termes de $f^3(a)$.
  — On a alors $\mu X \cdot (All(X); Try(S; X))$ et $f(f(a))$.
    — Puis $\mu X \cdot (All(X); Try(S; X))$ et $f(a)$.
      — Puis $\mu X \cdot (All(X); Try(S; X))$ et a.
      — $All(X)$ réussit vu que a est une constante.
      — Comme la première stratégie a réussi, on passe à la seconde stratégie de la séquence qui est $Try(S; X)$.
      — On a alors $Try(S; X) = (f(f(y)) \to f(g(y)); X) \leftarrow\!\!\!\rho\ Identity$. La constante a ne pouvant pas être réécrite par notre règle de réécriture, la stratégie $(S; X)$ échoue. Grâce à l'opérateur $\leftarrow\!\!\!\rho$ on applique $Identity$ qui réussit et ne modifie pas notre sujet.
      — La stratégie est finie d'être appliquée à cette position. On remonte au parent.
    — $f(a)$ ne peut par non plus être réécrit donc le sujet reste $f(a)$ et on remonte au niveau supérieur.
  — On arrive à la stratégie $(f(f(y)) \to f(g(y)); X) \leftarrow\!\!\!\rho\ Identity$ avec le sujet $f(f(a))$. La réécriture est effectuée et le sujet devient $f(g(a))$.
  — On applique X
    — On a $\mu X \cdot (All(X); Try(S; X))$ et $g(a)$.
      — ...

On obtient à la fin le terme $f(g(g(a)))$. Le sujet aura eu comme différentes valeurs $\{f^3(a), f(f(g(a))), f(g(g(a)))\}$. Le descendant $f(g(f(a)))$ n'appartient pas à cet ensemble. En effet, c'est un descendant mais pas un descendant *innermost*.

Les deux stratégies que nous venons de voir sont des stratégies qui utilisent uniquement les positions dans les termes. Les deux stratégies suivantes utilisent les positions et les symboles présents dans les termes.

La stratégie de réécriture *context-sensitive* [43] restreint la réécriture à un ensemble donné de positions dans les termes. Afin de définir la stratégie de réécriture *context-sensitive*, il nous faut d'abord introduire plusieurs notations. Afin de décider si une étape de réécriture peut être effectuée, nous allons utiliser une relation qui associe à un symbole un ensemble de positions.

Notation 2.2.32 (Mapping de remplacement d'un alphabet)
Soit $(\Sigma, \mathcal{X})$ *une signature. On définit une fonction de mapping de* Σ, $\mu : \Sigma \to \mathcal{P}(\mathbb{N})$ [26] *telle que* $\forall f \in \Sigma, \mu(f) \subseteq \{1, \ldots, arity(f)\}$.

À l'aide de cette notation, nous pouvons définir un ensemble de positions autorisées.

Notation 2.2.33 (Ensemble de positions de remplacements autorisées)
Soient $(\Sigma, \mathcal{X})$ *une signature et* μ *un mapping de remplacement de* Σ. *Soit* $t \in T(\Sigma, \mathcal{X})$ *un terme. L'**ensemble des positions de remplacements autorisées** par* μ, *noté* $Pos^{\mu}(t)$ *est tel que* $Pos^{\mu}(t) \subseteq Pos(t)$ *et défini inductivement par :*

— $Pos^{\mu}(t) = \epsilon$ *si* $t \in \Sigma^{\backslash 0} \cup \mathcal{X}$,
— $Pos^{\mu}(t) = \{\epsilon\} \cup \{i \cdot p \mid i \in \mu(t(\epsilon)), p \in Pos^{\mu}(t_{|i})\}$ *sinon.*

Maintenant que nous savons définir l'ensemble des positions à l'aide d'une fonction de remplacement, nous donnons la définition de la stratégie de réécriture *context-sensitive*.

Définition 2.2.34 (Stratégie *context-sensitive*)
Soient $(\Sigma, \mathcal{X})$ *une signature,* R *un système de réécriture et* $t, t' \in T(\Sigma, \mathcal{X})$ *des termes. Soient* μ *un mapping de remplacement de* Σ *et* $p \in Pos(t)$. *Le terme* t *se réécrit en* t' *avec la stratégie context-sensitive (définie par* μ*) si* $t \to_R^p t'$ *avec* $p \in Pos^{\mu}(t)$. *On note alors* $t \to_{R_{cs},\mu} t'$ *ou simplement* $t \to_{R_{cs}} t'$ *s'il n'y a pas d'ambiguïté sur* μ.

Exemple 2.2.35
Soient $\Sigma = \{f^{\backslash 2}, g^{\backslash 2}, a^{\backslash 0}, b^{\backslash 0}\}$ un alphabet, $R = \{a \to b\}$ un système de réécriture et $\mu(f) = \{1\}$, $\mu(g) = \{2\}$ une fonction de mapping. Pour le terme $\mathbf{f}(\mathbf{a}, a)$ [27], il n'y a qu'une réécriture possible : $\mathbf{f}(\mathbf{a}, a) \to_{R_{cs}} f(b, a)$. Pour le terme $t = \mathbf{f}(\mathbf{g}(a, \mathbf{a}), a)$, la seule réécriture possible est $t \to_{R_{cs}} f(g(a, b), a)$.

La stratégie de réécriture *prefix-constrained* [35] utilise un langage régulier de mots pour désigner les positions où une réécriture est autorisée. Nous avons besoin d'introduire quelques notations avant de pouvoir définir la stratégie de réécriture *prefix-constrained*.

Notation 2.2.36 (Ensemble des directions)
Soit Σ *un alphabet. L'ensemble des directions, noté* $Dir(\Sigma)$, *est le sous-ensemble de* $\Sigma \times \mathbb{N}$ *défini par :*

$$Dir(\Sigma) = \{\langle f, i\rangle \mid f \in \Sigma,\ 1 \leq i \leq arity(f)\}$$

Une direction est donc un couple constitué d'un symbole et d'une position. La notion de chemin représente en plus d'une position dans un terme, la liste des symboles qui séparent la racine de cette position.

26. $\mathcal{P}(E)$ désigne l'ensemble des parties de l'ensemble E.
27. Les positions de réécriture autorisées dans un terme sont notées en gras.

Définition 2.2.37 (Chemins dans un terme)
Soient Σ un alphabet et $t \in T(\Sigma)$ un terme clos. Le vecteur $path(t,p) \in Dir(\Sigma)^*$ est défini par :

— $path(t,\epsilon) = \epsilon$,
— $path(t, i \cdot p) = \langle t(\epsilon), i\rangle \cdot path(t_{|i}, p)$ *avec* $i \in \{1,\dots,arity(t(\epsilon))\}$ *et* $i \cdot p \in Pos(t)$.

À l'aide d'un automate, la stratégie *prefix-constrained* définit un ensemble de chemins pour lesquels la réécriture est autorisée.

Définition 2.2.38 (Stratégie *prefix-constrained*)
Soit $(\Sigma, \mathcal{X})$ une signature. Un système de réécriture ***prefix-constrained*** *est un ensemble fini R de règles prefix-constrained de la forme $L : l \to r$ avec $L \subseteq Dir(\Sigma)^*$ un langage régulier de chemins et $l, r \in T(\Sigma, \mathcal{X})$ des termes avec $l \notin \mathcal{X}$ et $Var(r) \subseteq Var(l)$.*
Soient $t, t' \in T(\Sigma, \mathcal{X})$ des termes. Le terme t se réécrit en t' avec la stratégie prefix-constrained si $t \to^p_{l\to r} t'$ et $path(t,p) \in L$. On écrira alors $t \to^p_{L:l\to r} t'$, ou simplement $t \to_{R_{pc}} t'$ s'il n'y a pas d'ambiguïté.

Exemple 2.2.39
Soient $\Sigma = \{f^{\backslash 2}, g^{\backslash 2}, a^{\backslash 0}, b^{\backslash 0}\}$ un alphabet, $R = \{L : a \to b\}$ un système de réécriture *prefix-constrained* avec $L = (\langle f,1\rangle \cdot \langle g,2\rangle)^*$. Pour le terme $t = f(g(a,\mathbf{a}),a)$ on a $t_{|1\cdot 2} = a$ et $path(t, 1\cdot 2) = \langle f,1\rangle \cdot \langle g,2\rangle \in L$. On peut donc effectuer une étape de réécriture $t \to_{R_{pc}} f(g(a,b),a)$. On remarque que le terme $f(a,a)$ est irréductible avec notre système de réécriture *prefix-constrained* alors que ce n'est pas le cas avec le système de réécriture *context-sensitive* de l'exemple 2.2.35.

Chapitre 3

Impact de la non-linéarité droite sur la complétion de langages synchronisés de tuples d'arbres

Dans ce chapitre nous étudions l'impact de la non linéarité droite du système de réécriture sur la méthode de complétion présentée dans les travaux [9] et corrigés par [10]. Cette technique de complétion permet à partir d'un langage initial exprimé sous forme de CS-programme et d'un système de réécriture linéaire de calculer l'ensemble des descendants. La restriction de linéarité totale sur le système de réécriture est une restriction forte et contraignante. Nous proposons dans ce chapitre deux résultats permettant l'utilisation de système de réécriture non linéaire droit avec la technique de complétion présentée dans les travaux [9, 10]. Ces travaux ont donné lieu à une publication [15]

Dans un premier temps, nous donnons un état de l'art autour du calcul de sur-approximations des descendants (section 3.1). Par la suite, nous présentons des travaux sur lesquels ces résultats sont basés constitue la première 3.2. Ces travaux proposent une méthode pour calculer l'ensemble des descendants sans stratégie. Dans la section 3.3, nous proposons une modification mineure de l'algorithme initial permettant de calculer l'ensemble de descendants *innermost* avec un système de réécriture non linéaire droit. Dans la section 3.4, nous introduisons la notion d'uncopying de CS-clauses, afin de permettre au prix d'une sur-approximation supplémentaire de calculer l'ensemble de descendants (sans stratégie) avec un système de réécriture non linéaire droit. Enfin, la dernière section 3.5 conclut ce chapitre.

3.1 Sur-approximations d'ensembles de descendants

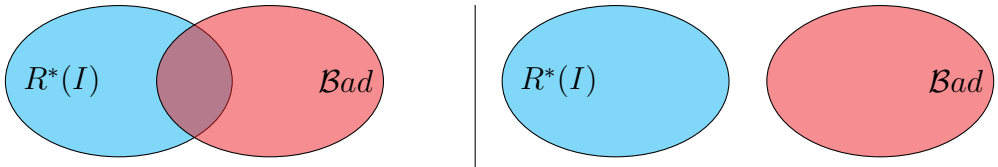
Le calcul de sur-approximations d'ensembles de descendants est utilisé pour l'analyse d'accessibilité. L'analyse d'accessibilité est une composante de la vérification de modèles, qui est l'une des méthodes formelles utilisée pour la vérification. Il est par exemple intéressant de chercher à savoir si un système peut se retrouver dans une configuration représentant une erreur.

Commençons par définir la modèlisation de systèmes que nous utilisons. Le modèle est constitué de deux langages formels de termes et d'un système de réécriture. Les deux

langages sont le langage I représentant l'ensemble des configurations initiales du système et le langage $\mathcal{B}ad$ représentant l'ensemble des configurations indésirables. Le système de réécriture représente la dynamique du système.

Une fois le modèle défini, il faut déterminer s'il y a des configurations indésirables accessibles. Il existe principalement deux approches pour aborder ce problème : l'approche en avant et l'approche en arrière. Comme expliqué dans l'introduction, nous utilisons l'approche en avant dans cette thèse.

Rappelons que cette approche commence par calculer l'ensemble des descendants, c'est-à-dire, l'ensemble des configurations accessibles par le système. Pour obtenir l'ensemble des descendants, le système de réécriture R est appliqué sur l'ensemble des configurations initiales I, et on obtient le langage $R(I)$. Le système de réécriture R est à nouveau appliqué sur l'ensemble calculé précédemment, et ainsi de suite jusqu'à obtenir un ensemble stable par application de R, noté $R^*(I)$. On peut alors vérifier que les configurations de $\mathcal{B}ad$ ne sont pas accessibles, en faisant l'intersection entre $R^*(I)$ et $\mathcal{B}ad$. Si $R^*(I) \cap \mathcal{B}ad = \emptyset$, aucune configuration indésirable n'est accessible, sinon il existe au moins une configuration indésirable accessible.

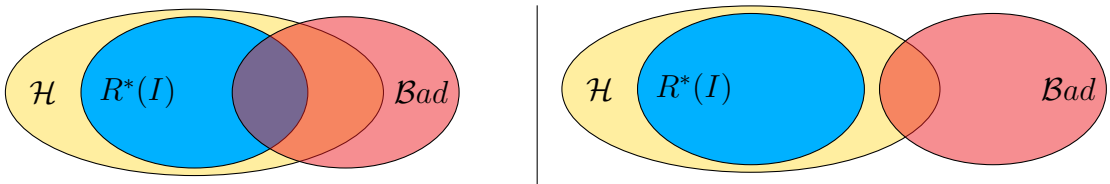


Il existe une configuration indésirable accessible

Aucune configuration indésirable n'est accessible

FIGURE 3.1 – Intersection de $R^*(I)$ et $\mathcal{B}ad$.

Dans le cas général, il est impossible de calculer exactement l'ensemble des descendants. Néanmoins, dans certains cas, cela n'empêche pas totalement l'analyse d'accessibilité, qui peut être abordée à l'aide de procédures incomplètes. En calculant une sur-approximation de l'ensemble des descendants notée $\mathcal{H}$ (telle que $R^*(I) \subseteq \mathcal{H}$), si nous avons $\mathcal{B}ad \cap \mathcal{H} = \emptyset$, cela prouve qu'aucune configuration indésirable n'est accessible. Toute fois, s'il existe un élément dans cette intersection, il n'est pas possible de déterminer si c'est un faux-positif ou non. C'est-à-dire que ce terme peut être dans $\mathcal{H} \setminus R^*(I)$. La finesse de la sur-approximation $\mathcal{H}$ est donc à prendre en considération.



Il existe une configuration indésirable réellement accessible

Aucune configuration indésirable n'est accessible, c'est un faux-positif

FIGURE 3.2 – Introduction des faux-positifs avec l'approximation de $R^*(I)$.

Nous allons dans un premier temps présenter des travaux existants autour du calcul de sur-approximations d'ensembles de descendants utilisant des langages réguliers. Ensuite, nous montrons la limite que présente l'utilisation de sur-approximations régulières. Enfin, nous abordons des travaux utilisant des classes de langages plus expressives, c'est-à-dire, des langages non réguliers.

3.1.1 Sur-approximations régulières d'ensembles de descendants

Le plus souvent, ce sont des sur-approximations régulières qui sont utilisées. Elles ont notamment été utilisées pour vérifier des protocoles cryptographiques [26] ou encore des programmes java [5, 6, 13]. Plusieurs méthodes ont été mises au point pour guider la précision de la sur-approximation.

Dans leurs travaux, Genet et Klay [25, 26] proposent d'utiliser des automates finis d'arbres ascendants non déterministes pour calculer l'ensemble des descendants. À partir d'un premier automate $\mathcal{A}$ qui reconnaît le langage initial ($\mathcal{L}(\mathcal{A}) = I$), ils construisent un nombre fini d'automates $\mathcal{A}_i$ tels que $\mathcal{L}(\mathcal{A}_i) \subseteq \mathcal{L}(\mathcal{A}_{i+1})$ jusqu'à obtenir l'automate $\mathcal{A}_k$, un automate point fixe. Le langage de $\mathcal{A}_k$ représente une sur-approximation de l'ensemble de descendants $R^*(I) \subseteq \mathcal{L}(\mathcal{A}_k)$. Dans ces différents automates, seul l'ensemble des états (Q_i) et l'ensemble des transitions (Δ_i) changent. Pour construire $\mathcal{A}_{i+1}$ à partir de $\mathcal{A}_i$, on cherche un terme $t \in \mathcal{L}(\mathcal{A}_i)$ tel que $t \rightarrow_R t'$ et $t' \notin \mathcal{L}(\mathcal{A}_i)$ et on ajoute des transitions à Δ_{i+1} de manière à ce que $t' \in \mathcal{L}(\mathcal{A}_{i+1})$. Vu que l'ensemble des états et celui des transitions de l'automate $\mathcal{A}_{i+1}$ contiennent ceux de l'automate $\mathcal{A}_i$ ($Q_i \subseteq Q_{i+1}$ et $\Delta_i \subset \Delta_{i+1}$), et qu'il n'y a que les ensembles d'états et de transitions qui varient entre ces automates, nous avons bien $\mathcal{L}(\mathcal{A}_i) \subseteq \mathcal{L}(\mathcal{A}_{i+1})$.

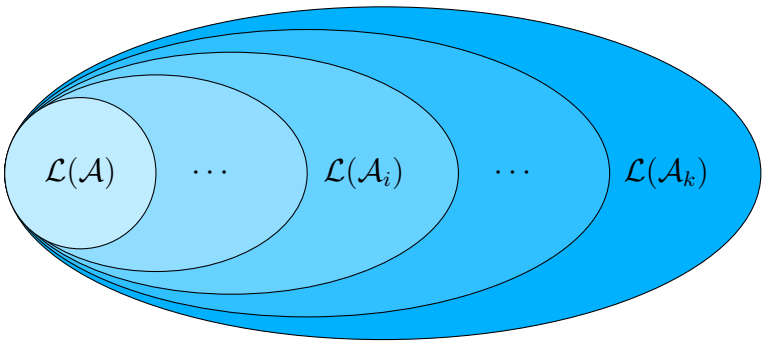


FIGURE 3.3 – Calcul de l'ensemble des descendants [26].

Hormis dans des cas décidables, cette procédure risque de ne pas converger et de produire un nombre infini d'automates. Afin de permettre à ce processus de terminer, les auteurs utilisent une fonction d'approximation notée γ. Cette fonction associe des états à des positions dans les membres droits des règles de réécriture. En associant plusieurs fois le même état à des positions différentes ou à des membres droits de règles de réécriture différents, il est possible de faire terminer le processus.

Afin d'améliorer la qualité de l'approximation, c'est-à-dire d'essayer autant que possible d'éviter les faux positifs, plusieurs méthodes ont été mises au point [12]. La méthode

CEGAR [1] [8] propose de vérifier si les contre-exemples trouvés sont accessibles à partir du langage initial ou s'ils sont dus à la sur-approximation. Si un contre-exemple est un faux [2], il faut alors le retirer de la sur-approximation.

Dans les travaux [8, 27], les auteurs proposent de calculer l'ensemble des descendants à l'aide d'un ensemble d'équations de fusion d'états et de $\mathcal{R}_{/E}$-automates. Un $\mathcal{R}_{/E}$-automate est un automate fini d'arbres descendant non déterministe qui a son ensemble de transitions divisé en plusieurs ensembles disjoints, permettant d'identifier les termes issus des différentes sur-approximations et de distinguer les termes obtenus après une ou plusieurs étapes de réécriture. À partir de ces automates, une fois un faux contre-exemple détecté, il est possible de modifier l'approximation afin de l'éliminer.

Comme vu précédemment, certaines stratégies de réécriture sont particulièrement adaptées à la modélisation du comportement de certains programmes. C'est notamment le cas de la stratégie *innermost*. Les travaux de Genet et Salmon [28] proposent une méthode de complétion permettant le calcul d'une sur-approximation de l'ensemble de descendants par réécriture *innermost*.

3.1.2 Sur-approximations non régulières d'ensembles de descendants

Boichut et Héam [14] montrent que pour un langage régulier I, un TRS R et un langage Bad quelconque tels que $R^*(I) \cap Bad = \emptyset$, il n'existe pas forcement une sur-approximation régulière $\mathcal{H}$ close par R et contenant I satisfaisant $\mathcal{H} \cap Bad = \emptyset$. Ces travaux montrent l'intérêt d'utiliser des classes de langages plus expressifs que les réguliers pour sur-approximer l'ensemble des descendants.

Nous allons donc parler dans cette sous-section de travaux qui présentent des techniques de complétion avec des langages de termes non réguliers [9, 11, 36, 56]. En admettant que $\mathcal{B}ad$ soit régulier, pour pouvoir vérifier $\mathcal{H} \cap \mathcal{B}ad = \emptyset$ il suffit que les classes de langages utilisées pour la sur-approximation soient closes par intersection avec un langage régulier et que le test du vide soit décidable. La classe des langages algébriques d'arbres possède ces propriétés. Dans [36], les auteurs proposent une méthode de calcul d'une sur-approximation des descendants à l'aide d'une grammaire linéaire indexée d'arbres. La classe des langages synchronisés de tuples d'arbres possède également ces propriétés. Boichut et al. [9] présentent une technique de complétion utilisant les CS-programmes. Enfin, les langages synchronisés algébriques de tuples d'arbres ont aussi ces propriétés et ont également fait l'objet d'une méthode de calcul d'une sur-approximation des descendants basée sur les SCF-programmes [11, 56].

Les travaux [11, 36, 56] utilisent des systèmes de réécriture linéaires gauches. Boichut et al. [9] (erratum [10]) utilisent des systèmes de réécriture linéaires.
Ces trois classes de langages contiennent les langages réguliers. Les langages algébriques d'arbres ne sont pas comparables aux langages synchronisés (algébriques ou non) de tuples d'arbres. En effet, les langages algébriques ne permettent pas d'exprimer dans certains cas des dépendances entre différentes branches d'un terme, alors que les langages synchronisés non algébriques ne permettent pas d'exprimer des dépendances verticales dans le terme. Les langages synchronisés algébriques permettent les deux, c'est-à-dire les dépen-

1. Counter Example Guided Abstraction Refinement.
2. Un faux contre-exemple est un terme t tel que $t \in \mathcal{H}$ et $t \notin R^*(I)$.

dances entre plusieurs branches et verticales dans les termes. Les langages synchronisés algébriques d'arbres contiennent les langages synchronisés d'arbres. Cependant, les langages synchronisés algébriques d'arbres contiennent les langages algébriques IO mais pas les langages algébriques.

Les travaux [9, 11, 36, 56] proposent des algorithmes terminants. Dans les travaux [36], lors de la réécriture d'un terme t issu de la grammaire courante donnant un terme t' qui est absent dans cette dernière, la substitution est stockée dans une pile. Afin que la procédure termine, il est nécessaire que l'alphabet de cette pile (contenant les différentes substitutions) soit fini. Pour ce faire, certaines substitutions sont élaguées ce qui revient à les fusionner entre elles. Dans les travaux [9, 11, 56], la technique de calcul de la sur-approximation de l'ensemble de descendants est similaire. Quand on détecte une étape de réécriture non couverte, une nouvelle clause est générée et intégrée au programme logique. Afin de faire terminer le processus, certains prédicats sont fusionnés. La technique de [36] est plus automatique que celle des travaux [9, 11, 56] car elle ne requiert pas d'heuristique.

3.2 La technique existante du calcul des descendants

Dans cette section, nous présentons l'algorithme introduit par Boichut et al. [9] et corrigé par les mêmes auteurs [10]. Cet algorithme calcule une sur-approximation de descendants obtenus par un TRS linéaire, et utilise des langages synchronisés de tuples d'arbres exprimés par des programmes logiques. Cette procédure termine toujours. Nous commençons par décrire la notion de paires critiques qui est au cœur de cette technique. En effet, elle permet de détecter les termes pouvant être réécrits. Afin de garantir la terminaison de l'algorithme, il est important qu'il y ait un nombre de paires critiques fini. La seconde partie de cette section proposera une technique qui permet de garantir cette propriété. Ensuite, afin de pouvoir ajouter ces termes nouvellement générés tout en conservant un CS-programme normalisé, nous présentons la procédure de normalisation de CS-clauses. Enfin, nous donnons la méthode de complétion à proprement parler.

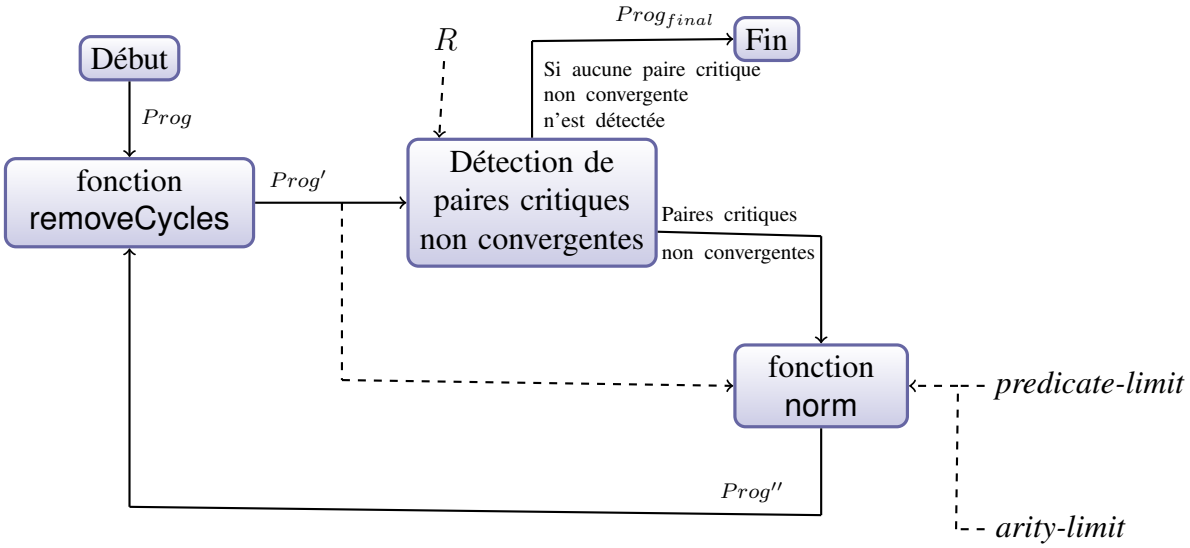


FIGURE 3.4 – Vue d'ensemble de l'algorithme de complétion

3.2.1 Paires critiques

Comme dit précédemment, la notion de paires critiques est centrale. Pour un CS-programme donné $Prog$, un symbole de prédicat P et une règle de réécriture $l \to r$, une paire critique permet de détecter une réécriture par la règle $l \to r$ du terme t issu d'un tuple de $\mathcal{L}_{Prog}(P)$. En ajoutant cette paire critique au langage, nous couvrirons ainsi cette étape de réécriture.

Définition 3.2.1 (Paire critique [9])
Soient $Prog$ un CS-programme et $l \to r$ une règle de réécriture linéaire gauche. Soit $x_1, \ldots, x_n$ un ensemble de variables distinctes tel que $\{x_1, \ldots, x_n\} \cap Var(l) = \emptyset$. S'il existe P et k tels que $P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \leadsto_\theta^+ G$ où la résolution est appliquée uniquement sur des atomes non plats, G est plat, et la clause $P(t_1, \ldots, t_n) \leftarrow B$ utilisée durant la première étape de dérivation satisfait t_k n'est pas une variable[3]*, alors la clause :*

$$\theta(P(x_1, \ldots, x_{k-1}, r, x_{k+1}, \ldots, x_n)) \leftarrow G$$

*est appelée **paire critique**.*

Remarque 3.2.2
La règle de réécriture étant linéaire gauche et les clauses utilisées pour la dérivation $P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \leadsto_\theta^+ G$ étant toutes des CS-clauses, la séquence d'atome G est forcément linéaire. Cette propriété garantit que la paire critique $\theta(P(x_1, \ldots, x_{k-1}, r, x_{k+1}, \ldots, x_n)) \leftarrow G$ est une CS-clause.

Afin de savoir si la paire critique obtenue représente une étape de réécriture déjà couverte par le CS-programme, nous définissons la propriété de convergence sur les paires critiques.

Définition 3.2.3 (Convergence d'une paire critique [9])
*Une paire critique $H \leftarrow B$ est dite **convergente** si $H \to_{Prog}^* B$ (Définition 2.1.84).*

Exemple 3.2.4 (Illustration des définitions 3.2.1 et 3.2.3)
Soit Σ l'alphabet $\{c^{\backslash 1}, h^{\backslash 1}, a^{\backslash 0}, b^{\backslash 0}\}$. Soit $Prog$ le CS-programme normalisé, non copiant et défini par :

$$Prog = \left\{ \begin{array}{ll} P(c(x), y) \leftarrow Q(x, y). & Q(h(x), y) \leftarrow Q(x, y). \\ Q(c(x), y) \leftarrow Q(x, y). & Q(a, b). \end{array} \right\}$$

Soit la règle de réécriture linéaire gauche $c(c(x')) \to h(h(x'))$. Deux paires critiques sont générées :

— Le terme $c(x)$ issu de la clause $P(c(x), y) \leftarrow Q(x, y)$ peut être unifié par le membre gauche de la règle $c(c(x')) \to h(h(x'))$. On obtient alors l'atome $P(c(c(x')), y)$ qui peut se dériver en

$$P(c(c(x')), y) \leadsto_{[P(c(x),y) \leftarrow Q(x,y)]} Q(c(x'), y) \leadsto_{[Q(c(x),y) \leftarrow Q(x,y)]} Q(x', y)$$

En remplaçant le membre gauche de la règle de réécriture par le membre droit, on obtient alors la paire critique $P(h(h(x')), y) \leftarrow Q(x', y)$ qui n'est pas convergente.
En effet, aucune clause de $Prog$ ne peut réécrire $P(h(h(x')), y)$.

3. En d'autres termes, l'application de l sur la tête de clause $P(t_1, \ldots, t_n)$ ne s'effectue pas sur une variable.

— Le terme $c(x)$ issu de la clause $Q(c(x), y) \leftarrow Q(x, y)$ peut être unifié par le membre gauche de la règle $c(c(x')) \rightarrow h(h(x'))$. On obtient alors l'atome $Q(c(c(x')), y)$ qui peut se dériver en

$$Q(c(c(x')), y) \rightsquigarrow_{[Q(c(x),y) \leftarrow Q(x,y)]} Q(c(x'), y) \rightsquigarrow_{[Q(c(x),y) \leftarrow Q(x,y)]} Q(x', y)$$

En remplaçant le membre gauche de la règle de réécriture par le membre droit, on obtient alors la paire critique $Q(h(h(x')), y) \leftarrow Q(x', y)$ qui est convergente.
L'atome $Q(h(h(x')), y)$ peut se réécrire en $Q(h(x'), y)$ avec la clause $Q(h(x), y) \leftarrow Q(x, y)$ de $Prog$, et $Q(h(x'), y)$ peut se réécrire en $Q(x', y)$ avec la même clause. Ce dernier atome correspond au corps de la paire critique et donc celle-ci est convergente.

3.2.2 Garantir un nombre fini de paires critiques

L'exemple suivant illustre une situation où, pour un CS-programme donné, le nombre de paires critiques est infini.

Exemple 3.2.5
Soient Σ l'alphabet $\{f^{\setminus 2}, c^{\setminus 1}, d^{\setminus 1}, s^{\setminus 1}, a^{\setminus 0}\}$ et $f(c(x), y) \rightarrow d(y)$ une règle de réécriture. Soit $Prog$ le CS-programme défini par :

$$Prog = \left\{ \begin{array}{ll} P_0(f(x, y)) \leftarrow P_1(x, y) & P_1(x, s(y)) \leftarrow P_1(x, y) \\ P_1(c(x), y) \leftarrow P_2(x, y) & P_2(a, a). \end{array} \right\}$$

Alors $P_0(f(c(x), y)) \rightarrow P_1(c(x), y) \rightsquigarrow_{y/s(y)} P_1(c(x), y) \rightsquigarrow_{y/s(y)} \cdots P_1(c(x), y) \rightarrow P_2(x, y)$. La résolution est appliquée uniquement sur des atomes non plats, et le dernier atome obtenu par ces dérivations est plat. La composition des substitutions de cette dérivation donne $y/s^n(y)$ pour $n \in \mathbb{N}$. On obtient ainsi un nombre infini de dérivations qui génèrent un nombre infini de paires critiques de la forme $P_0(d(s^n(y))) \leftarrow P_2(x, y)$.

Ceci est gênant puisque le processus de complétion présenté ici doit calculer toutes les paires critiques. Pour parer à cela, un processus donnant des conditions suffisantes pour garantir que le CS-programme a un nombre fini de paires critiques va être présenté ici.

Définition 3.2.6 (Empty-recursive [9])
*Un CS-programme $Prog$ est **empty-recursive** s'il existe un prédicat P et des variables distinctes $x_1, \ldots, x_n$ tels que $P(x_1, \ldots, x_n) \rightsquigarrow_\sigma^+ A_1, \ldots, P(x'_1, \ldots, x'_n), \ldots, A_k$ où $x'_1, \ldots, x'_n$ sont des variables, $\sigma(x_j)$ n'est pas une variable et $x'_j \in Var(\sigma(x_j))$.*

Exemple 3.2.7
Soit $Prog$ un CS-programme défini par :

$$Prog = \quad \{ \quad P(x', s(y')) \leftarrow P(x', y'). \quad P(a, b). \quad \}$$

À partir de $P(x, y)$, on obtient la dérivation suivante : $P(x, y) \rightsquigarrow_{[x/x',\, y/s(y')]} P(x', y')$. Par conséquent, $Prog$ est *empty-recursive* vu que $\sigma = [x/x',\, y/s(y')]$ et y' est une variable de $\sigma(y) = s(y')$.

Le lemme suivant montre que le fait qu'un CS-programme ne soit pas *empty-recursive* est suffisant pour assurer un nombre fini de paires critiques.

Lemme 3.2.8 ([9])
Soit $Prog$ un CS*-programme normalisé.*
Si $Prog$ n'est pas empty-recursive, alors le nombre de paires critiques est fini.

Remarque 3.2.9
Notons que le CS-programme de l'exemple 3.2.5 est normalisé et a un nombre infini de paires critiques. Cela est dû au fait qu'il est *empty-recursive* car $P_1(x, y) \rightsquigarrow_{[x/x', y/s(y')]} P_1(x', y')$.

Décider si un CS-programme est *empty-recursive* semble être un problème difficile (indécidable ?). Voici une condition syntaxique suffisante pour assurer que le CS-programme n'est pas *empty-recursive*.

Définition 3.2.10 (Clause *pseudo-empty*)
Soit une clause $P(t_1, \ldots, t_n) \leftarrow B$. S'il existe $A = Q(x_1, \ldots, x_k) \in B$ et i, j tels que :
- *t_i est une variable et $t_i \in Var(A)$*
- *t_j n'est pas une variable*
- *$\exists x \in Var(t_j)$, $x \neq t_i$ et $x \in Var(A)$*

*Alors la clause $P(t_1, \ldots, t_n) \leftarrow B$ est **pseudo-empty sur** Q.*
*La clause $H \leftarrow B$ est **pseudo-empty** s'il existe Q tel que $H \leftarrow B$ est pseudo-empty sur Q.*

Avoir une clause *pseudo-empty* revient à dire que lors d'une étape de résolution réalisée à partir d'un atome plat $P(y_1, \ldots, y_n)$, il existe deux variables $y_i, y_j \in \{y_1, \ldots, y_n\}$ telles que y_i n'est pas instanciée, et y_j est instanciée par un terme contenant une variable x différente de y_i, et $x, y_i \in Var(A)$.

Définition 3.2.11 (Clause empty)
Une CS*-clause $P(t_1, \ldots, t_n) \leftarrow A_1, \ldots, Q(x_1, \ldots, x_k), \ldots, A_m$ est **empty sur** Q si pour tout $x_i \in Var(Q(x_1, \ldots, x_k))$, il existe j tel que $t_j = x_i$ ou $x_i \notin Var(P(t_1, \ldots, t_n)))$.*

Exemple 3.2.12 (Illustration de la définition des propriétés *pseudo-empty* et *empty* d'une clause)
La CS-clause $P(x, f(x), z) \leftarrow Q(x, z)$ est *pseudo-empty* (grâce au second et au troisième argument de P) et est empty sur Q (grâce au premier et au troisième argument de P).

Définition 3.2.13 (Relations $\trianglerighteq_{Prog}$ et $>_{Prog}$)
Soit $Prog$ un CS*-programme. À partir des Définitions 3.2.10 et 3.2.11, deux relations sont définies sur les symboles de prédicats.*
- *$P_1 \trianglerighteq_{Prog} P_2$ s'il existe dans $Prog$ une clause empty sur P_2 de la forme $P_1(\ldots) \leftarrow A_1, \ldots, P_2(\ldots), \ldots, A_n$.*

- *$P_1 >_{Prog} P_2$ s'il existe dans $Prog$ des prédicats P_1', P_2' tels que $P_1 \trianglerighteq^*_{Prog} P_1'$ et $P_2' \trianglerighteq^*_{Prog} P_2$, et une clause pseudo-empty sur P_2' de la forme $P_1'(\ldots) \leftarrow A_1, \ldots, P_2'(\ldots), \ldots, A_n$.*

*La relation $>_{Prog}$ est **cyclique** s'il existe un prédicat P tel que $P >^+_{Prog} P$.*

Notation 3.2.14
La clôture réflexive-transitive de $\trianglerighteq_{Prog}$ est notée $\trianglerighteq^*_{Prog}$.
De manière similaire, la fermeture transitive de $>_{Prog}$ est notée $>^+_{Prog}$.

Exemple 3.2.15
Soient $\Sigma = \{f^{\setminus 1}, h^{\setminus 1}, a^{\setminus 0}\}$ et $Prog$ le CS-programme :

$$Prog = \left\{ \begin{array}{ll} P(x, h(y), f(z)) \leftarrow Q(x, z), R(y). & R(a). \\ Q(x, g(y, z)) \leftarrow P(x, y, z). & Q(a, a). \end{array} \right\}$$

On a $P >_{Prog} Q$ et $Q >_{Prog} P$. Ainsi, $>_{Prog}$ est cyclique.

L'absence de cycles est le point clé de cette technique puisqu'elle permet d'avoir un nombre fini de paires critiques.

Lemme 3.2.16
Soit $Prog$ un CS-programme. Si $>_{Prog}$ n'est pas cyclique, alors $Prog$ n'est pas empty-recursive.

Remarque 3.2.17
Une conséquence du Lemme 3.2.16 est que si un CS-programme est tel que $>_{Prog}$ n'est pas cyclique, alors le nombre de paires critiques est fini.

Si $>_{Prog}$ n'est pas cyclique dans le CS-programme $Prog$, alors il y a un nombre fini de paires critiques. Sinon, nous devons transformer $Prog$ en un autre CS-programme $Prog'$ tel que $>_{Prog'}$ n'est pas cyclique et $Mod(Prog) \subseteq Mod(Prog')$.

Cette transformation est basée sur les observations suivantes. Si $>_{Prog}$ est cyclique, alors il y a au moins une clause *pseudo-empty* sur un prédicat donné qui participe au cycle. On remarque que cette propriété est vérifiée dans l'exemple 3.2.15 où $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ est une clause *pseudo-empty* sur Q et est impliquée dans ce cycle. Pour enlever les cycles, on transforme les clauses *pseudo-empty* en clauses qui ne le sont pas. Cela revient à désynchroniser des variables. Ce processus est décrit dans la définition 3.2.20. Les définitions 3.2.18 et 3.2.19 sont nécessaires pour définir 3.2.20.

Ci-dessous, nous définissons simplify qui supprime les clauses non productives et retire du corps d'une clause les atomes qui contiennent uniquement des variables libres.

Définition 3.2.18 (La fonction simplify)
Soit $H \leftarrow A_1, \ldots, A_n$ une CS-clause. Pour tout i, on note $A_i = P_i(\ldots)$.
- *S'il existe P_i tel que $\mathcal{L}(P_i) = \emptyset$, alors* simplify$(H \leftarrow A_1, \ldots, A_n) = \emptyset$.
- *Sinon* simplify$(H \leftarrow A_1, \ldots, A_n) = \{H \leftarrow B\}$ *avec :*
  - $B \subseteq \{A_i \mid 0 \leq i \leq n\}$ *et*
  - $\forall i \in \{1, \ldots, n\}, Var(A_i) \cap Var(H) \neq \emptyset \Leftrightarrow A_i \in B$

Ci-dessous, nous introduisons unSync qui crée à partir d'une clause *pseudo-empty*, une nouvelle clause non *pseudo-empty* et qui reconnaît a minima le même langage que la clause d'origine. Ce processus génère deux variantes (B_0 et B_1) du corps de la clause d'origine. La variante B_0 est le corps de la clause d'origine dans lequel toutes les variables qui n'occurent jamais à la racine des paramètres de la tête de clause sont remplacées par des variables libres. La variante B_1 est le corps de la clause d'origine dans lequel toutes les variables qui occurent au moins une fois à la racine d'un des paramètres de la tête de clause sont remplacées par des variables libres. La clause créée a la même tête de clause que celle d'origine et a comme corps la variante B_0 et B_1 du corps de la clause d'origine. Enfin, on applique simplify à cette clause nouvellement créée.

Définition 3.2.19 (La fonction unSync)
Soit $P(t_1, \ldots, t_n) \leftarrow B$ une CS-clause pseudo-empty. On définit
unSync$(P(t_1, \ldots, t_n) \leftarrow B)$ = simplify$(P(t_1, \ldots, t_n) \leftarrow B_0, B_1)$ *avec :*

— $B_0 = \sigma_0(B)$ *et* σ_0 *la substitution construite de la manière suivante :*

$$\sigma_0(x) = \begin{cases} x & \text{s'il existe } i \text{ tel que } t_i = x \\ \text{une variable libre} & \text{sinon} \end{cases}$$

— $B_1 = \sigma_1(B)$ *et* σ_1 *la substitution construite de la manière suivante :*

$$\sigma_1(x) = \begin{cases} x & \text{s'il existe } i \text{ tel que } t_i \text{ n'est pas une variable} \\ & \text{et } x \in Var(t_i) \text{ et } \neg(\exists j,\ t_j = x) \\ \text{une variable libre} & \text{sinon} \end{cases}$$

Maintenant que nous sommes capables de transformer une clause *pseudo-empty* en une clause qui ne l'est plus au prix d'une sur-approximation, il nous reste à appliquer unSync sur l'ensemble des clauses *pseudo-empty* responsables d'au moins un cycle. Nous allons utiliser removeCycles défini ci-dessous pour réaliser cela.

Définition 3.2.20 (La fonction removeCycles)
Soit $Prog$ *un* CS*-programme. On définit*

$$\mathsf{removeCycles}(Prog) = \begin{cases} Prog & \text{Si } >_{Prog} \text{ n'est pas cyclique} \\ \mathsf{removeCycles}(\{\mathsf{unSync}(H \leftarrow B)\} \cup Prog') & \text{sinon} \end{cases}$$

où $H \leftarrow B$ *est une clause pseudo-empty responsable d'un cycle et* $Prog' = Prog \setminus \{H \leftarrow B\}$.

Exemple 3.2.21
Soit $Prog$ le CS-programme de l'exemple 3.2.15 :

$$Prog = \left\{ \begin{array}{ll} P(x, h(y), f(z)) \leftarrow Q(x, z), R(y). & R(a). \\ Q(x, g(y, z)) \leftarrow P(x, y, z). & Q(a, a). \end{array} \right\}$$

Comme $Prog$ est cyclique, nous appliquons removeCycles sur $Prog$. La CS-clause *pseudo-empty* $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ est responsable du cycle. Par conséquent, unSync est appliqué sur celle-ci. Vue la définition 3.2.19, on obtient σ_0 et σ_1 où $\sigma_0 = [x/x, y/x_1, z/x_2]$ et $\sigma_1 = [x/x_3, y/y, z/z]$. Ce qui nous donne la CS-clause :

$$P(x, h(y), f(z)) \leftarrow Q(x, x_2), R(x_1), Q(x_3, z), R(y).$$

Comme indiqué dans la définition 3.2.19, simplify doit être appliqué sur cette CS-clause nouvellement générée. D'après la définition 3.2.18, l'atome $R(x_1)$ doit être retiré du corps de la clause car il contient uniquement des variables libres. Il ne nous reste alors plus qu'à appliquer la définition 3.2.20 en remplaçant la clause $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ dans $Prog$ par la clause $P(x, h(y), f(z)) \leftarrow Q(x, x_2), Q(x_3, z), R(y)$. Remarquons qu'il n'y a plus de cycle.

Le lemme 3.2.22 garantit que ces transformations préservent au moins et peuvent étendre le plus petit modèle de Herbrand initial.

Lemme 3.2.22 ([9])
Soit $Prog$ *un* CS*-programme et* $Prog' = \mathsf{removeCycles}(Prog)$.
Alors $>_{Prog'}$ *n'est pas cyclique et* $Mod(Prog) \subseteq Mod(Prog')$. *De plus, si* $Prog$ *est normalisé alors* $Prog'$ *l'est également.*

Maintenant, pour un CS-programme donné $Prog$, on a :
— soit $>_{Prog}$ n'est pas cyclique et donc le nombre de paires critiques est fini ;
— soit on transforme $Prog$ en un autre CS-programme $Prog'$ tel que $>_{Prog'}$ n'est pas cyclique et $Mod(Prog) \subseteq Mod(Prog')$. Puisque $Prog'$ n'est pas cyclique, nous avons un nombre fini de paires critiques.

3.2.3 Normalisation de CS-clauses

Vu que la réécriture n'intervient qu'à la racine des termes dans les CS-clauses, nous avons besoin que le CS-programme soit normalisé. Cependant, les paires critiques générées ne sont pas normalisées. Afin de pouvoir les ajouter tout en gardant le CS-programme normalisé, nous utilisons la fonction de normalisation norm décrite dans la définition 3.2.28. Avant de présenter celle-ci, nous avons besoin d'introduire des notations ainsi que la fonction by-pass.

Notation 3.2.23 (Découpage de termes [9])
Soit un tuple d'arbres $\overrightarrow{t} = (t_1, \ldots, t_n)$.
Nous définissons :

— $\overrightarrow{t}^{cut} = (t_1^{cut}, \ldots, t_n^{cut})$, *où* $t_i^{cut} = \begin{cases} x'_{i,1} & \text{si } t_i \text{ est une variable} \\ t_i & \text{si } t_i \text{ est une constante} \\ t_i(\epsilon)(x'_{i,1}, \ldots, x'_{i,ar(t_i(\epsilon))}) \text{ [4]} & \text{sinon} \end{cases}$
Les variables $x'_{i,k}$ *sont de nouvelles variables qui n'apparaissent pas dans* $\overrightarrow{t}$.

— $\overrightarrow{t}^{rest} = (t_1^{rest} \cdots t_n^{rest})$ [5],
avec pour chaque i, t_i^{rest} *le tuple d'arbres tel que*
$t_i^{rest} = \begin{cases} (t_i) & \text{si } t_i \text{ est une variable} \\ \text{le tuple vide} & \text{si } t_i \text{ est une constante} \\ ((t_i)_{|1}, \ldots, (t_i)_{|ar(t_i(\epsilon))}) \text{ [6]} & \text{sinon} \end{cases}$

— $\overrightarrow{Var}(\overrightarrow{t})$ *dénote le tuple contenant les variables apparaissant dans le tuple* $\overrightarrow{p}$.

Exemple 3.2.24 (Illustration de la notation 3.2.23)
Soit $\overrightarrow{t}$ un tuple d'arbre tel que $\overrightarrow{t} = (x_1, x_2, g(x_3, h(x_4)), h(x_5), b)$ où les x_i sont des variables. On a alors,

— $\overrightarrow{t}^{cut} = (y_1, y_2, g(y_3, y_4), h(y_5), b)$ avec les variables y_i qui sont de nouvelles variables ;
— $\overrightarrow{Var}(\overrightarrow{t}^{cut}) = (y_1, y_2, y_3, y_4, y_5)$;
— $\overrightarrow{t}^{rest} = (x_1, x_2, x_3, h(x_4), x_5)$.

On remarquera que $\overrightarrow{t}^{cut}$ est normalisé et linéaire. De plus, $\overrightarrow{Var}(\overrightarrow{t}^{cut})$ et $\overrightarrow{t}^{rest}$ ont la même arité.

La fonction by-pass diminue le nombre de clauses créées lors de la normalisation afin de rendre cet algorithme plus performant au prix d'une sur-approximation.

Définition 3.2.25 (La fonction by-pass)
Soient $Prog$ *un* CS*-programme normalisé et* $H \leftarrow B$ *une* CS*-clause. La fonction* by-pass *est définie par :*

4. Cette expression donne le terme formé par le même symbole que t_i à la racine, et qui a pour chaque sous-terme une nouvelle variable.
5. Le symbole $\cdots$ représente ici la concaténation de tuples.
6. Cette expression représente le vecteur des sous-termes de t_i à la profondeur 1.

```
1 by-pass(H ← B, Prog)
2 if Il n'y a pas de clause (H' ← A) ∈ Prog synchronisante et non empty pouvant
    réécrire H then
3 |   return H ← B
4 else
5 |   // Dans ce cas, vu que la réécriture est possible,
6 |   // il existe une substitution θ telle que H = θ(H')
7 |   return by-pass(θ(A) ← B, Prog)
8 end
```

Lemme 3.2.26
La fonction by-pass *termine toujours.*

Argument de preuve. À chaque appel récursif, la taille des termes en paramètre de la tête de clause H diminue. □

Exemple 3.2.27
Soit $Prog = \{P(g(x)) \leftarrow P(x)\}$ un CS-programme. On souhaite exécuter la fonction by-pass sur la clause $P(g(g(f(x')))) \leftarrow Q(x')$. Le premier appel à la fonction by-pass transforme la clause en paramètre vu qu'elle peut être réécrite par $P(g(x)) \leftarrow P(x)$ qui est une clause synchronisante et non vide. On a alors $P(g(g(f(x')))) = \theta(P(g(x)))$ avec $\theta = (x/g(f(x')))$ et by-pass est appelée à nouveau avec la clause $P(g(f(x'))) \leftarrow Q(x')$ en paramètre. Lors du second appel à by-pass, la clause en paramètre est à nouveau transformée. On a alors $P(g(f(x'))) = \theta(P(g(x)))$ avec $\theta = (x/f(x'))$ et by-pass est appelée récursivement avec la clause $P(f(x')) \leftarrow Q(x')$. Lors du troisième appel, il n'est plus possible de réécrire la clause en paramètre, on retourne alors la clause inchangée ($P(f(x')) \leftarrow Q(x')$).

On a donc la clause $P(g(g(f(x')))) \leftarrow Q(x')$ qui a été by-passée par Prog en la clause $P(f(x')) \leftarrow Q(x')$.

Ajouter une paire critique après l'avoir normalisée dans un CS-programme peut générer de nouvelles paires critiques, ce qui peut entraîner la non terminaison du processus de complétion. Pour garantir la terminaison, nous fixons deux bornes :

- *predicate-limit* : cette borne permet de limiter l'explosion du nombre de symboles de prédicats de même arité. Quand le nombre de symboles de prédicats atteint *predicate-limit* pour une arité donnée, au lieu de créer un nouveau symbole de prédicats, on en réutilise un déjà existant. Le choix du prédicat est délégué à la fonction predicate-choice.
- *arity-limit* : cette borne permet de limiter l'arité des symboles de prédicats générés. Si la création d'un symbole de prédicats d'une arité supérieure à *arity-limit* est nécessaire, l'atome est découpé en plusieurs atomes dont l'arité ne dépasse pas *arity-limit*. Le choix du découpage de l'atome est délégué à la fonction atom-cut-choice.

Pour les fonctions predicate-choice et atom-cut-choice, des fonctions naïves qui font un choix aléatoire parmi l'ensemble des choix possibles permettent facilement de rendre l'algorithme totalement automatique. Cependant cela se ferait au détriment de la qualité de la sur-approximation, les choix faits par les fonctions predicate-choice et atom-cut-choice étant déterminants pour la précision de la sur-approximation. Afin de maîtriser

au mieux ce mécanisme, nous choisissons manuellement les prédicats que nous souhaitons fusionner ou les découpages d'atomes.

Définition 3.2.28 (La fonction norm [9])
Soient $Prog$ un CS*-programme normalisé et $H \leftarrow B$ une* CS*-clause.*
Soit $Pred$ l'ensemble des symboles de prédicats de $Prog$. Pour chaque entier positif i, on définit $Pred_i = \{P \in Pred \mid \textit{arity}(P) = i\}$.
Soient arity-limit et predicate-limit des entiers positifs tels que :

$$\begin{aligned} \forall P &\in Pred,\ \textit{arity}(P) \leq \textit{arity-limit} \\ \forall i &\in \{1, \ldots, \textit{arity-limit}\},\ card(Pred_i) \leq \textit{predicate-limit} \end{aligned}$$

La fonction norm *est définie par :*

```
1  norm(H ← B, Prog)
2  if H ← B est normalisée then
3  |  return Prog ∪ {H ← B}
4  else
5  |  H ← B = by-pass(H ← B, Prog)
6  |  On note H = P(t⃗)
7  |  if arity(Var⃗(t⃗^cut)) ≤ arity-limit then
8  |  |  if card(Pred_{ar(Var⃗(t⃗^cut))}) ≤ predicate-limit then
9  |  |  |  P' est un nouveau prédicat.
10 |  |  else
11 |  |  |  P' = predicate-choice(H ← B, Prog, ∅, ∅)
12 |  |  end
13 |  |  return norm(P'(t⃗^rest) ← B, Prog ∪ {P(t⃗^cut) ← P'(Var⃗(t⃗^cut))})
14 |  else
15 |  |  ⟨⟨t⃗_1^cut, t⃗_1^rest⟩, ..., ⟨t⃗_k^cut, t⃗_k^rest⟩⟩ = atom-cut-choice(H ← B, Prog,
   |  |   predicate-limit)
16 |  |  P⃗' = ⟨ ⟩
17 |  |  for j range 1 to k do
18 |  |  |  P_tmp = predicate-choice(H ← B, Prog, t⃗_j^cut, t⃗_j^rest)
19 |  |  |  P⃗' = concat(P⃗', P_tmp)
20 |  |  end
21 |  |  On note P⃗' = P'_1 ... P'_k
22 |  |  Soit c' la clause P(t⃗^cut) ← P'_1(Var⃗(t⃗_1^cut)), ..., P'_k(Var⃗(t⃗_k^cut))
23 |  |  Res = Prog ∪ {c'}
24 |  |  for j range 1 to k do
25 |  |  |  Res = norm((P'_j(t⃗_j^rest) ← B), Res)
26 |  |  end
27 |  |  return Res
28 |  end
29 end
```

3.2.4 Complétion

Dans les sous-sections 3.2.1 et 3.2.3, nous avons décrit comment détecter les paires critiques et comment les convertir en CS-clauses normalisées. De plus, dans la sous-section 3.2.2, nous avons présenté une technique permettant de transformer un CS-programme donné en un CS-programme contenant un nombre fini de paires critiques. Nous expliquons maintenant précisément comment la technique présentée dans [9, 10] calcule une sur-approximation de l'ensemble des descendants à partir de la complétion d'un CS-programme.

Définition 3.2.29 (La fonction comp [10])
Soient arity-limit et predicate-limit deux entiers positifs. Soient R un TRS *linéaire et $Prog$ un* CS*-programme fini, non copiant et normalisé tel que :*

— $>_{Prog}$ *n'est pas cyclique*
— $\forall P \in Pred(Prog)$, *arity*$(P) \leq$ *arity-limit,*
— $\forall i \in \{1, \ldots,$ *arity-limit*$\}$, $card(Pred_i(Prog)) \leq$ *predicate-limit.*
La fonction comp *est définie par :*

```
1 comp(Prog, R)
2 while Il existe une paire critique non convergente H ← B dans Prog do
3 |   Prog = RemoveCycles(norm(H ← B, Prog))
4 end
5 return Prog
```

Pour un TRS donné R et un CS-programme $Prog$, si toutes les paires critiques sont convergentes, alors pour n'importe quel ensemble de termes I tel que $I \subseteq Mod(Prog)$, $Mod(Prog)$ est une sur-approximation de l'ensemble de termes accessibles par réécriture de I par R.

Théorème 3.2.30 ([10])**.** *Soient R un* TRS *linéaire gauche*[7] *et $Prog$ un* CS*-programme non copiant et normalisé.*
Si toutes les paires critiques sont convergentes, alors $Mod(Prog)$ est clos par réécriture de R, c'est à dire $(A \in Mod(Prog) \wedge A \rightarrow_R^* A') \implies A' \in Mod(Prog)$.

Théorème 3.2.31 ([10])**.** *Soient R un* TRS *linéaire et $Prog$ un* CS*-programme normalisé et non copiant.*
La fonction comp *termine toujours et toutes les paires critiques sont convergentes dans* comp *$(Prog, R)$. Par conséquent, $R^*(Mod(Prog)) \subseteq Mod(\mathsf{comp}_R(Prog))$.*

Exemple 3.2.32
Soient $I = \{f(a, a)\}$ et $R = \{f(x, y) \rightarrow u(f(v(x), w(y)))\}$. Intuitivement, l'ensemble exact des descendants est $R^*(I) = \{u^n(f(v^n(a), w^n(a))) \mid n \in \mathbb{N}\}$. On définit

$$Prog = \{P_0(f(x, y)) \leftarrow P_1(x), P_1(y).\ \ P_1(a) \leftarrow .\}$$

On choisit *predicate-limit* $= 4$ et *arity-limit* $= 2$. La paire critique suivante est détectée : $P_0(u(f(v(x), w(y)))) \leftarrow P_1(x), P_1(y)$. La normalisation donne $P_0(u(x)) \leftarrow P_2(x).\ P_2(f(x, y)) \leftarrow P_3(x, y)$ et $P_3(v(x), w(y)) \leftarrow P_1(x), P_1(y)$. Ajouter ces trois CS-clauses dans $Prog$ crée la nouvelle paire critique $P_2(u(f(v(x), w(y)))) \leftarrow P_3(x, y)$. Elle peut être normalisée sans atteindre *predicate-limit* : $P_2(u(x)) \leftarrow P_4(x).\ P_4(f(x, y)) \leftarrow P_5(x, y)$. et $P_5(v(x), w(y)) \leftarrow P_3(x, y)$.

Une fois de plus, une nouvelle paire critique est générée : $P_4(u(f(v(x), w(y)))) \leftarrow P_5(x, y)$. Remarquons ici que nous ne pouvons plus ajouter de prédicat d'arité 1. Procédons à la normalisation de $P_4(u(f(v(x), w(y)))) \leftarrow P_5(x, y)$. étape par étape. Nous choisissons de réutiliser le prédicat P_4. Ainsi, nous générons d'abord la clause $P_4(u(x)) \leftarrow P_4(x)$. Il nous reste à normaliser $P_4(f(v(x), w(y))) \leftarrow P_5(x, y)$. Vu que $P_4(f(v(x), w(y))) \rightarrow_{Prog}^+ P_3(x, y)$, on ajoute la CS-clause $P_3(x, y) \leftarrow P_5(x, y)$ à $Prog$. Remarquons qu'il n'y a plus de paires critiques non convergentes.

7. D'un point de vu théorique, la linéarité gauche est suffisante quand toutes les paires critiques sont convergentes. Cependant, pour rendre toutes les paires critiques convergentes par complétion, les linéarités gauche et droite sont requises (voir le théorème 3.2.31).

Pour résumer, on obtient le CS-programme final $Prog_f$ composé des CS-clauses suivantes :

$$\left\{\begin{array}{lll} P_0(f(x,y)) \leftarrow P_1(x), P_1(y). & P_1(a). & P_0(u(x)) \leftarrow P_2(x). \\ P_3(v(x), w(y)) \leftarrow P_1(x), P_1(y). & P_2(f(x,y)) \leftarrow P_3(x,y). & P_2(u(x)) \leftarrow P_4(x). \\ P_5(v(x), w(y)) \leftarrow P_3(x,y). & P_4(f(x,y)) \leftarrow P_5(x,y). & P_4(u(x)) \leftarrow P_4(x). \\ P_3(x,y) \leftarrow P_5(x,y). & & \end{array}\right\}$$

Finalement, remarquons que pour $Prog_f$, $\mathcal{L}(P_0) = \{u^n(f(v^m(a), w^m(a))) \mid n, m \in \mathbb{N}\}$ et $R^*(I) \subseteq L(P_0)$.

3.3 Sur-approximation des descendants *innermost*

Nous commençons cette section en définissant le problème pour lequel l'algorithme présenté dans la section 3.2 n'arrive pas à calculer une sur-approximation des descendants à partir d'un TRS non linéaire droit.

En partant d'un programme $Prog$ et d'un TRS linéaire gauche et en utilisant l'algorithme de la section 3.2, le programme final $Prog'$ peut être copiant. Par conséquent, le langage reconnu par $Prog'$ peut ne pas être clos par réécriture, c'est-à-dire ne pas reconnaître une sur-approximation de l'ensemble des descendants.

Exemple 3.3.1 (Limite de l'algorithme de la section 3.2 pour un TRS non linéaire droit)
Soit $Prog = \{P(g(x)) \leftarrow Q(x).\ Q(a).\}$ un CS-programme qui reconnaît le langage $Mod(Prog) = \{P(g(a)), Q(a)\}$. Soit $R = \{a \rightarrow b, g(x) \rightarrow f(x,x)\}$.

En exécutant l'algorithme de complétion, on obtient :

$$\mathsf{comp}_R(Prog) = \left\{\begin{array}{ll} P(g(x) \leftarrow Q(x). & Q(a). \\ P(f(x,x)) \leftarrow Q(x). & Q(b). \end{array}\right\}$$

Ce qui donne $Mod(\mathsf{comp}_R(Prog)) = Mod(Prog) \cup \{P(g(b)), P(f(a,a)), P(f(b,b)), Q(b)\}$.

On remarque que $P(f(a,b)) \notin Mod(\mathsf{comp}_R(Prog))$ mais que $P(g(a)) \in Mod(Prog)$ alors que $P(g(a)) \rightarrow_R^* P(f(a,b))$. Donc certains descendants de $Mod(Prog)$ ne sont pas dans $Mod(\mathsf{comp}_R(Prog))$. Toutefois, l'ensemble des descendants obtenus par réécriture *innermost* sont dans $Mod(\mathsf{comp}_R(Prog))$, vu que le seul chemin de réécriture *innermost* issu de $g(a)$ est :

$$g(a) \rightarrow_R^{in} g(b) \rightarrow_{R_{in}} f(b,b)$$

Dans cette section, nous montrons qu'avec une modification mineure de l'algorithme de la section 3.2, un CS-programme initial non copiant et un TRS linéaire gauche (et donc pas nécessairement linéaire droit), nous effectuons une analyse d'atteignabilité pour la réécriture *innermost*. Le théorème 3.3.12 montre qu'avec ces préconditions nous calculons au moins tous les descendants obtenus par réécriture *innermost*.

Nous allons commencer par introduire une suite de définitions et les théorèmes principaux de cette section. Dans une seconde partie, nous présenterons la modification apportée à l'algorithme de la section 3.2. Ensuite, nous présenterons une preuve [8] qui garantit que

8. Par souci de lisibilité, nous choisissons ici de séparer les résultats obtenus de leurs preuves.

la version modifiée de cet algorithme calcule une sur-approximation de l'ensemble des descendants *innermost*. Enfin, nous présenterons un exemple qui prouve qu'une certaine instance du problème de correspondance de Post n'a pas de solution.

3.3.1 Définitions et théorèmes

À partir de la propriété d'irréductibilité d'un terme ou d'une substitution nous allons introduire une propriété plus forte appelée irréductibilité forte (Définition 3.3.2). Puis nous introduirons deux nouveaux types de dérivation (Définition 3.3.6), les dérivations NC et SNC. Puis nous introduirons Mod^R_{NC} (Définition 3.3.8), un sous-ensemble du modèle de Herbrand utilisant obligatoirement des dérivations NC. Finalement, nous présenterons les deux théorèmes principaux de cette section.

Par définition, pour effectuer une réécriture *innermost*, les termes réécrits doivent contenir des sous-termes non réductibles. Cependant, pour un TRS donné, la propriété d'irréductibilité n'est pas préservée par instanciation. C'est-à-dire qu'avoir un terme t et une substitution θ tous deux irréductibles n'implique pas que $\theta(t)$ est irréductible. Nous avons ainsi besoin de définir une propriété plus forte.

Définition 3.3.2 (Irréductibilité forte d'un terme ou d'une substitution)
Soit R un TRS.

- *Un terme t est **fortement irréductible** (par R) si pour tout $p \in Pos^{NonVar}(t)$, pour tout $l \to r \in R$, $t_{|p}$ et l ne sont pas unifiables.*
- *Une substitution θ est **fortement irréductible** si pour tout $x \in \mathcal{X}$, $\theta(x)$ est fortement irréductible.*

Exemple 3.3.3 (Illustration de la définition 3.3.2)
Soient $t = f(x)$, $\theta = (x/a)$, $R = \{f(b) \to b\}$.
Alors $\theta(t) = f(a)$ est fortement irréductible mais t ne l'est pas.

À partir des propriétés d'irréductibilité et d'irréductibilité forte, nous allons définir deux types de dérivations (NC et SNC) qui vont nous permettre de calculer tous les descendants *innermost* ; mais avant cela, nous allons introduire une nouvelle notation.

Notation 3.3.4
Pour un atome H, $Var^{mult}(H)$ représente l'ensemble des variables qui apparaissent plusieurs fois dans H.

Exemple 3.3.5
Nous avons par exemple, $Var^{mult}(P(f(x,y),x,z)) = \{x\}$.

Définition 3.3.6 (Dérivations NC et SNC)
Soient A un atome (éventuellement non clos) et $H \leftarrow B, \sigma$ une CS*-clause.*
L'étape $A \leadsto_{[H \leftarrow B, \sigma]} G$ est NC *(resp.* SNC[9]*) si pour tout $x \in Var^{mult}(H)$, $\sigma(x)$ est irréductible (resp. fortement irréductible) par R.*
Une dérivation est NC *(resp.* SNC*) si toutes ses étapes le sont.*

9. NC pour Non-Copiant, SNC pour Fortement (Strongly en anglais) Non-Copiant.

Exemple 3.3.7 (Illustration de la définition 3.3.6)
Soient $P(g(x,x)) \leftarrow Q(x)$ et $R = \{h(a) \rightarrow b\}$. L'étape de dérivation $P(g(h(y),h(y))) \rightsquigarrow_{[(x/h(y)]} Q(h(y))$ est NC ($h(y)$ est irréductible), mais cette dérivation n'est pas SNC ($h(y)$ n'est pas fortement irréductible).

Maintenant que nous avons introduit les dérivations NC, nous pouvons définir un sous-ensemble de $Mod(Prog)$ correspondant à ce type de dérivation.

Notation 3.3.8
Soient $Prog$ un CS*-programme et R un* TRS*. L'ensemble des atomes clos A tels qu'il existe une dérivation* NC $A \rightsquigarrow^* \emptyset$ *est noté* $Mod^R_{NC}(Prog)$.

Remarque 3.3.9
L'ensemble $Mod^R_{NC}(Prog) \subseteq Mod(Prog)$ et si $Prog$ est non-copiant, alors $Mod^R_{NC}(Prog) = Mod(Prog)$.

Exemple 3.3.10
Soient $R = \{a \rightarrow b\}$ et $Prog = \left\{ \begin{array}{ll} P(f(x),f(x)) \leftarrow Q(x). & \\ Q(a). & Q(b). \end{array} \right\}$.

Alors $P(f(a),f(a)) \notin Mod^R_{NC}(Prog)$, et donc $Mod^R_{NC}(Prog) \neq Mod(Prog)$.

Maintenant que nous avons introduit $Mod^R_{NC}(Prog)$, nous pouvons établir le théorème intermédiaire suivant qui traite de la clôture par la réécriture *innermost* :

Théorème 3.3.11. *Soient $Prog$ un* CS*-programme normalisé et R un* TRS *linéaire gauche. Si toutes les paires critiques sont convergentes par dérivations* SNC*, alors $Mod^R_{NC}(Prog)$ est clos par réécriture innermost par R, c'est-à-dire*

$$(A \in Mod^R_{NC}(Prog) \wedge A \rightarrow^*_{R_{in}} A') \implies A' \in Mod^R_{NC}(Prog)$$

Enfin, le théorème défini ci-dessus nous permet de poser le théorème final qui garantit que la version modifiée de l'algorithme calcule une sur-approximation de l'ensemble des descendants *innermost*.

Dans le résultat suivant, nous considérons le CS-programme initial $Prog$ non-copiant et le programme éventuellement copiant $Prog'$ composé des clauses ajoutées par le processus de complétion. Nous considérons également que la fonction de normalisation norm rend les paires critiques convergentes par dérivations SNC. Nous verrons comment s'assurer de ce fait dans la section suivante.

Théorème 3.3.12. *Soient R un système de réécriture linéaire gauche et $Prog'' = Prog \cup Prog'$ un* CS*-programme normalisé tel que $Prog$ est non-copiant et toutes les paires critiques de $Prog''$ sont convergentes par dérivations* SNC*. Si $A \in Mod(Prog)$ et $A \rightarrow^*_R A'$ par une stratégie innermost, alors $A' \in Mod(Prog'')$.*

Afin que ce théorème puisse s'appliquer, il est nécessaire que la fonction de normalisation norm rende les paires critiques convergentes par dérivation SNC. Nous présentons dans la partie suivante une modification de l'algorithme permettant de garantir cette propriété.

3.3.2 Modification de l'algorithme [9, 10]

Nous montrons dans l'exemple 3.3.13 qu'exécuter la complétion de [9] avec un système de réécriture non linéaire droit peut ajouter des clauses copiantes et que certains descendants *innermost* peuvent manquer.

Exemple 3.3.13
Soit R un TRS non linéaire droit :

$$R = \left\{ \begin{array}{ll} f(x) \to g(h(x), h(x)) & h(a) \to b \\ i(x) \to g(x, x) & \end{array} \right\}$$

Soit $Prog$ un programme initial non-copiant :

$$Prog = \left\{ \begin{array}{ll} P(i(x)) \leftarrow Q_1(x). & Q_2(a). \\ P(f(x)) \leftarrow Q_2(x). & \end{array} \right\}$$

Nous commençons avec $Prog' = \emptyset$. La méthode de complétion détecte les paires critiques suivantes :

1. $P(g(x, x)) \leftarrow Q_1(x)$ ajoutée à $Prog'$,
2. $P(g(h(x), h(x))) \leftarrow Q_2(x)$.
   On a $P(g(x, x)) \leftarrow Q_1(x)$ qui est une clause synchronisante et qui permet de by-passer cette clause en $Q_1(h(x)) \leftarrow Q_2(x)$.
   $Q_1(h(x)) \leftarrow Q_2(x)$ ajoutée à $Prog'$,
3. $Q_1(b) \leftarrow$, ajoutée à $Prog'$.

Toutes les paires critiques ont été détectées, et sont donc toutes convergentes dans $Prog'' = Prog \cup Prog'$. Cependant, $P(f(a)) \to_R P(g(h(a), h(a))) \to_R P(g(b, h(a)))$ par dérivation *innermost*, alors que $P(f(a)) \in Mod(Prog)$ et $P(g(b, h(a))) \notin Mod(Prog'')$. La clause $P(g(x, x)) \leftarrow Q_1(x)$ empêche la réduction de $P(g(b, h(a)))$ et par conséquent, il est impossible d'obtenir l'ensemble de tous les descendants *innermost*.

Afin d'éviter ce problème il est nécessaire de modifier l'algorithme présenté dans la section 3.2. Nous modifions la fonction norm en remplaçant la fonction by-pass par la fonction by-pass-SNC. La fonction by-pass-SNC a le même rôle que la fonction by-pass mais avec la contrainte supplémentaire de n'effectuer un by-pass qu'avec des dérivations SNC.

Définition 3.3.14 (La fonction by-pass-SNC)
Soit $Prog$ un CS*-programme normalisé et soient $H \leftarrow B$ une* CS*-clause.*

```
1 by-pass-snc(H ← B, Prog)
2 if Il n'y a pas de clause (H' ← A) ∈ Prog synchronisante
3 et non empty pouvant réécrire H par dérivation SNC then
4 |   return H ← B
5 else
6 |   // Dans ce cas, il existe une substitution θ telle que H = θ(H')
7 |   return by-pass-snc(θ(A) ← B, Prog)
8 end
```

Reprenons l'exemple 3.3.13 avec l'algorithme modifié.

Exemple 3.3.15
Regardons le comportement de la paire critique donnée $P(g(h(x), h(x))) \leftarrow Q_2(x)$ dans l'exemple 3.3.13. Ajouter la clause $Q_1(h(x)) \leftarrow Q_2(x)$ rend la clause convergente dans $Prog''$ (dans l'exemple 3.3.13), mais non convergente par dérivation SNC. En effet [10],

$$P(g(h(x'),\ h(x'))) \rightsquigarrow_{[x/h(x')]} Q_1(h(x'))\ \rightsquigarrow_{[x/x']} Q_2(x')$$

Mais l'étape de résolution suivante $P(g(h(x'), h(x'))) \rightsquigarrow_{[P(g(x,x))\leftarrow Q_1(x), x/h(x')]} Q_1(h(x'))$ n'est pas SNC car $x \in \mathit{Var}^{mult}(P(g(x, x)))$ et $x/h(x')$ n'est pas fortement irréductible.

Avec la version modifiée de l'algorithme, la normalisation de la clause $P(g(h(x), h(x))) \leftarrow Q_2(x)$ se fait par dérivation SNC. On a alors $P(g(h(x), h(x))) \leftarrow Q_2(x)$ qui ne sera pas modifiée par by-pass-SNC et donc normalisée par les clauses suivantes : $P(g(x, y)) \leftarrow Q_3(x, y)$. $Q_3(h(x), h(x)) \leftarrow Q_2(x)$. Après avoir ajouté ces clauses, de nouvelles paires critiques sont détectées, et les clauses $Q_3(b, h(x)) \leftarrow Q_2(x)$. $Q_3(h(x), b) \leftarrow Q_2(x)$. $Q_3(b, b) \leftarrow$. vont être ajoutées. Le programme final est alors :

$$Prog_f = Prog \cup \left\{ \begin{array}{ll} P(g(x, x)) \leftarrow Q_1(x). & Q_3(b, b). \\ P(g(x, y)) \leftarrow Q_3(x, y). & Q_3(h(x), h(x)) \leftarrow Q_2(x). \\ Q_3(b, h(x)) \leftarrow Q_2(x). & Q_3(h(x), b) \leftarrow Q_2(x). \end{array} \right\}$$

Les descendants *innermost* sont bien tous reconnus par $Prog_f$ et notamment $P(g(b, h(a))) \in Mod(Prog'')$.

3.3.3 Démonstrations

Pour rendre cette section plus lisible, nous avons décidé de séparer la solution proposée des preuves. Dans cette partie, nous donnons les preuves montrant que cette solution est correcte. Afin d'obtenir cette solution, il a été nécessaire de prouver un résultat sur la clôture par réécriture *innermost* (Théorème 3.3.11). Pour établir et prouver ce théorème, nous donnons une suite de lemmes intermédiaires ainsi que leurs preuves.

Nous avons défini la propriété d'irréductibilité forte (Définition 3.3.2) à partir de la propriété d'irréductibilité. Les trois lemmes et le corollaire suivants donnent des résultats autour de ces deux propriétés.

Lemme 3.3.16 (Irréductibilité forte ⇒ irréductibilité)
Si t est fortement irréductible, alors t est irréductible.

Démonstration. Nous montrons ce résultat par contraposée.
Si $t \rightarrow_{[p,l\rightarrow r,\sigma]} t'$, alors $t_{|p} = \sigma(l)$. Puisque $Var(t) \cap Var(l) = \emptyset$ par définition, alors $t_{|p}$ et l ne sont pas unifiables par σ. □

Lemme 3.3.17 (Préservation de l'irréductibilité forte dans les sous-termes)
Si t est fortement irréductible, alors pour tout $p \in Pos(t)$, $t_{|p}$ est fortement irréductible.
Pour une substitution θ, si $\theta(t)$ est fortement irréductible, alors pour tout $x \in Var(t)$, $\theta(x)$ est fortement irréductible (mais t ne l'est pas forcement).

10. On ajoute ici des prime aux noms de variables pour éviter les conflits de nommages.

Démonstration. Trivial. □

Corollaire 3.3.18
Soient α, θ des substitutions, Si $\alpha \circ \theta$ sont fortement irréductibles, alors α est fortement irréductible.

Les définitions et lemmes précédents s'appliquent aux atomes et aux séquences d'atomes de façon triviale.

Lemme 3.3.19 (Fermeture par instanciation)
Si t est fortement irréductible et θ et irréductible, alors $\theta(t)$ est irréductible.

Démonstration. Par contraposée.
Si $\theta(t) \rightarrow_{[p,l\rightarrow r,\sigma]} t'$, alors $(\theta(t))_{|p} = \sigma(l)$. Nous distinguons deux cas :
- Si $p \notin Pos^{NonVar}(t)$, alors il existe une variable x et une position p' telles que $(\theta(x))_{|p'} = \sigma(l)$. Donc θ est réductible.
- Sinon, $\theta(t_{|p}) = \sigma(l)$. Alors $t_{|p}$ et l sont unifiables, et donc t n'est pas fortement irréductible.

□

Exemple 3.3.20 (Illustration du lemme 3.3.19)
Soient $t = f(x)$, $\theta = (x/g(y))$, et $R = \{g(a) \rightarrow b\}$. Alors t est fortement irréductible, θ est irréductible et $\theta(t) = f(g(y))$ est irréductible. Remarquons cependant que $\theta(t)$ n'est pas fortement irréductible.

À partir des propriétés d'irréductibilité et d'irréductibilité forte, nous avons défini deux types de dérivation (Définition 3.3.6), les dérivations NC et les dérivations SNC. Les deux remarques et les deux lemmes suivants donnent des résultats autour de ces deux types de dérivations.

Remarque 3.3.21
La réduction SNC implique la réduction NC.

Remarque 3.3.22
Si la clause $H \leftarrow B$ est non-copiante, alors $Var^{mult}(H) = \emptyset$, et donc l'étape $A \rightsquigarrow_{[H\leftarrow B,\sigma]} G$ est SNC (et NC).

Lemme 3.3.23
Si $A \rightarrow_{[H\leftarrow B,\sigma]} G$ est SNC *et pour tout $x \in Var^{mult}(H)$, pour tout $y \in Var(\sigma(x))$, $\theta(y)$ est irréductible, alors $\theta(A) \rightarrow_{[H\leftarrow B,\theta\circ\sigma]} \theta(G)$ est* NC.

Démonstration. Soit $x \in Var^{mult}(H)$, alors $\sigma(x)$ est fortement irréductible. D'après le lemme 3.3.19, $\theta \circ \sigma(x)$ est irréductible. Donc $\theta(A) \rightarrow_{[H\leftarrow B,\theta\circ\sigma]} \theta(G)$ est NC. □

Lemme 3.3.24
Si $\sigma'(A) \rightsquigarrow_{[H\leftarrow B,\gamma]} G$ est NC*, alors $A \rightsquigarrow_{[H\leftarrow B,\theta]} G'$ est* NC *et il existe une substitution α tel que $\alpha(G') = G$ et $\alpha \circ \theta = \gamma \circ \sigma'$.*

Démonstration. Nous prouvons ce lemme par l'absurde.
L'atome $\sigma'(A)$ étant plus instancié que A, s'il est possible de dériver $\sigma'(A)$ par $H \leftarrow B$ alors il est possible de dériver A par cette même CS-clause. On a $A \leadsto_{[H \leftarrow B, \theta]} G'$ et il existe une substitution α telle que $\alpha(G') = G$ et $\alpha \circ \theta = \gamma \circ \sigma'$.
Si $A \leadsto_{[H \leftarrow B, \theta]} G'$ n'est pas NC, alors il existe $x \in Var^{mult}(H)$ telle que $\theta(x)$ est réductible. Alors $\gamma(x) = \gamma \circ \sigma'(x) = \alpha \circ \theta(x)$ est réductible. Donc $\sigma'(A) \leadsto_{[H \leftarrow B, \gamma]} G$ n'est pas NC, ce qui contredit l'énoncé du lemme. □

À partir des lemmes ci-dessus, nous pouvons maintenant démontrer le théorème 3.3.11.

Démonstration. Soit $A \in Mod^R_{NC}(Prog)$ tel que $A \rightarrow_{R_{in}, l \rightarrow r} A'$. Alors $A_{|i} = C[\sigma(l)]$ pour au moins un $i \in \mathbb{N}$, σ est irréductible et $A' = A[i \leftarrow C[\sigma(r)]]$.

Vu que $A \in Mod^R_{NC}(Prog)$, $A \leadsto^* \emptyset$ par dérivation NC. Vu que $Prog$ est normalisé, la résolution logique consomme un par un les symboles dans t_i, donc $G''_0 = A \leadsto^* G''_k \leadsto^* \emptyset$ par dérivation NC et il existe un atome $A'' = P(t_1, \ldots, t_j, \ldots, t_{arity(P)})$ dans G''_k et j tel que $t_j = \sigma(l)$ et la racine de t_j est consommée (ou t_j est retiré) durant l'étape $G''_k \leadsto G''_{k+1}$.

Vu que t_j est réductible par R et $A \in Mod^R_{NC}(Prog)$, $t_j = \sigma(l)$ a un seul antécédent dans A. Alors $A' \leadsto^* G''_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_{arity(P)})]$ par dérivations NC (I).

Soient les variables $x_1, \ldots, x_{arity(P)}$ telles que $\{x_1, \ldots, x_{arity(P)}\} \cap Var(l) = \emptyset$ et σ' définie par $\forall i,\ \sigma'(x_i) = t_i$ et $\forall x \in Var(l),\ \sigma'(x) = \sigma(x)$.
alors $\sigma'(P(x_1, \ldots, x_{j-1}, l, x_{j+1}, \ldots, x_{arity(P)})) = A''$.

On peut extraire de $G''_k \leadsto^* \emptyset$ la dérivation $G_k = A'' \leadsto_{[\gamma_k]} G_{k+1} \leadsto_{[\gamma_{k+1}]} G_{k+2} \leadsto^* \emptyset$, qui est NC. D'après le lemme 3.3.24, il existe un entier positif $u > k$, une dérivation NC $G'_k = P(x_1, \ldots, l, \ldots, x_{arity(P)}) \leadsto^*_{[\theta]} G'_u$ et une substitution α tels que $\alpha(G'_u) = G_u$, $\alpha \circ \theta = \gamma_{u-1} \circ \ldots \circ \gamma_k \circ \sigma'$, G'_u est plat et $\forall i,\ k < i < u$ implique G'_i n'est pas plat. Autrement dit, il existe une paire critique qui est convergente par dérivations SNC. Donc $\theta(G'_k[l \leftarrow r]) \rightarrow^* G'_u$ par dérivations SNC.

Soit $\gamma = \gamma_{u-1} \circ \ldots \circ \gamma_k$. S'il existe une clause $H \leftarrow B$ utilisée dans cette dérivation et $x \in Var^{mult}(H)$ tel que $\alpha \circ \theta(x)$ est réductible, alors il existe i et p tels que $\alpha \circ \theta(x) = \gamma \circ \sigma'(x) = \gamma((t_i)_{|p})$ (car σ est irréductible). Notons que γ est un unifieur, alors $\gamma(x) = \gamma((t_i)_{|p})$. Donc $\gamma(x) = \gamma(t_i|_p) = \gamma \circ \sigma'(x) = \alpha \circ \theta(x)$, qui est réductible. C'est impossible parce que $x \in Var^{mult}(H)$ et que $G_k \leadsto^*_{[\gamma]} G_u$ est une dérivation NC.

Par conséquent, d'après le lemme 3.3.23, $\alpha \circ \theta(G'_k[l \leftarrow r]) \rightarrow^* \alpha(G'_u) = G_u \leadsto^* \emptyset$ par dérivations NC. Notons que $\alpha \circ \theta(G'_k[l \leftarrow r]) = \gamma \circ \sigma'(P(x_1, \ldots, r, \ldots, x_{arity(P)})) = \gamma(P(t_1, \ldots, \sigma(r), \ldots, t_{arity(P)}))$, donc $\gamma(P(t_1, \ldots, \sigma(r), \ldots, t_{arity(P)})) \leadsto^* \emptyset$ par dérivations NC.
D'après le lemme 3.3.24 on a :
$P(t_1, \ldots, \sigma(r), \ldots, t_{arity(P)}) \leadsto^* \emptyset$ par dérivations NC. Considérons la dérivation (I) à nouveau, on a $A' \leadsto^* G''_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_{arity(P)})] \leadsto^* \emptyset$ par dérivations NC. Autrement dit, $A' \in Mod^R_{NC}(Prog)$.

Par induction triviale, cette preuve peut être étendue aux réécritures multiples. □

À l'aide du théorème que nous venons de prouver, nous pouvons maintenant prouver le théorème final de cette section (Théorème 3.3.12).

Démonstration. Vu que $Prog$ (le programme initial) est non-copiant, $Mod(Prog) = Mod^R_{NC}(Prog)$. Donc $A \in Mod^R_{NC}(Prog)$ et sachant que $Prog \subseteq Prog''$, on a $A \in Mod^R_{NC}(Prog'')$. D'après le théorème 3.3.11, $A' \in Mod^R_{NC}(\ Prog'')$ et sachant que $Mod^R_{NC}(Prog'') \subseteq Mod(Prog'')$, on a $A' \in Mod(Prog'')$. □

3.3.4 Exemple

Cette approche peut par exemple s'appliquer au Problème de Correspondance de Post PCP.

Exemple 3.3.25
Considérons l'instance du Problème de Correspondance de Post composée des dominos $(ab,\ aa)$ et $(bba,\ bb)$. Pour modéliser cette instance par un langage d'arbres on utilise l'alphabet $\Sigma = \{\mathit{Test}^{\backslash 1}, a^{\backslash 1}, b^{\backslash 1}, 0^{\backslash 0}\}$, le système de réécriture R donné ci-dessous :

$$R = \left\{ \begin{array}{ll} \mathit{Test}(x) \to g(x,x), & g(0,0) \to True, \\ g(a(b(x)), a(a(y))) \to g(x,y), & g(b(b(a(x))), b(b(y))) \to g(x,y) \end{array} \right\}$$

et le langage initial $I = \{Test(t) \mid t \in T(\{a,b,0\}),\ t \neq 0\}$ généré par le prédicat P_0 avec le CS-programme $Prog$:

$$Prog = \left\{ \begin{array}{ll} P_0(\mathit{Test}(z)) \leftarrow P_1(z). & P_1(a(z)) \leftarrow P_2(z). \\ P_1(b(z)) \leftarrow P_2(z). & P_2(a(z)) \leftarrow P_2(z). \\ P_2(b(z)) \leftarrow P_2(z). & P_2(0). \end{array} \right\}$$

Cette instance de PCP admet au moins une solution si et seulement si $True$ est atteignable dans $R^*(\mathcal{I})$. Remarquons que R n'est pas linéaire droit ; cependant tous les descendants sont *innermost*, et sont reconnus par le CS-programme obtenu par complétion (théorème 3.3.12) :

$$\mathsf{comp}_R(Prog) = Prog \cup Prog'$$

où :

$$Prog' = \left\{ \begin{array}{lll} P_0(g(x,x)) \leftarrow P_1(x). & P_0(g(x,y)) \leftarrow P_4(x,y). & P_4(x,a(x)) \leftarrow P_2(x). \\ P_0(g(x,y)) \leftarrow P_5(x,y). & P_5(x,b(x)) \leftarrow P_2(x). & P_0(g(x,y)) \leftarrow P_6(x,y). \\ P_6(x,b(y)) \leftarrow P_7(x,y). & P_7(x,a(x)) \leftarrow P_2(x). & \end{array} \right\}$$

Remarquons que $P_0(\mathit{True}) \notin Mod(\mathsf{comp}_R(Prog))$, ce qui prouve que cette instance du PCP n'as pas de solution.

Un prototype mettant en œuvre cette technique est en cours de développement. Nous avons modélisé et exécuté le calcul d'une sur-approximation de l'ensemble des descendants pour des programmes un peu plus gros.

Exemple 3.3.26
Nous proposons de modéliser le programme ci-dessous qui intervertit la valeur de deux variables et retourne `vrai` si les variables ont bien été interverties.

1 $\mathit{oldX} \leftarrow x$
2 $\mathit{oldY} \leftarrow y$
3 $\mathit{temp} \leftarrow x$
4 $x \leftarrow y$
5 $y \leftarrow \mathit{temp}$
6 $x = \mathit{oldY} \wedge y = \mathit{oldX}$

Algorithme 2 : `swap(x, y)`

Pour vérifier que ce programme fonctionne correctement pour tout $x, y \in \mathbb{N}$, prenons comme langage des configurations indésirables $\mathcal{B}ad = \{false\}$. Voici l'alphabet que nous allons utiliser :

— $s^{\backslash 1}, 0^{\backslash 0}$: les entiers de Peano.
— $true^{\backslash 0}, false^{\backslash 0}$: les littéraux vrai et faux.
— $eq^{\backslash 2}, and^{\backslash 2}$: l'égalité et le "et" logique.
— $null^{\backslash 0}$: la valeur par défaut des variables.
— $swap^{\backslash 2}$: l'appel à la fonction swap .
— $i1^{\backslash 5}, i2^{\backslash 5}, i3^{\backslash 5}, i4^{\backslash 5}, i5^{\backslash 5}, i6^{\backslash 5}$: chaque instruction de la fonction `swap`.

Pour $j \in \{1, \ldots, 6\}$, $ij(x, y, oldX, oldY, temp)$ représente l'état du programme, c'est-à-dire les valeurs des variables paramètres et locales.

Notre système de réécriture $R = R_{swap} \cup R_{eq} \cup R_{and}$ sera composé de trois ensembles de règles modélisant respectivement le programme swap, l'opération d'égalité et le "et" logique.

$$R_{swap} = \left\{ \begin{array}{lcl} swap(x,y) & \to & i1(\ x,\ y,\ null,\ null,\ null\ ) \\ i1(x,y,oldX,oldY,temp) & \to & i2(\ x,\ y,\ x,\ oldY,\ temp\ ) \\ i2(x,y,oldX,oldY,temp) & \to & i3(\ x,\ y,\ oldX,\ y,\ temp\ ) \\ i3(x,y,oldX,oldY,temp) & \to & i4(\ x,\ y,\ oldX,\ oldY,\ x\ ) \\ i4(x,y,oldX,oldY,temp) & \to & i5(\ y,\ y,\ oldX,\ oldY,\ temp\ ) \\ i5(x,y,oldX,oldY,temp) & \to & i6(\ x,\ temp,\ oldX,\ oldY,\ temp\ ) \\ i6(x,y,oldX,oldY,temp) & \to & and(eq(x,oldY), eq(y,oldX)\ ) \end{array} \right\}$$

$$R_{eq} = \left\{ \begin{array}{lcl} eq(s(x),s(y)) & \to & eq(x,y) \\ eq(0,0) & \to & true \\ eq(s(x),0) & \to & false \\ eq(0,s(x)) & \to & false \end{array} \right\}$$

$$R_{and} = \left\{ \begin{array}{lcl} and(true,true) & \to & true \\ and(false,x) & \to & false \\ and(x,false) & \to & false \end{array} \right\}$$

Enfin, il nous faut le CS-programme $Prog$ représentant le langage initial $Prog = \{swap(s^*(0), s^*(0))\}$:

$$Prog = \left\{ \begin{array}{l} P_{init}(swap(x,y)) \leftarrow P_{Nat}(x), P_{Nat}(y). \\ P_{Nat}(s(x)) \leftarrow P_{Nat}(x). \\ P_{Nat}(0). \end{array} \right\}$$

Grâce au prototype, nous pouvons calculer l'ensemble des descendants. Après la détection de 20 paires critiques, nous obtenons le CS-programme $Prog'$ suivant :

$$Prog' = \left\{ \begin{array}{l} P_{init}(swap(x,y)) \leftarrow P_{Nat}(x), P_{Nat}(y). \\ P_{Nat}(s(x)) \leftarrow P_{Nat}(x). \\ P_{Nat}(0). \\ P_{init}(i1(x,y,oldX,oldY,temp)) \leftarrow P_1(x,y,oldX,oldY,temp). \\ P_1(x,y,null,null,null) \leftarrow Pnat(x), Pnat(y). \\ P_{init}(i2(x,y,x,oldY,temp)) \leftarrow P_1(x,y,oldX,oldY,temp). \\ P_{init}(i3(oldX,y,oldX,y,temp)) \leftarrow P_1(oldX,y,oldX,oldY,temp). \\ P_{init}(i4(oldX,oldY,oldX,oldY,oldX)) \leftarrow P_1(oldX,oldY,oldX,oldY,temp). \\ P_{init}(i5(oldY,oldY,temp,oldY,temp)) \leftarrow P_1(temp,oldY,oldX,oldY,temp). \\ P_{init}(i6(oldY,temp,temp,oldY,temp)) \leftarrow P_1(temp,oldY,oldX,oldY,temp). \\ P_{init}(and(x,y)) \leftarrow P_2(x,y). \\ P_2(eq(oldY,oldY),eq(temp,temp)) \leftarrow P_1(temp,oldY,oldX,oldY,temp). \\ P_2(true,eq(temp,temp)) \leftarrow Pnat(temp). \\ P_2(eq(y,y),eq(temp,temp)) \leftarrow Pnat(temp),Pnat(y). \\ P_2(eq(oldY,oldY),true) \leftarrow Pnat(oldY). \\ P_2(true,true) \leftarrow . \\ P_{init}(true). \end{array} \right\}$$

Notre langage $\mathcal{B}ad$ étant constitué du seul terme $false$, pour tester s'il appartient au langage $\mathcal{L}_{Prog'}(P_{init})$, il suffit de regarder si l'atome $P_{init}(false)$ peut être dérivé en la séquence d'atomes vide. Aucune des clauses de $Prog'$ ayant P_{init} comme symbole de prédicat dans leur tête ne peut dériver cet atome. On en conclut que $\mathcal{B}ad \cap \mathcal{L}(Prog') = \emptyset$ et donc que notre programme fonctionne sur l'ensemble des entiers naturels.

Nous avons réalisé d'autre tests comme par exemple sur des programmes de tri de listes. Cependant, nous nous retrouvons souvent avec des cycles générant un nombre infini de paires critiques. La méthode mise au point pour les enlever a un impact trop important sur la précision pour permettre l'analyse d'atteignabilité sur ce genre de programmes plus complexes.

3.4 Élimination de clauses copiantes

Dans cette section, nous proposons un processus (voir définition 3.4.5) qui transforme une CS-clause copiante en un ensemble de CS-clauses non copiantes. Dans une seconde partie, nous présentons une méthode pour forcer la terminaison de ce processus en calculant une sur-approximation du langage généré. De cette manière, même si le TRS n'est pas linéaire droit et que par conséquent des clauses copiantes peuvent être générées durant le processus de complétion, celles-ci peuvent être gérées dès qu'elles apparaissent. Ainsi, le CS-programme final est non-copiant et le Théorème 3.2.30 s'applique et une sur-approximation de l'ensemble des descendants peut être calculée.

Par exemple, soit $Prog = \{P(f(x,x)) \leftarrow Q(x).\ Q(s(x)) \leftarrow Q(x).\ Q(a) \leftarrow\}$. On notera que le langage généré par P est $\{f(s^n(a), s^n(a)) \mid n \in \mathbb{N}\}$. Nous pouvons ajouter un nouveau symbole de prédicat Q^2 qui génère le langage $\{(t,t) \mid Q(t) \in Mod(Prog)\}$ et on transforme la clause copiante $P(f(x,x)) \leftarrow Q(x)$ en une non-copiante $P(f(x,y)) \leftarrow Q^2(x,y)$. Maintenant, Q^2 peut être défini par les clauses $Q^2(s(x),s(x)) \leftarrow Q(x)$ et $Q^2(a,a) \leftarrow$. Malheureusement, $Q^2(s(x),s(x)) \leftarrow Q(x)$ est copiante. En utilisant la même idée, on

transforme cette clause en la clause non-copiante $Q^2(s(x), s(y)) \leftarrow Q^2(x, y)$. Le corps de cette clause utilise le prédicat Q^2 qui est déjà défini. Alors, le processus termine avec

$$\begin{array}{rcl} Prog' = \{P(f(x,y)) & \leftarrow & Q^2(x,y).\ Q^2(s(x),s(y)) \leftarrow Q^2(x,y).\ Q^2(a,a) \leftarrow. \\ Q(s(x)) & \leftarrow & Q(x).\ Q(a) \leftarrow\} \end{array}$$

On notera que $Prog'$ est non-copiant et génère le même langage que $Prog$. Les clauses qui définissent Q sont inutiles dans $Prog'$, mais dans le cas général il est nécessaire de les conserver.

Formalisons le processus pour le cas général.

La fonction expand permet à partir d'un atome plat et linéaire utilisant un symbole de prédicat P et d'un entier n de générer un atome plat d'arité $arity(P) \times n$ avec le symbole de prédicat P^n.

Définition 3.4.1 (La fonction expand)
Soient $P(x_1, \ldots, x_k)$ un atome linéaire, $x_1, \ldots, x_k$ des variables et n un nombre positif. On définit :

$$\mathsf{expand}(P(x_1, \ldots, x_k), n) = \begin{cases} P^n(x_1^1, \ldots, x_k^1, \ldots, x_1^n, \ldots, x_k^n) & si\ n > 1 \\ P(\overrightarrow{t}) & avec\ \overrightarrow{t} = (x_1, \ldots, x_k)\ sinon. \end{cases}$$

La fonction copy permet à partir d'un atome utilisant un symbole de prédicat P et d'un entier n de générer un atome d'arité $arity(P) \times n$ avec le symbole de prédicat P^n et n fois les paramètres de l'atome en entrée.

Définition 3.4.2 (La fonction copy)
Soit $P(\overrightarrow{t})$ un atome (avec $\overrightarrow{t}$ un tuple de termes) et n un entier positif. On définit :

$$\mathsf{copy}(P(\overrightarrow{t}), n) = \begin{cases} P^n(\underbrace{\overrightarrow{t}, \ldots, \overrightarrow{t}}_{n\ fois}) & si\ n > 1 \\ P(\overrightarrow{t}) & sinon. \end{cases}$$

À l'aide de la fonction copy, nous définissons la fonction $\mathsf{clauses}^{new}$ qui permet à partir d'une clause et d'un entier n de générer une nouvelle clause qui a une tête d'arité $arity(Q) \times n$.

Définition 3.4.3 (La fonction $\mathsf{clauses}^{new}$)
Soit $Prog$ un ensemble de CS*-clauses. Soient $Q^n(\overrightarrow{t})$ un atome où Q est un symbole de prédicat qui apparaît dans $Prog$ et n un entier positif avec $n > 1$.*

$$\mathsf{clauses}^{new}(Q^n(\overrightarrow{t}), Prog) = \{\mathsf{copy}(Q(\overrightarrow{t}), n) \leftarrow B \mid Q(\overrightarrow{t}) \leftarrow B \in Prog\}.$$

À l'aide des fonctions expand et $\mathsf{clauses}^{new}$, on définit la fonction $\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Prog}}$ qui à partir d'une clause copiantes génère un ensemble de clauses non copiante qui reconnaissent le même langage.

Définition 3.4.4 ($\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Prog}}$)
Soit $Prog$ un CS-programme normalisé. Soit $P(\overrightarrow{t}) \leftarrow Q_1, \ldots, Q_n$ une clause copiante n'appartenant pas à $Prog$. Soit $Var(\overrightarrow{t}) = \{x_1, \ldots, x_k\}$ l'ensemble de variables apparaissant dans $\overrightarrow{t}$. Soit $\{m_1, \ldots, m_k\} \subseteq \mathbb{N}$ l'ensemble des entiers positifs tel que x_i apparaît exactement m_i fois dans $\overrightarrow{t}$. On définit :

$$\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Prog}}(P(\overrightarrow{t}) \leftarrow Q_1, \ldots, Q_n) = \begin{array}{l}\{P(\overrightarrow{t'}) \leftarrow Q'_1, \ldots, Q'_n\} \cup \\ \bigcup_{Q'_i \neq Q_i} (\mathsf{clauses}^{new}(Q'_i, Prog'))\end{array}$$

où $Prog' = Prog \cup \{P(\overrightarrow{t'}) \leftarrow Q'_1, \ldots, Q'_n\}$, $\overrightarrow{t'}$ est obtenu depuis $\overrightarrow{t}$ en remplaçant les différentes occurrences de $x_j \in Var(\overrightarrow{t})$ par $x_j^1, \ldots, x_j^{m_j}$ et $Q'_i = \mathsf{expand}(Q_i, max_i)$ avec $max_i = \left(\underset{x_i \in Var(Q_i)}{Max} \{m_i\} \right)$ quand $max_i > 1$.

À l'aide de la fonction $\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Prog}}$, nous définissons la fonction $\mathsf{uncopying}(Prog)$ qui transforme un CS-programme copiant en un CS-programme équivalent mais non copiant.

Définition 3.4.5 (La fonction $\mathsf{uncopying}(Prog)$)
Soit $Prog$ un CS-programme normalisé. On définit :

$$\mathsf{uncopying}(Prog) = \begin{cases} \mathsf{uncopying}(\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Reste}}(H \leftarrow B) \cup Reste) & si \text{ COND} \\ Prog & sinon. \end{cases}$$

Où COND *est : il existe une clause $H \leftarrow B$ copiante, auquel cas $Reste = Prog \setminus \{H \leftarrow B\}$.*

Illustrons les définitions précédentes dans l'exemple 3.4.6.

Exemple 3.4.6
Soit $Prog$ un CS-programme normalisé et copiant défini par :

$$Prog = \{P(f(x)) \leftarrow Q_1(x).\ Q_1(a) \leftarrow .\ Q_1(b) \leftarrow .\ P(g(x,x) \leftarrow Q_1(x)\}$$

Alors, d'après la définition 3.4.5, on a :

$$\mathsf{uncopying}(Prog) = \mathsf{uncopying}(\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Reste}}(P(g(x,x)) \leftarrow Q_1(x)) \cup Reste) \tag{3.1}$$

où $Reste = \{P(f(x)) \leftarrow Q_1(x).\ \ Q_1(a) \leftarrow .\ \ Q_1(b) \leftarrow .\}$.

En appliquant la définition 3.4.4,

$$\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Reste}}(P(g(x,x)) \leftarrow Q_1(x)) = \begin{array}{l}\{P(g(x^1,x^2)) \leftarrow \mathsf{expand}(Q_1(x),2)\} \bigcup \\ \mathsf{clauses}^{new}\Big(Q_1^2(x^1,x^2), \\ Reste \bigcup \{P(g(x^1,x^2)) \leftarrow \mathsf{expand}(Q_1(x),\ 2)\}\Big)\end{array}$$

Comme $\mathsf{expand}(Q_1(x),2) = Q_1^2(x^1,x^2)$ d'après la définition 3.4.1, on a :

$$\mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Reste}}(P(g(x,x)) \leftarrow Q_1(x)) = \begin{array}{l}\{P(g(x^1,x^2)) \leftarrow Q_1^2(x^1,x^2)\} \bigcup \\ \mathsf{clauses}^{new}\Big(\ Q_1^2(x^1,x^2), \\ Reste \bigcup \{P(g(x^1,x^2)\ ) \leftarrow Q_1^2(x^1,x^2)\})\end{array}$$

En appliquant la définition 3.4.3, on obtient :

$$\begin{array}{rl} & \mathsf{clauses}^{new}(Q_1^2(x^1,x^2), Reste \cup \{P(g(x^1,x^2)) \leftarrow Q_1^2(x^1,x^2)\}) \\ = & \{\mathsf{copy}(Q_1(a),2) \leftarrow .\} \cup \{\mathsf{copy}(Q_1(b),2) \leftarrow .\} \end{array}$$

Par conséquent, d'après la définition 3.4.2, on a :

$$\begin{array}{rl} & \mathsf{clauses}^{new}(Q_1^2(x^1,x^2), Reste \cup \{P(g(x^1,x^2)) \leftarrow Q_1^2(x^1,x^2)\}) \\ = & \{Q_1^2(a,a) \leftarrow .\ \ Q_1^2(b,b) \leftarrow .\} \end{array}$$

On a alors :

$$\begin{array}{rcl} Prog' & = & \mathsf{uncopy}^{\mathsf{one}}_{\mathsf{Reste}}(P(g(x,x)) \leftarrow Q_1(x)) \\ & = & \{P(g(x^1,x^2)) \leftarrow Q_1^2(x^1,x^2).\ Q_1^2(\ a,a) \leftarrow .\ Q_1^2(b,b) \leftarrow .\} \end{array}$$

En utilisant l'équation (3.1), on obtient :

$$\mathsf{uncopying}(Prog) = \mathsf{uncopying}(Prog') \cup \{P(f(x)) \leftarrow Q_1(x).\ Q_1(a) \leftarrow .\ Q_1(b) \leftarrow .\}$$

De plus, $Prog'$ est un CS-programme non-copiant et normalisé, alors vu la définition 3.4.5, $\mathsf{uncopying}(Prog') = Prog'$. Soit $Prog_f$ l'ensemble de clauses résultant de $\mathsf{uncopying}(Prog)$. On peut noter que $Prog_f$ est un CS-programme normalisé et $Prog_f$ génère le même langage que $Prog$.

Lemme 3.4.7
Si l'algorithme 3.4.5 termine, alors pour toutes les clauses copiantes $P(\overrightarrow{t}) \leftarrow B \in Prog$, $\mathcal{L}_{uncopying(Prog)}(P) = \mathcal{L}_{Prog}(P)$.

Cela vient du fait que Q_i a p arguments, donc $Q_i^{max_i}$ a $max_i \times p$ arguments et :

$$\mathcal{L}(Q_i^{max_i}) = \left\{ \underbrace{\overrightarrow{t} \ldots \overrightarrow{t}}_{max_i \text{ times}} \mid \overrightarrow{t} \in \mathcal{L}(Q_i) \right\}$$

Alors, $\mathcal{L}(Q_i^1) = \mathcal{L}(Q_i)$ et [11] $\mathcal{L}((Q_i^x)^y) = \mathcal{L}(Q_i^{x\times y})$ Ainsi on va utiliser Q_i à la place de Q_i^1, et $(Q_i^x)^y$ à la place de $Q_i^{x\times y}$.

Voici quelques exemples de la technique de complétion présentée dans la section 3.2 en utilisant des systèmes de réécriture non linéaires droits et la fonction uncopying.

Exemple 3.4.8
Soient $R = \{f(x) \to g(x,x),\ a \to b\}$ un système de réécriture non linéaire droit et $Prog_0 = \{P(f(x)) \leftarrow Q_1(x).\ \ Q_1(a) \leftarrow\}$ un CS-programme non copiant. Il y a deux paires critiques, $P(g(x,x)) \leftarrow Q_1(x).$ et $Q_1(b) \leftarrow$. Pour rendre ces paires critiques convergentes, on les ajoute au programme et on a :

$$Prog_1 = Prog_0 \cup \{P(g(x,x)) \leftarrow Q_1(x).\ Q_1(b) \leftarrow\}$$

11. Si la définition 3.4.5 itère beaucoup, des symboles de prédicats de la forme $(Q_i^x)^y$ peuvent apparaître.

Le CS-programme $Prog_1$ contient la clause copiante $P(g(x,x)) \leftarrow Q_1(x)$ et est exactement $Prog$ utilisé dans l'exemple 3.4.6. Donc,

$$\mathsf{uncopying}(Prog_1) = Prog_0 \cup \{Q_1(b) \leftarrow .\ P(g(x^1,x^2)) \leftarrow Q_1^2(x^1,x^2).\ Q_1^2(a,a) \leftarrow .\ Q_1^2(b,b) \leftarrow\}$$

Soit $Prog_2 = \mathsf{uncopying}(Prog_1)$. Maintenant, il y a deux paires critiques non convergentes, $Q_1^2(a,b) \leftarrow$ et $Q_1^2(b,a) \leftarrow$. Si on les ajoute à $Prog_2$, on obtient un CS-programme normalisé, non copiant et toutes les paires critiques sont convergentes. En appliquant le théorème 3.2.30, $Mod(Prog_2)$ est clos par réécriture.

Remarque 3.4.9
Si au moins un prédicat $Q_i^{max_i}$ n'est pas défini et qu'il y a une clause $Q_i(\overrightarrow{t_j}) \leftarrow B_j$ dans $Prog$ telle que $\overrightarrow{t_j}$ n'est pas clos, alors la fonction $\mathsf{uncopy}^{\mathsf{one}}$ va générer de nouvelles clauses copiantes.

Malheureusement, cet algorithme ne termine pas dans le cas général. L'exemple ci-dessous en est une illustration.

Exemple 3.4.10
Soit $Prog = \{P(c(x,x)) \leftarrow P(x).\ P(a) \leftarrow .\}$. $Prog$ est un CS-programme normalisé et copiant. Comme $Clause(1)$ [12] est copiante, on applique uncopying et on ajoute

$$\{P(c(x,x')) \leftarrow P^2(x,x').\ P^2(a,a) \leftarrow .\ P^2(c(x,x'),c(x,x')) \leftarrow P^2(x,x').\}$$

à $Prog$. De même, $Clause(5)$ est copiante ; le même processus est appliqué et les clauses

$$\left\{ \begin{array}{l} P^2(c(x_1,x_1'),\ c(x_2,x_2')) \leftarrow P^4(x_1,x_1',x_2,x_2'). \\ P^4(a,a,a,a) \leftarrow . \\ P^4(c(x_1,x_1'),c(x_2,x_2'),c(x_1,x_1'),c(x_2,x_2')) \leftarrow P^4(x_1,x_1',x_2,x_2'). \end{array} \right\}$$

sont ajoutées à $Prog$. Cependant $Clause(8)$ est copiante. L'algorithme ne termine pas, par conséquent nous n'aurons jamais un programme sans clauses copiantes.

Pour forcer la terminaison tout en se débarrassant des clauses copiantes, on fixe un entier positif $Uncopylimit$. Si on a besoin de générer un prédicat Q^x avec $x > UncopyingLimit$, nous coupons Q^x en $Q^{x_1},\ldots,Q^{x_n}$ avec $\sum_{i\in[1,n]} x_i = x$, ce qui génère une sur-approximation puisque

$$\mathcal{L}(Q^x) = \left\{ \underbrace{\overrightarrow{t} \ldots \overrightarrow{t}}_{x \text{ fois}} \mid \overrightarrow{t} \in \mathcal{L}(Q) \right\} \subseteq \mathcal{L}(Q^{x_1}) \times \ldots \times \mathcal{L}(Q^{x_n})$$

Exemple 3.4.11
Considérons l'exemple 3.4.10 à nouveau et définissons $UncopyingLimit = 4$. Notons que $Clause(8)$ est copiante. Appliquer la procédure sans sur-approximation devrait générer la clause

$$P^4(c(x_1,x_1'),c(x_2,x_2'),c(x_3,x_3'),c(x_4,x_4')) \leftarrow P^8(x_1,x_1',x_2,x_2',x_3,x_3',x_4,x_4')$$

Cependant $UncopyingLimit$ est atteint. On coupe alors P^8 et on obtient

12. Dans nos exemples, $Clause(i)$ dénote la i-ème clause présente dans $Prog$ en respectant l'ordre d'ajout. Par exemple, $Clause(6) = P^2(c(x_1,x_1'),\ c(x_2,x_2')) \leftarrow P^4(x_1,x_1',x_2,x_2')$.

$$P^4(c(x_1,x_1'),c(x_2,x_2'),c(x_3,x_3'),c(x_4,x_4')) \leftarrow P^4(x_1,x_1',x_2,x_2'),P^2(x_3,x_3'),P^2(x_4,x_4').$$

Les prédicats P^4 et P^2 étant déjà définis dans $Prog$, nous n'avons pas besoin d'ajouter de nouvelles clauses supplémentaires. Finalement, le CS-programme uncopying$(Prog)$ inclut les clauses non-copiantes (2), (3), (4), (6), (7) and (9). Rappelons que $\mathcal{L}(P^8)$ est censé être défini tel que :

$$\mathcal{L}(P^8) = \{\underbrace{\overrightarrow{t}\ldots\overrightarrow{t}}_{\text{8 fois}} \mid \overrightarrow{t} \in \mathcal{L}(P)\}$$

On remplace alors le corps de clause $P^8(x_1,x_1',x_2,x_2',x_3,x_3',x_4,x_4')$ par $P^4(x_1,x_1',x_2,x_2')$, $P^2(x_3,x_3'),P^2(x_4,x_4')$. Cela génère l'ensemble

$$\{\underbrace{\overrightarrow{t}\ldots\overrightarrow{t}}_{\text{4 fois}}.\overrightarrow{t'}.\overrightarrow{t'}.\overrightarrow{t''}.\overrightarrow{t''} \mid \overrightarrow{t},\overrightarrow{t'},\overrightarrow{t''} \in L(P)\} \subset L(P^8)$$

ce qui est une sur-approximation. En effet, notons par exemple que

$$P^4(c(a,a),\ c(a,a),\ c(c(a,a),c(a,a)),\ c(a,a))$$

est dans $Mod(\text{uncopying}(Prog))$ mais pas dans $Mod(Prog)$.

Nous donnons maintenant un exemple simple de complétion (Définition [10]) effectuée avec uncopying.

Exemple 3.4.12
Soient $R = \{f(x) \to g(x,x),\ a \to b\}$ un système de réécriture non linéaire droit, et $Prog_0 = \{P(f(x)) \leftarrow Q_1(x).\ Q_1(a) \leftarrow\}$un CS-programme normalisé et non copiant. Il y a deux paires critiques, $P(g(x_1,x_1)) \leftarrow Q_1(x_1).$ et $Q_1(b) \leftarrow$. Pour rendre ces paires critiques convergentes, on les ajoute au programme et on a :

$$Prog_1 = Prog_0 \cup \{P(g(x_1,x_1)) \leftarrow Q_1(x_1).\ Q_1(b) \leftarrow\}$$

Notons que $Prog_1$ contient la clause copiante $P(g(x_1,x_1)) \leftarrow Q_1(x_1)$. Alors, la définition 3.4.5 doit être appliquée à $Prog_1$. A partir de $P(g(x_1,x_1)) \leftarrow Q_1(x_1)$, on obtient la clause $P(g(x_1^1,x_1^2)) \leftarrow Q_1^2(x_1^1,x_1^2)$ en appliquant la définition 3.4.4. En parallèle, on calcule clauses$^{new}(Q_1^2(x_1^1,x_1^2),Prog_1)$. A partir de $Q_1(a) \leftarrow$ et $Q_1(b) \leftarrow$ on a respectivement $Q_1^2(a,a) \leftarrow$ et $Q_1^2(b,b) \leftarrow$ en utilisant la définition 3.4.3. Finalement,

$$\text{uncopying}(Prog_1) = Prog_0 \cup \{Q_1(b) \leftarrow.\ P(g(x_1^1,x_1^2)) \leftarrow Q_1^2(x_1^1,x_1^2) Q_1^2(a,a) \leftarrow.\ Q_1^2(b,b) \leftarrow\}$$

Alors, uncopying$(Prog_1)$ est un CS-programme normalisé et non copiant.

Soit $Prog_2 =$ uncopying$(Prog_1)$. Maintenant, il y a deux paires critiques non copiantes, $Q_1^2(a,b) \leftarrow$ et $Q_1^2(b,a) \leftarrow$. Si on les ajoute à $Prog_2$, on a un CS-programme normalisé et non copiant et toutes les paires critiques sont convergentes. En appliquant le théorème 3.2.30, $Mod(Prog_2)$ est clos par réécriture.

3.5 Conclusion

Dans ce chapitre, nous proposons à l'aide d'une modification mineure de l'algorithme [9, 10] de calculer une sur-approximation de l'ensemble des descendants *innermost* avec un système de réécriture non linéaire droit. Nous proposons également une technique permettant de transformer un CS-programme copiant en un CS-programme non copiant au prix d'une sur-approximation supplémentaire. À l'aide de cette transformation, nous montrons qu'il est possible de calculer une sur-approximation de l'ensemble de descendants (sans stratégie) avec l'algorithme [9, 10] en utilisant des systèmes de réécriture non linéaires droits.

Les travaux [5, 6, 13, 25, 26] autour du calcul de sur-approximations d'ensembles de descendants utilisent des langages réguliers. Bien que des techniques aient été mises au point pour contrôler ou affiner les sur-approximations régulières [8, 27] , la nature régulière des langages utilisés limite la précision de ces approximations [14].

Il existe principalement trois travaux utilisant des langages non réguliers : Boichut et al. [9, 10], Kochems [36] et Boichut et al. [11]. Ces techniques de calculs de sur-approximations de descendants utilisant des langages non réguliers étant très différentes, il est difficile de les comparer entre elles. Nous allons donc comparer dans un premier temps les contraintes imposées sur le système de réécriture puis les classes de langages utilisées.

Les travaux [9, 10] nécessitent un système de réécriture linéaire. Cependant, grâce aux apports présentés dans ce chapitre, il est maintenant possible d'utiliser des systèmes de réécriture non linéaires droits avec cette dernière technique. Les travaux [11] et [36] ont également besoin de systèmes de réécriture linéaires gauches.

Les travaux [36] utilisent les grammaires linéaires indexées d'arbres pour calculer une sur-approximation des descendants. Ces grammaires peuvent reconnaître les langages algébriques. Nos travaux utilisent les CS-programmes qui reconnaissent les langages synchronisés de tuples d'arbres. Aucune de ces deux classes de langages n'est incluse l'une dans l'autre. Il est donc possible que certaines approximations ne pouvant pas être effectuée par une de ces deux techniques puissent l'être par l'autre et inversement Les travaux [11] utilisent des SCF-programmes. La classe des langages reconnus par les SCF-programmes sont les langages synchronisés algébriques de tuples d'arbres, qui inclut la classe des langages synchronisés de tuples d'arbres. Les langages synchronisés algébriques n'incluent cependant pas les langages algébriques, la réciproque étant également vraie. Enfin, les techniques [9, 10] et [11] semblent moins automatiques que la technique [36].

Nous sommes actuellement en train de réaliser une implémentation de ces travaux. Ce prototype sera notamment utile pour développer des heuristiques afin d'améliorer la qualité des approximations et de rendre la méthode totalement automatique.

Le mécanisme removeCycles permet de rendre le nombre de pairs critiques fini au prix d'une sur-approximation. Cette sur-approximation est parfois trop imprécise pour permettre l'analyse d'atteignabilité. Dans certains cas, il est possible de répartir l'ensemble des paires critiques dans deux ensembles disjoints :

— un ensemble pouvant être infini et ne contenant que des paires critiques convergentes
— un ensemble fini de paires critiques.

Dans ce cas, il n'est pas forcément nécessaire d'utiliser le mécanisme removeCycles. Mettre au point un procédé le plus souvent capable de faire ce traitement permettrait d'améliorer sensiblement la qualité des sur-approximations.

Les travaux [11] proposent une technique similaire à la précédente mais avec des langages synchronisés algébriques de tuples d'arbres. Adapter les travaux que nous avons réalisés à cette classe plus large de langages semble intéressant.

De manière plus générale, beaucoup de techniques utilisées pour améliorer ou contrôler la qualité des sur-approximations régulières semblent intéressantes et plus ou moins difficiles à mettre en place pour les techniques utilisant des langages synchronisés. Il paraît notamment intéressant d'essayer de transposer le contrôle de la sur-approximation à l'aide d'équations [8, 27] ou encore la méthode CEGAR.

Dans ce chapitre nous proposons des travaux permettant d'utiliser des systèmes de réécriture non linéaires droits pour le calcul de sur-approximations de descendants avec des CS-programmes. Il peut être également intéressant d'utiliser des systèmes de réécriture non linéaires gauches. Dans le chapitre suivant, nous proposons une méthode permettant de transformer un programme logique quelconque en un CS-programme au prix d'une sur-approximation. Cette technique permet l'utilisation de systèmes de réécriture non linéaires gauches pour le calcul de sur-approximations de descendants avec des CS-programmes.

Chapitre 4

Sur-approximation de programmes logiques en langages synchronisés de tuples d'arbres

Dans la technique de vérification d'atteignabilité, il s'agit à partir d'un langage initial et d'un système de réécriture de calculer l'ensemble des descendants. Une fois l'ensemble des descendants calculé, il faut faire l'intersection entre l'ensemble des états indésirables et cet ensemble. Dans le chapitre 3, nous utilisons des CS-programmes pour représenter l'ensemble des descendants. La classe des langages reconnus par les CS-programmes, les langages synchronisés de tuples d'arbres, est stable par intersection avec les langages réguliers d'arbres. En représentant l'ensemble des états indésirables par des langages réguliers d'arbres, il est donc possible de calculer cette intersection. Cependant, il peut être intéressant de représenter l'ensemble des états indésirables par un langage non régulier comme par exemple un CS-programme. Malheureusement, les langages synchronisés de tuples d'arbres ne sont pas stables par intersection.

De plus, dans le chapitre 3, la technique de complétion nécessite un système de réécriture linéaire gauche. En effet, l'utilisation d'un système non linéaire gauche peut entraîner l'apparition de clauses qui auraient un corps non linéaire et donc qui ne seraient pas des CS-clauses. Les clauses générées n'étant pas des CS-clauses, il n'est pas possible de les ajouter à notre CS-programme ce qui pose problème pour cette technique de complétion. L'exemple ci-dessous illustre ce phénomène.

Exemple (Impact de la non linéarité gauche sur la technique de complétion du chapitre 3)
Soit l'alphabet $\Sigma = \{f^{\setminus 2}, g^{\setminus 1}\}$. *Soient le* CS*-programme* $Prog = \{P(f(x,y)) \leftarrow Q_1(x), Q_2(y).\}$ *et le système de réécriture non linéaire gauche* $R = \{f(z,z) \rightarrow g(z)\}$.

On a $f(x,y) = f(z,z)$ *avec la substitution* $\sigma = [x/z, y/z]$ *et* $P(f(z,z)) \rightsquigarrow Q_1(z), Q_2(z)$. *Vu que* $Q_1(z), Q_2(z)$ *est un ensemble d'atomes plats et que* $f(x,y)$ *n'est pas une variable, d'après la définition 3.2.1* [1]*, on a la paire critique suivante* $P(g(z)) \leftarrow Q_1(z), Q_2(z)$.

Cependant, $Q_1(z), Q_2(z)$ *n'est pas un ensemble d'atomes linéaire et par conséquent,* $P(g(z)) \leftarrow Q_1(z), Q_2(z)$. *n'est pas une* CS*-clause.*

Dans ce chapitre nous proposons un algorithme qui au prix d'une sur-approximation,

1. En omettant la précondition d'avoir un système de réécriture linéaire gauche, que nous omettons ici.

transforme un programme logique quelconque en un CS-programme. À l'aide de cette technique, nous pouvons calculer une sur-approximation de l'intersection entre l'ensemble des descendants et l'ensemble des états indésirables qui seraient tous les deux représentés par des CS-programmes ou encore, nous pouvons utiliser des systèmes de réécriture non linéaires gauches au prix d'une sur-approximation supplémentaire dans la technique de complétion présentée dans le chapitre 3.

Ce chapitre commence par introduire les travaux [41, 42] sur lesquels notre apport est basé (section 4.1). La section 4.2 présente une nouvelle règle d'inférence permettant de rendre l'algorithme terminant au prix d'une sur-approximation. Avant de conclure ce chapitre, la section 4.3 donne un exemple utilisant la technique de transformation de programmes logiques présentée dans ce chapitre pour effectuer une analyse d'accessibilité.

4.1 Préliminaires : Semi-algorithme de transformation de programmes logiques quelconques en CS-programmes

Dans [41, 42], Limet et Salzer ont proposé une technique permettant de transformer un programme logique quelconque en un CS-programme équivalent. Toutefois, cette technique est un semi-algorithme, c'est à dire qu'il peut ne pas terminer. Dans cette section, nous présentons cette technique qui est composée de deux règles d'inférences appelées Unfolding et Definition introduction.

Cette technique utilise la notion de définition logique qui représente une équivalence entre un atome et une séquence d'atomes.

Définition 4.1.1 (Définition logique)
*Une **définition logique** est un couple de la forme $P(x_1, \ldots, x_n) \triangleq B$, où P est un symbole de prédicat, B est une séquence d'atomes, et $\{x_1, \ldots, x_n\} \subseteq Var(B)$.*

On considère $P(x_1, \ldots, x_n) \triangleq B$ comme une équivalence. Cependant, $\triangleq$ n'est pas commutatif, et B est appelée le corps de la définition. Dans [41, 42], un état du semi-algorithme est un tuple $\langle Prog, D_{new}, D_{done}, C_{new}, C_{out} \rangle$ où :

- $Prog$ est un ensemble de clauses de Horn,
- D_{new} et D_{done} sont des ensembles de définitions,
- C_{new} est un ensemble de clauses de Horn,
- C_{out} est un ensemble de CS-clauses (considéré comme le CS-programme final une fois le processus terminé).

Ce semi-algorithme utilise deux types de transitions différentes. Nous définissons les opérateurs qui à partir d'un état et d'un type de transition donnent un nouvel état.

Définition 4.1.2
Soient S et S' deux tuples de la forme $\langle Prog, D_{new}, D_{done}, C_{new}, C_{out} \rangle$ représentant des états du semi-algorithme.

- *$S \Rightarrow^U S'$ signifie que l'état S' est issu de l'état S en utilisant la règle* Unfolding *(Définition 4.1.4).*

— $S \Rightarrow^I S'$ *signifie que l'état* S' *est issu de l'état* S *en utilisant la règle* Definition introduction *(Définition 4.1.10).*
— $S \Rightarrow S'$ *signifie que l'état* S' *est issu de l'état* S *en utilisant* $S \Rightarrow^I S'$ *ou* $S \Rightarrow^U S'$.

Notation 4.1.3
La relation $\Rightarrow^*$ *représente la clôture réflexive, transitive de* $\Rightarrow$.

Un état initial est de la forme $\langle Prog, D_{new}, \emptyset, \emptyset, \emptyset\rangle$. Un état final est de la forme $\langle Prog, \emptyset, D_{done}, \emptyset, C_{out}\rangle$. Par la suite, $\uplus$ représente l'union disjointe, ce qui signifie $A \uplus B = A \cup B$ et $A \cap B = \emptyset$.

Le premier type de transition, Unfolding génère un ensemble de clauses dans C_{new} à partir d'une définition issue de D_{new} en utilisant les clauses présentes dans $Prog$.

Définition 4.1.4 (Unfolding [42])
La transition Unfolding *prend une définition dans* D_{new}*, sélectionne une partie des atomes de son corps d'après une règle de sélection, et les déplie avec toutes les clauses de* $Prog$ *compatibles.*
Plus formellement :

$$\frac{\langle Prog, D_{new} \uplus \{L \triangleq R \uplus \{A_1, \ldots, A_n\}\}, D_{done}, C_{new}, C_{out}\rangle}{\langle Prog, D_{new}, D_{done} \cup \{L \triangleq R \uplus \{A_1, \ldots, A_n\}\}, C_{new} \cup C, C_{out}\rangle}$$

où C *est l'ensemble de toutes les clauses* $\mu(L \leftarrow R \cup B_1 \cup \ldots \cup B_n)$ *tel que chaque* $H_i \leftarrow B_i$ *est une clause de* $Prog$ *pour* $i = 1, \ldots, n$ *et tel que l'unificateur le plus général* μ *de* $(A_1, \ldots, A_n)$ *et* $(H_1, \ldots, H_n)$ *existe.*

Ci-dessous, nous donnons un exemple qui montre le fonctionnement de Unfolding.

Exemple 4.1.5 (Illustration de Unfolding)

$$\langle \{Q(s(x), s(y)) \leftarrow Q(x, y)\}, \{P(z) \triangleq Q(z, z)\}, \emptyset, \emptyset, \emptyset\rangle$$
$$\Rightarrow^U \langle \{Q(s(x), s(y)) \leftarrow Q(x, y)\}, \emptyset, \{P(z) \triangleq Q(z, z)\}, \{P(s(y)) \leftarrow Q(y, y)\}, \emptyset\rangle$$

Pour résumer, la règle Unfolding génère de nouvelles clauses. Celles-ci sont alors utilisées par la règle Definition introduction pour enrichir le programme de sortie C_{out}. La règle Definition introduction peut aussi bien ajouter de nouveaux symboles de prédicats que réutiliser ceux déjà existants.

Nous allons maintenant présenter la règle Definition introduction. L'objectif est de transformer une clause de Horn quelconque de C_{new} en une CS-clause, en remplaçant son corps par un corps plat et linéaire. Pour ce faire, des définitions logiques sont générées ou réutilisées. Avant d'introduire la règle d'inférence Definition introduction, des préliminaires sont nécessaires.

Afin de pouvoir réutiliser une définition $L \triangleq R$, il est nécessaire d'introduire la notion de correspondance entre $L \triangleq R$ et un ensemble d'atomes.

Définition 4.1.6 ([42])
Soient $H \leftarrow B$ une clause et B' un sous-ensemble de B tel que $Var(B') \cap Var(B \setminus B') = \emptyset$[2]*.*
Une définition $L \triangleq R$ ***correspond*** *à B' s'il existe un renommage de variables ρ pour R tel que $\rho(R) = B'$ et $Var(B') \setminus Var(H) = Var(\rho(R)) \setminus Var(\rho(L))$.*
La définition correspond aux atomes du corps via ρ.

Exemple 4.1.7
Dans l'état $\langle \{Q(s(x), s(y)) \leftarrow Q(x, y)\}, \emptyset, \{P(z) \triangleq Q(z, z)\}, \{P(s(y)) \leftarrow Q(y, y)\}, \emptyset\rangle$, la définition $P(z) \triangleq Q(z, z)$ correspond à l'atome $Q(y, y)$ via $\rho = [z/y]$.

Pour qu'une clause logique quelconque soit une CS-clause, il est nécessaire que le corps de celle-ci soit linéaire. Les prédicats pouvant avoir une arité supérieure à un, la décomposition d'un ensemble d'atomes en plusieurs ensembles d'atomes linéaires n'est pas triviale. En effet, il est possible que deux atomes ne partageant pas de variables se retrouvent dans le même ensemble à cause d'un troisième atome partageant des variables avec ces deux derniers. Nous donnons ci-dessous, une définition formelle de la décomposition d'un ensemble d'atomes en plusieurs ensembles d'atomes linéaires.

Définition 4.1.8 (Décomposition et Chain)
On définit $\leftrightsquigarrow$ comme la plus petite relation d'équivalence entre les atomes telle que pour tous atomes A, A', $Var(A) \cap Var(A') \neq \emptyset \implies A \leftrightsquigarrow A'$.
Une classe d'équivalence de $\leftrightsquigarrow$ est appelée une Chain.
Pour un ensemble d'atomes, une clause ou une définition X, $Chain(X)$ représente l'ensemble des Chain contenues respectivement dans l'ensemble d'atomes, dans le corps de la clause ou de la définition X.

Exemple 4.1.9
$Chain(P(x, y, v) \leftarrow P(x, z), Q(y), P(x, v), Q(z)) = \big\{\{P(x, z), P(x, v), Q(z)\}, \{Q(y)\}\big\}$.

Définition 4.1.10 (Definition introduction [42])
La transition Definition introduction *prend une clause de C_{new} et la transforme en* CS*-clause en créant de nouvelles définitions logiques qui seront ajoutées à D_{new}. La* CS*-clause générée est ajoutée à C_{out}.*

$$\frac{\langle Prog, D_{new}, D_{done}, C_{new} \uplus \{H \leftarrow B_1 \uplus \ldots \uplus B_n\}, C_{out}\rangle}{\langle Prog, D_{new} \cup D, D_{done}, C_{new}, C_{out} \cup \{H \leftarrow L_1, \ldots, L_n\}\rangle}$$

tel que $Chain(B_1 \uplus \ldots \uplus B_n) = \{B_1, \ldots, B_n\}$, et pour chaque $i \in \{1, \ldots, n\}$

$$L_i = \begin{cases} \rho(L) & \text{si } D_{done} \text{ contient une définition } L \triangleq R \text{ correspondant à } B_i \text{ via } \rho \\ P_i(x_1, \ldots, x_k) & \text{sinon, où } P_i \text{ est un nouveau symbole de prédicat et} \\ & \{x_1, \ldots, x_k\} = Var(B_i) \cap Var(H). \end{cases}$$

et D est l'ensemble de toutes les nouvelles définitions $P_i(x_1, \ldots, x_k) \triangleq B_i$.

Exemple 4.1.11
Prenons à nouveau l'exemple 4.1.5. En utilisant les règles Unfolding et Definition introduction, on a :
$\langle \{Q(s(x), s(y)) \leftarrow Q(x, y)\}, \{P(z) \triangleq Q(z, z)\}, \emptyset, \emptyset, \emptyset\rangle$
$\Rightarrow^U \langle \{Q(s(x), s(y)) \leftarrow Q(x, y)\}, \emptyset, \{P(z) \triangleq Q(z, z)\}, \{P(s(y)) \leftarrow Q(y, y)\}, \emptyset\rangle$
$\Rightarrow^I \langle \{Q(s(x), s(y)) \leftarrow Q(x, y)\}, \emptyset, \{P(z) \triangleq Q(z, z)\}, \emptyset, \{P(s(y)) \leftarrow P(y)\}\rangle$.
On remarque que $C_{out} = \{P(s(y)) \leftarrow P(y)\}$ est un CS-programme.

2. C'est à dire que B' ne partage pas de variable avec le reste du corps de la clause.

4.2 Généralisation de clauses

Dans la définition 4.2.1, on introduit la technique de ***Généralisation*** inspirée de [45]. De manière informelle, la généralisation transforme une clause en une autre dont le plus petit modèle de Herbrand contient celui de départ.

Nous commençons par donner la définition de stratégie de généralisation.

Définition 4.2.1 (Stratégie de généralisation)
*Une **stratégie de généralisation** (notée* Gen*) est une fonction définie sur un ensemble de clauses tels que pour chaque clause* $H \leftarrow B$ *de l'ensemble de départ, il existe une substitution* σ *telle que* $\sigma(\mathsf{Gen}(H \leftarrow B)) = (H \leftarrow B)$.

Exemple 4.2.2
Nous avons $P(f(x)) \leftarrow Q(g(x))$ peut être généralisée en $P(f(x)) \leftarrow Q(y)$.

En utilisant une stratégie de généralisation sur une clause d'un programme logique, on obtient un nouveau programme logique qui a un modèle de Herbrand contenant a minima celui du programme logique initial.

Lemme 4.2.3
Soient $Prog$ *un programme logique,* $H \leftarrow B$ *une clause et* G *une séquence d'atomes. Soient* $Prog_1 = Prog \cup \{H \leftarrow B\}$ *et* $Prog_2 = Prog \cup \{\mathsf{Gen}(H \leftarrow B)\}$.
La dérivation $G \rightsquigarrow^*_{Prog_1} \emptyset$ *implique* $G \rightsquigarrow^*_{Prog_2} \emptyset$.

Démonstration. Par induction sur la longueur n de la dérivation $G \rightsquigarrow^*_{Prog_1} \emptyset$.

- $n = 0$: alors $G = \emptyset$, donc $G \rightsquigarrow^*_{Prog_2} \emptyset$.
- $n \geq 1$: si $G \rightsquigarrow^*_{Prog_1} \emptyset$ n'utilise pas la clause $H \leftarrow B$, alors $G \rightsquigarrow^*_{Prog} \emptyset$ et par conséquent $G \rightsquigarrow^*_{Prog_2} \emptyset$.
  Sinon, la dérivation s'écrit de la manière suivante : $G \rightsquigarrow^*_{Prog} G_1 \rightsquigarrow_{H \leftarrow B} G_2 \rightsquigarrow^*_{Prog_1} \emptyset$ avec la longueur de $G_2 \rightsquigarrow^*_{Prog_1} \emptyset$ strictement plus petite que n. Par hypothèse d'induction, on a $G_2 \rightsquigarrow^*_{Prog_2} \emptyset$.
  Inversement, puisque $G_1 \rightsquigarrow_{H \leftarrow B} G_2$, on a $G_1 \rightsquigarrow_{\mathsf{Gen}(H \leftarrow B)} G'_2$ et il existe une substitution σ tel que $\sigma(G'_2) = G_2$. De plus, puisque $G_2 \rightsquigarrow^*_{Prog_2} \emptyset$, on a $G'_2 \rightsquigarrow^*_{Prog_2} \emptyset$. Finalement, $G \rightsquigarrow^*_{Prog} G_1 \rightsquigarrow_{\mathsf{Gen}(H \leftarrow B)} G'_2 \rightsquigarrow^*_{Prog_2} \emptyset$, c'est-à-dire $G \rightsquigarrow^*_{Prog_2} \emptyset$.

□

Corollaire 4.2.4
Nous avons $Mod(Prog \cup \{H \leftarrow B\}) \subseteq Mod(Prog \cup \{\mathsf{Gen}(H \leftarrow B)\})$.

Ici, nous voulons que le plus petit modèle de Herbrand du programme généré contienne (et pas forcément préserve) le plus petit modèle de Herbrand du programme logique initial. Alors, on peut remplacer une clause par une autre plus générale.

Dans [45], Pettorossi et Proietti veulent préserver exactement le plus petit modèle de Herbrand. Pour une clause de la forme $H \leftarrow A_1, \ldots, A_n, B_1, \ldots, B_m$, ils définissent un ensemble d'atomes $A'_1, \ldots, A'_n$ qui généralise $A_1, \ldots, A_n$. En d'autres termes, il existe une

substitution σ telle que $\sigma(A'_1, \ldots, A'_n) = A_1, \ldots, A_n$. Pettorossi et Proietti définissent alors une nouvelle clause $GenP(x_1, \ldots, x_k) \leftarrow A'_1, \ldots, A'_n$ où $Var(A'_1, \ldots, A'_n) = \{x_1, \ldots, x_k\}$, et remplacent la clause initiale par : $H \leftarrow \sigma(GenP(x_1, \ldots, x_k)), B_1, \ldots, B_m$. Donc le plus petit modèle de Herbrand est préservé, mais ceci n'aide pas à faire terminer [42].

4.2.1 Intégration de la généralisation à [42]

Soit Gen une stratégie de généralisation. On introduit une nouvelle règle d'inférence appelée Generalization. Pour un état donné $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out} \rangle$, l'idée est de transformer une clause de C_{new} en une autre clause plus générale.

Définition 4.2.5 (Generalization)
La transition Generalization *prend une clause de C_{new} et la transforme en une autre clause plus générale.*

$$\frac{\langle Prog, D_{new}, D_{done}, C_{new} \uplus \{H \leftarrow B\}, C_{out} \rangle}{\langle Prog, D_{new}, D_{done}, C_{new} \cup \{\mathsf{Gen}(H \leftarrow B)\}, C_{out} \rangle}$$

Notation 4.2.6
Comme pour les règles Unfolding *et* Definition introduction*, $S \Rightarrow^G S'$ signifie que S' est généré depuis S en utilisant la règle* Generalization*. Par conséquent, on étend la notation $\Rightarrow$ de la manière suivante : $S \Rightarrow S'$ si et seulement si $S \Rightarrow^I S'$ ou $S \Rightarrow^U S'$ ou $S \Rightarrow^G S'$.*

4.2.2 Résultats

Dans cette section, nous montrons dans le théorème 4.2.11 qu'en utilisant le processus de généralisation, nous sommes capables de calculer un CS-programme dont le plus petit modèle de Herbrand contient celui du programme initial. Nous montrons également avec le théorème 4.2.12 que le processus de généralisation permet de garantir que [42] termine toujours. Nous supposons que $Prog$ et Σ sont des ensembles finis.

Pour prouver le théorème 4.2.11, nous avons d'abord besoin de formaliser la sémantique d'un état quelconque qui sera préservée par les règles d'inférence.

Notation 4.2.7
Soient un programme logique $Prog$, un ensemble de clauses C et un ensemble de définitions D. On introduit l'opérateur $\hat{D} = \{(H \leftarrow B) \mid H \triangleq B \in D\}$. Remarquons que $\hat{D}$ est un ensemble de clauses.
Pour un état $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out} \rangle$, on définit $\hat{S} = Prog \cup \hat{D}_{new} \cup C_{new} \cup C_{out}$. Notons que l'ensemble $\hat{S}$ ne contient pas D_{done}.

Définition 4.2.8 (Cohérence d'état)
L'état $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$ *est dit cohérent si :*

- $Pred(C_{out}) \cap Pred(Prog) = \emptyset$,
- *pour tout symbole de prédicat* P *apparaissant dans* $Head(D_{done})$, $\mathcal{L}_{\hat{D}_{done}\cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}}(P)$ *(c'est à dire* D_{done} *est redondant),*
- *tout symbole de prédicat apparaissant dans* $Head(D_{new} \cup D_{done})$ *n'apparait qu'une fois dans* $D_{new} \cup D_{done} \cup Prog \cup Body(C_{new})$.

Lemme 4.2.9
Soient S *et* S' *deux tuples. Si* S *est cohérent et* $S \Rightarrow S'$*, alors* S' *est cohérent.*

Démonstration. Pour les règles Unfolding et Definition introduction, les éléments de preuves sont présentés dans [45]. La règle de Generalization ne fait que remplacer une clause $H \leftarrow B$ de C_{new} par $\mathsf{Gen}(H \leftarrow B)$, ce qui ne change pas les symboles de prédicats présents dans la clause. Maintenant, posons $S' = \langle Prog, D_{new}, D_{done}, C'_{new}, C_{out}\rangle$ et soit P un symbole de prédicat apparaissant dans $Head(D_{done})$. On a alors $\mathcal{L}_{\hat{D}_{done}\cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}}(P)$ car S est cohérent, et $\mathcal{L}_{\hat{S}}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$ grâce au corollaire 4.2.4. Donc $\mathcal{L}_{\hat{D}_{done}\cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$. □

Lemme 4.2.10
Soit $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$ *un état cohérent. Supposons que* $S \Rightarrow S'$ *et notons* $S' = \langle Prog, D'_{new}, D'_{done}, C'_{new}, C'_{out}\rangle$*. Alors chaque symbole de prédicat* P *apparaissant dans* $Head(D_{new} \cup D_{done})$ *apparaît aussi dans* $Head(D'_{new} \cup D'_{done})$*, et* $\mathcal{L}_{\hat{S}}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$.

Démonstration. Pour les règles Unfolding et Definition introduction, les éléments de preuve sont présentés dans [45], et pour la règle Generalization cela vient du corollaire 4.2.4. □

Théorème 4.2.11 (Extension de [42])**.** *Soient une stratégie de généralisation* Gen *et un état initial* $S^0 = \langle Prog, D^0_{new}, \emptyset, \emptyset, \emptyset\rangle$ *tels que chaque symbole de prédicat apparaissant dans* $Head(D^0_{new})$ *apparaît une seule fois dans* $D^0_{new} \cup Prog$*. S'il existe un état* S^n *tel que* $S^0 \Rightarrow^* S^n$ *et* $S^n = \langle Prog, \emptyset, D^n_{done}, \emptyset, C^n_{out}\rangle$*, alors pour chaque définition* $P(\overrightarrow{t}) \triangleq B \in D^0_{new}$, $\mathcal{L}_{Prog\cup\hat{D}^0_{new}}(P) \subseteq \mathcal{L}_{C^n_{out}}(P)$.

Démonstration. Puisque S^0 est cohérent et vu le lemme 4.2.9, S^n est aussi cohérent. D'après le lemme 4.2.10, $\mathcal{L}_{Prog\cup\hat{D}^0_{new}}(P) = \mathcal{L}_{S^0}(P) \subseteq \mathcal{L}_{S^n}(P)$. De plus, vu que S^n est cohérent, $\mathcal{L}_{S^n}(P) = \mathcal{L}_{C^n_{out}}(P)$. Par conséquent, $\mathcal{L}_{Prog\cup\hat{D}^0_{new}}(P) \subseteq \mathcal{L}_{C^n_{out}}(P)$. □

Montrons comment le processus de généralisation permet de garantir la terminaison de l'algorithme de [42].

Théorème 4.2.12 (Terminaison)**.** *Pour tout programme* $Prog$ *et pour tout état cohérent* $S^0 = \langle Prog, D^0_{new}, \emptyset, \emptyset, \emptyset\rangle$*, il existe toujours une stratégie de généralisation* Gen *et un état* S^n *tel que :*

$$S^0 \Rightarrow^* S^n \text{ et } S^n = \langle Prog, \emptyset, D^n_{done}, \emptyset, C^n_{out}\rangle$$

Démonstration. Soit un état cohérent arbitraire $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$. Appliquer une fois la règle Unfolding à S réduit strictement la taille de l'ensemble D_{new}. Alors, le nombre de fois où il est possible d'appliquer successivement cette règle à S est fini. Appliquer une fois la règle Definition introduction à S réduit strictement la taille de l'ensemble C_{new}. Ainsi, le nombre de fois où il est possible d'appliquer successivement cette règle à S est fini.

Donc, une dérivation infinie utilisant ces règles va obligatoirement contenir un nombre infini d'étapes avec la règle Unfolding (et également avec la règle Definition introduction), ce qui augmente la taille de D_{done} (les têtes des définitions sont deux à deux différentes puisque les états sont cohérents). On en conclut qu'une dérivation infinie rend D_{done} infini.

Soit Gen une stratégie de généralisation telle que pour chaque clause

$$H \leftarrow A_1(t_1^1, \ldots, t_{n_1}^1), \ldots, A_m(t_1^m, \ldots, t_{n_m}^m)$$

on ait

$$\mathsf{Gen}(H \leftarrow A_1(t_1^1, \ldots, t_{n_1}^1), \ldots, A_m(t_1^m, \ldots, t_{n_m}^m)) = (H \leftarrow A_1(x_1^1, \ldots, x_{n_1}^1), \ldots, A_m(x_1^m, \ldots, x_{n_m}^m))$$

où $x_1^1, \ldots, x_{n_m}^m$ sont de nouvelles variables n'apparaissant pas dans $Var(H) \cup (\cup_{i=1}^m Var(A_i(t_1^i, \ldots, t_{n_i}^i)))$.

Admettons qu'une étape de généralisation est appliquée dès qu'une clause est ajoutée dans C_{new}. Alors chaque clause de C_{new} est de la forme $H \leftarrow A_1(x_1^1, \ldots, x_{n_1}^1), \ldots, A_m(x_1^m, \ldots, x_{n_m}^m)$, et on remarque que :

$$Chain(A_1(x_1^1, \ldots, x_{n_1}^1), \ldots, A_m(x_1^m, \ldots, x_{n_m}^m)) = \{\{A_1(x_1^1, \ldots, x_{n_1}^1)\}, \ldots, \{A_m(x_1^m, \ldots, x_{n_m}^m)\}\}$$

En effet, le fait que les variables soient nouvelles garantit que chaque variable introduite par Gen n'apparaisse qu'une seule fois.

Par conséquent, durant une dérivation partant de S^0, chaque définition ajoutée dans D_{new} et donc dans D_{done} est de la forme $P_{new}^i \triangleq A_i(x_1^1, \ldots, x_{n_i}^i)$ et $A_i \in Pred(Prog)$. On peut alors en déduire qu'en utilisant une telle stratégie, la taille de D_{done} est bornée par $|D_{new}^0| + |Pred(Prog)|$. On en conclut que la dérivation issue de S^0 termine et qu'il existe un état $S^n = \langle Prog, \emptyset, D_{done}^n, \emptyset, C_{out}^n \rangle$ tel que $S^0 \Rightarrow^* S^n$. □

4.3 Exemple

Dans cette section, nous nous attaquons au problème d'atteignabilité décrit dans [14], qui ne peut être traité avec succès en utilisant une approximation régulière. On veut montrer que $\langle 0, 0, 0 \rangle \notin \mathcal{L}_{Prog}(P)$ où

$$Prog = \left\{ \begin{array}{ll} P(s(x), s(y), z) \leftarrow P(x, y, z). & P(x, y, z) \leftarrow P(s(x), s(y), z). \\ P(0, s(x), 0). & P(s(x), 0, 0). \end{array} \right\}$$

Pour faire simple, cela revient à créer un nouveau programme logique $Prog'$ tel que $Prog' = Prog \cup \{G \leftarrow P(0, 0, 0)\}$ et de vérifier que $\mathcal{L}_{Prog'}(G) = \emptyset$. Si $\mathcal{L}_{Prog'}(G) = \emptyset$ alors $\langle 0, 0, 0 \rangle \notin \mathcal{L}_{Prog}(P)$.

Définissons D_{new}^0 tel que $D_{new}^0 = \{G \triangleq P(0, 0, 0)\}$. On utilisera comme état initial cohérent $S^0 = \langle Prog, \{G \triangleq P(0, 0, 0)\}, \emptyset, \emptyset, \emptyset \rangle$. Sans utiliser la règle Generalization, on obtient (on utilise le caractère point pour séparer les clauses et les définitions) :

$$\begin{array}{lll} \langle Prog, \{G \triangleq P(0,0,0)\}, \emptyset, \emptyset, \emptyset \rangle & \Rightarrow^U & \langle Prog, \emptyset, D_{done}^1, \{G \leftarrow P(s(0), s(0), 0)\}, \emptyset \rangle \\ & \Rightarrow^I & \langle Prog, \{P_{new_1} \triangleq P(s(0), s(0), 0)\}, D_{done}^2, \emptyset, \{G \leftarrow P_{new_1}\} \rangle \\ & \Rightarrow^U & \langle Prog, \emptyset, D_{done}^3, \{P_{new_1} \leftarrow P(s^2(0), s^2(0), 0).\ P_{new_1} \leftarrow P(0, 0, 0)\}, C_{out}^3 \rangle \\ & \Rightarrow^I & \langle Prog, \{P_{new_2} \triangleq P(s^2(0), s^2(0), 0)\}, D_{done}^4, \emptyset, C_{out}^4 \rangle \\ & \Rightarrow^* & \ldots \end{array}$$

où :

$$
\begin{array}{l}
D^1_{done} = \{G \triangleq P(0,0,0)\} \\
D^2_{done} = \{G \triangleq P(0,0,0)\} \\
D^3_{done} = \{G \triangleq P(0,0,0).\ P_{new_1} \triangleq P(s(0),s(0),0)\} \\
C^3_{out} = \{G \leftarrow P_{new_1}\} \\
D^4_{done} = \{G \triangleq P(0,0,0).\ P_{new_1} \triangleq P(s(0),s(0),0)\} \\
C^4_{out} = \{G \leftarrow P_{new_1}.\ P_{new_1} \leftarrow P_{new_2}.\ P_{new_1} \leftarrow G\}
\end{array}
$$

Le processus (sans utiliser la règle Generalization) ne termine pas car chaque étape $\Rightarrow^I$ engendre une nouvelle définition de la forme $P_{new_i} \triangleq P(s^i(0), s^i(0), 0)$.

On utilise maintenant la stratégie de généralisation Gen définie par :
— Pour chaque clause $H \leftarrow P(s(0),s(0),0)$, $\mathsf{Gen}(H \leftarrow P(s(0),s(0),0)) = H \leftarrow P(x,x,z)$.
— Pour chaque clause de la forme $H \leftarrow P(s(x),s(x),z)$, $\mathsf{Gen}(H \leftarrow P(s(x),s(x),z)) = H \leftarrow P(y,y,z)$.
Si on applique la règle Generalization après chaque Unfolding, on a :

$$
\begin{array}{lll}
\langle Prog, \{G \triangleq P(0,0,0)\}, \emptyset, \emptyset, \emptyset\rangle & \Rightarrow^U & \langle Prog, \emptyset, D^1_{done}, \{G \leftarrow P(s(0),s(0),0)\}, \emptyset\rangle \\
& \Rightarrow^G & \langle Prog, \emptyset, D^1_{done}, \{G \leftarrow P(x,x,z)\}, \emptyset\rangle \\
& \Rightarrow^I & \langle Prog, \{P_{new_1} \triangleq P(x,x,z)\}, D^2_{done}, \emptyset, \{G \leftarrow P_{new_1}\}\rangle \\
& \Rightarrow^U & \langle Prog, \emptyset, D^3_{done}, \{P_{new_1} \leftarrow P(s(x),s(x),z).\ P_{new_1} \leftarrow P(x,x,z)\}, \\
& & \{G \leftarrow P_{new_1}\}\rangle \\
& \Rightarrow^G & \langle Prog, \emptyset, D^3_{done}, \{P_{new_1} \leftarrow P(y,y,z).\ P_{new_1} \leftarrow P(x,x,z)\}, \\
& & \{G \leftarrow P_{new_1}\}\rangle \\
& \Rightarrow^I & \langle Prog, \emptyset, D^3_{done}, \{P_{new_1} \leftarrow P(x,x,z)\}, \\
& & \{P_{new_1} \leftarrow P_{new_1}.\ G \leftarrow P_{new_1}\}\rangle \\
& \Rightarrow^I & \langle Prog, \emptyset, D^3_{done}, \emptyset, \{P_{new_1} \leftarrow P_{new_1}.\ G \leftarrow P_{new_1}\}\rangle
\end{array}
$$

où :

$$
\begin{array}{l}
D^1_{done} = \{G \triangleq P(0,0,0)\} \\
D^2_{done} = \{G \triangleq P(0,0,0)\} \\
D^3_{done} = \{G \triangleq P(0,0,0).\ P_{new_1} \triangleq P(x,x,z)\}
\end{array}
$$

car la clause $(P_{new_1} \leftarrow P_{new_1})$ générée par cette étape est déjà présente dans l'ensemble C_{out}.

La dérivation s'arrête avec $C_{out} = \{G \leftarrow P_{new_1}\}$. On remarque que $\mathcal{L}_{C_{out}}(G) = \emptyset$. Donc grâce au théorème 4.2.11, on peut en déduire que $\mathcal{L}_{Prog \cup \{G \leftarrow P(0,0,0)\}}(G) = \emptyset$.

4.4 Conclusion

Dans ce chapitre, nous avons présenté une technique permettant de transformer un programme logique quelconque en un CS-programme possédant un plus petit modèle de Herbrand qui inclut celui du programme logique d'origine. Nous nous sommes basés sur les travaux de Limet et Salzer [41, 42], qui proposent de transformer un programme logique quelconque en un CS-programme équivalent à l'aide d'une semi-algorithme, c'est-à-dire d'un algorithme non terminant. Nous proposons ainsi, à l'aide d'une règle d'inférence supplémentaire Generalization, de rendre cet algorithme terminant au prix d'une sur-approximation.

Dans ce chapitre, nous définissions également la notion de cohérence entre les étapes de transformation. Cette notion nous permet de prouver que le CS-programme obtenu possède bien un plus petit modèle de Herbrand qui inclut celui du programme logique d'origine. La règle d'inférence Generalization induit une sur-approximation pouvant ne pas être assez fine. Il paraît intéressant de développer des règles d'inférence supplémentaires permettant une sur-approximation de meilleure qualité. L'utilisation d'unification de langages semble ouvrir des possibilités intéressantes.

Chapitre 5

Transformation de systèmes de réécriture avec stratégie

Dans ce chapitre nous proposons la transformation de systèmes de réécriture avec stratégie *prefix-constrained* en systèmes de réécriture sans stratégie. La plupart des travaux réalisés autour de la réécriture, comme les résultats autour des preuves de terminaison, nécessitent des systèmes de réécriture sans stratégie [22, 30, 37, 48, 58]. Transformer un système de réécriture avec stratégie en un système de réécriture sans stratégie équivalent devient donc intéressant. Les travaux présentés dans ce chapitre ont fait l'objet de la publication [2].

Les travaux [18] présentent une technique de transformation des systèmes de réécriture avec des stratégies programmables en système de réécriture sans stratégie. Si on s'interdit de modifier le système de réécriture, les stratégies programmables (définition 2.2.27) sont des stratégies qui utilisent uniquement les positions dans les termes pour restreindre la réécriture. La stratégie de réécriture *prefix-constrained* utilise les positions ainsi que les symboles présents dans les termes.

Les travaux [29] proposent une technique de transformation des systèmes de réécriture *context-sensitive* (définition 2.2.34) en systèmes de réécriture sans stratégie. La stratégie *context-sensitive*, comme la stratégie *prefix-constrained* utilise les positions ainsi que les symboles présents dans les termes pour restreindre la réécriture. Cependant, la stratégie *context-sensitive* est un cas particulier de la stratégie *prefix-constrained*. De plus, la technique présentée dans [29] génère un nombre de règles plus important que notre technique pour transformer le système de réécriture en système sans stratégie. Dans la section 5.2, nous montrons en détail comment notre technique améliore les travaux [29].

Dans ce chapitre, nous commençons (section 5.1) par présenter notre technique de transformation d'un système de réécriture *prefix-constrained* (pCTRS) en un TRS sans stratégie. La section 5.1.1 propose une preuve de correction de cette transformation. La section 5.1.2 donne la preuve de la conservation de la terminaison entre le TRS d'origine et le TRS transformé. Avant de conclure ce chapitre, la section 5.2 montre que notre technique améliore une technique existante [29] de transformation des systèmes de réécriture *context-sensitive* (CSTRS).

5.1 Technique de transformation d'un système de réécriture *prefix-constrained* en un TRS sans stratégie

Pour un système de réécriture *prefix-constrained* $R = \{L_k : l_k \to r_k \mid 1 \leq k \leq n\}$ (Définition 2.2.38), on considère que chaque langage $L_k \subseteq Dir(\Sigma)^*$ est défini par un automate de mots $\mathcal{A}^k = (\Sigma, Q^k, Q^k_I, Q^k_f, \Delta^k)$ tel que :

$$\forall k, k' \in \{1, \ldots, n\},\ (k \neq k' \Longrightarrow Q^k \cap Q^{k'} = \emptyset)$$

Par conséquent, si $k \neq k'$ alors $Q^k_I \cap Q^{k'}_I = Q^k_f \cap Q^{k'}_f = \Delta^k \cap \Delta^{k'} = \emptyset$.
On écrit

$$Q = \cup_{1\leq k\leq n} Q^k Q_I = \cup_{1\leq k\leq n} Q^k_I Q_f = \cup_{1\leq k\leq n} Q^k_f \Delta = \cup_{1\leq k\leq n} \Delta^k$$

D'autre part, on considère aussi chaque état de Q comme un symbole unaire, et chaque transition $\delta = (q, \langle f, k\rangle, q')$ de Δ comme un symbole ayant la même arité que f.

On cherche à transformer un *p*CTRS R en un TRS sans stratégie R' qui simule le comportement de R.

Définition 5.1.1
Soit $R = \{L_k : l_k \to r_k \mid 1 \leq k \leq m\}$ *un p*CTRS *sur l'alphabet* Σ. *Le* TRS *correspondant* R' *sur l'alphabet* $\Sigma' = \Sigma \cup \{top^{\backslash 1}, j^{\backslash 1}\} \cup Q \cup \Delta$ *est* $R' = R'_1 \cup R'_2 \cup R'_3 \cup R'_4$ *avec* $x, x_1, \ldots, x_n$ *des variables et :*
$R'_1 = \{top(j(x)) \to top(q_I(x)) \mid q_I \in Q_I\}$
$R'_2 = \{q(f(x_1, \ldots, x_n)) \to \delta(x_1, \ldots, q'(x_i), \ldots, x_n) \mid \delta = (q, \langle f, i\rangle, q') \in \Delta\}$
$R'_3 = \{q_f(l_k) \to j(r_k) \mid q_f \in Q^k_f,\ (L_k : l_k \to r_k) \in R\}$
$R'_4 = \{\delta(x_1, \ldots, j(x_i), \ldots, x_n) \to j(f(x_1, \ldots, x_i, \ldots, x_n)) \mid \delta = (q, \langle f, i\rangle, q') \in \Delta\}$

Pour comprendre la transformation, on commence par considérer une version simplifiée de R' obtenue en remplaçant δ par f dans R'_2 et R'_4. Le rôle de δ est de préserver la terminaison. Si l'état q apparaît à la position p dans le terme $t'' \in T(\Sigma')$, cela signifie qu'il y a un chemin partant de la racine de t'' et allant jusqu'à la position p qui est partiellement reconnu par l'automate[1] dans l'état q. Le terme t'' est obtenu à partir de $top(j(t))$ en appliquant en premier une règle de R'_1 puis plusieurs règles de R'_2 qui descendent à chaque fois d'un pas en utilisant une transition de l'automate. Si $q \in Q_f$, une étape de réécriture de R peut être appliquée grâce à R'_3 ; q est remplacé par j, qui est un jeton. Alors j remonte à l'aide des règles de R'_4, pour permettre de faire la réécriture suivante en garantissant que l'automate valide un chemin qui part bien de la racine du terme. Le symbole *top* nous permet d'identifier la racine du terme.

Remarque 5.1.2
Le nombre de règles de R' est linéaire par rapport à la taille de l'automate global, car

$$|R'| = |Q_I| + |\Delta| + |Q_f| + |\Delta| \ \leq\ 2 \times (|Q| + |\Delta|)$$

Le théorème ci-dessous nous garantit que le TRS transformé est équivalent au TRS d'origine. La preuve de ce théorème est donnée dans la section 5.1.1.

1. Partiellement reconnu car q n'est pas forcément un état final.

Théorème 5.1.3. *Soit $t \in T(\Sigma)$. On a $t \to^*_{R_{pc}} t'$ si et seulement si $top(j(t)) \to^*_{R'} top(j(t'))$.*

Exemple 5.1.4
Soient $\Sigma = \{f^{\backslash 2}, g^{\backslash 2}, a^{\backslash 0}, b^{\backslash 0}\}$ et $R = \{(\langle f,1\rangle.\langle g,2\rangle)^* : a \to b\}$. L'automate de mots $\mathcal{A} = (Dir(\Sigma), Q, Q_I, Q_f, \Delta)$ est défini par :

- $Dir(\Sigma) = \{\langle f,1\rangle, \langle f,2\rangle, \langle g,1\rangle, \langle g,2\rangle\}$,
- $Q = \{q, q'\}$,
- $Q_I = \{q\}$,
- $Q_f = \{q\}$
- $\Delta = \{(q, \langle f,1\rangle, q'), (q', \langle g,2\rangle, q)\}$

On a $\delta_1 = (q, \langle f,1\rangle, q')$ et $\delta_2 = (q', \langle g,2\rangle, q)$ et
$R'_1 = \{top(j(x)) \to top(q(x))\}$,
$R'_2 = \{q(f(x,y)) \to \delta_1(q'(x), y),\ q'(g(x,y)) \to \delta_2(x, q(y))\}$,
$R'_3 = \{q(a) \to j(b)\}$,
$R'_4 = \{\delta_1(j(x), y) \to j(f(x,y)),\ \delta_2(x, j(y)) \to j(g(x,y))\}$.

On prend le terme $t = f(g(a,a),a)$. Le terme initial est $top(j(f(g(a,a),a)))$. On a :
$top(j(f(g(a,a),a)))$
$\to_{R'_1} top(q(f(g(a,a),a)))$
$\to_{R'_2} top(\delta_1(q'(g(a,a),a)))$
$\to_{R'_2} top(\delta_1(\delta_2(a, q(a)),a))$
$\to_{R'_3} top(\delta_1(\delta_2(a, j(b)),a))$
$\to_{R'_4} top(\delta_1(j(g(a,b)),a))$
$\to_{R'_4} top(j(f(g(a,b),a)))$

Et donc $f(g(a,a),a) \to_{R_{pc}} f(g(a,b),a)$.

Grâce au résultat suivant, la terminaison d'un pCTRS peut être prouvée avec les techniques habituellement utilisées pour les TRS sans stratégie.

Théorème 5.1.5. *Soit Σ un alphabet. Soit R un système de réécriture prefix-constrained défini sur Σ. Soit R' le système de réécriture défini sur Σ' issu de l'application de la Définition 5.1.1 sur R.*
Le pCTRS R est terminant sur Σ si et seulement si R' est terminant sur Σ'.

5.1.1 Preuve de correction de la transformation du *p*CTRS

Le théorème 5.1.3 est obtenu grâce au corollaire 5.1.9 et au lemme 5.1.11 qui sont présentés ci-dessous.

Lemme 5.1.6
Soient $t \in T(\Sigma)$, $p = (u_1 \cdot u_2 \cdot \ldots \cdot u_n) \in Pos(t)$, et $q_0, q_n \in Q^k$. Pour chaque $i \in \{1, \ldots, n\}$, on note $v_i = u_1 \ldots u_i$, et $v_0 = \epsilon$. Alors :

$$(q_0, path(t,p)) \to_{R_{pc}|\delta_1,\ldots,\delta_n \in \Delta^k} (q_n, \epsilon) \iff q_0(t) \to^*_{R'_2} t\{\delta_1\}_{v_0} \cdots \{\delta_n\}_{v_{n-1}}[q_n(t|_p)]_p$$

Démonstration. $\Rightarrow$**.** Par récurrence sur la longueur n de la dérivation

$$(q_0, path(t,p)) \to^*_{R_{pc}|\delta_1,\ldots,\delta_n \in \Delta^k} (q_n, \epsilon)$$

Cas de base : $n = 0$. On a $p = \epsilon$ et $(q_0, path(t, \epsilon)) = (q_n, \epsilon)$, c'est-à-dire $q_n = q_0$ et $path(t, \epsilon) = \epsilon$. On doit prouver que $q_0(t) \to^*_{R'_2} t[q_n(t|_p)]_p$. La dérivation peut être achevée en 0 étape vu que

$$t[q_n(t|_p)]_p = t[q_0(t|_\epsilon)]_\epsilon = t[q_0(t)]_\epsilon = q_0(t)$$

Étape de récurrence : on suppose que $n \geq 1$. On note $p' = u_2 \dots u_n$ et pour chaque $i \in \{2, \dots, n\}$, on écrit $v'_i = u_2 \dots u_i$, et $v'_1 = \epsilon$. On a $p = u_1.p'$ et $v_i = u_1.v'_i$. On remarque que $p \neq \epsilon$, alors t n'est pas une constante. On prend $t = f(t_1, \dots, t_{u_1}, \dots, t_{arity(f)})$. La dérivation que nous trouvons est

$$(q_0, path(t, p)) \to_{R_{pc}|\delta_1 \in \Delta^k} (q_1, path(t_{u_1}, p')) \to^*_{R_{pc}|\delta_2, \dots, \delta_n \in \Delta^k} (q_n, \epsilon)$$

On regarde la première étape. On a $\delta_1 = (q_0, \langle f, u_1 \rangle, q_1)$.
Vue la définition 5.1.1, $(q_0(f(x_1, \dots, x_{arity(f)})) \to \delta_1(x_1, \dots, q_1(x_{u_1}), \dots, x_{arity(f)})) \in R'_2$. Alors

$$q_0(t) = q_0(f(t_1, \dots, t_{u_1}, \dots, t_{arity(f)}) \to_{R'_2} \delta_1(t_1, \dots, q_1(t_{u_1}), \dots, t_{arity(f)})$$

D'un autre coté, la dérivation $(q_1, path(t_{u_1}, p')) \to^*_{R_{pc}|\delta_2, \dots, \delta_n \in \Delta^k} (q_n, \epsilon)$ inclut n–1 étapes. Grâce à l'hypothèse de récurrence, on a $q_1(t_{u_1}) \to^*_{R'_2} t_{u_1}\{\delta_2\}_{v'_1} \cdots \{\delta_n\}_{v'_{n-1}}[q_n(t_{u_1}|_{p'})]_{p'}$. Donc :

$$\begin{array}{rcl}
q_0(t) & \to_{R'_2} & \delta_1(t_1, \dots, q_1(t_{u_1}), \dots, t_{arity(f)}) \\
& \to^*_{R'_2} & \delta_1(t_1, \dots, t_{u_1}\{\delta_2\}_{v'_1} \cdots \{\delta_n\}_{v'_{n-1}}[q_n(t_{u_1}|_{p'})]_{p'}, \dots, t_{arity(f)}) \\
& = & \delta_1(t_1, \dots, t_{u_1}, \dots, t_{arity(f)})\{\delta_2\}_{u_1.v'_1} \cdots \{\delta_n\}_{u_1.v'_{n-1}}[q_n(t_{u_1}|_{p'})]_{u_1.p'} \\
& = & \delta_1(t_1, \dots, t_{u_1}, \dots, t_{arity(f)})\{\delta_2\}_{v_1} \cdots \{\delta_n\}_{v_{n-1}}[q_n(t|_p)]_p \\
& = & f(t_1, \dots, t_{u_1}, \dots, t_{arity(f)})\{\delta_1\}_\epsilon\{\delta_2\}_{v_1} \cdots \{\delta_n\}_{v_{n-1}}[q_n(t|_p)]_p \\
& = & t\{\delta_1\}_{v_0}\{\delta_2\}_{v_1} \cdots \{\delta_n\}_{v_{n-1}}[q_n(t|_p)]_p
\end{array}$$

$\Leftarrow$ Par récurrence sur la taille n de la dérivation $q_0(t) \to^*_{R'_2} t\{\delta_1\}_{v_0} \cdots \{\delta_n\}_{v_{n-1}}[q_n(t|_p)]_p$.

Cas de base : $n = 0$. On a $p = \epsilon$ et $q_0(t) = t[q_n(t|_\epsilon)]_\epsilon = q_n(t)$, c'est-à-dire $q_n = q_0$. On veut prouver que $(q_0, path(t, \epsilon)) \to^*_{R_{pc}} (q_0, \epsilon)$. Cela peut être fait en 0 étape, vu que $path(t, \epsilon) = \epsilon$ alors $(q_0, path(t, \epsilon)) = (q_0, \epsilon)$.

Étape de récurrence : on suppose que $n \geq 1$. On note $p' = u_2 \dots u_n$ et pour chaque $i \in \{2, \dots, n\}$, on écrit $v'_i = u_2 \dots u_i$, et $v'_1 = \epsilon$. Alors $p = u_1 \cdot p'$ et $v_i = u_1 \cdot v'_i$. On remarque que $p \neq \epsilon$, donc t n'est pas une constante. On note $t = f(t_1, \dots, t_{u_1}, \dots, t_{arity(f)})$. La dérivation s'écrit :

$$\begin{array}{l}
q_0(t) \to_{R'_2} t\{\delta_1\}_{v_0}[q_1(t|_{u_1})]_{u_1} \; et \\
q_1(t_{u_1}) \to^*_{R'_2} t_{u_1}\{\delta_2\}_{v'_1} \cdots \{\delta_n\}_{v'_{n-1}}[q_n(t_{u_1}|_{p'})]_{p'}
\end{array}$$

On remarque que

$$\begin{array}{rcl}
t\{\delta_1\}_{v_0}[q_1(t|_{u_1})]_{u_1} & = & f(t_1, \cdots, t_{u_1}, \cdots, t_{arity(f)})\{\delta_1\}_{v_0}[q_1(t|_{u_1})]_{u_1} \\
& = & \delta_1(t_1, \cdots, t_{u_1}, \cdots, t_{arity(f)})[q_1(t_{u_1})]_{u_1} \\
& = & \delta_1(t_1, \cdots, q_1(t_{u_1}), \cdots, t_{arity(f)})
\end{array}$$

Donc $q_0(t) \to_{R'_2} \delta_1(t_1, \cdots, q_1(t_{u_1}), \cdots, t_{arity(f)})$.
Vue la Définition 5.1.1, on a $\delta_1 = (q_0, \langle f, u_1\rangle, q_1)$.
D'autre part, $path(t,p) = \langle f, u_1\rangle.path(t_{u_1}, p')$. Alors $(q_0, path(t,p)) \to_{R_{pc}|\delta_1} (q_1, path(t_{u_1}, p'))$.

La dérivation $q_1(t_{u_1}) \to^*_{R'_2} t_{u_1}\{\delta_2\}_{v'_1} \cdots \{\delta_n\}_{v'_{n-1}}[q_n(t_{u_1}|_{p'})]_{p'}$ inclut $n-1$ étapes. D'après l'hypothèse de récurrence, on a $(q_1, path(t_{u_1}, p')) \to^*_{R_{pc}|\delta_2,\cdots,\delta_n \in \Delta^k} (q_n, \epsilon)$. Finalement on a

$$(q_0, path(t,p)) \to_{R_{pc}|\delta_1} (q_1, path(t_{u_1}, p')) \to^*_{R_{pc}|\delta_2,\cdots,\delta_n \in \Delta^k} (q_n, \epsilon)$$

□

Lemme 5.1.7
Soient $t \in T(\Sigma)$, $p = u_1 \dots u_n \in Pos(t)$, $q_I \in Q_I^k$ et $q_f \in Q_f^k$. Pour chaque $i \in \{1, \dots, n\}$, on note $v_i = u_1 \dots u_i$, et $v_0 = \epsilon$. Si $t \to_{[p, l_k \to r_k, \sigma]} t'$ et $(q_I, path(t,p)) \to^*_{R_{pc}|\delta_1,\dots,\delta_n \in \Delta^k} (q_f, \epsilon)$, alors

$$\begin{array}{rll}
top(j(t)) & \to_{R'_1} & top(q_I(t)) \\
& \to^*_{R'_2} & top(t\{\delta_1\}_{v_0} \cdots \{\delta_n\}_{v_{n-1}}[q_f(t|_p)]_p) \\
& = & top(t\{\delta_1\}_{v_0} \cdots \{\delta_n\}_{v_{n-1}}[q_f(\sigma(l_k))]_p) \\
& \to_{R'_3} & top(t\{\delta_1\}_{v_0} \cdots \{\delta_n\}_{v_{n-1}}[j(\sigma(r_k))]_p) \\
& \to^*_{R'_4} & top(j(t[\sigma(r_k)]_p)) \\
& = & top(j(t'))
\end{array}$$

Démonstration. Cela provient du lemme 5.1.6, et de la forme des règles de R'_3 et R'_4. □

Corollaire 5.1.8
Soit $t \in T(\Sigma)$. Si $t \to_{R_{pc}} t'$ alors $top(j(t)) \to^+_{R'} top(j(t'))$.

Corollaire 5.1.9
Soit $t \in T(\Sigma)$. Si $t \to^*_{R_{pc}} t'$ alors $top(j(t)) \to^*_{R'} top(j(t'))$.

Lemme 5.1.10
Soient $t \in T(\Sigma)$ et $q_I \in Q_I^k$.
Si $top(q_I(t)) \to^+_{R'_2 \cup R'_3 \cup R'_4} top(j(t'))$, alors il existe $p \in Pos(t)$, $t \to_{R_{pc}, [p, l_k \to r_k]} t'$.

Démonstration. Vu que $t \in T(\Sigma)$ et que $Q \cup \{j\} \notin \Sigma$, on peut facilement voir que le terme $top(q_I(t))$ contient exactement un élément de $Q \cup \{j\}$. Les règles de R'_2 préservent le nombre de symboles de Q (on ne tient pas compte des symboles de Q présents dans les symboles de Δ) et n'agissent pas sur le symbole j. Les règles de R'_3 consomment un symbole de Q et produisent le symbole j. Les règles de R'_4 préservent le symbole j et n'agissent pas sur le nombre de symboles de Q. On a donc chaque terme dans la dérivation qui contient exactement un élément de $Q \cup j$.

Pour pouvoir être appliquées, les règles R'_2 ont besoin d'une occurrence d'un élément de Q, celles de R'_3 remplacent un élément de Q par j, et celles de R'_4 ont besoin d'une occurrence de j. Alors la dérivation est de la forme :

$$\begin{array}{rll}
top(q_I(t)) & \to^*_{R'_2} & top(t\{\delta_1\}_{v_0} \cdots \{\delta_n\}_{v_{n-1}}[q_n(t|_p)]_p) \\
& \to_{R'_3} & top(t\{\delta_1\}_{v_0} \cdots \{\delta_n\}_{v_{n-1}}[j(t'|_p)]_p) \\
& \to^*_{R'_4} & top(j(t[t'|_p])_p) = top(j(t'))
\end{array}$$

On remarque que $q_n \in Q_f^k$. Vu le Lemme 5.1.6, $(q_I, path(t,p)) \rightarrow^*_{R_{pc}|\delta_1,\ldots,\delta_n \in \Delta^k} (q_n, \epsilon)$, Autrement dit, $path(t,p) \in L_k$. Donc $t \rightarrow_{R_{pc},[p,l_k \rightarrow r_k]} t'$. □

Lemme 5.1.11
Soit $t \in T(\Sigma)$. Si $top(j(t)) \rightarrow^+_{R'} top(j(t'))$, alors $t \rightarrow^+_{R_{pc}} t'$.

Démonstration. La dérivation est composée d'étapes de la forme :

$$top(j(t_i)) \rightarrow_{R'_1} top(q_i(t_i)) \rightarrow^*_{R'_2 \cup R'_3 \cup R'_4} top(j(t_{i+1}))$$

où $q_i \in Q_I$. Vu le Lemme 5.1.10, $t_i \rightarrow_{R_{pc}} t_{i+1}$. □

5.1.2 Preuve de conservation de la terminaison du TRS

Le théorème 5.1.5 est obtenu grâce au lemme 5.1.12 et au corollaire 5.1.22 ci-dessous.

Lemme 5.1.12
Si R' est terminant, alors R est terminant.

Démonstration. Nous prouvons le résultat par contraposée. On pose R non terminant, c'est-à-dire qu'il existe une dérivation infinie $t_0 \rightarrow_{R_{pc}} t_1 \rightarrow_{R_{pc}} \cdots t_n \rightarrow_{R_{pc}} \cdots$. Vu le corollaire 5.1.8, $top(j(t_0)) \rightarrow^+_{R'} top(j(t_1)) \rightarrow^+_{R'} \cdots top(j(t_n)) \rightarrow^+_{R'} \cdots$, c'est à dire que R' n'est pas terminant. □

Pour prouver le sens inverse, nous avons besoin d'utiliser le typage de termes et d'utiliser le résultat de H. Zantema [58] qui traite de la persistance de la terminaison en ajoutant et en retirant les types.

Définition 5.1.13
Soit l'ensemble de type $S = \{s_\Sigma, s_Q, s_\top\}$. On définit la fonction de typage de Σ' [2] vers S telle que :

- $\forall f \in \Sigma,\ f : s_\Sigma \times \cdots \times s_\Sigma \rightarrow s_\Sigma$
- $\forall q \in Q,\ q : s_\Sigma \rightarrow s_Q$
- $j : s_\Sigma \rightarrow s_Q$
- $\forall \delta = (q, \langle f, k \rangle, q') \in \Delta,\ \delta : s_\Sigma \times \cdots \times s_\Sigma \times s_Q \times s_\Sigma \times \cdots \times s_\Sigma \rightarrow s_Q$, *où s_Q est le k-ième argument du membre gauche.*
- $top : s_Q \rightarrow s_\top$

Soit R'_{st} le TRS *R' vu comme un* TRS *many-sorted, en considérant que toutes les variables apparaissant dans les règles de réécriture de R'_{st} sont du type s_Σ. La fonction de typage est consistante pour R' car pour chaque règle $l \rightarrow r$ de R'_{st}, les termes l et r sont bien typés, et les types de l et r sont égaux.*

On remarque que pour tous les termes bien typés $t, t' \in T(\Sigma')$, $t \rightarrow_{R'} t'$ si et seulement si $t \rightarrow_{R'_{st}} t'$. D'autre part, si $t \in T(\Sigma')$ n'est pas bien typé, alors t n'est pas réductible par R'_{st} tandis que t peut être réductible par R'. En d'autres mots, R'_{st} fonctionne uniquement avec des termes bien typés.

Remarque 5.1.14
Si $t \in T(\Sigma')$ est du type s_Σ, alors $t \in T(\Sigma)$.

2. Il s'agit ici de l'alphabet défini dans la Définition 5.1.1, qui sera réutilisé dans la suite de cette section.

Vu que R'_{st} n'inclut pas de règles collapsing, le théorème 21 de [58] s'applique, c'est à dire que la terminaison est persistante.

Par conséquent, on obtient le lemme suivant :

Lemme 5.1.15
Si R' n'est pas terminant, alors R'_{st} n'est pas terminant.

Lemme 5.1.16
Soit $t \in T(\Sigma')$ un terme bien typé du type $s_\top$. Alors $t(\epsilon) = top$ et $\forall p \in Pos(t)\backslash\{\epsilon\}$, $t(p) \neq top$.

Démonstration. Trivial. □

Lemme 5.1.17
Soit $t \in T(\Sigma')$ un terme bien typé. Si $t \to_{R'_1} t'$ alors t est du type $s_\top$.

Démonstration. Soit $p \in Pos(t)$ la position où l'étape de réécriture $t \to_{R'_1} t'$ s'effectue. Alors, $t(p) = top$ et $t|_p$ est du type $s_\top$. Si $p \neq \epsilon$, t n'est pas bien typé selon la fonction de typage donnée dans la définition 5.1.13. On a donc $p = \epsilon$. □

Lemme 5.1.18
Soit $t_0 \in T(\Sigma')$ un terme bien typé. Si $t_0 \to^+_{R'_{st}} t_n$ par une dérivation contenant au moins une étape utilisant une règle de R'_1, alors pour tout $i \in \{0, \ldots, n\}$, t_i est du type $s_\top$ et $t_i(\epsilon) = top$.

Démonstration. Il existe $j \in \{0, \ldots, n-1\}$ tel que $t_j \to_{R'_1} t_{j+1}$. D'après le lemme précédent, t_j est du type $s_\top$. Comme toutes les règles de R'_{st} préservent le typage, pour tout i, t_i est du type $s_\top$ et $t_i(\epsilon) = top$. □

Lemme 5.1.19
Soit $R' = R'_1 \cup R'_2 \cup R'_3 \cup R'_4$ le TRS *de la Définition 5.1.1. Alors $R'_2 \cup R'_3 \cup R'_4$ est terminant.*

Démonstration. En utilisant le multiset path ordering (MPO) [4] avec la relation de précédence définie par : $\forall q, q' \in Q$, $\forall \delta \in \Delta$, $\forall f \in \Sigma$, $q \sim q' > \delta \sim f > j$, et en utilisant pour chaque règle de réécriture de R'_3, le fait que $Var(r_k) \subseteq Var(l_k)$. □

Lemme 5.1.20
Soit $t'_0 \in T(\Sigma')$ un terme bien typé. Si la dérivation $t'_0 \to_{R'_{st}} t'_1 \to_{R'_{st}} t'_2 \to^*_{R'_{st}} \ldots$ est infinie, alors

- *Pour tout $i \in \mathbb{N}$, le type de t'_i est $s_\top$,*
- *et il existe un sous-ensemble infini $I = \{i_1, i_2, \ldots\}$ de $\mathbb{N}$ tel que pour tout $i \in I$ il existe $q_i \in Q_I$ et $t_i \in T(\Sigma)$ tels que $t'_i = top(q_i(t_i))$, et on a $t_{i_1} \to_{R_{pc}} t_{i_2} \to_{R_{pc}} \ldots$, ce qui est une dérivation infinie sur R.*

Démonstration. D'après le lemme 5.1.19, la dérivation contient nécessairement une infinité d'étapes utilisant R'_1. On pose $R'' = R'_2 \cup R'_3 \cup R'_4$.
La dérivation est de la forme $t'_0 \to^*_{R''} t'_{i_1-1} \to_{R'_1} t'_{i_1} \to^*_{R''} t'_{i_2-1} \to_{R'_1} t'_{i_2} \cdots$. D'après le lemme 5.1.18, tous les termes de cette dérivation sont du type $s_\top$. Par conséquent, pour chaque j, $t'_{i_j-1} = top(j(t_{i_j-1}))$ et $t'_{i_j} = top(q_{i_j}(t_{i_j}))$ où $q_{i_j} \in Q_I$ et $t_{i_j-1} = t_{i_j}$ sont du type s_Σ, c'est à dire $t_{i_j-1}, t_{i_j} \in T(\Sigma)$.
D'après le lemme 5.1.10, pour chaque j, $t_{i_j} \to_{R_{pc}} t_{i_{j+1}-1} = t_{i_{j+1}}$. Donc, il y a une dérivation infinie $t_{i_1} \to_{R_{pc}} t_{i_2} \to_{R_{pc}} \ldots$. □

Corollaire 5.1.21
Si R'_{st} n'est pas terminant, alors R n'est pas terminant.

Grâce au corollaire 5.1.21 et au lemme 5.1.15, on a :

Corollaire 5.1.22
Si R est terminant, alors R' est terminant.

5.2 Le cas de la stratégie context-sensitive

Un TRS *context-sensitive* (R_0, μ) peut être vu comme un *prefix-constrained* TRS particulier $R = \{L_k : l_k \to r_k \mid (l_k \to r_k) \in R_0\}$, où tous les langages L_k sont identiques (noté L), et L est défini par un automate de mots avec $Q = Q_I = Q_f = \{q\}$ et $\Delta = \{(q, \langle f, k\rangle, q) \mid f \in \Sigma,\ k \in \mu(f)\}$. Puisque q est l'unique état, on note $\langle f, k\rangle$ à la place de $(q, \langle f, k\rangle, q)$, ce qui représente une transition $\delta \in \Delta$ et est considéré comme un symbole ayant la même arité que f. Alors le système de réécriture obtenu par notre transformation s'écrit $R' = R'_1 \cup R'_2 \cup R'_3 \cup R'_4$ (où $x, x_1, \ldots, x_n$ sont des variables) :

$$
\begin{aligned}
R'_1 &= \{top(j(x)) \to top(q(x))\}\\
R'_2 &= \{q(f(x_1, ..., x_n)) \to \langle f, k\rangle(x_1, ..., q(x_k), ..., x_n) \mid f \in \Sigma,\ k \in \mu(f)\}\\
R'_3 &= \{q(l_k) \to j(r_k) \mid (l_k \to r_k) \in R_0\}\\
R'_4 &= \{\langle f, k\rangle(x_1, ..., j(x_k), ..., x_n) \to j(f(x_1, ..., x_k, ..., x_n)) \mid f \in \Sigma,\ k \in \mu(f)\}
\end{aligned}
$$

Maintenant, si on remplace q par *active*, et j par *mark*, et $\delta = \langle f, k\rangle$ par f, on a la seconde transformation de [29], à cela près que les règles de réécriture pour les symboles additionnels *proper* et *ok* sont manquantes. En d'autre termes, notre transformation est plus simple et on utilise moins de règles. Cependant, la transformation de [29], sans les règles pour *proper* et *ok* ne préserve pas la terminaison, comme les auteurs l'ont montré en utilisant un contre-exemple. Essayons de traiter le même contre-exemple en utilisant notre transformation.

Exemple 5.2.1 ([29])
Soit $R_0 = \{f(x, g(x), y) \to f(y, y, y),\ g(b) \to c,\ b \to c\}$ avec $\mu(f) = \emptyset$ et $\mu(g) = \{1\}$. Ce TRS context-sensitive est clairement terminant. En utilisant notre transformation, on a

$$
R' = \left\{
\begin{array}{l}
top(j(x)) \to top(q(x)),\\
q(g(x)) \to \langle g, 1\rangle(q(x)),\\
q(f(x, g(x), y)) \to j(f(y, y, y))\\
q(g(b)) \to j(c),\\
q(b) \to j(c),\\
\langle g, 1\rangle(j(x)) \to j(g(x))
\end{array}
\right\}
$$

Considérons le terme $t = top(q(f(s, s, s)))$ avec $s = q(g(b))$. Alors :

$$
\begin{aligned}
top(q(f(s, s, s))) &\to_{R'} top(q(f(j(c), s, s)))\\
&\to_{R'} top(q(f(j(c), \langle g, 1\rangle(q(b)), s)))\\
&\to_{R'} top(q(f(j(c), \langle g, 1\rangle(j(c)), s)))\\
&\not\to_{R'} top(j(f(s, s, s)))\\
&\to_{R'} top(q(f(s, s, s)))
\end{aligned}
$$

L'étape $\not\to_{R'}$ n'est plus possible grâce au fait que nous utilisons $\delta = \langle g, 1\rangle$ à la place de g. Par conséquence ce cycle n'est pas possible.

5.3 Expérimentations de preuves de terminaison avec AProVE

Nous proposons l'exemple suivant.

Exemple 5.3.1
Soit l'alphabet $\Sigma = \{f^{\setminus 1}, g^{\setminus 1}, a^{\setminus 0}\}$. On prend le *p*CTRS suivant : $R = \{f(g(x)) \to f(f(g(x))) \mid L = \{\langle f, 1\rangle^*.\langle g, 1\rangle\}\}$

Si on considère R sans les contraintes de préfixe, il n'est pas terminant. On peut par exemple prendre le terme $f(g(a))$. Ce terme ce réécrit alors en $f(f(g(a)))$ qui se réécrit en $f(f(f(g(a))))$, et ainsi de suite. Toutefois, le terme $f(g(a))$ ne peut pas être réécrit par le *p*CTRS R car le sous terme présent à la seule position de réécriture valide est a et le membre gauche de notre règle de réécriture ne matche pas avec ce dernier.

Transformons R en R' un TRS sans stratégie équivalent. L'ensemble des chemins est reconnu par l'automate fini déterministe de mots $\mathcal{A}_L = (\Sigma_{\mathcal{A}_L}, Q, q_i, \Delta, Q_F)$, avec :

$$\begin{aligned}
\Sigma_{\mathcal{A}_L} &= \{\langle f, 1\rangle, \langle g, 1\rangle\} \\
Q &= \{q_i, q_f\} \\
\Delta &= \{\delta = (q_i, \langle f, 1\rangle, q_i), \delta_1 = (q_i, \langle g, 1\rangle, q_f)\} \\
Q_F &= \{q_f\})
\end{aligned}$$

On a $\Sigma' = \Sigma \cup \{j^{\setminus 1}, top^{\setminus 1}\} \cup Q \cup \Delta$ et $R' = R'_1 \cup R'_2 \cup R'_3 \cup R'_4$ avec :

$$\begin{aligned}
R'_1 &= \{top(j(x)) \to top(q_i(x))\}, \\
R'_2 &= \{q_i(f(x)) \to \delta(q_i(x)), q_i(g(x)) \to \delta_1(q_f(x))\}, \\
R'_3 &= \{q_f(f(g(x))) \to j(f(f(g(x))))\} et \\
R'_4 &= \{\delta(j(x)) \to j(f(x)), \delta_1(j(x)) \to j(g(x))\}.
\end{aligned}$$

On peut maintenant utiliser la version en ligne de AProVE [3] pour essayer de prouver la terminaison de ce système de réécriture. Voici R' au format WST :

```
(VAR x)
(RULES
    top(j(x)) -> top(qi(x))

    qi(f(x)) -> delta(qi(x))
    qi(g(x)) -> delta1(qf(x))

    qf(f(g(x)) -> j(f(f(g(x))))
```

3. Disponible ici : http://aprove.informatik.rwth-aachen.de/index.asp?subform=termination_proofs.html

```
    delta(j(x)) -> j(f(x))
    delta1(j(x)) -> j(g(x))
)
```

AProVE arrive à prouver sans problème que R' est terminant, par conséquent d'après le théorème 5.1.5 R est également terminant.

Nous venons de présenter un exemple simple qui montre que nos travaux peuvent être utilisés pour démontrer la terminaison d'un pCTRS à l'aide d'AProVE.

5.4 Conclusion

Dans ce chapitre nous avons présenté une technique permettant de transformer un système de réécriture *prefix-constrained* en un système de réécriture sans stratégie. De plus, nous donnons la preuve que cette transformation est correcte et qu'elle préserve la terminaison du système de réécriture initial.

La stratégie *context-sensitive* est un cas particulier de la stratégie *prefix-constrained*. Les travaux [29] proposent une technique de transformation des systèmes de réécriture *context-sensitive* qui génère plus de règles que celle présentée dans ce chapitre. Notre technique améliore donc ces travaux.

Cependant, notre technique de transformation de pCTRS en TRS sans stratégie peut par exemple empêcher l'utilisation de certaines techniques de preuves de terminaison.

Prenons par exemple les RPO[4], une technique qui surpasse les MPO. Pour rappel, le principe du *path ordering* est, à partir d'un ordre noéthérien sur les symboles, de pouvoir ordonner des termes. On peut prouver qu'un TRS est terminant si pour chaque règle de réécriture, le membre gauche est un prédécesseur du membre droit.

On considère ici l'ensemble des pCTRS qui possèdent au moins une règle de réécriture avec un langage de chemin non vide et qui ne contient pas uniquement le mot vide. On a alors l'ensemble des règles de R'_1 qui contient au moins une règle de la forme $top(j(x)) \to top(q_i(x))$ avec q_i qui est l'état initial. Afin de pouvoir utiliser les RPO, il faut que le terme $top(j(x))$ soit un prédécesseur du terme $top(q_i(x))$. Pour ce faire on doit fixer l'ordre sur les symboles suivant $j > q_i$. L'ensemble R'_2 est lui au moins composé d'une règle partant de l'état initial. En effet, dans le cas contraire le langage de l'automate serait vide. On a donc une règle de la forme $q_i(f(x_1, \ldots, x_n)) \to \delta(x_1, \ldots, q(x_i), \ldots, x_n)$ avec $\delta = (q_i, \langle f, i\rangle, q)$. On doit fixer soit $q_i > \delta$ soit $q_i \sim \delta$. Enfin, l'ensemble R'_4 contient au moins une règle de la forme $\delta(x_1, \ldots, j(x_i), \ldots, x_n) \to j(f(x_1, \ldots, x_n))$. On doit alors fixer soit $\delta > j$ soit $\delta \sim j$. On a alors $j > q_i > \delta > j$ ou $j > q_i \sim \delta > j$ ou $j > q_i > \delta \sim j$ ou $j > q_i \sim \delta \sim j$. Or, aucun de ces ordres n'est noéthérien. Il n'existe donc pas d'ordre noéthérien sur les symboles permettant d'avoir chaque membre gauche qui est un prédécesseur de son membre droit.

On ne peut donc pas utiliser les RPO pour prouver la terminaison des systèmes de réécriture issus de notre technique de transformation d'un pCTRS possèdant au moins une règle de réécriture avec un langage de chemin non vide et qui ne contient pas uniquement

4. Recursive Path Ordering

le mot vide. Cependant, d'autres techniques (dont certaines implémentées dans AProVE) peuvent être utilisées.

Chapitre 6

Conclusion

Cette thèse s'intéresse à la question de l'analyse d'accessibilité. On peut distinguer deux cas pour ce problème.

Si l'ensemble des descendants est fini, alors il est possible de le représenter en énumérant chaque descendant. Si le langage initial est fini, avoir un système de réécriture terminant revient à avoir un ensemble fini de descendants. La terminaison des systèmes de réécriture sans stratégie a fortement été étudiée. C'est pourquoi nous proposons de transformer des systèmes de réécriture avec stratégie en systèmes équivalents mais sans stratégie. Notre technique permet la transformation des pCTRS. Les CSTRS étant un cas particulier des pCTRS, nous pouvons utiliser la même technique pour les CSTRS. Cette technique améliore les travaux [29] en réduisant le nombre de règles générées.

D'un autre coté, si l'ensemble des descendants est infini, il parfois possible de les représenter avec une structure finie, tel qu'un automate d'arbre ou un programme logique. Cependant, cela n'est pas toujours le cas, nous choisissons alors de transformer le problème d'accessibilité en un autre problème. Pour ce faire, nous utilisons une sur-approximation de l'ensemble des descendants. Ainsi, s'il n'y pas d'élément présent dans l'intersection de cette sur-approximation et de l'ensemble des configurations indésirables, nous prouvons qu'aucune configuration indésirable n'est accessible. S'il existe au moins un élément dans cette intersection, il n'est pas possible de savoir s'il s'agit d'un faux-positif ou d'une configuration réellement accessible.

Une technique utilisant les langages synchronisés de tuples d'arbres permet de faire le calcul d'une sur-approximation de l'ensemble des descendants à partir d'un système de réécriture linéaire [9, 10]. Nous proposons une modification de cette technique permettant de calculer une sur-approximation de l'ensemble des descendants *innermost* avec un système de réécriture non linéaire droit.

Nous présentons également, un algorithme qui permet de transformer les clauses copiantes en clauses non copiantes au prix d'une sur-approximation supplémentaire. Cette technique permet l'utilisation de systèmes de réécriture non linéaires droits pour le calcul de l'ensemble des descendants sans stratégie avec des langages synchronisés de tuples d'arbres.

Nous sommes actuellement en train de réaliser une implémentation de ces travaux. Ce prototype sera notamment utile pour développer des heuristiques afin d'améliorer la qualité des approximations et de rendre la méthode totalement automatique.

Le mécanisme removeCycles permet de rendre le nombre de pairs critiques fini au prix d'une sur-approximation. Cette sur-approximation est parfois trop violente pour permettre l'analyse d'atteignabilité. Dans certains cas, il est possible de répartir l'ensemble des paires critiques dans deux ensembles disjoints :

— un ensemble pouvant être infini et ne contenant que des paires critiques convergentes
— un ensemble fini de paires critiques.

Dans ce cas, il n'est pas forcément nécessaire d'utiliser le mécanisme removeCycles. Mettre au point un procédé le plus souvent capable de faire ce traitement permettrait d'améliorer sensiblement la qualité des sur-approximations.

Les travaux [11] proposent une technique similaire à la précédente mais avec des langages synchronisés algébriques de tuples d'arbres. Adapter les travaux que nous avons réalisés à cette classe plus large de langages semble intéressant.

De manière plus générale, beaucoup de techniques utilisées pour améliorer ou contrôler la qualité des sur-approximations régulières semblent intéressantes et plus ou moins difficiles à mettre en place pour les techniques utilisant des langages synchronisés. Il parait notamment intéressant d'essayer de transposer le contrôle de la sur-approximation à l'aide d'équations [8, 27] ou encore la méthode CEGAR.

L'autre résultat présenté dans cette thèse est un algorithme de transformation de programmes logiques en CS-programmes à l'aide d'une sur-approximation. Il est issu des travaux [42, 41] et ajoute la règle d'inférence appelée Generalization pour le rendre terminant.

L'utilisation d'un système de réécriture non linéaire gauche dans la technique [9, 10] génère des clauses qui ne sont pas des CS-clauses. En utilisant la transformation de programmes logiques en CS-programmes dès que l'on génère un programme qui n'est pas un CS-programme, on peut utiliser des systèmes de réécriture non linéaire gauche avec la technique issue des travaux [9, 10].

L'intersection de CS-programmes entre eux n'est pas stable. Cela nous force à utiliser un ensemble des configurations indésirables régulier. Transformer en CS-programme le programme logique représentant l'intersection de deux CS-programmes permet de calculer une sur-approximation de cette intersection. Cela nous permet d'utiliser un CS-programme pour représenter l'ensemble des configurations indésirables.

La sur-approximation induite par Generalization est souvent trop violente. La mise au point d'une autre règle d'inférence plus précise basée sur l'unification de langages permettrait d'améliorer la précision de cet algorithme.

Bibliographie

[1] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. The b-method. In Søren Prehn and W. J. Toetenel, editors, *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 2 : Tutorials*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 1991.

[2] Nirina Andrianarivelo, Vivien Pelletier, and Pierre Réty. Transforming prefix-constrained or controlled rewrite systems. In Mohamed Mosbah and Michaël Rusinowitch, editors, *SCSS 2017, The 8th International Symposium on Symbolic Computation in Software Science 2017, April 6-9, 2017, Gammarth, Tunisia*, volume 45 of *EPiC Series in Computing*, pages 49–62. EasyChair, 2017.

[3] Franz Baader, editor. *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[5] Emilie Balland, Yohan Boichut, Thomas Genet, and Pierre-Etienne Moreau. Towards an efficient implementation of tree automata completion. In José Meseguer and Grigore Rosu, editors, *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, volume 5140 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[6] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom : Piggybacking rewriting on java. In Baader [3], pages 36–47.

[7] Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Effective strategic programming for java developers. *Softw., Pract. Exper.*, 44(2) :129–162, 2014.

[8] Yohan Boichut, Benoît Boyer, Thomas Genet, and Axel Legay. Equational abstraction refinement for certified tree regular model checking. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, volume 7635 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2012.

[9] Yohan Boichut, Jacques Chabin, and Pierre Réty. Over-approximating descendants by synchronized tree languages. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 128–142. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[10] Yohan Boichut, Jacques Chabin, and Pierre Réty. Erratum of over-approximating descendants by synchronized tree languages. Technical report, LIFO, Université d'Orléans, 2015.

[11] Yohan Boichut, Jacques Chabin, and Pierre Réty. Towards more precise rewriting approximations. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 652–663. Springer, 2015.

[12] Yohan Boichut, Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Finer is better : Abstraction refinement for rewriting approximations. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.

[13] Yohan Boichut, Thomas Genet, Thomas P. Jensen, and Luka Le Roux. Rewriting approximations for fast prototyping of static analyzers. In Baader [3], pages 48–62.

[14] Yohan Boichut and Pierre-Cyrille Héam. A theoretical limit for safety verification techniques with regular fix-point computations. *Inf. Process. Lett.*, 108(1) :1–2, 2008.

[15] Yohan Boichut, Vivien Pelletier, and Pierre Réty. Synchronized tree languages for reachability in non-right-linear term rewrite systems. In Dorel Lucanu, editor, *Rewriting Logic and Its Applications - 11th International Workshop, WRLA 2016, Held as a Satellite Event of ETAPS, Eindhoven, The Netherlands, April 2-3, 2016, Revised Selected Papers*, volume 9942 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2016.

[16] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata : Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97 : Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[17] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2) :137 – 167, 1959.

[18] Horatiu Cirstea, Sergueï Lenglet, and Pierre-Etienne Moreau. A faithful encoding of programmable strategies into term rewriting systems. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 74–88. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[19] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude : specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2) :187–243, 2002.

[20] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of*

the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pages 238–252. ACM, 1977.

[22] Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2) :69–116, 1987.

[23] Joost Engelfriet and Erik Meineche Schmidt. IO and OI. I. *J. Comput. Syst. Sci.*, 15(3) :328–353, 1977.

[24] Joost Engelfriet and Erik Meineche Schmidt. IO and OI. II. *J. Comput. Syst. Sci.*, 16(1) :67–99, 1978.

[25] Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. In Tobias Nipkow, editor, *Rewriting Techniques and Applications, 9th International Conference, RTA-98, Tsukuba, Japan, March 30 - April 1, 1998, Proceedings*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

[26] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In David A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2000.

[27] Thomas Genet and Vlad Rusu. Equational approximations for tree automata completion. *J. Symb. Comput.*, 45(5) :574–597, 2010.

[28] Thomas Genet and Yann Salmon. Reachability analysis of innermost rewriting. *Logical Methods in Computer Science*, 13(1), 2017.

[29] Jürgen Giesl and Aart Middeldorp. Transformation techniques for context-sensitive rewrite systems. *J. Funct. Program.*, 14(4) :379–427, 2004.

[30] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with aprove. In van Oostrom [54], pages 210–220.

[31] Saul Gorn. Explicit definitions and linguistic dominoes. In *Systems and Computer Science, Proceedings of the Conference held at Univ. of Western Ontario*, pages 77–115, 1965.

[32] Valérie Gouranton, Pierre Réty, and Helmut Seidl. Synchronized tree languages revisited and new applications. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2001.

[33] Irène Guessarian. Pushdown tree automata. *Mathematical Systems Theory*, 16(4) :237–263, 1983.

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[35] Florent Jacquemard, Yoshiharu Kojima, and Masahiko Sakai. Term rewriting with prefix context constraints and bottom-up strategies. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2015.

[36] Jonathan Kochems and C.-H. Luke Ong. Improved functional flow and reachability analyses using indexed linear tree grammars. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 187–202. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[37] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009.

[38] Jan Leeuwen. *Handbook of Theoretical Computer Science*. Number vol. 1 in Handbook of Theoretical Computer Science. Elsevier, 1990.

[39] Sébastien Limet and Pierre Réty. E-unification by means of tree tuple synchronized grammars. *Discrete Mathematics & Theoretical Computer Science*, 1(1) :69–98, 1997.

[40] Sébastien Limet and Pierre Réty. E-unification by means of tree tuple synchronized grammars. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97 : Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*, volume 1214 of *Lecture Notes in Computer Science*, pages 429–440. Springer, 1997.

[41] Sébastien Limet and Gernot Salzer. Proving properties of term rewrite systems via logic programs. In van Oostrom [54], pages 170–184.

[42] Sébastien Limet and Gernot Salzer. Tree tuple languages from the logic programming point of view. *J. Autom. Reasoning*, 37(4) :323–349, 2006.

[43] Salvador Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), 1998.

[44] Pierre Réty Nirina Andrianarivelo, Vivien Pelletier. Transforming prefix-constrained or controlled rewrite systems. Technical report, LIFO, Université d'Orléans, 2017.

[45] Alberto Pettorossi and Maurizio Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program.*, 41(2-3) :197–230, 1999.

[46] Jing Chen Pierre Réty, Jacques Chabin. R-unification thanks to synchronized-contextfree languages. In *19th Workshop on Unification (UNIF'2005)*, 2005.

[47] Jing Chen Pierre Réty, Jacques Chabin. Synchronized contextfree tree-tuple languages. Technical report, LIFO, Université d'Orléans, 2006.

[48] David A Plaisted. *A recursively defined ordering for proving termination of term rewriting systems*. Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[49] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[50] John C. Reynolds. Separation logic : A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[51] William C. Rounds. Context-free grammars on trees. In Patrick C. Fischer, Seymour Ginsburg, and Michael A. Harrison, editors, *Proceedings of the 1st Annual ACM Symposium on Theory of Computing, May 5-7, 1969, Marina del Rey, CA, USA*, pages 143–148. ACM, 1969.

[52] Yann Salmon. *Analyse d'atteignabilité pour les programmes fonctionnels avec stratégie d'évaluation en profondeur*. PhD thesis, Université de Rennes 1, 2015. Thèse de doctorat dirigée par Genet, Thomas Informatique Rennes 1 2015.

[53] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, s2-43(1) :544–546, 1938.

[54] Vincent van Oostrom, editor. *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*. Springer, 2004.

[55] Jacques Chabin Yohan Boichut and Pierre Réty. Over-approximating descendants by synchronized tree languages. Technical report, LIFO, Université d'Orléans, 2013.

[56] Jacques Chabin Yohan Boichut and Pierre Réty. Towards more precise rewriting approximations (full version). Technical report, LIFO, Université d'Orléans, 2014.

[57] Vivien Pelletier Yohan Boichut and Pierre Réty. Synchronized tree languages for reachability in non-right-linear term rewrite systems (full version). Technical report, LIFO, Université d'Orléans, 2015.

[58] Hans Zantema. Termination of term rewriting : Interpretation and type elimination. *J. Symb. Comput.*, 17(1) :23–50, 1994.

Vivien PELLETIER

Sur-approximations non régulières et terminaison pour l'analyse d'accessibilité

Résumé :

L'analyse d'accessibilité est une des composantes de l'analyse de modèles. Elle consiste à modéliser un système complexe par trois ensembles : le langage initial, le langage des configurations indésirables et un système de réécriture. Le langage initial et le langage des configurations indésirables sont des ensembles de termes. Un terme est un mot mais construit à partir de symboles d'arités supérieures à 1. Le système de réécriture représente la dynamique du système complexe. C'est un ensemble de règles qui permettent d'obtenir un nouveau terme à partir d'un terme original. Pour effectuer une analyse d'accessibilité à partir de cette modélisation, on peut calculer l'ensemble des configurations accessibles. Cet ensemble aussi appelé ensemble des descendants est obtenu en appliquant le système de réécriture sur le langage initial jusqu'à ne plus obtenir de nouveaux termes. Une fois l'ensemble des descendants calculé, il reste à faire l'intersection entre celui-ci et l'ensemble des configurations indésirables. Si cette intersection est vide, alors il n'y a pas de configuration indésirable accessible, sinon les configurations présentes dans cette intersection sont accessibles. Cependant, l'ensemble des descendants n'est pas calculable dans le cas général. Pour contourner ce problème, nous calculons une sur-approximation des descendants. Ainsi, si l'intersection est vide, cela signifie toujours qu'aucune configuration indésirable n'est accessible. A contrario, s'il existe un terme dans l'intersection, il n'est pas possible de déterminer s'il s'agit d'un faux positif ou d'une configuration indésirable accessible. La précision de la sur-approximation est alors déterminante.

Mots clés : langages non réguliers d'arbres, réécriture, stratégie de réécriture, analyse d'accessibilité

Non-regular over-approximations and termination for reachability analysis

Abstract :

Reachability analysis is part of model checking. It consists to model complex systems by three sets : initial language, unwanted configurations and rewrite system. The initial language and the unwanted configurations language are sets of terms. Terms are words which are construct with symbols that have an arity that can be greater than 1. The rewrite system represent the dynamic of the complex system. It is a set of rules that permit from a initial term to obtain a new term. One of the approaches to analyze reachability from this modelling is to compute the set of reachable configurations. This set which is called set of descendants is obtained by applying the rewrite system on the initial language until obtaining no more new terms. After the set of descendants is computed, we need to compute the intersection between this set and the unwanted configurations set. If this intersection is empty then there is no unwanted configuration reachable, else the configurations in this intersection are reachable. However, the set of descendants is not computable in the general case. To bypass this problem, we compute an over-approximation of descendants. Now, if the intersection is empty, we keep proving that no unwanted configuration is reachable. Nevertheless, if the intersection is not empty, it is not possible to know if it comes from false-positives or form unwanted reachable configurations. So, the precision of the over-approximation is decisive.

Keywords : non regular tree languages, rewriting, rewriting strategies, reachability analysis

Centre Val de Loire UNIVERSITÉ
Spirit of the Loire Valley

LIFO
Bâtiment IIIA, Rue Léonard de Vinci
B.P. 6759
F-45067 ORLÉANS Cedex 2

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS