Département de formation doctorale en informatique **École doctorale IAE + M**
UFR STMIA

Contraintes d'ordre et automates d'arbres pour les preuves de terminaison

THÈSE

présentée et soutenue publiquement le 24 septembre 1998

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Thomas Genet

Composition du jury

Président :	D. Méry
Rapporteurs :	N. Dershowitz S. Tison
Examinateurs :	I. Gnaedig D. Hofbauer H. Kirchner

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Tout d'abord, je souhaite remercier mes deux directeurs de thèse Isabelle Gnaedig et Hélène Kirchner avec qui j'ai eu tant de plaisir à travailler et à collaborer.

C'est à toi, Isabelle, que je dois d'être réconcilié avec l'algèbre et c'est à toi que je dois de m'être inscrit en thèse. Cette idée saugrenue ne me serait jamais venue si tu n'avais su, dès le DEA, me communiquer ta passion pour la recherche en général et pour les preuves de terminaison en particulier. Toute la rigueur, en mathématiques comme en français, que le lecteur pourra trouver au fil de ces pages est le fruit de tes enseignements

C'est en grande partie à toi, Hélène, que je dois d'avoir achevé cette thèse. Je te remercie d'avoir toujours patiemment écouté et canalisé mes élucubrations pour en tirer l'essentiel. Merci de comprendre si vite ce que j'explique si mal! Enfin, je te remercie pour ta disponibilité constante et ce, malgré tes responsabilités croissantes au sein du projet PROTHEO.

Que ce travail soit, pour mon directeur de thèse bicéphale, le témoin de toute ma reconnaissance.

Je remercie Nachum Dershowitz pour les échanges que nous avons eu sur le *gpo* et pour l'intérêt qu'il a porté à ce travail. Je le remercie également d'avoir accepté de rapporter cette thèse malgré la barrière de la langue et les nombreux kilomètres qu'il a dû parcourir pour être présent à la soutenance.

Je remercie Sophie Tison d'avoir lu et disséqué cette thèse jusque dans les parties techniques. Je la remercie également d'avoir accepté la charge de rapporteur et d'être venue de Lille pour assister à la soutenance.

Je remercie Dominique Méry de s'être penché sur ce travail et d'en avoir donné un point de vue “vérification”. Je le remercie également d'avoir accepté de présider le jury de cette thèse.

Je remercie Dieter Hofbauer de s'être intéressé très tôt à ce travail et de l'avoir encouragé. Je le remercie également de s'être déplacé jusqu'à Nancy pour assister à la soutenance.

Je remercie toute l'équipe PROTHEO pour son accueil et sa bonne humeur, et en particulier Claude Kirchner qui a continuellement encouragé ce travail.

L'aide d'Eric Domenjoud et de Denis Rœgel en Latex et Metapost a été précieuse, je les remercie pour leur disponibilité et leur patience.

Je remercie mon fiston Maxime de m'avoir imposé une vie monastique et des horaires draconiens, Ô combien salvateur, pendant les quelques mois qu'a duré la rédaction de cette thèse.

Je remercie Valérie et toute ma famille d'avoir toujours été présents malgré l'éloignement.

Je remercie tous mes amis pour la musique, la piscine et les petites bouffes conviviales.

Enfin, à tous ceux qui ont goûté mon teurgoule, sincèrement, je demande pardon.

Table des matières

Table des figures

Chapitre 1

Introduction

Dans un monde où l'automatisation prend une part de plus en plus importante, s'assurer de la fiabilité des systèmes dits "critiques" devient indispensable. Qu'il s'agisse d'un programme de navigation aérienne ou d'un central téléphonique, les conséquences d'une erreur dans la conception ou la réalisation de tels systèmes sont incalculables. Pour *vérifier* un système automatique à risque, la solution communément utilisée consiste à tester tous ses comportements possibles et à vérifier qu'ils sont en accord avec le comportement attendu. Cependant, si le test est une solution satisfaisante pour garantir la fiabilité de systèmes pouvant avoir un nombre fini d'états, il est nécessairement incomplet, et donc incertain, dans le cas de systèmes ayant un nombre infini d'états. Ce qui est le cas pour beaucoup de processus automatiques et de programmes informatiques.

Il est donc nécessaire, dans ce cas, de faire appel à des outils de vérification basés sur des méthodes de preuve mathématique. Le plus souvent, la preuve n'est pas effectuée sur le système lui-même mais sur sa spécification formelle: un ensemble de formules mathématiques décrivant le comportement attendu du système avant même sa réalisation. Les *spécifications algébriques* sont un type de spécifications formelles dans lesquelles les propriétés attendues du système sont décrites à l'aide d'équations conditionnelles. Parmi les langages de spécification algébrique citons à titre d'exemple ASF+SDF [Kli93], Pluss [Gau84] et OBJ [GW88]. Sur une spécification algébrique, il est possible d'appliquer des méthodes de preuve afin de détecter des erreurs de conception ou, à l'inverse, de prouver automatiquement qu'elle vérifie certaines propriétés garantissant sa fiabilité. Beaucoup de méthodes de preuve utilisent les équations des spécifications algébriques sous une forme orientée. Une équation orientée est également nommée *règle de réécriture* et un ensemble de règles de réécriture est nommé *système de réécriture*. Pour toute équation, deux orientations sont possibles: de gauche à droite ou de droite à gauche. En général, l'orientation choisie pour les équations est celle qui assure la *terminaison* du système de réécriture correspondant, i.e. que toute suite de remplacement d'égaux par égaux aboutit en un nombre fini d'étapes à une expression dans laquelle plus aucun remplacement n'est possible. Cette expression est appelée *forme normale*.

Si l'implantation d'un système vérifie sa spécification formelle, alors elle *doit* également satisfaire les propriétés démontrées sur la spécification. Cependant, là encore l'adéquation entre l'implantation et sa spécification est difficile à montrer dans le cas de systèmes ayant un nombre infini d'états. En conséquence, même si certaines erreurs de conception peuvent être évitées, le système peut encore recéler des erreurs dues à une mauvaise implantation. Sur un programme, il est possible d'effectuer une étape de vérification supplémentaire qui est la preuve de propriétés du code. Parce qu'ils n'ont pas été prévus, dès le départ, pour ce genre de vérifications, les langages de programmation usuels tels C ou Pascal se prêtent assez mal aux preuves formelles. En revanche, dans les langages logiques tels Prolog ou dans les langages fonctionnels tels Lisp ou Caml, la logique sous-jacente autorise certaines preuves. La logique de réécriture est également un langage de programmation logique intéressant à plus d'un titre. Tout d'abord, la réécriture est une sémantique opérationnelle possible pour les spécifications algébriques. Il est ainsi possible d'obtenir un prototype exécutable directement à partir d'une spécification algébrique en orientant ses équations sous forme de règles. De plus, la logique de réécriture offre un cadre formel simple dont l'expressivité est suffisante aussi bien pour coder lisiblement des programmes que pour spécifier le comportement de processus automatiques concurrents et non-déterministes. Enfin, en codant un programme sous la forme d'un système de réécriture, il est possible d'exprimer formellement un grand nombre de propriétés attendues d'un programme et de les vérifier à l'aide de techniques de preuve développées en réécriture. Les propriétés essentielles sont la *terminaison* qui assure l'arrêt du programme en un temps fini, la *confluence* qui certifie que toute exécution d'un programme sur une même donnée aboutira au même résultat, la *complétude suffisante* qui, pour une définition par cas, établit que tous les cas possibles ont été pris en compte.

Parmi toutes ces propriétés, la terminaison est une propriété particulièrement importante puisque, d'une part, elle assure l'achèvement de tous les calculs en un temps fini, et d'autre part, elle est une prémisse indispensable à d'autres méthodes de preuve. Par exemple, la confluence d'un système de réécriture est une propriété indécidable en général, mais elle devient décidable si le système de réécriture termine. La terminaison des systèmes de réécriture est un problème indécidable [HL78, Dau89] mais, depuis les travaux fondateurs de Z. Manna et S. Ness [MN70], on dispose d'un critère de terminaison puissant basé sur les ordres bien fondés qui sont des ordres n'admettant pas de chaîne infinie décroissante. En montrant que la relation de réécriture est contenue dans un ordre bien fondé, tout calcul correspond à une chaîne décroissante et par conséquent est fini. Pour un système donné, la recherche d'un ordre bien fondé contenant la relation de réécriture nécessite, en général, une grande part d'expertise pour être menée à bien dans un délai raisonnable. De plus, si la recherche est facile sur des petits systèmes, elle devient complexe sur des systèmes de taille importante et son automatisation devient cruciale pour traiter des systèmes de réécriture de taille réelle. Or, l'automatisation de la recherche des ordres de terminaison est un problème doublement difficile. D'une part, il n'existe pas d'ordre unique capable de prouver la terminaison de tous les systèmes terminants mais des familles d'ordres plus ou moins adaptés suivant les systèmes. D'autre part, peu d'ordres utiles sont décidables et les ordres décidables ont, en général, une complexité exponentielle. En outre, il est difficile de réaliser une preuve de terminaison de façon incrémentale, la terminaison n'étant pas une propriété modulaire [Toy86] des systèmes de réécriture, i.e. l'union de deux systèmes de réécriture terminants n'est pas nécessairement un système terminant.

La motivation essentielle de cette recherche est de permettre la réalisation de preuves de terminaison sur des systèmes de réécriture représentant des programmes ou systèmes de taille réelle. Dans ce contexte, notre travail a porté sur deux points complémentaires qui sont: l'automatisation de la recherche d'ordres de terminaison et la mise au point d'un critère automatique assurant la terminaison d'un système modulaire à partir de la terminaison de ses modules. Sur ces deux points, les solutions adoptées utilisent la notion de contrainte.

La recherche d'un ordre de terminaison n'est pas un processus entièrement automatisable; les ordres de terminaison n'étant pas décidables en général. Cependant, nous verrons que l'utilisation de *contraintes d'ordre* dans un tel processus permet, successivement, de séparer les parties automatisables de la preuve de celles qui ne le sont pas, de résoudre efficacement les parties automatisables, et enfin, de ne soumettre à l'utilisateur *que* les parties non-automatisables.

Le critère de terminaison modulaire, quant à lui, est basé sur l'analyse des *formes normales* d'un ensemble de termes initiaux réduits successivement par chaque module du système de réécriture. Ceci n'est, a priori, réalisable que sur des ensembles finis de termes. Or, pour raisonner sur des ensembles infinis, nous proposons de les représenter à l'aide d'automates d'arbres et de réaliser l'analyse des formes normales à travers un mécanisme de résolution de *contraintes ensemblistes* sur les automates.

Dans le chapitre 2, nous présentons les définitions essentielles pour la lecture et la compréhension de cette thèse: alphabets, termes, systèmes de réécriture, ordres et automates d'arbres. Suite à la définition formelle de ces objets, dans le chapitre 3, nous tentons de donner une intuition plus informatique de ce qu'ils représentent en termes de programmes et de calculs. Dans ce même chapitre, nous présentons brièvement les propriétés et outils de preuve de la réécriture utilisés en vérification. Ensuite, les principaux ordres de terminaison utilisés dans cette thèse et la méthode des paires de dépendance de T. Arts et J. Giesl [Art96, AG97a, AG97b, Art97] sont présentés dans le chapitre 4. Dans ce chapitre, nous insistons tout particulièrement sur l'ordre général sur les chemins (*gpo*) conçu par N. Dershowitz et C. Hoot [DH95] qui généralise, dans un formalisme unique, les principaux ordres connus.

A partir du chapitre 5, nous arrivons à ce qui constitue l'apport original de cette thèse. Tout d'abord, en nous appuyant sur le *gpo*, nous proposons une nouvelle méthode pour automatiser la recherche d'ordre de terminaison basée sur un processus de résolution de contraintes d'ordre *gpo* [GG97a, GG97b]. Les deux grandes parties du chapitre 5 sont consacrées pour l'une à l'optimisation de la résolution d'un tel processus à l'aide de structures et d'algorithmes assurant un partage fort des contraintes et de leurs solutions, et pour l'autre à l'extraction automatique des parties difficiles de la preuve dans le cas d'une preuve semi-automatique.

Le chapitre 6 aborde l'étude du critère de terminaison modulaire. Ce critère est basé sur l'approximation des ensembles de descendants et de formes normales d'un système de réécriture et d'un ensemble de termes initiaux donnés [Gen98, Gen97b, Gen97a]. Une grande partie de ce chapitre est dédié à la construction d'un automate reconnaissant un sur-ensemble de l'ensemble des descendants. Viennent ensuite les applications parmi lesquelles figure le critère de terminaison modulaire. Les autres applications également dans le domaine de la vérification, sont le test d'atteignabilité, l'approximation des co-domaines de fonctions, la preuve de non-terminaison, et la preuve de complétude suffisante.

Nous présentons dans le chapitre 7, les deux prototypes réalisés et quelques exemples illustrant respectivement l'efficacité de notre procédure de recherche d'ordre par résolution de contrainte et la faisabilité des divers calculs d'approximation.

Dans le chapitre 8, en tenant compte des techniques de preuve existantes décrites dans le chapitre 4, et de nos propres travaux présentés dans les chapitres 5 et 6, nous donnons deux prolongements possibles de ce travail exploitant l'ensemble des résultats obtenus dans cette thèse. L'un est un nouveau critère de terminaison des systèmes de réécriture qui, en utilisant à la fois des contraintes d'ordre et des automates d'arbres, permet de capturer et d'exploiter les notions de stratégie d'application des règles et d'ensemble de termes initiaux. L'autre prolongement décrit l'architecture de ce que pourrait être un démonstrateur automatique dédié à la preuve de terminaison des systèmes de réécriture.

Enfin dans le chapitre 9, nous concluons sur ce travail en donnant d'autres pistes de recherche possibles et des extensions possibles des résultats présentés.

Chapitre 2

Préliminaires

Dans ce chapitre, nous donnons les définitions de base utilisées dans cette thèse. Pour plus de détails, le lecteur intéressé pourra se reporter aux documents de référence sur la réécriture [DJ90], sur la terminaison des systèmes de réécriture [Der87, Fer95], sur les propriétés modulaires des systèmes de réécriture [Mid90, Ohl94, Gra96] sur les automates et langages d'arbres [GS84, Jac96a, CDG$^+$97], sur les liens entre langages réguliers d'arbres et systèmes de réécriture [GT95, Jac96a].

2.1 Relations

Soit R une relation binaire sur un ensemble E.

- R est réflexive ssi $\forall a \in E$, $a \; R \; a$,
- R est irréflexive ssi $\forall a \in E$, $a \not R \; a$,
- R est symétrique ssi $\forall a, b \in E$, $a \; R \; b \implies b \; R \; a$,
- R est antisymétrique ssi $\forall a, b \in E$, $a \; R \; b \implies b \not R \; a$,
- R est transitive ssi $\forall a, b, c \in E$, $a \; R \; b$ et $b \; R \; c \implies a \; R \; c$,
- R est un ordre si R est transitive, irréflexive et antisymétrique,
- R est un préordre si R est transitive et réflexive,
- R est une relation d'équivalence si R est transitive, réflexive et symétrique.

La *clôture transitive* (resp. réflexive et transitive) de R, notée R^+ (resp. R^*), est la plus petite relation transitive (resp. réflexive et transitive) contenant R.

Définition 2.1 *(Relation totale) Une relation R sur un ensemble E est* totale *si pour tout couple d'éléments s, t de E on a $s = t$, $s \; R \; t$ ou $t \; R \; s$.*

2.2 Alphabets, termes et positions

Un *alphabet* (ou signature) $\mathcal{F}$ est un ensemble fini de symboles de fonction associé à une fonction d'*arité* notée ar, de $\mathcal{F}$ dans $\mathbb{N}$. Un symbole $f \in \mathcal{F}$ tel que $ar(f) = 0$ est appelé *constante.* On note $\mathcal{F} = \{f : 3, cons : 2, a : 0\}$, un alphabet dans lequel f est un symbole d'arité 3, $cons$ est un symbole d'arité 2 et a est une constante. Par commodité, au lieu de $f \in \mathcal{F}$ et $ar(f) = n$, on note parfois $f \in \mathcal{F}^n$. Soit $\mathcal{X}$ un ensemble dénombrable de symboles de variables. Dans la suite, on supposera toujours que $\mathcal{X}$ est disjoint de tous les alphabets considérés. L'ensemble $\mathcal{T}(\mathcal{F}, \mathcal{X})$ des *termes* sur $\mathcal{F}$ est défini récursivement par:

1. $X \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$, et
2. pour tout $f \in \mathcal{F}$ avec $ar(f) = n \in \mathbb{N}$, et tous termes $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, on a $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

Par commodité, une constante $c()$ est notée c. Un terme ne contenant aucune variable est dit *clos*, et l'ensemble des termes clos est noté $\mathcal{T}(\mathcal{F})$. L'ensemble des *positions* d'un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, noté $\mathcal{P}os(t)$, est un mot sur $\mathbb{N}$ défini inductivement par:

1. $\mathcal{P}os(t) = \{\epsilon\}$ si $t \in X$,
2. $\mathcal{P}os(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ et } p \in \mathcal{P}os(t_i)\}$ si $f \in \mathcal{F}^n$ et $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, où ϵ représente la suite d'entiers vide.

Les termes sont représentés par des arbres, étiquetés par des éléments de $\mathcal{F} \cup \mathcal{X}$, ainsi, le terme $f(t_1, \ldots, t_n)$ est représenté par l'arbre:

$$\begin{array}{ccc} & f & \\ / & & \backslash \\ t_1 & \cdots & t_n \end{array}$$

Si $p \in \mathcal{P}os(t)$ alors le *sous-terme* de t à la position p est noté $t|_p$. Les sous-termes $t|_p$ tels que $p \in \mathcal{P}os(t) \setminus \{\epsilon\}$ sont nommés *sous-termes stricts.* Le terme obtenu après remplacement dans t de $t|_p$ à la position p par le terme s est noté $t[s]_p$. Le symbole étiquetant le terme à la position ϵ est noté $\mathcal{R}oot(t)$. Pour tout terme $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, on note $\mathcal{P}os_{\mathcal{F}}(s)$ l'ensemble des positions fonctionnelles de s, qui est égal à $\{p \in \mathcal{P}os(s) \mid p \neq \epsilon \text{ et } \mathcal{R}oot(s|_p) \in \mathcal{F}\}$.

Exemple 2.1 *Soit $\mathcal{F} = \{f : 3,\ cons : 2,\ car : 1,\ a : 0,\ nil : 0\}$ un alphabet et $x \in \mathcal{X}$. Soient s et t les termes:*

- $s = f(a, x, nil) = \begin{array}{ccc} & f & \\ / & | & \backslash \\ a & x & nil \end{array}$, *et*
- $t = car(cons(a, cons(a, nil))) = \begin{array}{ccc} & car & \\ & | & \\ & cons & \\ / & & \backslash \\ a & & cons \\ & / & \backslash \\ & a & nil \end{array}$

On a $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, et $t \in \mathcal{T}(\mathcal{F})$. L'ensemble des positions de t est $\mathcal{P}os(t) = \{\epsilon, 1, 1.1, 1.2, 1.2.1, 1.2.2\}$, l'ensemble $\mathcal{P}os_{\mathcal{F}}(t)$ des positions fonctionnelles de t est tel que $\mathcal{P}os_{\mathcal{F}}(t) = \mathcal{P}os(t)$. En revanche, $\mathcal{P}os(s) \neq \mathcal{P}os_{\mathcal{F}}(s) = \{\epsilon, 1, 3\}$. Chaque position de $\mathcal{P}os(t)$ désigne un sous-terme unique de t; il est ainsi possible:

- *d'isoler un sous-terme précis, par exemple: $t|_{1.2} = cons(a, nil)$, $t|_{1.2.2} = nil$ et $t|_\epsilon = car(cons(a, cons(a, nil)))$,*

– *de remplacer un sous-terme par un autre, par exemple:* $t[s]_{1.1} = car(cons(f(a,x,nil), cons(a, nil)))$,

– *de connaître le symbole étiquetant un noeud, par exemple:* $\mathcal{R}oot(t|_\epsilon) = car$, $\mathcal{R}oot(t|_{1.2}) = cons$.

L'ensemble des variables d'un terme $s \in \mathcal{T}(\mathcal{F},\mathcal{X})$ est noté $\mathcal{V}ar(s)$. Un terme s est dit *linéaire* si chaque variable de $\mathcal{V}ar(s)$ a au plus une occurrence dans s. Une *substitution* est une application $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F},\mathcal{X})$ qui s'étend à $\mathcal{T}(\mathcal{F},\mathcal{X})$ en un endomorphisme: $\sigma\colon \mathcal{T}(\mathcal{F},\mathcal{X}) \mapsto \mathcal{T}(\mathcal{F},\mathcal{X})$. Le résultat de l'*application* d'une substitution σ à un terme $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ est le terme noté $t\sigma$.

Exemple 2.2 *Soit* $\mathcal{F} = \{f : 3, h : 1, g : 1, a : 0\}$, *soit* $\sigma = \{x \mapsto g(a), y \mapsto h(z)\}$ *et* $t = f(h(x), x, g(y))$. *On a* $t\sigma = f(h(g(a)), g(a), g(h(z)))$.

Définition 2.2 *Un contexte est un terme* $C[\,]$ *de* $\mathcal{T}(\mathcal{F}\cup\{\square\},\mathcal{X})$ *dans lequel le nouveau symbole de constante* $\square \notin \mathcal{F}$ *n'apparaît qu'une seule fois. Pour tout contexte* $C[\,]$ *et tout terme* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $C[t]$ *est le terme obtenu par remplacement de* $\square$ *par* t *dans* $C[\,]$.

Définition 2.3 *Un terme* $s = f(s_1, \ldots, s_n)$ *est plongé dans un terme* $t = g(t_1, \ldots, t_m)$ *si:*

– $f = g$ *et pour tout* i, $1 \leq i \leq n = m$, s_i *est plongé dans* t_i,

– *il existe un entier* k, $1 \leq k \leq m$ *tel que* s *est plongé dans* t_k.

Opérationnellement, pour déterminer si un terme s est plongé dans un terme t, il suffit de vérifier si s peut être obtenu en effaçant un nombre quelconque de symboles (éventuellement aucun) dans t. Par exemple, le terme x est plongé dans le terme $f(x)$, le terme $f(g(x))$ est plongé dans $f(h(g(k(x)), k(y)))$, $f(a)$ est plongé dans $f(a)$. Un terme s est *strictement plongé* dans un terme t si s est plongé dans t et $s \neq t$.

2.3 Systèmes de réécriture

Un *système de réécriture* est un couple $(\mathcal{F},\mathcal{R})$ où $\mathcal{F}$ est un alphabet et $\mathcal{R}$ est un ensemble de *règles de réécriture* $l \to r$, où $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $l \notin \mathcal{X}$, et $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. Dans la suite, un système de réécriture $(\mathcal{F},\mathcal{R})$ sera également noté $\mathcal{R}$, s'il ne peut y avoir de confusion sur $\mathcal{F}$.
Une règle de réécriture $l \to r$ est dite *linéaire à gauche* (resp. *linéaire à droite*) si le membre gauche (resp. le membre droit) de la règle est linéaire. Une règle est *linéaire* si elle est à la fois linéaire à gauche et à droite. Un système de réécriture $\mathcal{R}$ est linéaire (resp. linéaire à gauche, linéaire à droite) si toute règle $l \to r$ du système $\mathcal{R}$ est linéaire (resp. linéaire à gauche, linéaire à droite). La *relation de réécriture* $\to_{\mathcal{R}}$ induite par $\mathcal{R}$ est définie comme suit.

Définition 2.4 *(Relation de réécriture) Soit* $\mathcal{R}$ *un système de réécriture. Pour tous termes* $s, t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $s \to_{\mathcal{R}} t$ *s'il existe une règle de réécriture* $l \to r \in \mathcal{R}$, *une position* $p \in \mathcal{P}os(s)$ *et une substitution* σ *tels que* $l\sigma = s|_p$ *et* $t = s[r\sigma]_p$.

Le sous-terme $s|_p$ est nommé *redex*. La fermeture transitive (resp. réflexive transitive) de $\to_{\mathcal{R}}$ est notée $\to_{\mathcal{R}}{}^+$ (resp. $\to_{\mathcal{R}}{}^*$). Deux termes s et t sont joignables par $\mathcal{R}$, noté $s \downarrow_{\mathcal{R}} t$, s'il existe un terme v tel que $s \to_{\mathcal{R}}{}^* v$ et $t \to_{\mathcal{R}}{}^* v$. On dit qu'un terme s est *réductible* par $\mathcal{R}$ s'il existe un terme t tel que $s \to_R t$. On dira également que s est réduit en t. Le terme s est *irréductible* ou *en forme normale* s'il n'existe aucun terme t tel que $s \rightarrow_{\mathcal{R}} t$. L'ensemble des termes irréductibles par le système $\mathcal{R}$ est noté $IRR(\mathcal{R})$. Un terme s est *normalisable* en un terme t, noté $s \xrightarrow{!}_{\mathcal{R}} t$, si $s \rightarrow_{\mathcal{R}}^* t$ et $t \in IRR(\mathcal{R})$.

Exemple 2.3 *Voici un exemple simple de système de réécriture définissant la multiplication et l'addition sur les entiers où $\mathcal{F} = \{0 : 0, s : 1, + : 2, * : 2\}$, et $x, y \in \mathcal{X}$.*

$$\begin{array}{ll} (1) & 0 * x \rightarrow 0 \\ (2) & s(x) * y \rightarrow (x * y) + y \\ (3) & x + 0 \rightarrow x \\ (4) & x + s(y) \rightarrow s(x + y) \end{array}$$

Étant donné un système de réécriture $\mathcal{R}$ sur $\mathcal{F}$, il est possible de diviser $\mathcal{F}$ en deux sous-ensembles distincts: l'ensemble des symboles *définis* $\mathcal{D} = \{\mathcal{R}oot(l) \mid l \rightarrow r \in \mathcal{R}\}$ et l'ensemble des symboles constructeurs $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. Dans l'exemple 2.3, on a $\mathcal{C} = \{0, s\}$ et $\mathcal{D} = \{*, +\}$. Parmi les termes de $\mathcal{T}(\mathcal{F})$ on distingue les termes de $\mathcal{T}(\mathcal{C})$, dits *termes constructeurs*. Dans l'exemple précédent, $\mathcal{T}(\mathcal{C}) = \{0, s(0), s(s(0)), \dots\}$, qui est la représentation usuelle des entiers en termes de 0 et de successeurs. Voyons maintenant, quelques exemples de réécriture effectuées à l'aide du système de l'exemple 2.3.

Exemple 2.4 *Soit t le terme $s(s(0)) * s(s(0))$ et $\mathcal{R}$ le système de l'exemple 2.3. Sur t seule la règle (2) de $\mathcal{R}$ peut être appliquée: à la position ϵ de t on a $t = (s(x) * y)\sigma$ et $\sigma = \{x \mapsto s(0), y \mapsto s(s(0))\}$. Le redex est donc $t|_\epsilon$ soit t lui-même. Le terme t est réductible en $t_1 = t[((x * y) + y)\sigma]_\epsilon$, i.e. $t_1 = (s(0) * s(s(0))) + s(s(0))$. Nous noterons cette réduction de la façon suivante:*

$$\underline{s(s(0)) * s(s(0))} \rightarrow_{\mathcal{R}}^{(2)} (s(0) * s(s(0))) + s(s(0))$$

où le redex réduit à cette étape est souligné, et $\rightarrow_{\mathcal{R}}$ a pour exposant le numéro de la règle appliquée. Nous donnons maintenant la réécriture du terme t jusqu'à sa forme normale.

$$\begin{array}{l}
\underline{s(s(0)) * s(s(0))} \rightarrow_{\mathcal{R}}^{(2)} \\
(\underline{s(0) * s(s(0))}) + s(s(0)) \rightarrow_{\mathcal{R}}^{(2)} \\
((\underline{0 * s(s(0))}) + s(s(0))) + s(s(0)) \rightarrow_{\mathcal{R}}^{(1)} \\
\underline{(0 + s(s(0))) + s(s(0))} \rightarrow_{\mathcal{R}}^{(4)} \\
s(\underline{(0 + s(s(0))) + s(0)}) \rightarrow_{\mathcal{R}}^{(4)} \\
s(s(\underline{(0 + s(s(0))) + 0})) \rightarrow_{\mathcal{R}}^{(3)} \\
s(s(\underline{0 + s(s(0))})) \rightarrow_{\mathcal{R}}^{(4)} \\
s(s(s(\underline{0 + s(0)}))) \rightarrow_{\mathcal{R}}^{(4)} \\
s(s(s(s(\underline{0 + 0})))) \rightarrow_{\mathcal{R}}^{(3)} \\
s(s(s(s(0))))
\end{array}$$

La réécriture s'achève sur le terme $s(s(s(s(0))))$ en forme normale. On a donc $s(s(0)) * s(s(0)) \xrightarrow{!}_{\mathcal{R}} s(s(s(0)))$. Nous venons de montrer que $2 * 2 = 4$! Il faut remarquer que la réduction que nous venons de montrer n'est pas la seule possible. Par exemple, soit t_1 le premier terme obtenu par réécriture, i.e.

$$s(0) * s(s(0))) + s(s(0)).$$

Il existe plusieurs façons de réduire t_1: il existe dans le même terme deux redex à des positions différentes et réductibles par des règles distinctes. Le terme t_1 peut être réduit par la règle (4) à la position 1, comme précédemment, mais également par la règle (4) à la position ϵ, i.e.

$$\underline{(s(0) * s(s(0))) + s(s(0))} \rightarrow_{\mathcal{R}}^{(4)} s((s(0) * s(s(0))) + s(0))$$

Sur ce dernier terme, là encore plusieurs règles peuvent s'appliquer à des positions différentes.

La réécriture est également applicable à un ensemble de termes. Pour un système de réécriture $\mathcal{R}$ et un ensemble de termes E, nous définissons maintenant l'ensemble de $\mathcal{R}$-descendants de E, noté $\mathcal{R}^*(E)$, qui est l'ensemble des termes accessibles par $\rightarrow_{\mathcal{R}}{}^*$ à partir des termes de E, ainsi que l'ensemble des $\mathcal{R}$-formes normales de E, noté $\mathcal{R}^!(E)$, qui est l'ensemble des formes normales accessibles par $\rightarrow_{\mathcal{R}}{}^*$ à partir des termes de E.

Définition 2.5 *($\mathcal{R}$-descendants et $\mathcal{R}$-formes normales) Soit $\mathcal{R}$ un système de réécriture et E un ensemble (un langage) de termes.*

$$\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ t.q. } s \rightarrow_{\mathcal{R}}^* t\}.$$

$$\mathcal{R}^!(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ t.q. } s \xrightarrow{!}_{\mathcal{R}} t\}.$$

Les ensembles de descendants et les ensembles de formes normales sont liés par la proposition suivante.

Proposition 2.1 $\mathcal{R}^!(E) = \mathcal{R}^*(E) \cap IRR(\mathcal{R})$.

2.4 Ordres bien-fondés en réécriture

La propriété qui nous intéressera essentiellement tout au long de cette thèse est la terminaison des systèmes de réécriture.

Définition 2.6 *(Terminaison) Un système de réécriture $\mathcal{R}$* termine *si la relation de réécriture $\rightarrow_{\mathcal{R}}$ n'admet pas de chaîne infinie, i.e. s'il n'existe pas de suite infinie de termes $s_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $i \in \mathbb{N}$, tels que $s_0 \rightarrow_{\mathcal{R}} s_1 \rightarrow_{\mathcal{R}} \ldots$.*

La propriété de terminaison assure que toute chaîne de réécriture aboutit nécessairement à une forme normale. Il existe une autre notion de terminaison, plus faible, qui assure qu'à partir de tout terme, il existe *au moins* une chaîne aboutissant à une forme normale.

Définition 2.7 *(Terminaison faible) Un système de réécriture $\mathcal{R}$* termine faiblement *si pour tout terme $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, il existe $t \in IRR(\mathcal{R})$ tel que $s \rightarrow_{\mathcal{R}}{}^* t$.*

Le pouvoir d'expression des systèmes de réécriture étant comparable à celui des machines de Turing, la propriété de terminaison des systèmes de réécriture est indécidable et ce, même pour la classe très restreinte des systèmes linéaires à gauche ne comportant qu'une seule règle, comme l'a montré M. Dauchet [Dau89]. Cependant, et comme dans aucun autre modèle de calcul, il existe de nombreux critères de terminaison. Ces critères sont le plus souvent basés sur la recherche d'ordres bien fondés, définis ci-dessous, contenant la relation de réécriture sur les termes. Malgré la profusion de ces critères, la terminaison d'un système de réécriture reste une des propriétés les plus difficiles à établir pour un système de réécriture. Il y a deux raisons à ceci: d'une part, il n'existe pas d'ordre unique suffisamment puissant pour prouver la terminaison de tous les systèmes terminants. Pour mener à bien une preuve de terminaison, il est souvent nécessaire de connaître et de maîtriser une grande variété d'ordres et de méthodes. D'autre part, les critères de terminaison puissants sont indécidables ou de complexité exponentielle dans le meilleur des cas.

Dans cette section, nous rappelons les définitions et propriétés essentielles des ordres de terminaison: ordre bien-fondé, monotonie, stabilité par instanciation, ordre de simplification, ainsi que les théorèmes généraux liant les ordres et la propriété de terminaison.

2.4.1 Ordres bien fondés

La quasi-totalité des méthodes de preuve de terminaison en réécriture sont basées sur l'utilisation d'ordres. Nous donnons maintenant les définitions de base, empruntées à M. C. F. Ferreira [Fer95], concernant les relations particulières que sont les ordres. Une relation binaire R sur E est un *préordre* (ou *quasi-ordre*), souvent noté $\geq$, si elle est réflexive et transitive. On notera $\leq$ l'inverse de la relation $\geq$. Tout préordre $\geq$ contient deux relations particulières qui sont, un *ordre* (ou *ordre partiel*, *ordre strict*) `ord(` $\geq$ `)` correspondant à la relation $\geq \setminus \leq$, et une *relation d'équivalence* `eq(` $\geq$ `)` définie par la relation $\geq \cap \leq$. Par commodité, `ord(` $\geq$ `)` et `eq(` $\geq$ `)` seront également notés $>$ et $\simeq$, respectivement. L'ensemble $E/\simeq$ défini par $\{\langle s\rangle \mid s \in E\}$, où $\langle s\rangle$ est la *classe de $\simeq$-équivalence* de s, est appelé *quotient de E par $\simeq$* et on a $\langle s\rangle = \{t \in S \mid t \simeq s\}$. Tout ordre strict $>$ sur E, peut être transposé en un ordre $\sqsupset$ sur l'ensemble quotient par: $\langle s\rangle \sqsupset \langle t\rangle$ si $s > t$, où $s, t \in E$.

Définition 2.8 *(Ordre bien fondé) Un ordre $>$ est bien fondé (ou* nœthérien*) sur un ensemble E s'il n'existe pas de suite infiniment décroissante, i.e. s'il n'existe pas de suite infinie $s_0, s_1, \ldots$ d'éléments de E telle que $s_0 > s_1 > \ldots$.*

Exemple 2.5 *L'ordre usuel sur les nombres naturels, noté $>_{\mathbb{N}}$ est un ordre bien fondé sur $\mathbb{N}$. En effet, toutes les suites de naturels décroissantes sont finies, puisque bornées par 0. En revanche, l'ordre usuel sur les entiers relatifs, noté $>_{\mathbb{Z}}$, n'est pas bien fondé sur $\mathbb{Z}$. On peut également remarquer que l'ordre $>_{\mathbb{N}}$ est total puisque pour tout couple d'entiers naturels $s, t \in \mathbb{N}$, on a $s = t$, $s >_{\mathbb{N}} t$, ou $t >_{\mathbb{N}} s$. Il en est de même pour $>_{\mathbb{Z}}$.*

Dans cette thèse, la majeure partie des ordres que nous utilisons en pratique seront en fait des préordres bien fondés. un préordre $\geq$ est bien-fondé si son ordre strict `ord(` $\geq$ `)` l'est. Ceci revient à définir un ordre bien fondé sur les classes d'équivalences.

Définition 2.9 *(Préordre bien fondé) Soit $\geq$ un préordre sur un ensemble E. Soit $\sqsupset$ la transposition de* `ord(` $\geq$ `)` *à l'ensemble des classes d'équivalences de E, i.e. $E/\simeq$. Le préordre $\geq$ est bien fondé si $\sqsupset$ est un ordre bien fondé sur $E/\simeq$.*

Exemple 2.6 *Soit le préordre $\succeq_{10}$ sur les entiers naturels tel que pour tout $s, t \in \mathbb{N}$, on a $s \succeq_{10} t$ si $\lfloor \frac{s}{10} \rfloor \geq_{\mathbb{N}} \lfloor \frac{t}{10} \rfloor$, où pour tout nombre rationnel n, $\lfloor n \rfloor$ représente la partie entière de n. Soit $\simeq_{10}$ la relation d'équivalence* `eq(` $\succeq_{10}$ `)`, *$\succ_{10}$ l'ordre strict* `ord(` $\succeq_{10}$ `)`, *et $\sqsupset_{10}$ la transposition de $\succ_{10}$ aux classes d'équivalences de $\mathbb{N}/\simeq_{10}$. Clairement, l'ordre $\succ_{10}$ compare les entiers en faisant abstraction des unités. Tous les entiers ne différant que par les unités sont donc dans la même* *classe d'équivalence. On a par exemple,* $1 \simeq_{10} 2$*,* $2 \simeq_{10} 9$*,* $12 \succ_{10} 2$*, etc. Toute suite d'entiers* $s_1, s_2, \ldots$ *telle que* $\langle s_1 \rangle \sqsupset_{10} \langle s_2 \rangle \sqsupset_{10} \ldots$ *est nécessairement bornée par* $\langle 0 \rangle = \{0, \ldots, 9\}$*, et est donc finie. L'ordre* $\sqsupset_{10}$ *est donc bien fondé sur* $\mathbb{N}/\simeq_{10}$*, donc par définition,* $\succeq_{10}$ *est un préordre bien fondé sur* $\mathbb{N}$*.*

2.4.2 Ordres bien fondés sur les termes

La proposition suivante constitue la base des critères de terminaison des systèmes de réécriture.

Proposition 2.2 *[MN70]. Un système de réécriture* $\mathcal{R}$ *sur* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *termine si et seulement s'il existe un ordre* $>$ *bien fondé sur* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *tel que pour tous termes* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*,* $s \rightarrow_{\mathcal{R}} t$ *implique* $s > t$*.*

Pour tout système de réécriture non-vide, il existe un nombre infini de couples s, t tels que $s \rightarrow_{\mathcal{R}} t$. Etablir la terminaison revient donc à effectuer un nombre infini de vérifications. Afin de contourner ce problème, l'approche couramment retenue utilise des ordres bien fondés ayant des propriétés supplémentaires limitant le nombre de vérifications à effectuer à un nombre fini. Les principaux ordres de terminaison utilisés n'exigent, en fait, que $l > r$ pour toute règle $l \rightarrow r$ du système $\mathcal{R}$. Nous verrons dans la suite que ces conditions peuvent encore être affaiblies, grâce notamment à la méthode des paires de dépendance, voir section 4.3, ou à l'ordre général sur les chemins, voir section 4.4. Nous définissons maintenant les propriétés essentielles des ordres et des préordres, en considérant une relation binaire R quelconque. Par commodité, pour tous termes $s, t, s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $[f(s_1, \ldots, s_i, s, s_{i+1}, \ldots, s_n) \; R \; f(s_1, \ldots, s_i, t, s_{i+1}, \ldots, s_n)$ pour $f \in \mathcal{F}^n$ et $1 \leq i \leq n]$, sera abrégé en $f(\ldots, s, \ldots) \; R \; f(\ldots, t, \ldots)$.

Définition 2.10 *Soient* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *des termes et* R *une relation binaire sur* $\mathcal{T}(\mathcal{F}, \mathcal{X})$*.*

- *La relation* R *est dite* stable par instanciation *si pour tous termes* s, t*, et pour toute substitution* σ*,* $s \; R \; t \implies s\sigma \; R \; t\sigma$*.*
- *La relation* R *est dite* monotone *si pour tous termes* s, t*, et pour tout symbole de fonction* $f \in \mathcal{F}$*,* $s \; R \; t \implies f(\ldots, s, \ldots) \; R \; f(\ldots, t, \ldots)$*.*

Tout ordre (ou préordre) bien fondé, monotone et stable par instanciation est nommé *ordre* (ou *préordre*) *de réduction*. Stabilité et monotonie suffisent à ramener le problème général de la comparaison des termes s et t à chaque étape de réécriture $s \rightarrow_{\mathcal{R}} t$ au problème (décidable dans le cas où $\mathcal{R}$ est fini et l'ordre est décidable) de la comparaison des termes l et r pour chaque règle de réécriture $l \rightarrow r$ de $\mathcal{R}$.

Proposition 2.3 *[MN70] Un système de réécriture* $\mathcal{R}$ *sur* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *termine si et seulement s'il existe un ordre* $>$ *de réduction sur* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *tel que* $l > r$ *pour chaque règle* $l \rightarrow r$ *de* $\mathcal{R}$*.*

Cependant, montrer qu'un ordre sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est bien fondé est une tâche difficile. Pour pallier ceci, il existe deux solutions. La première solution consiste à ramener le problème d'orientation sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ à un problème équivalent sur un autre domaine où il existe un ordre bien fondé connu, par exemple l'ordre $>_{\mathbb{N}}$ sur $\mathbb{N}$. Cette approche, dite par interprétation, sera brièvement présentée dans la section 4.2. Une deuxième solution consiste à imposer une propriété supplémentaire sur l'ordre de façon à ce qu'il soit toujours, trivialement bien fondé: c'est la propriété de sous-terme qui assure que tout terme est plus grand que n'importe lequel de ses sous-termes.

Définition 2.11 *Soient* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *des termes et* $>$ *un ordre sur* $\mathcal{T}(\mathcal{F}, \mathcal{X})$*. L'ordre* $>$ a la propriété de sous-terme *si pour tout sous-terme strict* t *de* s *on a* $s > t$*.*

Le théorème de Kruskal [Kru60] établit que tout ordre monotone ayant la propriété de sous terme est bien fondé. Ainsi, tout ordre monotone, stable par instanciation et ayant la propriété de sous-terme est bien fondé et est donc un ordre de réduction. Un tel ordre est nommé *ordre de simplification*.

Définition 2.12 *(Ordre de simplification) Un ordre sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est un ordre de simplification s'il est monotone, stable par instanciation, et s'il a la propriété de sous-terme.*

La proposition liant la terminaison d'un système de réécriture $\mathcal{R}$ et l'existence d'un ordre de simplification est la suivante.

Proposition 2.4 *Un système de réécriture $\mathcal{R}$ sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ termine s'il existe un ordre de simplification $>$ tel que l'on ait $l > r$ pour toute règle $l \to r$ de $\mathcal{R}$.*

Cependant, il faut remarquer deux choses. Premièrement, avec cette dernière proposition, nous sommes passés d'une condition nécessaire et suffisante de terminaison (proposition 2.3), à une condition suffisante. Ceci est dû au fait que la classe des ordres de simplification n'est qu'une sous-classe des ordres de réduction. Deuxièmement, même si le critère de terminaison est simplifié, le problème de l'existence d'un ordre de simplification $>$ tel que $l > r$ pour toute règle $l \to r$ d'un système $\mathcal{R}$ reste indécidable [MG94]. Nous verrons, dans le chapitre 4, des ordres de simplification particuliers, pour lesquels ce problème devient décidable.

2.5 Automates d'arbres et langages réguliers

Les automates d'arbres et les grammaires d'arbres sont deux formalismes permettant de finiment représenter des ensembles réguliers d'arbres, et donc de termes. Nous utilisons principalement des automates d'arbres ascendants non-déterministes, dont les fonctionnalités de *reconnaisseurs* seront utiles dans la deuxième partie de ce travail.

2.5.1 Définitions

Soit $\mathcal{F}$ un alphabet et $\mathcal{Q}$ un ensemble fini de symboles d'arité 0, appelés *états*. L'ensemble des *configurations* $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ est défini inductivement par:

1. $\mathcal{Q} \subseteq \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$,
2. si $f \in \mathcal{F}$, $ar(f) = n$, et $s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ alors $f(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$.

Une *transition* est une règle de réécriture $c \to q$, où $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q \in \mathcal{Q}$. Une *transition normalisée* est une transition $c \to q$ où $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$, $ar(f) = n$, et $q_1, \ldots, q_n \in \mathcal{Q}$. Un automate d'arbres ascendant non-déterministe (ou simplement automate d'arbres) est un quadruplet $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, où $\mathcal{Q}_f$ est un ensemble d'états finaux et Δ est un ensemble de transitions normalisées. . La relation de réécriture induite par Δ sera notée $\to_\Delta$. Un terme t est reconnu par un état $q \in \mathcal{Q}$ si $t \to_\Delta^* q$. Un terme est reconnu par $\mathcal{A}$ si t est reconnu par un état final de $\mathcal{Q}_f$. Le langage régulier d'arbres reconnu par $\mathcal{A}$, est $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists q \in \mathcal{Q}_f \text{ t.q. } t \to_\Delta^* q\}$. Le langage régulier d'arbres reconnu par l'état q dans l'automate $\mathcal{A}$, est $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_\Delta^* q\}$. Un langage d'arbres E est *régulier* si il existe un automate d'arbres $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = E$. Un automate d'arbres $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ est *déterministe* si pour tout terme $t \in \mathcal{T}(\mathcal{F})$ il existe au plus un état $q \in Q$ tel que $t \to_\Delta^* q$. Pour nos besoins, nous utiliserons essentiellement des automates non-déterministes. Cependant, tout automate non-déterministe peut être ramené à un automate déterministe par le biais d'un algorithme de déterminisation, dont le principe est donné dans [CDG+97]. Pour tout ensemble de transitions Δ, on notera $etats(\Delta)$ l'ensemble des états apparaissant dans les transitions de Δ, i.e. $etats(\emptyset) = \emptyset$ et $etats(\{f(q_1, \ldots, q_n) \rightarrow q\} \cup \Delta) = \{q_1, \ldots, q_n, q\} \cup etats(\Delta)$.

Nous donnons quatre exemples illustrant le pouvoir d'expression des automates d'arbres et montrant tour à tour que les automates d'arbres sont un outil simple permettant de représenter de façon compacte un ensemble infini d'objets ayant une structure précise (exemple 2.7), tout en permettant d'exprimer des contraintes finies de numération (exemple 2.8), ou des motifs (exemple 2.9). Cependant, les automates sont incapables de représenter un ensemble non-régulier (exemple 2.10).

Exemple 2.7 *Nous donnons ici l'exemple de l'automate d'arbres A_0 reconnaissant l'ensemble des "listes plates non-vides composées de a et de b". On a $A_0 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, où $\mathcal{F} = \{cons : 2, a : 0, b : 0, nil : 0\}$, $\mathcal{Q} = \{q_0, q_1, q_2\}$, $\mathcal{Q}_f = \{q_0\}$ et $\Delta = \{$*

$$\begin{array}{lr}
(1) & cons(q_2, q_1) \rightarrow q_0 \\
(2) & nil \rightarrow q_1 \\
(3) & cons(q_2, q_1) \rightarrow q_1 \\
(4) & a \rightarrow q_2 \\
(5) & b \rightarrow q_2 \ \}
\end{array}$$

Dans cet exemple, on a $\mathcal{L}(A_0, q_2) = \{a, b\}$, $\mathcal{L}(A_0, q_1) = \{nil, cons(x, y) \mid x \in \mathcal{L}(A_0, q_2)$ et $y \in \mathcal{L}(A_0, q_1)\}$, ou encore l'ensemble

$$\{nil, cons(a, nil), cons(b, nil), cons(a, cons(a, nil)), cons(a, cons(b, nil)), \ldots\},$$

ou en utilisant la représentation usuelle des listes dans les langages fonctionnels ou logiques $\{[], [a], [b], [a, b], [a, a], [b, a, b, b] \ldots\}$, c'est à dire l'ensemble des listes plates formées de "a" et de "b". Enfin, $\mathcal{L}(A_0, q_0) = \mathcal{L}(A_0) = \{cons(x, y) \mid x \in \mathcal{L}(A_0, q_2)$ et $y \in \mathcal{L}(A_0, q_1)\}$, c'est à dire l'ensembles des listes plates constituées de a et de b, non-vides.

Montrons comment le terme $cons(b, cons(a, nil))$ est reconnu par l'automate A_0. Par commodité, les transitions sont numérotées et le redex en passe d'être réduit par une transition de Δ est souligné:

$$\begin{array}{l}
cons(b, cons(\underline{a}, nil)) \rightarrow_{\Delta}^{(4)} \\
cons(b, cons(q_2, \underline{nil})) \rightarrow_{\Delta}^{(2)} \\
cons(b, \underline{cons(q_2, q_1)}) \rightarrow_{\Delta}^{(3)} \\
cons(\underline{b}, q_1) \rightarrow_{\Delta}^{(5)} \\
\underline{cons(q_2, q_1)} \rightarrow_{\Delta}^{(1)} q_0.
\end{array}$$

Comme $q_0 \in \mathcal{Q}_f$, le terme est reconnu par l'automate. On peut remarquer que dans cet exemple la dérivation n'est pas unique et peut mener vers un terme irréductible ou vers un état n'appartenant pas à $\mathcal{Q}_f$.

Exemple 2.8 *Nous donnons ici un autre exemple d'automate d'arbres: A_1 reconnaissant l'ensemble des "listes quelconques composées de a et de b dont le nombre d'élément n'excède pas 3". Le terme "listes quelconques" signifie simplement que les listes peuvent être imbriquées. On a $A_1 = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$, où $\mathcal{F} = \{cons : 2, a : 0, b : 0, nil : 0\}$, $\mathcal{Q}' = \{q_0, q_1, q_2, q_3, q_4\}$, $\mathcal{Q}'_f = \{q_0\}$ et*

$\Delta' = \{$

$$
\begin{aligned}
nil &\to q_0\\
cons(q_1, q_2) &\to q_0\\
nil &\to q_1\\
cons(q_1, q_2) &\to q_1\\
a &\to q_1\\
b &\to q_1\\
nil &\to q_2\\
cons(q_1, q_3) &\to q_2\\
nil &\to q_3\\
cons(q_1, q_4) &\to q_3\\
nil &\to q_4 \ \}
\end{aligned}
$$

Dans cet exemple, $\mathcal{L}(A_1, q_1)$ est l'ensemble des éléments que l'on peut trouver dans une liste: soit un a, soit un b soit une liste quelconque composée de a et de b n'excédant pas 3 éléments. Dans $\mathcal{L}(A_1, q_2)$ on trouve toutes les listes quelconques composées de a et de b n'excédant pas 2 éléments. Enfin dans $\mathcal{L}(A_1, q_3)$ on trouve des listes quelconques composées de a et de b n'excédant pas 1 élément.

Exemple 2.9 *Voici un automate A_2 reconnaissant l'ensemble des "listes quelconques composées de a et de b et ordonnées". Par "liste ordonnée" on entendra ici que les "a", s'il y en a, sont placés avant les "b" et les "b", s'il y en a, sont placés avant les listes imbriquées, elles-aussi ordonnées. On a $A_2 = \langle \mathcal{F}, \mathcal{Q}'', \mathcal{Q}''_f, \Delta'' \rangle$, où $\mathcal{F} = \{cons : 2, a : 0, b : 0, nil : 0\}$, $\mathcal{Q}'' = \{q_0, q_1, q_2, q_3, q_4\}$, $\mathcal{Q}''_f = \{q_0\}$ et $\Delta'' = \{$*

$$
\begin{aligned}
nil &\to q_0\\
cons(q_3, q_0) &\to q_0\\
cons(q_4, q_1) &\to q_0\\
cons(q_0, q_2) &\to q_0\\
\\
nil &\to q_1\\
cons(q_4, q_1) &\to q_1\\
cons(q_0, q_2) &\to q_1\\
\\
nil &\to q_2\\
cons(q_0, q_2) &\to q_2\\
\\
a &\to q_3\\
b &\to q_4 \ \}
\end{aligned}
$$

Dans l'automate A_2, si on étudie les transitions aboutissant à l'état q, on obtient que

- *q_3 reconnaît exactement le terme a,*
- *q_4 reconnaît exactement le terme b,*
- *q_2 reconnaît la liste vide nil ou toute liste dont le premier élément est reconnaissable par q_0 et dont le reste est reconnaissable par q_2, i.e. q_2 reconnaît toute liste (éventuellement vide) constituée uniquement d'élément reconnaissables par q_0,*

- q_1 *reconnaît toute liste vide nil ou toute liste dont le premier élément est un b (reconnu par* q_4*) et dont le reste est reconnaissable par* q_1*, ou une liste semblable à celles qui sont reconnues par* q_2*. En clair,* q_1 *reconnaît toute liste constituée d'un nombre quelconque de b suivis d'une liste reconnaissable par* q_2*,*

- q_0 *reconnaît toute liste constituée d'un nombre quelconque de a suivis d'un nombre quelconque de b (reconnus par* q_1*) et d'un nombre quelconque de listes imbriquées du même type (reconnues par* q_2*).*

A titre d'exemple, voici quelques-unes des listes reconnues, données en utilisant la représentation usuelle des listes dans les langages fonctionnels ou logiques:
$[], [a], [b], [a, []], [[a]], [a, a, a, b, b, [a, b]], [a, a, b, [a, a, b[a, a, b, [...]]]], \ldots,$
en revanche, les listes suivantes ne sont pas reconnues, elles ne font pas partie de $\mathcal{L}(A_2)$*:*
$[a, b, a], [[], b], [a, b, [b, a], [a, b]], \ldots$

Exemple 2.10 *L'ensemble des arbres binaires équilibrés, i.e. le langage de la forme* $\{a,\ f(a,a),\ f(f(a,a),\ f(a,a)),\ \ldots\}$ *n'est pas reconnaissable par un automate d'arbres. En revanche, de tels ensembles sont reconnaissables par des automates plus évolués tels que les automates de la classe Rateg définis par M. Dauchet et J. Mongy [Mon81] ou les automates à contraintes définis par B. Bogaert et S. Tison [BT91, BT92]. Dans les automates de la classe Rateg, les états sont des symboles de fonction d'arité 1 et les règles de transitions sont plus générales et permettent notamment de tester l'égalité entre deux termes. A titre d'exemple, voici un automate de la classe Rateg reconnaissant le langage des arbres binaires équilibré:* $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ *où* $\mathcal{F} = \{f : 2, a : 0\}$*,* $\mathcal{Q} = \{q_0\}$*,* $\mathcal{Q}_f = \{q_0\}$ *et* $\Delta = \{f(q_0(x), q_0(x)) \to q_0(f(x,x)), a \to q_0\}$*.*

2.5.2 Opérations sur les langages

Les principales opérations sur les langages réguliers sont les opérations ensemblistes habituelles: l'appartenance $\in$, l'inclusion $\subset$, l'union $\cup$, l'intersection $\cap$, le complément d'un langage A noté $\overline{A}$, et la différence entre deux langages A et B notée $A \setminus B$. Nous donnons la propriété essentielle des langage régulier pour les opérations ensemblistes de base.

Proposition 2.5 *La classe des langages réguliers est fermée par les opérations d'union, d'intersection, de complément, et de différence. L'appartenance, le vide et l'inclusion sont décidables.*

La proposition précédente signifie que le résultat des opérations d'union, d'intersection, de complément et de différence, appliquées à des langages réguliers, est un langage régulier, et donc reconnaissable. Les algorithmes permettant de décider si un terme donné appartient à un langage régulier, si un langage régulier est vide, ou si un langage régulier est inclus dans un autre sont donnés dans la section suivante.

2.5.3 Algorithmes

Dans cette section, nous nous intéressons plus particulièrement au cas des ensembles réguliers de termes, et nous rappelons les algorithmes classiques associés aux opérations présentées dans la section précédente. Ces algorithmes permettent notamment de construire des automates reconnaissant l'union, l'intersection, le complément, et la soustraction de langages réguliers reconnus par des automates. Nous présentons également l'algorithme classique de décision du vide, basé sur le nettoyage de l'automate. Pour l'algorithme de déterminisation d'un automate, nous renvoyons le lecteur à [CDG$^+$97].

Avant toute chose, nous donnons l'algorithme classique permettant de ramener tout ensemble de transitions Δ à un ensemble de transitions normalisées Δ' reconnaissant le même langage, i.e. pour tout terme $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et tout état $q \in \mathcal{Q}$, $s \rightarrow^*_{\Delta'} q$ si et seulement si $s \rightarrow^*_{\Delta} q$.

Définition 2.13 *Soit $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres, $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, et $q \in \mathcal{Q}$. L'ensemble $Norm(s \rightarrow q)$ est défini par:*

- *si $s = q \in \mathcal{Q}$, alors $Norm(s \rightarrow q) = \emptyset$,*
- *si $s \in \mathcal{Q}$ et $s \neq q$, alors $Norm(s \rightarrow q) = \{s \rightarrow q\}$,*
- *si $s = f(t_1, \ldots, t_n)$ avec $f \in \mathcal{F}^n$, alors $Norm(s \rightarrow q) = \{f(q_1, \ldots, q_n) \rightarrow q\} \cup \bigcup_{i=1}^{n} Norm(t_i \rightarrow q_i)$, où $q_i = t_i$ si $t_i \in \mathcal{Q}$ et q_i est un nouvel état, sinon.*

Exemple 2.11 *Soit $F = \{f, g, a\}$ et $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, où $\mathcal{Q} = \{q_0, q_1\}$, $\mathcal{Q}_f = \{q_0\}$, et $\Delta = \{f(q_1) \rightarrow q_0, g(q_1, q_1) \rightarrow q_1, a \rightarrow q_1\}$.*

$Norm(q_1 \rightarrow q_0) = \{q_1 \rightarrow q_0\}$. $Norm(f(g(q_1, f(a))) \rightarrow q_0) = \{f(q_2) \rightarrow q_0, g(q_1, q_3) \rightarrow q_2, f(q_4) \rightarrow q_3, a \rightarrow q_4\}$, où q_2, q_3, q_4 sont des nouveaux états.

Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}_{\mathcal{A}}, \mathcal{Q}^f_{\mathcal{A}}, \Delta_{\mathcal{A}} \rangle$ et $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}_{\mathcal{B}}, \mathcal{Q}^f_{\mathcal{B}}, \Delta_{\mathcal{B}} \rangle$ deux automates d'arbres sur la même signature $\mathcal{F}$. Nous donnons maintenant les algorithmes de construction des automates $\mathcal{A} \cup \mathcal{B}$ et $\mathcal{A} \cap \mathcal{B}$, tels que $\mathcal{L}(\mathcal{A} \cup \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$ et $\mathcal{L}(\mathcal{A} \cap \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$.

Union $\mathcal{A} \cup \mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_{\mathcal{A}} \times \Delta_{\mathcal{B}} \rangle$, où $\mathcal{Q} = \mathcal{Q}_{\mathcal{A}} \times \mathcal{Q}_{\mathcal{B}}$, $\mathcal{Q}_f = (\mathcal{Q}^f_{\mathcal{A}} \times \mathcal{Q}_{\mathcal{B}}) \cup (\mathcal{Q}_{\mathcal{A}} \times \mathcal{Q}^f_{\mathcal{B}})$, et $\Delta_{\mathcal{A}} \times \Delta_{\mathcal{B}}$ est l'ensemble:

$$\{f((q_1, q'_1), \ldots, (q_n, q'_n)) \rightarrow (q, q') \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta_{\mathcal{A}} \text{ et } f(q'_1, \ldots, q'_n) \rightarrow q' \in \Delta_{\mathcal{B}}\}.$$

Intersection $\mathcal{A} \cap \mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_{\mathcal{A}} \times \Delta_{\mathcal{B}} \rangle$, où $\mathcal{Q} = \mathcal{Q}_{\mathcal{A}} \times \mathcal{Q}_{\mathcal{B}}$ et $\mathcal{Q}_f = \mathcal{Q}^f_{\mathcal{A}} \times \mathcal{Q}^f_{\mathcal{B}}$.

Les algorithmes de test d'inclusion et de soustraction de langages réguliers que nous donnons ici sont basés sur le calcul du complément qui est lui-même basé sur un algorithme de déterminisation. Un tel algorithme peut être trouvé par exemple dans [CDG+97].

Complément L'automate $\overline{\mathcal{A}}$ complément de $\mathcal{A}$ est obtenu par déterminisation de $\mathcal{A}$ en $\mathcal{A}_d = \langle \mathcal{F}, \mathcal{Q}_d, \mathcal{Q}_{fd}, \Delta_d \rangle$, puis $\overline{\mathcal{A}} = \langle \mathcal{F}, \mathcal{Q}_d, \mathcal{Q}_d \setminus \mathcal{Q}_{fd}, \Delta_d \rangle$.

Soustraction $\mathcal{A} \setminus \mathcal{B} = \mathcal{A} \cap \overline{\mathcal{B}}$.

Décision de l'inclusion On a $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ si et seulement si $\mathcal{L}(A \cap \overline{\mathcal{B}}) = \emptyset$.

Nous donnons maintenant plusieurs algorithmes indispensables à la manipulation pratique des automates, dont deux algorithmes de nettoyage, et un algorithme de décision du vide. Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$,

Calcul de l'ensemble des états accessibles L'ensemble S des états accessibles est défini par $S = \{q \mid \exists s \in \mathcal{T}(\mathcal{F}) \text{ t.q. } s \rightarrow^*_{\Delta} q\}$ (voir exemple 2.12). La construction de cet ensemble peut être effectuée à l'aide de l'algorithme suivant, exprimé par une unique règle de transition s'appliquant sur un ensemble S initialisé à vide, comme dans [Jac96a]:

$$\frac{\{q_1, \dots, q_n\} \cup S}{\{q_1, \dots, q_n\} \cup \{q\} \cup S}$$

si $q \notin \{q_1, \dots, q_n\} \cup S$ et il existe $f(q_1, \dots, q_n) \to q \in \Delta$.

Lorsque la règle ne s'applique plus, S contient l'ensemble des états accessibles.

Nettoyage d'automate par test d'accessibilité Si un état n'est pas accessible, cela signifie qu'il ne reconnaît aucun terme; il est donc inutile. De plus, toute transition y menant peut être supprimée. Pour nettoyer $\mathcal{A}$ de tout état et de toute transition inutile, il suffit supprimer de $\mathcal{Q}$ tout état inaccessible et de supprimer récursivement de Δ toute transition $f(q_1, \dots, q_n) \to q$ telle que q est un état inaccessible.

Décision du vide Pour la décision du vide, la méthode théorique nommée lemme de pompage (voir par exemple [CDG+97, Jac96a]) n'est pas utilisable en pratique. En revanche il existe d'autres algorithmes à la fois plus simples et plus efficaces, tel l'algorithme utilisant le calcul des états accessibles: le langage reconnu par $\mathcal{A}$ est vide si et seulement si l'ensemble des états accessibles de $\mathcal{A}$ ne contient aucun état final.

Calcul de l'ensemble des états utiles Un état q est dit *utile* s'il apparaît dans une réécriture menant à un état final, i.e. s'il existe une configuration c, une position p de c et un état final q' tels que $c|_p = q$ et $c \to^*_\Delta q'$. Un état accessible n'est pas nécessairement utile et vice-versa (voir exemple 2.12). L'ensemble S' des états utiles est calculé par application de la règle de transition suivante sur un ensemble S' initialisé à $\mathcal{Q}_f$:

$$\frac{\{q\} \cup S'}{\{q_1, \dots, q_n\} \cup \{q\} \cup S'}$$

s'il existe $f(q_1, \dots, q_n) \to q \in \Delta$ et s'il existe $i \in \{1, \dots, n\}$ tel que $q_i \notin \{q\} \cup S'$.

Nettoyage par test d'utilité Comme dans le cas du nettoyage par test d'accessibilité, tout état de $\mathcal{Q}$ inutile peut être supprimé et il en est de même pour toute transition menant à un état inutile.

Exemple 2.12 *Soit* $\mathcal{F} = \{f : 2, a : 0, b : 0\}$ *et* A *l'automate* $A = \langle \mathcal{F}, Q, \mathcal{Q}_f, \Delta \rangle$ *tel que* $\mathcal{Q} = \{q_0, q_1, q_2\}$*,* $\mathcal{Q}_f = \{q_0\}$*, et* $\Delta = \{f(q_1, q_0) \to q_0,\ a \to q_1,\ b \to q_2\}$*. L'ensemble des états accessibles est* $\{q_1, q_2\}$*, et l'ensemble des états utiles est* $\{q_1, q_0\}$*. Le langage reconnu par l'automate* A *est vide. Ceci peut être facilement déduit du fait qu'aucun état final n'est accessible.*

Chapitre 3

Programmes et preuves en réécriture

Dans ce chapitre, nous allons voir le lien étroit qui existe entre la réécriture et la programmation: comment programmer en logique de réécriture, et comment vérifier les programmes en démontrant des propriétés sur des systèmes de réécriture.

Jusqu'ici, nous avons présenté des systèmes d'équations et des systèmes de réécriture spécifiant des problèmes. Dans la section 3.1, nous allons voir comment rendre ces spécifications exécutables et montrer en quoi la réécriture est un mécanisme d'évaluation simple et efficace. Les systèmes de réécriture tels qu'ils ont été définis dans la section 2.3 sont suffisants pour programmer tout ce qui peut l'être puisqu'ils ont la puissance des machines de Turing [Dau89]. Cependant, pour être réellement utilisée à des fins de spécification et de programmation, la réécriture doit subir deux extensions. D'abord dans la section 3.2.1, nous verrons qu'une extension du langage par l'ajout de conditions dans les règles enrichit son pouvoir d'expression. Section 3.2.2, le mécanisme d'évaluation sera étendu, par l'ajout de stratégies sur l'application des règles de réécriture, ceci dans le but de déterminiser le processus d'application des règles et, par là même d'augmenter l'efficacité des réductions. Enfin, section 3.3, nous présenterons rapidement le langage de programmation ELAN [BKK+96] basé sur la réécriture conditionnelle sous stratégie.

Enfin, après cette présentation de la programmation en logique de réécriture, nous présenterons brièvement section 3.4 les principales propriétés des systèmes de réécriture – terminaison, confluence, complétude suffisante, propriétés inductives – et les outils de vérification associés.

3.1 Programmer en réécriture

Dans cette section, nous décrivons la réécriture du point de vue de l'informaticien, en associant à chaque notion un concept du domaine de la programmation, des programmes et de leur exécution. Nous espérons que cette tentative, qui semblera naïve aux "réécrivains" rompus aux mécanismes de la réécriture, aidera les autres à assimiler le formalisme de la section précédente, en donnant une intuition plus "informatique" des notions préliminaires. D'autre part, nous souhaitons montrer que la logique de réécriture offre un cadre de programmation qui a des caractéristiques particulièrement intéressantes, tant au niveau de la lisibilité des programmes, que de la finesse de l'exécution grâce à la description de stratégies d'application des règles de réécriture.

3.1.1 Termes et systèmes de réécriture

La syntaxe d'un programme est exprimé dans le formalisme de la réécriture par un l'alphabet $\mathcal{F}$, i.e. un ensemble de symboles et leur nombre d'arguments. On rappelle que $\mathcal{F}$ se subdivise en constructeurs $\mathcal{C}$ et symboles définis $\mathcal{D}$. Les termes *constructeurs*, appartenant à l'ensemble $\mathcal{T}(\mathcal{C})$, sont construits uniquement avec des symboles de $\mathcal{C}$ et représentent les valeurs et objets manipulés par un programme: entiers, listes, etc. Un programme est un système de réécriture. Une requête est un terme à évaluer contenant au moins un symbole défini. Les variables figurant dans le membre gauche d'une règle sont des paramètres formels au même titre que les paramètres formels d'une fonction en C, en Lisp ou en Caml. Ces paramètres formels sont *instanciés* au moment de l'application d'une règle de réécriture. En appliquant une règle de réécriture sur une requête, on obtient un nouveau terme qui est soit un résultat partiel, s'il peut être réécrit, soit un résultat final, s'il est irréductible.

Exemple 3.1 *Soit* $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0, b : 0\}$ *et* $\mathcal{R}$ *le système de réécriture suivant définissant la fonction app de concaténation de deux listes:*

$$app(nil, x) \to x$$
$$app(cons(x, y), z) \to cons(x, app(y, z))$$

On a $\mathcal{D} = \{app\}$ *et* $\mathcal{C} = \{cons, a, b, nil\}$*. Le terme* $cons(a, cons(b, nil))$ *est un terme constructeur et il représente la liste* $[a, b]$*. Le terme* $app(nil, cons(a, app(nil, nil)))$ *représente l'état d'un calcul en cours: il contient encore des symboles définis; ce n'est qu'un résultat partiel car certains sous-termes sont encore évaluables.*

3.1.2 Exécution par réécriture

L'*évaluation* d'une requête est ici effectuée par la réécriture qui transforme un terme s (un état du calcul) en un autre terme t, i.e. $s \to_{\mathcal{R}} t$. Notons que la réécriture est une opération totalement non déterministe, à cause des choix possibles pour la position de réécriture et la règle utilisée. Nous verrons dans la section 3.2.2 comment selon le cas, éviter ou tirer parti de ce non-déterminisme. Deux étapes importantes de la réécriture sont la recherche de la substitution σ ou *filtrage*, puis *l'instanciation* des variables du membre droit de la règle par σ, représentant l'instanciation des paramètres formels de la règle par les paramètres effectifs contenus dans σ. Si la relation $\to_{\mathcal{R}}$ correspond à un pas d'exécution d'un programme, la relation $\to_{\mathcal{R}}{}^*$ représente un nombre quelconque de pas d'exécution et $\xrightarrow{!}_{\mathcal{R}}$ une exécution jusqu'à l'obtention d'un résultat final (nommé forme normale dans le contexte de la réécriture). Ainsi $s \xrightarrow{!}_{\mathcal{R}} t$ signifie que t est le résultat final de l'exécution du programme $\mathcal{R}$ sur la requête s. Pour deux requêtes s, t, $s \downarrow_{\mathcal{R}} t$ signifie qu'il existe un terme u tel que $s \to_{\mathcal{R}}{}^* u$ et $t \to_{\mathcal{R}}{}^* u$, i.e. que ces deux requêtes ont un résultat (partiel ou final) identique.

Exemple 3.2 *Soit* $\mathcal{F} = \{rev : 1, app : 2, cons : 2, nil : 0, a : 0, b : 0\}$ *et* $\mathcal{R}$ *le système de réécriture suivant définissant la fonction rev d'inversion d'une liste:*

$$\begin{array}{ll} (1) & app(nil, x) \to x \\ (2) & app(cons(x, y), z) \to cons(x, app(y, z)) \\ (3) & rev(nil) \to nil \\ (4) & rev(cons(x, y)) \to app(rev(y), cons(x, nil)) \end{array}$$

On a $\mathcal{D} = \{app, rev\}$ et $\mathcal{C} = \{cons, a, b, nil\}$. Soit le terme initial $rev(cons(a, cons(b, nil)))$ sur lequel nous appliquons $\mathcal{R}$. Nous utilisons les mêmes conventions que dans l'exemple 2.4 pour mettre en évidence le redex et le numéro de règle concernés par chaque réécriture.

$\underline{rev(cons(a, cons(b, nil)))} \rightarrow_{\mathcal{R}}{}^{(4)}$

$app(\underline{rev(cons(b, nil)), cons(a, nil)}) \rightarrow_{\mathcal{R}}{}^{(4)}$

$app(app(\underline{rev(nil)}, cons(b, nil)), cons(a, nil)) \rightarrow_{\mathcal{R}}{}^{(3)}$

$app(\underline{app(nil, cons(b, nil))}, cons(a, nil)) \rightarrow_{\mathcal{R}}{}^{(1)}$

$\underline{app(cons(b, nil), cons(a, nil))} \rightarrow_{\mathcal{R}}{}^{(2)}$

$cons(b, \underline{app(nil, cons(a, nil))}) \rightarrow_{\mathcal{R}}{}^{(1)}$

$cons(b, cons(a, nil))$

3.1.3 Ensembles de résultats partiels et finaux

Pour un système de réécriture $\mathcal{R}$ donné et un ensemble de termes initiaux E, l'ensemble des $\mathcal{R}$-descendants $\mathcal{R}^*(E)$ est l'ensemble de tous les termes accessibles à partir des termes de E par réécriture par $\mathcal{R}$. Si l'on voit $\mathcal{R}$ comme un programme, et E comme un ensemble de requêtes, $\mathcal{R}^*(E)$ représente tous les résultats partiels atteignables (ou états du calcul) à partir des requêtes de E par le programme $\mathcal{R}$. L'ensemble des $\mathcal{R}$-formes normales $\mathcal{R}^!(E)$, lui, est plus restreint puisqu'il ne contient que les résultats finaux atteignables. Nous en donnons ici un exemple volontairement très simple.

Exemple 3.3 *Nous reprenons le système de réécriture de l'exemple 3.1, définissant la fonction app de concaténation de deux listes. Soit $\mathcal{F} = \{app : 2, cons : 2, nil : 0, a : 0, b : 0\}$ et $\mathcal{R}$:*

$app(nil, x) \rightarrow x$
$app(cons(x, y), z) \rightarrow cons(x, app(y, z))$

Soit E l'ensemble des termes de la forme $app(s, t)$ où s est n'importe quelle liste plate constituée uniquement de "a" et t est n'importe quelle liste plate constituée uniquement de "b". On a donc

$E = \{app(nil, nil),\ app(cons(a, nil), nil),\ \ldots,\ app(nil, cons(b, nil)),\ \ldots,\ app(cons(a, nil), cons(b, nil)),\ app(cons(a, cons(a, nil)), cons(b, nil)),\ \ldots\}$.

L'ensemble $\mathcal{R}^*(E)$ est l'ensemble contenant E et tous les termes résultant de la réécriture des termes de E en un nombre quelconque d'étapes, entre autres les termes

$\{nil,\ cons(a, app(nil, cons(b, nil))),\ \ldots\}$,

mais également tous les termes en forme normale, i.e. $\mathcal{R}^!(E)$. L'ensemble $\mathcal{R}^!(E)$ est l'ensemble des listes plates constituées de "a" et de "b" telles que tous les "a" sont nécessairement placés avant les "b".

On peut noter que dans l'exemple précédent les ensembles E et $\mathcal{R}^!(E)$ peuvent être aisément représentés par les automates suivants: $A_1 = \langle \mathcal{F}, \mathcal{Q}_1, \mathcal{Q}_{f1}, \Delta_1 \rangle$ et $A_2 = \langle \mathcal{F}, \mathcal{Q}_2, \mathcal{Q}_{f2}, \Delta_2 \rangle$, respectivement, où $\mathcal{Q}_1 = \{q_0, q_1, q_2, q_3, q_4\}$, $\mathcal{Q}_{f1} = \{q_0\}$, $\Delta_1 = \{$

$$\begin{aligned} app(q_1, q_2) &\rightarrow q_0 \\ nil &\rightarrow q_1 \\ cons(q_3, q_1) &\rightarrow q_1 \\ nil &\rightarrow q_2 \\ cons(q_4, q_2) &\rightarrow q_2 \\ a &\rightarrow q_3 \\ b &\rightarrow q_4\ \ \} \end{aligned}$$

et $\mathcal{Q}_2 = \{q_0, q_1, q_2, q_3\}$, $\mathcal{Q}_{f2} = \{q_0\}$, et $\Delta_2 = \{$

$$\begin{aligned}
nil &\to q_0 \\
cons(q_2, q_0) &\to q_0 \\
cons(q_2, q_1) &\to q_0 \\
nil &\to q_1 \\
cons(q_3, q_1) &\to q_1 \\
a &\to q_2 \\
b &\to q_3 \;\}
\end{aligned}$$

Dans la section 6.4, nous verrons comment construire automatiquement des automates reconnaissant des sur-ensembles des $\mathcal{R}$-descendants et des $\mathcal{R}$-formes normales d'un ensemble E décrit lui aussi par un automate.

3.2 Extension des systèmes de réécriture

3.2.1 Systèmes de réécriture conditionnels

Dans un système de réécriture conditionnel, l'application de chaque règle est soumise à une condition qui est une conjonction d'équations de la forme $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$, également notées $s_1 = t_1, \ldots, s_n = t_n$, où $s_1, t_1, \ldots, s_n, t_n$ sont des termes. Grâce à ces conditions, le pouvoir d'expression de la réécriture est enrichi par rapport à la réécriture standard. Parmi les trois grands types de systèmes conditionnels, ne différant que par la méthode d'évaluation de la condition, nous nous intéressons ici aux systèmes conditionnels dits "à condition joignables", dans lesquels les conditions égalitaires $s_1 = t_1, \ldots, s_n = t_n$ sont satisfaites si $s_1 \downarrow_{\mathcal{R}} t_1, \ldots, s_n \downarrow_{\mathcal{R}} t_n$. Afin d'éviter toute confusion avec d'autres types de systèmes conditionnels, nous noterons les conditions en utilisant le symbole de joignabilité "$\downarrow$" et non pas avec le symbole d'égalité "$=$".

Un *système de réécriture conditionnel* est un couple $(\mathcal{F}, \mathcal{R})$, où $\mathcal{F}$ est un alphabet et $\mathcal{R}$ est un ensemble de *règles de réécriture conditionnelles*. Chacune de ces règles est de la forme $l \to r$ if $s_1 \downarrow t_1, \ldots, s_n \downarrow t_n$ où $l, r, s_1, t_1, \ldots, s_n, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, et $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. Les expressions $s_1 \downarrow t_1, \ldots, s_n \downarrow t_n$ sont les *conditions* de la règle. Comme dans le cas des systèmes de réécriture standard, $(\mathcal{F}, \mathcal{R})$ sera noté $\mathcal{R}$ s'il ne peut y avoir de confusion sur $\mathcal{F}$. La *relation de réécriture conditionnelle* $\to_{\mathcal{R}}$ induite par $\mathcal{R}$ est définie comme suit.

Définition 3.1 *(Relation de réécriture conditionnelle) Soit $\mathcal{R}$ un système de réécriture conditionnel. Pour tous termes $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $u \to_{\mathcal{R}} v$ si il existe une règle de réécriture $l \to r$ if $s_1 \downarrow t_1, \ldots, s_n \downarrow t_n \in \mathcal{R}$, une position $p \in \mathcal{P}os(u)$ et une substitution σ tels que $l\sigma = u|_p$, $v = u[r\sigma]_p$, et $s_i\sigma \downarrow_{\mathcal{R}} t_i\sigma$ pour tout $i \in \{1, \ldots n\}$.*

Exemple 3.4 *[Bou94] Voici un exemple de système de réécriture conditionnel: une spécification du tri par insertion, où $\mathcal{F} = \{true : 0, false : 0, 0 : 0, s : 1, \leq : 2, \neq : 2, insert : 2, isort : 2\}$ et $x, y, z, l \in \mathcal{X}$. Le symbole isort définit le tri par insertion, le symbole insert définit l'insertion* *d"un élément dans une liste triée.*

$$\begin{array}{l}
0 \leq x \to true \\
s(x) \leq 0 \to false \\
s(x) \leq s(y) \to x \leq y \\
insert(x, nil) \to cons(x, nil) \\
insert(x, cons(y, z)) \to cons(x, cons(y, z)) \text{ if } x \leq y \downarrow true \\
insert(x, cons(y, z)) \to cons(y, insert(x, z)) \text{ if } x \leq y \downarrow false \\
isort(nil) \to nil \\
isort(cons(x, l)) \to insert(x, isort(l))
\end{array}$$

A supposer que l'on souhaite réduire un terme de la forme $insert(s(0), cons(0, nil))$ à l'aide de la dernière règle définissant *insert*, nous détaillons l'application de cette règle. Tout d'abord, le filtrage donne une substitution $\sigma = \{x \mapsto s(0), y \mapsto 0, z \mapsto nil\}$ solution. Ensuite la condition est évaluée: on a $x\sigma \leq y\sigma = s(0) \leq 0 \to_{\mathcal{R}}^* false$, d'où on a bien $x\sigma \leq y\sigma \downarrow false$. Ensuite, la règle est appliquée comme dans le cas non-conditionnel.

3.2.2 Réécriture avec stratégie

Tout calcul basé sur la relation de réécriture (conditionnelle ou non) $\to_{\mathcal{R}}$, telle qu'elle a été définie, est indéterministe à deux niveaux. En effet, plusieurs règles peuvent s'appliquer sur un même terme et une même règle peut être appliquée à plusieurs positions dans un terme. Or, il est parfois nécessaire de limiter l'indéterminisme de la réécriture, notamment en vue de son implantation. L'application des règles de réécriture est rendue déterministe à l'aide de *stratégies* guidant le choix des règles à appliquer ou le choix des positions à réécrire.

Dans beaucoup d'implantations de la réécriture, la seule stratégie implantée est une stratégie réduisant prioritairement les redex dont les positions sont les plus internes dans le terme (stratégie innermost). Si deux redex existent à une même profondeur, la stratégie réduit prioritairement soit le redex de gauche (stratégie leftmost innermost), soit le redex de droite (stratégie rightmost innermost). Le choix des règles, quant à lui, est arbitraire. Dans le cas particulier de la stratégie innermost, il est possible de définir une *relation de réécriture innermost*, qui donnera lieu à des preuves spécifiques dans le cas de la preuve de terminaison.

Définition 3.2 *(Relation de réécriture innermost) Soit $(\mathcal{F}, \mathcal{R})$ un système de réécriture. Pour tous termes $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s \underset{in}{\longrightarrow}_{\mathcal{R}} t$ si il existe une règle de réécriture $l \to r \in \mathcal{R}$, une position $p \in \mathcal{P}os(s)$, une substitution σ tels que $l\sigma = s|_p$, $t = s[r\sigma]_p$, et tout sous-terme strict de $s|_p$ est irréductible.*

Si un système de réécriture termine, alors l'application de ses règles sous n'importe quelle stratégie termine également. Cependant, parfois, le système de réécriture ne termine que pour des stratégies particulières. Plus fréquemment encore, nous trouvons des situations où la terminaison du système de réécriture est difficile à établir, alors que la terminaison sous stratégie l'est moins. C'est ce que nous verrons avec deux stratégies particulières auxquelles nous nous intéressons dans cette thèse: la *stratégie de réduction modulaire* (SRM) et la *stratégie de réduction séquentielle* (SRS). Dans ces deux stratégies, le système de réécriture $\mathcal{R}$ est séparé en un nombre fini de sous-systèmes, ou modules, $\mathcal{R}_1, \ldots, \mathcal{R}_n$ et la réécriture d'un terme consiste en sa normalisation par chacun des modules successivement et séparément. Dans le cas de la stratégie de réduction modulaire, l'ordre d'application des modules est quelconque, alors que dans le cas de la stratégie de réduction séquentielle les modules $\mathcal{R}_1, \ldots, \mathcal{R}_n$ sont appliqués par ordre croissant de leur indice. Ce type de stratégie est notamment utilisé pour l'exécution de spécifications algébriques modulaires [Roc97]. A ces stratégies correspondent des relations de réécriture particulières. La relation de réduction modulaire est définie de la façon suivante.

Définition 3.3 *(Kurihara & Kaji [KK90]) Soit* $(\mathcal{F}, \mathcal{R})$ *un système de réécriture et* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. *On notera* $s \Rrightarrow_{\mathcal{R}} t$ *si* $s \rightarrow^{+}_{\mathcal{R}} t$ *et* $t \in IRR(\mathcal{R})$.

Définition 3.4 *Soient* $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ *des systèmes de réécriture. Soit* $\mathcal{F} = \bigcup_{j=1}^{n} \mathcal{F}_j$. *Pour tous termes* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, *on notera* $s \Rrightarrow t$ *s'il existe* $j \in \{1, \ldots, n\}$ *tel que* $s \Rrightarrow_{\mathcal{R}_j} t$.

La relation de réduction séquentielle qui est un cas particulier de la relation de réduction modulaire est définie de la façon suivante.

Définition 3.5 *Soient* $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ *des systèmes de réécriture. Soit* $\mathcal{F} = \bigcup_{j=1}^{n} \mathcal{F}_j$. *Pour tous termes* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,

$$s \rightarrow_{\mathcal{R}_1; \ldots; \mathcal{R}_n} t$$

si s est réductible par le système $(\mathcal{F}, \mathcal{R}_1 \cup \ldots \cup \mathcal{R}_n)$ *et* $\exists s_1, \ldots, s_{n-1} \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *tels que* $s \underset{R_1}{\xrightarrow{!}} s_1$ *et* $s_1 \underset{R_2}{\xrightarrow{!}} s_2$ *et ... et* $s_{n-1} \underset{R_n}{\xrightarrow{!}} t$.

3.3 Le langage ELAN

Le langage de programmation ELAN [BKK+96] est basé sur la réécriture avec stratégies. En ELAN, un programme est défini par un ensemble de règles de réécriture et une stratégie contrôlant l'application de ces règles. Lorsque aucune stratégie n'est donnée, comme dans le cas du programme de l'exemple suivant, la stratégie par défaut est la stratégie innermost. Cette stratégie correspond à la stratégie d'appel par valeur utilisée dans les langages fonctionnels tels Lisp ou Caml. D'ailleurs, une fois compilée la réécriture innermost donne des résultats comparables en terme d'efficacité avec les langages fonctionnels, par exemple avec Caml.

Exemple 3.5 *Voici l'exemple du tri par insertion codé en* ELAN. *Par rapport à l'exemple 3.4, la seule différence est que nous utilisons ici les booléens du langage ainsi que les entiers builtin d'* ELAN *au lieu des entiers construits sur* 0 *et successeur.*

```
module tri
import bool builtinInt;
end

sort nat listnat;
end

operators
  global
        nil             : listnat;
        cons(@, @)      : (builtinInt listnat) listnat;
        insert(@, @)    : (builtinInt listnat) listnat;
        isort(@)        : (listnat) listnat;
        start(@)        : (builtinInt) listnat;
end
```

```
rules for listnat
        x, y: builtinInt;
        z: listnat;
global
[] insert(x, nil)        => cons(x, nil)                          end
[] insert(x, cons(y, z)) => cons(x, cons(y, z)) if x <= y         end
[] insert(x, cons(y, z)) => cons(y, insert(x, z)) if not(x <= y) end
[] isort(nil)            => nil                                   end
[] isort(cons(x, z))     => insert(x, isort(z))                   end
end
```

L'interprète ELAN *permet, en outre, d'interpréter ce programme et d'évaluer des requêtes comme nous le voyons maintenant sur un exemple simple:*

```
enter command finished by ';':
run isort(cons(4, cons(3, cons(5, cons(1, nil))))) ;

 [] result term:
    cons(1,cons(3,cons(4,cons(5,nil))))
```

A titre de comparaison, nous donnons maintenant le même programme écrit en Caml:

```
type l = nil
       | cons of int*l;;

let rec insert = function x -> function z ->
        match z with
          nil -> cons(x,nil)
        | cons(u,v) -> if x<= u then
                         cons(x, cons(u, v))
                       else
                         cons(u, (insert x v))
        ;;

let rec isort = function
          nil -> nil
        | cons(x,y) -> (insert x (isort y));;
```

L'exécution de ces deux programmes, compilés, sur des listes non-triées de 5000 entiers donne des temps de calcul[1] *d'environ une minute. Par exemple, dans le pire cas la liste de 5000 entiers est triée dans l'ordre décroissant et les temps sont les suivants: 59.98 secondes pour Caml et 68.22 secondes pour* ELAN.

Il faut remarquer que dans les langages fonctionnels, comme Caml, la stratégie d'évaluation – l'appel par valeur – est figée, et qu'il en est de même pour la stratégie de résolution des buts en Prolog – la SLD-résolution. En ELAN, la stratégie d'application des règles n'est pas figée mais est un des éléments du langage lui même. Il est ainsi possible pour l'utilisateur de donner sa propre stratégie d'application des règles si la simple stratégie innermost ne convient pas, ou si elle n'est pas la plus efficace. Par exemple si l'on souhaite réaliser une fonction donnant toutes les permutations d'une liste d'entiers donnée, ceci peut être codé en ELAN de la façon suivante:

1. Ces temps comparatifs, réalisés sur SUN UltraSparc 1, n'ont pas la prétention d'être des benchmarks, nous souhaitons juste montrer que la vitesse de calcul de ces deux langages est de même ordre dans le cas où la stratégie d'évaluation est l'appel par valeur.

```
rules for list[int]
        i: int;
        l1, l2, l3: list[int];
global
[r1]    permut(nil, nil)    => nil                        end

[r2]    permut(i.nil, l1)   => i.l2
        where l2:=(p_strat) permut(l1,nil)                end

[r3]    permut(i.l1, l2)  => i.l3
        where l3:=(p_strat) permut(append(l2, l1),nil)    end

[r4]    permut(i.l1, l2)  => permut(l1, append(l2, i.nil))  end
end

strategies for list[int]
implicit
  [.] p_strat =>
        repeat*(first(r1,r2, dk(r3,r4)))
  end
end
```

où pour obtenir toutes les permutations d'une liste d'entier `l` le terme à évaluer est `perm(l, nil)`. L'idée de la méthode utilisée ici est codée par les règles `[r3]` et `[r4]`: pour toute liste dont le premier élément est `i` et dont le reste est `l1`, soit on place `i` en tête de la liste permutée et on applique récursivement la permutation à `l` (avec `[r3]`), soit on recherche le premier élément de la liste permutée dans `l` (avec `[r4]`). En appliquant ces deux règles de façon non-déterministe, on obtient toutes les permutations possibles de la liste initiale. En ELAN l'application non-déterministe est codée par l'opérateur de stratégie `dk` ("don't know"). Les règles `[r1]` et `[r2]` traitent respectivement le cas où les deux listes sont vides et le cas où la première liste ne contient qu'un seul élément. Dans la stratégie utilisée ici, ces deux règles sont utilisées en priorité (opérateur `first`) par rapport aux règles `[r3]` et `[r4]`. Par conséquent, les cas limites pris en compte par `[r1]` et `[r2]` n'ont pas besoin d'être pris en compte par `[r3]` et `[r4]`. Par exemple, on sait en appliquant la règle [r3] que l'on a nécessairement `l1`$\neq$`nil`. En terme d'efficacité, la compilation des stratégies non-déterministes permet à ELAN de concurrencer les meilleurs Prologs dans la résolution efficace de problèmes nécessitant une gestion optimale du retour-arrière [Mor98].

Une autre caractéristique primordiale du langage ELAN est d'intégrer de nombreuses facilités pour la programmation d'algorithmes décrits par des règles de déduction. Outre le mécanisme de réécriture qui code l'application d'une règle de déduction, et le langage de description de stratégies, on dispose en ELAN d'un préprocesseur qui facilite la programmation de règles de déduction complexes. A titre d'exemple, dans la section 6.2.3, nous décrirons un algorithme de filtrage particulier dont une des règle est:

Decompose $$\frac{f(s_1,\ldots,s_n) \trianglelefteq f(q_1,\ldots,q_n)}{s_1 \trianglelefteq q_1 \wedge \ldots \wedge s_n \trianglelefteq q_n}$$

où f est n'importe quel symbole de la signature $\mathcal{F}$. Nous avons programmé cet algorithme de filtrage en ELAN. Le code correspondant à la règle de déduction ci-dessus est le suivant:

```
FOR EACH SS:pair[identifier,int]; F:identifier; N:int SUCH THAT
```

```
SS:=(listExtract) elem(Fss) AND F :=()first(SS) AND N:=()second(SS):{

rules for list[conj_match]
          s_1, ..., s_N: Cterm;
          q_1, ..., q_N : Cterm;
global
  [decompose]

((F(s_1, ..., s_N) ?= F(q_1, ..., q_N) ^ phi1) v phi2) with(R) matches(phi3, l)

=>

({s_I ?= q_I ^ }_I=1...N phi1 v phi2) with(R) matches(phi3, l)

end
}
```

où `Fss` représente la signature, `SS`=(`F`, `N`) un couple quelconque (symbole, arité) extrait de cette signature, et `phi1`, `phi2` le contexte d'application de cette règle. Les constructions du type `s_1, ..., s_N` ou `{s_I ?= q_I ^ }_I=1...N` rendues possibles par le préprocesseur ELAN permettent de calquer exactement le code ELAN sur l'expression formelle de ces règles et donc de prototyper de façon rapide et sûre des algorithmes à base de règles.

3.4 Preuves de propriétés

Nous avons vu que la réécriture est un mécanisme d'évaluation souple et efficace. Nous allons maintenant voir que c'est également un contexte dans lequel il est possible de démontrer automatiquement des propriétés assurant la fiabilité d'un programme. Parmi les propriétés des systèmes de réécriture, nous distinguerons les *propriétés fondamentales* et les *théorèmes inductifs*. Les trois propriétés fondamentales d'un système de réécriture sont la *terminaison* (ou décroissance dans le cas d'un système conditionnel), la *confluence* et la *complétude suffisante*. La terminaison de la réécriture et les outils de preuve associés seront vus plus en détails dans le chapitre suivant. Nous présentons brièvement les autres propriétés, les outils de preuve associés ainsi que leurs besoins en ordre et en preuve de terminaison.

3.4.1 Terminaison

Pour montrer qu'un système de réécriture termine, nous avons vu qu'il est suffisant de prouver l'existence d'un ordre de réduction orientant les règles du système. Dans le chapitre 4, nous verrons qu'il existe deux grandes familles de méthodes de preuve de terminaison. Certaines méthodes prouvent uniquement qu'il existe bien un tel ordre sans le construire, alors que d'autres construisent un ordre de réduction $>$ *explicite* et *calculable*, i.e. étant donnés deux termes s et t, il est possible de décider si $s > t$.

3.4.2 Confluence

Nous définissons maintenant la confluence. D'un point de vue opérationnel, la confluence d'un système de réécriture garantit l'unicité du résultat: si une requête s donne lieu à deux exécutions différentes $s \rightarrow_{\mathcal{R}}^* t$ et $s \rightarrow_{\mathcal{R}}^* u$ alors, ces deux exécutions doivent mener nécessairement à un même résultat v.

Définition 3.6 *(Confluence) Un système de réécriture $\mathcal{R}$ est confluent si pour tous termes $s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ tels que $s \to_{\mathcal{R}}{}^* t$ et $s \to_{\mathcal{R}}{}^* u$ alors il existe un terme $v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ tel que $t \to_{\mathcal{R}}{}^* v$ et $u \to_{\mathcal{R}}{}^* v$.*

La confluence est une propriété indécidable en général. En revanche, lorsque le système de réécriture termine, celle-ci devient décidable grâce au calcul des *paires critiques* et au lemme de Newmann.

Définition 3.7 *(Paire critique) Soit $\mathcal{R}$ un système de réécriture et $l \to r$, $g \to d$ deux règles de $\mathcal{R}$ ayant des ensembles de variables disjoints. S'il existe une position $p \in \mathcal{P}os_{\mathcal{F}}(l)$ et une substitution σ telle que $l|_p\sigma = g\sigma$ alors, $(l[d]_p\sigma, r\sigma)$ est une paire critique de $\mathcal{R}$.*

Notons que pour tout système de réécriture $\mathcal{R}$ fini, si l'on prend la précaution de ne considérer que les unificateurs σ les plus généraux pour $l|_p$ et g, le nombre de paires critiques est fini. Ainsi, grâce au lemme suivant, si le système $\mathcal{R}$ est fini et terminant, la confluence d'un système $\mathcal{R}$ devient décidable.

Lemme 3.1 *(Lemme de Newmann[New42]) Soit $\mathcal{R}$ un système de réécriture terminant. Le système $\mathcal{R}$ est confluent si et seulement si toute paire critique (p, q) de $\mathcal{R}$ est telle qu'il existe $w \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ tel que $p \to_{\mathcal{R}}{}^* w$ et $q \to_{\mathcal{R}}{}^* w$.*

Dans le cas où le système n'est pas confluent, il est possible de le *compléter* afin d'en assurer la confluence, à l'aide de la procédure de complétion de D. Knuth et P. B. Bendix [KB70]. Initialement, une procédure de complétion permet, à partir d'un ensemble d'équations E, de calculer un système de réécriture $\mathcal{R}$ tel que pour tous termes s, t, si $s =_E t$ alors $s \downarrow_R t$. Mais, si on initialise cette procédure sur un système de réécriture $\mathcal{R}$ non confluent, celle-ci construit un système $\mathcal{R}'$ confluent tel que $\to_R \subseteq \to_{R'}$. En d'autres termes, tout calcul fait par $\mathcal{R}$ peut être effectué par $\mathcal{R}'$ avec, pour $\mathcal{R}'$, la garantie supplémentaire de l'unicité du résultat.

Le principe général de la procédure de complétion d'un système $\mathcal{R}$ est le suivant. Initialement, on recherche un ordre de réduction $>$ prouvant la terminaison de $\mathcal{R}$. Ensuite, pour chaque paire critique (p, q) rencontrée, la procédure de complétion de D. Knuth et P. B. Bendix [KB70], force la confluence du système en ajoutant une règle $p \to q$ si $p > q$, et $q \to p$ si $q > p$.

A notre connaissance, les implantations utilisent en général un ordre $>$, calculable, et fixé au début de la procédure de complétion. En réalité, il existe une autre possibilité qui permet de s'abstraire d'un ordre calculable. Pour chaque paire critique (p, q) rencontrée, il suffit de tenter deux preuves de terminaison, une pour le système $\mathcal{R} \cup \{p \to q\}$ et une pour le système $\mathcal{R} \cup \{q \to p\}$, et d'ajouter la règle assurant la terminaison du système étendu. Notons cependant que la modification de l'ordre au cours de la procédure de complétion peut poser des problèmes [] (Sattler-Klein "about changing the ordering during KB completion, lncs 775, stacs94).

Notons également qu'une nouvelle approche des preuves par cohérence a été récemment présentée par H. Comon [Com] et que celle-ci ne nécessite aucune hypothèse de terminaison, et donc aucun ordre.

3.4.3 Complétude suffisante

La propriété de complétude suffisante d'un système de réécriture sur un ensemble de termes E, assure que tout terme de E peut être réécrit en un terme constructeur, i.e. en un résultat. Cette propriété a déjà beaucoup été étudiée notamment par [Com86, Kou85, NW83, KNZ87], dans le contexte des spécifications algébriques. Nous donnons, ici, une définition de la complétude suffisante d'un système de réécriture sur un ensemble de termes clos $E \subseteq \mathcal{T}(\mathcal{F})$.

Définition 3.8 *Un système de réécriture $\mathcal{R}$ est suffisamment complet sur $E \subseteq \mathcal{T}(\mathcal{F})$ si $\forall s \in E$, $\exists t \in \mathcal{T}(\mathcal{C})$ tel que $s \rightarrow^*_{\mathcal{R}} t$, où $\mathcal{C}$ est l'ensemble des symboles constructeurs de $\mathcal{F}$.*

Habituellement, les méthodes utilisées pour la vérification de cette propriété sur les spécifications algébriques sont basées sur l'énumération et le test [Kou85, NW83, KNZ87] ou encore sur la disunification [Com86]. Nous proposerons dans la section 6.6, une méthode basée sur le calcul de l'ensemble des $\mathcal{R}$-formes normales de E.

3.4.4 Théorèmes inductifs

Lorsque les propriétés à prouver peuvent être exprimées sous la forme de clauses égalitaire de la forme $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$, nommées *conjectures*, il est possible d'utiliser le mécanisme de preuve par induction. Une conjecture est un *théorème inductif* d'un système ou d'une spécification $\mathcal{R}$ si l'on a $s_1\sigma \xleftrightarrow[\mathcal{R}]{*} t_1\sigma \wedge \ldots \wedge s_n\sigma \xleftrightarrow[\mathcal{R}]{*} t_n\sigma$, pour toute substitution close σ.

Les méthodes de preuve par induction se divisent en deux grandes familles: l'induction explicite et l'induction implicite (ou "induction sans induction"). Parmi les méthodes d'induction implicite, on distingue encore deux approches principales que sont les preuves par cohérence et les preuves par réécriture. Pour chacune de ces approches, il existe des démonstrateurs particuliers tels NQTHM, INKA, PVS pour l'induction explicite, RRL, pour les preuves par cohérence et SPIKE [BKR92, Bou94] pour les preuves par réécriture.

Les preuves par cohérence sont basées sur une procédure de complétion des systèmes de réécriture semblable à celle utilisée pour assurer la confluence d'un système de réécriture (voir sections 3.4.2). Dans une preuve par cohérence, pour prouver qu'un système $\mathcal{R}$, terminant et confluent, vérifie une propriété $s = t$, il suffit de tenter une complétion du système $\mathcal{R} \cup \{s = t\}$. L'équation $s = t$ est un théorème inductif de $\mathcal{R}$ si et seulement si la complétion réussit. A partir de ce mécanisme de base, de nombreuses améliorations ont été développées, dont un survol peut être trouvé dans la thèse de A. Bouhoula [Bou94].

Dans les preuves inductives par réécriture, l'induction est plus apparente que dans le cas des preuves par cohérence, et elle s'effectue sur la relation de réécriture elle-même. Avant d'en préciser le principe, rappelons d'abord le principe général d'induction.

Soit E un ensemble, $>$ un ordre bien fondé sur E et P une propriété. Afin de montrer que pour tout élément $t \in E$ on a $P(t)$, dans une preuve par induction, en général, on cherchera à montrer que

- tous les éléments minimaux de E (vis-à-vis de l'ordre $>$) ont la propriété P,
- pour tous les éléments $t, t' \in E$, tels que $t > t'$, on a $P(t') \implies P(t)$.

Or, étant donné un ensemble E quelconque il est parfois difficile de concevoir un ordre bien fondé sur E. Ainsi, par commodité, dans une preuve réalisée à la main, l'induction est transposée sur l'ensemble $\mathbb{N}$ muni de $>_{\mathbb{N}}$, l'ordre habituel sur les entiers naturels: chaque élément de E est interprété par un entier. Lorsque l'ensemble E est un ensemble de termes, muni d'un système de réécriture terminant $\mathcal{R}$, alors $\rightarrow_{\mathcal{R}}$ est un ordre bien fondé pour E. Il est donc possible de baser l'induction directement sur $\rightarrow_{\mathcal{R}}$. Nous donnons maintenant le principe général de la procédure de preuve par réécriture telle qu'elle apparaît dans le démonstrateur SPIKE [BKR92, Bou94]. La procédure est définie par des règles de déduction s'appliquant sur un couple C, H où, C est l'ensemble des conjectures à établir ou à réfuter et H est un ensemble d'hypothèses d'induction. Le démonstrateur SPIKE peut également démontrer des conjectures conditionnelles. Cependant, afin de simplifier la présentation, nous ne donnons ici que les règles utilisées dans le cas de conjectures non-conditionnelles.

Expand	$\dfrac{C \cup \{p = q\}, H}{C \cup C', H \cup \{p = q\}}$	si $p \not< q$, et $C' = Exp(p = q)$
Delete	$\dfrac{C \cup \{p = p\}, H}{C, H}$	
Simplify	$\dfrac{C \cup \{p = q\}, H}{C \cup \{p' = q\}, H}$	si $p \rightarrow_{\mathcal{R} \cup H \cup C \setminus \{p=q\}} p'$.

$Exp(p = q)$ est l'ensemble des équations résultant de l'expansion de $p = q$ par un schéma d'induction. Le couple initial sur lequel sont appliquées les règles est de la forme $(C, \emptyset)$. Si l'application des règles s'achève sur un couple de la forme $(\emptyset, H)$ alors toutes les conjectures de C sont des théorèmes inductifs de $\mathcal{R}$.

Au regard de la propriété de terminaison, la règle qui nous intéresse ici principalement est la règle **Simplify** dans laquelle p est simplifié en p', i.e. $p \rightarrow_{\mathcal{R} \cup H \cup C \setminus \{p=q\}} p'$. Il s'agit d'une simplification, car en supposant qu'il existe un ordre de réduction $>$ prouvant la terminaison de $\mathcal{R}$ et orientant les équations de H et de $C \setminus \{p = q\}$ alors, si $p \rightarrow_{\mathcal{R} \cup H \cup C \setminus \{p=q\}} p'$, on a nécessairement $p > p'$ et en étendant cet ordre aux équations puis aux ensembles d'équations, on obtient: $C \cup \{p = q\} > C \cup \{p' = q\}$.

En pratique, pour réaliser une preuve inductive par réécriture, la première étape consiste à trouver un ordre de réduction prouvant la terminaison de $\mathcal{R}$. Ensuite, au cours de la preuve, des équations sont ajoutées aux ensembles H et C. Afin d'orienter ces équations pour les utiliser dans la phase de simplification, on a deux possibilités:

- on dispose d'un ordre explicite et calculable $>$ qui permet d'orienter les équations de H et C. C'est le cas le plus simple. Éventuellement, l'ordre $>$ peut être étendu s'il est incrémental,
- à chaque nouvelle utilisation de la règle **simplify**, une nouvelle preuve de terminaison est effectuée pour $\mathcal{R} \cup H \cup C \setminus \{p = q\}$ (et on conserve l'orientation des équations déjà orientées auparavant).

Dans l'implantation actuelle de SPIKE, c'est la première solution qui a été choisie. L'estimation du nombre des appels aux différentes procédures du code de SPIKE, permet de montrer que la procédure d'orientation est la procédure la plus souvent appelée lors d'une preuve. En conséquence, la deuxième solution n'est pas réaliste: pour cette application, un ordre explicite et calculable semble indispensable.

Chapitre 4

Méthodes de preuve de terminaison

Il est impossible de parler de preuve de terminaison des systèmes de réécriture sans parler d'ordre. Depuis les travaux initiaux de Z. Manna et S. Ness [MN70], ordres et preuves de terminaison de systèmes de réécriture sont étroitement liés. Il existe une autre méthode de preuve de terminaison des systèmes de réécriture qui ne sera pas traitée dans cette thèse: les *calculs de clôture* (forward closure) [LM78, Geu89, DH95].

La quantité d'ordres de réduction est telle qu'il est impossible de les présenter tous ici: il a fallu faire un choix. Ce choix est celui qui a guidé cette thèse, c'est le choix de l'automatisation. S'il existe de nombreux ordres, peu sont réellement utilisables dans un démonstrateur automatique car beaucoup d'entre eux sont des ordres *sémantiques*, basés sur des interprétations laissées au libre choix de l'utilisateur. Les ordres sémantiques sont parmi les ordres les plus puissants, de par la grande liberté que laissent les interprétations: il est possible de coder les "arguments de terminaison" – les raisons qui font qu'un système termine – dans une interprétation. Or, ces ordres, même s'ils viennent à bout de problèmes difficiles de façon élégante et concise[2], ne sont réservés qu'à des utilisateurs chevronnés capable de déduire les "arguments de terminaison" d'un système et de les exprimer sous la forme d'une interprétation. En outre, lorsqu'il s'agit de gros systèmes de réécriture, intervenant par exemple dans certaines spécifications algébriques, une preuve totalement manuelle avec de tels ordres n'est pas envisageable parce que trop longue et trop complexe à mettre en place.

Dans ce chapitre, nous définissons les ordres *syntaxiques lpo*, *rpo*, et *rpos*. Nous montrons l'évolution chronologique de la définition de ces ordres. Nous définissons également les *interprétations polynomiales* qui, bien qu'étant des ordres sémantiques, sont une méthode de preuve de terminaison automatisable [Ste94, Gie95a]. Tous ces ordres sont des ordres de simplification. Toute preuve de terminaison réalisée grâce à ces ordres repose sur la Proposition 2.4.

Les systèmes de réécriture dont la terminaison est montrée par un ordre de simplification sont aussi nommés *systèmes simplifiants*. Or, les ordres de simplification, même s'ils permettent de venir à bout de la majeure partie des preuves de terminaison, ne sont pas toujours suffisants. Par opposition, les systèmes de réécriture dont la terminaison ne peut être montrée par un ordre de simplification sont appelés *systèmes non-simplifiants*. Afin de prouver la terminaison des systèmes non-simplifiants, il est possible d'étendre l'applicabilité des ordres de simplification en affaiblissant les propriétés à prouver, par exemple avec la méthode des *paires de dépendance* présentée en section 4.3. Une autre voie consiste à utiliser l'ordre *gpo*, un ordre hybride mélangeant des carac-

2. voir par exemple la preuve de terminaison du système de substitution explicite σ_0 de T. Hardin et A. Laville [HL86] dont la terminaison a été prouvée grâce à l'étiquetage sémantique (semantic labelling) par H. Zantema [Zan95].

téristiques propres aux ordres syntaxiques et aux ordres sémantiques. Celui-ci généralise, en fait, un grand nombre d'ordres connus aussi bien syntaxiques que sémantiques; il est présenté dans la section 4.4. Dans le chapitre 5, nous proposerons une méthode originale automatisant l'utilisation de cet ordre par résolution de contraintes et permettant de guider l'utilisateur dans le cas de preuves incrémentales semi-automatiques. Le lecteur pourra consulter [Der87, Ste95, Fer95], pour un survol détaillé des ordres et [Art97] pour la méthode des paires de dépendance.

Dans la section 4.5, nous verrons plus en détail la notion d'ordre calculable. Nous définirons également la notion de décroissance d'un système conditionnel et nous montrerons que cette propriété peut être montrée à l'aide des paires de dépendance et de l'ordre *gpo*. Le problème lié à la modularité des preuves de terminaison sera évoqué dans la section 4.6 et celui de la prise en compte des stratégies d'application des règles dans la section 4.7. Enfin, nous donnerons un bref aperçu des implantations en rapport avec les méthodes de preuve vues dans ce chapitre, dans la section 4.8

4.1 Ordres syntaxiques

Avec un ordre syntaxique, les termes sont comparés de façon structurelle et grâce à un ordre sur les symboles fonctionnels nommé *précédence*. A chaque précédence correspond une *instance* différente de l'ordre. Dans cette thèse, les principaux ordres syntaxiques utilisés sont *rpo*, *lpo*, et *rpos*. Par *rpo* nous désignons l'ordre récursif sur les chemins (recursive path ordering) tel qu'il a été défini initialement par N. Dershowitz [Der82]. Par *lpo* nous désignons l'ordre lexicographique sur les chemins (lexicographic path ordering) tel qu'il a été défini par S. Kamin et J.-J. Lévy [KL82]. Par *rpos* nous désignons l'ordre récursif sur les chemins avec statut (recursive path ordering with statut) tel qu'il a été défini par P. Lescanne [Les84]. Cet ordre généralise le *rpo* et le *lpo*. Assez souvent, la dénomination *rpo* est utilisée pour désigner le *rpos*. Par souci de clarté, dans cette thèse, nous distinguerons *rpo*, *lpo*, et *rpos*.

4.1.1 Définitions préliminaires

Les définitions usuelles des ordres syntaxiques sont basées sur une *précédence*.

Définition 4.1 *(précédence) Soit $\mathcal{F}$ une signature, une précédence est un ordre (partiel) $>_{\mathcal{F}}$ sur $\mathcal{F}$.*

Avec les ordres syntaxiques, nous serons amenés à comparer non pas deux objets mais deux *multi-ensembles*, ou deux *suites* d'objets. Soit E un ensemble. Un multi-ensemble sur E est un ensemble avec occurrences multiples de ses éléments. Un multi-ensemble sur E est défini par une application $M : E \mapsto \mathbb{N}$, où $M(x)$ est le nombre d'occurrences de x dans M, pour $x \in E$. Un multi-ensemble $M(a) = 3$, $M(c) = 1$, et $M(x) = 0$ pour $x \neq a, c$, sera également noté $\{\!\{a, a, a, c\}\!\}$. L'appartenance à un multi-ensemble M sera notée $\in$ et définie par $x \in M$ si $M(x) \neq 0$. On notera $\emptyset$ le multi-ensemble vide, et $M(E)$ l'ensemble des multi-ensembles d'éléments de E. L'union de deux multi-ensembles $M : E \mapsto \mathbb{N}$ et $M' : E \mapsto \mathbb{N}$ est le multi-ensemble $M \cup M'$ tel que pour tout $x \in E$: $(M \cup M')(x) = M(x) + M'(x)$. De la même façon, la différence de M et M', notée $M \setminus M'$ est définie par $(M \setminus M')(x) = M(x) - M'(x)$ (si $M(x) < M'(x)$, alors $(M \setminus M')(x) = 0$). Initialement, la définition de l'extension multi-ensemble que nous utilisons est due à N. Dershowitz et Z. Manna [DM79], pour les ordres stricts. Cette définition a, depuis, été étendue au cas des relations binaires quelconques. La définition que nous donnons ici est extraite de [Mid90, Fer95].

Définition 4.2 *(Extension multi-ensemble [DM79, Mid90, Fer95]) Soit R une relation binaire sur un ensemble E et $X, Y \in M(E)$. L'*extension multi-ensemble *de R sur $M(E)$, est la relation binaire, notée R^{mul}, définie par $X\ R^{mul}\ Y$ si $\exists X_0, Y_0 \in M(E)$ tels que:*

- $X_0 \neq \emptyset$ *et* $X_0 \subseteq X$,
- $Y = (X \setminus X_0) \cup Y_0$,
- $\forall y \in Y_0$, $\exists x \in X_0$ *tel que* $x\ R\ y$.

Dans la section 5.6, nous verrons une définition plus opérationnelle de cette extension dans le cas où elle porte sur un préordre.

Proposition 4.1 *[DM79] Si $>$ est un ordre bien fondé sur E, alors l'extension multi-ensemble de $>$ sur $M(E)$ est un ordre bien fondé, noté $>^{mul}$.*

Si E est un ensemble, notons E^* l'ensemble des suites finies d'éléments de E. L'extension lexicographique d'un ordre sur E aux éléments de E^* est définie de la façon suivante.

Définition 4.3 *(Extension lexicographique) Soit R une relation binaire sur un ensemble E, $S, S' \in E^*$ et $s_0, \ldots, s_m, t_0, \ldots, t_n \in E$, avec $m, n \in \mathbb{N}$. L'*extension lexicographique *de R sur E^*, est la relation binaire, notée R^{lex}, définie par $S\ R^{lex}\ S'$, si:*

- $S \neq \lambda$ *et* $S' = \lambda$, *où λ représente le mot vide, ou*
- $S = s_0 \ldots s_m$, $S' = t_0 \ldots t_n$ *et* $s_0\ R\ t_0$, *ou*
- $S = s_0 \ldots s_m$, $S' = t_0 \ldots t_n$, $s_0 = t_0$, *et* $s_1 \ldots s_m\ R^{lex}\ t_1 \ldots t_n$.

Proposition 4.2 *Si $>$ est un ordre bien fondé sur E, alors l'extension lexicographique de $>$ sur E^* est un ordre bien fondé, noté $>^{lex}$.*

4.1.2 L'ordre *rpo*

Dans les définitions du *rpo*, du *lpo* et du *rpos* que nous allons voir, figurent toujours les trois mêmes cas: (1) le cas où le symbole de tête f du membre gauche est supérieur au symbole de tête g du membre droit, (2) le cas où ils sont identiques, et (3) le cas où un sous-terme du membre gauche est supérieur ou équivalent à tout le membre droit. Il est intéressant de noter que, pour les principaux ordres syntaxiques (dont *rpo*, *lpo* et *rpos*), le cas donnant à l'ordre toute sa spécificité est en fait le deuxième, dans lequel les symboles sont identiques. Ce cas est particulièrement important, puisqu'il entre en jeu dès que l'on considère une règle récursive: le membre gauche est un terme avec f au sommet et le membre droit contient un sous-terme avec le même symbole f au sommet. Dans le cas particulier du *rpo*, une congruence de permutation $\simeq_{permut}$ intervient également. Elle est définie de la façon suivante.

Définition 4.4 *(Congruence de permutation) Soit $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. On a $s \simeq_{permut} t$ si $s = t \in \mathcal{X}$, ou $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_n)$, $f, g \in \mathcal{F}^n$ et*

- $f = g$, *et*
- *il existe une permutation π de $\{1, \ldots, n\}$ telle que pour tout $1 \leq i \leq n$: $s_i \simeq_{permut} t_{\pi(i)}$.*

Définition 4.5 *(Recursive path ordering: rpo [Der82]) Soit* $>_{\mathcal{F}}$ *une précédence,* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s = f(s_1, \ldots, s_n)$, *et* $t = g(t_1, \ldots, t_m)$. *On a* $s >_{rpo} t$ *si*

1. $f >_{\mathcal{F}} g$ *et* $s >_{rpo} t_i$ *pour tout* $1 \leq i \leq m$, *ou*
2. $f = g$ *et* $\{\!\{s_1, \ldots, s_n\}\!\} >_{rpo}^{mul} \{\!\{t_1, \ldots, t_m\}\!\}$, *ou*
3. $\exists i,\ 1 \leq i \leq n$ *tel que* $s_i >_{rpo} t$ *ou* $s_i \simeq_{permut} t$.

Dans le cas où $f = g$, l'ordre *rpo* compare les sous-termes de s et t en les regroupant d'abord en un multi-ensemble. Voici un exemple de l'utilisation de cet ordre sur un système simple.

Exemple 4.1 *Soit* $\mathcal{F} = \{app : 2, cons : 2, nil : 0\}$, *et* $\mathcal{R}$ *le système décrivant la fonction app de concaténation de deux listes:*

$$\begin{array}{l} app(nil, x) \rightarrow x \\ app(cons(x, y), z) \rightarrow cons(x, app(y, z)) \end{array}$$

Pour prouver la terminaison de ce système avec rpo, il suffit de trouver une précédence adéquate de façon que $app(nil, x) >_{rpo} x$ *et* $app(cons(x, y), z) >_{rpo} cons(x, app(y, z))$. *Dans notre cas, toute précédence dans laquelle* $app >_{\mathcal{F}} cons$ *fera l'affaire. En effet, pour la première règle, on a bien*

$$app(nil, x) >_{rpo} x$$

par le cas 3. de la définition du rpo, puisque $x \simeq_{permut} x$. *Pour la deuxième règle, on souhaite montrer que*

$$app(cons(x, y), z) >_{rpo} cons(x, app(y, z))$$

Or la précédence est telle que l'on a $app >_{\mathcal{F}} cons$, *d'où par le cas 1. de la définition du rpo, pour montrer l'inégalité précédente, il est suffisant de montrer que*

$$app(cons(x, y), z) >_{rpo} x \text{ et } app(cons(x, y), z) >_{rpo} app(y, z)$$

La première inégalité est trivialement obtenue par le cas 3. de la définition. Pour la deuxième, puisque les symboles de tête des deux membres de l'inégalité sont égaux, on applique le cas 2. du rpo ainsi il reste à montrer

$$\{\!\{cons(x, y), z\}\!\} >_{rpo}^{mul} \{\!\{y, z\}\!\}$$

Enfin, cette dernière inégalité est trivialement vérifiée puisque l'on a

$$cons(x, y) >_{rpo} y$$

Cet ordre puissant s'impose encore comme ordre de référence parmi tous les ordres de terminaison. Cependant il a quelques faiblesses notamment lorsque la terminaison est due à la décroissance du premier argument d'un opérateur alors que globalement les arguments ne diminuent pas. En voici un exemple.

Exemple 4.2 *Soit la fonction rev effectuant une inversion de liste. Le système qui la définit a la particularité d'effectuer l'inversion de liste sans avoir recours à un opérateur de concaténation. Soit* $\mathcal{F} = \{rev : 1, rev2 : 2, cons : 2, nil : 0\}$ *et soit* $\mathcal{R}$ *le système:*

$$\begin{array}{l} rev(x) \to rev2(x, nil) \\ rev2(nil, x) \to x \\ rev2(cons(x,y), z) \to rev2(y, cons(x,z)) \end{array}$$

La dernière règle ne peut pas être orientée par le rpo standard. En effet, aucun cas de la définition du rpo ne peut s'appliquer. En particulier, pour le cas 2., il est impossible de montrer que

$$\{\!\{cons(x,y), z\}\!\} >^{mul}_{rpo} \{\!\{y, cons(x,z)\}\!\}$$

puisque $cons(x,z) >_{rpo} z$ *et* $cons(x,z)$ *est incomparable avec* $cons(x,y)$.

4.1.3 L'ordre *lpo*

Le type de décroissance évoqué dans l'exemple précédent est parfaitement capturé par l'ordre *lpo* puisque celui-ci compare les arguments, non pas regroupés en multi-ensembles, mais en séquence de gauche à droite. L'ordre *lpo* est défini de la façon suivante.

Définition 4.6 *(Lexicographic path ordering: lpo [KL82]) Soit* $>_{\mathcal{F}}$ *une précédence,* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s = f(s_1, \ldots, s_n)$, *et* $t = g(t_1, \ldots, t_m)$. *On a* $s >_{lpo} t$ *si*

1. $f >_{\mathcal{F}} g$ *et* $s >_{lpo} t_i$ *pour tout* $1 \leq i \leq m$, *ou*
2. $f = g$ *et* $s >_{lpo} t_i$ *pour tout* $1 \leq i \leq m$, *et* $s_1 \ldots s_n >^{lex}_{lpo} t_1 \ldots t_m$, *ou*
3. $\exists i$, $1 \leq i \leq n$ *tel que* $s_i >_{lpo} t$ *ou* $s_i = t$.

Les définitions de *lpo* et *rpo* sont très proches, mais il est utile de remarquer qu'outre le remplacement de l'extension lexicographique par l'extension multi-ensemble du cas 2., la congruence de permutation dans le cas 3. est remplacée par l'égalité stricte dans la définition du *lpo*.

Exemple 4.3 *Voici sur l'exemple 4.2 le résultat de l'application du lpo. Pour toute précédence dans laquelle on a* $rev >_{\mathcal{F}} nil$, $rev >_{\mathcal{F}} rev2$ *et* $rev2 >_{\mathcal{F}} cons$, *on doit prouver pour la première règle que*

$$rev(x) >_{lpo} rev2(x, nil).$$

Or on a $rev >_{\mathcal{F}} rev2$. *On applique donc le cas 1. du lpo. Il reste à montrer que*

$$rev(x) >_{lpo} x \text{ et } rev(x) >_{lpo} nil.$$

La première inégalité est montrée par le cas 3. et la deuxième par le cas 1. étant donné que $rev >_{\mathcal{F}} nil$. *La deuxième règle est orientée sans difficulté par le cas 3. du lpo. Enfin, pour la dernière règle, on souhaite montrer*

$$rev2(cons(x,y), z) >_{lpo} rev2(y, cons(x,z)).$$

Les deux symboles de tête sont identiques. On applique donc le cas 2. Il reste à montrer que

$$rev2(cons(x,y), z) >_{lpo} y,\ rev2(cons(x,y), z) >_{lpo} cons(x,z)$$

et

$$cons(x,y), z >_{lpo}^{lex} y, cons(x,z).$$

La première inégalité est triviale. Pour la troisième, il est suffisant d'appliquer la définition de l'extension lexicographique du lpo et le fait que $cons(x,y) >_{lpo} y$. *Pour la deuxième inégalité, en revanche, il est nécessaire d'appliquer à nouveau le cas 1. du lpo et le fait que* $rev2 >_{\mathcal{F}} cons$ *pour conclure.*

4.1.4 L'ordre *rpos*

Dans l'exemple précédent, la décroissance est due au premier argument de l'opérateur $rev2$, mais il est clair qu'elle dépend totalement du système à traiter. La décroissance peut être due au premier argument, au dernier, comme elle peut être due au i-ème. Il est évident que cela dépend des opérateurs: pour un opérateur f la décroissance peut provenir de son i-ème argument, alors que pour un opérateur g elle peut provenir du rang $j \neq i$. D'autre part, pour certains opérateurs, la décroissance n'est pas localisée sur un ou plusieurs arguments, elle est globale. Ceci constitue l'idée de base des *statuts* qui associent aux opérateurs un ordre de comparaison de leurs arguments, ou un multi-ensemble dans le cas d'une décroissance globale. Voici une définition des statuts empruntée à [Ste89].

Définition 4.7 *(statut) A une signature* $\mathcal{F}$ *on associe une fonction de statut* τ *qui pour tout symbole* $f \in \mathcal{F}^n$ *peut prendre deux valeurs:*

- *soit* $\tau(f) = mul$, *qui signifie que pour toute comparaison, si* f *est le symbole de tête d'un terme* $f(t_1, \ldots, t_n)$, *les arguments de* f *seront considérés en multi-ensemble:* $\{\!\{t_1, \ldots, t_n\}\!\}$,
- *soit* $\tau(f) = lex_\pi$, *où* π *est une permutation de l'ensemble* $\{1, \ldots, n\}$; *ce qui signifie que pour toute comparaison, si* f *est le symbole de tête d'un terme* $f(t_1, \ldots, t_n)$, *les arguments de* f *seront considérés en une suite lexicographique dont l'ordre est donné par* π: $t_{\pi(1)} \ldots t_{\pi(n)}$. *Deux permutations* π *classiques sont les permutations* $left = 1, \ldots, n$ *et* $right = n, \ldots, 1$.

Il faut remarquer que cette dernière définition généralise la première définition de l'extension lexicographique (définition 4.3 page 33) qui devient un cas particulier de cette dernière définition: le statut $left$. A partir de cette définition des statuts, la définition du $rpos$, où selon la valeur de $\tau(f)$, $>_{rpos}^{\tau(f)}$ est l'extension multi-ensemble ou l'extension lexicographique de $>_{rpos}$, est la suivante:

Définition 4.8 *(recursive path ordering avec statut: rpos (version 1)) Soit* $>_{\mathcal{F}}$ *une précédence,* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s = f(s_1, \ldots, s_n)$, *et* $t = g(t_1, \ldots, t_m)$. *On a* $s >_{rpos} t$ *si*

1. $f >_{\mathcal{F}} g$ *et* $s >_{rpos} t_i$ *pour tout* $1 \leq i \leq m$, *ou*
2. $f = g$, $s >_{rpos} t_i$ *pour tout* $1 \leq i \leq m$ *et* $(s_1, \ldots, s_n) >_{rpos}^{\tau(f)} (t_1, \ldots, t_m)$, *ou*
3. $\exists 1 \leq i \leq n$ *tel que* $s_i >_{rpos} t$ *ou* $s_i = t$.

Voici un exemple faisant intervenir deux statuts différents pour deux symboles d'un même système de réécriture. Cet exemple simple ne fait intervenir que deux statuts lexicographiques, mais parfois des statut multi-ensembles sont également nécessaires.

Exemple 4.4 *Voici l'exemple de la définition des opérateurs* $+$ *et* $-$. *Soit* $\mathcal{F} = \{+ : 2, - : 2, s :$

$1, 0 : 0\}$ *et soit* $\mathcal{R}$ *le système:*

$$\begin{array}{l} 0 + x \to x \\ s(x) + y \to s(x + y) \\ x + (y + z) \to (x + y) + z \\ 0 - x \to 0 \\ x - 0 \to x \\ s(x) - s(y) \to x - y \\ (x - y) - z \to x - (y + z) \end{array}$$

Dans cet exemple, les règles ne posent pas de problème d'orientation, excepté les règles d'associativité $x + (y + z) \to (x + y) + z$ *et de factorisation* $(x - y) - z \to x - (y + z)$ *qui nécessitent un choix particulier pour le statut des opérateurs + et −. En effet, pour la première, par la propriété de sous-terme, on a trivialement* $y + z >_{rpos} z$ *et en choisissant* $\tau(+) = right$*, on obtient* $x, y + z >_{rpos}^{right} x + y, z$ *et finalement* $x + (y + z) >_{rpos} (x + y) + z$*, par définition de rpos. A l'inverse, il faut choisir le statut* $\tau(-) = left$ *et une précédence telle que* $- >_{\mathcal{F}} +$*, pour orienter correctement la règle* $(x - y) - z \to x - (y + z)$ *avec rpos.*

4.1.5 Le préordre *rpos*

Le *rpos* étend à la fois le *rpo* et le *lpo*. Cependant, cet ordre souffre d'une faiblesse typique des premiers ordres de terminaison: seules les comparaisons strictes sont définies, tant au niveau de la précédence que du *rpos* lui-même, et l'égalité ne peut être que syntaxique. Dans un contexte où l'on recherche l'inégalité stricte, l'égalité peut effectivement sembler superflue. Or, en étendant les définitions de précédence et du *rpos* (ou de tout autre ordre syntaxique) avec un cas d'égalité, on augmente sensiblement sa puissance. Historiquement, dans les premières définitions des ordres syntaxiques, seul le cas d'égalité syntaxique de deux symboles est pris en compte. Dans [KL82], l'utilisation d'un préordre sur les symboles, au lieu d'un ordre, est évoquée mais son utilisation semble encore imprécise. Citons S. Kamin et J.-J. Lévy [KL82]: "bullshit, but maybe interesting in practice". Leur intuition se révèle être juste: ceci est effectivement utile en pratique et cette extension permet d'orienter de nombreux systèmes intéressants dont la terminaison n'aurait pas pu être prouvée avec l'ordre strict, soit directement (voir section 7.1.6) soit en les utilisant conjointement à la méthode des paires de dépendances (voir section 4.3). Nous définissons donc maintenant la précédence et le *rpos* sous la forme de préordres, suivant les définitions de [Der87, Ste89, Fer95].

Définition 4.9 *(précédence) Soit* $\mathcal{F}$ *une signature. Une précédence est un préordre partiel* $\geq_{\mathcal{F}}$ *sur* $\mathcal{F}$*.*

Nous donnons maintenant une nouvelle définition de *rpos* en définissant, non pas l'ordre strict, mais son préordre associé $\geq_{rpos}$. Dans cette définition, afin que le préordre $\geq_{rpos}$ soit réflexif sur les variables (que $x \geq_{rpos} x$) il est nécessaire d'étendre la précédence $\geq_{\mathcal{F}}$ à $\mathcal{F} \cup \mathcal{X}$ en supposant que pour toute variable $x \in \mathcal{X}$ on a $x \simeq_{\mathcal{F}} x$ et que x reste incomparable avec les éléments de $\mathcal{F}$ et toute autre variable $y \in \mathcal{X}$ telle que $x \neq y$. De plus il est nécessaire d'imposer une compatibilité de $\simeq_{\mathcal{F}}$ avec les statuts, i.e. si $f \simeq_{\mathcal{F}} g$ et $\tau(f) = mul$ alors $\tau(g) = mul$. En revanche, il est important de noter que deux symboles f et g tels que $f \simeq_{\mathcal{F}} g$ peuvent avoir des statuts lexicographiques différents.

Définition 4.10 *(recursive path ordering avec statut: rpos (version 2)) Soit* $\geq_{\mathcal{F}}$ *une précédence,*

$s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s = f(s_1, \ldots, s_n)$ avec $f \in \mathcal{F} \cup \mathcal{X}$, et $t = g(t_1, \ldots, t_m)$ avec $g \in \mathcal{F} \cup \mathcal{X}$ et $m \geq 0$. On a $s \geq_{rpos} t$ si

1. *$f >_{\mathcal{F}} g$ et $s >_{rpos} t_i$ pour tout $1 \leq i \leq m$, ou*
2. *$f \simeq_{\mathcal{F}} g$, $s >_{rpos} t_i$ pour tout $1 \leq i \leq m$ et $(s_1, \ldots, s_n) \geq_{rpos}^{\tau} (t_1, \ldots, t_m)$, ou*
3. *$\exists 1 \leq i \leq n$ tel que $s_i \geq_{rpos} t$.*

L'ordre strict $>_{rpos}$ n'est autre que `ord(` $\geq_{rpos}$ `)`. Si l'on considère une précédence ne contenant que des inégalités strictes, cet ordre est équivalent au *rpos* initial. Cependant, nous voyons dans l'exemple suivant que l'utilisation d'une précédence avec égalités permet d'orienter des règles qui ne pouvaient l'être avec le *rpos* initial.

Exemple 4.5 *Voici un exemple de système de réécriture ne pouvant être orienté que par un préordre rpos. Ce système est basé sur la signature $\mathcal{F} = \{* : 2, / : 2, inv : 1, 1 : 0\}$, et est le suivant:*

$$\begin{array}{l} x/x \rightarrow 1 \\ x/1 \rightarrow x \\ inv(x/y) \rightarrow y/x \\ inv(x) * x \rightarrow 1 \\ x/inv(y) \rightarrow x * y \\ 1 * y \rightarrow y \\ x * (y/z) \rightarrow (x * y)/z. \end{array}$$

En choisissant la précédence $inv >_{\mathcal{F}} / >_{\mathcal{F}} 1$, on oriente correctement la majeure partie des règles. Seules deux opposent quelque résistance: il s'agit de $x/inv(y) \rightarrow x * y$ et $x * (y/z) \rightarrow (x * y)/z$. La première suggère que $/ >_{\mathcal{F}} *$ et la deuxième $* >_{\mathcal{F}} /$, ce qui est contradictoire avec le fait que $>_{\mathcal{F}}$ doit être une relation irréflexive. Or, en choisissant ici $/ \simeq_{\mathcal{F}} *$, avec $\tau(/) = \tau(*) = right$, il est possible de conclure. En effet, pour la première règle, on a $inv(y) >_{rpos} y$ par la propriété de sous-terme (cas 3. de la définition du rpos), et donc $x, inv(y) >_{rpos}^{right} x, y$. Enfin par le cas 2. de la définition du rpos, puisque $/ \simeq_{\mathcal{F}} *$, on a bien $x/inv(y) >_{rpos} x * y$. Pour la deuxième règle on procède de la même façon. On a $y/z >_{rpos} z$, d'où $x, y/z >_{rpos}^{right} x * y, z$, et par le cas 2. de la définition de rpos on obtient $x * (y/z) >_{rpos} (x * y)/z$

4.2 Ordres sémantiques

Dans le cas général des ordres sémantiques, les termes sont interprétés sur un domaine A muni d'un ordre $\succ$ et leurs interprétations sont comparées avec l'ordre $\succ$. Les seules contraintes étant que A soit non-vide et que l'ordre $\succ$ soit bien fondé et monotone, il existe une infinité de domaines d'interprétation et sur chaque domaine une infinité d'ordres possibles. Une définition générale des ordres sémantiques peut être trouvée dans [Zan94, Fer95]. Chronologiquement, l'interprétation polynomiale a été une des premières méthodes de preuve de terminaison, d'abord proposée par D. S. Lankford [Lan75, Lan79], puis affinée par A. Ben Cherifa et P. Lescanne [BCL86], en vue de son implantation, elle reste la méthode sémantique la plus utilisée. Dans le cas de l'interprétation polynomiale, le domaine A et l'ordre $\succ$ sont fixés: A est l'ensemble $\mathbb{N}$ des entiers naturels, $\succ$ est l'ordre usuel $>$ sur les entiers, et chaque opérateur est interprété par un polynôme.

Définition 4.11 *(interprétation polynomiale) Une interprétation polynomiale pour la signature $\mathcal{F}$ associe à chaque symbole $f \in \mathcal{F}^n$ un polynôme, noté $f_{\mathbb{N}}$, tel que $f_{\mathbb{N}}(a_1, \ldots, a_n) \in \mathbb{N}$ pour tout $a_1, \ldots, a_n \in \mathbb{N}$.*

Pour prouver la terminaison d'un système de réécriture avec une interprétation polynomiale, il est nécessaire de trouver un polynôme $f_{\mathbb{N}}$ pour tout symbole $f \in \mathcal{F}$ tel que si $a > b$ alors $f_{\mathbb{N}}(\dots a \dots) > f_{\mathbb{N}}(\dots b \dots)$, pour tout $a, b \in \mathbb{N}$. Ensuite, il suffit de montrer que pour toute règle $l \to r \in \mathcal{R}$, l'interprétation polynomiale de l est strictement supérieure à celle de r vis-à-vis de l'ordre habituel sur les entiers $>$.

Exemple 4.6 *[SK93] Nous montrons la terminaison du système $\mathcal{R}$ suivant, spécifiant l'implication, avec une interprétation polynomiale. Soit $\mathcal{F} = \{\wedge : 2, \subset : 2, \neg : 1, false : 0\}$, et $\mathcal{R}$ le système suivant:*

$$
\begin{array}{l}
x \wedge false \to false \\
x \wedge (\neg false) \to x \\
\neg(\neg x) \to x \\
false \subset y \to \neg false \\
x \subset false \to \neg x \\
(\neg x) \subset (\neg y) \to y \subset (x \wedge y)
\end{array}
$$

Voici une interprétation polynomiale de ce système, prouvant sa terminaison:

$$
\begin{array}{l}
\wedge_{\mathbb{N}}(X, Y) = X + Y \\
\subset_{\mathbb{N}} (X, Y) = 2X + 2Y \\
\neg_{\mathbb{N}}(X) = 2X \\
false_{\mathbb{N}} = 1
\end{array}
$$

Chaque interprétation est telle que si $a > b$ alors $f_{\mathbb{N}}(\dots a \dots) > f_{\mathbb{N}}(\dots b \dots)$, pour tout $a, b \in \mathbb{N}$. Par exemple, pour $\wedge_{\mathbb{N}}$ et pour tout $X, Y, Z \in \mathbb{N}$ tels que $X > Y$, on a $\wedge_{\mathbb{N}}(X, Z) = X + Z > Y + Z = \wedge_{\mathbb{N}}(Y, Z)$ et $\wedge_{\mathbb{N}}(Z, X) = Z + X > Z + Y = \wedge\mathbb{N}(Z, Y)$.

Ensuite, pour chaque règle, on montre que l'interprétation est strictement décroissante. Par exemple, pour la règle $(false \subset y \to \neg false)$, le polynôme interprétant le membre gauche est $2 * 1 + 2Y = 2Y + 2$ et à droite $2 * 1 = 2$. Or, il est possible de borner les valeurs que peut prendre Y. En effet, le terme ayant la plus petite valeur vis-à-vis de l'interprétation est $false$, dont l'interprétation est 1. En conséquence, la variable Y est nécessairement supérieure ou égale à 1. On a donc trivialement $2Y + 2 > 2$ pour tout $Y \geq 1$.

Pour la règle $(\neg x) \subset (\neg y) \to y \subset (x \wedge y)$, on procède de la même façon. Pour le membre gauche on obtient le polynôme: $2 * (2X) + 2 * (2Y) = 4X + 4Y$. Pour le membre droit, on obtient le polynôme: $2Y + 2 * (X + Y) = 2X + 4Y$. Comme dans le cas précédent, on sait que les valeurs de X et Y sont nécessairement supérieures ou égales à 1. On a donc $4X + 4Y > 2X + 4Y$ pour tout $X \geq 1$ et $Y \geq 1$. En procédant de la même façon avec les autres règles, on montre que le système termine.

Dans les deux sections suivantes, nous allons présenter les deux principales méthodes permettant de venir à bout de systèmes de réécriture non simplifiants tout en conservant de bonnes propriétés pour l'automatisation. La première de ces méthodes – le critère des paires de dépendance initialement conçu par T. Arts [Art96] puis développé par T. Arts et J. Giesl [AG97a, AG97b, AG98] – consiste en un affaiblissement des propriétés nécessaires à la terminaison d'un système. La deuxième méthode est un ordre: le *gpo* conçu par N. Dershowitz et C. Hoot [DH95]. Celui-ci généralise les principaux ordres connus, sémantiques et syntaxiques. Il permet de combiner leurs avantages, et de par sa structure, de réaliser des preuves de terminaison de façon incrémentale.

4.3 La méthode des paires de dépendance

Les paires de dépendance [Art96, Art97, AG97a, AG97b, AG98] sont l'un des plus puissants outils d'automatisation des preuves de terminaison des systèmes de réécriture. Cette méthode est basée sur une vue fonctionnelle des systèmes de réécriture. En programmation fonctionnelle, pour prouver la terminaison d'une fonction, il est suffisant de montrer que les appels récursifs à cette fonction sont strictement décroissants vis-à-vis d'un ordre bien-fondé. Le principe des paires de dépendance est d'extraire des paires d'appels récursifs d'une chaîne de réécriture, de montrer que les paires sont décroissantes, et donc que la chaîne de réécriture est finie. Cette méthode est séparée en trois principales étapes qui sont: la génération d'un ensemble d'inégalités, leur simplification, et enfin leur résolution par recherche d'un ordre de terminaison les satisfaisant. Hormis [Gie95a], les travaux de T. Arts et J. Giesl portent principalement sur les deux premières phases. Or, la résolution de ces inégalités peut être réalisée à l'aide du *gpo* et de l'algorithme de résolution de contrainte que nous verrons dans le chapitre 5. Maintenant, nous donnons les définitions et les théorèmes essentiels, empruntés à T. Arts [Art97], pour une utilisation pratique des paires de dépendance. Pour une étude plus détaillée et pour les preuves, nous conseillons au lecteur de se reporter à sa thèse [Art97].

4.3.1 Paires de dépendance

Afin de rester compatible avec les notations utilisées dans [Art97], les paires de dépendance seront exprimées sur une signature étendue par des symboles en majuscules: pour tout symbole défini $g \in \mathcal{D}$ on crée un nouveau symbole constructeur G. Les paires de dépendance sont définies de la façon suivante.

Définition 4.12 *(paire de dépendance) Soit $\mathcal{R}$ un système de réécriture sur $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$, et $C[\,]$ un contexte quelconque de $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Si $f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_m)]$ est une règle de $\mathcal{R}$ et $f, g \in \mathcal{D}$, alors $\langle F(s_1, \ldots, s_n), G(t_1, \ldots, t_m)\rangle$ est une paire de dépendance de $\mathcal{R}$.*

Si le système $\mathcal{R}$ est fini, le nombre des paires de dépendance est fini et leur recherche est facilement automatisable.

Exemple 4.7 *Voici un exemple emprunté à [Art97]. Soit $\mathcal{F} = \{minus : 2, quot : 2, s : 1, 0 : 0\}$ et $\mathcal{R}$ le système de réécriture suivant, définissant la fonction quot telle que $quot(x, y) = \lceil \frac{x}{y} \rceil$, pour tout $x, y \in \mathbb{N}$ et tels que $y > 0$, où $\lceil t \rceil$ est égal à t si $t \in \mathbb{N}$ et à la partie entière de t plus 1 si $t \notin \mathbb{N}$.*

$$
\begin{array}{l}
minus(x, 0) \to x \\
minus(s(x), s(y)) \to minus(x, y) \\
quot(0, s(y)) \to 0 \\
quot(s(x), s(y)) \to s(quot(minus(x, y), s(y)))
\end{array}
$$

Les deux symboles définis sont minus et quot, i.e. $\mathcal{D} = \{quot : 2, minus : 2\}$. L'ensemble des paires de dépendance de ce système est défini sur une signature étendue par les symboles constructeurs $MINUS$ et $QUOT$. L'ensemble des paires de dépendance est le suivant:

$$
\begin{array}{l}
\langle MINUS(s(x), s(y)), MINUS(x, y)\rangle \\
\langle QUOT(s(x), s(y)), QUOT(minus(x, y), s(y))\rangle \\
\langle QUOT(s(x), s(y)), MINUS(x, y)\rangle
\end{array}
$$

4.3.2 Le critère de terminaison

Le critère de terminaison est basé sur l'absence de chaîne infinie de paires de dépendance. Les chaînes (de paires) de dépendance sont définies de la façon suivante.

Définition 4.13 *(Chaîne de dépendance) Soit $\mathcal{R}$ un système de réécriture. Une chaîne de dépendance est une suite $\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \ldots$ de paires de dépendance telle qu'il existe une substitution σ qui vérifie $t_i\sigma \rightarrow^*_{\mathcal{R}} s_{i+1}\sigma$ pour tout $i \geq 1$.*

A partir des paires de dépendance, le critère de terminaison d'un système de réécriture est le suivant.

Théorème 4.1 *(T. Arts [Art97]) Un système de réécriture $\mathcal{R}$ termine si et seulement s'il n'a pas de chaîne de dépendance infinie.*

Pour un système $\mathcal{R}$ donné, toutes les paires de dépendance ne donnent pas lieu à des chaînes de dépendance infinies. En pratique, il est possible de restreindre cette étude à un sous-ensemble de paires *pouvant* donner lieu à des chaînes infinies. Les chaînes de dépendance étant difficiles à visualiser, il est commode de les décrire par un graphe, nommé *graphe de dépendance*, regroupant toutes les chaînes de dépendance possibles et permettant d'isoler les sous-chaînes critiques.

Définition 4.14 *(Graphe de dépendance) Soit $\mathcal{R}$ un système de réécriture et $\langle s, t\rangle$, $\langle u, v\rangle$ des paires de dépendance de $\mathcal{R}$. Le graphe de dépendance de $\mathcal{R}$ est un graphe dirigé dont les nœuds sont étiquetés par des paires de dépendance de $\mathcal{R}$ et dans lequel il existe un arc entre un nœud étiqueté par $\langle s, t\rangle$ et un nœud étiqueté par $\langle u, v\rangle$ si et seulement s'il existe une substitution σ telle que $t\sigma \rightarrow^*_{\mathcal{R}} u\sigma$.*

Dans ce graphe, tout cycle trahit la possibilité d'une chaîne de dépendance infinie, construite sur les paires de dépendance du cycle. Or, si la présence d'un cycle est une condition nécessaire pour l'existence d'une chaîne de dépendance infinie, elle n'est pas suffisante. La recherche des cycles dans le graphe de dépendance permet uniquement d'isoler les sous-chaînes critiques dont la finitude est à contrôler.

Exemple 4.8 *Si l'on reprend le système de l'exemple 4.7, et son ensemble de paires de dépendance, il est possible de donner son graphe de dépendance:*

$$\begin{tikzcd}
\langle MINUS(s(x), s(y)), MINUS(x, y)\rangle \arrow[loop above] \\
\langle QUOT(s(x), s(y)), MINUS(x, y)\rangle \arrow[u] \\
\langle QUOT(s(x), s(y)), QUOT(minus(x, y), s(y))\rangle \arrow[u] \arrow[loop below]
\end{tikzcd}$$

La boucle supérieure sur la paire de dépendance $\langle MINUS(s(x), s(y)), MINUS(x, y)\rangle$ est due au fait qu'il existe une substitution simple $\sigma = \{x \mapsto s(x'), y \mapsto s(y')\}$ telle que $MINUS(x, y)\sigma = MINUS(s(x'), s(y'))$.

A l'inverse, il n'y a pas d'arc de la paire de dépendance $MINUS(s(x), s(y)), MINUS(x, y)\rangle$ vers $\langle QUOT(s(x), s(y)), MINUS(x, y)\rangle$, car il n'existe aucune substitution σ telle que

$$MINUS(x, y)\sigma \rightarrow^* QUOT(s(x), s(y))$$

(Nous rappelons que dans le contexte des paires de dépendances, les symboles $MINUS$ et $QUOT$ sont des symboles constructeurs).

Dans cet exemple particulier, il n'existe que deux cycles possibles, donc deux chaînes de dépendance infinies possibles. L'une de ces chaînes est constituée uniquement de la paire de dépendance $\langle MINUS(s(x),s(y)), MINUS(x,y)\rangle$, et l'autre de la paire $\langle QUOT(s(x),s(y)), QUOT(minus(x,y),s(y))\rangle$. Ceci est un cas particulier intéressant dans lequel il est possible de prouver la terminaison directement à partir du graphe avec des arguments de terminaison simples. Par exemple, dans le cas de la première paire, le nombre de symboles s est strictement décroissant, donc tout cycle construit sur cette paire fait décroître le nombre de s, et est donc fini. Pour la deuxième paire, avec une interprétation naturelle de $s(x)$ par $x+1$ et de $minus(x,y)$ par $x-y$, il est clair que $s(x)$ est strictement supérieur à $minus(x,y)$, d'où le cycle est là encore décroissant. Ceci est suffisant pour prouver la terminaison de ce système.

Dans l'exemple précédent, la construction du graphe de dépendance est simple et il est suffisant pour mener à bien la preuve, cependant ce cas est idéal et en général d'autres outils sont nécessaires.

4.3.3 Les paires de dépendance en pratique

L'utilisation du critère précédent pose en pratique un double problème.

- l'existence de chaînes de dépendance infinies est indécidable. En effet, dans le cas contraire, par le théorème 4.1, la terminaison des systèmes de réécriture serait décidable.
- la construction du graphe de dépendance est indécidable. En effet, il n'est pas possible de décider s'il existe une substitution σ telle que $s\sigma \to^*_{\mathcal{R}} t\sigma$, pour deux termes s et t donnés.

A défaut de calculer le graphe exact, il est possible d'en calculer une approximation: un graphe dans lequel l'ensemble des arcs est un sur-ensemble de l'ensemble des arcs réels. Autrement dit, si un arc figure dans le graphe réel, alors il figure dans le graphe approximation. Il existe différentes approximations possibles. Nous donnons ici l'approximation du graphe de dépendance proposée dans [Art97]. Celle-ci est calculée grâce aux fonctions CAP et REN dont les définitions suivent. Intuitivement, la fonction CAP abstrait par une nouvelle variable tout terme dont la racine est un symbole défini, et la fonction REN effectue un renommage de toutes les variables d'un terme par de nouvelles variables.

Définition 4.15 *Soit $f(t_1,\dots,t_n) \in \mathcal{T}(\mathcal{F},\mathcal{X})$ et $\mathcal{X}'$ un ensemble de nouvelles variables tel que $\mathcal{X} \cap \mathcal{X}' = \emptyset$. Soient $x, x' \in \mathcal{X} \cup \mathcal{X}'$ et $y \in \mathcal{X}'$. Les fonctions CAP et REN sont définies par:*

$$\begin{array}{rcl} CAP(x) & = & x \\ CAP(f(t_1,\dots,t_n)) & = & \left\{ \begin{array}{ll} y \text{ où } y \text{ est une nouvelle variable} & \text{si } f \in D \\ f(CAP(t_1),\dots,CAP(t_n)) & \text{si } f \notin D \end{array} \right. \end{array}$$

$$\begin{array}{rcl} REN(x) & = & y \text{ où } y \text{ est une nouvelle variable} \\ REN(f(t_1,\dots,t_n)) & = & f(REN(t_1),\dots,REN(t_n)) \end{array}$$

où, si $x \neq x'$, alors $REN(x) \neq REN(x')$.

Ces deux fonctions permettent de décider de façon approximative si, pour deux termes donnés u et v, il existe une substitution σ telle que $u\sigma \to^*_{\mathcal{R}} v\sigma$. L'approximation est la suivante: on considérera que $u\sigma \rightarrow^*_{\mathcal{R}} v\sigma$ si REN(CAP(u)) et v sont unifiables [Art97]. L'intuition derrière la fonction CAP est simple: tout terme dont le symbole de tête est un symbole constructeur ne peut pas être réécrit à la racine. A l'inverse, tout terme dont le symbole de tête est un symbole défini peut être réécrit, il est donc abstrait par une nouvelle variable.

Définition 4.16 *(graphe de dépendance) Soit $\mathcal{R}$ un système de réécriture et $\langle s,t\rangle$, $\langle u,v\rangle$ des paires de dépendance de $\mathcal{R}$. L'approximation du graphe de dépendance de $\mathcal{R}$ est un graphe dirigé dont les nœuds sont étiquetés par des paires de dépendance de $\mathcal{R}$ et, dans lequel il existe un arc entre un nœud étiqueté par $\langle s,t\rangle$ et un nœud étiqueté par $\langle u,v\rangle$ si et seulement si $REN(CAP(t))$ et u sont unifiables.*

D'autres approximations plus fines sont possibles. Certaines évoquées par T. Arts et J. Giesl sont basées sur le narrowing [AG97c].

Exemple 4.9 *Voici un autre système spécifiant la division sans faire appel à la soustraction [Art97]. Soit $\mathcal{F} = \{quot : 3, s : 1, 0 : 0\}$ et $\mathcal{R}$ tel que $quot(x,y,y)$ calcule le résultat de la division euclidienne de x par y:*

$$\begin{array}{l} quot(0,s(y),s(z)) \rightarrow 0 \\ quot(s(x),s(y),z) \rightarrow quot(x,y,z) \\ quot(x,0,s(z)) \rightarrow s(quot(x,s(z),s(z))) \end{array}$$

L'ensemble des paires de dépendance de ce système est le suivant:

$$\begin{array}{l} \langle QUOT(s(x),s(y),z), QUOT(x,y,z)\rangle \\ \langle QUOT(x,0,s(z)), QUOT(x,s(z),s(z))\rangle. \end{array}$$

Ensuite, nous construisons le graphe de dépendance approché. Par exemple, pour savoir s'il peut y avoir un arc de la paire $\langle QUOT(x,0,s(z)), QUOT(x,s(z),s(z))\rangle$ vers la paire $\langle QUOT(s(x),s(y),z),$ $QUOT(x,y,z)\rangle$, il suffit de rechercher un unificateur entre le terme

$$REN(CAP(QUOT(x,s(z),s(z)))) = QUOT(x_1,s(y_1),s(z_1))$$

et le terme $QUOT(s(x),s(y),z)$. C'est ici le cas, puisque $\sigma = \{x_1 \mapsto s(x), y_1 \mapsto y, z \mapsto s(z_1)\}$ est un unificateur pour les deux termes précédents.

En revanche, il n'y a pas d'arc cyclique sur la paire de dépendance

$$\langle QUOT(x,0,s(z)), QUOT(x,s(z),s(z))\rangle$$

puisque $REN(CAP(QUOT(x,s(z),s(z)))) = QUOT(x_1,s(y_1),s(z_1))$ n'est pas unifiable avec $QUOT(x,$ $0,s(z))$. En procédant de la même façon avec toutes les paires de dépendance, on obtient finalement le graphe de dépendance approché suivant:

$$\begin{tikzcd} \langle QUOT(s(x),s(y),z), QUOT(x,y,z)\rangle \arrow[loop above] \arrow[d, shift right] \\ \langle QUOT(x,0,s(z)), QUOT(x,s(z),s(z))\rangle \arrow[u, shift right] \end{tikzcd}$$

Dans le cas particulier de l'exemple précédent, le graphe approché et le graphe de dépendance coïncident. Cependant, ce n'est pas toujours le cas comme on le voit dans l'exemple suivant.

Exemple 4.10 *Soit $\mathcal{F} = \{f : 3, 0 : 0, 1 : 0\}$ et $\mathcal{R}$ le système constitué d'une seule règle $f(0,1,x) \rightarrow f(x,x,x)$. La seule et unique paire de dépendance de ce système est $\langle F(0,1,x), F(x,x,x)\rangle$. Le graphe de dépendance approché est:*

$$\begin{tikzcd} \langle F(0,1,x), F(x,x,x)\rangle \arrow[loop above] \end{tikzcd}$$

En effet, REN(CAP($F(x,x,x)$*))* $= F(x_1, x_2, x_3)$ *s'unifie avec* $F(0,1,x)$. *Or, le graphe de dépendance réel ne contient aucun arc et donc aucun cycle, puisqu'il n'existe aucune substitution* σ *telle que* $x\sigma \rightarrow_{\mathcal{R}}{}^* 0$ *et* $x\sigma \rightarrow_{\mathcal{R}}{}^* 1$. *L'approximation avec REN et CAP donne ici de mauvais résultats à cause de la fonction REN qui a linéarisé la membre droit de la paire de dépendance.*

Une fois l'approximation du graphe de dépendance construite, pour prouver la terminaison d'un système de réécriture $\mathcal{R}$, il est suffisant de montrer qu'il n'existe pas de chaîne de dépendance infinie constituée de paires de dépendance provenant d'un cycle de l'approximation du graphe de dépendance. Dans le cas général d'une preuve de terminaison, il est nécessaire de prouver que toute chaîne de réécriture est *strictement* décroissante par rapport à un ordre nœthérien. En utilisant les paires de dépendance, la condition utilisant l'ordre strict est affaiblie: il est suffisant qu'une seule paire de dépendance, par cycle, soit *strictement* décroissante pour qu'on ait la terminaison du système de réécriture.

Théorème 4.2 *Soit* $\mathcal{R}$ *un système de réécriture. S'il existe un préordre de réduction* $\succeq$ *tel que*

- $l \succeq r$ *pour toute règle* $l \rightarrow r$ *de* $\mathcal{R}$, *et*
- $s \succeq t$ *pour toute paire de dépendance* $\langle s, t\rangle$ *sur un cycle du graphe de dépendance, et*
- $s \succ t$ *pour au moins une paire de dépendance* $\langle s, t\rangle$ *pour chaque cycle du graphe de dépendance,*

alors $\mathcal{R}$ *termine.*

Les conditions de ce théorème par rapport à celles de la proposition 2.3 sont affaiblies. En effet, pour prouver la terminaison grâce à ce théorème, il est suffisant de montrer la décroissance stricte sur un sous-ensemble des paires de dépendance alors que dans la proposition 2.3, la décroissance stricte est à montrer pour toutes les règles de réécriture. De plus, cette méthode permet d'utiliser un préordre de réduction à la place d'un ordre de réduction: dans le cas d'un préordre de réduction, la propriété de monotonie est exigée sur le préordre et non sur la relation stricte. La classe d'ordres utilisable est par conséquent beaucoup plus vaste. Ceci est un autre avantage intéressant de la méthode qui permet de venir à bout de beaucoup de cas difficiles; en voici un exemple.

Exemple 4.11 *Pour prouver la terminaison du système de réécriture de l'exemple 4.7, en utilisant le théorème précédent, il suffit de montrer qu'il existe un préordre de réduction* $\succeq$ *tel que:*

$$\begin{array}{l}
minus(x,0) \succeq x \\
minus(s(x), s(y)) \succeq minus(x,y) \\
quot(0, s(y)) \succeq 0 \\
quot(s(x), s(y)) \succeq s(quot(minus(x,y), s(y))) \\
MINUS(s(x), s(y)) \succ MINUS(x,y) \\
QUOT(s(x), s(y)) \succ QUOT(minus(x,y), s(y)).
\end{array}$$

où les deux dernières inégalités strictes correspondent aux cycles du graphe de dépendance obtenu dans l'exemple 4.7. En imposant une décroissance stricte de l'ordre pour ces paires de dépendance, on impose que toute chaîne de dépendance empruntant ces cycles soit elle-aussi strictement décroissante vis-à-vis de cet ordre. Il est intéressant de noter que cet ensemble d'inégalités ne peut être satisfait par le rpos. En effet, l'inégalité $QUOT(s(x), s(y)) \succ QUOT(minus(x,y), s(y))$ *est impossible à montrer puisque* $s(x) \not\succ_{rpos} minus(x,y)$ *et* $s(y) \not\succ_{rpos} minus(x,y)$. *En revanche, il* *existe un préordre de réduction basé sur une interprétation polynomiale satisfaisant ces inégalités. Il suffit d'interpréter* 0 *par lui-même,* $s(x)$ *par* $x+1$*, et d'interpréter minus, quot,* $MINUS$*, et* $QUOT$ *par la projection sur le premier argument. Il faut bien noter ici qu'en choisissant cette interprétation* $minus_{\mathbb{N}}(x,y) = x$ *on tire pleinement parti du fait que la propriété de monotonie d'un préordre de réduction est exigée sur le préordre et non sur la relation stricte. Si la propriété de monotonie était exigée sur la relation stricte nous ne pourrions pas utiliser une telle interprétation. En effet, pour tout* $a, b, c \in \mathbb{N}$*, tels que* $a >_{\mathbb{N}} b$ *alors on a* $minus_{\mathbb{N}}(c,a) = c \not>_{\mathbb{N}} c = minus_{\mathbb{N}}(c,b)$*, et donc avec cette interprétation, la relation* $>_{\mathbb{N}}$ *n'est pas monotone. En revanche, le préordre* $\geq_{\mathbb{N}}$ *est bien monotone puisque si* $a \geq_{\mathbb{N}} b$*, on a bien* $minus_{\mathbb{N}}(c,a) = c \geq_{\mathbb{N}} c = minus_{\mathbb{N}}(c,b)$*, donc* $minus_{\mathbb{N}}(a,c) = a \geq_{\mathbb{N}} b = minus_{\mathbb{N}}(b,c)$*.*

4.3.4 Raffinements de la méthode

Dans l'exemple précédent, la satisfaisabilité des contraintes d'ordre générées a été vérifiée par une interprétation polynomiale. Sur cet exemple précis, les interprétations peuvent être automatiquement générées par un logiciel de type POLO [Gie95b]. Cependant, les interprétations polynomiales n'offrent pas toujours de réponse satisfaisante de par leur indécidabilité. D'autre part, comme on l'a vu les ordres syntaxiques décidables tels *rpo*, *lpo*, *rpos* sont parfois trop forts pour vérifier la satisfaisabilité des contraintes. En effet, ces ordres doivent être monotones, ce qui n'est pas possible sur des exemples comme le précédent où, par contre, la monotonie sur le préordre permet de conclure.

Nous donnons ici la méthode de simplification des preuves de terminaison utilisant une transformation des systèmes par des *schémas de programmes récursifs* (RPS en abrégé) extraite de [Art97]. Cette méthode de simplification, même si elle semble indépendante de la méthode des paires de dépendance, ne trouve son intérêt que dans le cas de preuves les utilisant, i.e. où seul $l \succeq r$ est exigé pour chaque règle $l \to r$ du système au lieu de l'inégalité stricte $l \succ r$. Une preuve de terminaison basée sur les paires de dépendance, consiste en la construction d'un ensemble d'inégalités correspondant aux conditions du théorème précédent, puis à la résolution de ces inégalités par la recherche d'un préordre les satisfaisant.

Avant de les résoudre, T. Arts propose de simplifier les inégalités en les réécrivant à l'aide d'un schéma de programme récursif défini comme suit.

Définition 4.17 *(schéma de programme récursif) Un schéma de programme récursif est un système de réécriture contenant au plus une règle de réécriture*

$$f(x_1, \ldots, x_n) \to t$$

pour tout symbole $f \in \mathcal{F}^n$*, avec* $x_1, \ldots, x_n$ *des variables distinctes deux à deux et* t *un terme quelconque.*

Tout schéma de programme récursif est trivialement confluent. Sa terminaison est décidable. Si un schéma de programme récursif P termine alors il est convergent et tout terme t a une forme normale unique notée $t\downarrow_P$.

Théorème 4.3 *(T. Arts [Art97]) Soit* P *un schéma de programme récursif terminant et* I *un ensemble d'inégalités. S'il existe un préordre de réduction* $\succeq$ *qui satisfait*

$$\{s\downarrow_P \succ t\downarrow_P \mid s > t \in I\} \cup \{s\downarrow_P \succeq t\downarrow_P \mid s \geq t \in I\}$$

alors il existe un préordre de réduction satisfaisant I*.*

Exemple 4.12 *Soient les inégalités obtenues dans l'exemple 4.11:*

$$\begin{aligned}
&minus(x,0) \succeq x\\
&minus(s(x),s(y)) \succeq minus(x,y)\\
&quot(0,s(y)) \succeq 0\\
&quot(s(x),s(y)) \succeq s(quot(minus(x,y),s(y)))\\
&MINUS(s(x),s(y)) \succ MINUS(x,y)\\
&QUOT(s(x),s(y)) \succ QUOT(minus(x,y),s(y)).
\end{aligned}$$

Ces inégalités peuvent être transformées et simplifiées à l'aide du schéma de programme récursif suivant: $minus(x,y) \to x$. *Les inégalités deviennent:*

$$\begin{aligned}
&x \succeq x\\
&s(x) \succeq x\\
&quot(0,s(y)) \succeq 0\\
&quot(s(x),s(y)) \succeq s(quot(x,s(y)))\\
&MINUS(s(x),s(y)) \succ MINUS(x,y)\\
&QUOT(s(x),s(y)) \succ QUOT(x,s(y))
\end{aligned}$$

Alors qu'avant la transformation, aucun rpos ne pouvait satisfaire ces inégalités, maintenant c'est possible, par exemple avec toute précédence telle que $quot >_{\mathcal{F}} s$.

4.3.5 Conclusion

Pour conclure sur la méthode des paires de dépendance, nous pouvons dire que les trois grandes qualités de cette méthode pour la preuve de terminaison sont que

- les paires de dépendance, dans de très nombreux cas, simplifient la preuve de terminaison,
- si la preuve n'est pas plus simple, au pire elle est de même complexité que la preuve originale,
- la construction des paires de dépendance est facilement automatisable.

En revanche, il faut remarquer qu'en prouvant la terminaison à l'aide des paires de dépendance, aucun ordre de réduction n'est explicitement construit. Ceci est un handicap pour certaines méthodes de déduction automatique où un ordre explicite et calculable doit être produit, par exemple dans le cas de la complétion, où l'ordre est utilisé pour choisir l'orientation des nouvelles équations produites, et dans le cas de la preuve par induction par réécriture où l'ordre impose l'orientation des hypothèses d'induction en vue de leur utilisation pour la simplification des conjectures (voir sections 3.4.2, et 3.4.4).

4.4 L'ordre *gpo*

L'ordre *gpo* défini par N. Dershowitz et C. Hoot [DH95] est un ordre généralisant les principaux ordres connus. Le but du *gpo* est d'unifier bon nombre de définitions d'ordre et de permettre à l'utilisateur d'utiliser la quasi-intégralité des ordres en ne manipulant qu'une seule définition. Le résultat est un ordre dont la définition est, au regard de sa puissance, relativement simple. Il permet, en outre, de construire la preuve de terminaison progressivement, de par sa conception incrémentale. Nous verrons dans le chapitre 5 comment nous utilisons une approche contrainte sur cet ordre pour l'automatisation des preuves de terminaison.

4.4.1 Définition

L'intuition derrière la définition de l'ordre *gpo* est que beaucoup d'ordres utilisés ont des définitions très semblables. C'est le cas pour la majeure partie des ordres sur les chemins et il suffit, à titre d'exemple, de comparer les définitions des ordres *rpo*, *lpo*, et *rpos* données dans la section 4.1. Or, cette similarité ne s'arrête pas aux ordres utilisés dans cette thèse mais elle s'étend également à d'autres ordres tels *spo* [KL80], *kbo* [KB70]. L'approche utilisée par N. Dershowitz et C. Hoot a donc été de dégager les aspects communs à tous ces ordres et de paramétrer la définition du *gpo* par des ordres abstraits, nommés ordres composants, qui expriment la spécificité de chaque ordre particulier. Les ordres particuliers deviennent ainsi des instances de *gpo*. La définition du *gpo*, telle qu'elle est donnée dans [DH95], est basée sur les ordres composants ϕ_i, $i = 1 \dots k$ définis comme suit.

Définition 4.18 *Un ordre composant ϕ_i est une paire $\langle \theta_i, \geq_i \rangle$ telle que*

- *θ_i est un homorphisme de l'ensemble des termes vers une algèbre A et $\geq_i$ est un préordre bien-fondé sur A, ou*
- *θ_i est une fonction (nommée fonction d'extraction multi-ensemble) de l'ensemble des termes vers l'ensemble des multi-ensembles de sous-termes, i.e. $\theta_i(f(s_1, \dots, s_n)) = \{s_{j_1}, \dots, s_{j_m}\}$, tel que $j_1, \dots, j_m \in \{1, \dots, n\}$ et $\geq_i$ est l'extension multi-ensemble du gpo.*

Les θ_i sont appelées *fonctions de terminaison*. Dans la définition du *gpo*, les ordres composants $\phi_i = \langle \theta_i, \geq_i \rangle$ ne sont pas utilisés de façon isolée, mais regroupés en t-uples. Par souci d'homogénéité, les fonctions de terminaison θ_i sont également regroupées en t-uples et les préordres associés $\geq_i$ sont regroupés sous la forme d'une extension lexicographique. Par commodité, pour tout terme $s \in \mathcal{T}(\mathcal{F})$, on note $\Theta_{i,j}(s)$ le t-uple de fonctions de terminaison $\langle \theta_i(s), \dots, \theta_j(s) \rangle$, où $0 \leq i < j$ et $\theta_i, \dots, \theta_j$ sont des fonctions de terminaison. Soit $>_{lex}$ l'ordre associé, i.e. la combinaison lexicographique des ordres $>_i, \dots, >_j$, et soit $\simeq_{lex}$ l'équivalence associée, i.e. la combinaison lexicographique des relations d'équivalence $\simeq_i, \dots, \simeq_j$. Ces deux relations étendent la définition 4.3 page 33 de l'extension lexicographique. En effet, celles-ci sont basées sur la combinaison de plusieurs préordres $\geq_i$ pour $i = 1 \dots k$, au lieu d'une seule relation. Ainsi, pour tout i, j, l tels que $0 \leq i \leq j \leq k$ et $0 \leq i \leq l \leq k$ on a:

$$\langle \theta_i(s), \dots, \theta_j(s) \rangle >_{lex} \langle \theta_i(t), \dots, \theta_l(t) \rangle$$

si $\theta_i(s) >_i \theta_i(t)$ ou si $\theta_i(s) \simeq_i \theta_i(t)$ et $\langle \theta_{i+1}(s), \dots, \theta_j(s) \rangle >_{lex} \langle \theta_{i+1}(t), \dots, \theta_l(t) \rangle$. D'autre part, on a:

$$\langle \theta_i(s), \dots, \theta_j(s) \rangle \simeq_{lex} \langle \theta_i(t), \dots, \theta_l(t) \rangle$$

si $j = l$, $\theta_i(s) \simeq_i \theta_i(t)$ et $\langle \theta_{i+1}(s), \dots, \theta_j(s) \rangle \simeq_{lex} \langle \theta_{i+1}(t), \dots, \theta_l(t) \rangle$. Enfin, si $\langle\,\rangle$ représente le tuple vide et T n'importe quel tuple non vide, on a:

$$T >_{lex} \langle\,\rangle \quad \text{et} \quad \langle\,\rangle \simeq_{lex} \langle\,\rangle$$

On note $\geq_{lex}$ la relation $>_{lex} \cup \simeq_{lex}$. On abrégera $\Theta_{0,k}$ en Θ.

Définition 4.19 *L'ordre général sur les chemins (General Path Ordering [DH95]).*
Soit $\langle \theta_i, \geq_i \rangle$ $i = 0, \dots, k$, des ordres composants, $s, t \in \mathcal{T}(\mathcal{F})$ tels que $s = f(s_1, \dots, s_n)$ et

$t = g(t_1, \ldots, t_m)$. L'ordre général sur les chemins $\geq_{gpo}$ sur $\mathcal{T}(\mathcal{F})$ est défini inductivement par $\geq_{gpo} = >_{gpo} \cup \simeq_{gpo}$ où

- *$s >_{gpo} t$ si*
  1. *$s_i \geq_{gpo} t$ pour un sous-terme quelconque s_i de s, ou*
  2. *$s >_{gpo} t_1, \ldots, s >_{gpo} t_m$ et $\Theta(s) >_{lex} \Theta(t)$.*
- *$s \simeq_{gpo} t$ si*

  $s >_{gpo} t_1, \ldots, s >_{gpo} t_m$, $t >_{gpo} s_1, \ldots, t >_{gpo} s_n$ et $\Theta(s) \simeq_{lex} \Theta(t)$.

Cet ordre est entièrement paramétré par le couple $(\Theta, \geq_{lex})$. En choisissant une valeur particulière $\Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex})$ pour $(\Theta, \geq_{lex})$, on obtient une *instance* de *gpo*, i.e. un ordre particulier noté $\gtrapprox^{\Phi}_{gpo}$. Nous donnons maintenant un exemple simple d'instanciation du *gpo* ainsi qu'un exemple d'utilisation de l'instance pour la comparaison de deux termes. Le but n'étant pas, pour l'instant, de montrer la puissance de l'ordre mais plutôt le mécanisme d'instanciation et d'utilisation des ordres composants, nous avons volontairement choisi un exemple de système de réécriture et une instance de *gpo* déjà présentés (voir exemple 4.1).

Exemple 4.13 *Soit $\Phi = (\mathcal{T}_{0,1}, \gtrapprox_{lex})$ une instance de $(\Theta, \geq_{lex})$, telle que $\mathcal{T}_{0,1} = \langle \tau_0, \tau_1 \rangle$ et $\gtrapprox_{lex} = \succ_{lex} \cup \approx_{lex}$ où $\succ_{lex}$ est la combinaison lexicographique des ordres $\succ_0$, $\succ_1$ et $\approx_{lex}$ est la combinaison lexicographique des relations d'équivalence $\approx_0$ et $\approx_1$. Soit τ_0 la fonction $\mathcal{R}oot$, τ_1 la fonction qui à tout terme associe le multi-ensemble de ses sous-termes directs, $\succ_0$ un ordre sur $\mathcal{F}$, $\approx_0$ l'identité et $\gtrapprox_1 = \succ_1 \cup \approx_1$ où $\succ_1$ et $\approx_1$ sont les extensions multi-ensemble de $\succ^{\Phi}_{gpo}$ et $\approx^{\Phi}_{gpo}$ respectivement.*

L'ordre $\succ^{\Phi}_{gpo}$ défini n'est autre que le rpo (voir définition 4.5 page 34). Reprenons l'exemple 4.1, dans lequel $\mathcal{F} = \{app : 2, cons : 2, nil : 0\}$, et $\mathcal{R}$ est le système décrivant la fonction app de concaténation de deux listes:

$$\begin{array}{l} app(nil, x) \to x \\ app(cons(x, y), z) \to cons(x, app(y, z)). \end{array}$$

Supposons que l'ordre $\succ_0$ soit tel que $app \succ_0 cons$, $x \approx_0 x$ pour toute variable x et tel que x et y soient incomparables si x et y sont des variables distinctes. Pour la première règle, on prouve que

$$app(nil, x) \succ^{\Phi}_{gpo} x$$

par le cas 1. de la définition du gpo (propriété de sous-terme), puisque l'on a trivialement

$$x \approx^{\Phi}_{gpo} x$$

par le cas 3. de la définition et grâce au fait que $\theta_0(x) \approx_0 \theta_0(x)$ implique $\mathcal{T}_{0,1}(x) \approx_{lex} \mathcal{T}_{0,1}(x)$. Pour la deuxième règle, on souhaite montrer que

$$app(cons(x, y), z) \succ^{\Phi}_{gpo} cons(x, app(y, z)).$$

Or, on a

$$\mathcal{T}_{0,1}(app(cons(x, y), z)) >_{lex} \mathcal{T}_{0,1}(cons(x, app(y, z))) \tag{4.1}$$

puisque

$$\tau_0(app(cons(x,y),z)) = app \succ_0 cons = \tau_0(cons(x, app(y,z))).$$

D'où en appliquant le cas 2. de la définition du gpo avec (4.1), il reste à montrer que

$$app(cons(x,y),z) \succ^{\Phi}_{gpo} x \tag{4.2}$$

et

$$app(cons(x,y),z) \succ^{\Phi}_{gpo} app(y,z). \tag{4.3}$$

L'inégalité (4.2), est trivialement obtenue par le cas 1. Pour (4.3), on doit utiliser le cas 2. et montrer

$$\mathcal{T}_{0,1}(app(cons(x,y),z)) \succ_{lex} \mathcal{T}_{0,1}(cons(x, app(y,z))), \tag{4.4}$$

et

$$app(cons(x,y),z) \succ^{\Phi}_{gpo} y \text{ et } app(cons(x,y),z) \succ^{\Phi}_{gpo} z. \tag{4.5}$$

Les inégalités (4.5) sont montrées par le cas 1. du gpo. Pour (4.4), on applique la définition lexicographique de $\succ_{lex}$. Tout d'abord, les termes sont comparés vis-à-vis de l'ordre composant $\phi_0 = \langle \tau_0, \gtrsim_0 \rangle$. Or celui-ci ne suffit pas, en effet:

$$\tau_0(app(cons(x,y),z)) = app \approx_0 \tau_0(app(y,z)).$$

Il faut donc comparer les deux termes à l'aide du deuxième ordre composant $\phi_1 = \langle \tau_1, \gtrsim_1 \rangle$. Soit montrer que

$$\tau_1(app(cons(x,y),z)) \succ_1 \tau_1(app(y,z)).$$

Par définition de τ_1 et $\gtrsim_1$, ceci revient à montrer

$$\{\!\{cons(x,y), z\}\!\} \succ_1 \{\!\{y, z\}\!\}$$

qui est trivialement obtenu par le fait que

$$cons(x,y) \succ^{\Phi}_{gpo} y.$$

En choisissant des instances Φ particulières pour $(\Theta, \geq_{lex})$, on peut obtenir bien d'autres ordres tels que l'ordre lexicographique sur les chemins (*lpo*), l'ordre récursif sur les chemins avec statut (*rpos*), l'ordre sémantique sur les chemins (*spo*) [KL80], l'ordre de Knuth et Bendix (*kbo*) [KB70], ou encore le polynomial path ordering (Lankford [Lan79]). Pour plus de détails, le lecteur pourra se reporter à [DH95].

Le théorème suivant donne les conditions suffisantes pour prouver la terminaison d'un système de réécriture à l'aide de l'ordre *gpo*. Il est important de remarquer que le *gpo* est un ordre qui n'est défini que pour les termes clos. Ceci a son importance, notamment dans le théorème suivant où la condition essentielle à vérifier est que $l\sigma >_{gpo} r\sigma$ pour toute substitution σ. En pratique, ce théorème n'est pas utilisable tel quel, puisque cette condition est indécidable en général. Nous verrons dans la suite qu'il est possible d'étendre la définition du *gpo* aux termes avec variables, en faisant intervenir une forme de stabilité par instanciation. Ceci facilitera la vérification et donc l'utilisation du théorème suivant.

Théorème 4.4 *(Dershowitz & Hoot [DH95]) Soit $\geq_{gpo}$ un gpo. Un système de réécriture $\mathcal{R}$ termine sur $\mathcal{T}(\mathcal{F})$ si*

- *$l\sigma >_{gpo} r\sigma$ pour toute règle $l \to r$ de $\mathcal{R}$, et pour toute substitution close σ et,*

– *$\forall s, t \in \mathcal{T}(\mathcal{F})$, $s \rightarrow_R t$ et $s \geq_{gpo} t$ implique que $f(\ldots, s, \ldots) \geq_{gpo} f(\ldots, t, \ldots)$ pour tout contexte clos $f(\ldots \ldots)$.*

4.4.2 Le *gpo* en pratique

En pratique, pour prouver la terminaison d'un système de réécriture avec le *gpo*, l'approche usuelle consiste à définir une instance de *gpo*: $\succ_{gpo}^{\Phi}$ sur les termes clos et à utiliser implicitement $\succ_{gpo}^{\Phi}$ comme un ordre sur les termes avec variables, en prouvant que pour chaque règle $l \rightarrow r$ ($l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$) du système, on a $l \succ_{gpo}^{\Phi} r$ et $\forall s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\forall \sigma$ telle que $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $s \succ_{gpo}^{\Phi} t \implies s\sigma \succ_{gpo}^{\Phi} t\sigma$. Ici, pour plus de clarté, nous avons choisi de définir explicitement l'ordre *gpo* sur les termes avec variables. Tout d'abord, nous étendons la définition de Θ, de $>_{lex}$ et de $\simeq_{lex}$ sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$, de la façon suivante.

Définition 4.20 *Soient $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.*

– *$\Theta(s) >_{lex} \Theta(t)$ si pour toute substitution σ t.q. $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $\Theta(s\sigma) >_{lex} \Theta(t\sigma)$,*

– *$\Theta(s) \simeq_{lex} \Theta(t)$ si pour toute substitution σ t.q. $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $\Theta(s\sigma) \simeq_{lex} \Theta(t\sigma)$,*

– *$\Theta(s) \geq_{lex} \Theta(t)$ si $\Theta(s) >_{lex} \Theta(t)$ ou $\Theta(s) \simeq_{lex} \Theta(t)$.*

Grâce à cette extension des fonctions de terminaison sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$, la définition du "general path ordering" est également étendue à $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A partir de maintenant, nous utiliserons la définition de *gpo* sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Théorème 4.5 *Le gpo: $\geq_{gpo}$ est un préordre sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ ayant la propriété de sous-terme.*

Preuve Cette preuve est une adaptation directe des preuves de Dershowitz & Hoot [DH95] au cas non clos (voir section 5.8.1). $\square$

Dans la proposition suivante, nous montrons que *gpo* est un ordre stable par instanciation pour toute substitution associant un terme clos à tout terme.

Proposition 4.3 *(Stabilité close) Soient $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $\Phi = (\mathcal{T}_{0,k}, \succsim_{lex})$. Si $s \succ_{gpo}^{\Phi} t$ (resp. $s \approx_{gpo}^{\Phi} t$) alors pour toute substitution σ telle que $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $s\sigma \succ_{gpo}^{\Phi} t\sigma$ (resp. $s\sigma \approx_{gpo}^{\Phi} t\sigma$).*

Preuve Par induction sur la hauteur des termes des deux cotés de l'inégalité. Si $s = f(s_1, \ldots, s_n) \succ_{gpo}^{\Phi} g(t_1, \ldots, t_m) = t$ alors par cas sur la définition de $>_{gpo}$ sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$, soit:

1. $s_i \succsim_{gpo}^{\Phi} t$ pour un sous-terme s_i de s. Alors, en appliquant l'hypothèse d'induction, on obtient que $\forall \sigma$ t.q. $s_i\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $s_i\sigma \succsim_{gpo}^{\Phi} t\sigma$. Puisque $\forall \sigma$, $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$ implique $s_i\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, nous obtenons que $\forall \sigma$ t.q. $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $s_i\sigma \succsim_{gpo}^{\Phi} t\sigma$. D'où, par définition de *gpo*, $s\sigma \succ_{gpo}^{\Phi} t\sigma$.
2. $s \succ_{gpo}^{\Phi} t_1, \ldots, s \succ_{gpo}^{\Phi} t_m$ et $\Theta(s) >_{lex} \Theta(t)$. Alors, en appliquant l'hypothèse d'induction, on obtient que $\forall \sigma_1, \ldots, \sigma_m$ t.q. $s\sigma_1, \ldots, s\sigma_m, t_1\sigma_1, \ldots, t_m\sigma_m \in \mathcal{T}(\mathcal{F})$, on a $s\sigma_1 \succ_{gpo}^{\Phi} t_1\sigma_1, \ldots, s\sigma_m \succ_{gpo}^{\Phi} t_m\sigma_m$. A fortiori, $\forall \sigma$ t.q. $s\sigma, t_1\sigma, \ldots, t_m\sigma \in \mathcal{T}(\mathcal{F})$, on a $s\sigma \succ_{gpo}^{\Phi} t_1\sigma, \ldots, s\sigma \succ_{gpo}^{\Phi} t_m\sigma$. Comme dans le cas précédent, $\forall \sigma$, $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$ implique $s\sigma, t_1\sigma, \ldots, t_m\sigma \in \mathcal{T}(\mathcal{F})$. Donc, $\forall \sigma$ t.q. $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $s\sigma \succ_{gpo}^{\Phi} t_1\sigma, \ldots, s\sigma \succ_{gpo}^{\Phi} t_m\sigma$. D'autre part, par définition de Θ sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$, on obtient de $\Theta(s) >_{lex} \Theta(t)$ que $\forall \sigma$ t.q. $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, $\Theta(s\sigma) >_{lex} \Theta(t\sigma)$. D'où finalement $\forall \sigma$ t.q. $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $s\sigma \succ_{gpo}^{\Phi} t\sigma$.

La preuve est similaire dans le cas de $s \simeq_{gpo} t$. $\square$

Grâce à la proposition précédente, pour prouver la première condition du théorème 4.4 (théorème de terminaison), il est suffisant de prouver que: $l \succ_{gpo}^{\Phi} r$ pour toute règle $l \to r$ de $\mathcal{R}$. Il est intéressant de noter que, pour pouvoir être utilisable en pratique, une instance de *gpo* doit vérifier les conditions de la définition 4.20 page 50. En effet, si ces conditions ne sont pas vérifiées, pour deux termes donnés $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, montrer que $\Theta(s\sigma) >_{lex} \Theta(t\sigma)$ pour toute substitution close σ est indécidable en général. En pratique, ne considérer que les instances vérifiant les conditions de la définition 4.20 n'est donc pas une restriction.

Exemple 4.14 *Nous reprenons le système de l'exemple 4.7, et nous prouvons sa terminaison à l'aide du gpo. Soit* $\mathcal{F} = \{minus : 2, quot : 2, s : 1, 0 : 0\}$ *et* $\mathcal{R}$ *le système de réécriture, définissant la fonction quot telle que* $quot(x,y) = \lceil \frac{x}{y} \rceil$*:*

$$\begin{array}{l} minus(x,0) \to x \\ minus(s(x),s(y)) \to minus(x,y) \\ quot(0,s(y)) \to 0 \\ quot(s(x),s(y)) \to s(quot(minus(x,y),s(y))). \end{array}$$

Afin de prouver sa terminaison, nous utilisons une instance $\succsim_{gpo}^{\Phi}$ *de gpo particulièrement utile, proche de l'instance "natural path ordering" utilisée par N. Dershowitz et C. Hoot dans [DH95]. Dans cette instance* Φ *du gpo, le premier ordre composant est* $\phi_0 = \langle \theta_0, \geq_0 \rangle = \langle \mathcal{R}oot, \geq_{\mathcal{F}} \rangle$ *une précédence, le deuxième ordre composant est* $\phi_1 = \langle \theta_1, \geq_1 \rangle$ *où* $\geq_1$ *est le préordre usuel sur* $\mathbb{N}$ *et* θ_1 *est le* morphisme conservatif *sur* $\mathbb{N}$ *(ou "value preserving" [DH95]) tel que*

$$\begin{array}{l} \theta_1(0) = 0, \\ \theta_1(s(x)) = \theta_1(x) + 1, \\ \theta_1(minus(x,y)) = \left\{ \begin{array}{l} \theta_1(x) - \theta_1(y) \ si\ \theta_1(x) \geq \theta_1(y) \\ 0 \ si\ \theta_1(x) < \theta_1(y) \end{array} \right. , \\ \theta_1(quot(x,y)) = \left\{ \begin{array}{l} \lceil \frac{\theta_1(x)}{\theta_1(y)} \rceil \ si\ \theta_1(y) > 0 \\ 0 \ si\ \theta_1(y) = 0 \end{array} \right. . \end{array}$$

Par morphisme conservatif, on entend que pour toute règle $l \to r$ *du système de réécriture* $\mathcal{R}$ *on a* $\theta_1(l\sigma) \simeq_1 \theta_1(r\sigma)$ *pour toute substitution* σ*. C'est bien entendu le cas pour ce système puisque le morphisme choisi n'est autre que le modèle attendu du système* $\mathcal{R}$*. Nous le montrons maintenant pour chaque règle de réécriture:*

Règle 1. *on a* $\theta_1(x) \geq 0$*, et donc* $\theta_1(minus(x,0)) = \theta_1(x) - 0 \simeq_1 \theta_1(x)$*,*

Règle 2. *si* $\theta_1(x) \geq \theta_1(y)$*, on a* $\theta_1(x) + 1 \geq \theta_1(y) + 1$ *et donc*

$\theta_1(minus(s(x),s(y))) = (\theta_1(x) + 1) - (\theta_1(y) + 1) \simeq_1 \theta_1(x) - \theta_1(y)$.

Par contre si $\theta_1(x) < \theta_1(y)$*, alors* $\theta_1(x) + 1 < \theta_1(y) + 1$ *et donc*

$\theta_1(minus(s(x),s(y))) = 0 \simeq_1 \theta_1(x) - \theta_1(y)$

Règle 3. $\theta_1(quot(0,s(y))) = \lceil \frac{0}{\theta_1(y)+1} \rceil = 0 \simeq_1 \theta_1(0)$

Règle 4. *si* $\theta_1(x) > \theta_1(y)$*, on a*

$\theta_1(quot(s(x),s(y))) = \lceil \frac{\theta_1(x)+1}{\theta_1(y)+1} \rceil \simeq_1 \lceil \frac{\theta_1(x)-\theta_1(y)}{\theta_1(y)+1} \rceil + 1 = \theta_1(s(quot(minus(x,y),s(y))))$

Par contre si $\theta_1(x) \leq \theta_1(y)$*, on a*

$\theta_1(quot(s(x),s(y))) = \lceil \frac{\theta_1(x)+1}{\theta_1(y)+1} \rceil = 1 \simeq_1 \lceil \frac{0}{\theta_1(y)+1} \rceil + 1 = \theta_1(s(quot(minus(x,y),s(y))))$

De plus, les ordres composants $\phi_i = \langle \theta_i, \geq_i \rangle$, $i = 2 \ldots n$ sont tels que $\theta_i(s)$ est la fonction d'extraction du $i-1$ème sous-terme de s et $\geq_i$ est l'ordre gpo $\succsim^{\Phi}_{gpo}$ lui-même. Ceci correspond en fait à une comparaison lexicographique des sous-termes telle que l'on peut la trouver dans le lpo ou dans le rpos avec un opérateur ayant un statut "left".

La première et la troisième règle du système sont trivialement orientables avec toute instance de gpo, grâce au cas (1) de l'ordre: la propriété de sous-terme. Pour la deuxième règle, on a

$\theta_0(minus(s(x), s(y))) = minus \simeq_0 minus = \theta_0(minus(x, y))$

$\theta_1(minus(s(x), s(y))) = \theta_1(minus(x, y))$puisque θ_1 est conservatif

$\theta_2(minus(s(x), s(y)) = s(x) \succ^{\Phi}_{gpo} x = \theta_2(minus(x, y))$

ce qui signifie que globalement on a:

$\Theta(minus(s(x), s(y))) >_{lex} \Theta(minus(x, y))$.

Or, on a de plus $minus(s(x), s(y)) \succ^{\Phi}_{gpo} x$ et $minus(s(x), s(y)) \succ^{\Phi}_{gpo} y$, d'où

$minus(s(x), s(y)) \succ^{\Phi}_{gpo} minus(x, y)$.

Enfin pour la dernière règle, pour toute précédence $\geq_0$ telle que $quot >_0 s$ on a:

$\theta_0(quot(s(x), s(y))) = quot >_0 s = \theta_0(s(quot(minus(x, y), s(y))))$

Il reste donc à montrer que

$quot(s(x), s(y))) \succ^{\Phi}_{gpo} quot(minus(x, y), s(y))$

Ceci est obtenu avec le deuxième ordre composant. En effet, si les interprétations des deux termes par θ_0 sont équivalentes, ce n'est pas le cas avec θ_1:

$\theta_1(quot(s(x), s(y))) = \lceil \frac{\theta_1(x)+1}{\theta_1(y)+1} \rceil >_1 \lceil \frac{\theta_1(x)-\theta_1(y)}{\theta_1(y)+1} \rceil = \theta_1(quot(minus(x, y), s(y)))$

A noter que si $x \leq y$ alors $\lceil \frac{\theta_1(x)+1}{\theta_1(y)+1} \rceil = 1 > 0 = \lceil \frac{0}{\theta_1(y)+1} \rceil$. Enfin, pour terminer la preuve il reste à montrer que $quot(s(x), s(y)) \succ^{\Phi}_{gpo} minus(x, y)$ et $quot(s(x), s(y)) \succ^{\Phi}_{gpo} s(y)$, qui peut être aisément montré avec toute précédence $\geq_0$ telle que $quot >_0 s$ et $quot >_0 minus$.

4.4.3 Conclusion

La preuve de l'exemple précédent fait intervenir une interprétation plus complexe que celle utilisée dans la méthode des paires de dépendance. Mais cette interprétation a, en revanche, deux qualités indéniables. Tout d'abord, elle est naturelle, en ce sens que chaque terme est interprété par son modèle attendu et que ceci suffit à prouver la terminaison du système. Ceci n'est en général pas possible avec les paires de dépendance. Enfin, l'ordre *gpo* est explicitement construit et donc réutilisable dans une procédure de complétion ou un système de preuve par réécriture.

4.5 Ordres calculables et systèmes conditionnels

4.5.1 Ordres calculables

Toutes les méthodes de preuve de terminaison ne construisent pas nécessairement l'ordre de réduction orientant les règles de réécriture du système. Les méthodes basées sur la recherche d'un ordre particulier sur les règles tel *lpo* ou *gpo* construisent explicitement un ordre $>$ *calculable* prouvant la terminaison du système, i.e. pour tout termes s, t il est possible de décider si $s > t$ ou non. En revanche, ce n'est pas le cas des méthodes par transformation et de la méthode des paires de dépendance qui construisent respectivement un ordre pour le système transformé et pour les paires de dépendances. Notons tout de même que, dans le cas d'une preuve par transformation, même si l'ordre n'est pas explicitement construit pendant la preuve, celui-ci reste calculable. En effet, en admettant que P soit le système transformant et $>$ l'ordre utilisé pour orienter le système transformé, il est possible de définir et de calculer un ordre $>_P$ par $s >_P t$ si $s\downarrow_P > t\downarrow_P$.

La construction d'un tel ordre calculable sur les règles n'est pas possible dans le cas d'une preuve de terminaison avec la méthode des paires de dépendance.

4.5.2 Systèmes conditionnels

Dans le cas des systèmes conditionnels, même si la terminaison de la relation $\rightarrow_{\mathcal{R}}$ est nécessaire, elle n'est pas suffisante pour montrer que tout terme a une forme normale. Par exemple, soit le système $\mathcal{R} = \{a \rightarrow b \text{ if } a \downarrow b\}$ défini sur $\mathcal{F} = \{a : 0, b : 0\}$. La relation $\rightarrow_{\mathcal{R}}$ est bien-fondée[3], mais il est impossible de savoir si le terme a a une forme normale. Il est même impossible de savoir si le terme a est réductible. La propriété usuellement utilisée dans le cas des systèmes conditionnels et assurant à la fois la terminaison de la relation et la décidabilité de la relation est la propriété dite de *décroissance* [DOS88] définie comme suit.

Définition 4.21 *(Système décroissant [DOS88]) Un système de réécriture conditionnel est* décroissant *s'il existe un ordre bien-fondé $\succ$ ayant la propriété de sous-terme tel que $\rightarrow_{\mathcal{R}} \subseteq \succ$ et $l\sigma \succ s_1\sigma, t_1\sigma, \ldots, s_n\sigma, t_n\sigma$ pour toute règle $l \rightarrow r \text{ if } s_1 \downarrow t_1, \ldots, s_n \downarrow t_n$ et pour toute substitution σ.*

Il existe d'autres approches assurant la décidabilité du calcul tels les *systèmes simplifiants* [Kap84, Kap87], ou encore les *systèmes réducteurs* [JW86], mais elles sont des cas particuliers de la notion de système décroissant. La propriété de décroissance assure la décidabilité de toutes les notions de base de la réécriture conditionnelle.

Proposition 4.4 *[DOS88] Si $\mathcal{R}$ est un système décroissant, la relation $\rightarrow_{\mathcal{R}}$ termine et pour tous termes $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, les problèmes suivant sont décidables:*

1. *réduction en un pas: peut-on réécrire s en t en un pas, i.e. $(s,t) \in \rightarrow_{\mathcal{R}}$?*
2. *réduction finie: peut-on réécrire s en t en un nombre fini de pas, i.e. $(s,t) \in \rightarrow_{\mathcal{R}}^*$?*
3. *joignabilité: existe-il un terme $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ tel que $s \rightarrow_{\mathcal{R}}^* u$ et $t \rightarrow_{\mathcal{R}}^* u$?*
4. *forme normale et réductibilité: s est-il en forme normale?*

4.5.3 Paires de dépendance, *gpo* et systèmes conditionnels

L'ordre *gpo* et la méthode des paires de dépendance sont deux des plus puissantes méthodes de preuve de terminaison présentant de bonnes propriétés pour l'automatisation. Dans la mesure où nous souhaitons également manipuler des systèmes de réécriture conditionnelle, nous étudions ici l'utilisation des paires de dépendance et de l'ordre *gpo* pour montrer la décroissance. Ceci n'est étudié ni dans les travaux traitant des paires de dépendance ni dans ceux traitant de l'ordre *gpo*. Aussi proposons-nous une adaptation de ces deux approches à la réécriture conditionnelle, avant de développer plus amplement nos travaux à partir du *gpo*. Nous rappelons que parmi les propriétés souhaitées pour l'ordre $\succ$ utilisé dans la définition 4.21 page 53, figurent la propriété de sous-terme et la propriété $\rightarrow_{\mathcal{R}} \subseteq \succ$. L'ordre *gpo* a la propriété de sous-terme. En revanche, il n'est pas monotone. Seul le préordre $\geq_{gpo}$ est monotone. En conséquence, seule la propriété $\rightarrow_{\mathcal{R}} \subseteq \geq_{gpo}$ est assurée, ce qui n'est pas suffisant pour montrer la décroissance d'un système selon la définition 4.21 page 53. Pour ce qui est des paires de dépendance, aucun ordre $\succ$ n'est

3. il est facile de trouver un ordre de simplification $\succ$ tel que $a \succ b$

explicitement engendré, mais si l'on s'en tient aux conditions du théorème 4.2, la condition $l \succeq r$ est suffisante. En conséquence, pour une réécriture s'effectuant à la racine d'un terme s, i.e. si $s = l\sigma$ et $l \to r \in \mathcal{R}$, on aura $s \to_{\mathcal{R}} t$ et $s \succeq t$. Là encore, ceci est insuffisant pour montrer la propriété de décroissance.

Cependant, dans les deux cas, il est facile de définir un ordre, explicitement dans le cas du *gpo* et implicitement dans le cas des paires de dépendance, tel que la décroissance puisse être montrée à l'aide de ces méthodes.

Proposition 4.5 *Soit $\mathcal{R}$ un système de réécriture conditionnel, et Φ une instance de gpo telle que*

1. $s \to_{\mathcal{R}} t$ *et* $s \succsim^{\Phi}_{gpo} t$ *implique* $f(\dots s \dots) \succsim^{\Phi}_{gpo} f(\dots t \dots)$,

2. $l\sigma \succ^{\Phi}_{gpo} r\sigma$, *et* $l\sigma \succ^{\Phi}_{gpo} s_1\sigma, t_1\sigma, \dots, s_n\sigma, t_n\sigma$ *pour toute règle* $l \to r$ if $s_1 \downarrow t_1, \dots, s_n \downarrow t_n$ *de* $\mathcal{R}$ *et pour toute substitution close* σ,

alors $\mathcal{R}$ est un système décroissant.

Preuve On utilise un ordre $\gg^{\Phi}_{gpo}$ qui est sous-entendu dans la preuve du théorème 2 de [DH95].

Pour tout terme $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ on définit le multi-ensemble $P(s)$ par

- $P(s) = \{\!\{s\}\!\}$ si $s = x \in \mathcal{X}$,
- $P(s) = \{\!\{f(s_1, \dots, s_n)\}\!\} \cup \bigcup_{i=1}^{n} P(s_i)$ si $s = f(s_1, \dots, s_n)$, $f \in \mathcal{F}^n$ et tous $s_1, \dots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

Soit $\gg^{\Phi}_{gpo}$ l'ordre défini par $s \gg^{\Phi}_{gpo} t$ si $P(s) \succ^{\Phi,mul}_{gpo} P(t)$, où $\succ^{\Phi,mul}_{gpo}$ est l'extension multi-ensemble de $\succ^{\Phi}_{gpo}$.

- L'ordre $\gg^{\Phi}_{gpo}$ est bien fondé puisque l'extension multi-ensemble de $\succ^{\Phi}_{gpo}$ l'est.
- L'ordre $\gg^{\Phi}_{gpo}$ a la propriété de sous terme puisque l'on a trivialement $P(f(\dots s \dots)) = \{\!\{f(\dots s \dots), s, \dots\}\!\} \succ^{\Phi,mul}_{gpo} \{\!\{s, \dots\}\!\} = P(s)$.
- Nous prouvons maintenant que $\to_{\mathcal{R}} \subseteq \gg^{\Phi}_{gpo}$. Si $s \succ^{\Phi}_{gpo} t$, grâce à la propriété de sous-terme de l'ordre *gpo*, on a $\{\!\{s\}\!\} \succ^{\Phi,mul}_{gpo} P(t)$. On a donc $s \succ^{\Phi}_{gpo} t \implies s \gg^{\Phi}_{gpo} t$. Soit $s = C[l\sigma] \to C[r\sigma] = t$. Par la condition 2. de la proposition, on obtient que $l\sigma \succ^{\Phi}_{gpo} r\sigma$, et par le résultat précédent, on obtient $P(l\sigma) \succ^{\Phi,mul}_{gpo} P(r\sigma)$. Ensuite par la condition 1. de la proposition on obtient que $f(\dots l\sigma \dots) \succsim^{\Phi}_{gpo} f(\dots r\sigma \dots)$, d'où $P(f(\dots l\sigma \dots)) = \{\!\{f(\dots l\sigma \dots)\}\!\} \cup P(\dots) \cup P(l\sigma) \succ^{\Phi,mul}_{gpo} \{\!\{f(\dots r\sigma \dots)\}\!\} \cup P(\dots) \cup P(r\sigma)$. On a donc $f(\dots l\sigma \dots) \gg^{\Phi}_{gpo} f(\dots r\sigma \dots)$. Par construction, on obtient ainsi $C[l\sigma] \to C[r\sigma] \implies C[l\sigma] \gg^{\Phi}_{gpo} C[r\sigma]$, pour tout contexte $C[\,]$ et toute substitution σ, d'où finalement $\to_{\mathcal{R}} \subseteq \gg^{\Phi}_{gpo}$.

En résumé, l'ordre $\gg^{\Phi}_{gpo}$ est bien-fondé, il a la propriété de sous-terme, et le système $\mathcal{R}$ est tel que $\to_{\mathcal{R}} \subseteq \gg^{\Phi}_{gpo}$ et $l\sigma \gg^{\Phi}_{gpo} s_1\sigma, t_1\sigma, \dots, s_n\sigma, t_n\sigma$ pour toute règle $l \to r$ if $s_1 \downarrow t_1, \dots, s_n \downarrow t_n$ de $\mathcal{R}$, d'où $\mathcal{R}$ est un système décroissant. $\square$

La méthode des paires de dépendance n'est pas directement utilisable pour montrer la terminaison des systèmes conditionnels. Cependant, nous donnons ici une méthode naturelle permettant de traiter le cas conditionnel, en ramenant le problème de la décroissance d'un système conditionnel à la terminaison d'un système non conditionnel.

Proposition 4.6 *Soit $\mathcal{R}$ un système de réécriture conditionnel et $\mathcal{R}'$ le système de réécriture non conditionnel $\mathcal{R}' = \{l \to r, l \to s_1, l \to t_1, \ldots l \to s_n, l \to t_n \mid l \to r \text{ if } s_1 \downarrow t_1, \ldots, s_n \downarrow t_n \in \mathcal{R}\}$. Si $\mathcal{R}'$ termine alors $\mathcal{R}$ est décroissant.*

La preuve de cette proposition est simple; elle est basée sur des arguments semblables à ceux qui sont utilisés dans les preuves de [DOS88].

Preuve Soit $>_{\mathcal{R}'}$ la relation $\to^+_{\mathcal{R}'}$. La relation $>_{\mathcal{R}'}$ est un ordre bien fondé puisque $\to^+_{\mathcal{R}'}$ est une relation transitive et bien-fondée ($\mathcal{R}'$ termine). De plus, l'ordre $>_{\mathcal{R}'}$ est trivialement monotone puisque $\to^+_{\mathcal{R}'}$ l'est. Du fait que $>_{\mathcal{R}'}$ est bien fondé et monotone, pour tout contexte C on a $s \not>_{\mathcal{R}'} C[s]$. Il est donc possible d'étendre $>_{\mathcal{R}'}$ en un ordre $\succ$ bien fondé et ayant la propriété de sous-terme. De plus, pour toute règle $l \to r \text{ if } s_1 \downarrow t_1, \ldots, s_n \downarrow t_n \in \mathcal{R}$ on a $l\sigma \to_{\mathcal{R}'} s_1\sigma, t_1\sigma, \ldots s_n\sigma, t_n\sigma$, et comme $\to_{\mathcal{R}'} \subseteq >_{\mathcal{R}'} \subseteq \succ$, on a bien $l\sigma \succ s_1\sigma, t_1\sigma, \ldots, s_n\sigma, t_n\sigma$ pour toute substitution σ. D'où, finalement, l'ordre $\succ$ montre la décroissance du système $\mathcal{R}$. $\square$

Il est clair que cette proposition pouvait être également utilisée dans le cas du *gpo*, mais nous avons préféré traiter celui-ci séparément, la construction explicite de l'ordre utilisé dans la proposition 4.5 pouvant s'avérer utile en pratique lorsque l'ordre doit être calculable.

4.6 Modularité des preuves de terminaison

Nous rappelons d'abord que la signature $\mathcal{F}$ d'un système de réécriture peut être partitionnée en un ensemble de *symboles définis* $\mathcal{D} = \{\mathcal{R}oot(l) \mid l \to r \in R\}$ et un ensemble de *symboles constructeurs* $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. Dans les parties de cette thèse traitant de modularité, nous utiliserons la notation $(\mathcal{F}, \mathcal{R})$ pour un système de réécriture défini sur une signature $\mathcal{F}$. Soient $(\mathcal{F}_1, \mathcal{R}_1)$ et $(\mathcal{F}_2, \mathcal{R}_2)$ des systèmes de réécriture dont les ensembles de symboles définis et ensembles de constructeurs sont respectivement $\mathcal{D}_1$, $\mathcal{C}_1$, $\mathcal{D}_2$ et $\mathcal{C}_2$. Les systèmes $(\mathcal{F}_1, \mathcal{R}_1)$ et $(\mathcal{F}_2, \mathcal{R}_2)$ sont dits *disjoints* si $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$; les systèmes sont dits *à constructeurs partagés* si $\mathcal{F}_1 \cap \mathcal{F}_2 = \mathcal{C}_1 \cap \mathcal{C}_2$, et les systèmes sont dits *hiérarchiques* si $\mathcal{F}_1 \cap \mathcal{D}_2 = \emptyset$ and $\mathcal{R}_2 \subset \mathcal{T}(\mathcal{F}_2 \setminus \mathcal{D}_1, \mathcal{X}) \times \mathcal{T}(\mathcal{F}_2, \mathcal{X})$. Nous donnons maintenant un exemple de système à constructeurs partagés.

Exemple 4.15 *Soient $\mathcal{F}_1 = \{app : 2, cons : 2, nil : 0\}$, $\mathcal{F}_2 = \{rev : 1, rev2 : 2, cons : 2, nil : 0\}$, $\mathcal{R}_1$ et $\mathcal{R}_2$ les ensembles de règles de réécriture définissant respectivement la concaténation et l'inversion de listes.*

$$\mathcal{R}_1 = \left\{ \begin{array}{l} app(nil, x) \to x \\ app(cons(x, y), z) \to cons(x, app(y, z)) \end{array} \right. \quad et\ \mathcal{R}_2 = \left\{ \begin{array}{l} rev(x) \to rev2(x, nil) \\ rev2(nil, x) \to x \\ rev2(cons(x, y), z) \to rev2(y, cons(x, z)) \end{array} \right.$$

Les systèmes $(\mathcal{F}_1, \mathcal{R}_1)$ et $(\mathcal{F}_2, \mathcal{R}_2)$ ne partagent que des constructeurs. Cet exemple est représentatif de ce qu'il est possible de coder à l'aide de systèmes à constructeurs partagés: deux modules travaillant sur une même structure de donnée, ici une liste.

Pour des exemples hiérarchiques et disjoints, voir respectivement le premier exemple[4] de la section 6.6.5 page 137 et l'exemple 4.16 de la section suivante.

4. Dans cet exemple, il faut remarquer que l'ordre des systèmes $\mathcal{R}_1$ et $\mathcal{R}_2$ est inversé, i.e. on a $\mathcal{F}_1 \cap \mathcal{D}_2 = \emptyset$ and $\mathcal{R}_2 \subset \mathcal{T}(\mathcal{F}_1 \setminus \mathcal{D}_2, \mathcal{X}) \times \mathcal{T}(\mathcal{F}_1, \mathcal{X})$.

4.6.1 Une mauvaise nouvelle

Pour tout problème complexe, comme l'est une preuve de terminaison, il est naturel d'envisager sa résolution en le divisant au préalable en sous-problèmes plus simples. Dans le cas des systèmes de réécriture qui nous intéressent, ceci est d'autant plus naturel qu'ils représentent des spécifications algébriques conçues de façon modulaire. Idéalement, la division en sous-problèmes peut suivre le découpage en modules de la spécification. Cependant, la propriété de terminaison n'est pas modulaire en général pour les systèmes de réécriture même si leurs signatures sont disjointes, comme l'a montré Toyama [Toy86] grâce à son contre-exemple, désormais célèbre:

Exemple 4.16 *Soit $\mathcal{F}_1 = \{f : 3\}$, $\mathcal{F}_2 = \{g : 2, 0 : 0, 1 : 0\}$, et $\mathcal{R}_1$, $\mathcal{R}_2$ les ensembles de règles de réécriture définis respectivement sur les signatures $\mathcal{F}_1$ et $\mathcal{F}_2$*

$$\mathcal{R}_1 = \left\{ \; f(0,1,x) \to f(x,x,x) \right. \quad \text{et} \;\; \mathcal{R}_2 = \left\{ \begin{array}{l} g(x,y) \to x \\ g(x,y) \to y \end{array} \right.$$

Les systèmes $(\mathcal{F}_1,\mathcal{R}_1)$ et $(\mathcal{F}_2,\mathcal{R}_2)$ terminent, ils sont disjoints, mais leur union $(\mathcal{F},\mathcal{R}) = (\mathcal{F}_1 \cup \mathcal{F}_2, \mathcal{R}_1 \cup \mathcal{R}_2)$ ne termine pas puisqu'il existe une chaîne infinie de réécriture:

$$f(g(0,1),g(0,1),g(0,1)) \to f(0,g(0,1),g(0,1)) \to f(0,1,g(0,1)) \to f(g(0,1),g(0,1),g(0,1)) \to \dots$$

Beaucoup de travaux ont été dédiés à la recherche de conditions suffisantes sur les systèmes $\mathcal{R}_1$ et $\mathcal{R}_2$ pour que leur union conserve la propriété de terminaison [Rus87, Dro89, Mid89, Ohl94]. Leurs résultats sont condensés dans le théorème suivant emprunté à E. Ohlebusch [Ohl94].

Théorème 4.6 *Soient $\mathcal{F}_1$, $\mathcal{F}_2$ des signatures, $\mathcal{R}_1$ et $\mathcal{R}_2$ des systèmes de réécriture terminants disjoints définis respectivement sur $\mathcal{F}_1$ et $\mathcal{F}_2$. L'union de $\mathcal{R}_1$ et $\mathcal{R}_2$ est terminante si l'une des conditions suivantes est vérifiée:*

1. *$\mathcal{R}_1$ et $\mathcal{R}_2$ ne contiennent pas de règles effondrantes [Rus87] (i.e. des règles du type $C[x] \to x$ où $C[\,]$ est un contexte),*

2. *$\mathcal{R}_1$ et $\mathcal{R}_2$ ne contiennent pas de règles duplicantes [Rus87, Dro89] (i.e. des règles du type $l \to r$ telles qu'il existe une variable x dont le nombre d'occurrences dans r est supérieur à son nombre d'occurrence dans l),*

3. *l'un des systèmes ne contient ni règles effondrantes ni règles duplicantes [Mid89].*

D'une façon générale pour un survol sur la modularité des propriétés des systèmes de réécriture et la modularité de la propriété de terminaison en particulier, nous conseillons au lecteur de consulter [Gra96, Ohl94, Mid90].

En pratique, même sans considérer les conditions du théorème précédent comme nécessaires pour qu'il y ait modularité, la restriction aux systèmes disjoints est beaucoup trop forte si l'on souhaite l'utiliser dans le cadre de spécifications par réécriture construites de façon modulaire. En effet, une spécification modulaire correspond plutôt à une combinaison hiérarchique de systèmes de réécriture.

Dans certains cas, où par exemple plusieurs modules travaillent sur une même structure de données, une combinaison à constructeurs partagés peut être suffisante mais très peu de cas pratiques peuvent être ramenés à une combinaison de systèmes disjoints. L'absence de modularité de la propriété de terminaison met l'accent sur le fait que pour traiter des spécifications de taille intéressante, il est *indispensable* d'automatiser la production de l'ordre, et plus encore, il faut que cette production soit très efficace. Dans le cas de la preuve de terminaison de programmes fonctionnels non-mutuellement récursifs, comme cela est fait dans [Wal94], l'efficacité n'est pas un problème crucial puisque la terminaison est une propriété totalement modulaire. Dans ce contexte, la terminaison d'une fonction f est prouvée de façon définitive. Si f est appelée depuis une autre fonction g, la preuve de terminaison de f n'est pas à reconsidérer, seule la terminaison de la fonction g est à établir. Dans notre cas, toute spécification terminante étendue par une nouvelle fonction doit être réanalysée et sa terminaison prouvée à nouveau, d'où la nécessité d'une production d'ordres de terminaison efficace.

Pour le cas des combinaisons hiérarchiques de deux systèmes $\mathcal{R}_1$ et $\mathcal{R}_2$, dans [Der95], N. Dershowitz propose différentes conditions suffisantes pour la modularité de la terminaison dans le cas de systèmes à constructeurs partagés, hiérarchiques et également dans le cas général. En particulier, N. Dershowitz étudie différentes combinaisons intéressantes de ces conditions pour lesquelles la terminaison de la combinaison hiérarchique est modulaire. Cependant, si l'on s'intéresse à des systèmes de réécriture représentant des programmes, les conditions utilisées sont restrictives: règles non-effondrantes, règles non-duplicantes, pas de symboles constructeurs en tête d'un membre droit de règle, uniquement des symboles constructeurs sous un symbole fonctionnel, ou encore systèmes confluents (voir [Der95] pour plus de détails).

4.6.2 Une alternative pratique

Dans certains cas, où seule la propriété de terminaison du système de réécriture sous une certaine stratégie est nécessaire, il est possible d'utiliser d'autres formes de modularité afin de réaliser les preuves de terminaison de façon incrémentale. Au lieu de considérer des restrictions sur le type de systèmes que l'on peut combiner pour que la propriété de terminaison soit modulaire, nous proposons de restreindre la relation de réécriture à l'aide d'une stratégie assurant de bonnes propriétés pour la modularité de la terminaison: la stratégie de réduction séquentielle (voir définition 3.5 page 24).

Théorème 4.7 *(Kurihara & Kaji [KK90]) Si $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ sont des systèmes de réécriture disjoints deux à deux, la relation de réduction modulaire $\Rrightarrow$ termine sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

Une autre preuve de ce théorème figure dans la thèse de A. Middeldorp [Mid90]. Il faut remarquer qu'aucune hypothèse de terminaison des systèmes $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ n'est explicitement demandée par ce théorème. La seule hypothèse est que pour tout terme s on sache trouver un terme t tel que $s \Rrightarrow t$ s'il existe, c'est à dire décider s'il existe un $j \in \{1, \ldots, n\}$ et un terme t tel que $s \to^+_{\mathcal{R}_j} t$ et $t \in IRR(\mathcal{R}_j)$. En clair, il est nécessaire que chaque système soit au moins faiblement terminant et que l'on connaisse pour chaque système $\mathcal{R}_j$ la stratégie menant tout terme à sa forme normale pour $\mathcal{R}_j$. Un autre résultat de [KK90] est que $\Rrightarrow$ est convergente si $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ sont au moins semi-convergents (confluents et faiblement terminants). Tous ces résultats ont étés étendus au cas des systèmes à constructeurs partagés.

Théorème 4.8 *(Kurihara & Ohuchi [KO91]) Si $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ sont des systèmes de réécriture à constructeurs partagés (deux à deux), alors la relation de réduction modulaire $\Rrightarrow$ termine sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

Une autre preuve de ce théorème peut être trouvée dans [Ohl94], avec l'hypothèse additionnelle que $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ soient au moins faiblement terminants. De la même façon, $\Rrightarrow$ est convergente si $(\mathcal{F}_1, \mathcal{R}_1), \ldots, (\mathcal{F}_n, \mathcal{R}_n)$ sont au moins semi-convergents [Ohl94].

Cependant, toujours dans le cas de spécifications algébriques, les combinaisons utiles sont rarement limitées au cas de systèmes à constructeurs partagés. En pratique, les combinaisons intéressantes sont hiérarchiques, voire quelconques. Or, il existe un contre-exemple simple à la terminaison de la relation $\Rrightarrow$ dans le cas de combinaison de systèmes hiérarchiques.

Exemple 4.17 *Soit* $\mathcal{F}_1 = \{f : 1, a : 0, b : 0\}$, $\mathcal{R}_1 = \{f(a) \to f(b)\}$, $\mathcal{F}_2 = \{a : 0, b : 0\}$ *et* $\mathcal{R}_2 = \{b \to a\}$. *Les systèmes* $\mathcal{R}_1$ *et* $\mathcal{R}_2$ *sont terminant et hiérarchiques, mais il existe une chaîne infinie pour la relation* $\Rrightarrow$: $f(a) \Rrightarrow_{\mathcal{R}_1} f(b) \Rrightarrow_{\mathcal{R}_2} f(a) \Rrightarrow_{\mathcal{R}_1} \dots$.

La relation de réduction modulaire n'étant pas nécessairement terminante dans le cas de la composition de systèmes hiérarchiques, nous proposons un critère suffisant pour la terminaison de la relation de réduction séquentielle. Nous montrons d'abord que la relation de réduction séquentielle termine pour les systèmes disjoints et à constructeurs partagés. Ceci est une conséquence de l'inclusion de la relation de réduction séquentielle dans la fermeture transitive de la relation de réduction modulaire, comme nous le montrons maintenant.

Lemme 4.1 *Soient* $(\mathcal{F}_1, \mathcal{R}_1), \dots, (\mathcal{F}_n, \mathcal{R}_n)$ *des systèmes de réécriture.*

$$\twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n} \subseteq \Rrightarrow^+$$

Preuve Soient $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ tels que $s \twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n} t$. Par définition de $\twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n}$, on sait que $\exists s_1, \dots, s_{n-1} \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ tels que $s \xrightarrow[R_1]{!} s_1$, $s_1 \xrightarrow[R_2]{!} s_2, \dots$, et $s_{n-1} \xrightarrow[R_n]{!} t$. Soit $s_0 = s$, $s_n = t$, et soit S l'ensemble d'indices i tels que $\mathcal{R}_i$ a été appliqué dans $s \twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n} t$, i.e. $S = \{i \mid 1 \leq i \leq n \text{ t.q. } s_{i-1} \to^+_{R_i} s_i\}$. En conséquence, on a $\forall i \in S$, $s_{i-1} \to^+_{R_i} s_i$ et s_i est en forme normale par rapport à $\mathcal{R}_i$. Ensuite, par définition de $\Rrightarrow$, on obtient que $\forall i \in S$ $s_{i-1} \Rrightarrow s_i$. De plus, $\forall i \in \{1 \dots n\} \setminus S$ on a $s_{i-1} = s_i$. Ainsi, $\forall i \in \{1 \dots n\}$ on a $s_{i-1} \Rrightarrow s_i$ ou $s_{i-1} = s_i$, d'où, on a finalement $s \Rrightarrow^* t$. L'ensemble S ne peut pas être vide puisque par définition de $\twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n}$, $s \twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n} t$ implique que s est réductible par au moins l'un des systèmes $\mathcal{R}_i$ avec $i \in \{1, \dots, n\}$. Donc, il y a au moins une étape de $\Rrightarrow$, et finalement on obtient $s \Rrightarrow^+ t$. $\square$

Théorème 4.9 *(Terminaison de la relation de réduction séquentielle) Si* $(\mathcal{F}_1, \mathcal{R}_1), \dots, (\mathcal{F}_n, \mathcal{R}_n)$ *sont des systèmes de réécriture disjoints deux à deux (ou à constructeurs partagés), alors la relation de réduction séquentielle est terminante sur* $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Preuve S'il existe une chaîne infinie $s_1 \twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n} s_2 \twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n} \dots$ alors, par le lemme précédent, il est possible de construire une chaîne infinie $s_1 \Rrightarrow^+ s_2 \Rrightarrow^+ \dots$ qui contredit la terminaison de $\Rrightarrow$. $\square$

Un critère possible de terminaison des combinaisons hiérarchiques (ou même générales) consiste à construire les ensembles contenant tous les termes résultant de toutes les dérivations possibles avec $\twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n}$ sur $\mathcal{T}(\mathcal{F})$, et à vérifier qu'au bout d'un nombre fini d'applications de $\twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n}$, tous les termes obtenus sont en forme normale.

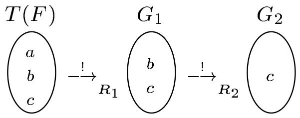
Tout d'abord, il faut s'assurer que si deux systèmes de réécriture $\mathcal{R}_1$ and $\mathcal{R}_2$ terminent sur $\mathcal{T}(\mathcal{F}_1)$ et $\mathcal{T}(\mathcal{F}_2)$ respectivement, alors $\mathcal{R}_1$ et $\mathcal{R}_2$ terminent sur $\mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2)$. Cette propriété, également valable pour la terminaison faible, est donné par la proposition suivante.

Proposition 4.7 *L'extension de la signature préserve la terminaison et la terminaison faible.*

Dans le cas de la terminaison, ce résultat est une conséquence de [Mid89], mais il en existe également une preuve dans [Gra94]. Pour le cas de la terminaison faible, c'est un corollaire direct de la modularité de la terminaison faible pour les systèmes disjoints [Mid90]).

Maintenant, nous donnons une idée du critère de terminaison de la relation de réduction séquentielle sur un exemple très simple, impliquant un ensemble fini de termes.

Exemple 4.18 *Soient* $\mathcal{R}_1 = \{a \to b\}$ *et* $\mathcal{R}_2 = \{b \to a, a \to c\}$*, deux systèmes de réécriture définis sur* $\mathcal{F}_1 = \{a, b\}$ *et sur* $\mathcal{F}_2 = \{a, b, c\}$*, respectivement. Soit* $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 = \mathcal{F}_2$*. Puisque* $\mathcal{F} = \mathcal{F}_2$*,* $\mathcal{R}_2$ *termine sur* $\mathcal{T}(\mathcal{F})$*. De la proposition 4.7, on tire que* $\mathcal{R}_1$ *termine sur* $\mathcal{T}(\mathcal{F})$*. La terminaison de* $\rightarrow_{\mathcal{R}_1;\mathcal{R}_2}$ *sur* $\mathcal{T}(\mathcal{F})$ *peut être montrée de la façon suivante. En réécrivant, chaque terme de* $\mathcal{T}(\mathcal{F})$ *avec* $\mathcal{R}_1$ *jusqu'à atteindre une forme normale, on obtient l'ensemble* $G_1 = \mathcal{R}_1^!(\mathcal{T}(\mathcal{F})) = IRR(\mathcal{R}_1)$*. Ensuite, si on réécrit chaque terme de* G_1 *avec* $\mathcal{R}_2$ *jusqu'à atteindre une forme normale, on obtient l'ensemble* $G_2 = \mathcal{R}_2^!(G_1)$*. Sur l'ensemble* G_2*, ni* $\mathcal{R}_1$ *ni* $\mathcal{R}_2$ *ne s'applique, d'où* $\rightarrow_{\mathcal{R}_1;\mathcal{R}_2}$ *termine sur* $\mathcal{T}(\mathcal{F})$*.*



Lorsque l'on considère des signatures contenant des symboles m-aires ($m > 0$) et donc des ensembles de termes potentiellement infinis, nous généralisons cette approche en utilisant des automates d'arbres. Dans la section 6.6 nous présentons un critère permettant de traiter le cas hiérarchique et le cas général pour des systèmes linéaires à gauche et faisant intervenir le calcul de l'ensemble des $\mathcal{R}$-formes normales d'un ensemble de terme représenté par un automate d'arbres.

Outre une forme de modularité de la terminaison, ces résultats permettent de montrer la terminaison de la réécriture sous les stratégies SRM et SRS vues dans la section 3.2.2. Dans la section suivante, nous voyons d'autres résultats existants pour la stratégie innermost.

4.7 Prise en compte des stratégies

Si un système de réécriture termine, c'est pour toute stratégie d'application des règles de réécriture. En revanche, certains systèmes ne terminent que pour certaines stratégies d'application des règles. La recherche de critères de terminaison des systèmes de réécriture sous stratégies est un domaine de recherche encore très récent. Il n'existe à notre connaissance que trois études portant sur la preuve de terminaison sous stratégies: des travaux de T. Arts et J. Giesl [AG97b], des travaux de N. Dershowitz et C. Hoot [Der87, DH95] dans le cas de la stratégie innermost et nos propres travaux [Gen97a, Gen98] dans le cas de la stratégie de réduction séquentielle.

Bien que la terminaison sous stratégie soit une propriété plus faible que la terminaison générale, ses critères de preuve sont plus complexes que dans le cas général. Alors que dans le cas particulier de la stratégie innermost et de la stratégie de réduction séquentielle, les méthodes citées ci-dessus permettent de prouver de façon automatique la terminaison d'un système de réécriture, pour une stratégie quelconque, il n'existe pas de critère général de terminaison [5] sous stratégie.

Dans [DH95], N. Dershowitz et C. Hoot ont montré que l'absence de chaîne infinie de réécriture innermost pour un système de réécriture donné était équivalente à l'absence de calcul de clôture innermost infini. Or, la finitude du calcul de clôture innermost n'est décidable que

5. Dans la section 8.1, nous proposons une première ébauche d'un tel critère.

dans des cas très restreints et donc son utilisation en tant que critère de terminaison innermost est difficilement envisageable. En revanche, la méthode de T. Arts et J. Giesl [AG97b], encore basée sur des paires de dépendance, donne de bons résultats en pratique pour des systèmes réputés difficiles (pour des exemples, voir notamment [Art97]). Nous ne détaillerons aucune de ces méthodes mais le lecteur intéressé pourra consulter [DH95] pour les calculs de clôture innermost et [AG97b, Art97] pour les paires de dépendance appliquées à la preuve de terminaison innermost.

4.8 Méthodes implantées

Alors que les procédures de complétion ou les systèmes de preuve par induction intègrent des procédures ad-hoc pour ordonner les termes ou les formules, il n'existe pas, à notre connaissance, de système indépendant, ayant dépassé le stade de prototype, dédié à la preuve de terminaison des systèmes de réécriture. D'autre part, beaucoup de méthodes implantées font appel à l'expertise de l'utilisateur pour la conception de l'ordre et n'automatisent, en fait, que la phase de vérification de la bonne orientation du système par rapport à cet ordre.

En fait, les recherches sur l'automatisation de la conception des ordres sont récentes. Elles se divisent entre les deux grandes familles d'ordres que sont les ordres polynomiaux et les ordres syntaxiques. Les travaux de J. Steinbach [Ste94] et de J. Giesl [Gie95a] portent sur l'automatisation des preuves de terminaison des systèmes de réécriture par recherche d'interprétations polynomiales. Les travaux de J. Giesl ont donné lieu à au prototype POLO [Gie95b][6], et ceux de J. Steinbach au système TETRES. En combinant la recherche d'interprétations polynomiales et les paires de dépendance [Art96], J. Giesl & T. Arts ont conçu l'une des méthodes d'automatisation des preuves de terminaison des systèmes de réécriture les plus puissantes connues à ce jour [AG97a, AG97b, AG98]. Récemment T. Arts a développé un prototype[7] implantant l'intégralité de la méthode des paires de dépendance décrite dans la section 4.3. Cependant, dans ce prototype, pour la résolution finale des contraintes d'ordre, T. Arts a préféré les RPSs et l'ordre *lpo* aux ordres polynomiaux. Ce prototype a permis de prouver de façon totalement automatique la terminaison nombreux systèmes difficiles.

Une autre voie pour l'automatisation de la conception des ordres porte sur les ordres "à précédence". La méthode proposée dans les travaux précurseurs de D. Detlef & R. Forgaard [FD85] a été implantée dans REVE [Les83] et subsiste dans Larch [GHG$^+$93]. Cette méthode a été totalement réimplantée de façon plus complète, par T. Arts dans le prototype cité précédemment.

Pour notre part, nous proposons dans cette thèse une méthode générale de conception d'ordre basé sur le GPO[8] (General Path Ordering) qui généralise à la fois les ordres polynomiaux et les ordres à précédence. Le procédé est basé sur la résolution de contraintes d'ordre *gpo* qui permet de synthétiser l'ordre de façon incrémentale et d'automatiser efficacement les phases simples et répétitives de la synthèse, afin que l'utilisateur puisse concentrer ses efforts sur les principales difficultés de la preuve, non automatisables. Un prototype, réalisant des preuves automatiques avec une instance particulière de *gpo*, le *lpo*, a été implanté en ECLiPSe [MS$^+$93], il a notamment permis de synthétiser automatiquement des ordres pour des spécifications algébriques conditionnelles de taille conséquentes. Dans le chapitre 7, nous donnons quelques exemples de preuves

6. Disponible à l'adresse `ftp://kirmes.inferenzsysteme.informatik.th-darmstadt.de/pub/termination/`

7. Disponible à l'adresse `http://www.ericsson.se:800/cslab/~thomas/deppairs/`

8. C. Hoot a également réalisé une implantation de l'ordre *gpo*, nommée GPOTC [Hoo97]. Cependant, cette implantation est avant tout un outil de vérification de la bonne orientation d'un système vis-à-vis d'une instance de *gpo* déjà construite et n'a pas pour but de la synthétiser automatiquement.

automatiques réalisées avec notre prototype et nous comparons avec les résultats obtenus par Larch et le prototype de T. Arts.

Chapitre 5

Preuve de terminaison par résolution de contraintes d'ordre

Nous avons vu que la méthode standard de preuve de terminaison d'un système de réécriture consiste à rechercher un ordre bien fondé, tel que l'application d'une règle du système sur un terme donné produise un terme strictement plus petit vis-à-vis de l'ordre. En pratique, la manipulation de ces ordres est encore semi-automatique. En effet, si la vérification de la stricte décroissance par rapport à un ordre donné est en général automatisable, le choix du bon ordre lui, nécessite une expertise humaine et est difficile à double titre. D'une part, parce qu'il existe un grand nombre d'ordres et il faut choisir l'ordre le mieux adapté au système dont on veut prouver la terminaison. D'autre part, les ordres utiles en preuve de terminaison sont en général indécidables et les ordres décidables sont NP-complets (comme dans le cas du *lpo* [KN85]).

L'approche choisie dans cette thèse, au lieu de voir la preuve de terminaison comme une vérification de la stricte décroissance d'un ensemble de règles pour un ordre donné a priori, propose au contraire de construire un ordre approprié en fonction d'un système donné. Par l'utilisation de contraintes d'ordre, nous restreignons l'espace de recherche de l'ordre en fonction de la preuve à réaliser, et nous retardons l'intervention de l'utilisateur en la focalisant sur les étapes cruciales du choix.

Dans la section 5.1, nous définissons les contraintes d'ordre *gpo*, les obligations de preuve qui sont une forme résolue intermédiaire des contraintes d'ordre *gpo* et nous donnons un premier algorithme de résolution des contraintes d'ordre *gpo*. Cependant, nous verrons que cet algorithme n'est pas utilisable en pratique car il provoque une explosion de la résolution. Dans la section 5.2, nous proposons une structure partagée pour les termes, les contraintes et les obligations de preuve et, section 5.3, nous donnons un algorithme de résolution des contraintes *gpo* qui limite l'explosion en s'appuyant sur cette structure partagée [GG97a, GG97b]. Cet algorithme est illustré sur un exemple dans la section 5.4. Ensuite, dans la section 5.5, nous étudions un mécanisme semi-automatique de résolution des obligations de preuve. Enfin, dans la section 5.6, l'algorithme de résolution des contraintes d'ordre *gpo* est étendu au cas de l'extension lexicographique et multi-ensemble de l'ordre.

5.1 Une première approche pour la construction d'ordres *gpo*

Nous présentons maintenant un premier algorithme de résolution de contraintes d'ordre basé sur le General Path Ordering (*gpo*), défini dans le chapitre précédent.

5.1.1 Les contraintes d'ordre *gpo*

Les contraintes d'ordre sont un outil déjà bien connu en démonstration automatique [Com90, NR92, Pla93, Nie93, JSA94]. Cependant, ces travaux traitent tous du problème de la *satisfaisabilité* des contraintes d'ordre. Soient s_i et t_i $(i = 1, \ldots, n)$ des termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$, un ordre $>$, et une contrainte $C = \bigwedge_{i=1}^{n} s_i > t_i$. Comon [Com90] et Nieuwenhuis [Nie93] ont donné des algorithmes pour décider s'il existe une substitution close σ telle que l'on ait $s_i\sigma >_{lpo} t_i\sigma$ pour tout $i = 1, \ldots, n$. Pour une contrainte C, Plaisted [Pla93], et Johann & Socher-Ambrosius [JSA94] ont donné des algorithmes polynomiaux pour décider s'il existe une substitution close et un ordre de simplification $\succ$ tels que l'on ait $s_i\sigma \succ t_i\sigma$ pour tout $i = 1, \ldots, n$. Les procédures de Plaisted [Pla93], et Johann & Socher-Ambrosius [JSA94] sont également des procédures de décision pour la terminaison simple des systèmes de réécriture clos.

Ici le problème que nous abordons est différent: étant donné un système de réécriture $(l_i \to r_i, i = 1, \ldots, n)$, nous voulons déduire une instance $\Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex})$ (vérifiant la définition 4.20 page 50) telle que l'on ait $l_i \succ_{gpo}^{\Phi} r_i$ pour tout $i = 1, \ldots, n$.

Dans la suite, les contraintes $s >_{gpo} t$ et $s \simeq_{gpo} t$ seront nommées *contraintes d'ordre gpo*.

Définition 5.1 *Une* solution *d'une contrainte d'ordre gpo $s >_{gpo} t$ (resp. $s \simeq_{gpo} t$) est une instance Φ, telle que $s \succ_{gpo}^{\Phi} t$ (resp. $s \approx_{gpo}^{\Phi} t$).*

La résolution d'une contrainte d'ordre *gpo* consiste à construire une solution Φ. Ainsi, la résolution des contraintes d'ordre *gpo* associées aux règles d'un système de réécriture $\mathcal{R}$ donne une instance de *gpo* prouvant la terminaison de $\mathcal{R}$.

La procédure de résolution de contraintes que nous proposons fonctionne en deux étapes. D'abord, une étape de résolution syntaxique (sans aucune hypothèse sur Θ ni sur $>_{lex}$) permet d'obtenir des contraintes sur Θ et $>_{lex}$ uniquement, nommées *obligations de preuves*. Ensuite par instanciations successives des θ_i de Θ, nous déduisons, quand cela est possible, les ordres $>_i$ associés.

Une méthode naturelle permettant d'obtenir les obligations de preuves ($\mathcal{O}$-preuves en abrégé), consiste à décomposer les contraintes *gpo* en des contraintes plus simples. Avant de donner les règles de déduction décrivant le processus de décomposition des contraintes *gpo*, nous définissons plus formellement les contraintes nécessaires à ce processus.

- Soit Γ_{gpo} l'ensemble des contraintes *gpo* défini par:
  - $\Lambda \in \Gamma_{gpo}$, où Λ est la contrainte *gpo* triviale toujours vraie,
  - $s > t \in \Gamma_{gpo}$, $s \sim t \in \Gamma_{gpo}$, où $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
  - $A \wedge B \in \Gamma_{gpo}$, où $A, B \in \Gamma_{gpo}$.

- Soit Γ_Θ l'ensemble des $\mathcal{O}$-preuves défini par:
  - $\top \in \Gamma_\Theta$, où $\top$ est la $\mathcal{O}$-preuve triviale
  - $\Theta(s) >_{lex} \Theta(t) \in \Gamma_\Theta$, $\Theta(s) \simeq_{lex} \Theta(t) \in \Gamma_\Theta$, où $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
  - $A \wedge B \in \Gamma_\Theta$, $A \vee B \in \Gamma_\Theta$ où $A, B \in \Gamma_\Theta$.

- Soit Γ_{mix} l'ensemble des contraintes hétérogènes défini par:
  - $\perp \in \Gamma_{mix}$, où $\perp$ est la contrainte hétérogène insatisfaisable,

- $(M \parallel N) \in \Gamma_{mix}$ où $M \in \Gamma_{gpo}$ et $N \in \Gamma_\Theta$,
- $A \vee B \in \Gamma_{mix}$, où $A, B \in \Gamma_{mix}$.

Définissons alors la *satisfaisabilité* d'une $\mathcal{O}$-preuve.

Définition 5.2 *Soit $\Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex})$, $P, A, B \in \Gamma_\Theta$ et $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. On dit que Φ satisfait P, noté $\Phi \models P$ si*

- *$P = \top$, ou*
- *$P = A \vee B$ et ($\Phi \models A$ ou $\Phi \models B$), ou*
- *$P = A \wedge B$ et ($\Phi \models A$ et $\Phi \models B$), ou*
- *$P = \Theta(s) >_{lex} \Theta(t)$ et $\mathcal{T}_{0,k}(s) \succ_{lex} \mathcal{T}_{0,k}(t)$, ou*
- *$P = \Theta(s) \simeq_{lex} \Theta(t)$ et $\mathcal{T}_{0,k}(s) \approx_{lex} \mathcal{T}_{0,k}(t)$.*

Une instance $\Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex})$ satisfait une contrainte hétérogène $A \vee B \in \Gamma_{mix}$ si Φ satisfait A ou B et Φ satisfait $(M \parallel N) \in \Gamma_{mix}$ où $M \in \Gamma_{gpo}$ et $N \in \Gamma_\Theta$, si Φ est une solution de M et $\Phi \models N$.

5.1.2 Résolution des contraintes d'ordre *gpo*: un premier algorithme

Nous présentons un algorithme de résolution de contraintes directement appliqué sur les contraintes hétérogènes. Il produit des contraintes sur Θ et $>_{lex}$ à partir des contraintes *gpo*. Le point de départ de la résolution est une contrainte hétérogène représentant une condition suffisante pour la terminaison du système de réécriture $\mathcal{R} = (l_i \to r_i, i = 1, \ldots, n)$, qui est:

$$(l_1 > r_1 \ \wedge \ l_2 > r_2 \ \wedge \ \ldots \ \wedge \ l_n > r_n \ \parallel \ \top)$$

Dans le cas où la résolution réussit, le processus s'achève sur la contrainte $(\Lambda \parallel N)$ où $N \in \Gamma_\Theta$. Si la résolution échoue, le processus s'achève sur $\perp$.

Soient $s, t, s_1, \ldots, s_n, t_1, \ldots, t_m \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $x \in \mathcal{X}$, $C \in \Gamma_\Theta$, $E \in \Gamma_{gpo}$ et $M \in \Gamma_{mix}$. Les symboles $\wedge$ and $\vee$ sont supposés associatifs et commutatifs. La règle **Décompose >** correspond aux cas 1. et 2. de la définition du *gpo*(définition 4.19), et la règle **Décompose ∼** correspond au dernier cas.

Décompose > où $s = f(s_1, \ldots, s_n)$ et $t = g(t_1, \ldots, t_m)$

$$\frac{(s > t \ \wedge \ E \parallel C)}{\bigvee_{i=1}^{n} (s_i > t \ \wedge \ E \parallel C) \ \vee \ \bigvee_{i=1}^{n} (s_i \sim t \ \wedge \ E \parallel C) \ \vee \ (\bigwedge_{i=1}^{n} s > t_i \ \wedge \ E \parallel C \ \wedge \ \Theta(s) >_{lex} \Theta(t))}$$

Décompose ∼ où $s = f(s_1, \ldots, s_n)$ et $t = g(t_1, \ldots, t_m)$

$$\frac{(s \sim t \ \wedge \ E \parallel C)}{(\bigwedge_{i=1}^{n} s > t_i \ \wedge \ \bigwedge_{i=1}^{n} t > s_i \ \wedge \ E \parallel C \ \wedge \ \Theta(s) \simeq_{lex} \Theta(t))}$$

Règles de simplification

1. $\dfrac{s>x}{\Lambda}$ si $x \in \mathcal{V}ar(s)$ et $s \neq x$.

2. $\dfrac{(s>t \wedge E \parallel C)}{\perp}$ si $\mathcal{V}ar(s) \not\supseteq \mathcal{V}ar(t)$.

3. $\dfrac{(s\sim t \wedge E \parallel C)}{\perp}$ si $\mathcal{V}ar(s) \neq \mathcal{V}ar(t)$.

4. $\dfrac{(x>t \wedge E \parallel C)}{\perp}$

5. $\dfrac{(s\sim s \wedge E \parallel C)}{(E \parallel C)}$

6. $\dfrac{(x\sim t \wedge E \parallel C)}{\perp}$ si $x \neq t$.

7. $\dfrac{(s>s \wedge E \parallel C)}{\perp}$

8. $\dfrac{\top \wedge C}{C}$

9. $\dfrac{\Lambda \wedge E}{E}$

10. $\dfrac{\perp \vee M}{M}$

Voici maintenant un exemple d'exécution de ce processus de décomposition de contraintes *gpo* sur un système simple: $ffx \to fgfx$. Dans cet exemple, nous utilisons les conventions suivantes:

– les contraintes *gpo* trivialement fausses, comme $s > s$, sont marquées par le symbole $\Box$ et sont supprimées à l'étape suivante,

– les contraintes *gpo* trivialement vraies, comme $s \sim s$ sont remplacées par Λ à l'étape suivante,

– chaque contrainte hétérogène est étiquetée par une lettre (x). A l'étape suivante, les nouvelles contraintes provenant de la décomposition de la contrainte hétérogène (x) sont étiquetée par une nouvelle lettre et regroupées sous l'étiquette (x).

Même sur cet exemple très simple d'un système avec une seule règle, des symboles d'arité 1 et une profondeur maximum de 3, on voit que la résolution d'une telle contrainte explose:

1	$(a)\ (ffx > fgfx \ \Vert\ \top)$
2	$(a)\begin{cases}(b)\ (fx > fgfx \ \Vert\ \top) \vee \\ (c)\ (fx \sim fgfx \ \Vert\ \top) \vee \\ (d)\ (ffx > gfx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx))\end{cases}$
3	$(b)\begin{cases}\Box\ (x > fgfx \ \Vert\ \top) \vee \\ \Box\ (x \sim fgfx \ \Vert\ \top) \vee \\ (e)\ (fx > gfx \ \Vert\ \Theta(fx) >_{lex} \Theta(fgfx)) \vee\end{cases}$ $(c)\ \{\ (f)\ (fx > gfx \ \wedge\ fgfx > x \ \Vert\ \Theta(fx) \simeq_{lex} \Theta(fgfx)) \vee$ $(d)\begin{cases}(g)\ (fx > gfx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx)) \vee \\ (h)\ (fx \sim gfx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx)) \vee \\ (i)\ (ffx > fx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(ffx) >_{lex} \Theta(gfx))\end{cases}$
4	$(e)\begin{cases}\Box\ (x > gfx \ \Vert\ \Theta(fx) \simeq_{lex} \Theta(fgfx)) \vee \\ \Box\ (x \sim gfx \ \Vert\ \Theta(fx) \simeq_{lex} \Theta(fgfx)) \vee \\ \Box\ (fx > fx \ \Vert\ \Theta(fx) \simeq_{lex} \Theta(fgfx) \ \wedge\ \Theta(fx) >_{lex} \Theta(gfx)) \vee\end{cases}$ $(f)\begin{cases}\Box\ (x > gfx \ \wedge\ fgfx > x \ \Vert\ \Theta(fx) >_{lex} \Theta(fgfx)) \vee \\ \Box\ (x \sim gfx \ \wedge\ fgfx > x \ \Vert\ \Theta(fx) >_{lex} \Theta(fgfx)) \vee \\ \Box\ (fx > fx \ \wedge\ fgfx > x \ \Vert\ \Theta(fx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(fx) >_{lex} \Theta(gfx)) \vee\end{cases}$ $(g)\begin{cases}\Box\ (x > gfx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx)) \vee \\ \Box\ (x \sim gfx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx)) \vee \\ \Box\ (fx > fx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(fx) >_{lex} \Theta(gfx)) \vee\end{cases}$ $(h)\ \{\ \Box\ (fx > fx \ \wedge\ gfx > fx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(fx) \simeq_{lex} \Theta(gfx)) \vee$ $(i)\begin{cases}\Box\ (fx > fx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(ffx) >_{lex} \Theta(gfx)) \vee \\ (j)\ (fx \sim fx \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(ffx) >_{lex} \Theta(gfx)) \vee \\ (k)\ (ffx > x \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(ffx) >_{lex} \Theta(gfx) \ \wedge \\ \qquad \Theta(ffx) >_{lex} \Theta(fx))\end{cases}$
5	$(j)\ \{\ (\Lambda \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(ffx) >_{lex} \Theta(gfx)) \vee$ $(k)\ \{\ (\Lambda \ \Vert\ \Theta(ffx) >_{lex} \Theta(fgfx) \ \wedge\ \Theta(ffx) >_{lex} \Theta(gfx) \ \wedge\ \Theta(ffx) >_{lex} \Theta(fx))$

Cette explosion de la résolution est due à un nombre important de duplications de types différents:

1. des duplications de termes. Par exemple, à l'étape 4 de la résolution, on trouve **13** occurrences différentes du terme fx (sans compter les sous-termes),

2. des duplications de contraintes *gpo*. Toujours à l'étape 4, on trouve **5** occurrences de la contrainte $fx > fx$,

3. des duplications d'$\mathcal{O}$-preuves. A l'étape 4, il y a **4** occurrences de la $\mathcal{O}$-preuve: $\Theta(ffx) >_{lex} \Theta(fgx)$.

En dupliquant les contraintes *gpo*, on duplique leur résolution. Une amélioration possible serait de conserver le résultat de la résolution d'une contrainte *gpo* et de remplacer chaque occurrence de cette même contrainte par son résultat. Cependant, cette amélioration ne toucherait que les duplications de contraintes *gpo*, et traiterait les conséquences des duplications au lieu d'en traiter les causes. Un autre point important est de remarquer que de nombreuses contraintes trivialement insatisfaisables par *gpo*, telles (*b*) $fx > fgx$ sont engendrées et nécessitent plusieurs inférences pour être totalement supprimées. Dans la section suivante, nous présentons un algorithme limitant les duplications de termes, de contraintes *gpo*, d'$\mathcal{O}$-preuves et évitant la génération de contraintes trivialement insatisfaisables, à l'aide d'une structure de données partagées.

5.2 Structure partagée pour les termes et contraintes

La méthode de résolution de contraintes d'ordre *gpo* que nous présentons maintenant est basée sur une structure de donnée particulière permettant de partager les termes, les contraintes ainsi que leurs solutions. Une partie de cette structure de donnée est inspirée des graphes SOUR de C. Lynch et P. Strogova [LS95], utilisés pour représenter les règles de réécriture dans une procédure de complétion.

Dans cette représentation, un terme est un graphe dont les nœuds sont étiquetés par des symboles et dont les arêtes représentent la relation de sous-terme. Cette représentation sous forme de "DAG" (Directed Acyclic Graph) permet à deux termes distincts de partager un sous-terme commun. Sur ces DAGs, nous avons défini des arêtes supplémentaires représentant les contraintes d'ordre étiquetées par leur obligation de preuve. Nous définissons maintenant une représentation des règles de réécriture sous forme de DAG que nous nommons *Ordering Constraint Solving Graphs* (graphes OCS en abrégé).

Définition 5.3 *Un graphe OCS est un graphe $G = (V, E)$ dans lequel V est un ensemble de sommets (ou nœuds) étiquetés par des symboles de $\mathcal{F}$ ou des variables de $\mathcal{X}$, et $E \subseteq V \times V$ est l'ensemble des arcs étiquetées par S, R, $>$ ou $\sim$ pour désigner respectivement des arcs de sous-terme, réécriture, inégalité et équivalence. Les arcs S, R, $>$ sont orientés. Les arcs étiquetés par $>$, et $\sim$ sont également étiquetés par une $\mathcal{O}$-preuve. Les arcs de sous-terme sont également étiquetés par un entier i désignant le* rang *du sous-terme. Pour tout nœud $\mathsf{F} \in V$, étiqueté par $f \in \mathcal{F}$ d'arité n, pour tout $i = 1 \ldots n$, il existe $\mathsf{G}_i \in V$ et un unique arc de sous-terme $(\mathsf{F}, \mathsf{G}_i) \in E$ de rang i.*

Les arcs de sous-terme et les arcs de réécriture des graphes OCS représentent, respectivement, la relation de sous-terme strict dans le terme et la relation de réécriture dans les règles. Les arcs étiquetés par des $\mathcal{O}$-preuves représentent les obligations de preuve sur $(\Theta, >_{lex})$ que l'on déduit des contraintes d'ordre *gpo* durant la première phase de l'algorithme de résolution.

Nous définissons maintenant la fonction $Term$ qui à chaque nœud F d'un graphe OCS associe un terme t, tel que F est le nœud sommet du graphe OCS représentant t.

Définition 5.4 *Soit $G = (V, E)$ un graphe OCS et $\mathsf{F} \in V$. La fonction $Term$ de V vers $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est définie inductivement comme suit:*

1. *si F est étiqueté par $x \in \mathcal{X}$, alors $Term(\mathsf{F}) = x$,*
2. *si F est étiqueté par $f \in \mathcal{F}$ où l'arité de f est n, alors $Term(\mathsf{F}) = f(Term(\mathsf{T}_1), \ldots, Term(\mathsf{T}_n))$ où, pour tout $i = 1 \ldots n$, $\mathsf{T}_i \in V$, et $(\mathsf{F}, \mathsf{T}_i) \in E$ est un arc de sous-terme de rang i.*

Définition 5.5 *Soit $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. La représentation OCS d'une règle de réécriture $l \to r$ est un graphe OCS $G = (V, E)$ tel que:*

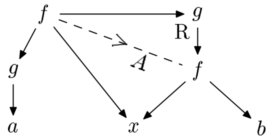
1. *$\forall \mathsf{F}, \mathsf{F}' \in V$ t.q. $\mathsf{F} \neq \mathsf{F}'$ on a $Term(\mathsf{F}) \neq Term(\mathsf{F}')$, et*

2. *il existe deux nœuds* $\mathsf{F}, \mathsf{G} \in V$ *et un unique arc de réécriture* $(\mathsf{F}, \mathsf{G}) \in E$ *tel que* $Term(\mathsf{F}) = l$ *et* $Term(\mathsf{G}) = r$.

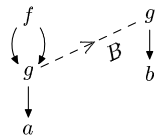
Dans la définition précédente, on peut remarquer que le cas 1. assure le partage des sous-termes dans les représentations OCS des règles de réécriture.

Dans la suite, pour tout graphe OCS $G = (V, E)$ avec $\mathsf{F}, \mathsf{G} \in V$, on notera $\mathsf{F} \mathrel{-\!\!>\!\!-} \mathsf{G}$ (resp. $\mathsf{F} \mathrel{-\!\!\sim\!\!-} \mathsf{G}$) un arc d'inégalité (resp. d'équivalence) $(\mathsf{F}, \mathsf{G}) \in E$. On notera $\mathsf{F} \mathrel{-\!\!\not>\!\!-} \mathsf{G}$ (resp. $\mathsf{F} \mathrel{-\!\!\not\sim\!\!-} \mathsf{G}$) s'il n'existe aucun arc d'inégalité (resp. d'équivalence) $(\mathsf{F}, \mathsf{G}) \in E$. Si $Term(\mathsf{F}) = s$ et $Term(\mathsf{G}) = t$, alors $\mathsf{F} \mathrel{-\!\!>\!\!-} \mathsf{G}$ (resp. $\mathsf{F} \mathrel{-\!\!\sim\!\!-} \mathsf{G}$) sera aussi noté $s \mathrel{-\!\!>\!\!-} t$ (resp. $s \mathrel{-\!\!\sim\!\!-} t$). Les arcs d'inégalité et d'équivalence seront nommés *arcs d'ordre*. Dans les figures suivantes, les flèches en traits pleins représentent les arcs de sous-terme, les flèches en traits pleins étiquetées par R représentent les arcs de réécriture et les lignes en pointillé représentent les arcs d'inégalité et d'équivalence. Les étiquettes de rang sont omises mais peuvent être facilement déduites des figures où les arcs de sous-terme seront toujours ordonnés par rang croissant de gauche à droite.

Exemple 5.1 *La représentation OCS de la règle de réécriture* $f(g(a), x) \to g(f(x, b))$ *avec un arc d'inégalité étiqueté par une* $\mathcal{O}$*-preuve* A *est présentée dans le graphe 5.1.1. Le graphe OCS 5.1.2 montre comment la contrainte* $g(a) > g(b)$, *est partagée et représentée par un arc unique étiqueté par une* $\mathcal{O}$*-preuve* B.



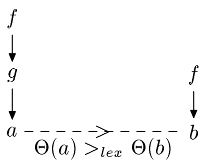
Graphe 5.1.1



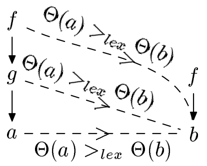
Graphe 5.1.2

Un graphe OCS permet de partager les termes et les contraintes, mais il peut contenir des duplications de $\mathcal{O}$-preuves. Nous donnons maintenant un exemple dans lequel les $\mathcal{O}$-preuves sont dupliquées. Ensuite, nous verrons comment éviter ces duplications.

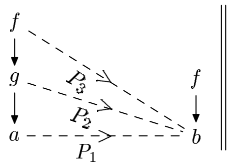
Exemple 5.2 *Soient* $f(g(a))$ *et* $f(b)$ *des termes, où* a *et* b *sont des constantes. Par la définition 5.2 page 65 et la définition du gpo,* $a \succ^{\Phi}_{gpo} b$ *si* $\Phi \models \Theta(a) >_{lex} \Theta(b)$. *On peut représenter ceci par le graphe OCS 5.2.1. Grâce à la propriété de sous-terme, on sait que* $g(a) \succ^{\Phi}_{gpo} b$ *si* $\Phi \models \Theta(a) >_{lex} \Theta(b)$, *puisque* $a \succ^{\Phi}_{gpo} b$ *implique* $g(a) \succ^{\Phi}_{gpo} b$. *De la même façon,* $g(a) \succ^{\Phi}_{gpo} b$ *implique* $f(g(a)) \succ^{\Phi}_{gpo} b$. *On obtient donc le graphe OCS 5.2.2. L'information que* $a \succ^{\Phi}_{gpo} b$ implique $g(a) \succ^{\Phi}_{gpo} b$ *n'apparaît pas dans le graphe 5.2.2. Ainsi, si l'on souhaite prouver qu'il existe un* Φ *tel que* $a \succ^{\Phi}_{gpo} b$ *et* $g(a) \succ^{\Phi}_{gpo} b$, *on devra prouver* deux fois *qu'il existe un* Φ *tel que* $\Phi \models \Theta(a) >_{lex} \Theta(b)$. *Afin de représenter l'implication entre les* $\mathcal{O}$*-preuves dans les graphes OCS, nous raffinons la représentation OCS en étiquetant les arcs d'ordre par des variables distinctes (à la place des formules) et nous factorisons les formules dans une substitution associant chaque variable à une formule. Ainsi, le graphe OCS 5.2.2 est transformé en un graphe 5.2.3, où la substitution est définie à droite.*



Graphe 5.2.1



Graphe 5.2.2



$P_1 \mapsto \Theta(a) >_{lex} \Theta(b)$
$P_2 \mapsto P_1$
$P_3 \mapsto P_2$

Graphe 5.2.3

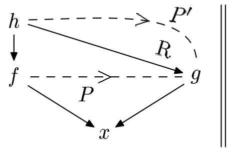
Définissons alors la structure utilisée dans le graphe 5.2.3 de façon plus formelle.

Définition 5.6 *Soit $\mathcal{X}_\Theta$ un ensemble de variables de $\mathcal{O}$-preuve. Une* Θ-substitution *σ est une application de $\mathcal{X}_\Theta$ dans Γ_Θ, qui peut être étendue de façon unique en un homomorphisme $\sigma : \Gamma_\Theta \mapsto \Gamma_\Theta$.*

La structure que nous allons utiliser pour résoudre les contraintes d'ordre *gpo* est composée d'un graphe OCS représentant une règle de réécriture et d'une Θ-substitution. Les arcs d'ordre sont étiquetés soit par la $\mathcal{O}$-preuve triviale $\top$, soit par une variable de $\mathcal{O}$-preuve. En appliquant la substitution à la variable étiquetant un arc d'inégalité (resp. d'équivalence), on obtient une $\mathcal{O}$-preuve de l'inégalité (resp. de l'équivalence) correspondante.

Définition 5.7 *Une* structure pour la résolution de contrainte d'ordre *(SOCS en abrégé) d'une règle de réécriture $l \to r$ est un couple $(G||\sigma)$ où G est un graphe OCS représentant la règle et σ une Θ-substitution.*

Exemple 5.3 *Voici un exemple de SOCS pour la règle $h(f(x)) \to g(x)$:*



$P \mapsto \Theta(f(x)) >_{lex} \Theta(g(x))$
$P' \mapsto P$

Dans ce SOCS, l'arc d'inégalité entre les nœuds étiquetés par f et g signifie qu'il existe au moins *une $\mathcal{O}$-preuve possible P pour $f(x) > g(x)$. A droite du SOCS, on trouve la Θ-substitution associant la variable P à la $\mathcal{O}$-preuve correspondante. L'application $P' \mapsto P$ signifie que la $\mathcal{O}$-preuve P est également une $\mathcal{O}$-preuve pour l'arc $h(f(x)) \longrightarrow\!\!\!\succ g(x)$.*

5.3 Résolution des contraintes d'ordre *gpo*: les règles de $\mathcal{C}$-déduction

Nous définissons maintenant les règles de déduction que nous utilisons, afin de déduire les $\mathcal{O}$-preuves des contraintes *gpo*. A la différence du processus descendant utilisé dans les règles de décomposition (section 5.1.2), le mécanisme de $\mathcal{C}$-déduction est ascendant et plus proche de la définition du *gpo* lui-même. En outre, il évite de synthétiser des arcs représentant des contraintes trivialement insatisfaisables, engendrées par l'algorithme descendant proposé dans la section précédente. Tout d'abord, nous introduisons la notion de *visibilité* qui traduit une notion de sous-formule dans le contexte des $\mathcal{O}$-preuves.

Définition 5.8 *Soient $P, Q \in \Gamma_\Theta$ des $\mathcal{O}$-preuves. On dit que P est* visible *dans Q, noté $P \triangleleft Q$ si ($P = Q$) ou [$Q = A \vee B$ et ($P \triangleleft A$ ou $P \triangleleft B$)].*

Pour résoudre les contraintes *gpo* sur un ensemble de règles, nous construisons d'abord un ensemble de *SOCS initiaux*, un pour chaque règle. Les SOCS initiaux sont des SOCS dont le graphe OCS ne contient aucun arc d'ordre et dont la Θ-substitution est vide. La résolution des contraintes d'ordre *gpo* sur les SOCS est effectuée par un ensemble de règles de déduction. Ces règles transforment un SOCS par adjonction d'arcs d'ordre au graphe et par construction de la Θ-substitution associant les $\mathcal{O}$-preuves aux arcs. La résolution est effectuée séparément pour chaque SOCS correspondant à une règle de réécriture. La résolution s'achève lorsque plus aucune règle de déduction ne s'applique. Nous noterons $\mathcal{C}$ l'ensemble des règles de déduction, $\vdash_\mathcal{C}$ la relation de déduction sur les SOCS engendrée par $\mathcal{C}$, et $\vdash^*_\mathcal{C}$ la fermeture réflexive transitive de

$\vdash_{\mathcal{C}}$. Un SOCS est en $\mathcal{C}$-forme normale lorsque plus aucune règle de $\mathcal{C}$ ne peut lui être appliquée. L'ensemble $\mathcal{C}$ est donné figure 5.1, où un arc $\xrightarrow{\gtrsim P}$ signifie qu'il existe un arc $\xrightarrow{> P}$ ou un arc $\xrightarrow{\sim P}$.

Définition 5.9 *Soit* $(\alpha \parallel \nu), (\beta \parallel \delta)$ *des SOCS. On dit qu'une règle* $\frac{\alpha \parallel \nu}{\beta \parallel \delta}$ *de* $\mathcal{C}$ filtre *un SOCS* $(G \parallel \sigma)$ *si* α *est un motif de* G *(c-à-d si* $\alpha = (V_\alpha, E_\alpha)$ *et* $G = (V_G, E_G)$*, il existe une bijection de* V_α *dans* V'_G *et une bijection de* E_α *dans* E'_G*, où* $V'_G \subseteq V_G$ *et* $E'_G \subseteq E_G$*), si* ν *filtre* σ*, et si la pré-condition de* $\frac{\alpha \parallel \nu}{\beta \parallel \delta}$ *est vérifiée.*

L'*application* de la règle consiste à remplacer le motif α de G par le motif β (parfois identiques) et à remplacer ν par δ dans σ. Chaque fois qu'un nouvel arc d'ordre est construit dans G, il est étiqueté par une nouvelle variable de $\mathcal{O}$-preuve. Il est nécessaire que les nœuds F et G des règles de $\mathcal{C}$ filtrent toujours des nœuds distincts dans G. Ceci empêche d'ajouter des arcs d'inégalité cycliques (toujours faux, par définition d'un ordre) et des arcs d'égalité cycliques (toujours inutiles).

Aucune stratégie spécifique n'est nécessaire pour l'application les règles de $\mathcal{C}$, et ce pour le choix du couple de nœuds, comme pour le choix de la règle à appliquer. Si on ajoute que grâce à la structure spécifique des SOCS, les déductions sur deux couples de nœuds distincts sont indépendantes, il est clair que le processus est directement parallélisable et ce, sans aucun besoin de synchronisation.

Dans le cas d'une implantation séquentielle, bien qu'aucune stratégie ne soit nécessaire, certaines stratégies adéquates donnent un algorithme plus efficace. Par exemple, dans notre implantation des règles de $\mathcal{C}$-déduction (voir chapitre 7), nous commençons par saturer le SOCS avec les arcs triviaux (étiquetés par une $\mathcal{O}$-preuve triviale) grâce aux règles **SUBTERM Property** and **SUBTERM Trivial**, avant d'appliquer les autres règles (exceptée la règle **SUBTERM Simplification** qui devient inutile puisque le SOCS est déjà saturé avec les arcs triviaux). Nous donnons maintenant trois théorèmes exprimant: la correction, la complétude et la terminaison de $\mathcal{C}$.

Définition 5.10 *Étant donné un SOCS* $(G \parallel \sigma)$*,* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*, un arc* $s \xrightarrow{> P} t$ *de* G *(resp.* $s \xrightarrow{\sim P} t$*) est* correct *vis-à-vis de gpo si pour tout* $\Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex})$ *tel que* $\Phi \models P$*, on a* $s \succ^{\Phi}_{gpo} t$ *(resp.* $s \approx^{\Phi}_{gpo} t$*). Un SOCS* $(G \parallel \sigma)$ *est* correct *vis-à-vis de gpo si tout arc de* G *est correct vis-à-vis de gpo.*

Théorème 5.1 *(Correction) Pour tout SOCS initial* S*, si* $S \vdash^*_{\mathcal{C}} S'$ *alors* S' *est correct vis-à-vis de gpo.*

Théorème 5.2 *(Complétude) Soit* $(G \parallel \sigma)$ *un SOCS en* $\mathcal{C}$*-forme normale, tel que* $G = (V, E)$*. Pour tous nœuds* $\mathsf{F}, \mathsf{G} \in V$*, t.q.* $Term(\mathsf{F}) = s$ *et* $Term(\mathsf{G}) = t$*, où* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*:*

$$\forall \Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex}) \text{ t.q. } s \succ^{\Phi}_{gpo} t, \text{ il existe un arc } s \xrightarrow{> P} t \text{ dans } G \text{ t.q. } \Phi \models P\sigma$$

$$\forall \Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex}) \text{ t.q. } s \approx^{\Phi}_{gpo} t, \text{ il existe un arc } s \xrightarrow{\sim P} t \text{ dans } G \text{ t.q. } \Phi \models P\sigma.$$

Théorème 5.3 *(Complexité) Soit* $l \to r$ *une règle de réécriture,* $(G \parallel \sigma)$ *le SOCS initial de* $l \to r$*,* N *le nombre de nœuds de* G*, et* M *l'arité maximale des symboles de fonction (on suppose* $M > 0$*) de la règle. La complexité en temps et en espace du processus de* $\mathcal{C}$*-déduction initié sur* $(G||\sigma)$ *est polynomiale en* N *et* M *dans le pire des cas* $(O(N^4M^2))$*.*

Pour des preuves détaillées, voir les sections 5.8.2, 5.8.3, 5.8.4.

Nous avons vu que chaque règle est traitée de façon indépendante. Pour résoudre les contraintes sur un ensemble de règles, nous allons maintenant voir comment rassembler les résultats relatifs à chaque règle.

Définition 5.11 *Soit $\mathcal{R}$ un système de réécriture ($l_i \rightarrow r_i$, $i = 1 \dots n$) et ($G_i \parallel \sigma_i$) les SOCS en $\mathcal{C}$-forme normale représentant les règles $l_i \rightarrow r_i$. Soit P_i la $\mathcal{O}$-preuve étiquetant l'arc $l_i \twoheadrightarrow r_i$ dans G_i pour $i = 1 \dots n$. La $\mathcal{O}$-*preuve globale *de $\mathcal{R}$ est la $\mathcal{O}$-preuve: $P_1\sigma_1 \,\wedge\, \dots \,\wedge\, P_n\sigma_n$.*

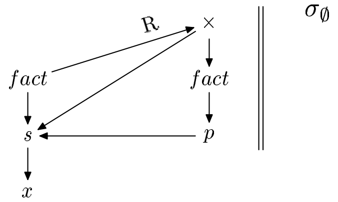
S'il existe une règle $l_i \rightarrow r_i$ dont le graphe OCS G_i ne comporte aucun arc $l_i \overset{P_i}{\twoheadrightarrow} r_i$ alors, il est impossible de prouver la terminaison de $\mathcal{R}$ avec *gpo*. La définition 5.11 page 72 montre, en outre, que les $\mathcal{O}$-preuves permettent de traiter le cas des systèmes de réécriture incrémentaux, dont le nombre de règles peut augmenter. Cette caractéristique est utile notamment dans les procédures de complétion. Il faut également remarquer que le processus construit des arcs d'inégalité et des arcs d'équivalence pour les deux orientations possible des règles de réécriture (de gauche à droite ou de droite à gauche). L'orientation finale du système de réécriture est choisie lors de la construction de son $\mathcal{O}$-preuve globale. On peut remarquer qu'il est également possible, comme dans [LS95] pour les ensembles d'équations, de considérer un seul SOCS pour l'ensemble du système de réécriture. Avec cette approche, les résultats de correction, de complétude et de terminaison sont aussi valides. De plus, pour certains systèmes, cette approche favorise le partage des $\mathcal{O}$-preuves et donc factorise le travail de preuve de satisfaisabilité des $\mathcal{O}$-preuves. Cependant, même si en théorie cette approche globale paraît plus convaincante, les résultats obtenus en pratique montrent que l'approche locale est aussi bonne tout en étant moins coûteuse en temps machine.

5.4 Un exemple de $\mathcal{C}$-déduction

Soit le système de réécriture suivant, emprunté à [DH95], qui calcule la factorielle d'un entier.

$p(s(x)) \rightarrow x$	(1)	$s(x) \times y \rightarrow (x \times y) + y$	(5)
$fact(0) \rightarrow s(0)$	(2)	$x + 0 \rightarrow x$	(6)
$fact(s(x)) \rightarrow s(x) \times fact(p(s(x)))$	(3)	$x + s(y) \rightarrow s(x + y)$	(7)
$0 \times y \rightarrow 0$	(4)		

On peut remarquer que la terminaison de $\mathcal{R}$ ne peut pas être prouvée avec un ordre de simplification puisque dans la règle (3) le membre gauche est plongé (voir définition 2.3 page 7) dans le membre droit, i.e. $fact(s(x))$ peut être obtenu à partir de $s(x) \times fact(p(s(x)))$ si on supprime les symboles $\times$ et p. Cependant, il est possible de prouver la terminaison de $\mathcal{R}$ avec *gpo*. Soit $\sigma_\emptyset$ la Θ-substitution vide. Notre procédure de $\mathcal{C}$-déduction débute sur le SOCS initial correspondant à la règle (3) de $\mathcal{R}$, comme défini dans la section 5.3:



Voici maintenant quelques-unes des étapes du processus de résolution. Pour une meilleure lisibilité, nous avons préféré cacher les arcs de réécriture, ne représenter que les arcs d'ordre utiles à une étape donnée et étiqueter les arcs concernés par des entiers.

1. Au départ, puisqu'il n'existe aucun arc d'ordre, la seule règle que nous pouvons appliquer

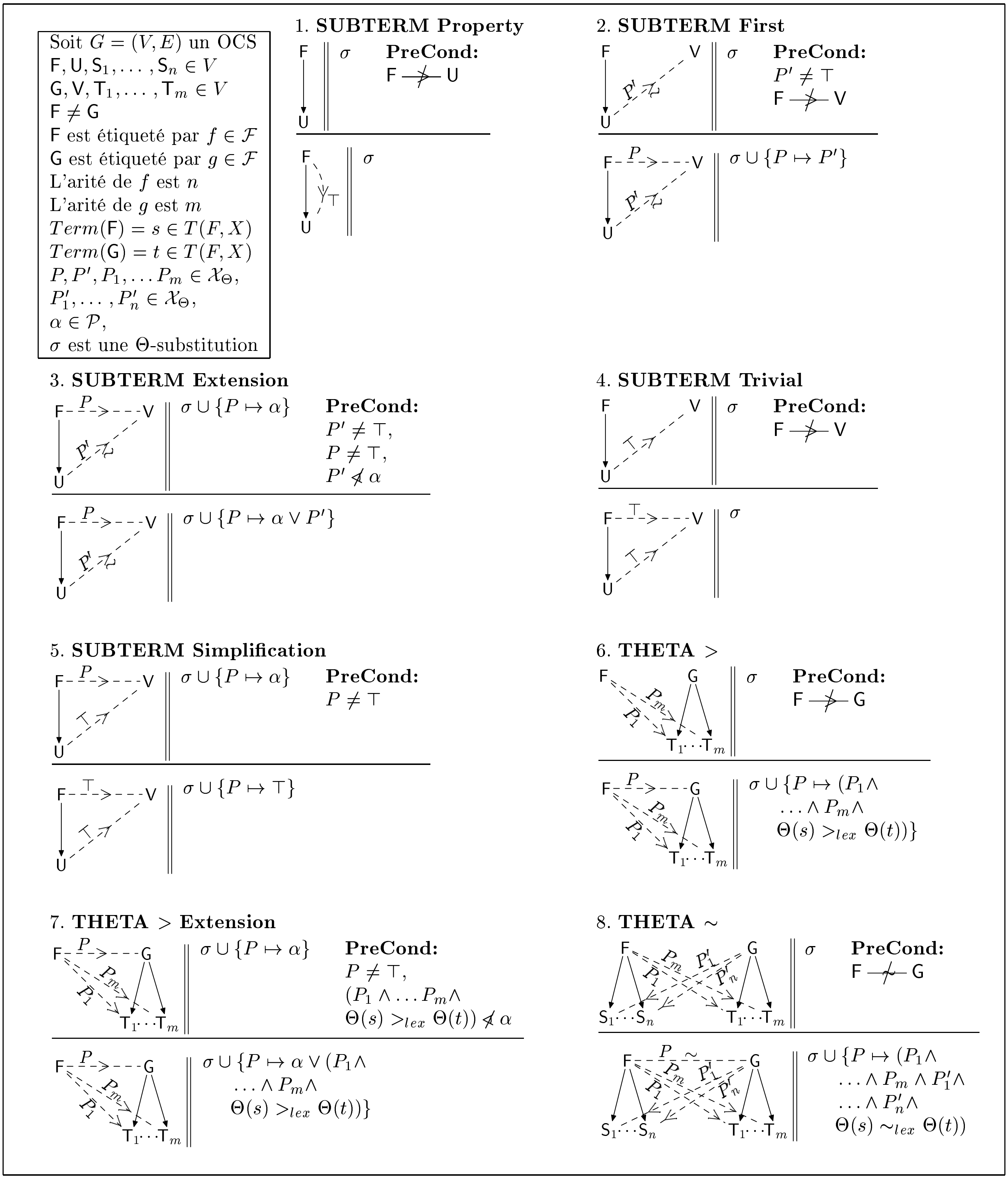
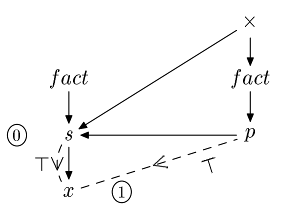


FIG. 5.1 – *Les règles de $\mathcal{C}$-déduction*

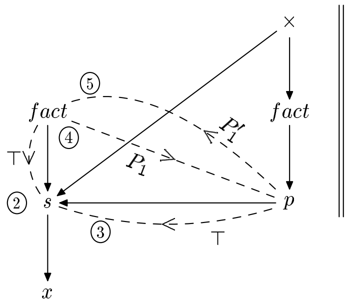
est **SUBTERM Property**. Grâce à l'application de cette règle, nous obtenons des arcs d'inégalité triviaux pour les sous-termes directs, tel 0.



$\sigma_\emptyset$

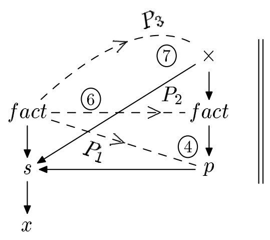
Ensuite, grâce à l'arc 0, nous pouvons alors appliquer la règle **SUBTERM Trivial** (Les nœuds F, U etV de la règle **SUBTERM Trivial** filtrent respectivement les nœuds étiquetés par les symboles p, s et x) et l'arc 1 est ajouté.

2. Plus tard, en supposant que l'arc 2 a été construit lors de déductions précédentes, nous pouvons appliquer la règle **THETA** $>$. Celle-ci ajoute l'arc 4 et introduit la Θ-substitution associée dans la partie droite du SOCS (comme la $\mathcal{O}$-preuvede l'arc 2 est $\top$, elle n'apparaît pas dans la Θ-substitution de P_1). De façon symétrique, grâce à l'arc 3 et grâce à une autre application de la règle **THETA** $>$, nous pouvons déduire l'arc 5 et introduire la Θ-substitution associée dans le SOCS. (grâce aux arcs 2, 3 et à la règle **THETA** $\sim$, nous pourrions également ajouter un arc d'équivalence).



$P_1 \mapsto \Theta(fact(s(x))) >_{lex} \Theta(p(s(x)))$
$P_1' \mapsto \Theta(p(s(x))) >_{lex} \Theta(fact(s(x)))$

3. Toujours grâce à la règle **THETA** $>$, nous pouvons déduire l'arc 6, grâce à l'existence de l'arc 4. De la même façon, l'arc 7 est obtenu à partir de l'arc 6 par la règle **THETA** $>$.



$P_1 \mapsto \Theta(fact(s(x))) >_{lex} \Theta(p(s(x)))$
$P_2 \mapsto P_1 \wedge \Theta(fact(s(x))) >_{lex} \Theta(fact(p(s(x))))$
$P_3 \mapsto P_2 \wedge \Theta(fact(s(x))) >_{lex} \Theta(s(x) \times fact(p(s(x))))$

Dans la suite, nous noterons $P_{(1)}, \ldots, P_{(7)}$ les $\mathcal{O}$-preuves des règles (1) à (7) respectivement. Nous noterons σ la Θ-substitution du SOCS final pour la règle (3). Dans le SOCS de la règle (3) en $\mathcal{C}$-forme normale, on a $fact(s(x)) \xrightarrow{P_3} s(x) \times fact(p(s(x)))$. La $\mathcal{O}$-preuve $P_{(3)}$ pour la règle (3) est donc $P_3\sigma$. Comme les $\mathcal{O}$-preuves pour les règles (1), (4) et (6) sont des $\mathcal{O}$-preuves triviales, la $\mathcal{O}$-preuve globale pour le système de réécriture complet est $P_{(2)} \wedge P_{(3)} \wedge P_{(5)} \wedge P_{(7)}$.

5.5 Preuve de satisfaisabilité des $\mathcal{O}$-preuves

A cette étape de la résolution, nous avons un ensemble de SOCS saturés (un pour chaque règle de réécriture), dont les $\mathcal{O}$-preuves contiennent des fonctions de terminaison non instanciées: aucune hypothèse n'est faite sur Θ ni sur $\geq_{lex}$. L'étape suivante consiste donc à prouver la satisfaisabilité des $\mathcal{O}$-preuves en déduisant des valeurs particulières pour $\Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex})$ de $(\Theta, \geq_{lex})$ afin de satisfaire la contrainte initiale. Nous proposons, ici, de réaliser cette dernière étape grâce à un processus d'*instanciation partielle* que nous définissons maintenant.

Dans la suite, ce que nous appelons des *littéraux instanciés* d'une $\mathcal{O}$-preuve sont de la forme $\tau_i(s) \succ_i \tau_i(t)$ ou $\tau_i(s) \approx_i \tau_i(t)$, où τ_i sont des fonctions de terminaison et $\succ_i$, $\approx_i$, les ordres et équivalences associées.

Définition 5.12 *Soient i et j deux entiers tels que $0 \leq i < j$. Une $\Theta_{i,j}$ $\mathcal{O}$-preuve est une $\mathcal{O}$-preuve dont chaque littéral non-instancié est soit de la forme $\Theta_{i,j}(s) >_{lex} \Theta_{i,j}(t)$ soit de la forme $\Theta_{i,j}(s) \simeq_{lex} \Theta_{i,j}(t)$ où s, t sont des termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

Un littéral non-instancié comporte une suite de fonctions de terminaison abstraites $\theta_i, \ldots, \theta_j$ telle que $0 \leq i < j$. L'instanciation partielle d'un littéral non-instancié remplace une fonction de terminaison abstraite de la suite par une instance particulière. Il est clair que le choix de la fonction à instancier est totalement arbitraire; nous proposons ici deux stratégies possibles – *l'instanciation partielle gauche* et *l'instanciation partielle droite* – privilégiant l'instanciation de la fonction de terminaison la plus à gauche ou la plus à droite.

Définition 5.13 *Soit $0 \leq i < j$ et P une $\Theta_{i,j}$ $\mathcal{O}$-preuve, une instanciation partielle gauche (IPG en abrégé) de P consiste en l'instanciation de chaque θ_i de P par une fonction de terminaison particulière τ_i.*

De la même façon, on peut définir une instanciation partielle droite basée sur le même principe.

Définition 5.14 *Soit $0 \leq i < j$ et P une $\Theta_{i,j}$ $\mathcal{O}$-preuve, une instanciation partielle droite (IPD en abrégé) de P consiste en l'instanciation de chaque θ_j de P par une fonction de terminaison particulière τ_j.*

Si l'on considère une $\mathcal{O}$-preuve $\Theta_{i,j}(s) >_{lex} \Theta_{i,j}(t)$, son IPG donne

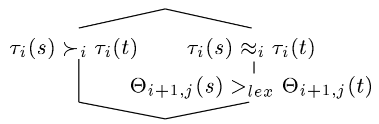
$$\tau_i(s) \succ_i \tau_i(t) \vee [\tau_i(s) \approx_i \tau_i(t) \wedge \Theta_{i+1,j}(s) >_{lex} \Theta_{i+1,j}(t)].$$

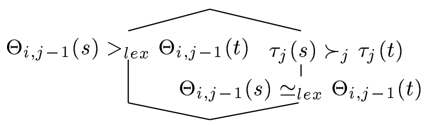
Si l'on considère une $\mathcal{O}$-preuve $\Theta_{i,j}(s) \simeq_{lex} \Theta_{i,j}(t)$, son IPG donne

$$\tau_i(s) \approx_i \tau_i(t) \wedge \Theta_{i+1,j}(s) \simeq_{lex} \Theta_{i+1,j}(t).$$

En pratique, la méthode que nous utilisons pour déduire une solution d'une $\mathcal{O}$-preuve globale grâce à l'IPG est basée sur une représentation des $\mathcal{O}$-preuves sous forme de DAGs. Les DAGs d'$\mathcal{O}$-preuve sont des DAGs et-ou représentant les $\mathcal{O}$-preuves: la $\mathcal{O}$-preuve conjonctive $\alpha \wedge \beta$ est représentée par le DAG $\begin{array}{c}A\\|\\B\end{array}$ et la $\mathcal{O}$-preuve disjonctive $\alpha \vee \beta$ est représentée par le DAG $A\ B$, où A et B sont des DAGs représentant α et β, respectivement. Ainsi les IPG et IPD peuvent être vues comme de simples procédures de transformation de graphes:

– chaque nœud de la forme $\Theta_{i,j}(s) >_{lex} \Theta_{i,j}(t)$ peut être remplacé par:

(IPG) 

(IPD) 

– chaque nœud $\Theta_{i,j}(s) \simeq_{lex} \Theta_{i,j}(t)$ peut être remplacé par:

$$\text{(IPG)} \quad \begin{array}{c} \tau_i(s) \approx_i \tau_i(t) \\ | \\ \Theta_{i+1,j}(s) \simeq_{lex} \Theta_{i+1,j}(t) \end{array} \qquad\qquad \text{(IPD)} \quad \begin{array}{c} \tau_j(s) \approx_j \tau_j(t) \\ | \\ \Theta_{i,j-1}(s) \simeq_{lex} \Theta_{i,j-1}(t) \end{array}$$

Informellement, dans ces graphes, une solution est un *chemin* du haut du graphe jusqu'en bas dont tous les nœuds sont instanciés et forment une conjonction de contraintes d'ordre *compatible*, i.e. satisfaisable. Maintenant, nous définissons plus formellement ces deux notions.

Définition 5.15 *Étant donné $0 \leq i < j$ et G un DAG de $\Theta_{i,j}$ $\mathcal{O}$-preuve, un* i-chemin *de G est un couple (p, A) où p est un chemin de la racine aux feuilles de G, et A est un t-uple d'ensembles $\langle A_0, \ldots, A_{i-1}, A_i\rangle$, où A_u $(0 \leq u \leq i-1)$ est l'ensemble $\{\alpha | \alpha \in p, \alpha = \tau_u(s) >_u \tau_u(t)$ ou $\alpha = \tau_u(s) \simeq_u \tau_u(t), s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})\}$ et $A_i = \{\alpha | \alpha \in p, \alpha = \Theta_{i,j}(s) >_{lex} \Theta_{i,j}(t)$ ou $\alpha = \Theta_{i,j}(s) \simeq_{lex} \Theta_{i,j}(t), s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})\}$.*

Définition 5.16 *Soit S un ensemble fini. Un ensemble d'inégalités et d'égalités $A = \{\alpha \succ \beta | \alpha, \beta \in S\} \cup \{\alpha \approx \beta | \alpha, \beta \in S\}$ est* compatible *si et seulement s'il existe un préordre $\succsim_S$ sur S tel que $\alpha \succ \beta \in A \implies \alpha \succ_S \beta$ et $\alpha \approx \beta \in A \implies \alpha \approx_S \beta$ (où $\succsim_S$ représente $\succ_S \cup \approx_S$).*

Parmi tous les i-chemins, certains sont à étudier en priorité: les *i-chemin minimaux*, qui minimisent l'ensemble des contraintes sur les fonctions de terminaison non instanciées $\Theta_{i,k}$ et les ordres associés $>_{lex}$.

Définition 5.17 *Soit (p, A) un i-chemin, où $A = \langle A_0, \ldots, A_{i-1}, A_i\rangle$. Le i-chemin (p, A) est minimal si $A_0, \ldots, A_{i-1}$ sont des ensembles compatibles, et s'il n'existe aucun i-chemin (p', B), tel que $B = \langle B_0, \ldots, B_{i-1}, B_i\rangle$, où $B_0, \ldots, B_{i-1}$ sont des ensembles compatibles et $B_i \subset A_i$.*

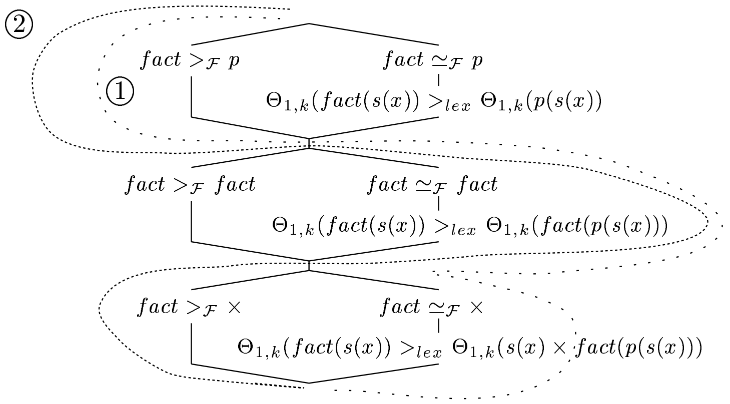
En général, un i-chemin minimal n'est pas unique.

Définition 5.18 *Un* i-chemin satisfaisable *est un i-chemin minimal tel que $A_i = \emptyset$.*

Dans un DAG d'$\mathcal{O}$-preuve, un i-chemin satisfaisable est une solution de la $\mathcal{O}$-preuve. Voyons maintenant l'illustration de ces définitions sur un exemple de preuve de satisfaisabilité d'une $\mathcal{O}$-preuve spécifique. Dans la section 5.4, nous avions obtenu $P_{(3)} = P_3\sigma$ où σ représentait la Θ-substitution du SOCS final de la règle (3). $P_3\sigma$ peut être représenté par le DAG d'$\mathcal{O}$-preuve suivant:

$$\begin{array}{c} \Theta(fact(s(x))) >_{lex} \Theta(p(s(x))) \\ | \\ \Theta(fact(s(x))) >_{lex} \Theta(fact(p(s(x)))) \\ | \\ \Theta(fact(s(x))) >_{lex} \Theta(s(x) \times fact(p(s(x)))) \end{array}$$

où Θ est une notation simplifiée pour $\Theta_{0,k}$. Afin de déduire un $\Theta_{0,k}$ particulier et un ordre $>_{lex}$ satisfaisant cette $\mathcal{O}$-preuve, nous appliquons une étape d'Instanciation Partielle Gauche sur $\Theta_{0,k}$. Pour ce faire, nous choisissons que $(\theta_0, \geq_0)$ soit une précédence: θ_0 est une fonction associant à chaque terme son symbole de tête, et $\geq_0$ est un ordre sur $\mathcal{F}$, encore inconnu, que nous souhaitons déduire. L'IPG, appliquée au DAG d'$\mathcal{O}$-preuve précédent donne:



Maintenant, nous cherchons une solution en commençant par la plus simple possible: un 1-chemin satisfaisable dans le DAG d'$\mathcal{O}$-preuve. Les chemins étiquetés par ① et ②, parmi d'autres, sont des 1-chemins. Le 1-chemin ① est associé au t-uple $A = \langle A_0, A_1\rangle$ où

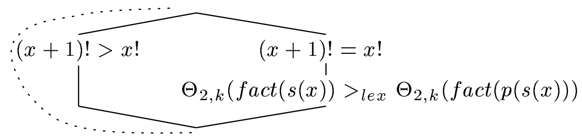
$$\begin{aligned} A_0 &= \{fact >_{\mathcal{F}} p, fact \simeq_{\mathcal{F}} fact, fact \simeq_{\mathcal{F}} \times\} \text{ et}\\ A_1 &= \{\Theta_{1,k}(fact(s(x))) >_{lex} \Theta_{1,k}(fact(p(s(x)))),\\ &\quad \Theta_{1,k}(fact(s(x))) >_{lex} \Theta_{1,k}(s(x) \times fact(p(s(x))))\} \end{aligned}$$

Le 1-chemin ② est associé au t-uple $B = \langle B_0, B_1\rangle$ où

$$\begin{aligned} B_0 &= \{fact >_{\mathcal{F}} p, fact \simeq_{\mathcal{F}} fact, fact >_{\mathcal{F}} \times\} \text{ et}\\ B_1 &= \{\Theta_{1,k}(fact(s(x))) >_{lex} \Theta_{1,k}(fact(p(s(x))))\} \end{aligned}$$

Les composants A_0 et B_0 sont tous les deux compatibles. Cependant, le 1-chemin ① n'est pas minimal puisque $B_1 \subset A_1$. Dans cet exemple particulier, il n'existe qu'un seul 1-chemin minimal qui est ②. En revanche, il n'existe pas de 1-chemin satisfaisable dans ce DAG d'$\mathcal{O}$-preuve puisque ② a un composant B_1 qui est non vide. Puisqu'il n'existe pas de 1-chemin satisfaisable, nous appliquons une deuxième IPG et nous cherchons un 2-chemin satisfaisable. Comme nous avons déjà trouvé les 1-chemins minimaux, il n'est pas nécessaire de reconsidérer tout le DAG d'$\mathcal{O}$-preuve, mais uniquement les nœuds non-instanciés des 1-chemins minimaux. Dans notre exemple, puisque l'on a un unique 1-chemin minimal, nécessairement $\geq_0$ vérifie $\{fact >_{\mathcal{F}} p, fact >_{\mathcal{F}} \times\}$ et nous devons prouver la satisfaisabilité de la $\mathcal{O}$-preuve de B restante, qui est: $\Theta_{1,k}(fact(s(x))) >_{lex} \Theta_{1,k}(fact(p(s(x))))$.

Jusqu'ici, qu'il s'agisse de l'IPG avec précédence, de la vérification de la compatibilité des composants A_0 et B_0, ou de la comparaison des 1-chemins vis-à-vis de $\subset$, tout peut être automatisé. Ainsi toute la déduction des 1-chemins minimaux peut être effectuée automatiquement. La recherche d'un 1-chemin minimal nous a permis, ici, de séparer la preuve de terminaison en deux parties: une première partie qui est résolue automatiquement (nous avons déduit une précédence), et une seconde partie qui nécessite l'expertise de l'utilisateur. Dans notre exemple, la preuve nécessitant une intéraction est la preuve de la satisfaisabilité de B_1. Pour effectuer cette preuve nous appliquons à nouveau l'IPG sur B_1 et nous recherchons un 2-chemin satisfaisable. Pour instancier θ_1, l'utilisateur peut choisir la fonction interprétant $fact$ par la fonction factorielle, s par la fonction successeur, p par le prédécesseur et 0 par zéro. Pour $\geq_1$, l'utilisateur peut choisir $\geq_{\mathbb{N}}$: c-à-d l'ordre sur les entiers naturels, comme dans [DH95]. La $\mathcal{O}$-preuve devient:



Ensuite, la validité de $(x+1)! >_{\mathbb{N}} x!$ doit être prouvée par l'utilisateur. Si l'on choisit une interprétation dans laquelle les constantes sont interprétées par des entiers naturels alors $(x+1)! >_{\mathbb{N}} x!$ est valide. Ainsi, nous avons obtenu un 2-chemin satisfaisable associé au t-uple:

$\langle\{fact >_{\mathcal{F}} p, fact \simeq_{\mathcal{F}} fact, fact >_{\mathcal{F}} \times\}, \{(x+1)! >_{\mathbb{N}} x!\}, \{\}\rangle$.

Si nous procédons de la même façon sur le DAG d'$\mathcal{O}$-preuve global pour le système de réécriture dans son ensemble, nous pouvons inférer automatiquement une précédence et un (unique) 1-chemin minimal associé au t-uple: $C = \langle C_0, C_1\rangle$ avec

$$\begin{aligned} C_0 = \{&fact >_{\mathcal{F}} s, fact >_{\mathcal{F}} p, fact \simeq_{\mathcal{F}} fact, fact >_{\mathcal{F}} \times, \times >_{\mathcal{F}} +, \times \simeq_{\mathcal{F}} \times,\\ &+ >_{\mathcal{F}} s, + \simeq_{\mathcal{F}} +\} \text{ and}\\ C_1 = \{&\Theta_{1,k}(fact(s(x))) >_{lex} \Theta_{1,k}(fact(p(s(x)))), \Theta_{1,k}(s(x)\times y) >_{lex} \Theta_{1,k}(x\times y),\\ &\Theta_{1,k}(x+s(y)) >_{lex} \Theta_{1,k}(x+y)\}. \end{aligned}$$

L'algorithme est donc parvenu à extraire la partie importante de la preuve de terminaison du système de réécriture: les trois inégalités restantes sur $(\Theta_{1,k}, >_{lex})$ de C_1. Si l'on procède à une IPG sur C_1, avec l'interprétation θ_1 définie précédemment, complétée par l'interprétation de $\times$ par la multiplication et $+$ par l'addition, alors la validité de $(x+1)! >_{\mathbb{N}} x!$, $(x+1)\times y >_{\mathbb{N}} x \times y$ et $x+y+1 >_{\mathbb{N}} x+y$ peut être montrée par l'utilisateur. Enfin l'algorithme s'achève sur un 2-chemin satisfaisable pour le DAG de la $\mathcal{O}$-preuve globale qui est:

$$\begin{aligned} \langle\{&fact >_{\mathcal{F}} s, fact >_{\mathcal{F}} p, fact \simeq_{\mathcal{F}} fact, fact >_{\mathcal{F}} \times, \times >_{\mathcal{F}} +, \times \simeq_{\mathcal{F}} \times, + >_{\mathcal{F}} s,\\ &+ \simeq_{\mathcal{F}} +\}, \{(x+1)! >_{\mathbb{N}} x!, (x+1)\times y >_{\mathbb{N}} x\times y, x+y+1 >_{\mathbb{N}} x+y\}, \{\}\rangle. \end{aligned}$$

Avec certaines instances de *gpo*, telle *lpo*, où la vérification de la compatibilité est automatique pour chaque fonction de terminaison, le processus de résolution de contraintes d'ordre peut être intégralement automatisé. Dans le cas de l'ordre *lpo*, à partir d'un ensemble d'inégalités représentant les règles du système, l'algorithme donne une précédence prouvant la terminaison du système (si une telle précédence existe). Une implantation de l'instance *lpo* du *gpo*, fournissant une procédure de décision pour l'existence d'un *lpo* pour un système donné, a été développée en ECLiPSe[9]. Pour des exemples d'exécution, voir la section 7.3.

Ce prototype est intéressant pour deux raisons. Tout d'abord, il a permis de vérifier l'efficacité de l'approche générale de résolution de contraintes d'ordre sur la structure de SOCS. Mais, d'autre part, il est intéressant en lui-même pour prouver de façon efficace la terminaison de systèmes avec *lpo* ou avec des méthodes basées sur la transformation de systèmes [BL90, Zan95]. Dans [Zan95], par exemple, un système de réécriture est transformé en un autre, étiqueté, tel que la terminaison du système étiqueté implique la terminaison du système initial. Le but est, en fait, de se ramener à des systèmes dont la terminaison peut être prouvée avec des ordres de simplification, tels *lpo*. Cependant, à cause de la transformation, le système étiqueté a très souvent un nombre de règles beaucoup plus important que le système initial. Dans ce cas, l'efficacité d'une procédure de décision pour *lpo* sur de gros systèmes de réécriture prend toute son importance.

5.6 Extension lexicographique et multi-ensemble du *gpo*

Dans cette partie, nous allons étudier l'extension de l'approche au cas des extensions lexicographique et multi-ensemble de *gpo*. La résolution d'une contrainte d'ordre sur l'extension lexicographique revient à une simple décomposition de cette contrainte jusqu'à l'obtention de contraintes d'ordre *gpo* pures. En revanche, la résolution des contraintes d'ordre sur l'extension multi-ensemble nécessite une approche particulière et également une nouvelle définition

9. ECRC Common Logic Programming System

de l'extension multi-ensemble, plus opérationnelle. Dans tous les cas, nous supposerons que les contraintes sont à résoudre pour une règle donnée dont le SOCS est déjà construit.

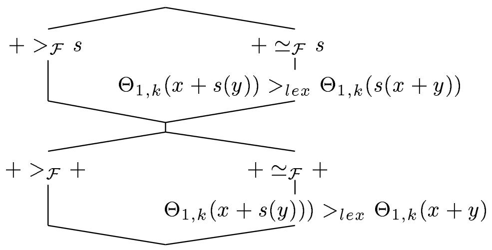
5.6.1 Extension lexicographique

La décomposition des contraintes d'ordre sur l'extension lexicographique de *gpo* est naturellement traitée par les procédures d'instanciation partielle proposées dans la section précédente, comme nous le voyons dans l'exemple suivant.

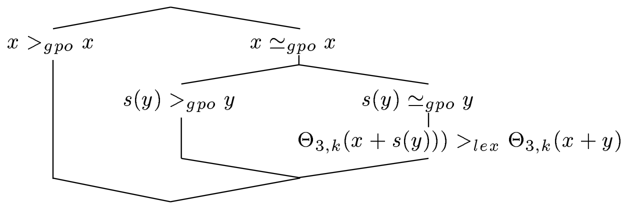
Exemple 5.4 *Soit la règle de réécriture suivante $x + s(y) \to s(x + y)$. L'$\mathcal{O}$-preuve de cette règle est:*

$$\begin{array}{c} \Theta(x+s(y)) >_{lex} \Theta(s(x+y)) \\ | \\ \Theta(x+s(y)) >_{lex} \Theta(x+y) \end{array}$$

Si l'on suppose que la première fonction de terminaison est une précédence, ceci permet de réaliser une IPG, et l'on obtient:



Dans cette $\mathcal{O}$-preuve, certaines branches peuvent être supprimées telle la branche portant la contrainte insatisfaisable $+ >_{\mathcal{F}} +$. Ensuite, un chemin possible dans le graphe de $\mathcal{O}$-preuve consiste à choisir $+ >_{\mathcal{F}} s$ et à garder comme seul littéral non-instancié: $\Theta_{1,k}(x + s(y)) >_{lex} \Theta_{1,k}(x+y)$. Puis, à partir de ce nœud unique, on choisit les fonctions de terminaison $\theta_2, \ldots, \theta_{N+1}$ (où N est l'arité maximum des symboles de fonctions de $\mathcal{F}$) de façon à représenter une extension lexicographique de gpo. Aussi les fonctions $\theta_2, \ldots, \theta_{N+1}$ extraient-elles respectivement le premier sous-terme, le deuxième, ..., le N-ième sous-terme. On obtient alors:



A partir de ce dernier graphe, il faut remarquer deux choses. Tout d'abord il n'est pas nécessaire de résoudre à nouveau les contraintes d'ordre gpo figurant dans le DAG de $\mathcal{O}$-preuve. Il suffit de les remplacer par les $\mathcal{O}$-preuves correspondantes figurant dans le SOCS de la règle, si elles existent. Les contraintes $x \simeq_{gpo} x$ et $s(y) >_{gpo} y$ sont triviales: elles sont remplacées par $\top$. Les contraintes $x >_{gpo} x$, $s(y) \simeq_{gpo} y$ sont insatisfaisables (il n'existe aucun arc de la sorte dans le SOCS de la règle). Ensuite, si les $\mathcal{O}$-preuves n'existent pas, comme dans le cas de la contrainte $s(y) \simeq_{gpo} y$ qui est insatisfaisable par gpo, les branches portant ces contraintes sont supprimées et le graphe de $\mathcal{O}$-preuve simplifié. Ceci peut mener à la suppression complète des *littéraux non-instanciés comme dans notre exemple, où le graphe final se réduit à:*

$$\begin{array}{c} \top \\ | \\ \top \end{array}$$

5.6.2 Extension multi-ensemble

Dans cette partie, nous proposons un algorithme de résolution de contraintes d'ordre sur l'extension multi-ensemble d'un préordre. Nous détaillerons également une instance de cet algorithme pour le cas particulier de la résolution de contraintes d'ordre sur l'extension multi-ensemble du préordre *gpo*. Tout d'abord, il convient de remarquer que la définition 4.2 page 32 construit une extension multi-ensemble telle que pour tout préordre $\geq = (> \cup \simeq)$, on a $\texttt{ord(}\ \geq^{mul}\ \texttt{)} \neq >^{mul}$. En fait, plus précisément, on a $\texttt{ord(}\ \geq^{mul}\ \texttt{)} \supseteq >^{mul}$. En voici un exemple. On rappelle que $M(E)$ est l'ensemble des multi-ensembles construits sur E.

Exemple 5.5 *Soit $E = \{a, b, c, d\}$ un ensemble, $A = \{\!\{a, b\}\!\}$, $B = \{\!\{c, d\}\!\}$, deux multi-ensembles de $M(E)$ et $\geq = (> \cup \simeq)$ un préordre sur E tel que $a > c$ et $b \simeq d$. Par la définition 4.2 page 32 de l'extension multi-ensemble de $\geq$, on a $A \geq^{mul} B$ puisque $a \geq c$ et $b \geq d$. On a également $B \not\geq^{mul} A$ puisque $c \not\geq a$ et $d \not\geq a$, d'où $A\,\texttt{ord(}\ \geq^{mul}\ \texttt{)}B$. En revanche, on n'a pas $A >^{mul} B$ puisque c et d ne sont pas comparables avec $>$.*

Il est clair que du point de vue de la preuve de terminaison, entre $\texttt{ord(}\ \geq^{mul}\ \texttt{)}$ et $>^{mul}$, l'ordre le plus intéressant est le premier parce qu'il permet d'orienter plus de cas. Par définition de `ord()`, on a $\texttt{ord(}\ \geq^{mul}\ \texttt{)} = \geq^{mul} \cap \not\leq^{mul}$. Or, jusqu'à présent nous avons uniquement manipulé des contraintes positives. Par conséquent l'ajout de contraintes d'ordre sur $\not\leq^{mul}$ n'est pas directement compatible avec notre approche. Nous proposons, ici, de donner une nouvelle définition de l'extension multi-ensemble n'engendrant que des contraintes d'ordre positives. Etant donné un préordre $\geq\ =\ >\ \cup \simeq$, nous donnons une définition de l'extension multi-ensemble de $\geq$ notée $\geq^M$, l'extension multi-ensemble de $>$ notée $>^M$, et de l'extension multi-ensemble de $\simeq$ notée $\simeq^M$, telles que $\geq^M = (>^M \cup \simeq^M)$.

Définition 5.19 *Soit $\geq$ un préordre sur un ensemble E, l'ordre strict $\texttt{ord(}\ \geq\ \texttt{)}$ noté $>$, et la relation d'équivalence $\texttt{eq(}\ \geq\ \texttt{)}$ notée $\simeq$. L'extension multi-ensemble de $\geq$, notée $\geq^M$ est l'union des relations $>^M$ et $\simeq^M$, où les relations $>^M$ et $\simeq^M$ sont définies inductivement par:*

Pour tout $X, Y, Z \in M(E)$ avec $Z = X \cap Y$, on a $X >^M Y$ s'il existe $X_0, Y_0, Z \in M(E)$ tels que $X = X_0 \cup Z$, $Y = Y_0 \cup Z$ et

1. *$Y_0 = \emptyset$ et $X_0 \neq \emptyset$, ou*
2. *$X_0 = \{\!\{a\}\!\} \cup X'_0$, $Y_0 = \{\!\{b_1, \ldots, b_m\}\!\} \cup Y'_0$, $m \in \mathbb{N}$, $\forall i = 1 \ldots m : a > b_i$, et $X'_0 \geq^M Y'_0$, ou*
3. *$X_0 = \{\!\{a\}\!\} \cup X'_0$, $Y_0 = \{\!\{b\}\!\} \cup Y'_0$, $a \simeq b$, et $X'_0 >^M Y'_0$.*

Pour tout $X, Y \in M(E)$, on a $X \simeq^M Y$ si

1. *$X = Y$, ou*
2. *$X = \{\!\{a\}\!\} \cup X'$, $Y = \{\!\{b\}\!\} \cup Y'$, $a \simeq b$ et $X' \simeq^M Y'$.*

Théorème 5.4 *Si $\geq$ est un préordre sur un ensemble E, dont l'ordre strict $\texttt{ord(}\ \geq\ \texttt{)}$ (noté $>$) est bien fondé, alors*

- *$\geq^M$ est un préordre sur $M(E)$,*

- $>^M$ *est un ordre bien fondé,*
- $>^M$ *est une extension monotone de* $>$*, i.e. pour tout ordre* $\succ \supseteq >$*, on a* $\succ^M \supseteq >^M$*.*

Preuve (idée) Une grande partie de la preuve est basée sur le fait qu'il existe une correspondance entre l'ordre $>^M$ sur $M(E)$ et l'ordre $>^{mul}$ (définition 4.2 page 32) sur $M(\overline{E})$, où $\overline{E}$ est l'ensemble des représentant canoniques des classes d'équivalence des éléments de E. Pour une preuve détaillée, voir section 5.8.5. $\square$

L'extension multi-ensemble ainsi définie vérifie également les deux propriétés d'incrémentalité suivantes, très intéressantes dans notre contexte car elles conduisent à la définition de règles de déduction simples.

Proposition 5.1 *Soit* $\geq$ *un préordre sur un ensemble* E*, tel que* $\geq = > \cup \simeq$ *et* X, Y, Z *des multi-ensembles sur* E *tels que* $Y \neq \emptyset$*. On a*

$$X \geq^{mul} Z \implies X \cup Y >^{mul} Z$$

Preuve A partir de $X \geq^{mul} Z$ en appliquant le cas 2. de la définition de $>^M$ successivement sur tous les éléments de Y et en choisissant un ensemble $\{\!\{b_1, \ldots, b_n\}\!\}$ tel que $n = 0$ on obtient le résultat. $\square$

Proposition 5.2 *Soit* $\geq$ *un préordre sur un ensemble* E*, tel que* $\geq = > \cup \simeq$*. Pour tout* $X_1, Y_1, X_2, Y_2 \in M(E)$ *tels que* $(X_1 \cup X_2) \cap (Y_1 \cup Y_2) = \emptyset$*, on a*

$$\begin{array}{rcl} X_1 >^M Y_1 \text{ et } X_2 \geq^M Y_2 & \implies & (X_1 \cup X_2) >^M (Y_1 \cup Y_2) \\ X_1 \simeq^M Y_1 \text{ et } X_2 \simeq^M Y_2 & \implies & (X_1 \cup X_2) \simeq^M (Y_1 \cup Y_2). \end{array}$$

Preuve (idée) On procède par induction sur le nombre d'éléments du multi-ensemble X_2. (voir section 5.8.6) $\square$

Maintenant nous donnons des règles de transition simples permettant de résoudre des contraintes d'ordre sur l'extension multi-ensemble de $\geq$ telle qu'elle est définie ci-dessus. Soit E un ensemble, $\geq = (> \cup \simeq)$ un préordre sur E. Les règles s'appliquent sur des formules logiques sans quantificateurs, de la forme $(A_1 \parallel B_1) \vee \ldots \vee (A_n \parallel B_n)$, où $B_1, \ldots, B_n$ sont des conjonctions de contraintes sur le préordre $\geq$ et $A_1, \ldots, A_n$ sont des conjonctions de contraintes d'ordre sur $\geq^M$ l'extension multi-ensemble de $\geq$. Soient $a, b \in E$, $X, Y \in M(E)$, P une conjonction de contraintes d'ordre sur l'extension multi-ensemble de $\geq$, C une conjonction de contraintes d'ordre sur $\geq$. Les règles de déduction sont les suivantes:

Subset

$$\frac{(X >^M Y \parallel C)}{(\top \parallel C)} \quad \text{si } X \supset Y$$

Décompose $>$

$$\frac{(X >^M Y \parallel C)}{\begin{array}{ll} \displaystyle\bigvee_{a \in X} [\bigvee_{Y' \subseteq Y, Y' \neq \emptyset} & (X \setminus \{\!\{a\}\!\} \geq^M Y \setminus Y' \parallel C \wedge \bigwedge_{b \in Y'} a > b) \vee \\ \displaystyle\qquad \bigvee_{b \in Y} & (X \setminus \{\!\{a\}\!\} >^M Y \setminus \{\!\{b\}\!\} \parallel C \wedge a \simeq b)] \end{array}}$$

Identity

$$\frac{(X \simeq^M Y \;\|\; C)}{(\top \;\|\; C)} \quad \text{si } X = Y$$

Décompose $\simeq$

$$\frac{(X \simeq^M Y \;\|\; C)}{\bigvee_{a\in X}[\bigvee_{b\in Y}(X \setminus \{\!\{a\}\!\} \simeq^M Y \setminus \{\!\{b\}\!\} \;\|\; a \simeq b \wedge C)]}$$

L'utilisation de ces règles en pratique donne de mauvais résultats. En fait, nous retrouvons le même type de problèmes que dans le cas de la résolution des contraintes d'ordre *gpo*: beaucoup de duplications et un nombre important de contraintes trivialement insatisfaisables dont la suppression peut nécessiter plusieurs pas de réduction. Pour illustrer notre propos, voici un exemple de déductions effectuées à l'aide de ces règles que nous avons implantées en ELAN.

Exemple 5.6 *Soient $E = \{a, b, c, d, e\}$ un ensemble quelconque, $\geq$ un préordre sur E,* `ord(` $\geq$ `)` $=>$ *et* `eq(` $\geq$ `)` $=\simeq$*. On souhaite déterminer $\geq$ à partir d'une unique contrainte sur l'extension multi-ensemble de $>$: $\{\!\{a, b\}\!\} >^M \{\!\{c, d, e\}\!\}$. Nous montrons étape après étape le résultat de la résolution de la contrainte initiale: $(\{\!\{a, b\}\!\} >^M \{\!\{c, d, e\}\!\} \;\|\; \top)$ à l'aide des règles précédentes. Les disjonctions et les multi-ensembles sont ici représentées à l'aide de listes. En appliquant ces règles sur la contrainte initiale exprimée en* ELAN *sous la forme:*

```
(a.(b.empty)>>c.(d.(e.empty)) ||T).nil end
```

on obtient:

```
[] result term:
   ((empty.a)>>empty||b>e^b>d^b>c^T).((empty.b)>>empty||a>e^a>d^a>c^T).
((empty.a)==empty||b>e^b>d^b>c^T).((empty.b)==empty||a>e^a>d^a>c^T).
((empty.a)>>(e.empty)||b>d^b>c^T).((empty.a)>>(d.empty)||b>e^b>c^T).
((empty.a)>>(c.empty)||b>e^b>d^T).((empty.a)>>((d.e).empty)||b>c^T).
((empty.a)>>((c.e).empty)||b>d^T).((empty.a)>>((c.d).empty)||b>e^T).
((empty.b)>>(e.empty)||a>d^a>c^T).((empty.b)>>(d.empty)||a>e^a>c^T).
((empty.b)>>(c.empty)||a>e^a>d^T).((empty.b)>>((d.e).empty)||a>c^T).
((empty.b)>>((c.e).empty)||a>d^T).((empty.b)>>((c.d).empty)||a>e^T).
((empty.a)==(e.empty)||b>d^b>c^T).((empty.a)==(d.empty)||b>e^b>c^T).
((empty.a)==(c.empty)||b>e^b>d^T).((empty.a)==((d.e).empty)||b>c^T).
((empty.a)==((c.e).empty)||b>d^T).((empty.a)==((c.d).empty)||b>e^T).
((empty.b)==(e.empty)||a>d^a>c^T).((empty.b)==(d.empty)||a>e^a>c^T).
((empty.b)==(c.empty)||a>e^a>d^T).((empty.b)==((d.e).empty)||a>c^T).
((empty.b)==((c.e).empty)||a>d^T).((empty.b)==((c.d).empty)||a>e^T).
((empty.a)>>(d.(c.empty))||b=e^T).((empty.b)>>(d.(c.empty))||a=e^T).
((empty.a)>>(e.(c.empty))||b=d^T).((empty.b)>>(e.(c.empty))||a=d^T).
((empty.a)>>(e.(d.empty))||b=c^T).((empty.b)>>(e.(d.empty))||a=c^T).nil
```

A cette étape de la résolution des contraintes, il faut remarquer qu'il existe des contraintes dupliquées, par exemple, la contrainte d'ordre $\{\!\{a\}\!\} >^M \{\!\{d,e\}\!\}$ apparaît 2 fois (sous les formes équivalentes `(empty.a)>>((d.e).empty)` et `(empty.a)>>(e.(d.empty))`). Or, si l'on se livre à une étude plus détaillée de ce premier résultat, on peut remarquer que quasiment toutes les contraintes construites sur `>>` sont dupliquées.

D'autre part, on trouve énormément de contraintes insatisfaisables telles `(empty.b)==(d.e)`, ou encore des contraintes de la forme `empty >> (d.e)` que l'on peut rencontrer après la deuxième étape de résolution. Au bout de trois étapes de résolution, on obtient la contrainte suivante:

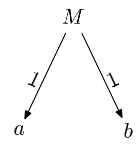
```
[] result term:
    F.(T||a>e^b>d^b>c^T).F.F.(T||a>d^b>e^b>c^T).F.F.(T||a>c^b>e^b>d^T).
F.F.(T||a>e^a>d^b>c^T).F.F.F.F.F.(T||a>e^a>c^b>d^T).F.F.F.F.F.
(T||a>d^a>c^b>e^T).F.F.F.F.F.(T||b>e^a>d^a>c^T).F.F.(T||b>d^a>e^a>c^T).
F.F.(T||b>c^a>e^a>d^T).F.F.(T||b>e^b>d^a>c^T).F.F.F.F.F.
(T||b>e^b>c^a>d^T).F.F.F.F.F.(T||b>d^b>c^a>e^T).F.F.F.F.
(T||a=e^b>d^b>c^T).(T||a=d^b>e^b>c^T).(T||a=c^b>e^b>d^T).
(T||b=e^a>d^a>c^T).(T||b=d^a>e^a>c^T).(T||b=c^a>e^a>d^T).F.
(T||a>d^a>c^b=e^T).F.F.F.F.F.(T||b>d^b>c^a=e^T).F.F.F.F.F.
(T||a>e^a>c^b=d^T).F.F.F.F.F.(T||b>e^b>c^a=d^T).F.F.F.F.F.
(T||a>e^a>d^b=c^T).F.F.F.F.F.(T||b>e^b>d^a=c^T).F.F.F.F.nil
```

où les symboles `F` représentent toutes les contraintes trivialement insatisfaisables simplifiées au cours de la résolution, et où toutes les autres sont des solutions possibles au problème du départ. Sur cet exemple pourtant très simple, on trouve un nombre important de contraintes trivialement insatisfaisables: 72, contre 24 contraintes satisfaisables.

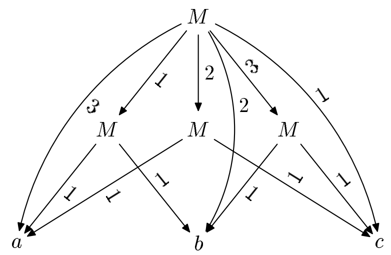
Aussi donnons-nous maintenant une méthode de résolution des contraintes d'ordre sur l'extension multi-ensemble de *gpo* évitant la construction de ces contraintes trivialement insatisfaisables obtenues par une méthode descendante. Comme dans le cas des contraintes *gpo*, afin de pallier ces deux problèmes, nous allons combiner les deux mêmes techniques: partage des termes, des contraintes, des solutions, et inférence ascendante afin de ne propager que des contraintes non-trivialement insatisfaisables. La solution que nous présentons est basée sur une représentation graphique des termes, des multi-ensembles, des contraintes d'ordre ainsi que des preuves, et peut éventuellement être superposée sur la structure adoptée dans le cas des contraintes d'ordre *gpo* de la section 5.2. Nous disposons déjà d'une structure de SOCS dans laquelle les termes, les contraintes d'ordre *gpo*, ainsi que les $\mathcal{O}$-preuves sont représentées. Nous proposons de superposer sur cette structure de SOCS des nœuds supplémentaires représentant les multi-ensembles de termes, ainsi que des arcs représentant les contraintes d'ordre sur l'extension multi-ensemble de *gpo*.

Tout d'abord, nous donnons la structure graphique représentant les multi-ensembles. Cette structure ne dépend que du nombre d'éléments contenus dans le multi-ensemble. Nous avons choisi une structure qui offrait les meilleures propriétés pour une approche de déduction ascendante, tout en conservant un partage fort et en restant compatible avec l'approche proposée dans la section 5.2 pour les contraintes d'ordre *gpo*. Pour faciliter l'approche ascendante, il est très intéressant de choisir une structure de multi-ensemble construite sur la relation de sous-ensemble, à la manière des termes: un terme est relié à ses sous-termes par la relation de sous-terme. Comme dans le cas des termes, la relation de sous-ensemble facilite beaucoup les déductions ascendantes et permet également un meilleur partage des contraintes. Par exemple, dans l'exemple 5.2 page 69, nous avons vu comment la $\mathcal{O}$-preuve P_1 d'un arc $a \xrightarrow{P_1} b$ peut être utilisée et partagée par les arcs $g(a) \xrightarrow{P_2} b$ et $f(g(a)) \xrightarrow{P_3} b$ en posant $P_2 \mapsto P_1$, et $P_3 \mapsto P_2$. Dans le cas des multi-ensembles, la relation de sous-ensemble joue le même rôle. En effet, si l'on a une $\mathcal{O}$-preuve P pour $\{\!\{a\}\!\} \geq^M \{\!\{b\}\!\}$, il est souhaitable qu'elle puisse être utilisée et partagée, grâce aux mêmes mécanismes, par les contraintes $\{\!\{a,c\}\!\} \geq^M \{\!\{b\}\!\}$ et $\{\!\{a,c,d\}\!\} \geq^M \{\!\{b\}\!\}$.

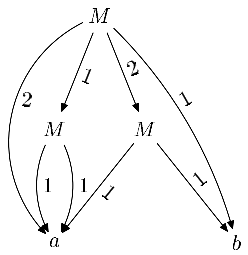
Dans un graphe OCS, nous identifierons tout multi-ensemble à un seul élément avec l'élément lui-même. Ainsi, si s est un terme, le singleton $\{\!\{s\}\!\}$ est représenté par le sommet du graphe de s. Tout autre multi-ensemble de plus d'un élément est représenté par un nœud particulier, nommé *nœud multi-ensemble*, étiqueté par le symbole M ($M \notin \mathcal{F}$) dont les fils sont soit des singletons soit des nœuds multi-ensembles. Un nœud multi-ensemble a un nombre de fils dépendant du cardinal du multi-ensemble qu'il représente. Tout nœud multi-ensemble représentant un multi-ensemble X de cardinal n, est lié par des arcs à des couples de fils, représentant des sous-ensembles de X. Ces couples de fils représentent, en fait, tous les découpages possibles de X en un singleton et un multi-ensemble de cardinal $n-1$. Tous les couples de fils sont liés au père par un arc étiqueté par un numéro de couple commun. Ces arcs sont appelés arcs multi-ensembles. Par exemple, voici la représentation du multi-ensemble $\{\!\{a,b\}\!\}$, dans laquelle il n'existe qu'une seule façon de construire $\{\!\{a,b\}\!\}$ à partir des multi-ensembles $\{\!\{a\}\!\}$ et $\{\!\{b\}\!\}$,



voici celle du multi-ensemble $\{\!\{a,b,c\}\!\}$ dans laquelle il existe 3 façons de construire $\{\!\{a,b,c\}\!\}$ à partir d'un singleton et d'un multi-ensemble de cardinal 2, nous les donnons par numéro de couple: (1) $\{\!\{c\}\!\} \cup \{\!\{a,b\}\!\}$, (2) $\{\!\{b\}\!\} \cup \{\!\{a,c\}\!\}$, et (3) $\{\!\{a\}\!\} \cup \{\!\{b,c\}\!\}$,



et voici celle du multi-ensemble $\{\!\{a,a,b\}\!\}$ dans laquelle il n'existe que 2 façons de construire $\{\!\{a,a,b\}\!\}$ à partir d'un singleton et d'un multi-ensemble de cardinal 2. Nous les donnons par numéro de couple: (1)$\{\!\{b\}\!\} \cup \{\!\{a,a\}\!\}$, et (2) $\{\!\{a\}\!\} \cup \{\!\{a,b\}\!\}$,



Nous donnons une nouvelle définition des graphes OCS intégrant les multi-ensembles.

Définition 5.20 *Un graphe OCS est un graphe $G = (V, E)$ dans lequel V est un ensemble de sommets (ou nœuds) étiquetés par des symboles de $\mathcal{F}$, des variables de $\mathcal{X}$, ou le symbole*

M, et $E \subseteq V \times V$ est l'ensemble des arcs étiquetées par S, R, $>$ ou $\sim$ pour arcs de Sous-terme, Réécriture, inégalité et équivalence respectivement. Les arcs S, R, $>$ sont orientés. Les arcs étiquetés par $>$, et $\sim$ sont également étiquetés par une $\mathcal{O}$-preuve. Les arcs de sous-terme sont également étiquetés par un entier i. Pour les arcs de sous-terme issus de sommets étiquetés par des symboles de $\mathcal{F}$, i représente le rang *du sous-terme. Les arcs de sous-terme issus de sommets étiquetés par M sont regroupés par couples et étiquetés par un entier i identique pour les deux arcs d'un même couple. Pour tout nœud $\mathsf{F} \in V$, étiqueté par $f \in \mathcal{F}$ où l'arité de f est n, pour tout $i = 1 \ldots n$, il existe $\mathsf{G}_i \in V$ et un unique arc de sous-terme $(\mathsf{F}, \mathsf{G}_i) \in E$ de rang i.*

Soit $l \to r$ une règle de réécriture, $G = (V, E)$ un graphe OCS et $(G \parallel \sigma)$ le SOCS représentant le résultat des $\mathcal{C}$-déductions effectuées sur $l \to r$. Soit L, R des multi-ensembles de sous-termes de l et r respectivement. Pour résoudre la contrainte $L >^M R$, nous proposons l'algorithme suivant:

1. suppression deux à deux des éléments communs à L et à R, i.e. si un terme t apparaît à la fois dans L et dans R, on supprime autant d'occurrences dans L et dans R de façon à ce que t ne soit plus un élément commun aux deux multi-ensembles, i.e. dans la suite par L et R on entendra $L \setminus (L \cap R)$ et $R \setminus (L \cap R)$ respectivement.

2. on construit un OCS $G' = (V', E')$ tel que

   – $V' = V'_L \cup V'_R \cup M_L \cup M_R$, où V'_L et V'_R contiennent les nœuds supérieurs de chaque graphe représentant un terme contenu dans L et R, respectivement, i.e. $V'_L = \{\mathsf{T} \in V \mid Term(\mathsf{T}) = S \in L\}$ et $V'_R = \{\mathsf{T} \in V \mid Term(\mathsf{T}) = S \in R\}$, et les ensembles M_L et M_R contiennent tous les sommets multi-ensembles étiquetés par M nécessaires à la représentation des multi-ensembles L et R, respectivement. Il faut remarquer que grâce à la suppression des égaux effectuée dans la première phase de l'algorithme, on peut assurer ici que $V'_L \cap V'_R = \emptyset$ et par conséquent, on a également $M_L \cap M_R = \emptyset$.

   – L'ensemble d'arêtes $E' \subseteq E$ contient tous les arcs d'ordre $(\mathsf{T}_1, \mathsf{T}_2) \in E$ tels que $\mathsf{T}_1 \in V'_L$ et $\mathsf{T}_2 \in V'_R$. De plus, E' contient tous les arcs multi-ensembles nécessaires à la représentation de L et de R.

3. on construit une Θ-substitution σ' telle que σ' est une restriction de σ aux variables de $\mathcal{O}$-preuve apparaissant sur les arcs de E',

4. on applique le processus de déduction multi-ensemble, défini ci-dessous, sur le SOCS $(G' \parallel \sigma')$.

Soient les règles de ce processus de déduction des arcs d'ordre de l'extension multi-ensemble définies dans la figure 5.2. Comme dans le cas des règles de déduction utilisées pour le *gpo*, nous trouvons des règles simples et leur “extension”. Les déductions effectuées par toute règle R et son extension R Extension sont similaires si l'on excepte le fait que la règle R ajoute un arc et une $\mathcal{O}$-preuve qui n'existaient pas alors que la règle R Extension complète une $\mathcal{O}$-preuve qui existait déjà. Il est nécessaire de bien dissocier ces deux règles afin de ne construire aucun arc représentant une contrainte insatisfaisable.

Nous donnons maintenant un exemple d'application de ces règles sur un exemple simple montrant que ce procédé ne construit aucun arc trivialement insatisfaisable.

Exemple 5.7 *Soit $\mathcal{F} = \{f : 2, g : 1, h : 1\}$ un ensemble de symboles, x, y, z des variables de $\mathcal{X}$ et $\{\!\{f(x,y), h(z)\}\!\} >^M \{\!\{g(y), g(z)\}\!\}$ une contrainte d'ordre sur l'extension multi-ensemble de gpo. Si nous appelons la procédure de décomposition sur la contrainte initiale*
`(fxy.hz.empty >>  gy.gz.empty ||T).nil`*, nous obtenons*

```
(T||hz>gz^hz>gy^T).(T||hz>gz^hz>gy^T).(T||fxy>gz^fxy>gy^T).
(T||fxy>gz^fxy>gy^T).(empty>>empty||fxy>gz^hz>gy^T).
(empty==empty||fxy>gz^hz>gy^T).(empty>>empty||fxy=gz^hz>gy^T).
(empty>>empty||fxy>gy^hz>gz^T).(empty==empty||fxy>gy^hz>gz^T).
(empty>>empty||fxy=gy^hz>gz^T).(empty>>empty||hz>gz^fxy>gy^T).
(empty==empty||hz>gz^fxy>gy^T).(empty>>empty||hz=gz^fxy>gy^T).
(empty>>empty||hz>gy^fxy>gz^T).(empty==empty||hz>gy^fxy>gz^T).
(empty>>empty||hz=gy^fxy>gz^T).(empty==empty||fxy=gz^hz>gy^T).
(empty==empty||fxy=gy^hz>gz^T).(empty==empty||hz=gz^fxy>gy^T).
(empty==empty||hz=gy^fxy>gz^T).(empty>>empty||fxy>gy^hz=gz^T).
(empty==empty||fxy>gy^hz=gz^T).(empty>>empty||fxy=gy^hz=gz^T).
(empty>>empty||hz>gy^fxy=gz^T).(empty==empty||hz>gy^fxy=gz^T).
(empty>>empty||hz=gy^fxy=gz^T).(empty>>empty||fxy>gz^hz=gy^T).
(empty==empty||fxy>gz^hz=gy^T).(empty>>empty||fxy=gz^hz=gy^T).
(empty>>empty||hz>gz^fxy=gy^T).(empty==empty||hz>gz^fxy=gy^T).
(empty>>empty||hz=gz^fxy=gy^T).nil
```

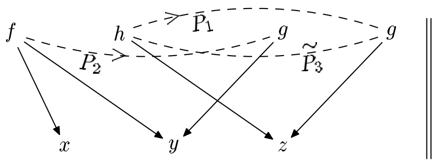
où les contraintes multi-ensembles trivialement insatisfaisables peuvent être simplifiées. Nous obtenons ainsi

```
[] result term:
F.(T||fxy>gz^hz>gy^T).F.F.(T||fxy>gy^hz>gz^T).F.F.(T||hz>gz^fxy>gy^T).
F.F.(T||hz>gy^fxy>gz^T).F.(T||fxy=gz^hz>gy^T).(T||fxy=gy^hz>gz^T).
(T||hz=gz^fxy>gy^T).(T||hz=gy^fxy>gz^T).F.(T||fxy>gy^hz=gz^T).F.F.
(T||hz>gy^fxy=gz^T).F.F.(T||fxy>gz^hz=gy^T).F.F.(T||hz>gz^fxy=gy^T).
F.nil
```

Parmi les 28 contraintes construites, seules 12 ne sont pas trivialement insatisfaisables. Or parmi ces 12 contraintes, certaines ne sont pas satisfaisables vis-à-vis de l'ordre gpo. Par exemple, la solution `(T||fxy=gy^hz>gz^T)` *est impossible puisque, quelle que soit l'instance de gpo, il est impossible d'avoir $f(x,y) \simeq_{gpo} g(y)$. Finalement, parmi ces 12 solutions, si l'ordre destiné à satisfaire les contraintes est gpo, il est encore possible d'en éliminer 8. Il ne reste plus que 4 solutions parmi lesquelles 2 duplications. Finalement il ne reste que deux solutions distinctes que voici:*

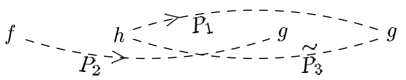
`(T||fxy>gy^hz>gz^T)` *et* `(T||fxy>gy^hz=gz^T)`

Nous voyons maintenant comment traiter cet exemple avec les règles de déduction précédentes. Nous supposons que nous disposons déjà du SOCS contenant les termes $f(x,y)$, $h(z)$, $g(y)$, $g(z)$ et les $\mathcal{O}$-preuves associées aux différentes contraintes d'ordre gpo entre ces termes:



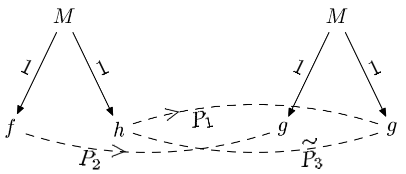
$P_1 \mapsto \Theta(h(z)) >_{lex} \Theta(g(z))$
$P_2 \mapsto \Theta(f(x,y)) >_{lex} \Theta(g(y))$
$P_3 \mapsto \Theta(h(z)) \simeq_{lex} \Theta(g(z))$

Dans ce SOCS on ne s'intéresse, en fait, qu'à un sous graphe $G' = (V', E')$ et une Θ-substitution σ'. On sait que $V' = V'_L \cup V'_R \cup M_L \cup M_R$. Dans cet exemple, V'_L est l'ensemble contenant les deux sommets étiquetés par f et h respectivement, et V'_R est l'ensemble contenant les deux sommets étiquetés par g. Par commodité, nous ne représenterons que le graphe utile: les sommets et arcs concernés par les déductions. Ceci revient à extraire du SOCS précédent, le SOCS suivant:



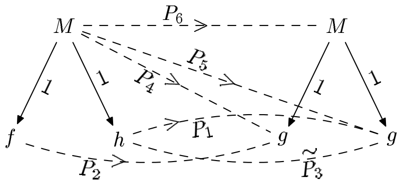
$P_1 \mapsto \Theta(h(z)) >_{lex} \Theta(g(z))$
$P_2 \mapsto \Theta(f(x,y)) >_{lex} \Theta(g(y))$
$P_3 \mapsto \Theta(h(z)) \simeq_{lex} \Theta(g(z))$

Ensuite, sur ce graphe, on ajoute la structure de multi-ensemble, et l'on obtient:



$P_1 \mapsto \Theta(h(z)) >_{lex} \Theta(g(z))$
$P_2 \mapsto \Theta(f(x,y)) >_{lex} \Theta(g(y))$
$P_3 \mapsto \Theta(h(z)) \simeq_{lex} \Theta(g(z))$

Finalement, seuls trois arcs sont constructibles et le SOCS obtenu après application des règles multi-ensemble est le suivant:



$P_1 \mapsto \Theta(h(z)) >_{lex} \Theta(g(z))$
$P_2 \mapsto \Theta(f(x,y)) >_{lex} \Theta(g(y))$
$P_3 \mapsto \Theta(h(z)) \simeq_{lex} \Theta(g(z))$

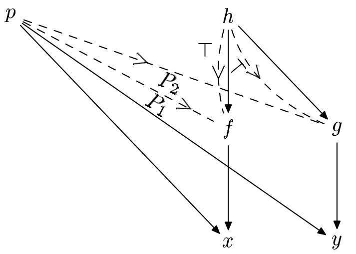
$P_4 \mapsto P_2$
$P_5 \mapsto P_1 \wedge P_3$
$P_6 \mapsto (P_1 \wedge P_2) \vee (P_3 \wedge P_3)$

Les règles de déductions multi-ensemble sont généralisables pour la résolution de contraintes sur l'extension multi-ensemble de n'importe quel ordre, elle ne sont pas particulière à l'ordre *gpo*. En revanche, il existe une optimisation possible lorsque cet ordre est *gpo*. En effet, dans ce cas, en saturant au préalable le graphe avec tous les arcs triviaux possible à l'aide de l'ensemble de règles de la figure 5.3, on déduit rapidement les solutions triviales si elles existent. L'application de ces règles doit précéder l'application des règles de la figure 5.2.

Il faut noter que les arcs étiquetés par $\top$ sont bloquants, i.e. ils ne peuvent, en aucun cas, être étendus par les règles d'extension (**SINGLETON Extension**, **SUBTERM Multiset Extension**, **COMMON $>$ Extension**, **COMMON $\sim$ Extension**) du premier ensemble. Ceci code naturellement le fait que les solutions triviales sont nécessairement les meilleures et qu'il n'est pas nécessaire d'en envisager d'autres.

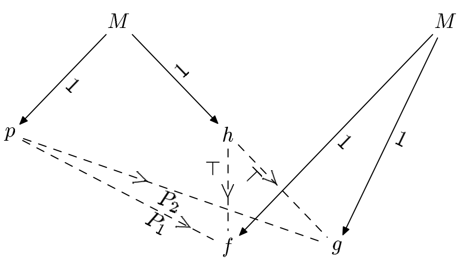
Afin d'illustrer l'intérêt de ce deuxième ensemble dans le cas de la résolution de contraintes d'ordre sur l'extension multi-ensemble de *gpo*, nous développons maintenant un deuxième exemple mettant en valeur la simplification du graphe SOCS.

Exemple 5.8 *Soit $\mathcal{F} = \{f : 1, g : 1, h : 2, p : 2\}$ un ensemble de symboles, x, y des variables de $\mathcal{X}$ et $\{\!\{p(x,y), h(f(x), g(y))\}\!\} >^M \{\!\{f(x), g(y)\}\!\}$ une contrainte d'ordre sur l'extension multi-ensemble de gpo. Le SOCS correspondant aux termes $p(x,y)$, $h(f(x), g(y))$, $f(x)$, $g(y)$ est le suivant:*



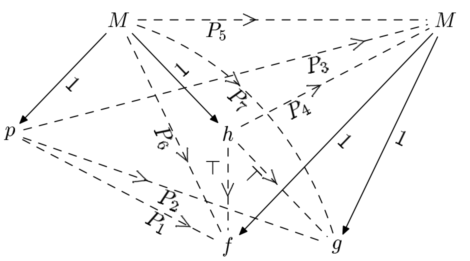
$P_1 \mapsto \Theta(p(x,y)) >_{lex} \Theta(f(x))$
$P_2 \mapsto \Theta(p(x,y)) >_{lex} \Theta(g(y))$

Si l'on extrait le graphe utile et que l'on superpose la structure de multi-ensemble au SOCS, nous obtenons:



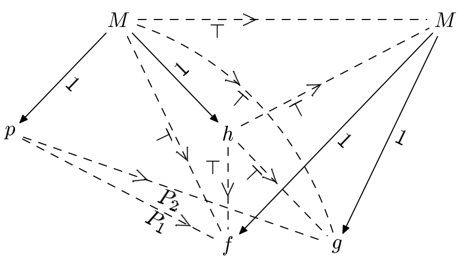
$$P_1 \mapsto \Theta(p(x,y)) >_{lex} \Theta(f(x))$$
$$P_2 \mapsto \Theta(p(x,y)) >_{lex} \Theta(g(y))$$

Si nous appliquons directement le premier ensemble de règles multi-ensemble, nous obtenons le résultat suivant:



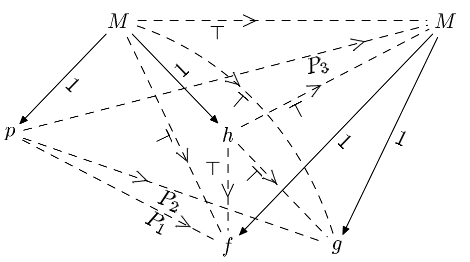
$$P_1 \mapsto \Theta(p(x,y)) >_{lex} \Theta(f(x))$$
$$P_2 \mapsto \Theta(p(x,y)) >_{lex} \Theta(g(y))$$
$$P_3 \mapsto P_1 \wedge P_2$$
$$P_4 \mapsto \top$$
$$P_5 \mapsto P_3 \vee P_4 \vee (P_1 \wedge \top) \vee (P_2 \wedge \top)$$
$$P_6 \mapsto P_1 \vee \top$$
$$P_7 \mapsto P_2 \vee \top$$

En revanche, si on applique auparavant l'ensemble de règles de simplification on obtient les preuves triviales suivantes:



$$P_1 \mapsto \Theta(p(x,y)) >_{lex} \Theta(f(x))$$
$$P_2 \mapsto \Theta(p(x,y)) >_{lex} \Theta(g(y))$$

Or, comme nous l'avons déjà signalé, les preuves triviales sont bloquantes: elles ne peuvent ni être remplacées ni être étendues. Ainsi, si on applique les règles multi-ensemble sur ce dernier SOCS, on obtient un SOCS complet plus simple:



$$P_1 \mapsto \Theta(p(x,y)) >_{lex} \Theta(f(x))$$
$$P_2 \mapsto \Theta(p(x,y)) >_{lex} \Theta(g(y))$$
$$P_3 \mapsto P_1 \wedge P_2$$

Nous donnons maintenant les théorèmes de correction et de complétude des ensembles de règles précédents pour le cas particulier où l'on s'intéresse à l'extension multi-ensemble de l'ordre *gpo*. Soit $\mathcal{C}_M$ l'ensemble contenant les règles de la figure 5.2 et de la figure 5.3. Tout d'abord nous définissons la fonction $Mult$ qui associe à chaque nœud d'un graphe OCS le multi-ensemble qu'il représente.

Soit $G' = (V', E')$ un OCS tel que $V' = V'_G \cup V'_D \cup M_G \cup M_D$, $\mathsf{S}, \mathsf{X}_1, \mathsf{X}_2 \in V'_G$, $\mathsf{Y}, \mathsf{Y}_1, \mathsf{Y}_2, \in V'_D$, $\mathsf{M}_1 \in M_G$, $\mathsf{M}_2 \in M_D$, $\mathsf{M}_1, \mathsf{M}_2$ sont étiquetés par M, S est étiqueté par un symbole de $\mathcal{F} \cup \mathcal{X}$, $i, j \in \mathbb{N}$, $P, P', P_1, P_2 \in \mathcal{X}_\Theta$, $\alpha \in \mathcal{P}$, σ est une Θ-substitution.

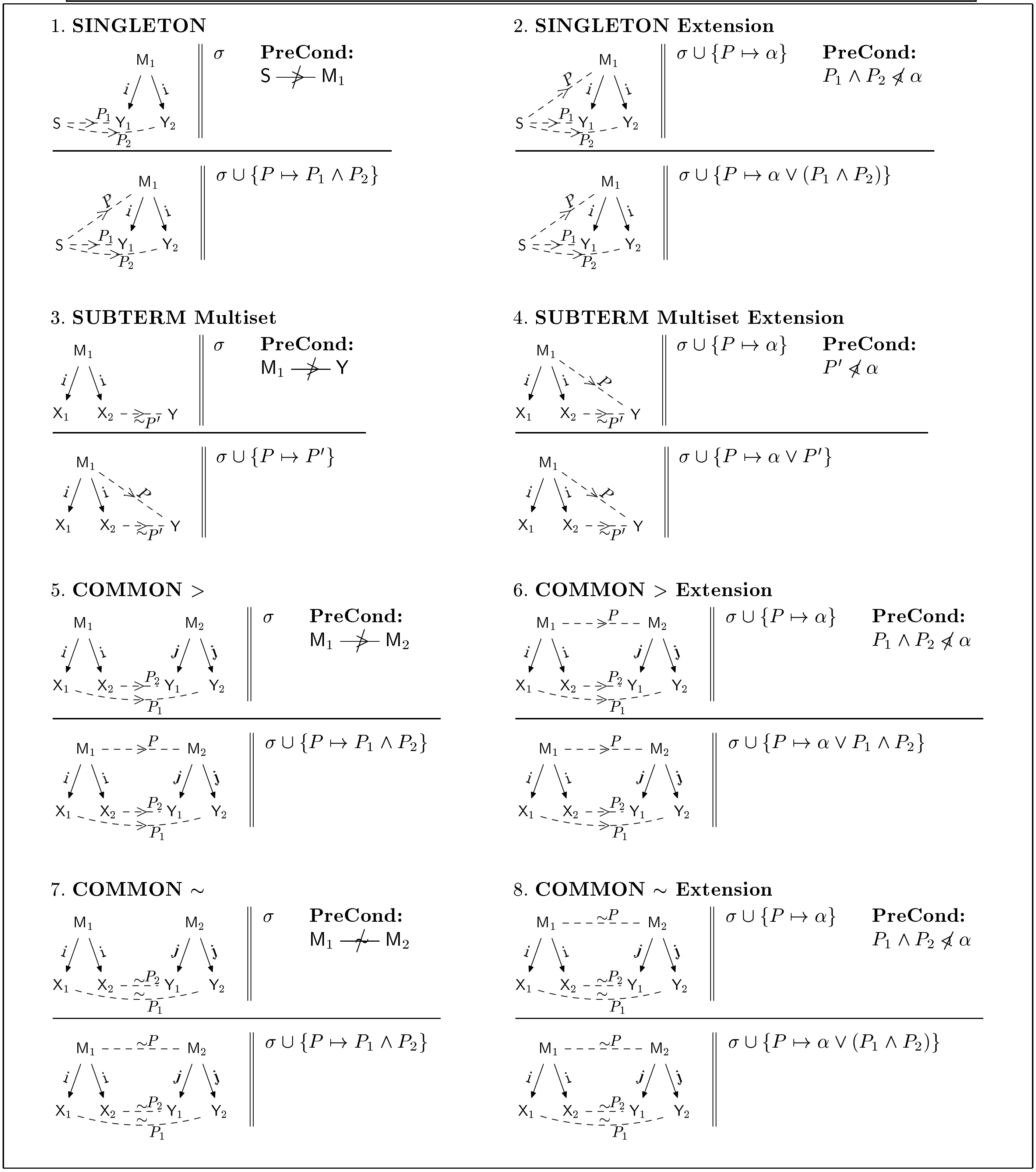
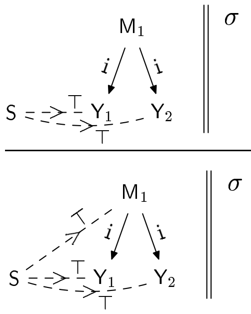


FIG. 5.2 – *Les règles de l'extension multi-ensemble*

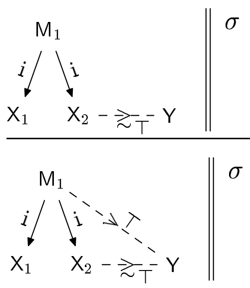
Soit $G' = (V', E')$ un OCS tel que $V' = V'_G \cup V'_D \cup M_G \cup M_D$, $\mathsf{S}, \mathsf{X}_1, \mathsf{X}_2 \in V'_G$, $\mathsf{Y}, \mathsf{Y}_1, \mathsf{Y}_2, \in V'_D$, $\mathsf{M}_1 \in M_G$, $\mathsf{M}_2 \in M_D$, $\mathsf{M}_1, \mathsf{M}_2$ sont étiquetés par M, S est étiqueté par un symbole de $\mathcal{F} \cup \mathcal{X}$, $i, j \in \mathbb{N}$, σ est une Θ-substitution.

9. **SINGLETON** Simplification



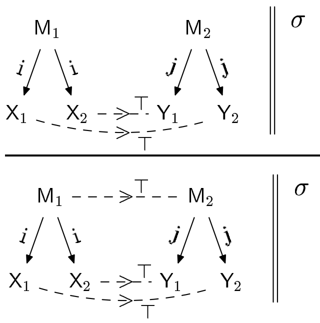
PreCond:
$\mathsf{S} \not> \mathsf{M}_1$

10. **SUBTERM Multiset** Simplification



PreCond:
$\mathsf{M}_1 \not> \mathsf{Y}$

11. **COMMON** $>$ Simplification



PreCond:
$\mathsf{M}_1 \not> \mathsf{M}_2$

FIG. 5.3 – *Règles supplémentaires pour l'extension multi-ensemble de gpo*

Définition 5.21 *Soit $G = (V, E)$ un OCS graph et $\mathsf{F} \in V$. La fonction Mult de V vers $M(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ est définie inductivement comme suit:*

1. *si F est étiqueté par un symbole $f \in \mathcal{F}$, alors $Mult(\mathsf{F}) = \{\!\{Term(\mathsf{F})\}\!\}$,*
2. *si F est étiqueté par le symbole M, alors $Mult(\mathsf{F}) = Mult(\mathsf{X}) \cup Mult(\mathsf{Y})$ où, $(\mathsf{F}, \mathsf{X}), (\mathsf{F}, \mathsf{Y}) \in E$ sont des arcs multi-ensemble étiquetés par le même indice de couple i.*

Définition 5.22 *Étant donné un SOCS $(G \parallel \sigma)$, et un arc $\mathsf{X} \overset{P}{\twoheadrightarrow} \mathsf{Y}$ de G (resp. $\mathsf{X} \overset{P}{\rightsquigarrow} \mathsf{Y}$) est* correct *vis-à-vis de l'extension multi-ensemble de gpo si pour tout $\Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex})$ tel que $\Phi \models P$, on a $Mult(\mathsf{X}) \succ_{gpo}^{\Phi,M} Mult(\mathsf{Y})$ (resp. $Mult(\mathsf{X}) \approx_{gpo}^{\Phi,M} Mult(\mathsf{Y})$). Un SOCS $(G \parallel \sigma)$ est* correct *vis-à-vis de l'extension multi-ensemble du gpo si c'est le cas pour tout arc de G.*

Le SOCS sur lequel débute l'application des règles de $\mathcal{C}_M$-déduction est un SOCS correct vis-à-vis de *gpo* sans arcs d'ordre entre les sommets multi-ensembles. En pratique, il est envisageable d'appliquer les règles de $\mathcal{C}$-déduction sur un SOCS initial S (au sens défini dans la section 5.3) représentant une règle de réécriture jusqu'à obtenir une $\mathcal{C}$-forme normale S'. Ensuite, on ajoute à S' les sommets représentant les multi-ensembles sur lesquels nous souhaitons résoudre des contraintes d'ordre sur l'extension multi-ensemble et nous normalisons S' à l'aide des règles de $\mathcal{C}_M$-déduction.

Théorème 5.5 *(Correction) Pour tout SOCS S' correct vis-à-vis de gpo, si $S' \vdash_{\mathcal{C}_M}^{*} S''$ alors S'' est correct vis-à-vis de l'extension multi-ensemble de gpo.*

Théorème 5.6 *(Complétude) Soit $(G \parallel \sigma)$ un SOCS en $\mathcal{C}_M$-forme normale, tel que $G = (V, E)$. Pour tous nœuds $\mathsf{F}, \mathsf{G} \in V$, t.q. $Mult(\mathsf{F}) = X$ et $Mult(\mathsf{G}) = Y$, où $X, Y \in M(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ et $X \cap Y = \emptyset$:*

$$\forall \Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex}) \text{ t.q. } X \succ_{gpo}^{\Phi,M} Y, \text{ il existe un arc } \mathsf{F} \overset{P}{\twoheadrightarrow} \mathsf{G} \text{ dans } G \text{ t.q. } \Phi \models P\sigma$$

$$\forall \Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex}) \text{ t.q. } X \approx_{gpo}^{\Phi,M} Y, \text{ il existe un arc } \mathsf{F} \overset{P}{\rightsquigarrow} \mathsf{G} \text{ dans } G \text{ t.q. } \Phi \models P\sigma.$$

5.7 Conclusion sur la résolution des contraintes d'ordre *gpo*

Dans ce chapitre, nous espérons avoir montré le double apport de la résolution de contraintes d'ordre pour la synthèse des ordres de terminaison: clarté et efficacité. En exprimant le problème de la terminaison d'un système de réécriture sur la forme d'une contrainte d'ordre, on en donne une représentation compacte, exacte et opérationnelle. Pour ce qui est de l'efficacité, nous avons proposé un algorithme de résolution de contraintes d'ordre *gpo* limitant le nombre de dupplications et de contraintes insatisfaisables grâce à des techniques de partage de termes, de contraintes et d'$\mathcal{O}$-preuves. En outre, dans un mode semi-automatique, cet algorithme permet de limiter l'intéraction avec l'utilisateur aux points les plus difficiles de la résolution. Les techniques d'optimisation de la résolution des contraintes d'ordre mises en œuvre ici ne sont pas limitées au cas du *gpo*. En guise d'exemple, nous avons montré qu'elles pouvaient être transposées au cas des contraintes d'ordre sur l'extension multi-ensemble d'un préordre quelconque.

5.8 Preuves

5.8.1 *gpo* est un préordre sur $\mathcal{T}(\mathcal{F},\mathcal{X})$ avec la propriété de sous-terme

Toutes les preuves des lemmes suivants sur $\mathcal{T}(\mathcal{F},\mathcal{X})$ sont identiques aux preuves respectives sur $\mathcal{T}(\mathcal{F})$ données dans [DH95]. Par conséquent, nous avons choisi de simplement pointer et prouver les propriétés additionnelles.

Lemme 5.1 *(Symétrie). Si $s \simeq_{gpo} t$ alors $t \simeq_{gpo} s$.*

Preuve La preuve est similaire à la preuve du lemme 1 de [DH95] avec la propriété additionnelle suivante à prouver: $\Theta(s) \simeq_{lex} \Theta(t)$ implique que $\Theta(t) \simeq_{lex} \Theta(s)$. A partir de la définition de $\Theta(s) \simeq_{lex} \Theta(t)$ on obtient que $\forall\sigma$ t.q. $s\sigma, t\sigma \in T(F)$, on a $\Theta(s\sigma) \simeq_{lex} \Theta(t\sigma)$. La relation $\simeq_i$ est supposée symétrique sur les termes clos, par définition de *gpo*. On peut remarquer que lorsque $\simeq_i$ est l'extension multi-ensemble de $\simeq_{gpo}$, la symétrie de $\simeq_i$ vient de la symétrie de *gpo* sur les termes clos. Ainsi, nous obtenons que $\simeq_{lex}$ est symétrique sur les termes clos. D'où, $\forall\sigma$ t.q. $s\sigma, t\sigma \in \mathcal{T}(\mathcal{F})$, on a $\Theta(t\sigma) \simeq_{lex} \Theta(s\sigma)$, qui est la définition de $\Theta(t) \simeq_{lex} \Theta(s)$ sur $\mathcal{T}(\mathcal{F},\mathcal{X})$. $\square$

Soient $s, t, u \in \mathcal{T}(\mathcal{F},\mathcal{X})$.

Lemme 5.2 *Pour le gpo, $s \geq_{gpo} t$ implique $s >_{gpo} t|_p$ pour tout sous-terme strict $t|_p$ de t.*

Preuve La preuve ne dépend ni de la définition de Θ ni de celle de $>_{lex}$ mais uniquement de la définition du *gpo* lui-même. Comme nous utilisons la même définition du *gpo* que [DH95], la preuve est similaire au cas clos: lemme 2 de [DH95]. $\square$

Lemme 5.3 *(Sous-terme). Le gpo satisfait la propriété de sous-terme stricte:*
$\forall s$: $f(\ldots, s, \ldots) >_{gpo} s$.

Preuve Là encore, la preuve ne dépend ni de la définition de Θ ni de celle de $>_{lex}$ mais uniquement de la définition du *gpo* lui-même. Comme nous utilisons la même définition du *gpo* que [DH95], la preuve est similaire au cas clos: lemme 3 de [DH95]. $\square$

Lemme 5.4 *(Reflexivité). L'ordre général sur les chemins $\geq_{gpo}$ est réflexif.*

Preuve La preuve est similaire à la preuve du lemme 4 de [DH95] avec la propriété additionnelle à prouver: $\Theta(s) \simeq_{lex} \Theta(s)$. La relation $\simeq_i$ est réflexive sur les termes clos. Par conséquent, $\simeq_{lex}$ est réflexive sur les termes clos également. Lorsque $\simeq_i$ est l'extension multi-ensemble de $\simeq_{gpo}$, la réflexivité de $\simeq_i$ vient du fait que $\simeq_{gpo}$ est réflexif sur les termes clos. D'où, $\forall\sigma$ t.q. $s\sigma \in \mathcal{T}(\mathcal{F})$, on a $\Theta(s\sigma) \simeq_{lex} \Theta(s\sigma)$, et par la définition 4.20, $\Theta(s) \simeq_{lex} \Theta(s)$. $\square$

Lemme 5.5 *Pour le gpo, $s \geq_{gpo} t$ implique que $u[s] >_{gpo} t$ pour tout contexte non-vide $u[\,]$.*

Preuve La preuve ne dépend ni de la définition de Θ ni de celle de $>_{lex}$ mais uniquement de la définition du *gpo* lui-même. Comme nous utilisons la même définition du *gpo* que [DH95], la preuve est similaire au cas clos: lemme 5 de [DH95]. $\square$

Lemme 5.6 *Soient $s, t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $s \geq_{gpo} t$ implique $\mathcal{V}ar(s) \supseteq \mathcal{V}ar(t)$.*

Preuve Supposons que $s \geq_{gpo} t$ et $\mathcal{V}ar(s) \not\supseteq \mathcal{V}ar(t)$. Il existe donc au moins une variable $x \in \mathcal{V}ar(t)$ et $x \notin \mathcal{V}ar(s)$. Soit $C[.]$ un contexte non-vide tel que $t = C[x]$. Soit σ une substitution telle que $Dom(\sigma) = \mathcal{V}ar(s) \cup \mathcal{V}ar(t) \setminus \{x\}$, $s\sigma \in \mathcal{T}(\mathcal{F})$, et $\mathcal{V}ar(t\sigma) = \mathcal{V}ar(C[x]\sigma) = \{x\}$. Soit $\sigma' = \sigma \cup \{x \mapsto s\sigma\}$. Dans ce cas, on a $s\sigma', t\sigma' \in \mathcal{T}(\mathcal{F})$, $s\sigma' = s\sigma$ et $t\sigma' = C\sigma'[s\sigma]$. Grâce à la propriété de sous-terme du *gpo*, on obtient que $C\sigma'[s\sigma] >_{gpo} s\sigma$. Donc $t\sigma' >_{gpo} s\sigma'$ avec $s\sigma', t\sigma' \in \mathcal{T}(\mathcal{F})$ qui est une contradiction avec $s \geq_{gpo} t$ et la proposition 4.3. $\square$

Lemme 5.7 *(Transitivité) Soient* $s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

1. $s >_{gpo} t \geq_{gpo} u \implies s >_{gpo} u$;
2. $s \simeq_{gpo} t >_{gpo} u \implies s >_{gpo} u$;
3. $s \simeq_{gpo} t \simeq_{gpo} u \implies s \simeq_{gpo} u$.

Preuve La preuve est similaire à la preuve du lemme 6 dans [DH95] compte-tenu de trois propriétés additionnelles à prouver:

1. pour $s >_{gpo} t \geq_{gpo} u \implies s >_{gpo} u$, la propriété additionnelle à prouver est que $\Theta(s) >_{lex} \Theta(t)$ et $\Theta(t) \geq_{lex} \Theta(u)$ implique $\Theta(s) >_{lex} \Theta(u)$. De $\Theta(s) >_{lex} \Theta(t)$ et $\Theta(t) \geq_{lex} \Theta(u)$ ainsi que de la définition 4.20, on extrait:

$$\forall \sigma \text{ t.q. } s\sigma, t\sigma \in \mathcal{T}(\mathcal{F}),\ \Theta(s\sigma) >_{lex} \Theta(t\sigma), \text{ et}$$

$$(\forall \sigma' \text{ t.q. } t\sigma', u\sigma' \in \mathcal{T}(\mathcal{F}),\ \Theta(t\sigma') >_{lex} \Theta(u\sigma')) \text{ ou}$$
$$(\forall \sigma' \text{ t.q. } t\sigma', u\sigma' \in \mathcal{T}(\mathcal{F}),\ \Theta(t\sigma') \simeq_{lex} \Theta(u\sigma'))\ .$$

Puisque $s >_{gpo} t$ et $t \geq_{gpo} u$, grâce au lemme 5.6, on sait que $\mathcal{V}ar(s) \supseteq \mathcal{V}ar(t)$, $\mathcal{V}ar(t) \supseteq \mathcal{V}ar(u)$. Par transitivité de $\subseteq$, on a $\mathcal{V}ar(s) \supseteq \mathcal{V}ar(u)$. D'où, $\forall \sigma$ t.q. $s\sigma \in \mathcal{T}(\mathcal{F})$, on a également $t\sigma, u\sigma \in \mathcal{T}(\mathcal{F})$, et on obtient:

$$\forall \sigma \text{ t.q. } s\sigma, t\sigma, u\sigma \in \mathcal{T}(\mathcal{F}),\ \Theta(s\sigma) >_{lex} \Theta(t\sigma) \text{ et } \Theta(t\sigma) >_{lex} \Theta(u\sigma), \text{ ou}$$

$$\forall \sigma \text{ t.q. } s\sigma, t\sigma, u\sigma \in \mathcal{T}(\mathcal{F}),\ \Theta(s\sigma) >_{lex} \Theta(t\sigma) \text{ et } \Theta(t\sigma) \simeq_{lex} \Theta(u\sigma).$$

De la transitivité de $>_i$ et de $\simeq_i$, on peut déduire la transitivité de $>_{lex}$ et $\simeq_{lex}$ sur les termes clos. Si $>_i$ (resp. $\simeq_i$) est l'extension multi-ensemble de $>_{gpo}$ (resp. $\simeq_{gpo}$), la transitivité de $>_i$ (resp. $\simeq_i$) vient de la transitivité de $>_{gpo}$ (resp. $\simeq_{gpo}$) sur les termes clos. Par la transitivité de $>_{lex}$ et de $\simeq_{lex}$ on obtient:

$$\forall \sigma \text{ t.q. } s\sigma, u\sigma \in \mathcal{T}(\mathcal{F}),\ \Theta(s\sigma) >_{lex} \Theta(u\sigma).$$

2. pour $s \simeq_{gpo} t >_{gpo} u \implies s >_{gpo} u$, la propriété additionnelle à prouver est que $\Theta(s) \simeq_{lex} \Theta(t) >_{lex} \Theta(u)$ implique que $\Theta(s) >_{lex} \Theta(u)$. Ceci peut être montré comme dans le cas précédent.

3. pour $s \simeq_{gpo} t \simeq_{gpo} u \implies s \simeq_{gpo} u$, la propriété à prouver est que $\Theta(s) \simeq_{lex} \Theta(t) \simeq_{lex} \Theta(u)$ implique que $\Theta(s) \simeq_{lex} \Theta(u)$. Ceci peut être montré comme dans le premier cas.

□

Lemme 5.8 *(Irréflexivité). Pour tout terme s, $s \not>_{gpo} s$.*

Preuve La preuve ne dépend ni de la définition de Θ ni de celle de $>_{lex}$ mais uniquement de la définition du *gpo* lui-même. Comme nous utilisons la même définition du *gpo* que [DH95], la preuve est similaire au cas clos: lemme 7 de [DH95]. □

Lemme 5.9 *Si $s >_{gpo} t$, alors $t \not\geq_{gpo} s$.*

Preuve Supposons que $t \geq_{gpo} s$. Par transitivité, on obtient que $s >_{gpo} s$, ce qui contredit le lemme 5.8. □

Lemme 5.10 *$s \geq_{gpo} t \geq_{gpo} s$ si et seulement si $s \simeq_{gpo} t$.*

Par les lemmes précédents, on obtient que $\geq_{gpo}$ est un préordre ayant la propriété de sous-terme.

5.8.2 Correction des règles de $\mathcal{C}$-déduction

Nous montrons d'abord trois lemmes qui sont nécessaires à la preuve de correction.

Lemme 5.11 *Soient $P, Q \in \mathcal{P}$, $\Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex})$, on a:*

$$P \lhd Q \text{ et } \Phi \models P \quad \Longrightarrow \quad \Phi \models Q.$$

Preuve On procède par induction sur la taille de Q. D'après la définition de $P \lhd Q$, soit:

- $Q = P$. Dans ce cas, puisque $\Phi \models P$, on a $\Phi \models Q$,
- $Q = A \vee B$ et on a $P \lhd A$ ou $P \lhd B$. En appliquant, l'hypothèse d'induction, on obtient alors que $\Phi \models A$ ou $\Phi \models B$. Enfin, en appliquant la définition de $\models$, on obtient $\Phi \models Q$.

□

Lemme 5.12 *Étant donnés $P, Q \in \mathcal{P}$ et σ une Θ-substitution,*

$$P \lhd Q \quad \Longrightarrow \quad P\sigma \lhd Q\sigma.$$

Preuve Là encore, on procède par induction sur la taille de Q. D'après la définition de $P \lhd Q$, soit:

- $Q = P$. Dans ce cas, $Q\sigma = P\sigma$, donc $P\sigma \lhd Q\sigma$,
- $Q = A \vee B$ et on a $P \lhd A$ ou $P \lhd B$. On a $Q\sigma = A\sigma \vee B\sigma$ et par induction, on obtient que $P\sigma \lhd A\sigma$ ou $P\sigma \lhd B\sigma$. Enfin en appliquant la définition de $\lhd$, $P\sigma \lhd Q\sigma$.

□

Dans la suite, $s \stackrel{P}{-\!\diamond\!-} t$ représente n'importe quel arc d'ordre, qu'il s'agisse d'un arc $s \stackrel{P}{-\!\!>} t$, d'un arc $s \stackrel{P}{-\!\!<} t$ ou d'un arc $s \stackrel{P}{-\!\!\sim} t$. Nous abrégerons parfois *correct par rapport à gpo* en *correct.*

Définition 5.23 *Soit $(G \parallel \sigma)$ un SOCS. Soit $s \stackrel{P}{-\!\diamond\!-} t$ et $s' \stackrel{P'}{-\!\diamond\!-} t'$ des arcs d'ordre de G. L'arc*

- *$s' \stackrel{P'}{-\!\diamond\!-} t'$ est* au-dessus de *$s \stackrel{P}{-\!\diamond\!-} t$ si s et t sont des sous-termes (éventuellement stricts) de s' et t' respectivement,*
- *$s' \stackrel{P'}{-\!\diamond\!-} t'$ est* couvert par *$s \stackrel{P}{-\!\diamond\!-} t$ s'il n'est pas au-dessus. de $s \stackrel{P}{-\!\diamond\!-} t$.*

Lemme 5.13 *Soient S un SOCS initial et $S \vdash_{\mathcal{C}}^{*} S'$ tel que $S' = (G \parallel \sigma \cup \{P \mapsto \alpha\})$. Soit $s \stackrel{P}{-\!\diamond\!-} t$ un arc d'ordre de G tel que $P \in \mathcal{X}_{\Theta}$.*

1. *Si $s' \stackrel{P'}{-\!\diamond\!-} t'$ est un arc de G couvert par $s \stackrel{P}{-\!\diamond\!-} t$ alors $P'(\sigma \cup \{P \mapsto \alpha\}) = P'\sigma$, et*
2. *$\alpha(\sigma \cup \{P \mapsto \alpha\}) = \alpha\sigma$.*

Preuve Lorsqu'un arc $s \stackrel{P}{-\!\diamond\!-} t$ est construit dans G, P est nécessairement une nouvelle variable. Donc, P ne peut apparaître dans les $\mathcal{O}$-preuves d'autres arcs que si le nouvel arc $s \stackrel{P}{-\!\diamond\!-} t$ est utilisé comme une précondition pour d'autres $\mathcal{C}$-déductions. Dans ce cas, les nouveaux arcs ajoutés (dont les $\mathcal{O}$-preuves contiennent P) sont nécessairement des arcs *au-dessus de $s \stackrel{P}{-\!\diamond\!-} t$* puisque la construction des arcs par les règles de $\mathcal{C}$ est effectué du bas vers le haut. Ainsi, la variable P ne peut pas apparaître dans α ni dans les $\mathcal{O}$-preuves d'arcs couverts par de $s \stackrel{P}{-\!\diamond\!-} t$. D'où,

1. pour tout arc $s' \stackrel{P'}{-\!\diamond\!-} t'$ couvert par $s \stackrel{P}{-\!\diamond\!-} t$, on a $P'(\sigma \cup \{P \mapsto \alpha\}) = P'\sigma$, et
2. $\alpha(\sigma \cup \{P \mapsto \alpha\}) = \alpha\sigma$.

□

Nous rappelons le théorème de correction des règles de déduction.

Théorème 5.1 *Pour tout SOCS initial S, si $S \vdash_{\mathcal{C}}^{*} S'$ alors S' est correct par rapport à gpo.*

Preuve Dans la suite, on supposera que $S = (G \parallel \rho)$, $S' = (G' \parallel \varphi)$; $\mathsf{F}, \mathsf{U}, \mathsf{G}, \mathsf{T}_1, \ldots, \mathsf{T}_m$ et V sont des nœuds de G (et donc de G') t.q. $Term(\mathsf{F}) = s$, $Term(\mathsf{G}) = t$, $Term(\mathsf{U}) = u$, $Term(\mathsf{V}) = v$, $Term(\mathsf{T}_1) = t_1, \ldots, Term(\mathsf{T}_m) = t_m$, F est étiqueté par f et G est étiqueté par g. En premier, nous prouvons que l'application de toute règle de déduction de $\mathcal{C}$ sur un SOCS S correct mène à un autre SOCS S' correct. Par cas sur la règle de déduction appliquée:

SUBTERM Property Cette règle s'applique si l'on a un arc de sous-terme entre F et U. Dans ce cas, S' est obtenu par addition d'un arc $\mathsf{F} \stackrel{\top}{-\!\!>} \mathsf{U}$ à G. Comme chaque arc de G est supposé correct, il est suffisant de prouver que ce nouvel arc est correct. Ceci est trivialement vrai puisque $Term(\mathsf{U}) = u$, $Term(\mathsf{F}) = f(\ldots, u, \ldots)$ et $\forall\Phi\ f(\ldots, u, \ldots) \succ_{gpo}^{\Phi} u$, grâce à la propriété de sous-terme de *gpo*.

SUBTERM First Comme dans le cas de la règle **SUBTERM Property**, on a $s = f(\dots, u, \dots)$, et G' ne diffère de G que par un nouvel arc d'inégalité. L'extension de la Θ-substitution ρ en $\varphi = \rho \cup \{P \mapsto P'\}$ ne concerne pas les arcs de G existants, puisque P est une nouvelle variable. Nous montrons alors que ce nouvel arc est correct. Nous savons déjà qu'il existe un arc $\mathsf{U} \xrightarrow{P'} \mathsf{V}$ ou $\mathsf{U} \overset{P'}{\rightsquigarrow} \mathsf{V}$ dans G. Puisque S est correct, $\mathsf{U} \xrightarrow{P'} \mathsf{V}$ (ou $\mathsf{U} \overset{P'}{\rightsquigarrow} \mathsf{V}$) est correct, et donc $\forall \Phi$ t.q. $\Phi \models P'\rho$, on a $u \succapprox_{gpo}^{\Phi} v$. Le nouvel arc est étiqueté par P. La nouvelle Θ-substitution est $\varphi = \rho \cup \{P \mapsto P'\}$. D'où, $P\varphi = P'\varphi$. Puisque P est une nouvelle variable, P ne peut pas apparaître dans P' ni dans ρ, d'où $P'\varphi = P'\rho$. Ensuite, par transitivité de $=$, $P\varphi = P'\rho$, d'où $\forall \Phi$ t.q. $\Phi \models P\varphi$ on a $u \succapprox_{gpo}^{\Phi} v$. De plus, $s = f(\dots, u, \dots)$ et $\forall \Phi$ on a $s \succ_{gpo}^{\Phi} u$ par la propriété de sous-terme de *gpo*. Enfin, par transitivité de *gpo*, on obtient que $\forall \Phi$ t.q. $\Phi \models P\varphi$ on a $s \succ_{gpo}^{\Phi} v$.

SUBTERM Extension On a $s = f(\dots, u, \dots)$. La seule transformation de S opérée par la règle est effectuée sur la substitution de P. Nous faisons cette preuve de correction en trois étapes: nous montrons d'abord qu'après la transformation de la substitution, les arcs couverts par $s \xrightarrow{P} v$ restent corrects. Ensuite, nous montrons que l'arc $s \xrightarrow{P} v$ lui-même reste correct, puis que les arcs au-dessus de $s \xrightarrow{P} v$ sont toujours corrects.

Soit $k \xrightarrow{Q} l$ un arc de $(G||\rho)$ couvert par $s \xrightarrow{P} v$. L'arc $(k \xrightarrow{Q} l||\rho)$ est correct puisque tout le graphe $(G||\rho)$ l'est. Soit $\rho = \sigma \cup \{P \mapsto \alpha\}$ et $\varphi = \sigma \cup \{P \mapsto \alpha \vee P'\}$ la Θ-substitution obtenue après application de la règle. Puisque $k \xrightarrow{Q} l$ est correct, on a: $\forall \Phi$ t.q. $\Phi \models Q\rho$ on a $k \succ_{gpo}^{\Phi} l$. Par le lemme 5.13, on obtient que $Q\varphi = Q\sigma = Q\rho$. Donc, $\forall \Phi$ t.q. $\Phi \models Q\varphi$, on a $\Phi \models Q\rho$, d'où $k \succ_{gpo}^{\Phi} l$. La preuve est similaire pour un arc $k \overset{Q}{\rightsquigarrow} l$.

Soit l'arc $s \xrightarrow{P} v$ lui-même. Soit $\rho = \sigma \cup \{P \mapsto \alpha\}$ et $\varphi = \sigma \cup \{P \mapsto \alpha \vee P'\}$ la Θ-substitution obtenue après l'application de la règle. On a $P\varphi = \alpha\varphi \vee P'\varphi$. Par définition de $\models$, $\Phi \models P\varphi$ si $\Phi \models \alpha\varphi$ ou $\Phi \models P'\varphi$. Puisque P n'apparaît pas dans $\alpha\sigma$ ni dans $P'\sigma$, on obtient $\alpha\varphi = \alpha\sigma$ et $P'\varphi = P'\sigma$. D'où $\Phi \models P\varphi$ si $\Phi \models \alpha\sigma$ ou $\Phi \models P'\sigma$. Maintenant,

- soit $\Phi \models \alpha\sigma$; comme par hypothèse le SOCS S est correct, $\forall \Phi$ t.q. $\Phi \models \alpha\rho$, on a $s \succ_{gpo}^{\Phi} v$. Par le lemme 5.13, on obtient que $\alpha\rho = \alpha\sigma$. D'où, $\forall \Phi$ t.q. $\phi \models \alpha\sigma$, on a $s \succ_{gpo}^{\Phi} v$.
- soit $\Phi \models P'\sigma$; par le lemme 5.13 on montre que $P'\rho = P'\sigma$. Puisque par hypothèse le SOCS S est correct, on a $\forall \Phi$ t.q. $\Phi \models P'\rho$: $u \succapprox_{gpo}^{\Phi} v$, qui induit $s \succ_{gpo}^{\Phi} v$. Puisque $P'\rho = P'\sigma$ on obtient que $\forall \Phi$ t.q. $\Phi \models P'\sigma$, on a $s \succ_{gpo}^{\Phi} v$.

Il reste toujours à montrer que les arcs au-dessus de $s \xrightarrow{P} v$ restent corrects en dépit de la transformation de ρ en φ. En effet, dans les arcs au-dessus de $s \xrightarrow{P} v$, on peut trouver des arcs d'ordre dont les $\mathcal{O}$-preuves incluent P, où φ associe maintenant une $\mathcal{O}$-preuve différente à P. Mais, dans le processus décrit par les règles de $\mathcal{C}$, la correction des arcs au-dessus de $s \xrightarrow{P} t$ est uniquement fondée sur la correction de $(s \xrightarrow{P} v \;\|\; \rho)$. Comme l'arc $(s \xrightarrow{P} v \;\|\; \varphi)$ est toujours correct, les arcs du dessus restent corrects.

SUBTERM Trivial Comme dans le cas de **SUBTERM First**, on a $s = f(\dots, u, \dots)$, et G' ne diffère de G que par un nouvel arc d'inégalité. L'application de la règle n'a pas modifié la Θ-substitution: $\rho = \varphi = \sigma$. En conséquence, tout arc de G reste correct dans G'. Tout ce que nous avons à prouver est la correction du nouvel arc. Par hypothèse, $u \xrightarrow{\top} v$ est correct. D'où $\forall \Phi$ t.q. $\Phi \models \top$, on a $u \succ_{gpo}^{\Phi} v$. Par la propriété de sous-terme, on obtient que

$\forall\Phi$ t.q. $\Phi \models \top$ on a $s \succ^{\Phi}_{gpo} v$.

SUBTERM Simplification Il y a deux transformations: P est remplacé par la $\mathcal{O}$-preuve triviale $\top$ et P est associé à $\top$ dans la Θ-substitution. Comme dans la preuve de **SUBTERM Extension**, tout arc couvert par $s \xrightarrow{\top} v$ reste correct dans S', grâce au lemme 5.13. La preuve que l'arc $s \xrightarrow{\top} v$ est correct est similaire à la preuve de la règle **SUBTERM First**, où les $\mathcal{O}$-preuve sont triviales. Le cas des arcs au-dessus de $s \xrightarrow{\top} v$ peut être traité comme le cas de la règle **SUBTERM Extension**: i.e. les arcs au dessus de $s \xrightarrow{\top} v$ sont corrects si $s \xrightarrow{\top} v$ reste correct.

THETA $>$ Supposons que $Term(\mathsf{F}) = s$, $Term(\mathsf{G}) = t = g(t_1, \dots, t_m)$, $Term(\mathsf{T}_1) = t_1, \dots$, et $Term(\mathsf{T}_m) = t_m$. On rappelle que, avant l'application de la règle, le graphe OCS est G et la Θ-substitution est $\rho = \sigma$, et après l'application, le graphe devient G' et la Θ-substitution devient $\varphi = \sigma \cup \{P \mapsto P_1 \wedge \dots \wedge P_m \wedge \Theta(s) >_{lex} \Theta(t)\}$. Chaque arc du graphe G est toujours présent dans G'. Pour chaque arc $k \xrightarrow{Q} l$ de G, puisque P est une nouvelle variable, nous avons $Q\varphi = Q\sigma$. Puisque $(G||\rho)$ est correct, $(k \xrightarrow{Q} l||\rho)$ est correct, et puisque $Q\varphi = Q\sigma$, on obtient que $(k \xrightarrow{Q} l||\varphi)$ est correct. Ainsi, tout arc de G reste correct dans G', et il reste uniquement à montrer la correction du nouvel arc lui-même. Comme les arcs $(\mathsf{F} \xrightarrow{P_1} \mathsf{T}_1||\varphi), \dots, (\mathsf{F} \xrightarrow{P_m} \mathsf{T}_m||\varphi)$ sont corrects, on obtient que:

- $\forall\Phi_1$ tel que $\Phi_1 \models P_1\varphi$ on a $s \succ^{\Phi_1}_{gpo} t_1$,
- ...
- $\forall\Phi_m$ tel que $\Phi_m \models P_m\varphi$ on a $s \succ^{\Phi_m}_{gpo} t_m$.

En conséquence, $\forall\Phi$ tel que $\Phi \models P_1\varphi$ et $\dots$ et $\Phi \models P_m\varphi$, on a $s \succ^{\Phi}_{gpo} t_1, \dots, s \succ^{\Phi}_{gpo} t_m$. Puisque, $\varphi = \sigma \cup \{P \mapsto P_1 \wedge \dots \wedge P_m \wedge \Theta(s) >_{lex} \Theta(t)\}$, on a $P\varphi = P_1\varphi \wedge \dots P_m\varphi \wedge \Theta(s) >_{lex} \Theta(t)$. Par définition de $\models$, on a $\forall\Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex})$ t.q. $\Phi \models \Theta(s) >_{lex} \Theta(t)$ alors $\mathcal{T}_{0,k}(s) >_{lex} \mathcal{T}_{0,k}(t)$. D'où, $\forall\Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex})$ t.q. $\Phi \models P\varphi$, on a $s \succ^{\Phi}_{gpo} t_1, \dots, s \succ^{\Phi}_{gpo} t_m$ et $\mathcal{T}_{0,k}(s) >_{lex} \mathcal{T}_{0,k}(t)$ et finalement, par définition de *gpo*, $\forall\Phi$ tel que $\Phi \models P\varphi$ on a $s \succ^{\Phi}_{gpo} t$.

THETA $>$ **Extension** La preuve pour le cas étendu est similaire au cas de la règle **THETA** $>$ case et pour les arcs au dessus, la preuve est similaire au cas de la règle **SUBTERM Extension**.

THETA $\sim$ La preuve est similaire au cas de la règles **THETA** $>$.

□

Ainsi, nous venons de montrer que si S est un SOCS correct et $S \vdash_{\mathcal{C}} S'$, alors S' est correct. Par induction sur le nombre d'applications de $\vdash_{\mathcal{C}}$, on obtient que si S est correct et $S \vdash^*_{\mathcal{C}} S''$ alors S'' est également correct. De plus, un SOCS initial est toujours correct vis-à-vis de *gpo*. D'où tout SOCS engendré par l'application des règles de déduction sur un SOCS initial est correct.

5.8.3 Complétude des règles de $\mathcal{C}$-déduction

Tout d'abord, nous rappelons le théorème à montrer.

Théorème 5.2 *Soit $(G||\sigma)$ un SOCS en $\mathcal{C}$-forme normale, dans lequel $G = (V, E)$. Pour tous noeuds $\mathsf{F}, \mathsf{G} \in V$, t.q. $Term(\mathsf{F}) = s$ et $Term(\mathsf{G}) = t$, où $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$:*

$$\forall \Phi = (\mathcal{T}_{0,k}, \succsim_{lex}) \text{ t.q. } s \succ^{\Phi}_{gpo} t, \text{ il existe un arc } s \xrightarrow{P} t \text{ dans } G \text{ t.q. } \Phi \models P\sigma$$

$$\forall \Phi = (\mathcal{T}_{0,k}, \succsim_{lex}) \text{ t.q. } s \approx^{\Phi}_{gpo} t, \text{ il existe un arc } s \overset{P}{\rightsquigarrow} t \text{ dans } G \text{ t.q. } \Phi \models P\sigma.$$

Preuve Nous montrons d'abord que si $s \succ^{\Phi}_{gpo} t$ (resp. $s \approx^{\Phi}_{gpo} t$) alors il existe nécessairement un arc $s \xrightarrow{P} t$ (resp. $s \overset{P}{\rightsquigarrow} t$). Supposer que $s \succ^{\Phi}_{gpo} t$ (resp. $s \approx^{\Phi}_{gpo} t$) et que l'application des règles s'est arrêtée sans engendrer un arc $s \xrightarrow{P} t$ (resp. $s \overset{P}{\rightsquigarrow} t$), conduit à une contradiction. Nous démontrons ceci par induction sur la taille des termes de chaque côté de l'inégalité (ou de l'égalité).

Supposons que $s = f(s_1, \ldots, s_n) \succ^{\Phi}_{gpo} g(t_1, \ldots, t_m) = t$ et qu'il n'y a aucun arc $s \xrightarrow{P} t$. Comme $s \succ^{\Phi}_{gpo} t$, par définition de *gpo*, on a soit:

- $\exists 1 \geq i \geq n$ tel que $s_i \succsim^{\Phi}_{gpo} t$. En appliquant l'hypothèse d'induction, on obtient qu'il existe un arc $s_i \xrightarrow{P_i} t$ ou $s_i \overset{P_i}{\rightsquigarrow} t$. Puisqu'il n'y a toujours aucun arc entre s et t, la règle de déduction **SUBTERM First** ou **SUBTERM Trivial** peut être appliquée. Ceci est une contradiction avec le fait que le SOCS S est en $\mathcal{C}$-forme normale.

- $s \succ^{\Phi}_{gpo} t_1, \ldots, t_m$ et $\Theta(s) >_{lex} \Theta(t)$. En appliquant l'hypothèse d'induction sur les m inégalités, on obtient qu'il existe m arcs $s \xrightarrow{P_i} t_i$. Comme de plus il n'existe pas d'arc entre entre s et t, la règle de déduction **THETA** $>$ peut être appliquée, ce qui contredit l'hypothèse que le SOCS S est en $\mathcal{C}$-forme normale.

De la même façon, si on suppose que $s = f(s_1, \ldots, s_n) \approx^{\Phi}_{gpo} g(t_1, \ldots, t_m) = t$ et qu'il n'existe pas d'arc $s \overset{P}{\rightsquigarrow} t$. Comme $s \approx^{\Phi}_{gpo} t$, par définition de *gpo*: $s \succ^{\Phi}_{gpo} t_1, \ldots, t_m$, $t \succ^{\Phi}_{gpo} s_1, \ldots, s_m$ et $\Theta(s) \sim \Theta(t)$. Par induction, on obtient qu'il existe des arcs $s \xrightarrow{P_i} t_i$ $(i = 1 \ldots m)$ et $t \xrightarrow{P'_j} s_j$ $(j = 1 \ldots n)$. Par conséquent, la règle **THETA** $\sim$ peut être appliquée, ce qui contredit le fait que S est en $\mathcal{C}$-forme normale.

Maintenant, nous montrons que pour toute instance Φ de *gpo* telle que $s \succ^{\Phi}_{gpo} t$ (resp. $s \approx^{\Phi}_{gpo} t$), si le SOCS $S = (G \parallel \sigma)$ est en $\mathcal{C}$-forme normale et G a un arc $s \xrightarrow{P} t$ (resp. $s \overset{P}{\rightsquigarrow} t$), alors $\Phi \models P\sigma$. Si $P = \top$, alors par définition de $\models$, on a $\Phi \models P$. Pour le cas où $P \neq \top$, on peut procéder par induction sur la taille des termes de chaque côté de l'inégalité (ou égalité). Supposons que $s = f(s_1, \ldots, s_n) \succ^{\Phi}_{gpo} g(t_1, \ldots, t_m) = t$. Soit $\Phi = (\mathcal{T}_{0,k}, \succsim_{lex})$. Par définition de *gpo*, soit:

1. $\exists 1 \geq i \geq n$ tel que $s_i \succsim^{\Phi}_{gpo} t$. Grâce à la première partie de la preuve, on sait qu'il existe nécessairement un arc $s \xrightarrow{P} t$, et soit un arc $s_i \xrightarrow{P_i} t$ soit $s_i \overset{P_i}{\rightsquigarrow} t$. Si $P_i = \top$ alors, puisque $P \neq \top$, on peut appliquer la règle **SUBTERM Simplification**, qui contredit le fait que S est en $\mathcal{C}$-forme normale. D'où $P_i \neq \top$. En appliquant l'hypothèse d'induction sur $s_i \succsim^{\Phi}_{gpo} t$, on obtient que $\Phi \models P_i\sigma$. Comme de plus $\{P \mapsto \alpha\} \in \sigma$, si $P_i \not\trianglelefteq \alpha$, alors on peut appliquer la règle **SUBTERM Extension** qui contredit le fait que S est en $\mathcal{C}$-forme normale. D'où $P_i \triangleleft \alpha$, et par le lemme 5.12, $P_i\sigma \triangleleft \alpha\sigma$. Par le lemme 5.11, on obtient que $\Phi \models \alpha\sigma$. Puisque $\{P \mapsto \alpha\} \in \sigma$, $P\sigma = \alpha\sigma$, et nous obtenons finalement que $\Phi \models P\sigma$.

2. $s \succ_{gpo}^{\Phi} t_1, \ldots, t_m$ et $\mathcal{T}_{0,k}(s) >_{lex} \mathcal{T}_{0,k}(t)$. Grâce à la première partie de la preuve, on déduit qu'il existe des arcs $s \xrightarrow{P_1} t_1, \ldots, s \xrightarrow{P_m} t_m$ et $s \xrightarrow{P} t$. En appliquant l'hypothèse d'induction aux m inégalités, on obtient que $\Phi \models P_1\sigma, \ldots, \Phi \models P_m\sigma$. Puisque $\mathcal{T}_{0,k}(s) >_{lex} \mathcal{T}_{0,k}(t)$ et $\Phi = (\mathcal{T}_{0,k}, \gtrapprox_{lex})$, par définition de $\models$, on déduit que $\Phi \models \Theta(s) >_{lex} \Theta(t)$. Comme dans le cas 1, si l'on tient compte du fait que $\{P \mapsto \alpha\} \in \sigma$ et si l'on suppose que la $\mathcal{O}$-preuve contient $P_1, \ldots, P_m$ et $\Theta(s) >_{lex} \Theta(t)$ ne sont pas visibles dans α, alors on obtient une contradiction, puisqu'on peut appliquer la règle **THETA** $>$ **Extension**. D'où, la $\mathcal{O}$-preuve contenant $P_1, \ldots, P_m$ et $\Theta(s) >_{lex} \Theta(t)$ est visible dans α. Finalement, comme dans le cas 1 et grâce au lemme 5.12 et au lemme 5.11, on obtient que $\Phi \models P\sigma$.

Pour le cas $s = f(s_1, \ldots, s_n) \approx_{gpo}^{\Phi} g(t_1, \ldots, t_m) = t$, la preuve est plus simple: $s \approx_{gpo}^{\Phi} t$ implique que $s \succ_{gpo}^{\Phi} t_1, \ldots, t_m$ et $t \succ_{gpo}^{\Phi} s_1, \ldots, s_n$ et $\Theta(s) \simeq_{lex} \Theta(t)$. Comme dans les cas précédents, il existe des arcs $s \xrightarrow{P_1} t_1, \ldots, s \xrightarrow{P_m} t_m$ et $t \xrightarrow{P'_1} s_1, \ldots, t \xrightarrow{P'_n} s_n$ et $s \xrightarrow{P} t$. En appliquant l'hypothèse d'induction sur les $m+n$ inégalités, on déduit que $\Phi \models P_1, \ldots, P_m$ et $\Phi \models P'_1, \ldots, P'_n$. Par hypothèse, on a $s \approx_{gpo}^{\Phi} t$, et par définition de *gpo*: $\mathcal{T}_{0,k}(s) \simeq_{lex} \mathcal{T}_{0,k}(t)$. Ainsi, par définition de $\models$, $\Phi \models \Theta(s) \simeq_{lex} \Theta(t)$. Finalement, $\Phi \models P\sigma$. $\square$

5.8.4 Complexité des règles de $\mathcal{C}$-déduction

Nous donnons une borne supérieure pour le nombre de déductions possible (i.e. du nombre de règles appliquées) sur un SOCS donné. Soit N le nombre maximal de noeuds dans le graphe OCS du SOCS, et M l'arité maximale des symboles de fonction de $\mathcal{F}$. Le nombre de couples de noeuds distincts est $N \times (N-1)$ (il faut considérer séparément les couples symétriques: (F, G) et (G, F)).

Nous donnons une borne supérieure pour le nombre d'applications de chaque règle de déduction sur un couple de noeuds. Soient F (étiqueté par f) et G (étiqueté par g) deux noeuds d'un graphe OCS G t.q. $Term(\mathsf{F}) = s = f(s_1, \ldots, s_n)$ et $Term(\mathsf{G}) = t = g(t_1, \ldots, t_m)$. Par cas sur les règles appliquées: **SUBTERM Property**, **SUBTERM First**, **SUBTERM Trivial**, **THETA** $>$ et **THETA** $\sim$ ne peuvent être appliquées que s'il n'existe pas d'arc entre F et G. Lorsqu'elles sont appliquées, elle engendrent un arc entre F et G. Comme aucune règle ne retire d'arc au graphe, ces règles ne peuvent être appliquées qu'une seule fois pour chaque couple de noeuds.

Les règles **SUBTERM Extension**, **SUBTERM Simplification** et **THETA** $>$ **Extension** peuvent être appliquées même s'il existe déjà un arc entre F et G. Nous prouvons maintenant que le nombre d'application est borné.

Remarquons d'abord que dans un SOCS $S = (G||\rho)$ avec $\rho = \sigma \cup \{P \mapsto \alpha\}$ tel qu'il existe une $\mathcal{O}$-preuve β avec $\beta \lessdot \alpha$, alors quelle que soit la règle de déduction appliquée au SOCS S par la suite, soit:

- P est remplacée par $\top$, soit
- le SOCS devient $S' = (G'||\rho')$ où $\rho' = \sigma' \cup \{P \mapsto \alpha'\})$ et $\beta \lessdot \alpha'$.

La précondition de **THETA** $>$ **Extension** vérifie que $(P_1 \wedge \ldots \wedge P_m \wedge \Theta(s) >_{lex} \Theta(t)) \not\lessdot \alpha$. Après l'application de cette règle, on a $P_1 \wedge \ldots \wedge P_m \wedge \Theta(s) >_{lex} \Theta(t) \lessdot \alpha'$ où α' est la nouvelle $\mathcal{O}$-preuve de l'arc. Ainsi, cette règle ne peut être appliquée qu'une seule fois pour chaque couple de noeuds F et G.

La précondition de **SUBTERM Simplification** est similaire: puisque P doit être différent de $\top$, cette règle ne peut être appliquée qu'une seule fois.

La précondition $P' \not\prec \alpha$ de **SUBTERM Extension** empêche d'ajouter une $\mathcal{O}$-preuve P' à P deux fois. Par conséquent, il est suffisant de montrer que le nombre d'$\mathcal{O}$-preuves éventuellement ajoutées par la règle **SUBTERM Extension** est borné. Comme l'arité maximale des symboles de fonction de $\mathcal{F}$ est M, on ne pourra ajouter au maximum que M $\mathcal{O}$-preuves venant d'arcs $s_i \overset{P_i}{\mathrel{-\!\!\gg\!\!-}} t$ et M $\mathcal{O}$-preuves venant d'arcs $s_i \overset{P'_i}{\mathrel{-\!\!\sim\!\!-}} t$. Ainsi, le nombre d'$\mathcal{O}$-preuves ajoutées est borné par $2M$.

Si l'on ajoute les nombres de $\mathcal{O}$-preuves ajoutées par chacune des règles sur le graphe OCS, on obtient:

- pour les arcs $s \mathrel{-\!\!\gg\!\!-} t$, il y a M $\mathcal{O}$-preuves possibles construites par les règles **SUBTERM** à partir d'arcs de la forme $s_i \mathrel{-\!\!\gg\!\!-} t$. Il y a également M $\mathcal{O}$-preuves possibles construites à partir des arcs $s_i \mathrel{-\!\!\sim\!\!-} t$. Finalement, si on ajoute une $\mathcal{O}$-preuve construite par une règle **THETA**, on obtient un nombre de $2M + 1$ $\mathcal{O}$-preuves construites.

- pour les arcs $s \mathrel{-\!\!\sim\!\!-} t$, on peut construire une $\mathcal{O}$-preuve unique grâce à la règle **THETA** $\sim$.

Donc, pour tout couple de noeuds du graphe, le nombre maximal de règles appliquées est $2M+2$. En conséquence, pour le graphe OCS complet, la borne supérieure pour le nombre de déductions est:

$$N \times (N - 1) \times (2M + 2).$$

Puisque la construction des $\mathcal{O}$-preuve est linéaire sur M en espace (pour les règles **THETA** $>$, **THETA** $>$ **Extension**, **THETA** $\sim$) ou constant (pour les autres règles), la complexité totale *en espace* est polynomiale en fonction de M et N: $O(N^2M^2)$.

Pour calculer la complexité en temps, il est nécessaire d'étudier en plus d'autres aspects du processus de déduction. Tout d'abord, un facteur important est le temps nécessaire pour savoir si une règle de déduction s'applique ou non. C'est pour la règle **THETA** $\sim$ que ce coût est le plus important: vérifier qu'elle s'applique revient à vérifier qu'il existe des arcs entre le symbole f au sommet du terme et chacun des sous-termes $t_1, \ldots, t_m$ du terme t, et vice versa avec g et les sous-termes $s_1, \ldots, s_n$ de s. Si on suppose que le graphe est stocké dans un tableau, l'existence d'un arc peut être testée en temps constant. Ainsi, savoir si l'on peut appliquer la règle **THETA** $\sim$ est linéaire sur M. L'application de la règle consiste à éventuellement ajouter un nouvel arc et ajouter ou modifier une $\mathcal{O}$-preuve. Là encore, c'est la règle **THETA** $\sim$ qui a le coût d'application le plus élevé: il est nécessaire d'ajouter un nouvel arc étiqueté par une nouvelle variable d'$\mathcal{O}$-preuve et de construire une nouvelle $\mathcal{O}$-preuve dans la Θ-substitution qui est une conjonction des variables $P_1, \ldots, P_m$ et $P'_1, \ldots, P'_n$. La construction du nouvel arc dans le tableau représentant le graphe peut être effectuée en temps constant et la construction de la conjonction d'au plus $2M$ variables est linéaire en M.

Par conséquent, pour tout couple de noeuds, tester si une règle s'applique et l'appliquer (i.e. construire la $\mathcal{O}$-preuve) peut être effectué en temps linéaire: $O(M)$. La complexité d'un processus essayant d'appliquer et appliquant les 8 règles de déduction sur un couple de noeuds est donc également en $O(M)$. Or, on a vu que le nombre de couples de noeuds à considérer est $N \times (N - 1)$. D'où, un processus appliquant toutes les règles de déduction une fois sur tous les couples de noeuds a une complexité en $O(N^2M)$. Or, dans le pire des cas, tous les couples de noeuds du graphe sont passés en revue, et une seule règle est appliquée sur un seul couple de noeuds. Plus haut, nous avons vu qu'au plus $N \times (N - 1) \times (2M + 2)$ règles peuvent être appliquées. Donc, la complexité global en temps du processus est en $O(N^4M^2)$.

Il faut remarquer que ce dernier résultat est une borne supérieure pour une stratégie naïve d'application des règles. Il est facile d'obtenir de meilleurs résultats en pratique en utilisant des stratégies adaptées (voir section 5.3).

5.8.5 Extension multi-ensemble

Dans cette partie, nous montrons que pour tout préordre $\geq$ sur un ensemble E, $\geq^M$ est un préordre sur $M(E)$, $>^M$ est un ordre bien fondé sur $M(E)$, $\geq^M$ est une extension monotone de $\geq$. Pour mener à bien cette preuve, nous allons utiliser une correspondance entre l'ordre $>^M$ sur $M(E)$ et l'ordre $>^{mul}$ (définition 4.2) sur $M(\overline{E})$, où $\overline{E}$ est l'ensemble des représentant canoniques des classes d'équivalence des éléments de E modulo la relation $\simeq$ associée à $\geq$. Tout d'abord, nous définissons la notation de représentant canonique.

Définition 5.24 *Soit $\geq$ un préordre sur un ensemble E et* `eq(()` $\geq$`)` $=\simeq$. *On note $\overline{t}$ le* représentant canonique *de $\langle t\rangle$, où $\langle t\rangle$ est la classe d'équivalence de t, i.e. $\langle t\rangle = \{s \in E \mid s \simeq t\}$. On note $\overline{E}$ l'ensemble des représentants canoniques des éléments de E, i.e. $\overline{E} = \{\overline{t} \mid t \in E\}$. De la même façon, pour tout multi-ensemble $X = \{s_1, \ldots, s_n\} \in M(E)$ on note $\overline{X}$ le multi-ensemble $\{\!\{\overline{s_1}, \ldots, \overline{s_n}\}\!\} \in M(\overline{E})$.*

Maintenant, nous montrons le lemme essentiel utilisé dans les preuves.

Lemme 5.14 *Soit $\geq$ un préordre sur un ensemble E, l'ordre strict* `ord(` $\geq$ `)` *noté $>$, la relation d'équivalence* `eq(` $\geq$ `)` *notée $\simeq$, $X, Y \in M(E)$. Le préordre $\geq^M$ sur $M(E)$ et l'ordre $>^{mul}$ sur $M(\overline{E})$ vérifient les propriétés suivantes:*

$$X >^M Y \iff \overline{X} >^{mul} \overline{Y}$$
$$X \simeq^M Y \iff \overline{X} = \overline{Y}$$

Preuve Nous montrons d'abord la deuxième propriété.

$$\begin{array}{lll}
X \simeq^M Y & \iff & X = \{\!\{x_1, \ldots, x_n\}\!\} \simeq^{mul} \{\!\{y_1, \ldots, y_n\}\!\} = Y \\
& & \text{car } \forall i = 1 \ldots n : x_i \simeq y_i, \\
& & \text{et par définition de } \simeq^{mul}. \\
& \Longrightarrow & \overline{x_i} = \overline{y_i} \text{ pour tout } i = 1 \ldots n \text{ d'après la définition 5.24} \\
& \iff & \overline{X} = \{\!\{\overline{x_1}, \ldots, \overline{x_n}\}\!\} = \{\!\{\overline{y_1}, \ldots, \overline{y_n}\}\!\} = \overline{Y}
\end{array}$$

De la même façon, on a:

$$\begin{array}{lll}
\overline{X} = \overline{Y} & \iff & \overline{X} = \{\!\{\overline{x_1}, \ldots, \overline{x_n}\}\!\} = \{\!\{\overline{y_1}, \ldots, \overline{y_n}\}\!\} = \overline{Y} \\
& \iff & \overline{x_i} = \overline{y_i} \text{ pour tout } i = 1 \ldots n \\
& \Longrightarrow & x_i \simeq y_i \text{ pour tout } i = 1 \ldots n \text{ puisque } x_i \text{ et } y_i \text{ sont des éléments} \\
& & \text{de la même classe d'équivalence: celle dont } \overline{x_i} \text{ (ou } \overline{y_i}) \\
& & \text{est le représentant canonique.} \\
& \iff & X \simeq^M Y \text{ par définition de } \simeq^M.
\end{array}$$

Maintenant, nous démontrons la première propriété, i.e. $X >^M Y \iff \overline{X} >^{mul} \overline{Y}$ en deux temps. Dans la suite de la preuve, nous dirons qu'un multi-ensemble X *domine* un multi-ensemble Y si $\forall y \in Y$, $\exists x \in X$ tel que $x > y$ (ceci correspond, en fait, à la troisième condition de la définition de $>^{mul}$). Tout d'abord, montrons que $X >^M Y \implies \overline{X} >^{mul} \overline{Y}$. On suppose que l'on a $X >^M Y$ et $X_0, Y_0 \in M(E)$ tels que $X = X_0 \cup (X \cap Y)$ et $Y_0 = Y_0 \cup (X \cap Y)$. Tout d'abord, on peut remarquer que si $\overline{X_0} >^{mul} \overline{Y_0}$ alors on a nécessairement $\overline{X} = (\overline{X_0} \cup \overline{X \cap Y}) >^{mul} (\overline{Y_0} \cup \overline{X \cap Y}) = \overline{Y}$. En conséquence, on montre que $\overline{X_0} >^{mul} \overline{Y_0}$. Procédons par cas sur la définition de $>^M$ et par induction sur la taille de X_0 et de Y_0.

1. Si $Y_0 = \emptyset$ et $X_0 \neq \emptyset$, alors on a trivialement $\overline{X_0} >^{mul} \overline{Y_0} = \emptyset$.

2. Si $X_0 = \{\!\{a\}\!\} \cup X'_0$, $Y_0 = \{\!\{b_1, \ldots, b_m\}\!\} \cup Y'_0$, $a > b_i$ pour tout $i = 1 \ldots m$, et $X'_0 \geq^M Y'_0$, alors on a
$\overline{X_0} = \{\!\{\overline{a}\}\!\} \cup \overline{X'_0}$,
$\overline{Y_0} = \{\!\{\overline{b_1}, \ldots, \overline{b_m}\}\!\} \cup \overline{Y'_0}$, et
$\overline{a} > \overline{b_i}$ pour tout $i = 1 \ldots m$ par transitivité du préordre et antisymétrie de l'ordre strict.

   D'autre part, comme on sait que $\geq^M$ est l'union des relations $>^M$ et $\simeq^M$,

   - Soit $X'_0 >^M Y'_0$ et en appliquant l'hypothèse d'induction, on obtient $\overline{X'_0} >^{mul} \overline{Y'_0}$. Par définition de $>^{mul}$, il existe deux ensembles $\overline{X'_1}$ et $\overline{Y'_1}$ tels que:
     - $\overline{X'_1} \neq \emptyset$ et $\overline{X'_1} \subset \overline{X'_0}$,
     - $\overline{Y'_0} = (\overline{X'_0} \setminus \overline{X'_1}) \cup \overline{Y'_1}$,
     - $\overline{X'_1}$ domine $\overline{Y'_1}$, i.e. $\forall y \in \overline{Y'_1}$, $\exists x \in \overline{X'_1}$ tel que $x > y$.

     Or, on a
     $\overline{X_0} = \overline{X'_1} \cup (\overline{X'_0} \cap \overline{Y'_0}) \cup \{\!\{\overline{a}\}\!\}$ et
     $\overline{Y_0} = \overline{Y'_1} \cup (\overline{X'_0} \cap \overline{Y'_0}) \cup \{\!\{\overline{b_1}, \ldots, \overline{b_n}\}\!\}$.
     Définissons deux nouveaux multi-ensembles
     $\overline{X''_0} = \overline{X'_1} \cup \{\!\{\overline{a}\}\!\}$ et
     $\overline{Y''_0} = \overline{Y'_1} \cup \{\!\{\overline{b_1}, \ldots, \overline{b_n}\}\!\}$.
     Or, les multi-ensembles $\overline{X_0}, \overline{Y_0}$ et $\overline{X''_0}, \overline{Y''_0}$ vérifient toutes les conditions de la définition de $>^{mul}$:
     - $\overline{X''_0} \neq \emptyset$ et $\overline{X''_0} \subseteq \overline{X_0}$,
     - $\overline{Y_0} = (\overline{X_0} \setminus \overline{X''_0}) \cup \overline{Y''_0}$,
     - $\overline{X''_0}$ domine $\overline{X''_0}$. En effet, $\overline{Y''_0} = \overline{Y'_1} \cup \{\!\{\overline{b_1}, \ldots, \overline{b_n}\}\!\}$, $\overline{X'_1}$ domine $\overline{Y'_1}$ et $\{\!\{\overline{a}\}\!\}$ domine $\{\!\{\overline{b_1}, \ldots, \overline{b_n}\}\!\}$.

     En conséquence, de la définition de $>^{mul}$ on déduit $\overline{X_0} >^{mul} \overline{Y_0}$.
   - Soit $X'_0 \simeq^M Y'_0$. On a montré que cela implique nécessairement $\overline{X'_0} = \overline{Y'_0}$. Dans ce cas, il est possible d'appliquer directement la définition de $>^{mul}$ aux ensembles $\overline{X_0}$, $\overline{Y_0}$, $\overline{X'_0} = \{\!\{\overline{a}\}\!\}$ et $\overline{Y'_0} = \{\!\{\overline{b_1}, \ldots, \overline{b_m}\}\!\}$:
     - $\overline{X'_0} \neq \emptyset$ et $\overline{X'_0} \subseteq \overline{X_0}$,
     - $\overline{Y_0} = (\overline{X_0} \setminus \overline{X'_0}) \cup \overline{Y'_0}$,
     - $\overline{X'_0}$ domine $\overline{Y'_0}$, puisque $\overline{a} > \overline{b_i}$ pour tout $i = 1 \ldots n$.

     D'où $\overline{X_0} >^{mul} \overline{Y_0}$.

3. Si $X_0 = \{\!\{a\}\!\} \cup X'_0$, $Y_0 = \{\!\{b\}\!\} \cup Y'_0$, $a \simeq b$, et $X'_0 >^M Y'_0$, alors on a
$\overline{X_0} = \{\!\{\overline{a}\}\!\} \cup \overline{X'_0}$,
$\overline{Y_0} = \{\!\{\overline{b}\}\!\} \cup \overline{Y'_0}$, et
$\overline{X'_0} >^{mul} \overline{Y'_0}$ en appliquant l'hypothèse d'induction à $X'_0 >^M Y'_0$.

   La preuve est plus simple que dans le cas précédent, puisque de la même façon, $\overline{X'_0} >^{mul} \overline{Y'_0}$ implique l'existence de deux ensembles $\overline{X'_1}$ et $\overline{Y'_1}$ tels que:

   - $\overline{X'_1} \neq \emptyset$ et $\overline{X'_1} \subset \overline{X'_0}$,
   - $\overline{Y'_0} = (\overline{X'_0} \setminus \overline{X'_1}) \cup \overline{Y'_1}$,

– $\overline{X'_1}$ domine $\overline{Y'_1}$.

Or, les multi-ensembles $\overline{X_0}, \overline{Y_0}$ et $\overline{X'_1}, \overline{Y'_1}$, vérifient toutes les conditions de la définition de $>^{mul}$:

– $\overline{X'_1} \neq \emptyset$ et $\overline{X'_1} \subseteq \overline{X_0}$,
– $\overline{Y_0} = (\overline{X_0} \setminus \overline{X'_1}) \cup \overline{Y'_1}$,
– $\overline{X'_1}$ domine $\overline{Y'_1}$.

D'où $\overline{X_0} >^{mul} \overline{Y_0}$.

Maintenant, nous démontrons à l'inverse que $\overline{X} >^{mul} \overline{Y} \implies X >^M Y$, par induction sur le nombre d'élément de X et de Y. Si l'on suppose que $\overline{X} >^{mul} \overline{Y}$, de la définition de $>^{mul}$, on peut déduire qu'il existe deux multi-ensembles $\overline{X_1}$ et $\overline{Y_1}$ tels que

– $\overline{X_1} \neq \emptyset$, $\overline{X_1} \subseteq \overline{X}$,

– $\overline{Y} = (\overline{X} \setminus \overline{X_1}) \cup \overline{Y_1}$,

– $\overline{X_1}$ domine $\overline{Y_1}$.

Soient $\overline{X_0} = \overline{X} \setminus (\overline{X} \cap \overline{Y})$, et $\overline{Y_0} = \overline{Y} \setminus (\overline{X} \cap \overline{Y})$. On a
$\overline{X_1} = \overline{X_0} \cup \overline{B}$ et
$\overline{Y_1} = \overline{Y_0} \cup \overline{B}$
où B peut être vide. Par hypothèse $\overline{X_1}$ domine $\overline{Y_1}$, donc $\overline{X_0}$ domine $\overline{Y_0}$. Soit $\overline{a} \in \overline{X_0}$ et $\overline{b_1}, \ldots, \overline{b_n}$ tous les éléments majorés par $\overline{a}$. Dans ce cas, on a:

– $\overline{X_0} = \{\!\{\overline{a}\}\!\} \cup \overline{X'_0}$,

– $\overline{Y_0} = \{\!\{\overline{b_1}, \ldots, \overline{b_n}\}\!\} \cup \overline{Y'_0}$,

– $\overline{a} > \overline{b_i}$ pour tout $i = 1 \ldots n$,

– $\overline{X'_0}$ domine $\overline{Y'_0}$.

Or, on a $X_0 = \{\!\{a\}\!\} \cup X'_0$, $Y_0 = \{\!\{b_1, \ldots, b_n\}\!\} \cup Y'_0$, et $a > b_i$ pour tout $i = 1 \ldots n$. De plus, comme $\overline{X'_0}$ domine $\overline{Y'_0}$ et $\overline{X'_0} \cap \overline{Y'_0} = \emptyset$, par définition de $>^{mul}$, on a $\overline{X'_0} >^{mul} \overline{Y'_0}$. De plus, comme $\overline{X'_0} \subset \overline{X_0} \subseteq \overline{X}$ et $\overline{Y'_0} \subset \overline{Y_0} \subseteq \overline{Y}$, en appliquant l'hypothèse d'induction, on a:

$$\overline{X'_0} >^{mul} \overline{Y'_0} \implies X'_0 >^M Y'_0$$

Par le cas 2. de la définition de $>^M$, on déduit

$$X_0 = (\{\!\{a\}\!\} \cup X'_0) >^M (\{\!\{b_1, \ldots, b_n\}\!\} \cup Y'_0) = Y_0$$

et enfin par le cas 3. de la définition de $>^M$:

$$X = (X_0 \cup (X \cap Y)) >^M (Y_0 \cup (X \cap Y)) = Y.$$

□

Les deux lemmes suivant permettent d'établir que $\geq^M$ est un préordre sur $M(E)$.

Lemme 5.15 *La relation $\geq^M$ est réflexive: $\forall X \in M(E) : X \geq^M X$.*

Preuve Évidente d'après la définition de $\simeq^M$. $\square$

Lemme 5.16 *La relation $\geq^M$ est transitive:* $\forall X, Y, Z \in M(E)$:

$$\begin{array}{lcl} X >^M Y \textit{ et } Y >^M Z & \Longrightarrow & X >^M Z \\ X >^M Y \textit{ et } Y \simeq^M Z & \Longrightarrow & X >^M Z \\ X \simeq^M Y \textit{ et } Y >^M Z & \Longrightarrow & X >^M Z \\ X \simeq^M Y \textit{ et } Y \simeq^M Z & \Longrightarrow & X \simeq^M Z \end{array}$$

Preuve Pour chacune de ces implications on procède de la même façon en utilisant le lemme 5.14.

$$\begin{array}{lcl} X >^M Y \text{ et } Y >^M Z & \Longleftrightarrow & \overline{X} >^{mul} \overline{Y} \text{ et } \overline{Y} >^{mul} \overline{Z} \\ & & \text{par le lemme 5.14} \\ & \Longrightarrow & \overline{X} >^{mul} \overline{Z} \text{ par la transitivité de } >^{mul} \\ & \Longrightarrow & X >^M Z \text{ (lemme 5.14).} \end{array}$$

$$\begin{array}{lcl} X >^M Y \text{ et } Y \simeq^M Z & \Longleftrightarrow & \overline{X} >^{mul} \overline{Y} \text{ et } \overline{Y} = \overline{Z} \\ & & \text{par le lemme 5.14} \\ & \Longrightarrow & \overline{X} >^{mul} \overline{Z} \text{ par la transitivité de } >^{mul} \\ & \Longrightarrow & X >^M Z \text{ (lemme 5.14).} \end{array}$$

$$\begin{array}{lcl} X \simeq^M Y \text{ et } Y >^M Z & \Longleftrightarrow & \overline{X} = \overline{Y} \text{ et } \overline{Y} >^{mul} \overline{Z} \\ & & \text{par le lemme 5.14} \\ & \Longrightarrow & \overline{X} >^{mul} \overline{Z} \text{ par la transitivité de } >^{mul} \\ & \Longrightarrow & X >^M Z \text{ (lemme 5.14).} \end{array}$$

$$\begin{array}{lcl} X \simeq^M Y \text{ et } Y \simeq^M Z & \Longleftrightarrow & \overline{X} = \overline{Y} \text{ et } \overline{Y} = \overline{Z} \\ & & \text{par le lemme 5.14} \\ & \Longrightarrow & \overline{X} = \overline{Z} \\ & \Longrightarrow & X \simeq^M Z \text{ (lemme 5.14).} \end{array}$$

$\square$

Lemme 5.17 *L'ordre $>^M$ est bien fondé.*

Preuve Supposons qu'il existe une chaîne infinie décroissante pour $>^M$:

$X_1 >^M X_2 >^M \ldots$

Alors d'après le lemme 5.14, on obtient une chaîne infinie décroissante pour $>^{mul}$:

$\overline{X_1} >^{mul} \overline{X_2} >^{mul} \ldots$

ce qui contredit la bonne fondation de $>^{mul}$, d'où $>^M$ est bien fondé. $\square$

Lemme 5.18 *L'ordre $>^M$ est une extension monotone de $>$:*

$$\succ \supseteq > \implies \succ^M \supseteq >^M$$

Preuve Soient $>$ et $\succ$ deux ordres tels que $\succ \supseteq >$. Pour tout $X, Y \in M(E)$ tels que $X >^M Y$, par le lemme 5.14, on obtient que $\overline{X} >^{mul} \overline{Y}$. Or, l'ordre $>^{mul}$ est une extension monotone de $>$ (voir [JL82]), d'où $\overline{X} \succ^{mul} \overline{Y}$. Enfin en réappliquant le lemme 5.14, on obtient $X \succ^M Y$. $\square$

5.8.6 Preuve de la proposition 5.2

Tout d'abord, nous rappelons la proposition à prouver.

Proposition 5.2 *Soit $\geq$ un préordre sur un ensemble E, tel que $\geq = > \cup \simeq$. Pour tout $X_1, Y_1, X_2, Y_2 \in M(E)$ tels que $(X_1 \cup X_2) \cap (Y_1 \cup Y_2) = \emptyset$, on a*

$$\begin{array}{lcl} X_1 >^M Y_1 \textit{ et } X_2 \geq^M Y_2 & \Longrightarrow & (X_1 \cup X_2) >^M (Y_1 \cup Y_2) \\ X_1 \simeq^M Y_1 \textit{ et } X_2 \simeq^M Y_2 & \Longrightarrow & (X_1 \cup X_2) \simeq^M (Y_1 \cup Y_2). \end{array}$$

On procède par induction. Pour le cas de base, si $X_2 = \emptyset$ alors on a nécessairement $X_2 = Y_2 = \emptyset$. D'où $X_1 \cup X_2 = X_1$ et $Y_1 \cup Y_2 = Y_1$, et selon le cas:
$X_1 >^M Y_1 \implies (X_1 \cup X_2) >^M (Y_1 \cup Y_2)$, ou
$X_1 \simeq^M Y_1 \implies (X_1 \cup X_2) \simeq^M (Y_1 \cup Y_2)$.

Maintenant, nous nous intéressons au cas général de cette preuve par induction sur le nombre d'éléments de X_2, en supposant que $X_2 \neq \emptyset$. Il faut noter que de $(X_1 \cup X_2) \cap (Y_1 \cup Y_2) = \emptyset$, on peut déduire que les intersections $X_1 \cap Y_1$ et $X_2 \cap Y_2$ sont vides. Tout d'abord nous montrons que:

$$X_1 >^M Y_1 \text{ et } X_2 \geq^M Y_2 \implies (X_1 \cup X_2) >^M (Y_1 \cup Y_2).$$

On procède par cas sur la définition de $X_2 >^M Y_2$, puis par cas sur la définition de $X_2 \simeq^M Y_2$.

- Si $X_2 >^M Y_2$ alors par cas sur la définition de $>^M$, on a:
  - si $Y_2 = \emptyset$, alors $Y_1 \cup Y_2 = Y_1$ et $X_1 >^{mul} Y_1 \cup Y_2$. Or, comme $X_2 >^M Y_2$, on sait que $X_2 \neq \emptyset$ et l'on peut appliquer la proposition 5.1 qui nous donne que $X_1 \cup X_2 >^{mul} Y_1 \cup Y_2$,
  - si $X_2 = \{\!\{a\}\!\} \cup X'_2$, et $Y_2 = \{\!\{b_1, \ldots, b_m\}\!\} \cup Y'_2$, avec $X'_2, Y'_2 \in M(E)$, $X'_2 \geq^M Y'_2$ et $\forall i = 1, \ldots, m : a > b_i$. En appliquant l'hypothèse d'induction à $X_1 >^M Y_1$ et $X'_2 \geq^M Y'_2$, on obtient

    $(X_1 \cup X'_2) >^M (Y_1 \cup Y'_2)$.

    Enfin, en appliquant le cas 2. de la définition de $>^M$, on obtient

    $(\{\!\{a\}\!\} \cup X_1 \cup X'_2) >^M (\{\!\{b_1, \ldots, b_m\}\!\} \cup Y_1 \cup Y'_2)$

    d'où $(X_1 \cup X_2) >^{mul} (Y_1 \cup Y_2)$.
  - si $X_2 = \{\!\{a\}\!\} \cup X'_2$, et $Y_2 = \{\!\{b\}\!\} \cup Y'_2$, avec $X'_2, Y'_2 \in M(E)$, $X'_2 >^M Y'_2$ et $a \simeq b$, alors, comme dans le cas précédent, on applique l'hypothèse d'induction et l'on obtient facilement $(X_1 \cup X'_2) >^{mul} (Y_1 \cup Y'_2)$. En appliquant le cas 3. de la définition de $>^M$, on obtient

    $(\{\!\{a\}\!\} \cup X_1 \cup X'_2) >^M (\{\!\{b\}\!\} \cup Y_1 \cup Y'_2)$.
- Si $X_2 \simeq^M Y_2$, on procède par cas sur la définition de $\simeq^M$. Comme on ne peut avoir $X_2 = Y_2$ (puisque $X_2 \neq \emptyset$ et $X_2 \cap Y_2 = \emptyset$), on a nécessairement $X_2 = \{\!\{a\}\!\} \cup X'_2$, et $Y_2 = \{\!\{b\}\!\} \cup Y'_2$, avec $X'_2, Y'_2 \in M(E)$, $X'_2 \simeq^M Y'_2$ et $a \simeq b$. En appliquant l'hypothèse d'induction on obtient $(X_1 \cup X'_2) >^{mul} (Y_1 \cup Y'_2)$. En appliquant de nouveau le cas 3. de la définition de $>^M$, on obtient

  $(\{\!\{a\}\!\} \cup X_1 \cup X'_2) >^M (\{\!\{b\}\!\} \cup Y_1 \cup Y'_2)$

  d'où le résultat.

Maintenant, montrons que

$$X_1 \simeq^M Y_1 \text{ et } X_2 \simeq^M Y_2 \implies (X_1 \cup X_2) \simeq^M (Y_1 \cup Y_2)$$

en procédant par cas sur la définition de $\simeq^{mul}$. Nous savons que $X_2 \neq Y_2$ pour les mêmes raisons que précédemment ($X_2 \neq \emptyset$ et $X_2 \cap Y_2 = \emptyset$). Donc, seul le cas 2. de la définition de $\simeq^M$ peut s'appliquer. On a donc $X_2 = \{\!\{a\}\!\} \cup X_2'$, et $Y_2 = \{\!\{b\}\!\} \cup Y_2'$, avec $X_2', Y_2' \in M(E)$, $X_2' \simeq^M Y_2'$ et $a \simeq b$. En appliquant l'hypothèse d'induction à $X_1 \simeq^M Y_1$ et $X_2' \simeq^M Y_2'$, on obtient $(X_1 \cup X_2') \simeq^{mul} (Y_1 \cup Y_2')$. En appliquant le cas 2. de la définition de $\simeq^{mul}$ on a

$(\{\!\{a\}\!\} \cup X_1 \cup X_2') \simeq^M (\{\!\{b\}\!\} \cup Y_1 \cup Y_2')$

d'où le résultat. □

5.8.7 Correction des règles de $\mathcal{C}_M$-déduction

Tout d'abord, rappelons le théorème de correction.

Théorème 5.5 *(Correction) Pour tout SOCS S correct vis-à-vis de gpo, si $S \vdash^*_{\mathcal{C}_M} S'$ alors S' est correct vis-à-vis de l'extension multi-ensemble de gpo.*

Le principe général de la preuve est très similaire à la preuve de correction des règles de $\mathcal{C}$-déduction. Cependant, nous donnons ici un schéma général de la preuve avec toutes ses spécificités. Dans la suite, s'il n'y a pas d'ambiguïté possible, nous abrégeons "correct vis-à-vis de l'extension multi-ensemble de *gpo*" par "correct".

Tout d'abord, il faut remarquer que dans l'algorithme d'application des règles de $\mathcal{C}_M$-déduction donné section 5.6.2, il est supposé que les deux multi-ensembles à comparer X et Y sont tels que $X \cap Y = \emptyset$. Il en est donc de même pour tout multi-ensemble $X' \subseteq X$ et $Y' \subseteq Y$, i.e. $X' \cap Y' = \emptyset$.

Le premier point à montrer est qu'un SOCS S correct vis-à-vis de *gpo* est correct vis-à-vis de l'extension multi-ensemble du *gpo*. Initialement, il n'existe aucun arc entre les nœuds multi-ensemble. Les seuls arcs d'ordre éventuellement présents sont des arcs entre les termes, où, rappelons-le, chaque terme t représente également le multi-ensemble $\{\!\{t\}\!\}$. Cependant, tout arc $s \xrightarrow{\succ P} t$ (resp. $s \xrightarrow{\sim P} t$) du graphe est correct, i.e. pour toute instance Φ de *gpo* telle que $\Phi \models P\sigma$ on a $s \succ^{\Phi}_{gpo} t$ (resp. $s \approx^{\Phi}_{gpo} t$). Par la définition 5.19, on obtient que $\{\!\{s\}\!\} \succ^{\Phi,M}_{gpo} \{\!\{t\}\!\}$ (resp. $\{\!\{s\}\!\} \approx^{\Phi,M}_{gpo} \{\!\{t\}\!\}$. Donc tout SOCS correct vis-à-vis de *gpo* est correct vis-à-vis de l'extension multi-ensemble de *gpo*.

Maintenant, montrons que pour tout SOCS S, S' tel que $S \vdash_{\mathcal{C}_M} S'$, si S est correct alors il en est de même pour S'. Ceci revient à montrer que l'application de toute règle de l'ensemble $\mathcal{C}_M$ sur un SOCS correct donne un SOCS correct. Les seules modifications pouvant être apportées par ces règles sont:

- l'ajout d'un nouvel arc d'ordre $\mathsf{F} \xrightarrow{\succ P} \mathsf{G}$ (resp. $\mathsf{F} \xrightarrow{\succ P} \mathsf{G}$) entre deux sommets étiquetés par des symboles multi-ensemble M, par une des règles **SINGLETON**, **SUBTERM Multiset**, **COMMON** $>$, ou **COMMON** $\sim$, ou
- l'ajout d'un nouvel arc d'ordre trivial $\mathsf{F} \xrightarrow{\succ P} \mathsf{G}$ (resp. $\mathsf{F} \xrightarrow{\succ P} \mathsf{G}$) entre deux sommets étiquetés par des symboles multi-ensemble M, par une des règles **SINGLETON** Simplification, **SUBTERM Multiset** Simplification, **COMMON** $>$ Simplification, ou
- la modification de la substitution associant une $\mathcal{O}$-preuve à une variable d'$\mathcal{O}$-preuve P d'un arc $\mathsf{F} \xrightarrow{\succ P} \mathsf{G}$ (resp. $\mathsf{F} \xrightarrow{\succ P} \mathsf{G}$) existant, par une des règles parmi **SINGLETON Extension**, **SUBTERM Multiset Extension**, **COMMON** $>$ **Extension**, ou **COMMON** $\sim$ **Extension**.

Comme dans le cas des règles de $\mathcal{C}$-déduction, si l'arc $\mathsf{F} \twoheadrightarrow^{P} \mathsf{G}$ (resp. $\mathsf{F} \twoheadrightarrow^{P} \mathsf{G}$) ajouté ou modifié est correct après l'application des règles, il en est de même pour tous les arcs situés au dessus, ou couverts (voir définition 5.23). De la même façon, les preuves concernant les règles normales (**SINGLETON**, **SUBTERM Multiset**, **COMMON** $>$, et **COMMON** $\sim$) et leurs extensions respectives (**SINGLETON Extension**, **SUBTERM Multiset Extension**, **COMMON** $>$ **Extension**, et **COMMON** $\sim$ **Extension**) sont similaires. D'autre part, les preuves pour les règles de simplification (**SINGLETON**Simplification, **SUBTERM Multiset**Simplification, **COMMON** $>$Simplification, et **COMMON** $\sim$Simplification) ne sont que des cas particuliers des règles normales.

Maintenant, par cas sur les règles **SINGLETON**, **SUBTERM Multiset**, **COMMON** $>$, et **COMMON** $\sim$ montrons que les arcs ajoutés sont toujours corrects.

SINGLETON On sait que S est un multi-ensemble ne contenant qu'un seul élément. Soit $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ cet élément. On a donc $Mult(\mathsf{S}) = \{\!\{s\}\!\}$. Dans les prémisses de la règle, on trouve les arcs $\mathsf{S} \twoheadrightarrow^{P_1} \mathsf{Y}_1$ et $\mathsf{S} \twoheadrightarrow^{P_1} \mathsf{Y}_2$. Or on sait par hypothèse que ces deux arcs sont corrects. On a donc pour toute instance Φ de *gpo* la propriété suivante: si $\Phi \models P_1\sigma$ et $\Phi \models P_2\sigma$ alors

$$\{\!\{s\}\!\} \succ_{gpo}^{\Phi,M} Mult(\mathsf{Y}_1) \text{ et}$$

$$\{\!\{s\}\!\} \succ_{gpo}^{\Phi,M} Mult(\mathsf{Y}_2).$$

Or, par la définition 5.19 de l'extension multi-ensemble, on sait que l'on a ces deux dernières inégalités si et seulement si: $\forall t \in (Mult(\mathsf{Y}_1) \cup Mult(\mathsf{Y}_2)) : s \succ_{gpo}^{\Phi} t$. D'où, finalement, pour toute instance Φ de *gpo* telle que $\Phi \models (P_1\sigma \wedge P_2\sigma)$, on a $Mult(\mathsf{S}) \succ_{gpo}^{\Phi,M} (Mult(\mathsf{Y}_1) \cup Mult(\mathsf{Y}_2)) = Mult(\mathsf{M}_1)$.

SUBTERM Multiset Les prémisses de la règle nous indiquent que pour toute instance Φ de *gpo* telle que $\Phi \models P'\sigma$, on a $Mult(\mathsf{X}_2) \succsim_{gpo}^{\Phi,M} Mult(\mathsf{Y})$. Par la proposition 5.1, on déduit de ceci que $(Mult(\mathsf{X}_1) \cup Mult(\mathsf{X}_2)) \succ_{gpo}^{\Phi,M} Mult(\mathsf{Y})$. D'où

$$Mult(\mathsf{M}_1) = (Mult(\mathsf{X}_1) \cup Mult(\mathsf{X}_2)) \succ_{gpo}^{\Phi,M} Mult(\mathsf{Y})$$

COMMON $>$ Les prémisses de la règle assurent que pour toute instance Φ de *gpo* telle que $\Phi \models P_1\sigma$ et $\Phi \models P_2\sigma$, on a $Mult(\mathsf{X}_1) \succsim_{gpo}^{\Phi,M} Mult(\mathsf{Y}_2)$ et $Mult(\mathsf{X}_2) \succ_{gpo}^{\Phi,M} Mult(\mathsf{Y}_1)$. Or, comme on l'a précisé au début de cette preuve, on sait que les multi-ensembles $Mult(\mathsf{X}_1) \cup Mult(\mathsf{X}_2)$ et $Mult(\mathsf{Y}_1) \cup Mult(\mathsf{Y}_2)$ n'ont pas d'éléments communs. Il est donc possible d'appliquer la proposition 5.2, et l'on obtient ainsi:

$$(Mult(\mathsf{X}_1) \cup Mult(\mathsf{X}_2)) \succ_{gpo}^{\Phi,M} (Mult(\mathsf{Y}_1) \cup Mult(\mathsf{Y}_2))$$

ce qui implique que $Mult(\mathsf{M}_1) \succ_{gpo}^{\Phi,M} Mult(\mathsf{M}_2)$.

COMMON $\sim$ Le raisonnement est semblable au cas de la règle **COMMON** $>$, mais on utilise le deuxième cas de la propriété 5.2. □

5.8.8 Complétude des règles de $\mathcal{C}_M$-déduction

Le théorème de la section 5.6.2 est le suivant.

Théorème 5.6 *(Complétude) Soit* $(G \parallel \sigma)$ *un SOCS en* $\mathcal{C}_M$*-forme normale, tel que* $G = (V, E)$*. Pour tous noeuds* $\mathsf{F}, \mathsf{G} \in V$*, t.q.* $Mult(\mathsf{F}) = X$ *et* $Mult(\mathsf{G}) = Y$*, où* $X, Y \in M(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ *et* $X \cap Y = \emptyset$*:*

$$\forall \Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex}) \text{ t.q. } X \succ_{gpo}^{\Phi,M} Y, \text{ il existe un arc } \mathsf{F} \xrightarrow{P} \mathsf{G} \text{ dans } G \text{ t.q. } \Phi \models P\sigma$$

$$\forall \Phi = (\mathcal{T}_{0,k}, \gtrsim_{lex}) \text{ t.q. } X \approx_{gpo}^{\Phi,M} Y, \text{ il existe un arc } \mathsf{F} \overset{P}{\rightsquigarrow} \mathsf{G} \text{ dans } G \text{ t.q. } \Phi \models P\sigma.$$

On procède par induction sur le nombre d'éléments contenus dans le multi-ensemble X. Par définition de $Mult$, le multi-ensemble $X = Mult(\mathsf{F})$ ne peut pas être vide. Il contient au minimum un élément. En conséquence, le cas de base de cette preuve par induction porte sur un multi-ensemble X ne contenant qu'un seul élément. Soit s cet élément; on a donc $X = \{\!\{s\}\!\}$.

Tout d'abord, nous examinons le cas où $X \succ_{gpo}^{\Phi,M} Y$. Pour toute instance Φ de *gpo* telle que $X = \{\!\{s\}\!\} \succ_{gpo}^{\Phi,M} Y$, selon la définition 5.19 de l'extension multi-ensemble, si $Y = \{\!\{t_1, \ldots, t_n\}\!\}$, on a nécessairement $\forall i = 1 \ldots n : s \succ_{gpo}^{\Phi} t_i$. Du théorème de complétude des règles de $\mathcal{C}$-déduction (théorème 5.2), on peut déduire qu'il existe nécessairement dans le graphe des arcs $s \xrightarrow{P_i} t_i$ $i = 1 \ldots n$, avec $\Phi \models P_i\sigma$ $i = 1 \ldots n$. Comme dans notre représentation, nous confondons les termes avec les multi-ensembles singletons les contenant. Nous avons également: $\forall i = 1 \ldots n$: $\{\!\{s\}\!\} \xrightarrow{P_i} \{\!\{t_i\}\!\}$ avec $\forall i = 1 \ldots n : \Phi \models P_i\sigma$. Or, en appliquant répétitivement la règle **SINGLETON** sur ces arcs on construit nécessairement un arc

$$X = \{\!\{s\}\!\} \xrightarrow{P} \{\!\{t_1, \ldots, t_n\}\!\} = Y$$

où P est associé à l'$\mathcal{O}$-preuve $P_1 \wedge \ldots \wedge P_n$ dans la Θ-substitution. Or il n'existe qu'un seul sommet multi-ensemble G tel que $Mult(\mathsf{G}) = Y$. Donc, on a bien $\mathsf{F} \xrightarrow{P} \mathsf{G}$ et $\Phi \models P\sigma$.

Maintenant, voyons le cas où $X \approx_{gpo}^{\Phi,M} Y$. Pour toute instance Φ de *gpo* telle que $X = \{\!\{s\}\!\} \succ_{gpo}^{\Phi,M} Y$, selon la définition 5.19 de l'extension multi-ensemble, Y est nécessairement un singleton $Y = \{\!\{t\}\!\}$ et l'on a $s \approx_{gpo}^{\Phi} t$. Toujours grâce au théorème de complétude des règles de $\mathcal{C}$-déduction (théorème 5.2), on peut déduire qu'il existe nécessairement dans le graphe un arc $s \overset{P}{\rightsquigarrow} t$, avec $\Phi \models P\sigma$. Or, comme nous confondons les termes et les multi-ensembles singleton les contenant, on a directement $\{\!\{s\}\!\} \overset{P}{\rightsquigarrow} \{\!\{t\}\!\}$, i.e. $\mathsf{F} \overset{P}{\rightsquigarrow} \mathsf{G}$ et $\Phi \models P\sigma$.

Dans le cas général de cette preuve par induction, pour tous noeuds $\mathsf{F}, \mathsf{G} \in V$, tels que $Mult(\mathsf{F}) = X$ et $Mult(\mathsf{G}) = Y$, et pour toute instance Φ de *gpo* telle que $X \succ_{gpo}^{\Phi,M} Y$ (resp. $X \approx_{gpo}^{\Phi,M} Y$), nous montrons que s'il n'existe pas d'arc $\mathsf{F} \xrightarrow{P} \mathsf{G}$ (resp. $\mathsf{F} \overset{P}{\rightsquigarrow} \mathsf{G}$) tel que $\Phi \models P\sigma$ alors une règle de $\mathcal{C}_M$-déduction peut être appliquée, contredisant le fait que le SOCS est en $\mathcal{C}_M$-forme normale.

- Si $X \succ_{gpo}^{\Phi,M} Y$, alors on procède par cas sur la définition de l'extension multi-ensemble. On a $Y \neq \emptyset$ puisque aucun noeud du SOCS ne représente l'ensemble vide. Il ne reste donc plus que les cas 2. et 3. de la définition à étudier.
  - Soit $X = \{\!\{s\}\!\} \cup X'$ et $Y = \{\!\{t_1, \ldots, t_n\}\!\} \cup Y'$ tels que $X' \gtrsim_{gpo}^{\Phi,M} Y'$ et $\forall i = 1 \ldots n$: $s \succ_{gpo}^{\Phi} t_i$. En appliquant le cas de base, de $\forall i = 1 \ldots n : s \succ_{gpo}^{\Phi} t_i$ on peut déduire qu'il existe un arc

    $$\{\!\{s\}\!\} \xrightarrow{P'} \{\!\{t_1, \ldots, t_n\}\!\}$$

    tel que $\Phi \models P'\sigma$. Si $Y' = \emptyset$, alors en appliquant la règle **SUBTERM Multiset** (ou son extension) sur ce dernier arc, on obtient l'arc suivant ($X' \neq \emptyset$ car dans ce cas X serait un singleton et le cas a déjà été traité)

    $$(\{\!\{s\}\!\} \cup X') \xrightarrow{P} \{\!\{t_1, \ldots, t_n\}\!\}$$

et $\{P \mapsto P'\}$ est ajouté à la Θ-substitution σ. On a donc $\Phi \models P\sigma$.

Si $Y' \neq \emptyset$, en appliquant l'hypothèse d'induction à $X' \succapprox^{\Phi,M}_{gpo} Y'$ ($X' \neq \emptyset$ car sinon X serait un singleton) on obtient qu'il existe un arc $X' \overset{P''}{\twoheadrightarrow} Y'$ ou un arc $X' \overset{P''}{\rightsquigarrow} Y'$ tel que $\Phi \models P''\sigma$. Or, à partir de l'arc:

$\{\!\{s\}\!\} \overset{P'}{\twoheadrightarrow} \{\!\{t_1, \dots, t_n\}\!\}$

et d'un arc $X' \overset{P''}{\twoheadrightarrow} Y'$ ou $X' \overset{P''}{\rightsquigarrow} Y'$

tels que $\Phi \models P'\sigma$ et $\Phi \models P''\sigma$, on peut appliquer la règle **COMMON** $>$ (ou son extension), et obtenir un arc $X \overset{P'''}{\twoheadrightarrow} Y$ tel que $\{P''' \mapsto P' \wedge P''\}$ est ajouté à la Θ-substitution σ, et donc $\Phi \models P'''\sigma$.

– Soit $X = \{\!\{s\}\!\} \cup X'$ et $Y = \{\!\{t\}\!\} \cup Y'$ tels que $X' \succ^{\Phi,M}_{gpo} Y'$ et $s \approx^{\Phi}_{gpo} t$. La preuve est très similaire au cas précédent. De $s \approx^{\Phi}_{gpo} t$, on déduit un arc $\{\!\{s\}\!\} \overset{P'}{\rightsquigarrow} \{\!\{t\}\!\}$. Si $Y' = \emptyset$, alors cet arc permet de construire un arc

$(\{\!\{s\}\!\} \cup X') \overset{P}{\twoheadrightarrow} \{\!\{t\}\!\}$

et $\{P \mapsto P'\}$ est ajouté à la Θ-substitution σ. On a donc $\Phi \models P\sigma$. Si $Y' \neq \emptyset$, en appliquant l'hypothèse d'induction à $X' \succ^{\Phi,M}_{gpo} Y'$ on obtient qu'il existe un arc $X' \overset{P''}{\twoheadrightarrow} Y'$ tel que $\Phi \models P''\sigma$. A partir de ce dernier arc et de $\{\!\{s\}\!\} \overset{P'}{\rightsquigarrow} \{\!\{t\}\!\}$, on voit qu'il est possible d'appliquer la règle **COMMON** $>$ (ou son extension) et de construire $X \overset{P'''}{\twoheadrightarrow} Y$ tel que $\{P''' \mapsto P' \wedge P''\}$ est ajouté à la Θ-substitution σ, et donc $\Phi \models P'''\sigma$.

– On peut utiliser les mêmes arguments de preuve dans le cas $X \approx^{\Phi,M}_{gpo} Y$. La preuve est plus simple puisque l'on a $X \approx^{\Phi,M}_{gpo} Y$ et X ne peut être ni vide ni un singleton. En conséquence, on a nécessairement $X = \{\!\{s\}\!\} \cup X'$ et $Y = \{\!\{t\}\!\} \cup Y'$, $s \approx^{\Phi}_{gpo} t$ et $X' \approx^{\Phi,M}_{gpo} Y'$. Comme X' ne peut être vide, et comme $X' \approx^{\Phi,M}_{gpo} Y'$, Y' ne peut pas être le multi-ensemble vide. De $s \approx^{\Phi}_{gpo} t$ et $X' \approx^{\Phi,M}_{gpo} Y'$ on déduit respectivement qu'il existe un arc $\{\!\{s\}\!\} \overset{P}{\rightsquigarrow} \{\!\{t\}\!\}$ et (par induction) un arc $X' \overset{P'}{\rightsquigarrow} Y'$ tels que $\Phi \models P\sigma$ et $\Phi \models P'\sigma$. Donc, on peut appliquer la règle **COMMON** $\sim$ (ou son extension). $\square$

Chapitre 6

Résolution de contraintes ensemblistes pour la preuve de terminaison

Dans cette partie, nous allons aborder le problème de la construction de l'ensemble des $\mathcal{R}$-descendants et des $\mathcal{R}$-formes normales d'un ensemble de termes E, pour un système de réécriture $\mathcal{R}$ donné. Initialement, nous nous sommes intéressés à la construction de ces ensembles afin de proposer un critère pour la terminaison de la relation de réduction séquentielle présentée dans la section 3.2.2. Or, il s'est avéré que ce critère de terminaison n'est qu'une application, parmi d'autres, du calcul des $\mathcal{R}$-descendants et des $\mathcal{R}$-formes normales. En conséquence, dans ce chapitre, nous présentons d'abord les problèmes liés au calcul de ces ensembles et nous verrons en fin de chapitre quelques-unes des applications possibles. Tout d'abord, dans la section 6.1, nous rappelons certains résultats concernant la décidabilité de la construction des ensembles $\mathcal{R}^*(E)$ et $\mathcal{R}^!(E)$. De ces résultats, il ressort qu'un calcul exact à l'aide d'automates d'arbres simples n'est possible que dans des cas très restreints trop limitatifs si l'on considère, en particulier, des systèmes qui représentent des programmes. Or, intuitivement, les similitudes fortes qui existent entre les automates d'arbres et les systèmes de réécriture semblent exploitables pour le calcul, ou au moins l'approximation, de tels ensembles. Cette intuition est confirmée dans la section 6.2. Dans cette section, nous montrons également que, pour approcher ces ensembles, il est nécessaire de donner une version généralisée de l'algorithme de normalisation des transitions. Celle-ci est détaillée dans la section 6.3. Enfin, dans la section 6.4, nous définissons formellement l'approximation: pour un système $\mathcal{R}$ linéaire à gauche et un ensemble régulier de terme, nous construisons des sur-ensembles réguliers de $\mathcal{R}^*(E)$ et $\mathcal{R}^!(E)$ paramétrés par une *fonction d'approximation* permettant de doser le degré de finesse du calcul [Gen98, Gen97a]. Dans la section 6.5, nous donnons quelques éléments de comparaison entre cette approximation et l'approximation de F. Jacquemard [Jac96b]. Nous évoquons également les limites de la méthode d'approximation. Dans la section 6.6, nous donnons les principales applications de ce calcul. Ce sont le test d'atteignabilité, l'approximation de co-domaine, la preuve de non-terminaison, la preuve de complétude suffisante et la preuve de terminaison de la réduction séquentielle.

6.1 Descendants, ensembles de formes normales et décidabilité

Dans le cas des systèmes qui nous intéressent, les systèmes de réécriture linéaires à gauche, on sait que $IRR(\mathcal{R})$ est un langage d'arbres régulier [GB85], et il existe un algorithme pour construire une grammaire régulière d'arbres (resp. un automate d'arbres) produisant (resp. reconnaissant) $IRR(\mathcal{R})$ [CR87].

Ici nous nous intéressons à la régularité des ensembles $\mathcal{R}^!(E)$ et $\mathcal{R}^*(E)$. On sait que ces deux ensembles sont liés par la relation suivante: $\mathcal{R}^!(E) = \mathcal{R}^*(E) \cap IRR(\mathcal{R})$. Donc, si $\mathcal{R}^*(E)$ est régulier, puisque l'intersection préserve la régularité, $\mathcal{R}^!(E)$ le sera aussi. Cependant, $\mathcal{R}^*(E)$ n'est pas nécessairement un langage d'arbres régulier même si E l'est. Le langage $\mathcal{R}^*(E)$ est régulier si E est régulier et si $\mathcal{R}$ est soit:

- un système de réécriture clos [Bra69, DT90],
- un système de réécriture linéaire à droite et monadique [Sal88], où monadique signifie que le membre droit de chaque règle de $\mathcal{R}$ est soit une variable, soit un terme $f(t_1, \ldots, t_n)$ où $f \in \mathcal{F}$, $ar(f) = n$, et t_i, $i = 1, \ldots, n$ est une variable ou une constante,
- un système de réécriture linéaire et semi-monadique [CDGV91], où semi-monadique signifie que le membre droit de chaque règle de $\mathcal{R}$ est soit une variable, soit un terme $f(t_1, \ldots, t_n)$ où $f \in \mathcal{F}$, $ar(f) = n$, et $\forall i = 1, \ldots, n$, t_i est une variable ou un terme clos,
- un système "décroissant" [Jac96b] [10], où décroissant signifie que le membre droit de chaque règle de $\mathcal{R}$ est soit une variable, soit un terme $f(t_1, \ldots, t_n)$ où $f \in \mathcal{F}$, $ar(f) = n$, et $\forall i = 1, \ldots, n$, t_i est une variable, un terme clos, ou un terme dont les variables n'apparaissent pas dans le membre gauche.

Cependant, pour un langage régulier E, $\mathcal{R}^*(E)$ n'est pas nécessairement régulier, même si $\mathcal{R}$ est un système linéaire confluent et terminant [GT95]. D'autre part, si on oublie l'une des restriction des systèmes décroissants, alors $\mathcal{R}^*(E)$ n'est pas nécessairement régulier [Jac96b].

6.2 Calcul des descendants

Pour un système de réécriture $\mathcal{R}$ et un ensemble de termes $E \subseteq \mathcal{T}(\mathcal{F})$ donnés, nous souhaitons calculer les ensembles $\mathcal{R}^*(E)$ et $\mathcal{R}^!(E)$ définis dans la section 2.3. Comme $\mathcal{R}^!(E) = \mathcal{R}^*(E) \cap IRR(\mathcal{R})$, et $IRR(\mathcal{R})$ est régulier et calculable pour tout système linéaire à gauche, nous pouvons concentrer nos efforts sur $\mathcal{R}^*(E)$. Dans la section 2.5, nous avons vu qu'un automate d'arbres $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ est un outil simple, basé sur la relation de réécriture $\to_\Delta$, qui permet de représenter des ensembles réguliers de termes. Un terme t appartient au langage reconnu par A, noté $\mathcal{L}(A)$ si il existe un état $q \in \mathcal{Q}_f$ tel que $t \to_\Delta^* q$. A partir de là, il est possible de voir le problème du calcul de $\mathcal{R}^*(\mathcal{L}(A))$, de deux façons différentes: comme une réécriture des termes reconnus par l'automate A ou comme une complétion de Δ à l'aide de $\mathcal{R}$. Ces deux vues du même procédé de calcul sont présentées informellement dans les sections 6.2.1 et 6.2.2 (les définitions correspondantes seront données dans la section 6.4). Enfin, dans la section 6.2.4, nous montrons que ce procédé de calcul de l'ensemble des descendants, tel qu'il a été ébauché, ne termine pas toujours et nous en verrons un exemple simple. Sur cet exemple, nous montrons également qu'il est possible de forcer le calcul à terminer mais que cela nécessite d'abord la généralisation de l'algorithme de normalisation des transitions.

6.2.1 Réécriture d'automates

Cette idée s'impose tout naturellement lorsque l'on considère un automate d'arbres reconnaissant exactement un seul terme, comme on le voit sur l'automate et le système de réécriture suivant.

10. plus précisément, F. Jacquemard a défini la classe des systèmes *croissants* et a montré que l'ensemble des *antécédents* était régulier. Dans cette thèse, nous avons préféré inverser ces notions dans un but d'homogéinité.

Soit l'automate d'arbres A reconnaissant exactement le terme $h(f(g(a)))$: $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, où $\mathcal{F} = \{h : 1, f : 1, g : 1, a : 0\}$, $\mathcal{Q} = \{q_0, q_1, q_2, q_3\}$, $\mathcal{Q}_f = \{q_0\}$, et $\Delta = \{$

$$\begin{array}{r}h(q_1) \to q_0 \\ f(q_2) \to q_1 \\ g(q_3) \to q_2 \\ a \to q_3\end{array} \;\}$$

Soit $\mathcal{R}$ le système de réécriture $\mathcal{R} = \{f(g(x)) \to g(f(x))\}$. On peut voir l'automate A comme un découpage du terme $h(f(g(a)))$ dans lequel les états représentent des positions du terme et les transitions associent un terme à chaque position. Comme on l'a vu dans la section 2.3, l'application d'une règle de réécriture $l \to r$ sur un terme t passe par la recherche d'une position $p \in \mathcal{P}os_{\mathcal{F}}(t)$ et d'une substitution (un filtre) σ telle que $t|_p = l\sigma$, puis par le remplacement de $l\sigma$ par $r\sigma$ dans t à la même position p.

De la même façon, pour réécrire le terme $h(f(g(a)))$, codé par A, avec la règle $f(g(x)) \to g(f(x))$, il semble naturel de rechercher un filtre $f(g(x))$ *à toutes les positions* du terme. Comme nous avons vu que dans A, les positions sont codées par des états, ceci revient donc à rechercher un état $q \in \mathcal{Q}$ (une position) reconnaissant un terme de la forme $f(g(x))$. Or, si l'on passe en revue toutes les transitions de Δ, on peut voir que le seul état (la seule position) pouvant reconnaître un terme dont le symbole de tête est f est q_1. De la même façon, en décomposant, q_1 reconnaît une instance de $f(g(x))$, si l'on trouve une instance de $g(x)$ à la position q_2, et donc si l'on trouve une instance de x à la position q_3. Dans notre cas, toutes ces conditions sont vérifiées, y compris la dernière: tout terme figurant à la position q_3 est nécessairement une instance de x. Ainsi, dans la substitution σ, à x on associe le terme figurant à la position q_3.

A la position q_1, le terme sur lequel s'applique la règle de réécriture est donc $f(g(x))\sigma$. Vient ensuite le remplacement de $f(g(x))\sigma$ par $g(f(x))\sigma$ à la position q_1. Il faut remarquer que dans notre cas, nous souhaitons obtenir un automate reconnaissant $\mathcal{R}^*(\mathcal{L}(A))$, il ne faut donc pas remplacer $f(g(x))\sigma$ par $g(f(x))\sigma$ mais ajouter $g(f(x))\sigma$ à la position q_1. Dans le codage utilisé pour le découpage des termes, ceci revient à ajouter à Δ les transitions:

$$\begin{array}{r}g(q_4) \to q_1 \\ f(q_3) \to q_4\end{array}$$

où q_4 est un nouvel état ajouté à $\mathcal{Q}$. Soit A' l'automate obtenu par adjonction de ces transitions et ces états à A. Le langage reconnu par A' est $\mathcal{L}(A') = \{h(f(g(a))), h(g(f(a)))\} = \mathcal{R}^*(\mathcal{L}(A))$.

Dans la section suivante, nous donnons une autre vue du même procédé basé sur une procédure de complétion des règles de $\mathcal{R}$ et de Δ.

6.2.2 Compléter Δ par $\mathcal{R}$

Pour reconnaître $\mathcal{R}^*(\mathcal{L}(A))$ au lieu de $\mathcal{L}(A)$, il est intuitif de chercher à *étendre* la relation $\to_\Delta$ de façon à ce que l'automate vérifie la propriété suivante: pour tout terme t tel que $t \to_\Delta^* q$ et $t \to_{\mathcal{R}}{}^* u$, on a $u \to_\Delta^* q$. Cette propriété est très proche de la propriété de confluence du système de réécriture $\mathcal{R} \cup \Delta$. Cependant, même si elle est proche, la propriété que doit vérifier la relation $\to_\Delta^*$ "étendue" est plus forte que la confluence puisque pour tout terme t tel que $t \to_\Delta^* q$ et $t \to_{\mathcal{R}}{}^* u$, il faut remarquer que q et u sont obtenus par des systèmes de réécriture disjoints et, d'autre part, il faut assurer $u \to_\Delta^* q$ et non pas $u \to_{\mathcal{R}\cup\Delta}^* q$.

Malgré ces différences, en utilisant une technique très semblable à la complétion (voir section 3.4.2), il est possible d'étendre Δ afin de reconnaître les descendants des termes de $\mathcal{L}(A)$.

Ainsi pour toute règle $l \to r$ et pour toute substitution σ associant aux variables de l des états de $\mathcal{Q}$, il est possible de rechercher des paires critiques $(r\sigma, q)$ telles que $l\sigma \to_{\mathcal{R}}{}^* r\sigma$ et $l\sigma \to_{\Delta}^* q$. Si l'on ajoute la transition $r\sigma \to q$ à Δ, on assure bien que le descendant direct de $l\sigma$ sera également reconnu par l'état q. Ainsi si l'on reprend l'exemple de la section précédente, la seule paire critique entre $\mathcal{R}$ et Δ est $f(g(q_3)) \to_{\Delta}^* q_1$ et $f(g(q_3)) \to_{\mathcal{R}}{}^* g(f(q_3))$. La transition à ajouter à Δ est donc $g(f(q_3)) \to q_1$. Mais, afin de conserver un automate A normalisé, il est nécessaire de remplacer cette transition par une forme normalisée, par exemple $g(q_4) \to q_1$ et $f(q_3) \to q_4$, où q_4 est un nouvel état ajouté à $\mathcal{Q}$. Cependant, comme toute complétion, celle-ci peut diverger, nous en verrons des exemples dans la section 6.2.4.

6.2.3 Filtrage dans les automates

Pour une règle de réécriture $l \to r \in \mathcal{R}$ et pour un automate $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, afin de trouver une substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ et un état $q \in \mathcal{Q}$ tels que $l\sigma \to_{\Delta}^* q$, il est possible d'énumérer toutes les combinaisons possibles de σ et de q, et de vérifier si $l\sigma \to_{\Delta}^* q$. Cependant, en pratique et lorsque $\mathcal{Q}$ est un grand ensemble, cette solution n'est pas viable à cause du nombre important de paires (σ, q) à considérer. Nous proposons maintenant un algorithme naturel, proche d'un algorithme de filtrage standard, permettant de déduire ces états et substitutions. Tout d'abord nous définissons ces substitutions particulières qui associent un état à une variable.

Définition 6.1 *Une Q-substitution est une application de $\mathcal{X}$ vers $\mathcal{Q}$. Soit $\Sigma(\mathcal{Q}, \mathcal{X})$ l'ensemble des Q-substitutions définies sur $\mathcal{X}$ et $\mathcal{Q}$.*

Dans la suite, ce que nous nommons un *problème de filtrage* est une formule logique du premier ordre, sans quantificateurs, construite sur les littéraux $\perp$, $s \trianglelefteq c$ où $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, et sur les symboles usuels de disjonction $\vee$ et de conjonction $\wedge$. Une conjonction vide $\bigwedge_{\emptyset}$ est un problème de filtrage trivialement vrai.

Définition 6.2 *Soit ϕ, ϕ_1, ϕ_2 des problèmes de filtrage, $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ un terme linéaire, $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, et $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres. Une* solution *d'un problème de filtrage ϕ est une Q-substitution $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$ telle que*

- *si $\phi = s \trianglelefteq c$, alors $s\sigma \to_{\Delta}^* c$, ou*
- *si $\phi = \phi_1 \wedge \phi_2$, alors σ est une solution de ϕ_1 et une solution de ϕ_2, ou*
- *si $\phi = \phi_1 \vee \phi_2$, alors σ est une solution de ϕ_1 ou une solution de ϕ_2.*

Nous supposons que le filtrage est appliqué à un automate sans epsilon-transitions. Une epsilon-transition est une transition de la forme $q \to q'$ où q et q' sont des états. Il faut remarquer que tout ensemble de transitions $\Delta \cup \{q \to q'\}$ peut être remplacé par un ensemble de transitions équivalent $\Delta \cup \{c \to q' \mid c \to q \in \Delta\}$. Nous donnons maintenant l'algorithme de filtrage.

Définition 6.3 *Soit $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres, $f \in \mathcal{F}$, $ar(f) = n$, $g \in \mathcal{F}$, $ar(g) = m$, $q, q_1, \ldots, q_n \in \mathcal{Q}$, $q'_1, \ldots, q'_n \in \mathcal{Q}$, $c_1, \ldots, c_d \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $s, s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et ϕ_1, ϕ_2, ϕ_3 des problèmes de filtrage non-vides. L'algorithme de filtrage consiste en la normalisation de tout problème de filtrage de la forme $s \trianglelefteq q$ par l'ensemble suivant de règles.*

Decompose $\dfrac{f(s_1,\dots,s_n) \trianglelefteq f(q_1,\dots,q_n)}{s_1 \trianglelefteq q_1 \wedge \dots \wedge s_n \trianglelefteq q_n}$

Clash $\dfrac{f(s_1,\dots,s_n) \trianglelefteq g(q'_1,\dots,q'_m)}{\perp}$

si $f \neq g$.

Configuration $\dfrac{s \trianglelefteq q}{\bigvee_{c\in C_f} s \trianglelefteq c \vee \perp}$

si $s \notin \mathcal{X}$, *où* $C_f = \{c \mid c \to q \in \Delta\}$

De plus, après chaque application de ces règles, les problèmes de filtrage sont normalisés à l'aide de l'ensemble de règles ξ *suivant:*

$$\frac{\phi_1 \wedge (\phi_2 \vee \phi_3)}{(\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)} \qquad \frac{\phi_1 \vee \perp}{\phi_1} \qquad \frac{\phi_1 \wedge \perp}{\perp}$$

La correction, la complétude et la terminaison de l'algorithme sont données par le théorème suivant.

Théorème 6.1 *Etant donné* $s \in \mathcal{T}(\mathcal{F},\mathcal{X})$, *un ensemble d'états* $\mathcal{Q}$, *un ensemble de transitions* Δ *et* $q \in \mathcal{Q}$, *tout problème de filtrage* $s \trianglelefteq q$ *a une forme normale telle que*

- *s'il s'agit de* $\perp$ *alors, il n'existe pas de Q-substitution* σ *t.q.* $s\sigma \to^*_\Delta q$,
- *s'il s'agit d'un problème de filtrage vide, alors pour toute Q-substitution* σ, $s\sigma \to^*_{\Delta'} q$,
- *sinon, la forme normale est une disjonction* $\bigvee_{i=1}^k \phi_i$ *t.q.* $\phi_i = \bigwedge_{j=1}^{n_i} x^i_j \trianglelefteq q^i_j$, *où* $x^i_j \in \mathcal{X}$, $q^i_j \in \mathcal{Q}$, *et* $\sigma_1 = \{x^1_j \mapsto q^1_j \mid j = 1\dots n_1\},\dots,\sigma_k = \{x^k_j \mapsto q^k_j \mid j = 1\dots n_k\}$ *sont les seules Q-substitutions t.q.* $s\sigma_i \to^*_\Delta q$.

Preuve La preuve de ce théorème est donnée dans la section 6.7.1. □

Voici un exemple d'exécution de cet algorithme sur un problème de filtrage simple.

Exemple 6.1 *Soit* $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$, *où* $\mathcal{F} = \{f,g,a\}$, $\mathcal{Q} = \{q_0,q_1\}$, $\mathcal{Q}_f = \{q_0\}$ *et* $\Delta = \{f(q_1) \to q_0, g(q_1) \to q_1, a \to q_1\}$. *Le langage* $\mathcal{L}(A)$ *est* $\{f(g^*(a))\}$. *Soit* $\mathcal{R} = \{f(g(x)) \to g(f(x))\}$. *Si on applique l'algorithme de filtrage sur le problème de filtrage* $f(g(x)) \trianglelefteq q_0$, *on obtient les déductions suivantes, où le nom de la règle appliquée est donné à droite, et les étapes de normalisation à l'aide des règles de* ξ *sont omises.*

$f(g(x)) \trianglelefteq q_0$	*règle* **Configuration**
$f(g(x)) \trianglelefteq f(q_1)$	*règle* **Decompose**
$g(x) \trianglelefteq q_1$	*règle* **Configuration**
$g(x) \trianglelefteq g(q_1)$ *ou* $g(x) \trianglelefteq a$	*règle* **Clash**
$g(x) \trianglelefteq g(q_1)$	*règle* **Decompose**
$x \trianglelefteq q_1$	

Soit σ *la Q-substitution* $\sigma = \{x \mapsto q_1\}$. *Nous avons déduit que* $l\sigma = f(g(q_1)) \to^*_\Delta q_0$.

6.2.4 Divergence du calcul

Nous avons utilisé, dans les sections 6.2.1 et 6.2.2, un exemple dans lequel le calcul, tel que nous l'avons informellement présenté, est fini et exact. Voici maintenant un exemple simple dans lequel se produit une divergence.

Exemple 6.2 *Soit* $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ *l'automate tel que* $\mathcal{F} = \{f : 1, g : 1, a : 0\}$, $Q = \{q_0, q_1\}$, $Q_f = \{q_0\}$, $\Delta = \{$

$$\begin{aligned} f(q_0) &\to q_0, \\ g(q_1) &\to q_0, \\ a &\to q_1 \ \} \end{aligned}$$

et $\mathcal{R}$ *le système de réécriture* $\mathcal{R} = \{f(g(x)) \to g(f(x))\}$. *Le langage reconnu par* A *est* $\mathcal{L}(A) = \{f^*(g(a))\}$. *L'ensemble* $\mathcal{L}(A)$, *l'ensemble des descendants en au plus un pas de* $\rightarrow_{\mathcal{R}}$ *et l'ensemble des descendants* $\mathcal{R}^*(E)$ *sont représentés figure 6.1. Or, si l'on applique sur* A *et* $\mathcal{R}$ *le procédé*

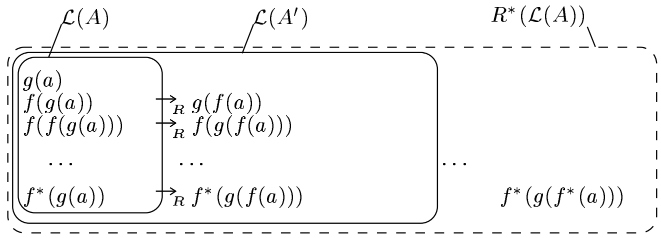


FIG. 6.1 – *Ensemble des* $\mathcal{R}$ *descendants de* $\mathcal{L}(A)$

de calcul décrit dans les sections précédentes sous la forme d'une complétion par exemple, nous obtenons la paire critique suivante: $f(g(q_1)) \rightarrow^*_\Delta q_0$ *et* $f(g(q_1)) \rightarrow_{\mathcal{R}}{}^* g(f(q_1))$. *Si nous ajoutons à* Δ *les nouvelles transitions normalisées* $g(q_2) \to q_0$ *et* $f(q_1) \to q_2$ *où* q_2 *est un nouvel état, nous assurons bien que* $g(f(q_1)) \rightarrow^*_\Delta q_0$. *Soit* A' *l'automate obtenu, son ensemble de transitions est* $\Delta' = \{$

$$\begin{aligned} f(q_0) &\to q_0, \\ g(q_1) &\to q_0, \\ a &\to q_1, \\ g(q_2) &\to q_0 \\ f(q_1) &\to q_2 \ \}. \end{aligned}$$

L'automate A' *reconnaît l'ensemble des descendants en au plus un pas de* $\rightarrow_{\mathcal{R}}$. *Or, sur* A', *il existe une nouvelle paire critique:* $f(g(q_2)) \rightarrow^*_\Delta q_0$ *et* $f(g(q_2)) \rightarrow_{\mathcal{R}} g(f(q_2))$. *Il est possible d'ajouter à nouveau des transitions normalisées* $g(q_3) \to q_0$ *et* $f(q_2) \to q_3$ *où* q_3 *est un nouvel état, assurant que* $g(f(q_2)) \rightarrow^*_\Delta q_0$, *mais on voit que ce processus peut se poursuivre indéfiniment.*

Le calcul ne terminant pas, il est impossible de construire un automate reconnaissant $\mathcal{R}^*(\mathcal{L}(A))$ de cette façon. Pourtant, sur cet exemple, le langage $\mathcal{R}^*(\mathcal{L}(A))$ est régulier: $\mathcal{R}^*(\mathcal{L}(A)) = \{f^*(g(f^*(a)))\}$. A ce stade, il est intéressant de remarquer que, pour normaliser la transition $g(f(q_2)) \to q_0$ correspondant à la paire critique de A', si au lieu d'utiliser un nouvel état q_3 nous avions *réutilisé* l'état q_2, les transitions normalisées que nous aurions ajouté à Δ' aurait été: $g(q_2) \to q_0$ et $f(q_2) \to q_2$, et l'automate A'' obtenu aurait été tel que $\mathcal{L}(A'') = \mathcal{R}^*(A)$.

A partir de ces observations, dans la section 6.4 nous donnons une façon systématique de localiser les divergences potentielles et également de choisir les états à réutiliser pour la normalisation des transition. Cette méthode est, entre autres, basée sur la recherche de récursivité dans l'application des règles de réécriture. Cependant, cette méthode n'est pas exacte en général, elle calcule une approximation: un sur-ensemble de l'ensemble des descendants.

D'autre part, il faut remarquer que la normalisation, telle que nous venons de la pratiquer, ne correspond pas à la définition standard (voir définition 2.13 page 16). Avant toute chose, il est nécessaire de généraliser cette définition, c'est ce que nous faisons dans la section suivante.

6.3 Généralisation de la normalisation des transitions

L'algorithme proposé ici est une forme généralisée de l'algorithme standard (définition 2.13 page 16). Dans cet algorithme, les transitions sont normalisées à l'aide d'états dont le choix est guidé par une fonction d'abstraction α, et qui ne sont pas nécessairement des nouveaux états. Nous définissons d'abord la fonction d'abstraction.

Définition 6.4 *Soit $\mathcal{F}$ un ensemble de symboles, et $\mathcal{Q}$ un ensemble d'états. Pour une configuration donnée $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q}$, une abstraction de s est une fonction surjective α:*

$$\alpha : \{s|_p \ \mid \ p \in \mathcal{P}os_{\mathcal{F}}(s)\} \mapsto \mathcal{Q}$$

La fonction α est étendue à $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ en définissant α comme étant l'identité sur $\mathcal{Q}$.

Définition 6.5 *Soit $\mathcal{F}$ un ensemble de symboles, $\mathcal{Q}$ un ensemble d'états, $s \to q$ une transition telle que $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q \in \mathcal{Q}$, et α une abstraction de s. L'ensemble de transitions normalisées $Norm_\alpha(s \to q)$ est défini inductivement par:*

- *si $s = q$, alors $Norm_\alpha(s \to q) = \emptyset$,*
- *si $s \in \mathcal{Q}$ et $s \neq q$, alors $Norm_\alpha(s \to q) = \{s \to q\}$,*
- *si $s = f(t_1, \ldots, t_n)$, alors $Norm_\alpha(s \to q) = \{f(\alpha(t_1), \ldots, \alpha(t_n)) \to q\} \cup \bigcup_{i=1}^{n} Norm_\alpha(t_i \to \alpha(t_i))$.*

Exemple 6.3 *Soit $\mathcal{F} = \{f, g, a\}$ et $\mathcal{Q} = \{q_0, q_1, q_2, q_3, q_4\}$. Soit $s = f(g(q_1, f(a)))$, et α_1 une abstraction de s telle que $\alpha_1(a) = q_4$, $\alpha_1(f(a)) = q_3$, $\alpha_1(g(q_1, f(a))) = q_2$. Avec cette fonction α_1, le comportement de l'algorithme est équivalent à celui de la définition 2.13 page 16: $Norm_{\alpha_1}(f(g(q_1, f(a))) \to q_0) = \{f(q_2) \to q_0, g(q_1, q_3) \to q_2, f(q_4) \to q_3, a \to q_4\}$. En revanche avec une abstraction α_2 telle que $\alpha_2(a) = q_0$, $\alpha_2(f(a)) = q_0$, $\alpha_2(g(q_1, f(a))) = q_2$, on obtient: $Norm_{\alpha_2}(f(g(q_1, f(a))) \to q_0) = \{f(q_2) \to q_0, g(q_1, q_0) \to q_2, f(q_0) \to q_0, a \to q_0\}$.*

6.4 Approximation de l'ensemble des descendants $\mathcal{R}^*(E)$

Notre but étant de travailler avec des systèmes de réécriture représentant des programmes, nous ne pouvons pas nous en tenir à la classe des systèmes "décroissants" qui ne permet pas, par exemple, d'exprimer un système contenant un appel récursif. Nous avons donc choisi de définir une *approximation de* $\mathcal{R}^*(E)$, c'est à dire un sur-ensemble régulier de $\mathcal{R}^*(E)$ pour tout système de réécriture $\mathcal{R}$ linéaire à gauche et tout langage régulier E. De plus, comme les langages réguliers sont clos par intersection, l'intersection entre le sur-ensemble régulier de $\mathcal{R}^*(E)$ et $IRR(\mathcal{R})$ donnera un sur-ensemble régulier de $\mathcal{R}^!(E)$.

6.4.1 Définition de l'approximation

Pour construire l'approximation de $\mathcal{R}^*(E)$, nous allons nous appuyer sur la notion d'ensemble *clos par réécriture*. Soit $\mathcal{R}$ un système de réécriture. Un ensemble de termes $E \subseteq \mathcal{T}(\mathcal{F})$ est *clos par* $\mathcal{R}$ si pour tout $s \in E$ tel que $s \to_{\mathcal{R}} t$ alors $t \in E$. Soit A un automate d'arbres et $\mathcal{R}$ un système de réécriture. Nous donnons, dans la proposition suivante, une condition pour que le langage reconnu par A soit clos par $\mathcal{R}$.

Proposition 6.1 *Soit $\mathcal{R}$ un système de réécriture linéaire à gauche, et $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres. L'ensemble $\mathcal{L}(A)$ est clos par $\mathcal{R}$ si*

$$\forall l \to r \in \mathcal{R},\ \forall q \in \mathcal{Q},\ \forall \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}),\ l\sigma \to_{\Delta}^* q \text{ implique } r\sigma \to_{\Delta}^* q.$$

Preuve Supposons qu'il existe un terme s reconnu par $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ tel que $s \to_{\mathcal{R}} t$ et $t \notin \mathcal{L}(A)$. De $s \to_{\mathcal{R}} t$, on peut déduire qu'il existe un contexte clos $C[\,]$, une règle $l \to r \in \mathcal{R}$ et une substitution σ tels que $s = C[l\sigma] \to_{\mathcal{R}} C[r\sigma] = t$. D'autre part, comme $s \in \mathcal{L}(A)$, on sait que $C[l\sigma] \to_{\Delta}^* q$ avec $q \in \mathcal{Q}_f$. Les transitions de Δ étant normalisées, on peut en déduire qu'il existe un état $q' \in \mathcal{Q}$ tel que $C[q'] \to_{\Delta}^* q$ et $l\sigma \to_{\Delta}^* q'$. Soient $x_1, \ldots, x_n$ les variables de l, et $q_1, \ldots, q_n \in \mathcal{Q}$ les états tels que $x_1\sigma \to_{\Delta}^* q_1, \ldots, x_n\sigma \to_{\Delta}^* q_n$ et $l\sigma' \to_{\Delta}^* q'$ où $\sigma' \in \Sigma(\mathcal{Q}, \mathcal{X})$ est définie par $\sigma' = \{x_i \mapsto q_i \mid i = 1 \ldots n\}$.

De $l\sigma' \to_{\Delta}^* q'$, on déduit que $r\sigma' \to_{\Delta}^* q'$. Or, comme $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ et $x_1\sigma \to_{\Delta}^* q_1, \ldots, x_n\sigma \to_{\Delta}^* q_n$ pour toutes les variables de l, on a $r\sigma \to_{\Delta}^* q'$. Finalement, de ce dernier résultat et de $c[q'] \to_{\Delta}^* q$, on obtient $t = C[r\sigma] \to_{\Delta}^* q$ avec $q \in \mathcal{Q}_f$. D'où $t \in \mathcal{L}(A)$ ce qui contredit l'hypothèse de départ. $\square$

Maintenant, nous donnons un algorithme qui, à partir d'un automate $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ et d'un système de réécriture $\mathcal{R}$ linéaire à gauche, construit un automate $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ tel que $\mathcal{L}(A) \subseteq \mathcal{L}(\mathcal{T}_{\mathcal{R}} \uparrow (A))$ et $\mathcal{L}(\mathcal{T}_{\mathcal{R}} \uparrow (A))$ est clos par $\mathcal{R}$. Cet algorithme part de l'automate A et ajoute incrémentalement des transitions à Δ de façon à remplir la condition de la proposition précédente.

Algorithme Tant qu'il existe une paire critique entre les règles de Δ et celles de $\mathcal{R}$:

$$\begin{tikzcd} l\sigma \arrow[r, "*", "\Delta"'] \arrow[d, "R"'] & q \\ r\sigma \arrow[ur, dashed, bend right, "*"', "\Delta"] & \end{tikzcd}$$

telle que $r\sigma \not\to_{\Delta}^* q$, on ajoute la transition $r\sigma \to q$ à Δ. Si la transition $r\sigma \to q$ n'est pas normalisée, on la normalise à l'aide de la fonction $Norm$ (voir définition 6.5 page 117).

Le choix des nouveaux états utilisés pour la normalisation de $r\sigma \to q$ est guidé par la fonction d'approximation γ que nous définissons maintenant. Pour un état q, une substitution σ et une règle $l \to r$, γ calcule une séquence d'états nouveaux dont la longueur est le nombre de positions fonctionnelles de r.

Définition 6.6 *Soit $\mathcal{Q}$ un ensemble d'états, $\mathcal{Q}_{new}$ un ensemble de nouveaux états tels que $\mathcal{Q} \cap \mathcal{Q}_{new} = \emptyset$, et $\mathcal{Q}_{new}^*$ l'ensemble des séquences $q_1 \cdots q_k$ d'états de $\mathcal{Q}_{new}$. Une fonction d'approximation est une application $\gamma : \mathcal{R} \times (\mathcal{Q} \cup \mathcal{Q}_{new}) \times \Sigma(\mathcal{Q} \cup \mathcal{Q}_{new}, \mathcal{X}) \mapsto \mathcal{Q}_{new}^*$, telle que $\gamma(l \to r, q, \sigma) = q_1 \cdots q_k$, où $k = \mathcal{C}ard(\mathcal{P}os_{\mathcal{F}}(r))$.*

Dans la suite, pour toute séquence $S = q_1 \cdots q_k \in \mathcal{Q}_{new}^*$, et pour tout i tel que $1 \leq i \leq k$, on notera par $\pi_i(S)$ la $i$ème projection de la séquence S, i.e. q_i.

Définition 6.7 *(Automate d'approximation) Soit $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres, $\mathcal{R}$ un système de réécriture linéaire à gauche, $\mathcal{Q}_{new}$ un ensemble de nouveaux états tels que $\mathcal{Q} \cap \mathcal{Q}_{new} = \emptyset$, et γ une fonction d'approximation. Un automate approximation $\mathcal{T}_\mathcal{R} \uparrow (A)$ est un automate d'arbres $\langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ tel que*

- $\mathcal{Q}' = \mathcal{Q} \cup \mathcal{Q}_{new}$, *et*
- $\Delta \subseteq \Delta'$, *et*
- $\forall l \to r \in \mathcal{R}$, $\forall q \in \mathcal{Q}'$, $\forall \sigma \in \Sigma(\mathcal{Q}', \mathcal{X})$, $l\sigma \to_{\Delta'}^* q$ *implique*

$$Norm_\alpha(r\sigma \to q) \subseteq \Delta'$$

où α est l'abstraction de $r\sigma$ définie par: $\alpha(r\sigma|_{p_i}) = \pi_i(\gamma(l \to r, q, \sigma))$, pour tout $p_i \in \mathcal{P}os_\mathcal{F}(r) = \{p_1, \ldots, p_k\}$, tel que $p_i \prec p_{i+1}$ pour $i = 1 \ldots k-1$ (où nous rappelons que $\prec$ est l'ordre lexicographique et $\pi_i(S)$ est la ième projection de la séquence S.).

En choisissant une fonction d'approximation particulière γ, on définit une fonction d'abstraction particulière α, on conditionne donc l'algorithme de normalisation défini dans la section 6.5 et finalement on obtient des automates approximation spécifiques. Dans la section 6.4, nous verrons l'exemple de la fonction d'approximation ancêtre.

Théorème 6.2 *Pour tout automate d'arbres A, pour tout système de réécriture $\mathcal{R}$ linéaire à gauche, et pour toute fonction d'approximation γ, le langage $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$ est clos par $\mathcal{R}$.*

Preuve (idée) Afin de prouver ce résultat, il suffit de montrer que l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ vérifie la condition de la proposition 6.1. Ceci revient à montrer que $Norm_\alpha(r\sigma \to q) \subseteq \Delta'$ implique $r\sigma \to_{\Delta'}^* q$. Voir la section 6.7.2 pour une preuve plus détaillée. □

Corollaire 6.1 *Pour tout automate d'arbres A, pour tout système de réécriture $\mathcal{R}$ linéaire à gauche, et pour toute fonction d'approximation γ,*

$$\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A)) \supseteq \mathcal{R}^*(\mathcal{L}(A))$$

Preuve Tout d'abord, on sait que $\mathcal{L}(A) \subseteq \mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$. En effet, l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ est construit en ajoutant des états et des transitions à l'automate A, en conséquence, pour tout terme s, si $s \in \mathcal{L}(A)$, i.e. s'il existe une dérivation $s \to_\Delta^* q$ où q est un état final, alors cette dérivation est encore possible avec $\mathcal{T}_\mathcal{R} \uparrow (A)$.

En outre, par le théorème précédent, on sait que $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$ est clos par réécriture, d'où $\mathcal{R}^*(\mathcal{L}(A)) \subseteq \mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$ □

Le langage $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$ est clos par réécriture, mais il faut remarquer que ce n'est pas nécessairement le cas pour tous les sur-ensembles de $\mathcal{R}^*(E)$. Par exemple, soient $E = \{a\}$ et $\mathcal{R} = \{a \to b, c \to d\}$. On a $\mathcal{R}^*(E) = \{a, b\}$ et l'ensemble $\{a, b, c\}$ bien qu'étant un sur-ensemble de $\mathcal{R}^*(E)$ n'est pas clos par réécriture puisque $c \to_\mathcal{R} d$ et $d \notin \mathcal{R}^*(E)$.

6.4.2 Une approximation finie

Suivant la fonction d'approximation γ utilisée, l'ajout de transitions à Δ ne termine pas nécessairement, comme nous le voyons dans l'exemple suivant.

Exemple 6.4 *Soit A un automate d'arbres où $\Delta = \{app(q_0, q_0) \to q_1, cons(q_2, q_1) \to q_0, nil \to q_0, nil \to q_1, a \to q_2\}$, $rl = app(cons(x, y), z) \to cons(x, app(y, z))$, $\mathcal{R} = \{rl\}$, et soit γ la fonction d'approximation associant chaque t-uple (rl, q, σ) à un nouvel état unique (puisque dans ce cas particulier, $\mathcal{C}ard(\mathcal{P}os_{\mathcal{F}}(cons(x, app(y, z)))) = 1$).*

Étape 1 *Si l'on applique l'algorithme de filtrage sur $app(cons(x, y), z) \trianglelefteq q_1$, on obtient une solution $\sigma = \{x \mapsto q_2, y \mapsto q_1, z \to q_0\}$, correspondant à la paire critique suivante:*

$$\begin{tikzcd} app(cons(q_2, q_1), q_0) \arrow[r, "*", "\Delta"'] \arrow[d, "R"'] & q_1 \\ cons(q_2, app(q_1, q_0)) \arrow[ur, dashed, bend right, "*", "\Delta"'] & \end{tikzcd}$$

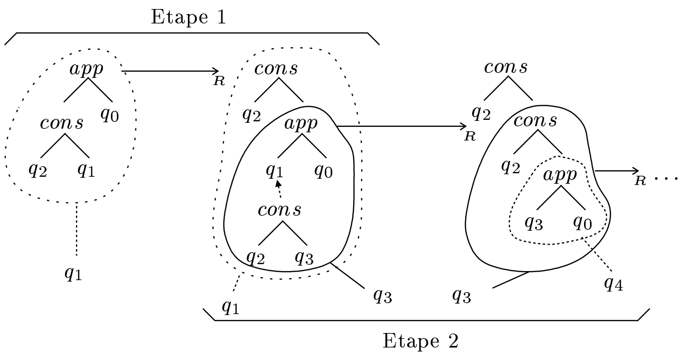
Soit q_3 un nouvel état et $\gamma(rl, q_1, \sigma) = q_3$. Maintenant, puisque $\mathcal{P}os_{\mathcal{F}}(cons(q_2, app(q_1, q_0))) = \{p_1\} = \{2\}$, on a $\alpha(app(q_1, q_0)) = \pi_1(\gamma(rl, q_1, \sigma)) = q_3$, et l'ensemble des transitions normalisées à ajouter à Δ est $Norm_\alpha(cons(q_2, app(q_1, q_0)) \to q_1) = \{cons(q_2, q_3) \to q_1, app(q_1, q_0) \to q_3\}$.

Étape 2 *En appliquant une nouvelle fois l'algorithme de filtrage sur $app(cons(x, y), z) \trianglelefteq q_3$, on obtient une solution $\sigma' = \{x \mapsto q_2, y \mapsto q_3, z \mapsto q_0\}$, correspondant à la paire critique suivante:*

$$\begin{tikzcd} app(cons(q_2, q_3), q_0) \arrow[r, "*", "\Delta"'] \arrow[d, "R"'] & q_3 \\ cons(q_2, app(q_3, q_0)) \arrow[ur, dashed, bend right, "*", "\Delta"'] & \end{tikzcd}$$

Soit q_4 le nouvel état et $\gamma(rl, q_3, \sigma') = q_4$. On a $\alpha(app(q_3, q_0)) = q_4$, et l'ensemble des transitions normalisées à ajouter à Δ est $Norm_\alpha(cons(q_2, app(q_3, q_0)) \to q_3) = \{cons(q_2, q_4) \to q_1, app(q_3, q_0) \to q_4\}$.

Ce processus pourrait s'exécuter indéfiniment et créer une infinité de nouveaux états. Ceci est dû au fait que nous pouvons appliquer récursivement la règle $app(cons(x, y), z) \to cons(x, app(y, z))$ sur des termes de taille croissante reconnus par l'automate A (avec un ensemble de transitions Δ), comme on le voit sur la figure suivante.



Afin d'obtenir un automate *fini* reconnaissant un sur-ensemble de $\mathcal{R}^*(\mathcal{L}(A))$, l'idée intuitive est de *replier* une cascade d'appels récursifs dans un unique nouvel état. Dans l'exemple précédent, au cours de l'étape 1, en appliquant la règle $app(cons(x, y), z) \to cons(x, app(y, z))$ sur $app(cons(q_2, q_1), q_0)$, nous avons obtenu la configuration $cons(q_2, app(q_1, q_0))$, et nous avons créé un nouvel état q_3 reconnaissant le sous-terme $app(q_1, q_0)$. Au cours de l'étape 2, nous avons appliqué *la même règle* sur le sous-terme $app(q_1, q_0)$ reconnu par q_3. Afin de replier cet appel récursif au cours de l'étape 2, nous *réutilisons* l'état q_3, au lieu de créer un nouvel état q_4 pour normaliser la transition $cons(q_2, app(q_3, q_0)) \to q_3$ obtenue au cours de l'étape 2. Ainsi, nous obtenons un ensemble de transitions normalisées $\{cons(q_2, q_3) \to q_3, app(q_3, q_0) \to q_3\}$ à ajouter à Δ. Plus aucun autre état ni aucune transition n'est à ajouter et cet automate reconnaît un sur-ensemble de $\mathcal{R}^*(\mathcal{L}(A))$. Ceci est une des idées essentielles de l'approximation de type *ancêtre* que nous présentons, maintenant, de façon plus formelle.

Soit $q' \in \mathcal{Q}_{new}$ un état utilisé pour normaliser une nouvelle transition $r\sigma \to q$ tel que $q \in \mathcal{Q}$. On dira que l'état q' est un *fils* de q. De la même façon, si on utilise un état q'' pour normaliser une nouvelle transition $r'\sigma' \to q'$, alors q'' est un fils de q'. A l'inverse, on dira que q est le *pére* de q', q' est le père de q'' et que q est l'*ancêtre* de q' et de q''. Tout état $q \in \mathcal{Q}' = \mathcal{Q} \cup \mathcal{Q}_{new}$ a un *ancêtre* unique $q_a \in \mathcal{Q}$. L'ancêtre de n'importe quel état $q \in \mathcal{Q}$ est q lui-même, et l'ancêtre de n'importe quel état $q' \in \mathcal{Q}_{new}$ apparaissant dans la séquence $\gamma(l \to r, q, \sigma)$ (utilisée pour normaliser une nouvelle transition $r\sigma \to q$ avec $q \in \mathcal{Q}'$), est l'ancêtre de q. Dans l'approximation de type ancêtre, (1) la fonction γ ne dépend pas de son paramètre σ et, (2) pour tous les fils $q_1 \ldots q_k$, d'un état q, obtenus par une règle $l \to r$, la valeur de la fonction d'approximation (pour $q_1, \ldots, q_k$) pour $l \to r$ est la même que celle de leur père, i.e. celle de q. Par transitivité, tout état a nécessairement la même approximation que son ancêtre.

Définition 6.8 *Une fonction d'approximation γ est appelée fonction d'approximation de type ancêtre si*

1. $\forall l \to r \in \mathcal{R}$, $\forall q \in \mathcal{Q}'$, $\forall \sigma_1, \sigma_2 \in \Sigma(\mathcal{Q}', \mathcal{X})$,

$$\gamma(l \to r, q, \sigma_1) = \gamma(l \to r, q, \sigma_2), \text{ et}$$

2. $\forall l_1 \to r_1, l_2 \to r_2 \in \mathcal{R}, \forall q \in \mathcal{Q}', \forall q_1, \ldots, q_k \in \mathcal{Q}_{new}, \sigma_1, \sigma_2 \in \Sigma(\mathcal{Q}', \mathcal{X})$,

$$\gamma(l_1 \to r_1, q, \sigma_1) = q_1 \ldots q_k \Rightarrow \forall i = 1 \ldots k, \gamma(l_2 \to r_2, q_i, \sigma_2) = \gamma(l_2 \to r_2, q, \sigma_2).$$

Il faut remarquer que dans le cas particulier de l'exemple 6.4, en utilisant l'approximation de type ancêtre, nous aurions obtenu $\gamma(rl, q_1, \sigma) = q_3$, et par le cas 2 de la définition 6.8 page 121 on obtient $\gamma(rl, q_3, \sigma') = \gamma(rl, q_1, \sigma')$. Puis par le cas 1, on obtient que $\gamma(rl, q_1, \sigma') = \gamma(rl, q_1, \sigma) = q_3$, ainsi $\gamma(rl, q_3, \sigma') = q_3$, et la construction de $\mathcal{T}_\mathcal{R} \uparrow (A)$ devient finie.

Théorème 6.3 *Tout automate approximation construit avec une fonction d'approximation de type ancêtre est fini.*

Preuve (idée) L'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ est fini si l'ensemble des nouveaux états $\mathcal{Q}_{new}$ est lui-même fini. Comme $\mathcal{Q}$ est fini, $\mathcal{R}$ est fini, et γ ne dépend pas de son paramètre σ, il existe un nombre fini de séquences distinctes $\gamma(l \to r, q, \sigma)$ pour $l \to r \in \mathcal{R}$, $q \in \mathcal{Q}$, et ces séquences sont finies. D'autre part, tout état $q' \in \mathcal{Q}_{new}$ a un unique ancêtre $q \in \mathcal{Q}$, et $\gamma(l \to r, q', \sigma) = \gamma(l \to r, q, \sigma)$. Donc, il existe un nombre fini de séquences distinctes $\gamma(l \to r, q', \sigma) = q'_1 \ldots q'_n$ avec $q', q'_1, \ldots, q'_n \in \mathcal{Q}_{new}$. Donc, un nombre fini d'états est utilisé pour normaliser les transitions, d'où $\mathcal{Q}_{new}$ est fini. Voir la section 6.7.3 pour une preuve détaillée. $\square$

6.5 Comparaison et limites

6.5.1 Comparaison avec l'approximation de la relation de réécriture

F. Jacquemard a proposé dans [Jac96b] un algorithme permettant de calculer exactement les $\mathcal{R}$-descendants pour tout ensemble régulier E et pour tout système $\mathcal{R}$ "décroissant", i.e. le membre droit de chaque règle de $\mathcal{R}$ est soit une variable, soit un terme $f(t_1, \ldots, t_n)$ où $f \in \mathcal{F}$, $ar(f) = n$, et $\forall i = 1, \ldots, n$, t_i est une variable, un terme clos, ou un terme dont les variables n'apparaissent pas dans le membre gauche. Pour tous ces systèmes, le calcul est exact.

Toujours dans [Jac96b], F. Jacquemard a proposé une méthode d'approximation des $\mathcal{R}$-descendants de E lorsque $\mathcal{R}$ n'est pas décroissant. Cette méthode est basée sur l'approximation de la relation de réécriture elle-même, i.e. en approchant un système $\mathcal{R}$ non-décroissant par un système $\mathcal{R}_{ap}$ décroissant, l'ensemble $\mathcal{R}^*_{ap}(E)$ est un sur-ensemble régulier de $\mathcal{R}^*(E)$. La méthode d'approximation des systèmes utilisée dans [Jac96b] est la suivante: tout système de réécriture $\mathcal{R} = \{l_1 \to r_1, \ldots, l_n \to r_n\}$ linéaire à gauche, est transformé en un système $\mathcal{R}_{ap} = \{l_1 \to r'_1, \ldots, l_n \to r'_n\}$ où pour tout $i = 1 \ldots n$, r'_i est obtenu par renommage des variables de r_i apparaissant dans l_i et dont la profondeur dans r_i est supérieure à 1. Voici ce que cela donne sur un exemple simple.

Exemple 6.5 *Soit $\mathcal{R}$ le système de réécriture suivant:*

$$append(nil, x) \to x$$
$$append(cons(x, y), z) \to cons(x, append(y, z)).$$

Le système $\mathcal{R}_{ap}$ approchant $\mathcal{R}$ est le suivant:

$$append(nil, x) \to x$$
$$append(cons(x, y), z) \to cons(x, append(u, v)).$$

où u et v sont les nouvelles variables renommant y et z.

On peut remarquer que ces systèmes ne répondent pas à la définition classique des systèmes de réécriture qui veut que pour toute règle $l \to r$ on ait $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. Cependant, dans le calcul proposé par F. Jacquemard, il est possible de tenir compte de telles règles. Lorsque l'on applique une règle, comportant des variables $x_1, \ldots, x_n$ figurant dans le membre droit de la règle mais pas dans le membre gauche, on suppose que ces variables peuvent être instanciées par n'importe quel terme de l'algèbre initiale $\mathcal{T}(\mathcal{F})$ [11]. Ainsi, dans le calcul des $\mathcal{R}$-descendants, lorsqu'une telle règle est appliquée, les variables $x_1, \ldots, x_n$ sont instanciées par un état particulier $q_{\mathcal{T}(\mathcal{F})}$ qui reconnaît toute l'algèbre $\mathcal{T}(\mathcal{F})$.

Pour les systèmes auxquels nous nous intéressons, cette approximation se révèle être trop forte, elle ne répond pas à nos besoins. En effet, la majorité des systèmes que nous utilisons sont des programmes ou des spécifications comportant des appels récursifs. Or, dans le cas de ces systèmes, l'approximation remplace les variables des appels récursifs (généralement situés dans des sous-termes stricts du membre droit) par des nouvelles variables, i.e. par n'importe quel terme de $\mathcal{T}(\mathcal{F})$. C'est le cas de l'exemple précédent. Nous donnons, maintenant, le résultat de l'approximation obtenue avec le système $\mathcal{R}_{ap}$ de l'exemple précédent.

Exemple 6.6 *Soit $\mathcal{R}_{ap}$ le système*

11. Dans [Jac96b], le calcul proposé utilise les règles de réécriture dans le sens inverse et débute d'un ensemble de terme initiaux qui est $IRR(\mathcal{R})$. Cet ensemble ayant des propriétés supplémentaires, il est possible de restreindre les possibilités d'instanciation aux termes de $IRR(\mathcal{R})$. Ce n'est pas le cas ici, il nous faut donc considérer tout $\mathcal{T}(\mathcal{F})$.

$append(nil, x) \rightarrow x$
$append(cons(x, y), z) \rightarrow cons(x, app(u, v))$.

Maintenant, nous calculons l'ensemble des $\mathcal{R}_{ap}$-descendants en appliquant le calcul de F. Jacquemard à l'automate d'arbres $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ tel que $\mathcal{F} = \{append : 2, cons : 1, nil : 0, a : 0, b : 0\}$, $\mathcal{Q} = \{q_0, q_1, q_2\}$, $\mathcal{Q}_f = \{q_0\}$ et $\Delta = \{$

$$\begin{aligned} append(q_1, q_1) &\rightarrow q_0 \\ nil &\rightarrow q_1 \\ cons(q_2, q_1) &\rightarrow q_1 \\ a &\rightarrow q_2 \\ b &\rightarrow q_2 \ \ \}. \end{aligned}$$

L'ensemble des termes initiaux auxquels nous nous intéressons ici, i.e. $\mathcal{L}(A)$ est l'ensemble des termes de la forme $append(l_1, l_2)$ où l_1 et l_2 sont des listes plates composées de "a" et de "b". Le résultat du calcul de $\mathcal{R}^*_{ap}(\mathcal{L}(A))$ est l'automate $A' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ où $\mathcal{Q}' = \mathcal{Q} \cup \{q_3, q_{\mathcal{T}(\mathcal{F})}\}$ et $\Delta' = \Delta \cup \{$

$$\begin{aligned} append(q_1, q_1) &\rightarrow q_0 \\ nil &\rightarrow q_1 \\ cons(q_2, q_1) &\rightarrow q_1 \\ a &\rightarrow q_2 \\ b &\rightarrow q_2 \\ \\ nil &\rightarrow q_0 \\ cons(q_2, q_1) &\rightarrow q_0 \\ cons(q_2, q_3) &\rightarrow q_0 \\ append(q_{\mathcal{T}(\mathcal{F})}, q_{\mathcal{T}(\mathcal{F})}) &\rightarrow q_3 \\ nil &\rightarrow q_3 \\ cons(q_{\mathcal{T}(\mathcal{F})}, q_{\mathcal{T}(\mathcal{F})}) &\rightarrow q_3 \\ a &\rightarrow q_3 \\ b &\rightarrow q_3 \\ append(q_{\mathcal{T}(\mathcal{F})}, q_{\mathcal{T}(\mathcal{F})}) &\rightarrow q_{\mathcal{T}(\mathcal{F})} \\ nil &\rightarrow q_{\mathcal{T}(\mathcal{F})} \\ cons(q_{\mathcal{T}(\mathcal{F})}, q_{\mathcal{T}(\mathcal{F})}) &\rightarrow q_{\mathcal{T}(\mathcal{F})} \\ a &\rightarrow q_{\mathcal{T}(\mathcal{F})} \\ b &\rightarrow q_{\mathcal{T}(\mathcal{F})} \ \ \} \end{aligned}$$

où $\mathcal{L}(A', q_{\mathcal{T}(\mathcal{F})}) = \mathcal{T}(\mathcal{F})$. Comme il est difficile d'estimer l'erreur directement sur $\mathcal{R}^*_{ap}(\mathcal{L}(A))$, nous réalisons ici une intersection entre cet automate et l'automate $A_{IRR(\mathcal{R})}$ reconnaissant l'ensemble des termes irréductibles pour $\mathcal{R}$ afin de calculer l'approximation de $\mathcal{R}^!(\mathcal{L}(A))$. Cet ensemble $\mathcal{R}^!(\mathcal{L}(A))$ est l'ensemble des listes plates de "a" et de "b". L'intersection entre l'approximation et $A_{IRR(\mathcal{R})}$ est calculée à l'aide du prototype qui sera décrit en détail dans la section 7.2. Le résultat de l'intersection est l'automate nommé $A(1)$, dans lequel on trouve tout d'abord un ensemble d'états $\{q_0, \ldots, q_5\}$ (représenté par la liste d'états `q|0.q|1.q|2.q|3.q|4.q|5.nil`*), puis vient ensuite l'ensemble d'états finaux, réduit ici à $\{q_5\}$. Enfin, on trouve la liste des transitions de l'automate.*

```
[] result term:
    Description of A(1) states q|0.q|1.q|2.q|3.q|4.q|5.nil final states
q|5.nil transitions append(q|0,q|1)->q|0.b->q|0.a->q|0.nil->q|1.nil->q|2.
```

```
nil->q|4.nil->q|5.append(q|0,q|1)->q|1.append(q|0,q|1)->q|2.cons(q|1,q|1)
->q|1.cons(q|1,q|1)->q|2.cons(q|3,q|2)->q|5.cons(q|3,q|4)->q|4.cons(q|3,q|4)
->q|5.b->q|1.b->q|2.b->q|3.a->q|1.a->q|2.a->q|3.nil End of Description
```

Le langage $\mathcal{L}(A(1))$ contient bien $\mathcal{R}^!(\mathcal{L}(A))$ mais il contient également des termes mal structurés, par exemple des termes de la forme $cons(a, append(a, cons(nil, nil)))$.

Maintenant, sur ce même exemple, nous donnons l'automate d'approximation ancêtre $\mathcal{T}_\mathcal{R} \uparrow (A)$ tel qu'il est présenté dans la section 6.4. Voici l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ tel qu'il peut être calculé par notre prototype:

```
[] result term:
    Description of A(2)states q|4.q|0.q|1.q|2.nil final states q|0.nil
transitions cons(q|2,q|4)->q|4.append(q|1,q|1)->q|4.cons(q|2,q|4)->q|0.
append(q|1,q|1)->q|0.cons(q|2,q|1)->q|1.cons(q|2,q|1)->q|4.cons(q|2,q|1)->
q|0.nil->q|1.nil->q|4.nil->q|0.a->q|2.b->q|2.nil End of Description
```

Comme dans le cas de l'approximation précédente, on réalise une intersection avec l'automate $A_{IRR(\mathcal{R})}$ reconnaissant l'ensemble des termes irréductibles par $\mathcal{R}$ et l'on obtient l'automate $A(3)$ suivant:

```
[] result term:
    Description of A(3) states q|0.q|1.q|2.q|3.nil final states q|3.nil
transitions b->q|1.a->q|1.nil->q|3.nil->q|2.nil->q|0.cons(q|1,q|0)->q|3.
cons(q|1,q|0)->q|2.cons(q|1,q|0)->q|0.cons(q|1,q|2)->q|3.cons(q|1,q|2)->q|2.
nil End of Description
```

Cette fois-ci le langage reconnu, i.e. $\mathcal{L}(A(3))$ est exactement *l'ensemble des listes plates de composées de "a" et de "b".*

Grâce à son principe d'approximation basé sur la recherche et le repliage des "appels récursifs", la technique d'approximation de la section 6.4 semble donner, en général, de meilleurs résultats sur les systèmes récursifs que l'approximation de la relation de réécriture telle qu'elle est proposée dans [Jac96b]. Dans l'exemple précédent, le calcul de $\mathcal{T}_\mathcal{R} \uparrow (A)$ se révèle être, en fait, exact. Cependant, ce n'est pas le cas en général comme on le voit dans la section suivante avec les limites de cette méthode.

6.5.2 Limites de la méthode

Les limites de la méthode sont d'abord liées au fait que nous ne manipulons ici *que* des ensembles réguliers de termes: les ensembles de termes initiaux et les ensembles de descendants. En particulier, il est impossible de représenter tout ensemble de termes dans lequel les hauteurs de deux termes sont liées par une quelconque contrainte arithmétique. Par exemple, il est impossible d'exprimer un ensemble de requêtes de la forme $E{=}\{leq(s^n(0), s^m(0)) \mid n, m \in \mathbb{N} \text{ et } n \leq m\}$. Ainsi, montrer que l'ensemble des $\mathcal{R}$-formes normales de E avec $\mathcal{R} = \{$

$$
\begin{array}{l}
leq(0,0) \to true \\
leq(0,s(x)) \to true \\
leq(s(x),0) \to false \\
leq(s(x),s(y)) \to leq(x,y)\}
\end{array}
$$

est $\mathcal{R}^!(E) = \{true\}$ n'est pas réalisable à l'aide de langages régulier, mais est plutôt du domaine de la preuve par induction. Une autre limite importante de la méthode est qu'il s'agit, en général, d'une approximation. Ainsi certaines caractéristiques de l'ensemble des descendants ne sont pas nécessairement capturées par l'approximation. En voici un exemple simple.

Exemple 6.7 *Soit $\mathcal{F}$ la signature $\mathcal{F} = \{0 : 0, equal : 2, s : 1, cons : 2, nil : 0, member : 2, true : 0, false : 0, if : 3\}$ et $\mathcal{R}$ le système de réécriture suivant définissant le prédicat member qui vérifie l'appartenance d'un entier à une liste d'entiers:*

$$\begin{array}{l}
equal(0,0) \to true\\
equal(s(x),0) \to false\\
equal(0,s(y)) \to false\\
equal(s(x),s(y)) \to equal(x,y)\\
if(true,x,y) \to x\\
if(false,x,y) \to y\\
member(x,cons(y,z)) \to if(equal(x,y),true,member(x,z))\\
member(x,nil) \to false
\end{array}$$

Soit A l'automate d'arbres $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ tel que $\mathcal{Q} = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $\mathcal{Q}_f = \{q_0\}$ et $\Delta = \{$

$$\begin{array}{r}
member(q_1,q_2) \to q_0\\
0 \to q_1\\
cons(q_3,q_2) \to q_2\\
cons(q_4,q_5) \to q_2\\
cons(q_3,q_5) \to q_5\\
nil \to q_5\\
0 \to q_3\\
s(q_3) \to q_3\\
0 \to q_4\ \}
\end{array}$$

dont le langage reconnu $\mathcal{L}(A)$ est l'ensemble des termes $member(0,l)$ où l est toute liste d'entier contenant au moins un entier égal à 0. Or si l'on calcule l'approximation de $\mathcal{R}^!(\mathcal{L}(A))$, au lieu de l'ensemble $\{true\}$, on obtient un automate nommé $A(0)$

```
[] result term:
    Description of A(0)states q|0.nil final states q|0.nil transitions
true->q|0.false->q|0.nil End of Description
```

qui reconnaît $\{true, false\}$.

Ce dernier exemple, illustre bien le fait qu'il s'agit d'une approximation et qu'il est donc possible d'obtenir des résultats approximatifs. Cependant, lorsque l'approximation n'est pas assez fine, il doit être possible d'affiner le calcul. En effet, nous rappelons que l'automate d'approximation est paramètré par une fonction d'approximation. Dans cette thèse, nous nous sommes limités à l'utilisation d'une fonction d'approximation très simple et donc très brutale: la fonction d'approximation ancêtre. Comme on l'a vu dans la section 6.1, l'ensemble des descendants n'est pas un langage régulier en général. Mais à l'image de l'exemple précédent, quand l'ensemble des descendants est un langage régulier, nous pensons qu'en utilisant d'autres fonctions d'approximation, il est possible d'obtenir une approximation plus fine, voire exacte. Cependant c'est un point que nous n'aborderons pas dans cette thèse.

Enfin, comme nous le voyons dans la section suivante, bien que l'approximation ancêtre soit une approximation relativement brutale, elle permet néanmoins de démontrer automatiquement des propriétés intéressantes sur des systèmes de réécriture représentant des programmes ou spécifiant le comportement de systèmes complexes.

6.6 Applications directes à la vérification de programmes et de systèmes

Pour un système de réécriture $\mathcal{R}$ donné et pour un ensemble de termes initiaux $E \subseteq \mathcal{T}(\mathcal{F})$, nous rappelons que l'ensemble $\mathcal{R}^*(E)$ contient tous les termes t atteignables à partir des termes de E, i.e. tous les termes t tels que $s \in E$ et $s \rightarrow_{\mathcal{R}}{}^* t$. Du point de vue de la programmation, les termes de $\mathcal{R}^*(E)$ représentent tous les calculs intermédiaires possibles avec le programme $\mathcal{R}$ à partir des requêtes de E. Dans la section 6.4, nous avons montré comment, à partir d'un système de réécriture linéaire à gauche et d'un automate A, construire un automate $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ reconnaissant un sur-ensemble régulier de $\mathcal{R}^*(\mathcal{L}(A))$. Dans cette construction, en réécrivant les configurations de l'automate, nous schématisons et approchons la réécriture d'ensembles de termes. Si $\mathcal{R}$ est un programme et l'ensemble $\mathcal{L}(A)$ est un ensemble de requêtes, en réécrivant les configuration de A lors du calcul de $\mathcal{T}_{\mathcal{R}} \uparrow (A)$, on réalise une forme de test sur $\mathcal{R}$. Ce test est à la fois *approximatif* et *exhaustif*; approximatif parce que dans $\mathcal{L}(\mathcal{T}_{\mathcal{R}} \uparrow (A))$ on prend en compte des termes (i.e. des calculs intermédiaires) qui ne sont pas réellement atteignables; exhaustif parce que, contrairement au test classiquement pratiqué sur les programmes, on prend en compte tous les comportements possibles à partir de schémas de requêtes donnés sous la forme de l'automate A.

A partir de ces observations, et de l'automate $\mathcal{T}_{\mathcal{R}} \uparrow (A)$, il est possible d'effectuer différents types de vérifications: le test d'atteignabilité, l'approximation de co-domaine, la preuve de non-terminaison forte, la preuve de terminaison de la relation de réduction séquentielle, et la preuve de suffisante complétude. Dans les sections suivantes, nous décrivons chacune de ces vérifications et nous donnons quelques exemples. Tous les exemples figurant dans la suite de cette section sont donnés dans le format utilisé par notre prototype: donnée d'un ensemble de variables, d'une signature, d'un système de réécriture linéaire à gauche, et d'une liste d'automates (souvent réduite à un seul automate).

6.6.1 Test d'atteignabilité

Dans le cadre de la programmation par réécriture ou de la spécification de système, le *test d'atteignabilité* permet de vérifier qu'un comportement donné ne peut se produire. Si l'on est capable de représenter le comportement proscrit sous la forme d'un terme ou d'un ensemble de termes, ceci revient à vérifier que tous les termes proscrits sont inatteignables. Puisque $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ reconnaît un sur-ensemble des termes atteignables, une condition suffisante pour montrer ceci est que pour tout terme t proscrit on a $t \notin \mathcal{L}(\mathcal{T}_{\mathcal{R}} \uparrow (A))$. Nous définissons maintenant la notion d'atteignabilité plus formellement.

Définition 6.9 *Soit $\mathcal{R}$ un système de réécriture, $E \subseteq \mathcal{T}(\mathcal{F})$ et $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Le* pattern *$t$* est *$\mathcal{R}$*-atteignable à partir de *E s'il existe un contexte clos $C[\,]$, un terme $s \in E$, et une substitution σ telle que $s \rightarrow_{\mathcal{R}}^* C[t\sigma]$.*

En pratique, afin de vérifier qu'un pattern t n'est pas $\mathcal{R}$-atteignable à partir d'un ensemble de termes initiaux reconnu par un automate A, une première méthode consiste à calculer $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ puis simplement à rechercher une substitution σ et un état $q \in \mathcal{Q}$ tel que $t\sigma \rightarrow_{\Delta}^* q$, où $\mathcal{Q}$ et Δ sont respectivement les ensembles d'états et ensembles de transitions de $\mathcal{T}_{\mathcal{R}} \uparrow (A)$. Nous avons vu que la recherche de telles substitutions est assurée par l'algorithme de filtrage proposé dans la section 6.2.3. Prouver que t n'est pas $\mathcal{R}$-atteignable revient donc vérifier que le problème de filtrage suivant n'a pas de solution:

$$\bigvee_{q \in \mathcal{Q}} t \trianglelefteq q$$

Voici maintenant deux exemples de ce qui peut être fait à l'aide de notre implantation. Le premier exemple est un système de réécriture calculant la valeur d'un arrangement: $A_n^p = \frac{n!}{(n-p)!}$.

```
specification Anp

Vars    x y n p

Ops
    Ar:2 minus:2 div:2 o:0 s:1 fact:1 plus:2 mult:2

R1
    Ar(n, p) -> div(fact(n), fact(minus(n, p)))
    fact(s(x)) -> mult(s(x), fact(x))
    fact(o) -> s(o)
    mult(o, x) -> o
    mult(s(x), y) -> plus(mult(x, y), y)
    div(o, s(y)) -> o
    div(s(x), s(y)) -> s(div(minus(x, y), s(y)))
    plus(x, o) -> x
    plus(x, s(y)) -> s(plus(x, y))
    minus(x, o) -> x
    minus(o, x) -> o
    minus(s(x), s(y)) -> minus(x, y)
    nil

Automata
   Description of A(0)
   states q|0.q|1.nil
   final states q|0.nil
   transitions Ar(q|1, q|1) -> q|0
               o-> q|1
               s(q|1) -> q|1
               nil
   End of Description
   nil
end of specification
```

L'automate $A(0)$ décrit les requêtes sur lesquelles nous souhaitons effectuer notre vérification: il s'agit de toutes les requêtes de la forme $\mathcal{L}(A(0)) = \{Ar(n,p) \quad | \quad n,p \in Nat\}$, où $Nat = \{0, s(0), \dots\}$. Sur ce système et cet ensemble de termes de départ, nous souhaitons vérifier qu'aucune division par 0 ne peut se produire. Dans le codage de la division par l'opérateur `div`, le dénominateur est le deuxième argument. Vérifier qu'il ne peut y avoir de division par 0 revient donc à s'assurer qu'il n'existe pas de pattern `div(x, o)` dans l'automate $\mathcal{T}_{\mathcal{R}_1} \uparrow (A(0))$. C'est ce que nous faisons maintenant. Tout d'abord, nous calculons $\mathcal{T}_{\mathcal{R}_1} \uparrow (A(0))$ à l'aide de notre implantation, nous obtenons un automate nommé $A(1)$ tel que $A(1) = \mathcal{T}_{\mathcal{R}_1} \uparrow (A(0))$:

```
[] start with term :
    T_up(R1) on (!A(0))

[] result term:
    Description of A(1) states q|12.q|13.q|11.q|10.q|9.q|8.q|6.q|7.q|2.
q|5.q|3.q|0.q|1.nil final states q|0.nil transitions
s(q|11)->q|11.o->q|11.minus(q|10,q|10)->q|12.minus(q|8,q|10)->q|12.
s(q|10)->q|13.s(q|10)->q|3.minus(q|10,q|8)->q|12.minus(q|8,q|8)->q|12.
s(q|8)->q|13.div(q|12,q|13)->q|11.s(q|11)->q|0.plus(q|9,q|7)->q|3.s(q|8)->
```

```
q|3.mult(q|6,q|7)->q|3.plus(q|9,q|10)->q|10.plus(q|9,q|10)->q|12.s(q|10)->
q|7.s(q|10)->q|10.s(q|10)->q|12.s(q|10)->q|9.plus(q|9,q|8)->q|10.
plus(q|9,q|8)->q|12.s(q|10)->q|2.plus(q|9,q|7)->q|2.plus(q|9,q|7)->q|9.
plus(q|9,q|7)->q|10.plus(q|9,q|7)->q|12.o->q|9.o->q|10.o->q|12.
mult(q|1,q|7)->q|9.mult(q|1,q|7)->q|10.mult(q|1,q|7)->q|12.plus(q|9,q|7)->
q|7.s(q|8)->q|2.o->q|8.s(q|8)->q|7.mult(q|6,q|7)->q|7.s(q|1)->q|6.
fact(q|1)->q|7.mult(q|6,q|7)->q|2.fact(q|1)->q|2.minus(q|1,q|1)->q|5.
fact(q|5)->q|3.div(q|2,q|3)->q|0.Ar(q|1,q|1)->q|0.o->q|1.o->q|5.
s(q|1)->q|1.s(q|1)->q|5.nil
End of Description
```

Ensuite, si nous lançons l'algorithme de filtrage sur ce dernier automate, nommé $A(1)$, et que nous recherchons le pattern `div(x, o)` dans tous les états de $A(1)$, nous obtenons:

```
[] start with term :
    (div(x, o) ?= states) with (!A(1))

[] result term:
    nil
```

signifiant qu'il n'existe pas de substitution σ telle que $div(x, 0)\sigma$ soit reconnu par un état de $A(1)$. Donc aucune division par 0 ne peut se produire à partir des requêtes données dans $\mathcal{L}(A(0))$.

Toujours pour le test d'atteignabilité, voici un deuxième exemple illustrant une deuxième méthode de test. Dans cet exemple, au lieu de rechercher un pattern, nous décrivons sous la forme d'un langage régulier les termes proscrits. Ensuite, nous montrons que l'intersection entre $\mathcal{T}_{\mathcal{R}_1} \uparrow (A)$ et cet automate est vide. Le système de réécriture suivant spécifie le comportement d'un système de comptage d'objets, distribué sur deux processus parallèles P_1 et P_2 recevant séparément des objets et incrémentant pour chaque objet reçu un compteur partagé. Le compteur partagé est considéré comme une section critique et protégé par un système de sémaphores simplifié.

Nous décrivons brièvement le codage des processus, du compteur et des sémaphores: `Proc(état, données)` représente un processus qui peut prendre deux "états" `free` ou `busy` selon qu'il accède ou n'accède pas à la section critique. Le champs "données" du processus contient une pile d'objets à traiter. Les objets, représentés par des `o`, arrivent dans la pile du processus de façon indéterminée. Les piles d'objets sont représentées par des listes construites sur les symboles usuels `null` et `cons`. Les entiers sont construit sur `o` et `s`. L'opérateur `S(`P_1`, `P_2`, statut, compteur)` représente l'état (ou une configuration) du système à un instant donné, i.e. les deux processus P_1, P_2, le statut de la section critique `free` ou `busy` et enfin la valeur du compteur partagé. Les cinq premières règles du système $\mathcal{R}_1$ suivant décrivent les transitions possibles de ce système. Les deux dernières règles codent le fait que l'arrivée des objets dans la pile du processus est indéterminée.

Il faut également noter qu'un processus qui ne reçoit plus d'objets, patiente jusqu'à l'arrivée de nouvelles données, puis reprend son activité de comptage dès qu'un nouvel objet arrive dans sa pile. L'attente des processus est codée, entre autres, par la cinquième règle de $\mathcal{R}_1$. On peut noter que les trois dernières règles rendent ce système de réécriture radicalement non-terminant.

```
specification 2processes

Vars    x y z u
Ops
        S:4 Proc:2 cons:2 null:0 busy:0 free:0 o:0 s:1

R1
  S(Proc(free, cons(x, y)), z, free, u) -> S(Proc(busy, cons(x, y)), z, busy, u)
```

```
  S(Proc(busy, cons(x, y)), z, busy, u) -> S(Proc(free, y), z, free, s(u))
  S(z,Proc(free, cons(x, y)), free, u) -> S(z, Proc(busy, cons(x, y)), busy, u)
  S(z, Proc(busy, cons(x, y)), busy, u) -> S(z, Proc(free, y), free, s(u))
  S(Proc(x, null), Proc(y, null), z, u) -> S(Proc(x, null), Proc(y, null), z, u)
  cons(x, y) -> cons(o, cons(x, y))
  null -> cons(o, null)
nil

Automata
        Description of A(0)
                states q|0 q|1 q|2 q|3 q|4 nil
                final states q|0 nil
                transitions
                        S(q|1, q|1, q|2, q|3) -> q|0
                        free -> q|2
                        o -> q|3
                        Proc(q|2, q|4) -> q|1
                        null -> q|4
                        cons(q|3, q|4) -> q|4
                        nil
        End of Description
        nil
end of specification
```

L'automate $A(0)$, quant à lui, décrit la configuration initiale du système dans laquelle le compteur partagé est à zéro, les statuts des processus et de la section critique sont à `free`, et les listes de données des processus P_1 et P_2 ont un nombre quelconque d'objets en attente.

Classiquement sur ce genre de système, on souhaite assurer l'*exclusion mutuelle*, i.e. interdire que les processus P_1 et P_2 accèdent en même temps à la section critique. Ceci peut être fait en montrant qu'aucune configuration dans laquelle P_1 et P_2 ont un statut `busy` n'est atteignable. Soit $A(1)$ l'automate obtenu par calcul de $\mathcal{T}_{\mathcal{R}_1} \uparrow (A(0))$:

```
[] result term:
    Description of A(1)states q|20.q|21.q|17.q|18.q|15.q|11.q|13.q|14.q|9.
q|10.q|5.q|7.q|0.q|1.q|2.q|3.q|4.nil final states q|0.nil transitions
S(q|1,q|17,q|18,q|3)->q|0.busy->q|20.cons(q|3,q|4)->q|21.Proc(q|20,q|21)->
q|17.busy->q|18.S(q|11,q|17,q|18,q|14)->q|0.s(q|14)->q|14.S(q|5,q|1,q|7,q|14)
->q|0.free->q|15.Proc(q|15,q|4)->q|11.free->q|13.s(q|3)->q|14.S(q|11,q|1,
q|13,q|14)->q|0.busy->q|9.cons(q|3,q|4)->q|10.Proc(q|9,q|10)->q|5.busy->q|7.
S(q|5,q|1,q|7,q|3)->q|0.S(q|1,q|1,q|2,q|3)->q|0.free->q|2.o->q|3.
Proc(q|2,q|4)->q|1.null->q|4.cons(q|3,q|4)->q|4.nil End of Description
```

Au lieu de rechercher un pattern dans cet automate, nous montrons qu'il est également possible de décrire le langage de termes proscrits à l'aide d'un automate, nommé ici $A(2)$, et de vérifier que l'intersection entre $A(1)$ et $A(2)$ est vide. Soit $A(2)$ l'automate suivant

```
Description of A(2)
                states q|0.q|1.q|2.q|3.q|4.q|5.q|6.nil
                final states q|0 nil
                transitions
                        S(q|1, q|1, q|2, q|3) -> q|0
                        busy -> q|2
                        free -> q|2
                        o -> q|3
                        s(q|3) -> q|3
```

```
                Proc(q|5, q|4) -> q|1
                null -> q|4
                cons(q|6, q|4) -> q|4
                busy -> q|5
                o -> q|6
        End of Description
```

reconnaissant le langage des termes proscrits de la forme $S(Proc(l_1, busy), Proc(l_2, busy), s, c)$ où l_1, l_2 sont des listes quelconques d'objets, $s \in \{\texttt{busy},\texttt{free}\}$ et c est un entier. Le résultat de l'intersection des automates $A(1)$ et $A(2)$ donne:

```
[] result term:
    Description of nil states nil final states nil transitions nil
End of Description
```

Le résultat de l'intersection, après nettoyage par test d'accessibilité (voir section 2.5.3), est un automate dont les ensembles d'états, d'états finaux et de transitions sont vides: cet automate reconnaît $\emptyset$. Ce résultat montre qu'aucun terme proscrit n'est atteignable et donc que l'exclusion mutuelle est assurée sur le système précédent.

6.6.2 Approximation de co-domaine

Si l'on code une fonction f sous la forme d'un système de réécriture $\mathcal{R}$, et que l'on décrit un ensemble d'appels type (i.e. le domaine de la fonction) à l'aide d'un automate d'arbres A, il est possible d'obtenir un sur-ensemble du co-domaine de f en calculant l'automate $\mathcal{T}_{\mathcal{R}} \uparrow (A) \cap A_{IRR(\mathcal{R})}$ où $A_{IRR(\mathcal{R})}$ est l'automate reconnaissant les termes irréductibles par $\mathcal{R}$ (voir section 6.6.6). L'analyse de l'approximation du co-domaine de f, permet de signaler et parfois même de localiser des erreurs dans la définition de f. Par exemple, supposons que nous souhaitions coder une fonction d'aplatissement pour des listes imbriquées telles qu'on en rencontre en Lisp, i.e. des listes dans lesquelles peuvent cohabiter des atomes (ici des entiers) et d'autres listes, par exemple $[1, 2, [4, [5]], 5]$. Voici une première version de la fonction d'aplatissement de listes, ainsi que l'ensemble d'appels type sur lesquels on souhaite vérifier cette fonction.

```
specification flatten

Vars    x y z
Ops
        flatten:1 s:1 o:0 cons:2 append:2 null:0

R1
        flatten(null) -> null
        flatten(cons(x, y)) -> append(flatten(x), flatten(y))
        append(null, x) -> x
        append(cons(x, y), z) -> cons(x, append(y, z))
        nil

Automata
        Description of A(0)
                states q|0 q|1 q|2 nil
                final states q|0 nil
                transitions
                        flatten(q|1) -> q|0
                        cons(q|1, q|1) -> q|1
                        cons(q|2, q|1) -> q|1
                        null -> q|1
```

```
                    o -> q|2
                    s(q|2) -> q|2
                    nil
        End of Description
        nil
end of specification
```

L'automate $A(0)$ reconnaît les termes de la forme $flatten(l)$ où l est n'importe quelle liste imbriquée d'entiers (construits sur 0 et $s()$). Le co-domaine $\mathcal{R}_1^!(\mathcal{L}(A(0)))$ attendu de cette fonction est l'ensemble des listes plates d'entiers. Or, le lecteur attentif aura remarqué que la fonction telle qu'elle est codée par le système $\mathcal{R}_1$ ne répond pas à nos attentes. Si l'on calcule l'automate $\mathcal{T}_{\mathcal{R}_1} \uparrow (A(0)) \cap A_{IRR(\mathcal{R}_1)}$ [12] reconnaissant un sur-ensemble de $\mathcal{R}_1^!(\mathcal{L}(A(0)))$, on obtient:

```
[] result term:
    Description of nil states q|0.q|1.q|2.q|3.q|4.nil final states
q|4.nil transitions o->q|1.s(q|0)->q|1.null->q|4.null->q|3.append(q|2,q|3)->q|2.
append(q|2,q|3)->q|4.append(q|2,q|3)->q|3.o->q|0.s(q|0)->q|0.flatten(q|1)->q|2.nil
End of Description
```

et l'on peut remarquer que dans cet automate, on trouve la transition $flatten(q_1) \rightarrow q_2$. Cela signifie qu'il existe un terme, reconnu par l'état q_1 qui n'a pu être réduit par $\mathcal{R}_1$. Si l'on détaille les termes reconnus par q_1, il s'agit des entiers 0, $s(0), \ldots$ Ceci montre qu'avec la définition de $flatten$ donnée dans l'exemple précédent, et à partir des appels types, il semble que $flatten$ ait dû être appliquée à un entier. C'est clairement le cas lorsque l'on applique la deuxième règle du système à une liste non-imbriquée, par exemple le terme $flatten([1, 2])$, qui va être réécrit en $append(flatten(1), flatten([2]))$. Cette définition étant erronée, nous en proposons une deuxième:

```
specification flatten2

Vars    x y z
Ops
        flatten:1 s:1 o:0 cons:2 append:2 null:0

R1
        flatten(null) -> null
        flatten(cons(cons(x, y), z)) -> append(flatten(cons(x, y)), flatten(z))
        flatten(cons(o, y)) -> cons(o, flatten(y))
        flatten(cons(s(x), y)) -> cons(s(x), flatten(y))
        append(null, x) -> x
        append(cons(x, y), z) -> cons(x, append(y, z))
        nil
```

et nous considérons le même automate $A(0)$. Or, cette définition est encore erronée! L'automate d'approximation $\mathcal{T}_{\mathcal{R}_1} \uparrow (A(0)) \cap A_{IRR(\mathcal{R}_1)}$ va encore nous permettre de localiser l'erreur:

```
[] result term:
    Description of nil states q|0.q|1.q|2.q|3.q|4.q|5.q|6.q|7.q|8.q|9.
q|10.q|11.q|12.q|13.q|14.q|15.nil final states q|15.nil transitions
s(q|4)->q|4.o->q|4.null->q|1.null->q|0.cons(q|4,q|1)->q|1.cons(q|0,q|1)->
q|3.cons(q|1,q|1)->q|1.flatten(q|3)->q|15.null->q|15.append(q|5,q|11)->q|15.
flatten(q|3)->q|10.flatten(q|3)->q|14.flatten(q|3)->q|11.flatten(q|2)->q|5.
null->q|14.null->q|11.append(q|5,q|11)->q|10.append(q|5,q|11)->q|14.append(q|5,q|11)->
q|11.cons(q|0,q|1)->q|2.append(q|5,q|11)->q|5.flatten(q|3)->q|8.flatten(q|3)->
q|6.o->q|12.null->q|6.append(q|5,q|11)->q|8.append(q|5,q|11)->q|6.cons(q|12,q|6)->
```

12. où $A_{IRR(\mathcal{R}_1)}$ est l'automate reconnaissant les termes irréductibles par $\mathcal{R}_1$

```
q|6.cons(q|12,q|6)->q|14.cons(q|12,q|6)->q|11.cons(q|12,q|6)->q|15.
cons(q|13,q|7)->q|6.flatten(q|3)->q|9.flatten(q|3)->q|7.s(q|4)->q|13.null->
q|7.append(q|5,q|11)->q|9.append(q|5,q|11)->q|7.cons(q|12,q|6)->q|7.cons(q|13,q|7)
->q|7.cons(q|13,q|7)->q|14.cons(q|13,q|7)->q|11.cons(q|13,q|7)->q|15.
cons(q|12,q|14)->q|7.append(q|8,q|11)->q|10.append(q|8,q|11)->q|14.cons(q|12,q|14)
->q|14.cons(q|13,q|14)->q|14.append(q|9,q|11)->q|10.append(q|9,q|11)->q|14.
append(q|10,q|11)->q|10.append(q|10,q|11)->q|14.cons(q|13,q|14)->q|7.
cons(q|12,q|14)->q|6.cons(q|13,q|14)->q|6.cons(q|12,q|14)->q|11.
cons(q|13,q|14)->q|11.cons(q|12,q|14)->q|15.cons(q|13,q|14)->q|15.nil
End of Description
```

Là encore, on retrouve des configurations, par exemple $flatten(q_3)$, attestant que la réduction de ces symboles n'est pas complète. Or, une seule transition mène à q_3: il s'agit de $cons(q_0, q_1) \to q_3$. De la même façon, une seule transition mène à q_0: il s'agit de $null \to q_0$. Le schéma de terme posant problème est donc $flatten(cons(null, x))$ et c'est effectivement un cas qui n'est absolument pas pris en compte dans le système $\mathcal{R}_1$ précédent. Si l'on rajoute la règle

```
flatten(cons(null, x)) -> flatten(x)
```

et que l'on recalcule l'automate approximation, on obtient bien un automate qui reconnaît des listes plates d'entiers:

```
[] result term:
    Description of nil states q|0.q|1.q|2.q|3.q|4.q|5.nil final states
q|5.nil transitions s(q|0)->q|0.o->q|0.null->q|5.o->q|1.null->q|2.
cons(q|1,q|2)->q|2.cons(q|1,q|2)->q|5.cons(q|3,q|4)->q|2.s(q|0)->q|3.
null->q|4.cons(q|1,q|2)->q|4.cons(q|3,q|4)->q|4.cons(q|3,q|4)->q|5.nil
End of Description
```

On peut remarquer que, si l'on avait codé ces exemples à l'aide de réécriture typée, la première erreur (application de flatten à un entier) aurait pu être détectée par un système de vérification de type, et la deuxième erreur ($flatten$ indéfinie sur $cons(null, x)$) aurait sans doute été mise à jour par une vérification de complétude suffisante. Cependant, nous souhaitons attirer l'attention du lecteur sur le fait que typage et test de complétude suffisante ne couvrent pas tous les cas d'erreur détectables par notre méthode. Soit l'exemple simple suivant dans lequel on définit la fonction $reverse$ qui inverse une liste plate:

```
specification reverse

Vars    x y z
Ops
    a:0 b:0 rev:1 cons:2 append:2 null:0

R1
    rev(null) -> null
    rev(cons(x, y)) -> append(rev(y), cons(x, null))
    append(null, x) -> null
    append(cons(x, y), z) -> cons(x, append(y, z))
    nil

Automata
    Description of A(0)
    states q|0.q|1.q|2.nil
    final states q|0.nil
    transitions rev(q|1) -> q|0.
            cons(q|2, q|1) -> q|1.
```

```
            null -> q|1.
            a -> q|2.
            b -> q|2.
            nil
    End of Description
```

où $\mathcal{L}(A(0))$ reconnait tous les termes $rev(l)$ où l est n'importe quelle liste plate constituée de "a" et de "b". Ce système de réécriture est confluent et terminant. De plus, en supposant que l'on donne un type aux opérateurs, par exemple

$a :\mapsto \alpha$,
$b :\mapsto \alpha$,
$null :\mapsto list_\alpha$,
$cons : \alpha, list_\alpha \mapsto list_\alpha$,
$append : list_\alpha, list_\alpha \mapsto list_\alpha$, et
$rev : list_\alpha \mapsto list_\alpha$ (où α et $list_\alpha$ sont des types),

il est possible de montrer que le système $\mathcal{R}_1$ précédent préserve les types, i.e. tout terme correctement typé est nécessairement réécrit en un terme correctement typé. De plus, $\mathcal{R}_1$ est, dans ce cas, suffisamment complet, i.e. les fonctions *append* et *rev* sont complètement définies.

Cependant, la fonction *rev* telle qu'elle est définie par le système $\mathcal{R}_1$ n'a pas le comportement attendu. En effet, si l'on calcule le sur-ensemble régulier approchant $\mathcal{R}_1^!(\mathcal{L}(A(0)))$, on obtient:

```
[] result term:
   Description of A(1) states q|0.nil final states q|0.nil
   transitions null->q|0.nil End of Description
```

Cet automate nommé $A(1)$ reconnaît exactement le langage fini $\{null\}$. En d'autre termes, $\mathcal{R}_1^!(\mathcal{L}(A(0))) \subseteq \{null\}$. Ce n'est, a priori, pas ce qui est attendu de la fonction *rev*. Ceci est simplement dû à la mauvaise définition de *append*: le membre droit de la troisième règle devrait être x au lieu de *null*.

6.6.3 Preuve de non-terminaison

Une autre application originale du calcul de l'approximation de $\mathcal{R}^!(E)$ est la preuve de ce que nous appellerons ici la *non-terminaison forte* d'un système de réécriture. Soit $E \subset \mathcal{T}(\mathcal{F})$ et $\mathcal{R}$ un système de réécriture. Le système $\mathcal{R}$ est fortement non-terminant sur E si l'ensemble des $\mathcal{R}$ formes normales de E est vide, i.e. $\mathcal{R}^!(E) = \emptyset$. Par exemple soit le système de réécriture et l'automate suivant:

```
specification test


Vars    x
Ops
        a:0 f:1 g:1

R1
        f(f(x)) -> f(g(f(x)))
        g(x) -> x
        nil


Automata
        Description of A(0)
```

```
                states q|0 q|1 q|2 nil
                final states q|0 nil
                transitions
                        f(q|1) -> q|0
                        f(q|2) -> q|1
                        a -> q|2
                        f(q|2) -> q|2
                        nil
        End of Description
        nil
end of specification
```

où $\mathcal{L}(A(0)) = \{f^n(a) \mid n \geq 2\}$. Maintenant nous calculons l'automate d'approximation et nous obtenons l'automate $A(1) = \mathcal{T}_{\mathcal{R}_1} \uparrow (A(0))$ qui est:

```
[] start with term :
    T_up(R1)on(!A(0))

[] result term:
    Description of A(1)states q|8.q|7.q|6.q|5.q|4.q|3.q|0.q|1.q|2.nil
final states q|0.nil transitions f(q|7)->q|8.f(q|7)->q|7.f(q|2)->q|8.
f(q|2)->q|7.g(q|8)->q|7.f(q|7)->q|2.f(q|5)->q|6.f(q|5)->q|5.f(q|2)->q|6.
f(q|2)->q|5.g(q|6)->q|5.f(q|5)->q|1.f(q|3)->q|4.f(q|3)->q|3.f(q|2)->q|4.
f(q|2)->q|3.g(q|4)->q|3.f(q|3)->q|0.f(q|1)->q|0.f(q|2)->q|1.a->q|2.
f(q|2)->q|2.nil End of Description
```

Ensuite, si l'on réalise une intersection avec l'automate $A_{IRR(\mathcal{R})}$ reconnaissant les termes irréductibles, on obtient le résultat suivant:

```
[] start with term :
    simplify(!A(1) inter build_nf(R1))

[] result term:
    Description of nil states nil final states nil transitions nil End
of Description
```

Ce dernier automate reconnaît l'ensemble vide, d'où $\mathcal{R}_1^!(\mathcal{L}(A(0))) = \emptyset$, donc aucun terme de $\mathcal{L}(A(0))$ n'a de forme normale, et donc $\mathcal{R}_1$ est fortement non-terminant sur $\mathcal{L}(A(0))$.

Maintenant, donnons un autre exemple, plus concret. Reprenons le système de réécriture donné dans la section 6.6.1 et qui, rappelons-le, décrit le comportement de deux processus parallèles qui reçoivent des objets dans des piles distinctes et les comptabilisent dans un compteur commun. Outre la propriété d'exclusion mutuelle, montrée dans la section 6.6.1, une autre propriété digne d'intérêt est l'absence de blocage d'un tel système. Dans notre contexte, on considérera qu'il y a blocage s'il existe une configuration pour laquelle plus aucune transition n'est applicable, i.e. un terme irréductible par les cinq premières règles du système de réécriture (les deux autres règles codant un phénomène extérieur aux transitions proprement dites). Ceci revient à vérifier qu'il n'existe pas dans les termes atteignables, reconnus par $\mathcal{T}_{\mathcal{R}_1} \uparrow (A(0))$, de terme en forme normale par rapport aux cinq première règles du système. Or si l'on calcule l'intersection entre l'automate $\mathcal{T}_{\mathcal{R}_1} \uparrow (A)$ obtenu dans la section 6.6.1 et l'automate reconnaissant les termes irréductibles par les règles de transition (les cinq premières règles du système de réécriture), on obtient le résultat suivant:

```
[] result term:
    Description of nil states nil final states nil transitions nil
End of Description
```

Ceci assure bien que toutes les configurations atteignables sont réductibles et donc qu'il existe toujours au moins une transition du système qui peut s'appliquer, ce qui assure l'absence de blocage du système.

6.6.4 Complétude suffisante

Nous proposons une preuve de complétude suffisante à l'aide de l'approximation de l'ensemble $\mathcal{R}^!(E)$ grâce à la proposition suivante.

Proposition 6.2 *Soit $\mathcal{R}$ un système de réécriture faiblement terminant sur $E \subseteq \mathcal{T}(\mathcal{F})$. Si $\mathcal{R}^!(E) \subseteq \mathcal{T}(\mathcal{C})$, alors $\mathcal{R}$ est suffisamment complet sur E.*

En effet, puisque $\mathcal{R}$ est faiblement terminant sur E, pour tout terme $s \in E$, il existe $t \in IRR(\mathcal{R})$ tel que $s \rightarrow^*_{\mathcal{R}} t$. Or, $t \in \mathcal{R}^!(E)$. Si, de plus $\mathcal{R}^!(E) \subseteq \mathcal{T}(\mathcal{C})$ alors $t \in \mathcal{T}(\mathcal{C})$. Nous avons vu, dans la section 6.4, comment calculer $\mathcal{T}_{\mathcal{R}} \uparrow (A) \cap A_{IRR(\mathcal{R})}$, sur-ensemble régulier de $\mathcal{R}^!(\mathcal{L}(A))$, pour un ensemble régulier de requêtes reconnues par un automate A. Ainsi, si $\mathcal{R}$ est faiblement terminant sur $\mathcal{L}(A)$ et que l'on parvient à montrer que $\mathcal{T}_{\mathcal{R}} \uparrow (A) \cap A_{IRR(\mathcal{R})} \subseteq \mathcal{T}(\mathcal{C})$, alors $\mathcal{R}$ est suffisamment complet sur $\mathcal{L}(A)$. Reprenons, l'exemple du calcul des arrangements proposé dans la section 6.6.1. Si l'on réalise une intersection entre l'automate $A(1) = \mathcal{T}_{\mathcal{R}_1} \uparrow (A(0))$ déjà obtenu page 127 et l'automate reconnaissant $IRR(\mathcal{R})$, on obtient:

```
[] start with term :
    simplify(A(1) inter build_nf(R1))

[] result term:
    Description of A(2) states q|0.q|1.nil final states q|1.nil
    transitions s(q|0)->q|1.s(q|0)->q|0.o->q|0.nil End of Description
```

Nous obtenons un sur-ensemble régulier de $\mathcal{R}_1^!(\mathcal{L}(A(0)))$ reconnu par $A(2)$. Le langage reconnu $A(2)$ est $\mathcal{L}(A(2)) = \{s(x) \mid x \in Nat\}$ où $Nat = \{0, s(0), \dots\}$. On a donc $\mathcal{L}(A(2)) = Nat^* = Nat \setminus \{0\}$. D'où $\mathcal{R}_1^!(\mathcal{L}(A(0))) \subseteq Nat^* \subseteq \mathcal{T}(\mathcal{C})$. Si de plus $\mathcal{R}_1$ est faiblement terminant sur $\mathcal{L}(A(0))$, alors $\mathcal{R}_1$ est suffisamment complet pour $\mathcal{L}(A(0))$. On peut noter que si $A(2)$ est plus complexe, pour vérifier qu'il ne reconnaît que des termes constructeurs il suffit de s'assurer qu'aucun symbole défini n'apparaît dans les transitions de cet automate[13].

Il faut noter que cette méthode permet de montrer la complétude suffisante dans des cas particuliers intéressants: en présence de fonctions partiellement définies et lorsque l'application des règles est soumise à la stratégie modulaire descendante. Dans la stratégie modulaire descendante [Roc97], un système de réécriture $\mathcal{R}$ est séparé en sous-systèmes, nommés modules, $\mathcal{R}_1, \dots, \mathcal{R}_n$ tels que $\mathcal{R}_1 \succ \mathcal{R}_2 \succ \dots \succ \mathcal{R}_n$, où $\succ$ est un ordre total sur les modules. En général, l'ordre choisi sur les modules est calqué sur la hiérarchie des fonctions définies dans $\mathcal{R}_1, \dots, \mathcal{R}_n$. Pour simplifier, cela signifie que si une fonction d'un module $\mathcal{R}_i$ est définie à l'aide d'une fonction d'un module $\mathcal{R}_j$ alors $\mathcal{R}_i \succ \mathcal{R}_j$. Pour un terme t donné, la stratégie modulaire descendante consiste à réécrire *une seule fois*[14] le terme t à l'aide de la relation de réduction séquentielle $\twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n}$ (définition 3.5 page 24). En modularisant totalement la réécriture des termes, cette stratégie permet, entre autres, de simplifier la réécriture avec un système de taille importante (cela simplifie et optimise notamment le filtrage). Elle facilite également la cohabitation et l'exécution de programmes constitués de modules prototypés à l'aide de systèmes de réécriture et

13. Ceci n'est valable que si l'automate a subi au préalable un nettoyage par test d'accessibilité et un nettoyage par test d'utilité, ce qui est le cas de tous les automates que nous manipulons ici.

14. à l'inverse de la stratégie SRS où un terme peut être réécrit en plusieurs pas de $\twoheadrightarrow_{\mathcal{R}_1;\dots;\mathcal{R}_n}$.

de modules déjà implantés (voir [Roc97] pour plus de détails). Cependant, un des problèmes principaux liés à cette stratégie est de montrer sa complétude suffisante pour un ensemble de modules $\mathcal{R}_1, \ldots, \mathcal{R}_n$. Cette stratégie n'est pas toujours complète, i.e. tout terme s n'est pas nécessairement réécrit en un terme $t \in \mathcal{T}(\mathcal{C})$ et, dans sa thèse, C. Rocques [Roc97] s'est attaché à donner des conditions suffisantes sur $\mathcal{R}_1, \ldots, \mathcal{R}_n$ pour assurer la complétude. Or, dans le cas où la stratégie n'est complète que sur un sous-ensemble $E \subseteq \mathcal{T}(\mathcal{F})$, les conditions de [Roc97] ne permettent pas de conclure, comme dans l'exemple suivant.

```
specification fact
Vars    x y z
Ops
        o:0 s:1 fact:1 plus:2 mult:2
R1
        fact(s(x)) -> mult(s(x), fact(x))
        fact(o) -> s(o)
        nil
R2
        mult(o, x) -> o
        mult(s(x), y) -> plus(mult(x, y), y)
        plus(x, o) -> x
        plus(x, s(y)) -> s(plus(x, y))
        nil
```

Ici, nous donnons un critère, basé sur l'approximation de $\mathcal{R}_n^!(\ldots \mathcal{R}_1^!(E)\ldots)$ pour un ensemble E de termes initiaux, pouvant compléter les conditions syntaxiques de [Roc97]. Soit l'ensemble de termes initiaux reconnu par l'automate $A(0)$ suivant:

```
Automata
        Description of A(0)
                states q|0 q|1 nil
                final states q|0 nil
                transitions
                        fact(q|1) -> q|0
                        o -> q|1
                        s(q|1) -> q|1
                        nil
        End of Description
nil
end of specification
```

Cet automate reconnaît tout terme de la forme $fact(n)$ où n est n'importe quel entier naturel de l'ensemble $Nat = \{0, s(0), \ldots\}$. Nous calculons l'approximation en deux temps: tout d'abord nous calculons un automate, $A(1) = \mathcal{T}_{\mathcal{R}_1} \uparrow (A(0)) \cap A_{IRR(\mathcal{R}_1)}$:

```
[] start with term :
    simplify((T_up(R1)on(!A(0))inter build_nf(R1)))


[] result term:
    Description of A(1) states q|0.q|1.q|2.q|3.q|4.nil final states
q|4.nil transitions s(q|0)->q|0.o->q|0.mult(q|1,q|2)->q|4.s(q|0)->q|1.
mult(q|1,q|2)->q|2.s(q|3)->q|2.o->q|3.s(q|3)->q|4.nil End of Description
```

Ensuite, en reprenant les termes reconnus par ce dernier automate comme termes de départ des réécritures avec $\mathcal{R}_2$, nous calculons l'automate $A(2) = \mathcal{T}_{\mathcal{R}_2} \uparrow (A(1)) \cap A_{IRR(\mathcal{R}_2)}$ que voici:

```
[] start with term :
```

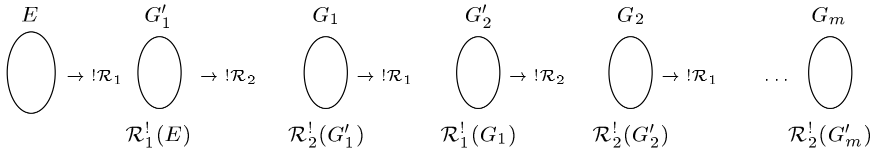
```
    simplify((T_up(R2) on (!A(1)) inter build_nf(R2)))


[] result term:
    Description of nil states q|0.q|1.q|2.nil final states q|2.nil
transitions s(q|0)->q|2.o->q|0.o->q|1.s(q|1)->q|1.s(q|1)->q|2.nil
End of Description
```

Le résultat de cette approximation de $\mathcal{R}_2^!(\mathcal{R}_1^!(\mathcal{L}(A(0))))$ est inclus dans $\mathcal{T}(\mathcal{C})$, ce qui prouve la complétude de la stratégie modulaire descendante pour les termes initiaux $\mathcal{L}(A(0))$.

6.6.5 Critère de terminaison de la relation de réduction séquentielle

Dans la section 3.2.2, nous avons défini la relation de réduction séquentielle $\rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n}$ et donné une idée du critère de terminaison de $\rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n}$ sur un ensemble E. Maintenant, afin de simplifier les explications, nous avons choisi de détailler d'abord ce critère dans le cas particulier où l'on ne dispose que de deux modules: $\mathcal{R}_1$ et $\mathcal{R}_2$. L'idée, très simple, consiste à construire les ensembles $G_1 = \mathcal{R}_2^!(\mathcal{R}_1^!(E))$, $G_2 = \mathcal{R}_2^!(\mathcal{R}_1^!(G_1))$, $\ldots$ jusqu'à atteindre un point fixe G_m tel que $G_m = \mathcal{R}_2^!(\mathcal{R}_1^!(G_m))$.



Pour chaque étape de normalisation d'un ensemble E avec $\mathcal{R}_1$ (resp. avec $\mathcal{R}_2$), l'existence des formes normales doit être garantie, i.e. $\mathcal{R}_1$ (resp. $\mathcal{R}_2$) doit être au moins faiblement terminant sur E. On peut remarquer qu'opérationnellement, il est indispensable de connaître la stratégie menant tout terme à une forme normale, mais cela n'est pas nécessaire d'un point de vue théorique; c'est également le cas dans les résultats concernant la relation de réduction modulaire. Dans la suite, nous associons à chaque système de réécriture $\mathcal{R}_i$ un domaine E_i sur lequel la terminaison faible est assurée, i.e. tout $t \in E_i$ a une forme normale par $\mathcal{R}_i$. Ainsi, si $\mathcal{R}_1$ et $\mathcal{R}_2$ terminent faiblement sur $E_1 \subseteq \mathcal{T}(\mathcal{F})$ et $E_2 \subseteq \mathcal{T}(\mathcal{F})$ respectivement, et si $E \subseteq E_1$, $G_i \subseteq E_1$, $G'_i \subseteq E_2$ pour tout $i = 1 \ldots m-1$, alors $\mathcal{R}_1$ termine faiblement sur E, G_i pour tout $i = 1 \ldots m-1$ et $\mathcal{R}_2$ termine faiblement sur G'_i pour tout $i = 1 \ldots m-1$. Si de plus $G_m \subseteq IRR(\mathcal{R}_1 \cup \mathcal{R}_2)$, alors nous obtenons les trois résultats suivant:

- $\rightarrow_{\mathcal{R}_1;\mathcal{R}_2}$ termine sur E,
- $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ termine sous la stratégie SRS sur E,
- $\mathcal{R}$ termine faiblement sur E.

Plus généralement, nous obtenons le résultat suivant pour n systèmes.

Proposition 6.3 *Soit $\mathcal{R}_1, \ldots, \mathcal{R}_n$ des systèmes de réécriture faiblement terminant respectivement sur $E_1, \ldots, E_n$ des sous-ensembles de $\mathcal{T}(\mathcal{F})$. La réécriture sous stratégie SRS $\rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n}$ termine sur E si la suite itérée d'ensembles $G_{k+1} = \mathcal{R}_n^!(\ldots \mathcal{R}_1^!(G_k)\ldots)$, initialisée avec $G_0 = E$, est telle que:*

- *$\forall k \geq 0$, $G_k \subseteq E_1, \mathcal{R}_1^!(G_k) \subseteq E_2, \ldots$, et $\mathcal{R}_{n-1}^!(\ldots \mathcal{R}_1^!(G_k)\ldots) \subseteq E_n$, et*

– *il existe un point fixe* G_k, $k \geq 0$ *tel que* $G_k \subseteq IRR(\mathcal{R}_1 \cup \ldots \cup \mathcal{R}_n)$.

Sur cette proposition, il faut remarquer deux choses. Tout d'abord, les conditions de cette proposition générale sont considérablement simplifiées si $\mathcal{R}_1, \ldots, \mathcal{R}_n$ terminent sur $\mathcal{T}(\mathcal{F})$; le critère est ramené à la seule recherche du point fixe irréductible. C'est le cas des systèmes que nous verrons en exemple. D'autre part, le fait que nous ne manipulons ici que des approximations nous donne, malgré tout, un critère de terminaison. En effet, si le sur-ensemble de $\mathcal{R}_n^!(\ldots \mathcal{R}_1^!(G_k)\ldots)$ est un sous-ensemble de $IRR(\mathcal{R}_1 \cup \ldots \cup \mathcal{R}_n)$, alors c'est nécessairement le cas de l'ensemble exact $\mathcal{R}_n^!(\ldots \mathcal{R}_1^!(G_k)\ldots)$.

Voici un premier exemple dans lequel on souhaite montrer la terminaison du calcul d'un arrangement quelconque. Au lieu de reconsidérer la preuve de terminaison dans son ensemble avec toutes les difficultés que cela suppose, nous souhaitons tirer parti des résultats de terminaison établis ou facilement prouvables et utiliser notre critère pour montrer automatiquement la terminaison de ce système pour la stratégie SRS et pour l'ensemble de requêtes qui nous intéressent.

```
specification arrangement
Vars    x y z n p

Ops
        Ar:2 minus:2 div:2 o:0 s:1 fact:1 plus:2 mult:2
R1
        Ar(n, p) -> div(fact(n), fact(minus(n, p)))
        fact(s(x)) -> mult(s(x), fact(x))
        fact(o) -> s(o)
        mult(o, x) -> o
        mult(s(x), y) -> plus(mult(x, y), y)
        plus(x, o) -> x
        plus(x, s(y)) -> s(plus(x, y))
        nil
R2
        div(o, s(y)) -> o
        div(s(x), s(y)) -> s(div(minus(x, y), s(y)))
        minus(x, o) -> x
        minus(o, x) -> o
        minus(s(x), s(y)) -> minus(x, y)
        nil

Automata
        Description of A(0)
        states q|0.q|1.nil
        final states q|0.nil
        transitions Ar(q|1, q|1) -> q|0
                o-> q|1
                s(q|1) -> q|1
                nil
        End of Description
        nil
end of specification
```

Ici, la terminaison du système $\mathcal{R}_1$ est automatiquement montrable par *lpo* à l'aide de notre premier prototype, voir section 7.3.9. D'autre part, la terminaison du système $\mathcal{R}_2$ a été montrée par T. Arts [Art97]. Pour montrer la terminaison du calcul d'un arrangement $Ar(n, p)$ pour tout entier naturel n et p, sous la stratégie SRS, nous appliquons la proposition 6.3 et nous recherchons un automate A_k point fixe, reconnaissant un sur-ensemble régulier d'un ensemble G_k, et tel que $\mathcal{L}(A_k) \subseteq IRR(\mathcal{R}_1 \cup \mathcal{R}_2)$. Dans le cas de ce système, un tel automate existe[15]:

```
[] result term:
   [true,Description of nil states q|0.q|1.nil final states q|1.nil
transitions s(q|0)->q|1.o->q|0.s(q|0)->q|0.nil End of Description]
```

Nous venons donc de montrer la terminaison de $\mathcal{R}_1 \cup \mathcal{R}_2$ sous la stratégie SRS et la terminaison faible de $\mathcal{R}_1 \cup \mathcal{R}_2$ sur $\mathcal{L}(A(0))$. A noter que, sur cet exemple, on vient également de montrer la complétude suffisante de la stratégie SRS avec $\mathcal{R}_1$ et $\mathcal{R}_2$ sur l'ensemble de termes initiaux $\mathcal{L}(A(0))$. En effet, le langage reconnu par l'automate ne contient que des termes constructeurs.

Voici maintenant un deuxième exemple illustrant la spécificité de la relation de réduction séquentielle et son pour prouver la terminaison d'un programme comportant deux modules ne partageant pas la même syntaxe et disposant chacun d'opération de traduction d'une syntaxe vers l'autre. Voici par exemple le cas d'un module $\mathcal{R}_1$ qui définit les opérations usuelles *plus* et *mult* sur des entiers "compressés", construits sur 0, *s* et *cs* où *cs* représente le double successeur. Le module $\mathcal{R}_2$ quant à lui n'a pas été conçu pour manipuler des entiers compressés et dispose d'opérations de traduction vers des entiers standards.

```
specification leq
Vars    x y

Ops
        o:0 s:1 cs:1 plus:2 mult:2 leq:2 true:0 false:0

R1
        s(s(o)) -> cs(o)
        s(s(cs(x))) -> cs(cs(x))
        plus(o, x) -> x
        plus(s(x), y) -> s(plus(x, y))
        plus(cs(x), y) -> cs(plus(x, y))
        mult(o, x) -> o
        mult(s(x), y) -> plus(mult(x, y), y)
        mult(cs(x), y) -> plus(y, plus(y, mult(x, y)))
        nil

R2
        cs(o) -> s(s(o))
        cs(x) -> s(s(x))
        leq(o, x) -> true
        leq(s(x), o) -> false
        leq(s(x), s(y)) -> leq(x, y)
        nil
end of specification

Automata
        Description of A(0)
        states q|0.q|1.nil
        final states q|0.nil
        transitions leq(q|1, q|1) -> q|0
                o -> q|1
                s(q|1) -> q|1
```

15. Dans le couple du terme résultat, le symbole `true` atteste que cet automate ne reconnaît bien que des termes irréductibles par $\mathcal{R}_1 \cup \mathcal{R}_2$.

```
                mult(q|1, q|1) -> q|1
                plus(q|1, q|1) -> q|1
                nil
        End of Description
        nil
end of specification
```

Les systèmes $\mathcal{R}_1$ et $\mathcal{R}_2$ terminent (voir les preuves automatiques de la section 7.3.10). Mais, il faut remarquer que l'union de ces deux systèmes ne termine pas puisque le terme $cs(0)$ peut être réécrit indéfiniment $cs(0) \to s(s(0)) \to cs(0) \to \ldots$. De la même façon, la relation de réécriture séquentielle ne termine pas, non plus, sur $\mathcal{T}(\mathcal{F})$ puisque l'on a également $s(s(0)) \xrightarrow[\mathcal{R}_1]{!} cs(0) \xrightarrow[\mathcal{R}_2]{!} s(s(0)) \xrightarrow[\mathcal{R}_1]{!} \ldots$, i.e. $s(s(0)) \to_{\mathcal{R}_1;\mathcal{R}_2} s(s(0)) \to_{\mathcal{R}_1;\mathcal{R}_2} \ldots$. Cependant la terminaison de la relation de réduction séquentielle sur le langage $\mathcal{L}(A(0))$ est montrable automatiquement:

```
[] result term:
   [true,Description of nil states q|0.nil final states q|0.nil
transitions true->q|0.false->q|0.nil End of Description]
```

et le sur-ensemble des résultats finaux possible est $\{true, false\}$. Enfin, voici un troisième exemple qui illustre le fait que la relation de réduction séquentielle permet également de combiner des stratégies de réductions incompatibles ou même antagonistes. Dans le système suivant, le système $\mathcal{R}_1$ termine sous une stratégie innermost alors que le système $\mathcal{R}_2$ termine sous une stratégie outermost (où l'on réécrit le redex situé à la plus haute position dans le terme).

```
specification test


Vars    x y z
Ops
        a:0 b:0 f:3 g:2 h:1

R1
        f(a, b, x) -> f(x, x, x)
        g(x, y) -> x
        g(x, y) -> y
        nil

R2
        f(x, y, z) -> h(a)
        g(x, y) -> f(g(x, y), x, y)
        nil
end of specification
```

Si l'on suppose que les étapes de normalisation par $\mathcal{R}_1$ et $\mathcal{R}_2$ sont guidées respectivement par la stratégie innermost et outermost, il est possible de montrer que la réécriture sous stratégie de réduction séquentielle termine sur $\mathcal{T}(\mathcal{F})$:

```
[] result term:
   [true,Description of nil states q|0.nil final states q|0.nil
transitions h(q|0)->q|0.b->q|0.a->q|0.nil End of Description]
```

6.6.6 Automate de forme normale

L'implantation de l'algorithme de construction d'un automate de forme normale n'est pas une mince affaire. Dans [CR87], H. Comon et J-L. Rémy donnent le premier algorithme de calcul d'une grammaire d'arbres produisant le langage $IRR(\mathcal{R})$, pour tout système de réécriture $\mathcal{R}$ linéaire à gauche. Cependant, l'implantation de cet algorithme, basé sur une procédure de disunification elle-même très complexe, est loin d'être directe. Toutefois, toute la puissance de la disunification n'étant pas requise pour le calcul d'un automate de forme normale, nous proposons ici un autre algorithme, sans procédure de disunification.

Nous avons également opté pour une approche modulaire du calcul de l'automate de forme normale. Si $\mathcal{R} = \{l_1 \to r_1, \dots l_n \to r_n\}$, nous calculons les automates $\mathcal{A}_{IRR(l_1\to r_1)}, \dots, \mathcal{A}_{IRR(l_n\to r_n)}$ reconnaissant respectivement $IRR(l_1 \to r_1), \dots, IRR(l_n \to r_n)$ et enfin nous calculons l'automate reconnaissant $IRR(\mathcal{R})$ qui est $A_{IRR(\mathcal{R})} = \mathcal{A}_{IRR(l_1\to r_1)} \cap \dots \cap \mathcal{A}_{IRR(l_n\to r_n)}$. Ce choix a de nombreuses conséquences sur l'implantation. Tout d'abord l'implantation de l'algorithme pour une règle est plus simple que dans le cas d'un système complet. Ceci nous permet également de modulariser le calcul, qui, dans le cas global, peut s'avérer très coûteux en temps machine, et à terme de distribuer le calcul. Enfin, cette approche est incrémentale.

Nous décrivons maintenant cet algorithme par deux règles de déduction: **Eliminate** et **Produce** qui s'appliquent à un triplet $(P; Q; R)$ où P et Q sont des ensembles d'états, et R est un ensemble de transitions. Initialement P contient un seul état, Q et R sont des ensembles vides. Étape après étape, les deux règles ajoutent de nouveaux états à Q et de nouvelles transitions à R. Lorsque plus aucune règle de déduction ne s'applique, Q représente l'ensemble des états de l'automate $\mathcal{A}_{IRR(l\to r)}$ et R l'ensemble de ses transitions. Durant la construction, l'ensemble P regroupe tous les états qui restent à produire. Pour produire un état $q \in P$, on ajoute dans R toute les transitions pouvant mener à q, puis on déplace q vers Q.

Soit $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ un terme linéaire donné. Dans l'automate que nous allons construire, l'état $\mathsf{diff}_l(E)$ reconnaîtra l'ensemble $T_l^{\mathsf{diff}}(E)$ des termes ne filtrant pas les termes de E et ne filtrant pas l à aucune des positions, i.e. $T_l^{\mathsf{diff}}(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \forall s \in E\, \forall \sigma : t \neq s\sigma$ et $\forall p \in \mathcal{P}os(t) : t|_p \neq l\sigma\}$. Soit $\mathcal{Q}_l^{\mathsf{diff}}$ l'ensemble d'états défini par $\mathcal{Q}_l^{\mathsf{diff}} = \{\mathsf{diff}_l(E) | E \subseteq \{l|_p \mid p \in \mathcal{P}os(l)\}\}$. Pour un ensemble de termes E donné, pour tout $f \in \mathcal{F}$ avec $ar(f) = n$, on notera E_f l'ensemble des termes de $E \cup \{l\}$ dont le symbole de tête est f, i.e. $E_f = \{t \mid t \in E \cup \{l\}$ et $\mathcal{R}oot(t) = f\}$ (où l est le membre gauche de la règle $l \to r$). Pour tout ensemble de termes E, on notera $\Pi_i(E)$ l'ensemble des i-èmes sous-termes des termes de E. La projection Π_i est inductivement définie par $\Pi_i(\emptyset) = \emptyset$ et $\Pi_i(\{f(s_1, \dots, s_n)\} \cup E) = \{s_i\} \cup \Pi_i(E)$. Pour une règle $l \to r$, le triplet initial est $(\{\mathsf{diff}_l(\emptyset)\}; \emptyset; \emptyset)$. Les règles de déduction **Eliminate** et **Produce** sont données figure 6.2.

Remarque 6.1 *Pour la règle* **Produce**, *s'il existe une constante a telle que l'ensemble $E_a = \{a\}$, on aura $\Delta_a = \{a \to \mathsf{diff}_l(E) \mid \bigcup_{i\in\emptyset} E_i = E_a$ et $\bigcap_{i\in\emptyset} E_i = \emptyset\}$. Or, on sait par convention $\bigcup_{i\in\emptyset} E_i = \emptyset \neq E_a$, d'où $\Delta_a = \emptyset$.*

Remarque 6.2 *Dans l'implantation de cet algorithme, nous terminons la construction de l'automate $A_{IRR(\mathcal{R})}$ par un renommage complet et une phase de nettoyage par test d'accessibilité.*

Proposition 6.4 *Le processus de déduction défini par l'application des règles* **Eliminate** *et* **Produce** *termine si le triplet initial est de la forme $(\{\mathsf{diff}_l(\emptyset)\}; \emptyset; \emptyset)$ où $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Le processus s'arrête sur un triplet de la forme $(\emptyset; Q; R)$.*

Preuve Tout d'abord, nous remarquons que l'ensemble $\mathcal{Q}_l^{\mathsf{diff}}$ est fini pour tout terme fini $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. A tout moment de la déduction, l'ensemble P du triplet $(P; Q; R)$ est fini. En effet, initialement, P est fini, la règle **Eliminate** supprime des états dans P, et la règle **Produce** n'en ajoute qu'un nombre fini à chaque application. D'autre part, la règle **Produce** ne peut être appliquée que si $\mathsf{diff}_l(E) \notin Q$. Or, après l'application de la règle, on a $\mathsf{diff}_l(E) \in Q$,

Eliminate

$$\frac{(P \cup \{\mathsf{diff}_l(E)\}; Q; R)}{(P; Q; R)}$$

si il existe une variable $x \in \mathcal{X}$ telle que $x \in E$, ou $\mathsf{diff}_l(E) \in Q$.

Produce

$$\frac{(P \cup \{\mathsf{diff}_l(E)\}; Q; R)}{(P \cup \bigcup_{f \in \mathcal{F}} etats(\Delta_f); Q \cup \{\mathsf{diff}_l(E)\}; R \cup \bigcup_{f \in \mathcal{F}} \Delta_f)}$$

si il n'existe pas de variable $x \in \mathcal{X}$ telle que $x \in E$, et $\mathsf{diff}_l(E) \notin Q$, où $\Delta_f = \{f(\mathsf{diff}_l(\Pi_1(E_1)), \ldots, \mathsf{diff}_l(\Pi_n(E_n))) \to \mathsf{diff}_l(E) \mid E_1 \cup \ldots \cup E_n = E_f$ et $E_1 \cap \ldots \cap E_n = \emptyset\}$.

FIG. 6.2 – *Les règles de construction de l'automate $A_{l \to r}$*

donc, la règle **Produce** ne peut être appliquée qu'une fois par état de $\mathcal{Q}_l^{\mathsf{diff}}$, soit un nombre fini de fois puisque $\mathcal{Q}_l^{\mathsf{diff}}$ est fini. Enfin, puisque **Produce** est appliquée un nombre fini de fois, et qu'elle n'ajoute à chaque fois qu'un nombre fini d'états, l'ensemble des états ajoutés à P tout au long de la déduction est fini, et donc **Eliminate** ne pourra également être appliquée qu'un nombre fini de fois. Donc le processus d'application des règles **Eliminate** et **Produce** débutant sur un triplet initial de la forme $(\{\mathsf{diff}_l(\emptyset)\}; \emptyset; \emptyset)$ termine.

Supposons maintenant que le processus s'arrête sur un triplet $(P; Q; R)$ où $P \neq \emptyset$. Il existe donc au moins un état $\mathsf{diff}_l(E)$ dans P. Si cet état est déjà dans Q, ou si si cet état est tel qu'il existe une variable x dans E, alors la règle **Eliminate** s'applique, ce qui contredit le fait que le processus se soit arrêté. D'autre part si $\mathsf{diff}_l(E) \notin \mathcal{Q}$ et s'il n'existe pas de variable x dans E, la règle **Produce** s'applique, contredisant à nouveau l'hypothèse que plus aucune règle ne peut être appliquée. D'où $P = \emptyset$. □

Théorème 6.4 *Pour toute règle $l \to r$, linéaire à gauche, le processus de déduction défini par les règles* **Eliminate** *et* **Produce** *appliqué au triplet initial $(\{\mathsf{diff}_l(\emptyset)\}; \emptyset; \emptyset)$ s'achève sur un triplet $(\emptyset; Q; R)$ tel que le langage reconnu par l'automate $\langle \mathcal{F}, Q, \{\mathsf{diff}_l(\emptyset)\}, R\rangle$ est exactement $IRR(l \to r)$.*

Preuve Nous définissons une interprétation pour chaque triplet $(A; B; C)$ à chaque étape de la déduction pour une règle $l \to r$ donnée. L'interprétation d'un triplet $(A; B; C)$ est un automate représenté par son ensemble d'états et son ensemble de transitions. La signature $\mathcal{F}$ et l'ensemble des états finaux, inutiles ici, sont ignorés afin de simplifier les notations. Pour tout triplet $(A; B; C)$, l'automate interprétation est $\langle A \cup B, \Delta_A \cup C\rangle$, où $\Delta_A = \{t \to \mathsf{diff}_l(E) \mid \mathsf{diff}_l(E) \in A$ et $t \in T_l^{\mathsf{diff}}(E)\}$.

Maintenant, nous montrons qu'à chaque étape de la déduction, le triplet $(A; B; C)$ vérifie la propriété $\mathbb{P}$ suivante: *pour tout état $\mathsf{diff}_l(E)$ de B, dans l'automate interprétation on a $t \to_{\Delta_A}^{*} t' \to_C^{+} \mathsf{diff}_l(E)$ si et seulement si $t \in T_l^{\mathsf{diff}}(E)$.*

Tout d'abord, nous montrons qu'initialement, le triplet $(A; B; C)$ vérifie cette propriété.

Pour le triplet initial $(\{\mathsf{diff}_l(\emptyset)\};\emptyset;\emptyset)$, l'automate est $\langle\{\mathsf{diff}_l(E)\},\Delta_A\rangle$. Puisque l'ensemble B est vide, la propriété est trivialement vérifiée.

Maintenant, nous montrons pour chaque règle de déduction $\frac{(A;B;C)}{(A';B';C')}$, que si $(A;B;C)$a cette propriété alors, $(A';B';C')$ l'a également. Tout d'abord, on peut remarquer que le principe des règles **Eliminate** et **Produce** est identique: on supprime un état q de l'ensemble des états à produire: $A' = A \setminus \{q\}$. Par rapport à l'interprétation, en supprimant un état de A, on supprime également des transitions dans Δ_A, i.e. $\Delta_{A'} \subseteq \Delta_A$. Afin de vérifier la conservation de la propriété, il est donc nécessaire de montrer que la perte de ces transitions ne modifie pas la relation de réécriture de l'automate interprétation, i.e. $\rightarrow_{\Delta_A\cup C}=\rightarrow_{\Delta_{A'}\cup C'}$ pour tout les termes $t \in T_l^{\mathsf{diff}}(E)$ tels que $\mathsf{diff}_l(E) \in B'$.

- La seule modification apportée par la règle **Eliminate** touche A' et donc $\Delta_{A'}$: l'ensemble $\Delta_{A'}$ est amputé des transitions $\{t \rightarrow \mathsf{diff}_l(E) \mid t \in T_l^{\mathsf{diff}}(E)\}$.

  Dans un cas, il existe une variable $x \in \mathcal{X}$ telle que $x \in E$. Dans ce cas on a $T_l^{\mathsf{diff}}(\{x\}) = T_l^{\mathsf{diff}}(E) = \emptyset$. En conséquence, les ensembles de transitions Δ_A et $\Delta_{A'}$ sont égaux et puisqu'il en est de même pour C et C', on a bien $\rightarrow_{\Delta_A\cup C}=\rightarrow_{\Delta_{A'}\cup C'}$, donc les dérivations sont identiques et la propriété est conservée.

  Dans l'autre cas, $\mathsf{diff}_l(E) \in B = Q$. Dans ce cas, puisque le triplet $(A;B;C)$ a la propriété, on sait que $t \rightarrow^*_{\Delta_A} t' \rightarrow^+_C \mathsf{diff}_l(E)$ si et seulement si $t \in T_l^{\mathsf{diff}}(E)$. D'autre part, comme $\mathsf{diff}_l(E) \in A$, on a $t \rightarrow^*_{\Delta_A} \mathsf{diff}_l(E)$ ssi $t \in T_l^{\mathsf{diff}}(E)$. De plus, les deux dérivations (1) $t \rightarrow^*_{\Delta_A} t' \rightarrow^+_C \mathsf{diff}_l(E)$ et (2) $t \rightarrow^*_{\Delta_A} \mathsf{diff}_l(E)$ sont disjointes puisqu'il existe au moins un pas de $\rightarrow_C$ dans la première. Après application de la règle, on a $C = C'$, et $\Delta_{A'} = \Delta_A \setminus \{t \rightarrow \mathsf{diff}_l(E) \mid t \in T_l^{\mathsf{diff}}(E)\}$. En conséquence, avec $\Delta_{A'}$ la dérivation (2) n'existe plus mais (1) reste intacte. La propriété $\mathbb{P}$ est donc conservée par $(A';B';C')$.

- Dans le cas de la règle **Produce**, l'état $\mathsf{diff}_l(E)$ passe de A à B'. En conséquence, il faut vérifier d'une part que (a) malgré l'absence des transitions $\{t \rightarrow \mathsf{diff}_l(E) \mid t \in T_l^{\mathsf{diff}}(E)\}$ dans $\Delta_{A'}$, pour tout état $\mathsf{diff}_l(E') \in B$, on conserve la propriété $\mathbb{P}$, et d'autre part que (b) pour le nouvel état $\mathsf{diff}_l(E)$ de B', on a bien $t \rightarrow^*_{\Delta'_A} t' \rightarrow^+_C \mathsf{diff}_l(E)$ ssi $t \in T_l^{\mathsf{diff}}(E)$.

  Puisque seules les transitions se rapportant à l'état $\mathsf{diff}_l(E)$ ont été modifiées, (b) implique (a). On montre donc (b). Tout d'abord, on montre que si $t \in T_l^{\mathsf{diff}}(E)$, alors $t \rightarrow^*_{\Delta'_A} t' \rightarrow^+_{C'} \mathsf{diff}_l(E)$. Soit $t = f(t_1,\ldots,t_n)$. Soit $E_f = \{s \mid s \in E \cup \{l\}$ t.q. $\mathcal{R}oot(s) = f\}$, et on suppose que $E_f = \{f(t_1^1,\ldots,t_n^1),\ldots,f(t_1^m,\ldots,t_n^m)\}$. Puisque $t \in T_l^{\mathsf{diff}}(E)$, on a: $\forall\sigma\ \forall j = 1\ldots m\ \exists i = 1\ldots n$ tel que $t_i \neq t_i^j\sigma$, et $\forall\sigma$ $\forall i = 1\ldots n\ \forall p \in \mathcal{P}os(t_i)$, $t_i|_p \neq l\sigma$.

  Nous avons vu que pour tout terme $f(t_1^j,\ldots,t_n^j)$ de E_f il existait au moins un sous-terme t_i^j tel que $t_i \neq t_i^j\sigma$. Maintenant, nous souhaitons construire les ensembles $S_1,\ldots,S_n$ tels que $\forall i = 1\ldots n$ on a $t_i \in T_l^{\mathsf{diff}}(S_i)$. La méthode est la suivante: dans chaque terme $f(t_1^j,\ldots,t_n^j)$ de E_f on prend un seul sous-terme t_i^j tel que $t_i \neq t_i^j\sigma$, et on place t_i^j dans S_i. On a $S_i = \{t_j^i \mid t_i \neq t_i^j\sigma\}$ et si $t_i^j \in S_i$ alors $\forall k = 1\ldots n$ t.q. $k \neq i$ on a $t_j^k \notin S_k$. On a $\forall i = 1\ldots n\ \forall\sigma\ \forall s_i \in S_i\ t_i \neq s_i\sigma$ et également $\forall i = 1\ldots n\ \forall\sigma$ $\forall p \in \mathcal{P}os(t_i)\ t_i|_p \neq l\sigma$, d'où $\forall i = 1\ldots n$ on a $t_i \in T_l^{\mathsf{diff}}(S_i)$. D'autre part, il existe un partage de E_f en n parties, i.e. n ensembles $E_1,\ldots,E_n$ tels que $E_1 \cup \ldots \cup E_n = E_f$

et $E_1 \cap \ldots \cap E_n = \emptyset$, tel que $S_i = \Pi_i(E_i)$ et donc $T_l^{\mathsf{diff}}(S_i) = T_l^{\mathsf{diff}}(\Pi_i(E_i))$. Par définition de $\Delta_{A'}$, $\mathsf{diff}_l(\Pi_i(E_i)) \in A'$ implique que pour tout terme $s \in T_l^{\mathsf{diff}}(\Pi_i(E_i))$, on a $s \rightarrow_{\Delta_{A'}} \mathsf{diff}_l(\Pi_i(E_i))$. Donc comme tous les t_i, $i = 1 \ldots n$, appartiennent à $T_l^{\mathsf{diff}}(S_i)$ et $T_l^{\mathsf{diff}}(S_i) = T_l^{\mathsf{diff}}(\Pi_i(E_i))$, on a $t_i \rightarrow_{\Delta_{A'}} \mathsf{diff}_l(\Pi_i(E_i))$. Or on ajoute également la règle $f(\mathsf{diff}_l(\Pi_1(E_1)), \ldots, \mathsf{diff}_l(\Pi_n(E_n))) \rightarrow \mathsf{diff}_l(E)$ dans C', d'où finalement:

$$t = f(t_1, \ldots, t_n) \rightarrow^*_{\Delta_{A'}} f(\mathsf{diff}_l(\Pi_1(E_1)), \ldots, \mathsf{diff}_l(\Pi_n(E_n))) \rightarrow_{C'} \mathsf{diff}_l(E)$$

Nous montrons maintenant l'implication inverse: si $t \rightarrow^*_{\Delta_{A'}} t' \rightarrow^+_{C'} \mathsf{diff}_l(E)$, alors $t \in T_l^{\mathsf{diff}}(E)$. Comme $\mathsf{diff}_l(E) \notin B$ cela signifie qu'il n'existe pas de transitions menant à $\mathsf{diff}_l(E)$ dans C. Les seules transitions menant à $\mathsf{diff}_l(E)$ sont donc celles ajoutées par cette règle: $\bigcup_{f \in \mathcal{F}} \Delta_f$, elles sont de la forme $f(\mathsf{diff}_l(\Pi_1(E_1)), \ldots, \mathsf{diff}_l(\Pi_n(E_n))) \rightarrow \mathsf{diff}_l(E)$, où les ensembles $E_1, \ldots, E_n$ sont un partage de E_f. En conséquence, pour tout terme $t = f(t_1, \ldots, t_n)$, on a nécessairement $t_i \rightarrow_{\Delta_{A'}} \mathsf{diff}_l(\Pi_i(E_i))$. Or par définition de $\Delta_{A'}$, on sait que $t_i \rightarrow_{\Delta_{A'}} \mathsf{diff}(\Pi_i(E_i))$ si et seulement si $t_i \in T_l^{\mathsf{diff}}(\Pi_i(E_i))$. D'autre part, pour tout partage $E_1, \ldots, E_k$ de E_f et pour tout terme $s = f(s_1, \ldots, s_n) \in E_f$, il existe un entier k, $1 \leq k \leq n$ tel que $s \in E_k$. Comme on a $t_k \in T_l^{\mathsf{diff}}(\Pi_k(E_k))$, on a nécessairement $t_k \neq s_k\sigma$, pour toute substitution σ, d'où $\forall s \in E_f$ on a $t = f(t_1, \ldots, t_n) \neq f(s_1, \ldots, s_n)\sigma = s\sigma$ pour toute substitution σ. Trivialement, $\forall s \in E_g$ t.q. $g \neq f$, on a $t \neq s\sigma$. D'où, quel que soit $s \in E$, $t \neq s\sigma$. Tous les t_i sont tels que $t_i \neq l\sigma$, et si $l = f(s_1, \ldots, s_n)$ alors $l \in E_f$ et donc $t \neq l\sigma$. D'où finalement $t \in T_l^{\mathsf{diff}}(E)$.

Par la proposition 6.4, nous savons que le processus s'achève sur un triplet de la forme $(P; Q; R)$, où $P = \emptyset$. Son automate interprétation est donc $T = \langle P \cup Q, \Delta_P \cup R \rangle$ qui peut être simplifié en $T = \langle Q, R \rangle$. De plus, l'état $\mathsf{diff}_l(\emptyset)$ appartient à Q. En effet, cet état est initialement dans l'ensemble P du triplet, il ne peut être supprimé par la règle **Eliminate**: il a donc été supprimé de P et ajouté à Q par la règle **Produce**. Or, nous venons de montrer que pour tout état $\mathsf{diff}_l(E)$ de Q, dans l'automate interprétation T on a $t \rightarrow^*_{\Delta_P} t' \rightarrow^+_R \mathsf{diff}_l(E)$ si et seulement si $t \in T_l^{\mathsf{diff}}(E)$. Or ici l'ensemble Δ_P est vide, donc $t \rightarrow^+_R \mathsf{diff}_l(E)$ si et seulement si $t \in T_l^{\mathsf{diff}}(E)$. Pour le cas particulier de l'état $\mathsf{diff}_l(\emptyset)$ on a $t \rightarrow^+_R \mathsf{diff}_l(\emptyset)$ si et seulement si $t \in T_l^{\mathsf{diff}}(\emptyset) = IRR(l \rightarrow r)$. Finalement, l'automate complet $T' = \langle \mathcal{F}, Q, \{\mathsf{diff}_l(\emptyset)\}, R \rangle$ est tel que $\mathcal{L}(T') = IRR(l \rightarrow r)$. □

Exemple 6.8 *Soit $\mathcal{F} = \{a : 0, b : 0, nil : 0, cons : 2, app : 2\}$, la règle $app(nil, x) \rightarrow x$ et $l = app(nil, x)$. Nous détaillons maintenant les étapes de l'algorithme appliqué au triplet initial: (P_0, Q_0, R_0) où*

$P_0 = \{\mathsf{diff}_l(\emptyset)\}$
$Q_0 = \emptyset$
$R_0 = \emptyset$.

Produce

$E_{app} = \{app(nil, x)\}$ et $\forall f \neq app$ on a $E_f = \emptyset$. Les différents partages possibles de E_{app} en deux ensembles sont: $E_1 = \{app(nil, x)\}$, $E_2 = \emptyset$, et $E_1 = \emptyset$ et $E_2 = \{app(nil, x)\}$. En conséquence, on a $\Delta_{app} = \{app(\mathsf{diff}_l(\{nil\}), \mathsf{diff}_l(\emptyset)) \rightarrow \mathsf{diff}_l(\emptyset)$, $app(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\})) \rightarrow \mathsf{diff}_l(\emptyset)\}$.

D'autre part $\Delta_{cons} = \{cons(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset)\}$, $\Delta_{nil} = \{nil \to \mathsf{diff}_l(\emptyset)\}$, $\Delta_a = \{a \to \mathsf{diff}_l(\emptyset)\}$ *et* $\Delta_b = \{b \to \mathsf{diff}_l(\emptyset)\}$. *Le triplet devient donc:*

$P_1 = \{\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\}), \mathsf{diff}_l(\{nil\})\}$
$Q_1 = \{\mathsf{diff}_l(\emptyset)\}$
$R_1 = \{app(\mathsf{diff}_l(\{nil\}), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), app(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\})) \to \mathsf{diff}_l(\emptyset), cons(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), nil \to \mathsf{diff}_l(\emptyset), a \to \mathsf{diff}_l(\emptyset), b \to \mathsf{diff}_l(\emptyset)\}$

Eliminate

$P_2 = \{\mathsf{diff}_l(\{x\}), \mathsf{diff}_l(\{nil\})\}$
$Q_2 = \{\mathsf{diff}_l(\emptyset)\}$
$R_2 = \{app(\mathsf{diff}_l(\{nil\}), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), app(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\})) \to \mathsf{diff}_l(\emptyset), cons(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), nil \to \mathsf{diff}_l(\emptyset), a \to \mathsf{diff}_l(\emptyset), b \to \mathsf{diff}_l(\emptyset)\}$

Eliminate

$P_3 = \{\mathsf{diff}_l(\{nil\})\}$
$Q_3 = \{\mathsf{diff}_l(\emptyset)\}$
$R_3 = \{app(\mathsf{diff}_l(\{nil\}), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), app(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\})) \to \mathsf{diff}_l(\emptyset), cons(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), nil \to \mathsf{diff}_l(\emptyset), a \to \mathsf{diff}_l(\emptyset), b \to \mathsf{diff}_l(\emptyset)\}$

Produce

$E_{nil} = \{nil\}$ *et* $E_{app} = \{app(nil, x)\}$, *et* $\forall f \neq nil$ *et* $f \neq app$ *on a* $E_f = \emptyset$. *On a* $\Delta_{nil} = \emptyset$ *(voir remarque 6.1) et* $\Delta_{app} = \{app(\mathsf{diff}_l(\{nil\}), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\{nil\})$, $app(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\})) \to \mathsf{diff}_l(\{nil\})\}$. *D'autre part,* $\Delta_{cons} = \{cons(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\{nil\})\}$, $\Delta_a = \{a \to \mathsf{diff}_l(\{nil\})\}$ *et* $\Delta_b = \{b \to \mathsf{diff}_l(\{nil\})\}$. *Le triplet devient donc:*

$P_4 = \{\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{nil\})\}$
$Q_4 = \{\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{nil\})\}$
$R_4 = \{app(\mathsf{diff}_l(\{nil\}), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), app(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\})) \to \mathsf{diff}_l(\emptyset), cons(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\emptyset), nil \to \mathsf{diff}_l(\emptyset), a \to \mathsf{diff}_l(\emptyset), b \to \mathsf{diff}_l(\emptyset),$
$app(\mathsf{diff}_l(\{nil\}), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\{nil\}), app(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\{x\})) \to \mathsf{diff}_l(\{nil\}), cons(\mathsf{diff}_l(\emptyset), \mathsf{diff}_l(\emptyset)) \to \mathsf{diff}_l(\{nil\}), a \to \mathsf{diff}_l(\{nil\}), b \to \mathsf{diff}_l(\{nil\})\}$

Finalement, dans P_4 *il ne reste plus que des états appartenant à* Q_4 *et l'algorithme se termine par deux applications de la règle* **Eliminate**.

6.7 Preuves

6.7.1 Filtrage dans les automates d'arbres

Dans cette section, nous montrons le théorème 6.1.

Tout d'abord, nous montrons que tout problème de filtrage $s \trianglelefteq q$ a une forme normale par l'algorithme de filtrage, i.e. l'algorithme de filtrage termine sur tout problème initial $s \trianglelefteq q$. Les règles de ξ terminent simplement. Si l'on suppose que l'algorithme de filtrage ne termine pas alors, il existe une chaîne infinie de problèmes de filtrage ϕ_1, ϕ_2, ϕ_3, ... t.q. chacun d'entre-eux est en ξ-forme normale et ϕ_{i+1} est obtenu par application de la règle **Decompose**, **Clash**, or **Configuration** à ϕ_i et normalisation à l'aide de ξ. Si l'on considère la taille des membres gauche des littéraux $s \trianglelefteq c$ d'un problème de filtrage problèmes de filtrage, les règles **Decompose** et **Clash** la font strictement décroître, alors que la règle **Configuration** la conserve. Cependant, si l'on applique **Configuration** sur un problème de filtrage donné $s \trianglelefteq q$, on obtient une disjonction finie de littéraux $s \trianglelefteq c_1 \vee \ldots \vee s \trianglelefteq c_d \vee \perp$. Sur chaque littéral $s \trianglelefteq c_i$, la seule règle pouvant être appliquée est **Decompose** ou **Clash**. En d'autes termes, chaque pas d'application de la règle **Configuration** est nécessairement suivi d'au moins une application de la règle **Decompose** ou **Clash**, et donc fait strictement décroître la taille des membres gauche. D'où, l'algorithme de filtrage termine.

Maintenant, nous démontrons que la forme normale est soit $\perp$, la formule vide, ou une forme de la forme $\bigvee_{i=1}^{k} \phi_i$ telle que $\phi_i = \bigwedge_{j=1}^{n_i} x_j^i \trianglelefteq q_j^i$, où $x_j^i \in \mathcal{X}$ et $q_j^i \in \mathcal{Q}$. Soit Φ une forme normale que ne soit ni la formule vide ni $\perp$. Si Φ n'est pas en forme normale disjonctive ou si elle contient des symboles $\perp$, alors les règles de ξ peuvent être appliquées, ce qui contredit le fait que Φ est en forme normale. Ainsi, tout problème de filtrage Φ en ξ-normal form est de la forme $\Phi = \bigvee_{i=1}^{k} \phi_i$ telle que $\phi_i = \bigwedge_{j=1}^{n_i} s_j^i \trianglelefteq c_j^i$ où $s_j^i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $c_j^i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. De plus, les membres droits des littéraux ne peuvent être autre chose qu'un état ou un terme de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ de profondeur 1. En effet, l'algorithme de filtrage part d'un problème de la forme $s \trianglelefteq q$, et chaque chaque membre droit des nouveaux littéraux est soit un état, obtenu par la règle **Decompose**, soit un membre gauche d'une transition normalisée, obtenu par la règle **Configuration**. D'autre part, les membres gauche des littéraux sont nécessairement des variables. Dans le cas contraire, la règle **Decompose** ou **Clash** peut être appliquée ce qui contredit le fait que Φ est en forme normale. Ainsi, nous obtenons que $\phi_i = \bigwedge_{j=1}^{n_i} x_j^i \trianglelefteq c_j^i$ où $x_j^i \in \mathcal{X}$, $c_j^i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et c_j^i est de profondeur 1 au plus. On peut également remarquer, qu'il est impossible d'obtenir un problème de filtrage de la forme $x \trianglelefteq s$ où $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \backslash \mathcal{Q}$, puisque le problème de filtrage initial est de la forme $s \trianglelefteq q$, la règle **Decompose** construit des littéraux de la forme $s \trianglelefteq q$, et la règle **Configuration** ne peut être appliquée si le membre gauche est une variable. Finalement, $\Phi = \bigvee_{i=1}^{k} \phi_i$ tel que $\phi_i = \bigwedge_{j=1}^{n_i} x_j^i \trianglelefteq q_j^i$ où $x_j^i \in \mathcal{X}$ et $q_j^i \in \mathcal{Q}$.

Les résultats de correction et de complétude sont obtenus en démontrant que pour toute règle $\frac{N}{D}$, pour toute Q-substitution σ, σ est une solution de N si et seulement si σ est une solution de D.

Decompose : supposons que σ soit une solution du problème de filtrage $s_1 \trianglelefteq q_1 \wedge \ldots \wedge s_n \trianglelefteq q_n$. Par la définition 6.2, on obtient que $s_1\sigma \rightarrow_\Delta^* q_1$, et $\ldots$, et $s_n \rightarrow_\Delta^* q_n$. Alors, on a $f(s_1, \ldots, s_n)\sigma = f(s_1\sigma, \ldots, s_n\sigma) \rightarrow_\Delta^* f(q_1, \ldots, q_n)$.

Inversement, si σ est une solution de $f(s_1, \ldots, s_n) \trianglelefteq f(q_1, \ldots, q_n)$, alors on a $f(s_1, \ldots, s_n)\sigma = f(s_1\sigma, \ldots, s_n\sigma) \rightarrow_\Delta^* f(q_1, \ldots, q_n)$. Par construction des automates d'arbres ascendant, on obtient que pour tout $i = 1, \ldots, n$, on a nécessairement $s_i\sigma \rightarrow_\Delta^* q_i$. D'où, σ est une solution du problème de filtrage $s_1 \trianglelefteq q_1 \wedge \ldots \wedge s_n \trianglelefteq q_n$.

Clash : pour toute Q-substitution σ, par construction des automates d'arbres ascendants, on sait que $f(s_1, \ldots, s_n)\sigma \not\rightarrow_\Delta^* g(q'_1, \ldots, q'_m)$.

Configuration : supposons que σ soit une solution de $\bigvee_{c \in C_f} s \trianglelefteq c \vee \perp = s \trianglelefteq c_1 \vee \ldots \vee s \trianglelefteq c_d \vee \perp$. Par la définition 6.2, on obtient que $s\sigma \rightarrow_\Delta^* c_1$, ou $\ldots$, ou $s\sigma \rightarrow_\Delta^* c_d$. Soit l l'entier tel que $s\sigma \rightarrow_\Delta^* c_l$. Puisque $\forall i = 1 \ldots d$, $c_i \rightarrow q \in \Delta$, on a $c_l \rightarrow q \in \Delta$, et finalement on obtient $s\sigma \rightarrow_\Delta^* c_l \rightarrow_\Delta^* q$.

Inversement, si les seules transitions de Δ menant à q sont $c_1 \rightarrow q, \ldots, c_d \rightarrow q$, alors $s\sigma \rightarrow_\Delta^* q$ implique qu'il existe un entier $l \in \{1 \ldots d\}$ tel que $s\sigma \rightarrow_\Delta^* c_l \rightarrow_\Delta^* q$. Finalement, $s\sigma \rightarrow_\Delta^* c_l$ implique que σ est une solution du problème de filtrage $s \trianglelefteq c_1 \vee \ldots \vee s \trianglelefteq c_d$.

6.7.2 Preuve du Théorème 5.2

Nous rappelons le théorème à montrer:

Pour tout automate A, pour tout système de réécriture linéaire $\mathcal{R}$, et pour toute fonction d'approximation γ, le langage $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$ est clos par $\mathcal{R}$.

Preuve Il est suffisant de montrer que l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ vérifie la condition de la proposition 6.1, pour toute fonction d'approximation γ. On doit donc montrer que $Norm_\alpha(r\sigma \to q) \subseteq \Delta'$ implique $r\sigma \to^*_{\Delta'} q$.

Soit s' un sous-terme de $r\sigma$ (éventuellement non strict) et $q' \in \mathcal{Q}'$. Par induction sur la taille de s', nous montrons maintenant que $Norm_\alpha(s' \to q') \subseteq \Delta'$ implique $s' \to^*_{\Delta'} q'$:

- si $s' = q'$, alors on a trivialement $s' \to^*_{\Delta'} q'$.
- si $s' = q'' \in \mathcal{Q}'$ tel que $q'' \neq q'$ alors, pqr le cas 2 de la définition de $Norm$, on obtient que $Norm_\alpha(s' \to q') = \{s' \to q'\}$. Puisque $Norm_\alpha(s' \to q') \subseteq \Delta'$, on a $s' \to^*_{\Delta'} q'$.
- si $s' = g(t_1, \ldots, t_m) \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, en appliquant le cas 3 de la définition de $Norm$, on obtient que

  (a) $\{g(\alpha(t_1), \ldots, \alpha(t_m)) \to q'\} \subseteq \Delta'$, et

  (b) $\bigcup_{i=1}^n Norm_\alpha(t_i \to \alpha(t_i)) \subseteq \Delta'$,

  où $\forall i = 1 \ldots n$, $\alpha(t_i) \in \mathcal{Q}_{new} \subseteq \mathcal{Q}'$. En appliquant l'hypothèse d'induction à (b), on déduit que $\forall i = 1 \ldots n$, $t_i \to^*_{\Delta'} \alpha(t_i)$. D'autre part, (a) implique que $g(\alpha(t_1), \ldots, \alpha(t_m)) \to_{\Delta'} q'$. En conséquence, $g(t_1, \ldots, t_m) \to^*_{\Delta'} g(\alpha(t_1), \ldots, \alpha(t_m)) \to_{\Delta'} q'$.

D'où $Norm_\alpha(r\sigma \to q) \subseteq \Delta'$ implique $r\sigma \to^*_{\Delta'} q$, et la condition de la proposition 6.1 est satisfaite par $\mathcal{T}_\mathcal{R} \uparrow (A)$. $\square$

6.7.3 Preuve du Théorème 6.3

Tout d'abord, nous rappelons le théorème:

Les automates approximation construits avec la fonction d'approximation ancêtre sont finis.

Preuve Puisque la fonction d'approximation γ utilisée ne dépend pas de σ, dans la suite, nous écrirons $\gamma(l \to r, q)$ pour $\gamma(l \to r, q, \sigma)$.

En premier, il faut remarquer que si l'arité des symboles de $\mathcal{F}$ est finie, si $\mathcal{Q}$ est fini et si l'ensemble des nouveaux états $\mathcal{Q}_{new}$ est fini, alors $\mathcal{Q}' = \mathcal{Q} \cup \mathcal{Q}_{new}$ est fini, le nombre de transition pouvant être ajoutées à Δ' est fini, et donc l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ est fini. Comme nous ne considérons que le cas où $\mathcal{F}$ et $\mathcal{Q}$ sont finis, afin de prouver que $\mathcal{T}_\mathcal{R} \uparrow (A)$ est fini, il est suffisant de montrer que $\mathcal{Q}_{new}$ is. Dans le cas particulier de l'approximation ancêtre, on a

(1) $\mathcal{Q}_{new} = \{\pi_i(\gamma(l \to r, q)) \mid l \to r \in \mathcal{R}, q \in \mathcal{Q}', 1 \leq i \leq Card(\mathcal{P}os_\mathcal{F}(r))\}$.

Si l'on applique le fait que $\mathcal{Q}' = \mathcal{Q} \cup \mathcal{Q}_{new}$ à (1), on obtient que $\mathcal{Q}_{new} = \mathcal{Q}_1 \cup \mathcal{Q}_2$ où:

$$\begin{aligned} \mathcal{Q}_1 &= \{\pi_i(\gamma(l \to r, q)) \mid l \to r \in \mathcal{R}, q \in \mathcal{Q}, 1 \leq i \leq Card(\mathcal{P}os_\mathcal{F}(r))\} \\ \mathcal{Q}_2 &= \{\pi_i(\gamma(l \to r, q)) \mid l \to r \in \mathcal{R}, q \in \mathcal{Q}_{new}, 1 \leq i \leq Card(\mathcal{P}os_\mathcal{F}(r))\} \end{aligned}$$

Chaque état de $\mathcal{Q}_2$ est de la forme

$$\pi_{i_1}(\gamma(l_1 \rightarrow r_1, \pi_{i_2}(\gamma(l_2 \rightarrow r_2, \ldots, \pi_{i_n}(\gamma(l_n \rightarrow r_n, q))\ldots))))$$

où $q \in \mathcal{Q}$, $l_j \rightarrow r_j \in \mathcal{R}$, et $1 \leq i_j \leq \mathcal{C}ard(\mathcal{P}os_{\mathcal{F}}(r_j))$, pour $j = 1 \ldots n$. D'autre part, le cas 2 de la definition 6.8 est équivalent à:

$\forall l_1 \rightarrow r_1, l_2 \rightarrow r_2 \in \mathcal{R}, \forall q \in \mathcal{Q}', 1 \leq i \leq \mathcal{C}ard(\mathcal{P}os_{\mathcal{F}}(r_1))$:

$$\gamma(l_2 \rightarrow r_2, \pi_i(\gamma(l_1 \rightarrow r_1, q))) = \gamma(l_2 \rightarrow r_2, q).$$

D'où,

$$\gamma(l_1 \rightarrow r_1, \pi_{i_2}(\gamma(l_2 \rightarrow r_2, \ldots, \pi_{i_n}(\gamma(l_n \rightarrow r_n, q))\ldots))) = \gamma(l_1 \rightarrow r_1, q)$$

et donc

$$\pi_{i_1}(\gamma(l_1 \rightarrow r_1, \pi_{i_2}(\gamma(l_2 \rightarrow r_2, \ldots, \pi_{i_n}(\gamma(l_n \rightarrow r_n, q))\ldots)))) = \pi_{i_1}(\gamma(l_1 \rightarrow r_1, q))$$

Ainsi, $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$ et $\mathcal{Q}_{new} = \mathcal{Q}_1$. Puisque $\mathcal{Q}$, $\mathcal{R}$, et $\mathcal{P}os_{\mathcal{F}}(r)$ sont des ensembles finis, $\mathcal{Q}_1$ est fini, et il en est de même pour $\mathcal{Q}_{new}$. $\square$

Chapitre 7

Implantations

Dans ce chapitre, nous décrivons l'implantation des travaux exposés dans les chapitres précédents: la résolution de contrainte d'ordre *gpo*, le calcul d'approximation des ensembles de descendants et de formes normales à l'aide d'automates d'arbres. Les buts des deux prototypes implantés sont très différents. Le but du premier est de montrer l'efficacité de la résolution des contraintes d'ordre par rapport à l'approche classique en preuve de terminaison. Afin de pouvoir réaliser des comparaisons, en termes de rapidité de calcul, avec des outils existants, nous avons volontairement choisi une instance décidable de *gpo*: l'ordre *lpo* pour lequel il existe des implantations recherchant les précédences automatiquement. Nous verrons ici les implantations de D. Detlef & R. Forgaard dans REVE [FG84], réutilisée dans Larch [GHG$^+$93], et celle de T. Arts [16]. Nous verrons d'ailleurs que le problème de la recherche de précédence pour l'ordre *lpo*, généralement considéré comme "bien connu", réserve quelques surprises.

La deuxième implantation a d'autres buts. D'une part, fournir une bibliothèque d'outils implantant quelques-uns des algorithmes classiques sur les automates d'arbres; d'autre part, démontrer la faisabilité du calcul d'approximation des descendants et des formes normales; enfin illustrer sur des exemples simples les applications de ces calculs à la vérification de programmes et de processus. Depuis peu, d'autres implantations des algorithmes de base sur les automates d'arbres ont vu le jour, parmi lesquelles on trouve celle de Florent Jacquemard dans le système Maude [CDE$^+$98] et le système RX de Johannes Waldmann [17]. Dès lors que l'on travaille sur des automates comptant une dizaine d'états, même pour réaliser une opération simple comme une intersection, de tels outils de calculs sont indispensables.

Dans une première partie, nous allons préciser quelques aspects de l'implantation du prototype de résolution de contraintes d'ordre *gpo* et de son instance particulière *lpo*. Nous justifierons quelques choix qui ont été faits dans l'implantation de ce prototype et nous verrons quelques comparaisons avec d'autres systèmes et d'autres prototypes existants.

Dans une deuxième partie, nous donnerons un aperçu des fonctionnalités de la bibliothèque d'outils de manipulation des automates d'arbres développée: les algorithmes classiques ainsi que ceux développés au cours de cette thèse.

7.1 Résolution des contraintes d'ordre *gpo* en ECLiPSe

Le langage choisi pour l'implantation de la résolution des contraintes d'ordre *gpo* est une implantation particulière de Prolog: ECLiPSe [MS$^+$93] (ECRC Common Logic Programming

16. Disponible à l'adresse `http://www.ericsson.se:800/cslab/~thomas/deppairs/`
17. `http://www5.informatik.uni-jena.de/~joe/rx/`

System). L'indéterminisme induit par la recherche des chemins satisfaisables dans les graphes de $\mathcal{O}$-preuves (voir section 5.5) a guidé notre choix vers un langage facilitant la gestion du retour arrière (backtrack), tel Prolog. Une autre contrainte forte concernait la représentation des graphes OCS, et la nécessité d'une structure simple permettant de les manipuler efficacement. Les tableaux statiques à n dimensions disponibles en ECLiPSe ont été un autre critère en faveur de ce langage.

Le prototype réalisé implante un analyseur syntaxique simplifié pour les systèmes de réécriture, un outil de transformation des règles conditionnelles en règles non-conditionnelles, l'algorithme de $\mathcal{C}$-déduction, des algorithmes de propagation et de simplification des $\mathcal{O}$-preuves, des outils d'instanciation partielle des $\mathcal{O}$-preuves pour la précédence et l'extension lexicographique, un algorithme de parcours indéterministe du DAG de $\mathcal{O}$-preuve global et un solveur simple pour vérifier la consistance d'une précédence. Chaque système de réécriture est stocké dans un fichier séparé. Voici un exemple illustrant la syntaxe attendue pour les systèmes de réécriture (pour d'autres exemples, voir section 7.3). Ce système est stocké dans un fichier nommé `specif1.trs`:

```
var. x, y.

f(x) -> g(x).
h(x, y) -> f(x) if[h(x, a) = g(x)].
end.
```

où les conditions sont représentées par une liste d'équations. Voici la trace de preuve obtenue lors de l'appel du prédicat de preuve de terminaison `tipi` sur le système précédent. Sur cet exemple, nous avons volontairement reproduit l'affichage complet des graphes OCS; cela ne sera pas le cas dans les exemples de la section 7.3. Dans la trace suivante, comme il s'agit d'un système conditionnel, toutes les conditions sont remplacées par de nouvelles règles. Ceci explique le nombre de règles lues: 3. Dans la description des graphes OCS de chaque règle, on trouve des indices entiers, qui correspondent à la numérotation des sommets du graphe OCS, ainsi que les $\mathcal{O}$-preuves associées aux arcs: soit une $\mathcal{O}$-preuve triviale `true` soit une variable de $\mathcal{O}$-preuve, par exemple `var(3)`.

```
[eclipse 4]: tipi("specif1.trs", ng).

3 rules parsed

Rule 2---------------------
3 : g(x)---=---3 : g(x)   (true)
3 : g(x)--->---2 : h(x, a)   (var(3))
3 : g(x)---=---2 : h(x, a)   (var(2))
3 : g(x)--->---1 : a   (var(0))
3 : g(x)--->---0 : x   (true)

2 : h(x, a)--->---3 : g(x)   (var(1))
2 : h(x, a)---=---3 : g(x)   (var(2))
2 : h(x, a)---=---2 : h(x, a)   (true)
2 : h(x, a)--->---1 : a   (true)
2 : h(x, a)--->---0 : x   (true)

1 : a---=---1 : a   (true)

0 : x---=---0 : x   (true)


Rule 1---------------------
```

```
3 : f(x)---=---3 : f(x)   (true)
3 : f(x)--->---0 : x   (true)

2 : h(x, y)--->---3 : f(x)   (var(4))
2 : h(x, y)---=---2 : h(x, y)   (true)
2 : h(x, y)--->---1 : y   (true)
2 : h(x, y)--->---0 : x   (true)

1 : y---=---1 : y   (true)

0 : x---=---0 : x   (true)


Rule 0---------------------
2 : g(x)---=---2 : g(x)   (true)
2 : g(x)--->---1 : f(x)   (var(5))
2 : g(x)---=---1 : f(x)   (var(6))
2 : g(x)--->---0 : x   (true)

1 : f(x)--->---2 : g(x)   (var(7))
1 : f(x)---=---2 : g(x)   (var(6))
1 : f(x)---=---1 : f(x)   (true)
1 : f(x)--->---0 : x   (true)

0 : x---=---0 : x   (true)



Solving is successful
Precedence: [h > f, f > g]

Termination proof done in 0.0299997 seconds, cpu time

0.0199995 seconds, cpu time for deducting obligations of proof
0.0100002 seconds, cpu time for satisfiability testing

yes.
[eclipse 5]:
```

7.1.1 Remarques sur les structures

Les structures mises en oeuvre dans l'implantation de cet algorithme sont classiques. Chaque graphe OCS est représenté par un tableau statique à deux dimensions, dont chaque case contient les étiquettes de l'arête correspondante si elle existe. La structure représentant les DAGs de $\mathcal{O}$-preuve, plus dynamique, est composée d'objets de type nœud dont les champs sont

- un identifiant unique,
- un contenu qui correspond à l'étiquette du nœud: une contrainte sur Θ et $>_{lex}$, une précédence, une $\mathcal{O}$-preuve triviale, une $\mathcal{O}$-preuve insatisfaisable, ou encore une variable d'$\mathcal{O}$-preuve,
- les identifiants d'un successeur (deux successeurs s'il s'agit d'un nœud débutant une disjonction).

On dispose également d'une base dynamique de relations, liant les variables de $\mathcal{O}$-preuves à l'identifiant d'un noeud, et représentant les Θ-substitutions.

Nous avons choisi d'associer un SOCS à chaque règle. Un autre choix possible pour la représentation des règles est de rassembler toutes les règles en un SOCS unique et complet dont le graphe OCS contient tous les termes de toutes les règles et toutes les contraintes associées. Nous avons étudié cette possibilité en pratique en réalisant une autre version du prototype et en comparant les résultats dans les deux cas. Les résultats montrent, à l'encontre de ce qu'on pouvait attendre, que le partage des $\mathcal{O}$-preuves, bien meilleur avec un SOCS global, ne permet pas de compenser le coût excessif de l'application des règles de $\mathcal{C}$-déduction sur un graphe OCS global dont la taille est beaucoup plus importante.

7.1.2 Visualisation des DAGs de $\mathcal{O}$-preuves

La visualisation des DAGs de $\mathcal{O}$-preuves a initialement été développée dans un but de contrôle. La visualisation des DAGs de $\mathcal{O}$-preuves s'est avérée cruciale dans le déboggage du prototype, mais elle a également permis d'améliorer considérablement ses performances en rendant possible des observations simples sur la structure des DAGs de $\mathcal{O}$-preuves obtenus en pratique (voir section 7.1.4).

Nous avons choisi d'utiliser le logiciel de visualisation de graphes CGRAPH[18] de F. Bertault qui possède des algorithmes de manipulation puissants et un langage de description de graphes simple et suffisant pour nos besoins. Dans un premier temps, nous avons écrit un module de traduction de la représentation mémoire du DAG de $\mathcal{O}$-preuve global dans le langage de description de graphe de CGRAPH. Voici, un exemple de ce qui peut être obtenu à l'aide de ce premier module.

Exemple 7.1 *Soit* $\mathcal{F} = \{append : 2, cons : 2, nil : 0\}$ *et* $\mathcal{R}$ *le système de réécriture suivant:*

$$
\begin{array}{l}
append(nil, x) \rightarrow x \\
append(cons(x, y), z) \rightarrow cons(x, append(y, z)) \\
rev(nil) \rightarrow nil \\
rev(cons(x, y)) \rightarrow append(rev(y), cons(x, nil)).
\end{array}
$$

Le graphe de $\mathcal{O}$-preuve correspondant à ce système est donné figure 7.1 page 175.

Dans ce graphe, le partage des $\mathcal{O}$-preuves est optimal pour chaque règle[19]. Ce graphe est peu lisible ; il ne présente que peu de similitudes avec les graphes présentés dans la section 5.5. Ce graphe représente, en fait, la $\mathcal{O}$-preuve globale du système telle qu'elle est manipulée en mémoire par le prototype.

Bien que cette représentation ait permis de déboguer le prototype, il est clair qu'elle n'est pas la plus adaptée pour visualiser la structure de la $\mathcal{O}$-preuve et guider l'utilisateur au cours d'une preuve semi-automatique. Afin de rendre ces graphes lisibles et éventuellement exploitables, nous avons choisi de développer un autre module produisant une représentation "dépliée" de ces graphes en dupliquant tous les nœuds partagés autant de fois que nécessaire. Le graphe précédent peut être ramené au graphe de la figure 7.2 page 176, plus lisible. Il est intéressant de remarquer qu'en outre, ces graphes permettent de "visualiser" concrètement ce que peut être la complexité de la recherche d'un ordre pour un système de réécriture.

18. `http://www.loria.fr/~bertault/cgraph.html`
19. mais deux $\mathcal{O}$-preuves figurant dans deux règles distinctes ne seront pas partagées dans l'implantation où les SOCS sont distincts pour chaque règle de réécriture.

Remarque 7.1 *Dans les graphes obtenus avec CGRAPH, la disposition des branches des graphes disjonctifs ne reflètent pas nécessairement la réalité, i.e. la branche la plus à gauche n'est pas nécessairement affichée comme étant la plus à gauche par CGRAPH.*

7.1.3 Stratégies de $\mathcal{C}$-déduction: efficacité, type de solution

Le processus de résolution des contraintes d'ordre *gpo*, tel qu'il est décrit par les règles de $\mathcal{C}$-déduction dans la section 5.3, est correct, et termine quelle que soit la stratégie d'application des règles. De plus, il est complet si la stratégie est équitable, i.e. si elle assure que toute règle applicable est finalement appliquée. Mais, la stratégie d'application des règles de $\mathcal{C}$-déduction a une très grande influence à la fois sur l'efficacité et sur la structure des $\mathcal{O}$-preuves.

Pour ce qui est de l'efficacité, nous avons déjà évoqué une stratégie d'application des règles de $\mathcal{C}$-déduction dans la section 5.3. La stratégie d'application des règles de $\mathcal{C}$-déduction conditionne un autre aspect de la preuve de terminaison qui est la structure des $\mathcal{O}$-preuves construites. D'autre part, comme la structure de la $\mathcal{O}$-preuve conditionne le type de solution engendrée, la stratégie d'application permet de privilégier un type de solution particulier lorsqu'il en existe plusieurs.

En effet, comme nous l'avons vu, la structure choisie pour la représentation des $\mathcal{O}$-preuves est celle d'un DAG dans lequel toute conjonction $\alpha \wedge \beta$ est représentée par le DAG $\begin{smallmatrix} A \\ | \\ B \end{smallmatrix}$ et toute disjonction $\alpha \vee \beta$ est représentée par le DAG $A\ B$, où A et B sont des DAGs représentant α and β, respectivement. Or, pour la recherche d'une solution dans un graphe de $\mathcal{O}$-preuve, il est également nécessaire d'user de stratégies lorsque l'on atteint un nœud disjonctif $A\ B$ puisqu'il faut choisir entre le DAG A et le DAG B pour la poursuite du parcours. En supposant que la stratégie utilisée soit de toujours débuter sa recherche par la branche gauche, on voit en quoi la structure du graphe de $\mathcal{O}$-preuve peut influencer le type de solution obtenue. Supposons que les règles de $\mathcal{C}$-déduction **SUBTERM Extension**, **THETA $>$ Extension** qui étendent une $\mathcal{O}$-preuve α en $\alpha \vee \ldots$ conservent toujours l'ancienne $\mathcal{O}$-preuve α dans la branche de gauche. En choisissant une stratégie d'application particulière pour les règles de $\mathcal{C}$-déduction, il est tout à fait possible de placer dans la branche la plus à gauche la $\mathcal{O}$-preuve que l'on souhaite privilégier et conditionner, ainsi, le type de solution que l'on souhaite obtenir.

Exemple 7.2 *Le DAG de la figure 7.3 page 177 privilégie les hypothèses du type $\Theta(s) \simeq_{lex} \Theta(t)$ par rapport aux hypothèses du type $\Theta(s) >_{lex} \Theta(t)$, en plaçant toujours les égalités dans les branches les plus à gauche*[20].

Pour donner une priorité moindre aux hypothèses du type $\Theta(s) >_{lex} \Theta(t)$ et au contraire favoriser les hypothèses du type $\Theta(s) \simeq_{lex} \Theta(t)$, comme dans l'exemple 7.2, il suffit de donner une priorité supérieure à l'application des règles **THETA $\sim$**, **SUBTERM Property**, **SUBTERM First**, **SUBTERM Extension**, **SUBTERM Simplification**, par rapport aux règles **THETA $>$** et **THETA $>$ Extension**.

Exemple 7.3 *Soient $\mathcal{F} = \{f : 1, g : 1, a : 0, b : 0\}$ et $\mathcal{R} = \{f(g(a)) \to b\}$. Si l'on n'applique aucune stratégie particulière sur les règles de $\mathcal{C}$-déduction, une précédence possible pour lpo est:*

```
Solving is successful
Precedence: [f > b]
```

20. dans ce graphe, l'ordonnancement des branches de gauche à droite reproduit celui du graphe mémoire, mais ce n'est pas toujours le cas (voir remarque 7.1).

En revanche, si l'on favorise les hypothèses du type $\Theta(s) \simeq_{lex} \Theta(t)$, le résultat obtenu est la précédence suivante:

```
Solving is successful
Precedence: [b = a, a = b]
```

Exemple 7.4 *Soit $\mathcal{R}$ le système 7.3.1 page 166. Voici un exemple de précédence que l'on peut obtenir avec une stratégie quelconque:*

```
Solving is successful
Precedence: [dx > x, dx > neg, dx > two, dx > div, dx > ln,
dx > minus, dx > exp, dx > plus, dx > times, one = x, x = one, zero = a,
a = zero]
```

Si au contraire, on privilégie l'application des règles **THETA** $>$ *et* **THETA** $>$ **Extension** *on obtient la précédence suivante ne comportant plus d'égalités:*

```
Solving is successful
Precedence: [dx > zero, dx > neg, dx > two, dx > div, dx > ln, dx > one,
dx > minus, dx > exp, dx > plus, dx > times]
```

Dans notre prototype standard, hormis les priorités fortes pour les règles **SUBTERM Property** et **SUBTERM Trivial** qui assurent la minimalité de la précédence (elles évitent d'ajouter des hypothèses de précédence là où cela est inutile), nous avons choisi de laisser l'utilisateur fixer la priorité des autres règles par le biais d'un prédicat `strategy` prenant en argument une liste ordonnée des sept abbréviations des noms de règles: `tg` pour **THETA** $>$, `tge` pour **THETA** $>$ **Extension**, `sfg` pour **SUBTERM First** cas $>$, `sfe` pour **SUBTERM First** cas $\simeq$, `seg` pour **SUBTERM Trivial** cas $>$, `see` pour **SUBTERM Trivial** cas $\simeq$, et `te` pour **THETA** $\sim$. On peut remarquer que les règles **SUBTERM First** et **SUBTERM Extension** sont chacunes divisées ici en deux cas: $>$ et $\simeq$. Lors du chargement du système, la stratégie par défaut est fixée par l'évaluation du prédicat suivant:

```
strategy([tg, tge, sfg, seg, sfe, see, te]).
```

Cette stratégie donne une priorité supérieure aux règles **THETA** $>$ et **THETA** $>$ **Extension** par rapport aux autres règles, tout en privilégiant toujours les hypothèses du type $\Theta(s) >_{lex} \Theta(t)$ par rapport aux hypothèses du type $\Theta(s) \simeq_{lex} \Theta(t)$ dans les cas des règles **SUBTERM First** et **SUBTERM Extension**.

7.1.4 Instanciation, simplification et parcours des DAGs de $\mathcal{O}$-preuve

Nous décrivons maintenant l'implantation des trois étapes essentielles menant à la découverte d'une précédence pour l'instance *lpo* de *gpo*, à partir d'un DAG de $\mathcal{O}$-preuve global: l'instanciation, la simplification et le parcours du DAG de $\mathcal{O}$-preuve. On pourra remarquer que parmi ces trois phases, la simplification et le parcours ne sont en aucun cas spécifiques au *lpo* mais sont réutilisables pour le traitement de n'importe quelle autre instance de *gpo*. Soit $\mathcal{F} = \{* : 2, + : 2, s : 1, 0 : 0\}$ et $\mathcal{R}$ le système de réécriture suivant:

$$\begin{aligned}
&0 * x \to 0\\
&s(x) * y \to (x * y) + y\\
&x + 0 \to x\\
&x + s(y) \to s(x + y).
\end{aligned}$$

Le DAG de $\mathcal{O}$-preuve global et général de ce système est donné figure 7.4 page 177.

Tout d'abord, sur ce DAG de $\mathcal{O}$-preuve globale, nous effectuons plusieurs IPGs (instanciations partielles gauches): une IPG pour la précédence figure 7.5, et autant d'IPG que nécessaires pour représenter une extension lexicographique (voir section 5.6.1). Nous obtenons le DAG de la figure 7.6 page 179.

Vient ensuite une phase de simplification du DAG de $\mathcal{O}$-preuve qui consiste à propager les contraintes insatisfaisables (représentées par des nœuds `False` dans le DAG) obtenues par les IPGs successives, puis à supprimer les branches insatisfaisables. Nous obtenons le graphe simplifié figure 7.7 page 180.

Enfin, pour la recherche d'une solution nous parcourons ce DAG simplifié en partant du nœud supérieur et cherchons un chemin jusqu'au noeud inférieur tel que toutes les hypothèses de précédence collectées sur ce chemin soient compatibles. Dans le cas particulier de ce DAG, un seul chemin est possible. Le test de compatibilité, très simple dans le cas de la précédence, peut être effectué par un solveur externe lorsqu'il s'agit d'une interprétation plus complexe.

Il faut remarquer que le nombre de chemins possibles dans un DAG de $\mathcal{O}$-preuve peut être considérable. Par exemple, le DAG d'$\mathcal{O}$-preuve instancié et simplifié du système 7.3.4 comporte environ 10^{11} chemins possibles: chaque disjonction à n branches multiplie d'autant le nombre de chemins possibles. Ainsi, dans cet exemple, dont le DAG de $\mathcal{O}$-preuve est donné figure 7.8 page 181, on a de haut en bas $4 \times 6 \times 6 \times 6 \times 6 \times 6 \times 6 \times 6 \times 4 \times 6 \times 2 \times 6 \times 2 \times 2 \times 4 \times 4 = 61917364224$ chemins possibles.

Afin d'avoir une idée de la complexité de la recherche d'une solution dans ce DAG en terme de temps de calcul, nous avons tenté une recherche de solution en force brute dans ce DAG qui a dû être interrompue, sans succès, après une nuit de calcul.

Or, grâce à la visualisation du DAG de $\mathcal{O}$-preuve (figure 7.8 page 181), il est possible d'effectuer une observation simple: les noeuds imposant les contraintes les plus fortes (i.e. les noeuds ne figurant dans aucune disjonction), sont tous rassemblés dans le bas du graphe, et ne sont donc étudiés qu'en fin de parcours. Or, tout chemin allant jusqu'au noeud inférieur passe nécessairement par ces noeuds et doit donc satisfaire les contraintes associées. Si l'on impose ces contraintes inévitables au processus de recherche de solution, celui-ci converge beaucoup plus rapidement: au lieu d'une nuit de calculs infructueux, nous obtenons la solution en quelques secondes (voir exemple 7.3.4). Cette méthode d'exploitation des contraintes inévitables en priorité a été intégrée à la procédure de parcours des graphes de $\mathcal{O}$-preuves.

7.1.5 Extension au cas des règles conditionnelles

Pour montrer la décroissance d'un système de réécriture conditionnel à l'aide d'un ordre de simplification tel *lpo*, nous avons vu dans la section 4.5.2 qu'il est nécessaire de montrer $l >_{lpo} t$, $l >_{lpo} s_1$, $l >_{lpo} t_1$, ..., $l >_{lpo} s_n$, $l >_{lpo} t_n$ pour toute règle $l \to r$ if $s_1 \downarrow t_1, \ldots, s_n \downarrow t_n$ du système. Pour traiter le cas des règles conditionnelles, la solution la plus simple est de transformer tout système conditionnel en un système non conditionnel, tel que cela a été proposé dans la section 4.5.2, et d'effectuer une preuve de terminaison avec notre prototype sur le système non conditionnel.

Une autre solution possible consiste à étendre l'algorithme de la section 5.3 au cas des règles conditionnelles. Les trois modifications à effectuer sont les suivantes:

- ajouter au graphe OCS représentant la règle, les graphes représentant les termes de la condition,

- modifier la construction de la $\mathcal{O}$-preuve globale: considérer la conjonction des $\mathcal{O}$-preuves des inégalités $l > r$, $l > s_1, \ldots, l > s_n$ et $l > t_1, \ldots, l > t_n$ pour toute règle $l \rightarrow r$ if $s_1 \downarrow t_1, \ldots, s_n \downarrow t_n$, au lieu de $l > r$ pour toute règle non-conditionnelle $l \rightarrow r$.

L'intérêt d'une telle approche par rapport à la première est de concentrer dans le même SOCS la résolution de toutes les inégalités $l > r$, $l > s_1, \ldots, l > s_n$ et $l > t_1, \ldots, l > t_n$. Afin d'étudier l'efficacité d'une telle approche en pratique, nous avons réalisé une version étendue de notre prototype prenant en compte ces modifications, et nous avons effectué des tests comparatifs par rapport à la première approche. En terme d'efficacité pure avec l'instance *lpo* de *gpo*, il ressort de ces tests que la première approche est, de loin, la plus efficace. Nous avons rassemblé quelques temps comparatifs dans le tableau suivant.

Système	Transformation en non-conditionnel	Prototype spécifique conditionnel
Tri par insertion section 7.3.3	1.89 s	2.24 s
Conformité ABR section 7.3.7	8.92 s	26.21 s
Gilbreath card trick section 7.3.8	9.09 s	11.09 s

7.1.6 Comparaison avec d'autres systèmes et prototypes

Pour être largement utilisable en tant qu'outil de preuve de terminaison, le prototype actuellement implanté devrait être complété. En effet, de toutes les instances de *gpo* seule l'une d'entre-elles est automatisée: le *lpo*. Cependant, ce que nous souhaitons montrer ici est l'efficacité de l'approche contrainte pour la recherche d'ordre. L'algorithme de de $\mathcal{C}$-déduction proposé dans la section 5.3, valable quelle que soit l'instance de *gpo* considérée, est implanté tel quel dans notre prototype. Afin de montrer dans quelle mesure cet algorithme préliminaire général de construction de $\mathcal{O}$-preuves peut accélérer la recherche de précédence dans le cas d'une instance particulière de *gpo* (le *lpo*), nous le comparons avec d'autres outils, eux, totalement dédiés au *lpo*.

Pour *lpo*, la méthode la plus simple pour la recherche d'une précédence satisfaisant un ensemble de contraintes d'ordre est la recherche exhaustive: considérer toutes les précédences possibles jusqu'à ce que l'une d'entre elles satisfasse les contraintes vis-à-vis du *lpo*. Or, cette méthode n'est pas efficace en pratique parce que le nombre de précédences possibles à considérer augmente de façon exponentielle avec le nombre de symboles dans la signature $\mathcal{F}$. Pour pallier à ce problème, D. Detlef & R. Forgaard ont proposé un algorithme de déduction automatique de la précédence [FD85] pour les ordres à précédence usuels: aussi bien *lpo* que *rpo*. Pour un ordre $\succ^P$ paramètré par une précédence P et un système de réécriture $\mathcal{R}$, cet algorithme construit incrémentalement P de façon à ce que $\succ^P$ prouve la terminaison de $\mathcal{R}$. Initialement, P est une précédence vide. Tant qu'il reste des règles de $\mathcal{R}$ ne pouvant être orientées avec $\succ^P$, des hypothèses de précédence sont ajoutées à P.

Larch Prover

L'algorithme de D. Detlef & R. Forgaard, a été successivement implanté dans REVE 2.3 [FG84], puis dans LP [GHG+93] (Larch Prover). Cette implantation est, a priori, plus générale que celle d'un simple *lpo* puisqu'elle permet de déduire les statuts des opérateurs (multi-ensemble, lexico gauche-droite, ou droite-gauche) et la précédence pour un ordre, nommé EPOS, très proche d'un *rpo* avec statut. Une autre différence importante, par rapport à notre implantation, est que le système de recherche d'ordre s'applique à des équations et peut donc choisir l'orientation qui lui convient le mieux. Au contraire, notre prototype travaille sur des règles de réécriture, i.e. des équations déjà orientées. Cependant, il est possible d'imposer à Larch l'orientation des équations à l'aide de la méthode suivante. Pour toute inégalité $s > t$ définie sur une signature $\mathcal{F}$, on tente d'orienter l'équation $f(s,x) = f(t,a)$ où f et a sont des nouveaux symboles de fonctions n'apparaissant pas dans $\mathcal{F}$, x est une variable n'apparaissant pas dans l'inégalité, le statut de f est lexicographique de gauche à droite, et on impose que a soit le plus petit élément de la précédence. Avec de telles contraintes, on se trouve dans la situation suivante:

- l'équation ne peut être orientée que de gauche à droite. En effet, pour qu'elle puisse être orientée de droite à gauche il faudrait, entre autre, que $f(t,a) > x$. Or, c'est impossible avec la classe d'ordres considérés puisque x ne figure pas dans $f(t,a)$,
- comme le symbole de tête f est identique de part et d'autre de l'équation, que le statut de f est lexicographique de gauche à droite, et que x et a sont incomparables, on a $f(s,x) > f(t,a)$ si et seulement si $s > t$.

En utilisant cette méthode, nous avons pu réaliser avec LP des preuves de terminaison automatiques très rapides. Par exemple, en transformant le système de la section 7.3.8, en un système d'équations non-conditionnelles, en "orientant" artificiellement les équations à l'aide de la méthode précédente et en tentant une preuve automatique à l'aide de LP, on obtient une précédence satisfaisante en environ 5 secondes. Dans la trace suivante, `nf` et `na` représentent les nouveaux symboles utilisés pour l'orientation artificielle des équations:

```
All equations have been oriented into rewrite rules.
The rewriting system is guaranteed to terminate.

LP0.1.12: display ordering

Ordering constraints:

nf > na
neg > (r, na, b, mytrue)
pairedlist > (na, myfalse, mytrue)
paired > (na, myfalse, mytrue)
rotate > (append, cons, na, myfalse, null, mytrue)
r > (na, b, mytrue)
b > na
mytrue > na
myfalse > (na, mytrue)
append > (cons, na, myfalse, mytrue)
cons > (na, myfalse, mytrue)
null > (na, mytrue, myfalse)
```

A titre de comparaison, sur ce même exemple, notre prototype ne répond qu'après 9.09 secondes. Cependant, nous souhaitons attirer l'attention du lecteur sur le fait que le système de recherche de précédence de LP n'est pas complet. Ceci apparaît, par exemple, lorsque l'on tente une preuve sur le système de la section 7.3.1 (toujours en utilisant la méthode d'orientation artificielle à l'aide de nouveaux symboles de fonction), sur lequel nous obtenons:

```
The formulas cannot be ordered using the current ordering.

LP0.1.14: display ordering
```

```
Ordering constraints:

x > (na, one)
a > (na, zero)
ln > (na, div, dx, neg, plus, minus, times, exp, two)
div > na
dx > (neg, na, plus, minus, times, div, exp, two)
neg > na
plus > na
minus > na
times > na
```

L'affirmation, faite par LP, selon laquelle les formules ne peuvent pas être orientées à l'aide de l'ordre courant est un peu forte puisqu'il existe bien un *lpo* et une précédence qui permettent d'orienter ces règles (voir système de la section 7.3.1). C'est donc bien une heuristique incomplète qui est utilisée ici. Afin de la déterminer, nous pouvons simplement analyser le résultat intermédiaire livré par LP dans la trace précédente. Remarquons principalement trois des hypothèses effectuées par LP: `x > one`, `a > zero` et `ln > div` (rappelons que `x` est une constante dans ce système). Les seules règles dans lesquelles ces symboles sont confrontés sont respectivement:

$$\begin{array}{l} dx(x) \to one \\ dx(a) \to zero \\ dx(ln(alpha)) \to div(dx(alpha), alpha) \end{array}$$

Or, un autre choix possible pour orienter ces règles consistait à prendre `dx > one`, `dx > zero` et `dx > div`. LP a privilégié la recherche de précédence assurant qu'un sous-terme du membre gauche est supérieur au membre droit de la règle. Grâce à cette observation, nous pouvons facilement prendre LP en défaut. Voici un système fournissant un autre contre-exemple simple de la complétude de LP. Soit $\mathcal{F} = \{f : 1, g : 1, h : 1\}$ et $\mathcal{R}$:

$$\begin{array}{l} f(h(x)) \to g(x) \\ g(x) \to h(x) \end{array}$$

qui donne l'exécution suivante:

```
The formulas cannot be ordered using the current ordering.

LP0.1.14: display ordering

Ordering constraints:

h > (na, g)
```

Clairement pour la première règle, en choisissant `h > g`, LP a privilégié $h(x) > g(x)$ pour assurer $f(h(x)) > g(x)$. Or, de façon étonnante, ce choix n'est absolument pas remis en cause par la deuxième règle qui, elle, impose `g > h` et la preuve termine sur un échec. De la même façon, LP n'envisage pas les cas d'équivalence dans la recherche de précédence. Ceci est sans doute lié au fait qu'à l'époque de la première implantation de cet algorithme, comme on l'a vu dans la section 4.1, les précédences n'étaient généralement définies que comme des ordres stricts. Ainsi, le système $\mathcal{R}$ suivant

$$\begin{array}{l} f(s(x), y) \to g(x, y) \\ g(s(x), y) \to f(x, y) \end{array}$$

ne peut être orienté automatiquement avec LP. Or, ce type de règle se retrouve dans tous les systèmes mutuellement récursifs et leur preuve de terminaison nécessite, en général de poser des équivalences entre symboles dans la précédence. C'est le cas du système précédent dont la preuve de terminaison peut être réalisée avec notre prototype:

```
Solving is successful
Precedence: [g = f, f = g]
```

Ainsi, l'outil de preuve de terminaison de LP, bien qu'efficace dans de nombreux cas, ne constitue une procédure de décision ni pour *rpo* avec statut ni pour *lpo*. Notre prototype, moins rapide sur des cas comme l'exemple de la section 7.3.8, fournit en revanche une procédure de décision pour *lpo*. Par exemple, LP a donné des résultats négatifs sur la spécification de l'algorithme de conformité ABR (système de la section 7.3.7), sur le tri par insertion section 7.3.3 ou encore sur la spécification de circuit section 7.3.2, alors qu'il existe bien un *lpo* prouvant la terminaison de ces spécifications. Notons, en outre que les cas d'équivalence, négligés par LP, sont importants si l'on souhaite utiliser le préordre bien fondé *lpo* pour satisfaire des inégalités non-strictes, comme le fait T. Arts dans la méthode des paires de dépendance.

Le prototype de T. Arts

Afin de comparer notre prototype avec une méthode complète, nous nous sommes intéressé à une implantation plus récente, celle de T. Arts[21] pour l'ordre *lpo*. Cette implantation est, à notre connaissance, l'implantation complète la plus efficace de l'algorithme de D. Detlef & R. Forgaard. Dans le tableau suivant, nous donnons quelques temps comparatifs sur des exemples de systèmes de diverses tailles. Le prototype de T. Arts est plus récent que le notre et a été programmé dans le langage fonctionnel Erlang 4.5.3[22] qui est d'une efficacité supérieure[23] à ECLiPSe 3.5.2. De ces essais comparatifs, il ressort plusieurs choses. Tout d'abord, si sur des systèmes de taille réduite, l'algorithme de Detlef & Forgaard implanté par T. Arts se comporte bien, le temps de calcul augmente de façon très importante avec la taille des systèmes. Au contraire, notre méthode par résolution de contraintes s'avère lourde à mettre en oeuvre pour des petits systèmes, mais elle résiste mieux à l'augmentation du nombre de règles. Voici un tableau regroupant quelques essais comparatifs[24] menés sur les trois implantations pour les exemples de la section 7.3. Il faut noter que pour Larch, tous les systèmes ont dû être transformés en équations orientées artificiellement à l'aide de la méthode de la section 7.1.6. Dans ce tableau, “Nb. cont.” représente le nombre de contraintes *gpo* à résoudre pour chaque système. Ce nombre peut être différent du nombre de règles lorsqu'il s'agit d'un système conditionnel (voir section 7.1.5). Pour chaque implantation, et pour chaque système, la colonne “Sol.” précise, si oui ou non une solution (une précédence) a pu être obtenue. Les temps de calculs de Larch sont approximatifs car il ne dispose pas de

21. Disponible à l'adresse `http://www.ericsson.se:800/cslab/~thomas/deppairs/`
22. `http://www.erlang.se/`
23. Pour des calculs équivalents, un programme en Erlang 4.5.3 s'exécute approximativement 4 à 5 fois plus vite qu'un programme en ECLiPSe 3.5.2
24. réalisés sur un SUN UltraSparc 1

fonction de mesure du temps de calcul satisfaisante.

Système	Nb. règles	Nb. cont.	Larch		Implant. T. Arts		Proto. contraintes	
			Sol.	Temps	Sol.	Temps	Sol.	Temps
Dérivation symbolique	9	9	**non**	$\approx$ 2.00 s	oui	1.81 s	oui	10.43 s
Circuit intégré	10	10	**non**	$\approx$ 2.00 s	oui	0.66 s	oui	1.92 s
Tri par insertion	21	27	**non**	$\approx$ 4.00 s	oui	5.48 s	oui	1.89 s
Système de freinage	24	28	oui	$\approx$ 4.00 s	oui	64.70 s	oui	6.32 s
Freinage cont. suppl.	35	44	oui	$\approx$ 5.00 s	oui	401.21 s	oui	29.39 s
Freinage non-orient.	40	49	**non**	$\approx$ 5.00 s	**?**	$>$ 3 jours.	**non**	7.51 s
Conformité ABR	35	60	**non**	$\approx$ 5.00 s	oui	3696.15 s	oui	8.92 s
Gilbreath card trick	37	57	oui	$\approx$ 5.00 s	oui	174.81 s	oui	9.09 s

Un autre point particulièrement important est le temps de réponse sur les systèmes *ne pouvant pas* être orientés par l'ordre *lpo*. En effet, un des intérêts tout particulier du *lpo* est qu'il est décidable. Avoir une réponse négative dans un temps court est au moins aussi important, puisque cela permet d'envisager plus vite une autre méthode de preuve. Il convient de voir ce qu'il en est en pratique, dans les cas négatifs. Dans le tableau précédent, un seul système (le système de la section 7.3.6) n'est pas orientable par *lpo*, mais illustre bien les choix qui ont été fait dans les trois approches. Dans notre approche, la détection et la propagation des contraintes insatisfaisables constituent la base du mécanisme de simplification utilisé. Ceci permet d'avoir une réponse négative très rapidement. En revanche, l'algorithme de Detlef & Forgaard ne possède pas un tel mécanisme et se focalise sur la recherche de précédence *satisfaisant* les inégalités. C'est pourquoi dans l'implantation de T. Arts, ces contraintes insatisfaisables ne sont pas détectées et l'exécution a du être interrompue. Larch, grâce à son heuristique fixe, répond très rapidement mais il ne fait pas de différence particulière entre ce système inorientable par *lpo* et tout autre système *qu'il ne parvient pas* à orienter.

Dans notre approche, les contraintes insatisfaisables sont répercutées dans le DAG de $\mathcal{O}$-preuve et peuvent être rapidement détectées avant même le début du parcours. Par exemple, un extrait du DAG de $\mathcal{O}$-preuve global du système de la section 7.3.6 est donné figure 7.9 page 182; on y trouve deux contraintes insatisfaisables `False` correspondant aux deux règles

$$\begin{array}{c} leq(x + (-(y)), u) \to leq(x, y + u) \\ leq(x, u + (-(y))) \to leq(x + y, u) \end{array}$$

qui, après la phase d'instanciation, s'avèrent inorientables par *lpo*.

Pour conclure sur notre prototype, nous souhaitons attirer l'attention du lecteur sur le fait que, bien qu'il soit limité en tant que procédure de preuve de terminaison à part entière à cause de la relative faiblesse de l'ordre *lpo*, associé à la méthode des paires de dépendance ou à une méthode de transformation de système telle que l'étiquetage sémantique [Zan95] (semantic labelling), il peut se révèler extrêmement efficace. Dans le cas particulier de l'étiquetage sémantique, la phase d'étiquetage peut multiplier le nombre de règles. Aussi, l'efficacité d'une procédure de décision pour *lpo* sur de gros systèmes est-elle particulièrement importante.

7.2 Automates en ELAN

Outre les algorithmes d'approximation du calcul des descendants et des formes normales, nous avons implanté les algorithmes classiques d'intersection, d'union, de nettoyage par test d'accessibilité, de nettoyage par test d'utilité, l'algorithme de matching dans les automates proposé dans la section 6.2.3 et l'algorithme de calcul de l'automate de forme normale décrit section 6.6.6. Cette bibliothèque d'outils a été entièrement implantée dans le langage ELAN [BKK+96]. Celle-ci est utilisable à travers trois interfaces différentes:

Calculs classiques qui, à partir de la donnée d'une signature et d'une liste d'automates d'arbres nommés définis sur la signature, permet de réaliser toutes les opérations standard sur les automates: les deux types de nettoyage, le renommage, l'intersection, et l'union. Le module ELAN correspondant est nommé `automaton.lgi`.

Approximations qui, à partir de la donnée d'une signature, d'un ensemble de variables, d'un système de réécriture `R1` et d'une liste d'automates d'arbres nommés, permet de réaliser toutes les opérations du module précédent avec en plus le calcul de l'automate approximation (ancêtre) et le calcul de l'automate reconnaissant les termes irréductibles si `R1` est un système linéaire à gauche. Le module ELAN correspondant est nommé `nf_automaton.lgi`.

Terminaison qui, à partir de la donnée d'une signature, d'un ensemble de variables, de deux systèmes de réécriture `R1`, `R2` et d'une liste d'automates d'arbres nommés, permet de réaliser des preuves de terminaison pour la relation de réduction séquentielle $\rightarrow_{\mathcal{R}_1;\mathcal{R}_2}$. Le module ELAN correspondant est nommé `termination.lgi`.

7.2.1 Calculs classiques

Tout d'abord, nous montrons l'utilisation du module `automaton.lgi` sur les exemples 2.7, 2.8 et 2.9 de la section 2.5. Voici le fichier spécification `exemple.spc` décrivant et nommant les automates.

```
specification exemple

Ops
        a:0 b:0 cons:2 null:0

Automata
        Description of A(0)
                states  q|0 q|1 q|2 nil
                final states q|0 nil
                transitions
                        cons(q|2, q|1) -> q|0
                        null -> q|1
                        cons(q|2, q|1) -> q|1
                        a -> q|2
                        b -> q|2
                        nil
        End of Description
```

```
        Description of A(1)
                states  q|0 q|1 q|2 q|3 q|4 nil
                final states q|0 nil
                transitions
                        null -> q|0
                        cons(q|1, q|2) -> q|0
                        null -> q|1
                        cons(q|1, q|2) -> q|1
                        a -> q|1
                        b -> q|1
                        null -> q|2
                        cons(q|1, q|3) -> q|2
                        null -> q|3
                        cons(q|1, q|4) -> q|3
                        null -> q|4
                        nil
        End of Description

        Description of A(2)
                states q|0 q|1 q|2 q|3 q|4 nil
                final states q|0 nil
                transitions
                        null -> q|0
                        cons(q|3, q|0) -> q|0
                        cons(q|4, q|1) -> q|0
                        cons(q|0, q|2) -> q|0

                        null -> q|1
                        cons(q|4, q|1) -> q|1
                        cons(q|0, q|2) -> q|1

                        null -> q|2
                        cons(q|0, q|2) -> q|2

                        a -> q|3
                        b -> q|4
                        nil
        End of Description
        nil
end of specification
```

Ensuite, nous appelons l'interprète `ELAN` sur le module `automaton.lgi` et la spécification `exemple.spc`:

```
elan -C automaton.lgi exemple.spc
```

Voici maintenant quelques calculs réalisables. Tout d'abord un calcul d'intersection entre l'automate `A(0)` et l'automate `A(1)`, où `A(0)` et `A(1)` sont des noms (ou références) d'automates. Pour les déréférencer, nous utilisons ici l'opérateur "`!`".

```
enter command finished by ';':
run !A(0) inter !A(1) ;

[] result term:
    Description of nil states(q|0.q|1.q|2.nil x q|0.q|1.q|2.q|3.q|4.nil)
```

```
final states(q|0.nil x q|0.nil)transitions b->[q|2,q|1].a->[q|2,q|1].
cons([q|2,q|1],[q|1,q|4])->[q|1,q|3].cons([q|2,q|1],[q|1,q|3])->[q|1,q|2].
cons([q|2,q|1],[q|1,q|2])->[q|1,q|1].cons([q|2,q|1],[q|1,q|2])->[q|1,q|0].
null->[q|1,q|4].null->[q|1,q|3].null->[q|1,q|2].null->[q|1,q|1].
null->[q|1,q|0].cons([q|2,q|1],[q|1,q|4])->[q|0,q|3].cons([q|2,q|1],
[q|1,q|3])->[q|0,q|2].cons([q|2,q|1],[q|1,q|2])->[q|0,q|1].cons([q|2,q|1],
[q|1,q|2])->[q|0,q|0].nil End of Description
```

Dans ce résutat, on retrouve bien la définition de l'intersection de deux automates: l'ensemble des états est le produit cartésien des deux ensembles d'états et les transitions sont définies sur des couples d'états. On peut remarquer que la totalité des états, résultat du produit cartésien des deux ensembles d'états des automates initiaux, n'est pas générée explicitement, mais conservée sous sa forme implicite:

```
(q|0.q|1.q|2.nil x q|0.q|1.q|2.q|3.q|4.nil)
```

Ceci permet d'éviter l'explosion de l'ensemble des états de l'automate intersection. De plus, il est possible d'obtenir une forme renommée et plus lisible, dont l'ensemble des états est restreint à l'ensemble des états utiles par la requête suivante.

```
enter command finished by ';':
run simplify(!A(0) inter !A(1));

[] result term:
    Description of nil states q|0.q|1.q|2.q|3.q|4.nil final states
q|4.nil transitions b->q|2.a->q|2.cons(q|2,q|0)->q|1.cons(q|2,q|1)->q|3.
null->q|0.null->q|1.null->q|3.cons(q|2,q|3)->q|4.nil End of Description
```

L'automate résultat obtenu reconnait l'ensemble des listes plates non vides, constituées de a ou de b, et dont la longueur n'excède pas 3. Pour réaliser l'intersection entre ce dernier automate et l'automate `A(2)`, nous effectuons l'opération suivante (dans laquelle `R` représente le dernier résultat obtenu).

```
enter command finished by ';':
run simplify(R inter !A(2)) ;

[] result term:
    Description of nil states q|0.q|1.q|2.q|3.q|4.q|5.q|6.q|7.q|8.nil
final states q|8.nil transitions cons(q|4,q|5)->q|8.cons(q|6,q|7)->q|8.
null->q|5.null->q|7.null->q|3.null->q|2.null->q|1.null->q|0.
cons(q|4,q|3)->q|5.cons(q|4,q|3)->q|7.cons(q|6,q|2)->q|7.cons(q|4,q|1)->q|3.
cons(q|4,q|1)->q|2.cons(q|6,q|0)->q|2.a->q|6.b->q|4.nil End of Description
```

Cet automate reconnait l'ensemble des listes plates non-vides, composées de a et de b, ordonnées, et dont la longueur ne dépasse pas 3.

Les autres commandes disponibles sont `union` pour l'union d'automates, `clean` pour le nettoyage par test d'accessibilité, et `renum` pour le nettoyage par test d'utilité avec renommage. La commande `simplify` correspond à la succession de `renum` suivi de `clean`.

7.2.2 Calculer des approximations et des ensembles de termes irréductibles

Les spécifications utilisées par le module `nf_automaton.lgi` font intervenir, en plus des champs communs avec les spécifications précédentes prise en compte par `automaton.lgi`, des déclarations de variables et la déclaration d'un système de réécriture $\mathcal{R}_1$. Voici un exemple de spécification.

```
specification append
Vars    x y z
Ops
        a:0 b:0 cons:2 null:0 append:2

R1
        append(null, x) -> x
        append(cons(x, y), z) -> cons(x, append(y, z))
        nil

Automata
        Description of A(0)
        states q|0 q|1 q|2 nil
        final states q|0 nil
        transitions
                append(q|1, q|1) -> q|0
                cons(q|2, q|1) -> q|1
                null -> q|1
                a -> q|2
                b -> q|2
                nil
        End of Description
        nil
end of specification
```

L'opération implantant le calcul de l'automate reconnaissant les termes irréductibles $A_{IRR(\mathcal{R}_1)}$, proposé dans la section 6.6.6 se nomme `build_nf`. Voici ce qui est obtenu pour le système `R1` de la spécification précédente:

```
enter command finished by ';':
run build_nf(R1);

[] result term:
    Description of nil states q|0.q|1.nil final states q|1.nil
transitions a->q|1.b->q|1.cons(q|1,q|1)->q|1.null->q|1.append(q|0,q|1)->
q|1.a->q|0.b->q|0.append(q|0,q|1)->q|0.nil End of Description
```

La construction d'un automate approximation $\mathcal{T}_{\mathcal{R}_1} \uparrow (\mathcal{L}(A_0))$ reconnaissant un sur-ensemble des $\mathcal{R}_1$-descendants de $\mathcal{L}(A_0)$, est invoquée à l'aide de la commande `T_up`, de la façon suivante:

```
enter command finished by ';':
run T_up(R1) on (!A(0));

[] result term:
    Description of A(0)states q|4.q|0.q|1.q|2.nil final states q|0.nil
transitions cons(q|2,q|4)->q|4.append(q|1,q|1)->q|4.cons(q|2,q|4)->q|0.
append(q|1,q|1)->q|0.cons(q|2,q|1)->q|1.cons(q|2,q|1)->q|4.cons(q|2,q|1)->q|0.
null->q|1.null->q|4.null->q|0.a->q|2.b->q|2.nil End of Description
```

Enfin l'aproximation des $\mathcal{R}_1$-formes normales de $\mathcal{L}(A_0)$ peut être obtenue soit en réalisant une intersection entre l'avant dernier et le dernier résultat, soit, comme ici, par un calcul direct.

```
enter command finished by ';':
```

```
run simplify(T_up(R1) on (!A(0)) inter build_nf(R1));

[] result term:
    Description of nil states q|0.q|1.q|2.q|3.nil final states q|3.nil
transitions null->q|3.null->q|2.null->q|0.cons(q|1,q|0)->q|3.cons(q|1,q|0)->
q|2.cons(q|1,q|0)->q|0.cons(q|1,q|2)->q|3.cons(q|1,q|2)->q|2.b->q|1.a->q|1.
nil End of Description
```

7.2.3 Prouver la terminaison

Ce dernier module automatise l'application de quelques-unes des opérations précédentes de façon à montrer la terminaison de la relation de réduction séquentielle. Les spécifications utilisées par ce module sont de la forme:

```
specification mapfact
Vars    x y z

Ops
        o:0 p:1 s:1 fact:1 plus:2 mult:2 cons:2 map_fact:1 null:0
R1
        map_fact(null) -> null
        map_fact(cons(x, y)) -> cons(fact(x), map_fact(y))
        nil
R2
        p(s(x)) -> x
        mult(o, x) -> x
        mult(s(x), y) -> plus(mult(x, y), y)
        plus(x, o) -> x
        plus(x, s(y)) -> s(plus(x, y))
        fact(s(x)) -> mult(s(x), fact(p(s(x))))
        fact(o) -> s(o)
        nil

Automata
        Description of A(0)
        states q|0 q|1 q|2 nil
        final states q|0 nil
        transitions
                map_fact(q|1) -> q|0
                null -> q|1
                cons(q|2, q|1) -> q|1
                o -> q|2
                s(q|2) -> q|2
                nil
        End of Description
        nil
end of specification
```

Il existe plusieurs façons de réaliser une preuve de terminaison de la relation de réduction séquentielle $\rightarrow_{\mathcal{R}_1;\mathcal{R}_2}$: soit sur toute l'algèbre initiale, à l'aide de la commande `start`, soit sur un ensemble de termes de départ grâce à la commande `start(`A`)`, où A est l'automate reconnaissant les termes initiaux. Voici un exemple sur un ensemble de termes initiaux donné:

```
enter command finished by ';':
run start(!A(0));
```

```
[] result term:
   [true,Description of nil states q|0.q|1.q|2.q|3.q|4.nil final states q|4.
nil transitions cons(q|2,q|3)->q|3.null->q|3.cons(q|2,q|3)->q|4.null->q|4.
s(q|0)->q|1.o->q|0.s(q|1)->q|1.s(q|1)->q|2.s(q|0)->q|2.nil End of Description]
```

où le premier champ précise si la terminaison de la relation de réduction séquentielle a pu être prouvée et le deuxième champ donne l'automate final. Dans ces calculs, l'enchaînement des opérations de base est le suivant. Tout d'abord, on calcule les automates $A_{IRR(\mathcal{R}_1)}$ et $A_{IRR(\mathcal{R}_2)}$. Ensuite, si la commande invoquée est `start(`A`)` on applique `T_up(R1)` sur A, on réalise une intersection avec $A_{IRR(\mathcal{R}_1)}$, puis on fait de même avec $\mathcal{R}_2$ sur le résultat de l'intersection. A la fin d'une normalisation complète avec $\mathcal{R}_1$ et $\mathcal{R}_2$, on vérifie s'il existe un terme réductible par $\mathcal{R}_1$ et si c'est le cas, on itère le procédé.

Si la commande invoquée est `start`, la seul différence est que l'on initialise le procédé sur $A_{IRR(\mathcal{R}_1)}$ auquel on applique `T_up(R2)`, on réalise une intersection avec $A_{IRR(\mathcal{R}_2)}$. Ensuite on vérifie s'il existe un terme réductible par $\mathcal{R}_1$ et si c'est le cas, on itère le procédé comme dans le cas de `start(`A`)`.

7.3 Quelques systèmes de réécriture et leur preuve de terminaison

Nous donnons ici quelques exemples de systèmes dont la terminaison a été prouvée à l'aide du prototype de résolution de contraintes d'ordre *gpo* et de son instance spécifique: le *lpo*(voir section 7.1).

7.3.1 Système de dérivation symbolique

Ce système, emprunté à [Der87, Gie95b], décrit un programme simplifié de dérivation symbolique.

```
var. alpha, beta.

dx(x) -> one.
dx(a) -> zero.
dx(plus(alpha, beta)) -> plus(dx(alpha), dx(beta)).
dx(times(alpha, beta)) -> plus(times(beta, dx(alpha)), times(alpha, dx(beta))).
dx(minus(alpha, beta)) -> minus(dx(alpha), dx(beta)).
dx(neg(alpha)) -> neg(dx(alpha)).
dx(div(alpha, beta)) -> minus(div(dx(alpha), beta),
                              times(alpha, div(dx(beta), exp(beta,two)))).
dx(ln(alpha)) -> div(dx(alpha), alpha).
dx(exp(alpha, beta)) -> plus(times(beta, times(exp(alpha, minus(beta,one)),
            dx(alpha))), times(exp(alpha, beta), times(ln(alpha),
            dx(beta)))).
end.
```

```
Solving is successful
Precedence: [dx > x, dx > neg, dx > two, dx > div, dx > ln, dx > minus,
dx > exp, dx > plus, dx > times, one = x, x = one, zero = a, a = zero]

Termination proof done in 10.43 seconds, cpu time

2.84 seconds, cpu time for deducting obligations of proof
7.59 seconds, cpu time for satisfiability testing
```

7.3.2 Spécification d'un circuit intégré

Ce système est une spécification simplifié de circuit électronique, utilisée dans [Bou94] comme exemple pour la preuve inductive par réécriture avec SPIKE.

```
var. i,t, x, y.

not(x + y) -> not(x) + y.
not(not(x)) -> x.
c(0, t) -> input(t).
c(s(i), t) -> c(i, t) * q(i, t).
q(i, 0) -> init(i).
q(i, s(t)) -> q(i, t) + c(i, t).
c1(0, t) -> input(t).
c1(s(i), t) -> c1(i, t) * not(q1(i, t)).
q1(i, 0) -> not(init(i)).
q1(i, s(t)) -> q1(i, t) + c1(i, t).
end.
```

```
Solving is successful
Precedence: [(not) > (+), q > init, q1 > init, c > input, c > (*),
q > (+), c1 > input, c1 > (not), c1 > (*), q = c, c = q, q1 = c1, c1 = q1]

Termination proof done in 1.92 seconds, cpu time

0.41 seconds, cpu time for deducting obligations of proof
1.51 seconds, cpu time for satisfiability testing
```

7.3.3 Le tri par insertion

Ce système décrit l'algorithme de tri par insertion ainsi que les prédicats utilisés pour la vérification des bonnes propriétés de l'algorithme (par exemple `sorted` qui est vrai si une liste est triée) à l'aide de SPIKE [BKR92, Bou94].

```
var. x, x1, x2, x4, y, z, l.

leq(0,x) -> true.
leq(s(x),0) -> false.
leq(s(x), s(y)) -> leq(x,y).
dif(0,0) -> false.
dif(0,s(x)) -> true.
dif(s(x),0) -> true.
dif(s(x),s(y)) -> dif(x,y).
length(nil) -> 0.
length(cons(x,y)) -> s(length(y)).
count(x,nil) -> 0.
count(x,cons(y,z)) -> s(count(x,z)) if [dif(x,y) = false].
count(x,cons(y,z)) -> count(x,z) if [dif(x,y) = true].
insert(x,nil) -> cons(x,nil).
insert(x,cons(y,z)) -> cons(x,cons(y,z)) if [leq(x, y) = true].
insert(x,cons(y,z)) -> cons(y,insert(x,z)) if [leq(x, y) = false].
isort(nil) -> nil.
isort(cons(x,l)) -> insert(x,isort(l)).
sorted(cons(x,cons(y,z))) -> sorted(cons(y,z)) if [leq(x, y) = true].
```

```
sorted(nil) -> true.
sorted(cons(x2, nil)) -> true.
sorted(cons(x2,cons(x1,x4))) -> false if [leq(x2,x1) = false].
end.
```

```
Solving is successful
Precedence: [length > s, sorted > 0, leq > 0, dif > 0, isort > insert,
count > s, insert > cons, false = nil, nil = false, true = false,
0 = true, true = 0, nil = 0]

Termination proof done in 1.89 seconds, cpu time

0.43 seconds, cpu time for deducting obligations of proof
1.46 seconds, cpu time for satisfiability testing
```

7.3.4 Spécification d'un système de freinage

Le système suivant, emprunté à [Aza98], décrit le calcul du freinage que doit effectuer un train à l'approche d'un obstacle en fonction du temps, de sa vitesse et de la distance restant à parcourir.

```
var. x,y,t,u.

(- x) + (- y) ->  - (x + y).
-( 0) ->  0.
x + (- x) ->  0.
x + 0 ->  x.
x + s(y) ->  s(x + y).
leq(0,0) ->  true.
leq(0,s(x)) ->  true.
leq(s(x),s(y)) ->  leq(x,y).
leq(0,vm(t)) ->  true.
leq(vm(t),vmax(t)) ->  true.
leq(0,dinit(t)) ->  true.
leq(0,afu(t)) ->  true.
leq(0,vmax(t)) ->  true.
leq(vmax(t),afu(t)) ->  true.
v(t) ->  vv(t) if [leq(0,vv(t)) = true].
v(t) ->   0  if [leq(0,vv(t)) = false].
f1(t0) ->  dinit(t0).
vv(t) ->  (f2(t) + (- afu(t))) if [fu(t) = true].
vv(t) ->  vm(t) if [fu(t) = false].
fu(t) -> leq(d(t),vm(t)).
f2(t0) -> 0 .
f2(af(t)) ->  v(t).
d(t) ->  (f1(t) + (- v(t))).
f1(af(t)) ->  d(t).
end.
```

```
Solving is successful
Precedence: [(+) > t0, leq > 0, af > v, fu > af, (+) > (-), (+) > s, v
> vv, f1 > dinit, vv > afu, vv > f2, vv > (+), vv > vm, fu > leq, d >
f1, true = t0, 0 = true, t0 = false, false = 0, 0 = false, d = af, af
= d]

Termination proof done in 6.32 seconds, cpu time

0.75 seconds, cpu time for deducting obligations of proof
5.57 seconds, cpu time for satisfiability testing
```

7.3.5 Système de freinage avec lemmes

Ce système contient les règles du système précédent auxquel sont ajouter des lemmes orientés utilisé pour la vérification de la spécification.

```
var. x,y,t,u.

(- x) + (- y) ->  - (x + y).
-( 0) ->  0.
x + (- x) ->  0.
x + 0 ->  x.
x + s(y) ->  s(x + y).
leq(0,0) ->  true.
leq(0,s(x)) ->  true.
leq(s(x),s(y)) ->  leq(x,y).
leq(0,vm(t)) ->  true.
leq(vm(t),vmax(t)) ->  true.
leq(0,dinit(t)) ->  true.
leq(0,afu(t)) ->  true.
leq(0,vmax(t)) ->  true.
leq(vmax(t),afu(t)) ->  true.
d(t) ->  (f1(t) + (- v(t))).
f1(af(t)) ->  d(t).
v(t) ->  vv(t) if [leq(0,vv(t)) = true].
v(t) ->   0  if [leq(0,vv(t)) = false].
f1(t0) ->  dinit(t0).
vv(t) ->  (f2(t) + (- afu(t))) if [fu(t) = true].
vv(t) ->  vm(t) if [fu(t) = false].
fu(t) -> leq(d(t),vm(t)).
f2(t0) -> 0 .
f2(af(t)) ->  v(t).
leq(0, x+y) -> true if[leq(0,x) = true, leq(0,y) = true].
s(x) + (-(s(y))) ->  x + (-y).
- (- x) ->  x.
leq(x + (- y), u) ->  leq(x,(y + u)).
leq(0, vv(t)) -> true.
leq(0, vv(t)) -> false.
fu(t) -> true.
fu(t) -> false.
leq(y, x) ->false if[leq(x,y) = true].
leq(x, y) -> true if[leq(x,0) = true, leq(0, y) = true].
leq(0,d(t)) -> true.
end.
```

```
Solving is successful
Precedence: [(+) > true, fu > af, af > v, leq > (+), (+) > (-),
(+) > s, d > f1, v > vv, f1 > dinit, vv > afu, vv > f2, vv > (+),
vv > vm, fu > leq, d = af, af = d, t0 = true, 0 = t0, true = false,
false = 0, 0 = false]

Termination proof done in 29.39 seconds, cpu time

1.33 seconds, cpu time for deducting obligations of proof
28.06 seconds, cpu time for satisfiability testing
```

7.3.6 Spécification d'un système de freinage avec lemmes non-orientables

Ce système contient en plus du précédent des lemmes qui ne peuvent être orientés par *lpo*.

```
var. x,y,t,u.

(- x) + (- y) ->  - (x + y).
-( 0) ->  0.
x + (- x) ->  0.
x + 0 ->  x.
x + s(y) ->  s(x + y).
leq(0,0) ->  true.
leq(0,s(x)) ->  true.
leq(s(x),s(y)) ->  leq(x,y).
leq(0,vm(t)) ->  true.
leq(vm(t),vmax(t)) ->  true.
leq(0,dinit(t)) ->  true.
leq(0,afu(t)) ->  true.
leq(0,vmax(t)) ->  true.
leq(vmax(t),afu(t)) ->  true.
d(t) ->  (f1(t) + (- v(t))).
f1(af(t)) ->  d(t).
v(t) ->  vv(t) if [leq(0,vv(t)) = true].
v(t) ->   0  if [leq(0,vv(t)) = false].
f1(t0) ->  dinit(t0).
vv(t) ->  (f2(t) + (- afu(t))) if [fu(t) = true].
vv(t) ->  vm(t) if [fu(t) = false].
fu(t) -> leq(d(t),vm(t)).
f2(t0) -> 0 .
f2(af(t)) ->  v(t).
leq(0, x+y) -> true if[leq(0,x) = true, leq(0,y) = true].
s(x) + (-(s(y))) ->  x + (-y).
- (- x) ->  x.
leq(x + (- y), u) ->  leq(x,(y + u)).
leq(x,(u + (- y))) -> leq((x + y), u).
leq((- x),(- y)) ->  leq(y,x).
leq(0, vv(t)) -> true.
leq(0, vv(t)) -> false.
fu(t) -> true.
fu(t) -> false.
leq(y, x) ->false if[leq(x,y) = true].
leq(x, y) -> true if[leq(x,0) = true, leq(0, y) = true].
afu(t) -> afu(t0).
```

```
vmax(t) -> vmax(t0).
dinit(t) -> dinit(t0).
leq(0,d(t)) -> true.
end.
```

```
Rule dinit(t) -> dinit(t0) cannot be oriented by lpo!
Rule vmax(t) -> vmax(t0) cannot be oriented by lpo!
Rule afu(t) -> afu(t0) cannot be oriented by lpo!
Rule leq(-(x), -(y)) -> leq(y, x) cannot be oriented by lpo!
Rule leq(x, +(u, -(y))) -> leq(+(x, y), u) cannot be oriented by lpo!
Echec de la preuve !

Time: 7.51 seconds, cpu time
```

7.3.7 Spécification de l'algorithme de conformité ABR

Ce système (non-diffusable) proposé dans le cadre d'un projet commun avec le CNET, spécifie un algorithme de conformité utilisé dans un protocole de communication de type ATM. Il contient 35 règles conditionnelles. La précédence solution est calculée en 8.92 secondes.

7.3.8 Le tour de carte de Gilbreath

```
var. x,y,z, x1, x2, x3, y1, y2, y3.

neg(r) ->b.
neg(b) ->r.
paired(r,b) -> true.
paired(b,r) -> true.
paired(x,x) -> false.
append(null,y) -> y.
append(cons(x,y),z) -> cons(x,append(y,z)).
rotate(null) -> null.
rotate(cons(x,y)) -> append(y,cons(x,null)).
even(null) -> true.
even(cons(x,null)) -> false.
even(cons(x1,cons(x2,y))) -> even(y).
opposite(null,y) -> false.
opposite(x,null) -> false.
opposite(cons(x1,y1),cons(x2,y2)) -> paired(x1,x2).
pairedlist(null) -> true.
pairedlist(cons(x,null)) -> true.
pairedlist(cons(x1,cons(x2,x))) -> pairedlist(x) if [paired(x1,x2) = true].
pairedlist(cons(x1,cons(x2,x))) -> false if[paired(x1,x2)=false].
alter(null) -> true.
alter(cons(x1,null)) -> true.
alter(cons(x1,cons(x2,x))) -> alter(cons(x2,x)) if [paired(x1,x2) = true].
alter(cons(x1,cons(x2,x))) -> false if [paired(x1,x2) = false].
shuffle(null,null,null) ->  true.
shuffle(x1,cons(x2,x3),null) -> false.
shuffle(cons(x1,x2),x3,null) -> false.
shuffle(null,null,cons(x1,x2)) -> false.
shuffle(cons(x1,y1),null,cons(x3,y3))-> false if [paired(x1,x3) = true].
shuffle(cons(x1,y1),null,cons(x3,y3)) -> shuffle(y1,null,y3)
```

```
     if[paired(x1,x3) = false].

shuffle(null,cons(x2,y2),cons(x3,y3)) -> false if [paired(x2,x3) = true].
shuffle(null,cons(x2,y2),cons(x3,y3)) -> shuffle(null,y2,y3)
     if [paired(x2,x3) = false].

shuffle(cons(x1,y1),cons(x2,y2),cons(x3,y3)) -> true
     if [paired(x1,x3) = false, shuffle(y1,cons(x2,y2),y3)=true].

shuffle(cons(x1,y1),cons(x2,y2),cons(x3,y3)) -> true
     if [paired(x2,x3) = false, shuffle(cons(x1,y1),y2,y3)=true].

shuffle(cons(x1,y1),cons(x2,y2),cons(x3,y3)) -> false
     if [paired(x1,x3) = true, paired(x2,x3) = true].

shuffle(cons(x1,y1),cons(x2,y2),cons(x3,y3)) -> false
     if [shuffle(y1,cons(x2,y2),y3)=false, shuffle(cons(x1,y1),y2,y3)=false].

shuffle(cons(x1,y1),cons(x2,y2),cons(x3,y3)) -> false
     if [paired(x1,x3) = true, shuffle(cons(x1,y1),y2,y3)=false].
shuffle(cons(x1,y1),cons(x2,y2),cons(x3,y3)) ->false
     if [paired(x2,x3) = true, shuffle(y1,cons(x2,y2),y3)=false].
end.
```

```
Solving is successful
Precedence: [rotate > r, even > r, pairedlist > r, alter > r,
shuffle > r, paired > r, append > cons, rotate > append,
opposite > paired, r = null, true = r, null = b, b = false, true = b,
false = true]

Termination proof done in 9.09 seconds, cpu time

1.54 seconds, cpu time for deducting obligations of proof
7.55 seconds, cpu time for satisfiability testing
```

7.3.9 Arrangement sans division et soustraction

Ce système est une sous-partie du calcul des arrangements de la section 6.6.5.

```
var. x,y,n,p.

Ar(n, p) -> div(fact(n), fact(minus(n, p))).
fact(s(x)) -> mult(s(x), fact(x)).
fact(0) -> s(0).
mult(0, x) -> 0.
mult(s(x), y) -> plus(mult(x, y), y).
plus(x, 0) -> x.
plus(x, s(y)) -> s(plus(x, y)).
end.
```

```
Solving is successful
Precedence: [ar > minus, ar > fact, ar > div, fact > mult, mult > plus,
plus > s]

Termination proof done in 0.42 seconds, cpu time

0.18 seconds, cpu time for deducting obligations of proof
0.24 seconds, cpu time for satisfiability testing
```

7.3.10 Parties de systèmes

Ces deux systèmes sont les deux modules de l'exemple de la section 6.6.5, utilisant les entiers compressés.

```
var. x, y.

s(s(0)) -> ds(0).
s(s(ds(x))) -> ds(ds(x)).
plus(0, x) -> x.
plus(s(x), y) -> s(plus(x, y)).
plus(ds(x), y) -> ds(plus(x, y)).
mult(0, x) -> 0.
mult(s(x), y) -> plus(mult(x, y), y).
mult(ds(x), y) -> plus(y, plus(y, mult(x, y))).
end.
```

```
Solving is successful
Precedence: [plus > s, mult > plus, ds = s, s = ds]

Termination proof done in 0.51 seconds, cpu time

0.2 seconds, cpu time for deducting obligations of proof
0.31 seconds, cpu time for satisfiability testing
```

```
var. x, y.

cs(x) -> s(s(x)).
leq(0, x) -> true.
leq(s(x), 0) -> false.
leq(s(x), s(y)) -> leq(x, y).
end.
```

```
Solving is successful
Precedence: [cs > s, true = false, 0 = true, false = 0]

Termination proof done in 0.13 seconds, cpu time

0.06 seconds, cpu time for deducting obligations of proof
0.07 seconds, cpu time for satisfiability testing
```

7.4 Graphes d'$\mathcal{O}$-preuves

Dans cette section sont regroupés quelques-uns des graphes de $\mathcal{O}$-preuves engendrés par notre prototype et visualisés à l'aide de CGRAPH (voir section 7.1.2).

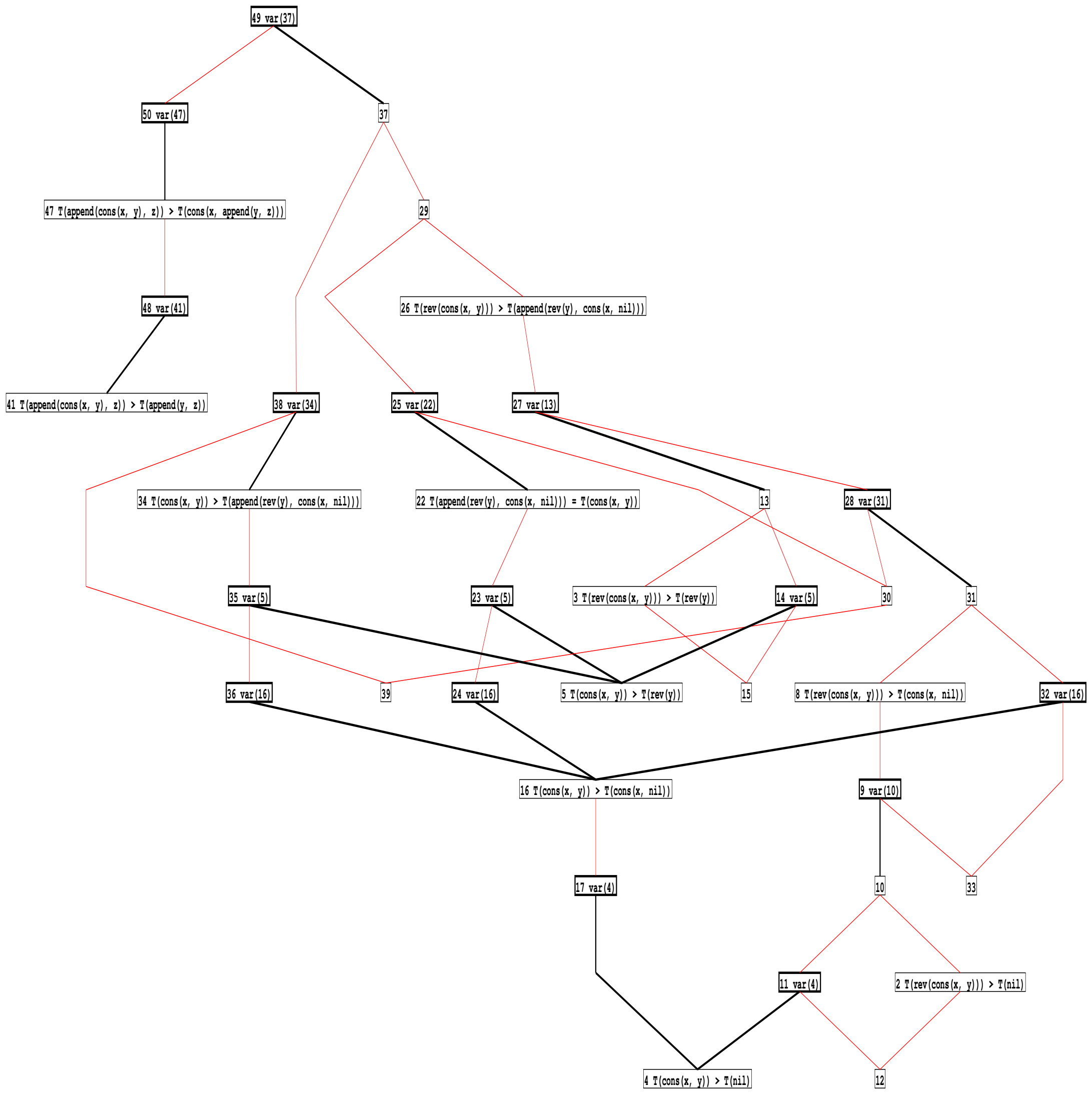


FIG. 7.1 – *Exemple de DAG de $\mathcal{O}$-preuve "non-déplié"*

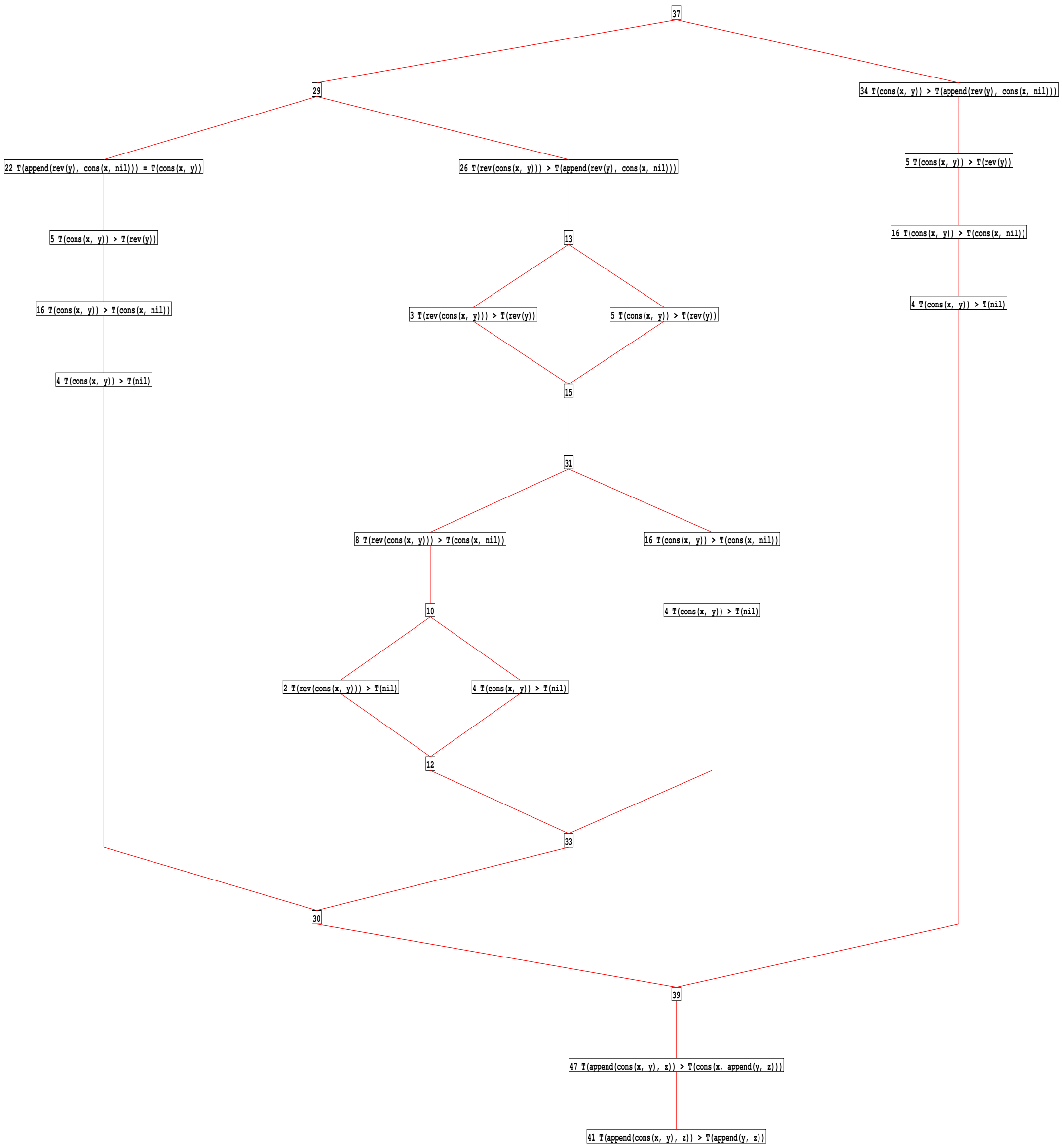


FIG. 7.2 – *Exemple de DAG de $\mathcal{O}$-preuve "déplié"*

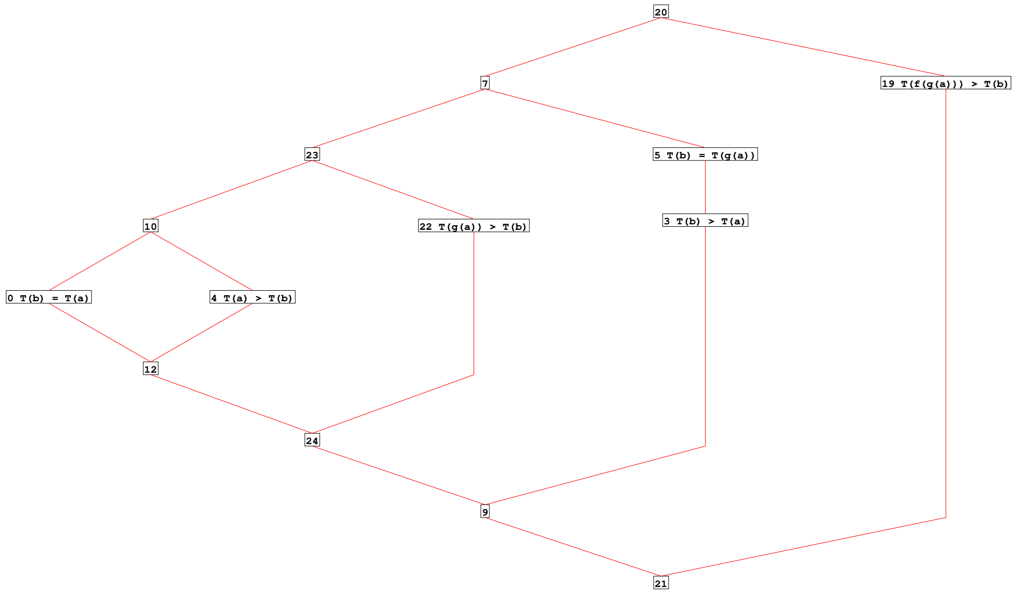


FIG. 7.3 – *Exemple de DAG de $\mathcal{O}$-preuve privilégiant les égalités*

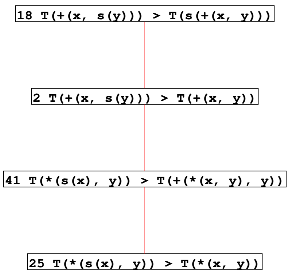


FIG. 7.4 – *DAG de $\mathcal{O}$-preuve avant instanciation*

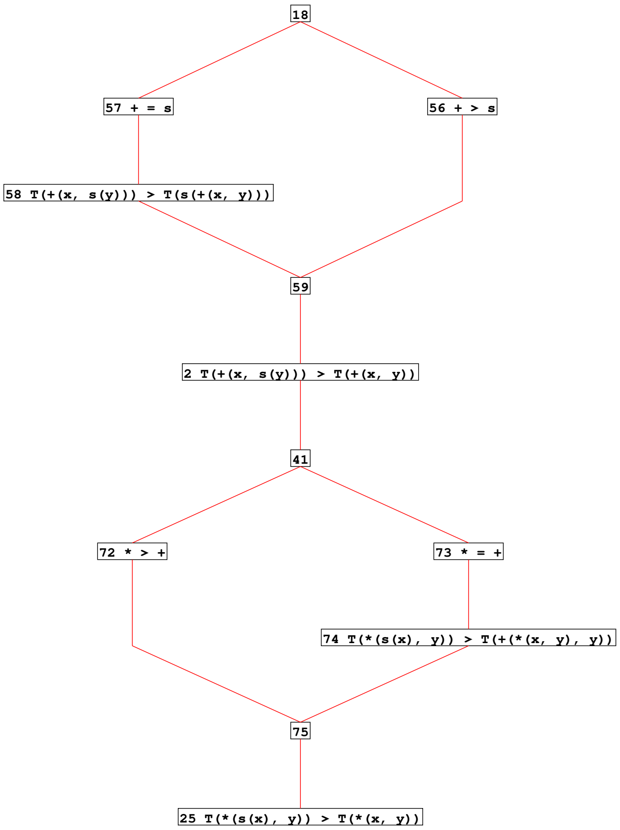


FIG. 7.5 – *DAG de $\mathcal{O}$-preuve après IPG pour la précédence*

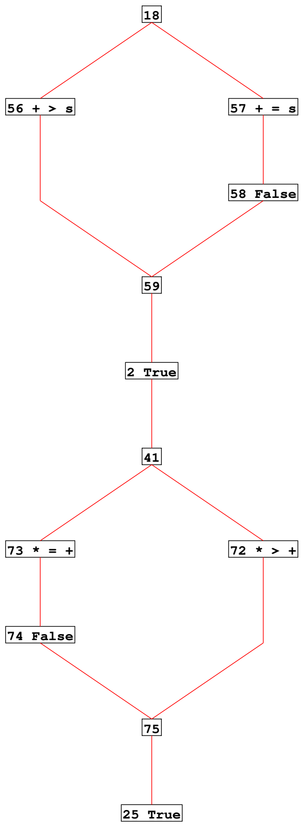


FIG. 7.6 – *DAG de $\mathcal{O}$-preuve après IPG pour l'extension lexicographique*

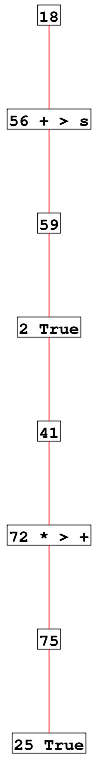


FIG. 7.7 – *DAG de $\mathcal{O}$-preuve après simplification*

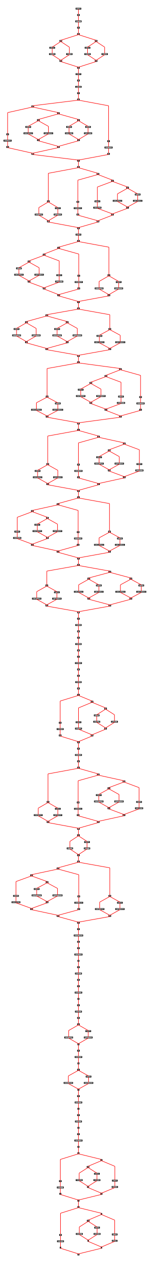


FIG. 7.8 – *DAG de $\mathcal{O}$-preuve de l'exemple 7.3.4*

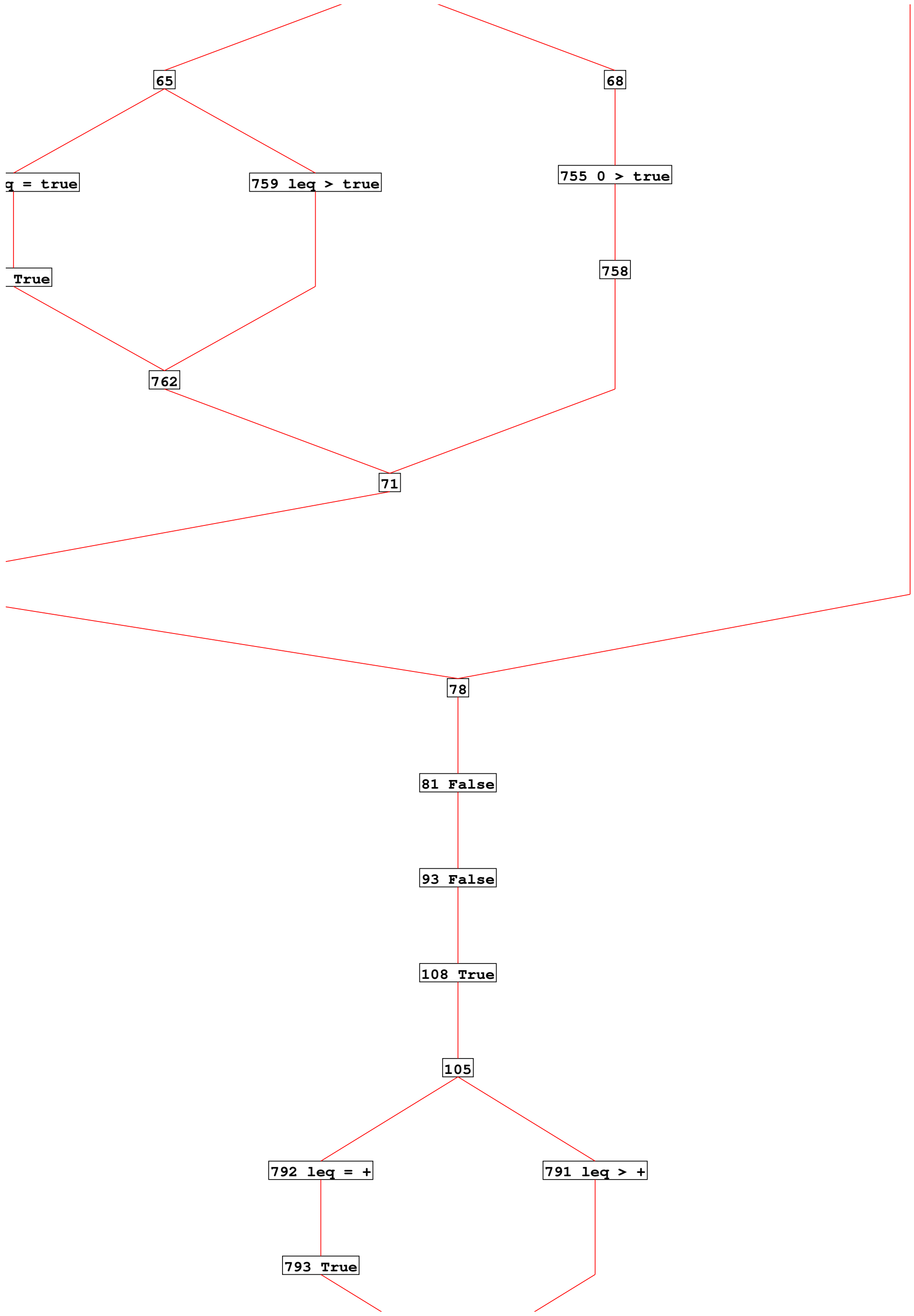


FIG. 7.9 – *Zoom sur les contraintes insatisfaisables dans le DAG de l'O-preuve globale de l'exemple 7.3.6*

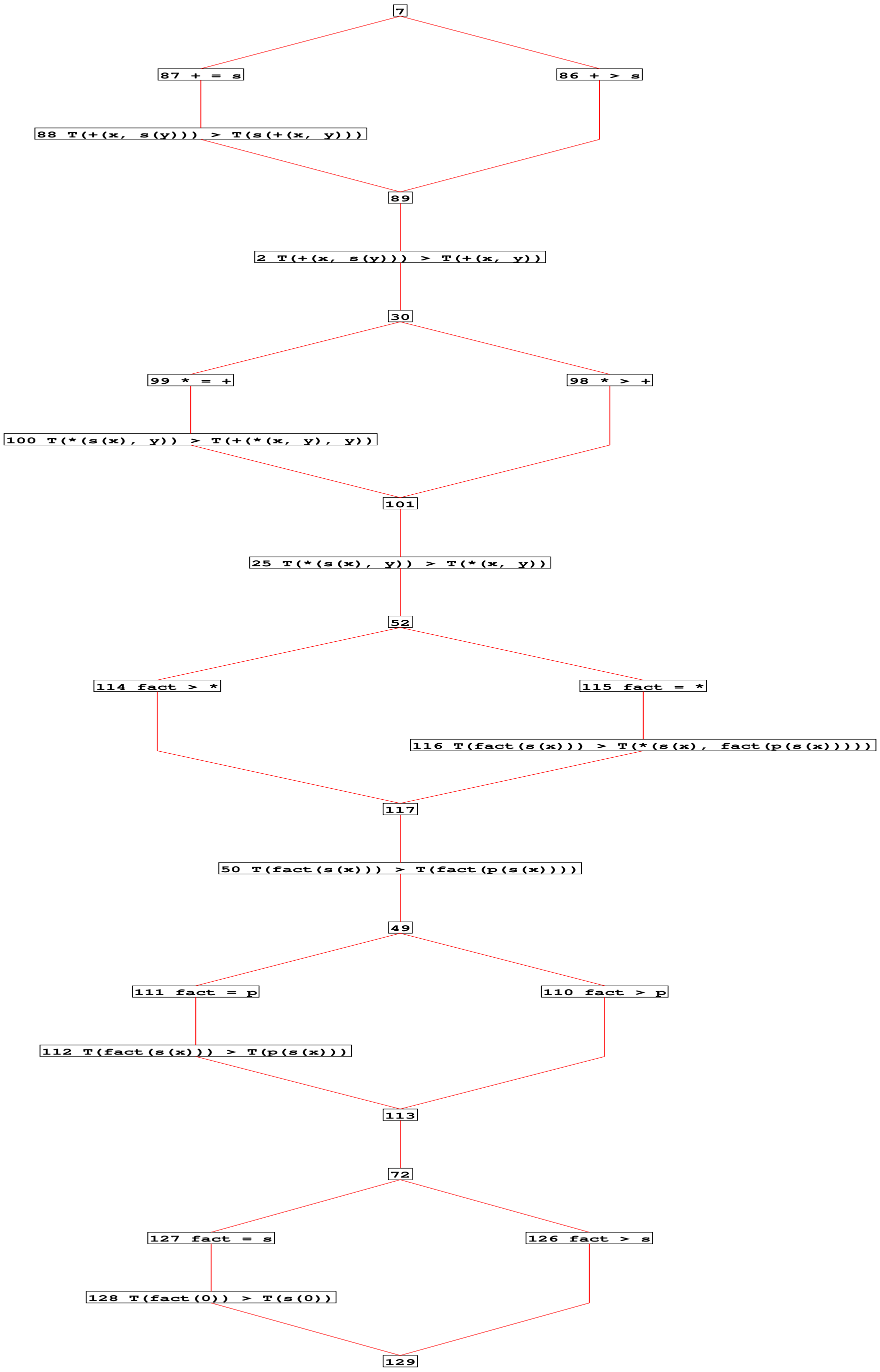


FIG. 7.10 – *DAG de $\mathcal{O}$-preuve de factorielle (section 5.4)*

Chapitre 8

Synthèse

Dans les chapitres précédents, nous avons vu plusieurs outils et techniques dédiées à la preuve de terminaison en réécriture. Dans ce chapitre, nous nous proposons d'en faire une synthèse sous deux éclairages différents: l'un théorique et l'autre plus pratique. En tant que synthèse théorique, dans la section 8.1, nous présentons informellement un nouveau critère de terminaison qui, en combinant contraintes d'ordre et automates d'arbres, permet d'exploiter les notions de stratégies et d'ensemble de termes initiaux dans la preuve de terminaison. D'un point de vue plus pratique, nous décrivons, dans la section 8.2, ce que pourrait être l'architecture d'un environnement de preuve de terminaison construit sur les principaux outils vus dans les chapitres précédents. Nous donnons, notamment, différentes combinaisons intéressantes d'outils et discutons leur efficacité en terme de puissance de preuve et de vitesse de calcul.

8.1 Allier contraintes d'ordre et contraintes ensemblistes

Nos réflexions sur les travaux exposés précédemment nous ont amenés à rechercher des critères de terminaison moins restrictifs que les critères usuels de façon à prendre en compte des stratégies d'application de règles et les ensembles de termes initiaux. Comme on l'a vu, la propriété de terminaison est souvent trop forte et seule la terminaison pour une stratégie donnée ou sur un domaine donné est nécessaire. Par exemple, dans le cas des programmes en logique de réécriture, bien souvent seule la preuve de terminaison du système de réécriture pour la stratégie innermost et sur une classe de termes initiaux (ou requêtes) est nécessaire. D'autre part, lorsque le système de réécriture ne termine pas de façon générale, il existe parfois des stratégies particulières ou des sous-ensembles de termes initiaux pour lesquels la terminaison est démontrable.

Il existe des méthodes de preuve de terminaison dans le cas de la stratégie innermost [AG97b, DH95]. D'autre part, la preuve de terminaison de systèmes de réécriture typés [Gna92, OCL96], s'apparente à la restriction de la preuve de terminaison à un sous-ensemble de termes initiaux. Pour notre part, nous avons proposé dans la section 6.6.5 un critère de terminaison pour la stratégie de réduction séquentielle et pour un ensemble donné de termes initiaux. Cependant, il n'existe pas de méthode générale pouvant s'adapter à toute stratégie d'application des règles et à tout ensemble de termes initiaux. Ceci est dû au fait que la notion d'ordre utilisée dans les grands théorèmes de terminaison s'accorde mal avec ces restrictions. Pour la prise en compte de ces deux restrictions, au lieu d'imposer $l > r$ pour toutes les règles $l \to r$ du système comme dans les preuves traditionnelles, il semble naturel de n'imposer la décroissance que sur les *chaînes de réécriture admissibles*, i.e. les chaînes débutant d'un terme appartenant à l'ensemble de termes initiaux et dont les pas de réécriture respectent la stratégie.

Nous ébauchons ici une autre méthode de preuve de terminaison qui permettrait de capturer de façon générale les restrictions liées aux stratégies et celles liées aux ensembles de termes initiaux. Au lieu de montrer la terminaison de la relation de réécriture, l'idée est d'établir la terminaison de toutes les chaînes de réécriture à partir d'un terme ou d'un ensemble de termes. On dira qu'un terme t *termine* s'il n'existe aucune chaîne infinie de réécriture issue de t. Soit $\mathcal{R}$ un système de réécriture, et E un ensemble de termes muni d'un ordre $>$ bien fondé possédant la propriété de sous-terme. Nous nous proposons de montrer la terminaison des termes de E par induction sur E muni de l'ordre $>$.

Tout d'abord, nous montrons comment il est possible de capturer la notion de stratégie, par exemple la stratégie innermost (voir section 3.2.2). Soit $\mathcal{F} = \{fact : 1, \times : 2, + : 2, s : 1, 0 : 0\}$ et $\mathcal{R}$ le système de réécriture suivant, emprunté à [DH95], qui définit la factorielle:

$$
\begin{array}{ll}
p(s(x)) \to x & (1)\\
fact(0) \to s(0) & (2)\\
fact(s(x)) \to s(x) \times fact(p(s(x))) & (3)\\
0 \times x \to 0 & (4)\\
s(x) \times y \to (x \times y) + y & (5)\\
x + 0 \to x & (6)\\
x + s(y) \to s(x + y) & (7)
\end{array}
$$

Il est impossible de montrer la terminaison de ce système à l'aide d'un ordre de simplification car, dans la règle $fact(s(x)) \to s(x) \times fact(p(s(x)))$, le membre gauche est plongé dans le membre droit. Ici, nous allons montrer la terminaison de ce système sous la stratégie innermost pour un ensemble de termes initiaux $E = \mathcal{T}(\mathcal{F})$. On suppose que $\mathcal{T}(\mathcal{F})$ est muni d'un ordre de simplification $>$ (ordre bien fondé, monotone et ayant la propriété de sous-terme). Soit P la propriété suivante: un terme $t \in \mathcal{T}(\mathcal{F})$ a la propriété P, noté $P(t)$, si t termine et pour tout terme u tel que $t \underset{in}{\longrightarrow}_{\mathcal{R}}^{+} u$ et $u \in IRR(\mathcal{R})$, on a $t > u$. Nous montrons maintenant par induction que pour tout $t \in \mathcal{T}(\mathcal{F})$ on a $P(t)$. Un point important est que nous construisons l'ordre d'induction $>$ au fur et à mesure, en construisant un ensemble de contraintes d'ordre C que doit satisfaire $>$. Initialement, l'ensemble de contraintes C est vide; la seule contrainte implicite est que $>$ doit être un ordre de simplification, ce qui assure que $>$ a la propriété de sous-terme et la monotonie. Soit $f \in \mathcal{F}^n$, $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$, et $t = f(t_1, \ldots, t_n)$. Trois cas peuvent se présenter. Soit

1. t est en forme normale. Dans ce cas, on a trivialement $P(t)$ puisque t termine et qu'il n'existe pas de terme u tel que $t \underset{in}{\longrightarrow}_{\mathcal{R}}^{+} u$.

2. tous les sous-termes de t sont en forme normale, i.e. $\forall i = 1, \ldots, n$: $t_i \in IRR(\mathcal{R})$, et t n'est pas en forme normale. Dans ce cas, seule une réécriture au sommet du terme t (à la position ϵ) peut avoir lieu. Nous montrons que pour toute chaîne de réécriture innermost $t \underset{in}{\longrightarrow}_{\mathcal{R}} u_1 \underset{in}{\longrightarrow}_{\mathcal{R}} t_2 \ldots$ issue de t il existe un entier $i \in \mathbb{N}$ tel que $t > t_i$. Pour ce faire, nous considérons tous les termes u_1 tels que $t \underset{in}{\longrightarrow}_{\mathcal{R}} u_1$ et nous vérifions que $C \cup \{t > u_1\}$ est une contrainte satisfaisable. Si pour un u_1 particulier ce n'est pas le cas, alors nous considérons tous les termes u_2 tels que $u_1 \underset{in}{\longrightarrow}_{\mathcal{R}} u_2$ et nous vérifions que $C \cup \{t > u_2\}$ est satisfaisable, et ainsi de suite. Pour connaître tous les termes u_1 il suffit de raisonner par cas sur les règles de réécriture applicables. Traitons le cas de notre exemple:

   - $t = p(s(t_1)) \to t_1$. Comme nous avons supposé que $>$ a la propriété de sous-terme, nous savons que la contrainte $p(s(t_1)) > t_1$ est trivialement vérifiée par $>$.

- $t = fact(0) \rightarrow s(0)$. Contrairement au cas précédent, tout ordre de simplification $>$ ne satisfait pas nécessairement l'inégalité $fact(0) > s(0)$. Or, il est facile de trouver un ordre de simplification (par exemple un *lpo*) qui vérifie $C \cup \{fact(0) > s(0)\}$ (C était vide jusqu'ici). Donc, C devient $C = \{fact(0) > s(0)\}$ et nous poursuivons la preuve.
- $t = fact(s(t_1)) \rightarrow s(t_1) \times fact(p(s(t_1))) = u_1$. Ici, $C \cup \{t > u_1\}$ n'est satisfaisable par aucun ordre de simplification. Or, comme le terme u n'est pas en forme normale, nous choisissons donc de raisonner *un pas plus loin*, i.e. sur tous les termes u_2 tels que $u_1 \underset{in}{\longrightarrow}_{\mathcal{R}} u_2$. Comme t_1 est en forme normale, la seule réécriture innermost pouvant intervenir sur le terme u_1 porte nécessairement sur le sous-terme $p(s(t_1))$ qui se réécrit obligatoirement en t_1. On a donc $u_1 = s(t_1) \times fact(p(s(t_1))) \rightarrow s(t_1) \times fact(t_1) = u_2$ et il s'agit bien de la seule réécriture innermost possible. Or, $C \cup \{t > u_2\}$ est satisfaisable par un ordre de simplification (*lpo* par exemple). L'ensemble de contraintes C devient donc $C = \{fact(0) > s(0),\ fact(s(t_1)) > s(t_1) \times fact(t_1)\}$.
- $t = 0 \times t_1 \rightarrow 0$. La contrainte $0 \times t_1 > 0$ est vraie pour toute ordre de simplification $>$, il est inutile de l'ajouter à C comme dans le cas de la règle (1).
- $t = s(t_1) \times t_2 \rightarrow (t_1 \times t_2) + t_2$. La contrainte $C \cup \{s(t_1) \times t_2 > (t_1 \times t_2) + t_2\}$ est satisfaisable. C devient $C = \{fact(0) > s(0),\ fact(s(t_1)) > s(t_1) \times fact(t_1),\ s(t_1) \times t_2 > (t_1 \times t_2) + t_2\}$.
- $t = t_1 + 0 \rightarrow t_1$. Aucune contrainte supplémentaire n'est nécessaire, comme dans le cas de la règle (1).
- $t = t_1 + s(t_2) \rightarrow s(t_1 + t_2)\}$. La contrainte $C \cup \{t_1 + s(t_2) > s(t_1 + t_2)\}$ est satisfaisable. L'ensemble C devient $C = \{fact(0) > s(0),\ fact(s(t_1)) > s(t_1) \times fact(t_1),\ s(t_1) \times t_2 > (t_1 \times t_2) + t_2,\ t_1 + s(t_2) > s(t_1 + t_2)\}$.

Si l'ensemble de contraintes est satisfaisable alors pour toute chaîne de réécriture innermost $t \underset{in}{\longrightarrow}_{\mathcal{R}} u_1 \underset{in}{\longrightarrow}_{\mathcal{R}} u_2 \ldots$ issue de t il existe un entier $i \in \mathbb{N}$ tel que $t > u_i$. Or, à partir de $t > u_i$, en appliquant l'hypothèse d'induction, on obtient $P(u_i)$, i.e. u_i termine et pour tout terme $u'_i \in IRR(\mathcal{R})$ tel que $u_i \underset{in}{\longrightarrow}_{\mathcal{R}}^{+} u'_i$, on a $u_i > u'_i$. Ainsi, comme toutes les chaînes issues de t passent par un terme qui termine, on en déduit que t termine. D'autre part, de $t > u_i$ et $u_i > u'_i$, et de la transitivité de $>$ on déduit que $t > u'_i$. D'où finalement, on a $P(t)$.

3. les sous-termes de t ne sont pas tous en forme normale, i.e. $\exists i = 1, \ldots, n$ tel que $t_i \notin IRR(\mathcal{R})$. Suivant la stratégie innermost, il est d'abord nécessaire de normaliser les sous-termes. Par hypothèse, l'ordre $>$ a la propriété de sous-terme, d'où $\forall i = 1, \ldots, n$: $f(t_1, \ldots, t_n) > t_i$. Par induction, on en déduit donc que $\forall i = 1, \ldots, n : P(t_i)$, i.e. pour tout sous-terme $t_i \notin IRR(\mathcal{R})$, il existe $t'_i \in IRR(\mathcal{R})$ tel que $t_i \underset{in}{\longrightarrow}_{\mathcal{R}}^{+} t'_i$ et $t_i > t'_i$. La stratégie innermost privilégiant les réécritures dans les sous-termes par rapport à la réécriture au sommet, on en déduit que le terme t est nécessairement réduit de la façon suivante: $f(t_1, \ldots, t_n) \underset{in}{\longrightarrow}_{\mathcal{R}}^{+} f(t'_1, \ldots, t'_n)$. Ensuite, du fait que pour tous les sous-termes t_i réduits on a $t_i > t'_i$, et par la propriété de monotonie de l'ordre $>$, on obtient que $f(t_1, \ldots, t_n) > f(t'_1, \ldots, t'_n)$. Il suffit alors de montrer $P(f(t'_1, \ldots, t'_n))$ avec $t'_1, \ldots, t'_n \in IRR(\mathcal{R})$ pour assurer $P(f(t_1, \ldots, t_n))$. Nous nous ramenons donc au cas précédent.

L'existence d'un ordre de simplification tel que C est satisfaisable est assuré pendant tout le processus. Grâce à notre prototype, étape après étape, il est notamment possible de vérifier automatiquement qu'il existe bien un *lpo* satisfaisant C. Ainsi, grâce à la puissance du raisonnement par induction, nous venons de montrer la terminaison innermost d'un système de réécriture non-simplifiant à l'aide d'un ordre de simplification.

Il faut remarquer que, dans cet exemple, nous avons bénéficié du fait que lors de la réécriture d'un terme au sommet, la stratégie innermost impose que tous les sous-termes soient en forme normale par rapport à $\mathcal{R}$. Cependant, il est possible de relâcher cette condition sur la stratégie en choisissant un préordre $\geq$ bien fondé, ayant la propriété de sous-terme et tel que lors de toutes les étapes de réécriture, tous les termes soient *au moins équivalents*, i.e. $s \to_{\mathcal{R}} t \implies s \geq t$. La propriété $P(t)$ devient alors: t termine et pour tout terme u tel que $t \to_{\mathcal{R}}{}^{+} u$ on a $t \geq u$. Si pour tout terme t, on étudie de la même façon toutes les chaînes de réécriture possibles issues de t: trois cas sont possibles.

1. Soit t est irréductible et il est facile de conclure.

2. Soit la première réécriture de la chaîne a lieu au sommet de t et l'on procède par cas sur les règles comme dans l'exemple précédent. Là encore, plusieurs cas sont possibles: soit la première réécriture $t \to_{\mathcal{R}} u_1$ est telle que $C \cup \{t > u_1\}$ est une contrainte satisfaisable, soit on étudie tous les termes u_2 tels que $u_1 \to_{\mathcal{R}} u_2$, et ainsi de suite, sachant que tous ces termes restent comparables, i.e. $t \geq u_1$, $u_1 \geq u_2, \ldots$.

3. Soit la première réécriture de la chaîne $t \to_{\mathcal{R}} u_1$ n'a pas lieu au sommet. Or, on sait par induction que les sous-termes $t_1, \ldots, t_n$ de t terminent (puisque $t > t_i$, $i = 1 \ldots n$). Il ne peut donc y avoir de chaîne infinie de réécriture dans les sous-termes; il existe donc nécessairement un terme u_i et une chaîne de réécriture telle que $t \to_{\mathcal{R}}{}^{+} u_i$ (finiment) et u_i est irréductible ou se réécrit au sommet en u_{i+1}. De plus comme $t \geq u_i$ et t termine si u_i termine, $P(t)$ est ramené au problème de $P(u_i)$. Or, $P(u_i)$ se montre comme dans les cas précédents.

Le lecteur attentif aura remarqué qu'en exigeant $s \to_{\mathcal{R}} t \implies s \geq t$, on retrouve des conditions très proches de celles utilisées dans la méthode des paires de dépendance (voir théorème 4.2).

Maintenant nous voyons comment capturer, à la fois, les restrictions dues à une stratégie et à un ensemble de termes initiaux. Soit $\mathcal{R}$ un système de réécriture et $E \subseteq \mathcal{T}(\mathcal{F})$ un ensemble de termes initiaux. On souhaite montrer que tout terme $t \in E$ termine. Tout d'abord, il faut remarquer qu'il n'est pas toujours possible d'utiliser n'importe quel E comme domaine d'induction puisqu'en réécrivant un terme s de E à l'aide d'une règle de $\mathcal{R}$ il est possible d'obtenir un terme $t \notin E$. Dans ce cas, il est impossible d'appliquer une hypothèse d'induction sur t qui est en dehors du domaine. Ce problème ne s'était pas posé dans l'exemple précédent car nous raisonnions sur $\mathcal{T}(\mathcal{F})$, d'où il n'est pas possible de sortir par $\to_{\mathcal{R}}$. En revanche, au lieu de raisonner sur E, il est possible de fixer $\mathcal{R}^*(E)$ comme domaine d'induction. En effet, toute réécriture appliquée sur n'importe quel terme de $\mathcal{R}^*(E)$ donne nécessairement un terme de $\mathcal{R}^*(E)$. Ici, nous allons représenter l'ensemble des termes initiaux à l'aide d'un automate d'arbres et restreindre notre domaine d'induction aux termes reconnus par l'automate $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ approximant $\mathcal{R}^*(\mathcal{L}(A))$.

Soit $\mathcal{R}$ un système de réécriture soumis à une double stratégie: innermost pour le choix des positions de réécriture et une priorité sur les règles pour le choix de la règle à appliquer: on applique toujours en premier la règle qui a la plus grande priorité. Soit $\mathcal{F} = \{0:0, s:1, +:2\}$

et $\mathcal{R}$ le système de réécriture suivant dans lequel les règles sont données par ordre décroissant de priorité:

$$\begin{array}{ll} 0 + x \to x & (1) \\ (x + y) + z \to x + (y + z) & (2) \\ x + y \to y + x & (3) \end{array}$$

et soit E l'ensemble de termes initiaux représenté par l'automate d'arbres $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ tel que $\mathcal{Q} = \{q_0, q_1\}$, $\mathcal{Q}_f = \{q_0\}$ et $\Delta = \{$

$$\begin{array}{l} 0 \to q_0 \\ s(q_0) \to q_0 \\ q_0 + q_1 \to q_0 \\ q_1 + q_0 \to q_0 \\ 0 \to q_1\} \end{array}$$

L'automate A reconnaît l'ensemble des termes $E = \{0, s(x), x + 0, 0 + x \mid x \in E\}$. Il est clair que le système $\mathcal{R}$ ne termine pas de façon générale, notamment à cause de la règle de commutativité $x + y \to y + x$. De plus, même si l'on applique les règles de ce système avec une stratégie innermost et en respectant les priorités, ce système ne termine pas sur $\mathcal{T}(\mathcal{F})$. Par exemple, le terme $s(0) + s(0)$ ne peut être réécrit que par la dernière règle, mais il peut l'être de façon infinie. De la même façon, le système $\mathcal{R}$ ne termine pas sur $\mathcal{L}(A)$ si l'on applique les règles sans priorités: le terme $0+0$ peut être réécrit sans fin par la règle (3). Nous montrons maintenant que ce système termine sur $\mathcal{L}(A)$ si l'on applique les règles de façon innermost et en respectant les priorités. En restreignant notre domaine d'induction à $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$, nous espérons éliminer, par exemple, le cas des termes $u + v$, $u, v \in \{s(0), s(s(0)), \dots\}$ dont la terminaison ne peut pas être montrée. Tout d'abord, nous calculons l'automate $\mathcal{T}_\mathcal{R} \uparrow (A) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$ reconnaissant un sur-ensemble de $\mathcal{R}^*(\mathcal{L}(A))$:

```
[] result term:
    Description of A(0)states q|3.q|0.q|1.nil final states q|0.nil
transitions plus(q|3,q|0)->q|0.plus(q|3,q|3)->q|3.plus(q|1,q|3)->q|3.
plus(q|1,q|1)->q|3.plus(q|0,q|3)->q|0.o->q|0.s(q|0)->q|0.
plus(q|1,q|0)->q|0.plus(q|0,q|1)->q|0.o->q|1.o->q|3.nil End of Description
```

Ensuite nous raisonnons sur tous les termes de $\mathcal{T}_\mathcal{R} \uparrow (A)$, et nous montrons que $\forall t \in \mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A)) : P(t)$, i.e. t termine et pour tout terme u tel que $t \underset{in}{\longrightarrow}_\mathcal{R}^+ u$ et $u \in IRR(\mathcal{R})$, on a $t > u$.

. Soit $t = f(t_1, \dots, t_n) \in \mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$. Comme dans l'exemple précédent, en supposant que l'ordre $>$ est un ordre de simplification, le cas d'un terme t dont les sous-termes $t_1, \dots, t_n$ ne sont pas en forme normale peut être ramené au cas d'un terme t' tel que $t > t' = f(t'_1, \dots, t'_n)$ tel que $t'_1, \dots, t'_n \in IRR(\mathcal{R})$. Nous nous intéressons donc exclusivement au cas d'un terme $t = f(t_1, \dots, t_n)$ tel que $t_1, \dots, t_n \in IRR(\mathcal{R})$. Si le terme est irréductible au sommet, comme dans le cas précédent, t termine et on a trivialement $P(t)$. Si le terme est réductible au sommet, on procède par cas sur les règles. Notons que pour chaque cas: (a) t est une instance de membre gauche de règle, (b) les sous-termes de t sont en forme normale, (c) t est un terme ou un sous-terme de $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$ et, enfin, comme il s'agit d'un système à priorité, (d) si l'on considère un terme instance du membre gauche d'une règle $l \to r$ alors ce terme ne peut être instance du membre gauche d'une règle $l' \to r'$ plus prioritaire. Selon les cas, on utilisera une ou plusieurs de ces assertions. Initialement, l'ensemble de contraintes C est vide et $>$ est supposé être un ordre de simplification.

- $t = 0 + t_1 \to t_1$. Cette contrainte est satisfaite par n'importe quel ordre de simplification, il est inutile de l'ajouter à C.

- $t = (t_1+t_2)+t_3 \to t_1+(t_2+t_3) = t'$. Dans ce cas, on ajoute la contrainte $t = (t_1+t_2)+t_3 > t_1 + (t_2 + t_3) = t'$ à C qui reste satisfaisable.

- $t = t_1 + t_2 \to t_2 + t_1 = u$. La contrainte $t_1 + t_2 > t_2 + t_1$ n'est pas satisfaisable quelle que soit l'ordre de simplification $>$. Or, puisqu'il s'agit d'un système à priorité, on sait que pour si on peut appliquer cette règle, on n'a pas pu appliquer les règles précédentes. On en tire les diséquations suivantes: $t_1 \neq 0$ et $\forall h, k \in \mathcal{T}(\mathcal{F})\ t_1 \neq h + k$. D'autre part, on sait que t doit être reconnu par un des états[25] de $\mathcal{T}_\mathcal{R} \uparrow (A)$, i.e. $\exists q \in \mathcal{Q}$ s.t. $t \to_{\Delta'} q$. Si l'on applique l'algorithme de filtrage de la section 6.2.3 au problème $t_1 + t_2 \trianglelefteq q_0 \vee t_1 + t_2 \trianglelefteq q_1 \vee t_1 + t_2 \trianglelefteq q_3$ on obtient les solutions suivantes:

$$\begin{aligned}
\sigma_1 &= \{t_1 \mapsto q_3, t_2 \mapsto q_0\}\\
\sigma_2 &= \{t_1 \mapsto q_3, t_2 \mapsto q_3\}\\
\sigma_3 &= \{t_1 \mapsto q_1, t_2 \mapsto q_3\}\\
\sigma_4 &= \{t_1 \mapsto q_1, t_2 \mapsto q_1\}\\
\sigma_5 &= \{t_1 \mapsto q_0, t_2 \mapsto q_3\}\\
\sigma_6 &= \{t_1 \mapsto q_1, t_2 \mapsto q_0\}\\
\sigma_7 &= \{t_1 \mapsto q_0, t_2 \mapsto q_1\}
\end{aligned}$$

  Or, parmi ces 7 solutions, il est possible d'en éliminer 5 immédiatement puisqu'elles doivent également vérifier les diséquations précédentes: $t_1 \neq 0$ et $\forall h, k \in \mathcal{T}(\mathcal{F})\ t_1 \neq h + k$. Or, l'état q_1 ne reconnait que 0 et q_3 ne reconnait que 0 ou tout terme ayant un + au sommet. Ainsi, toutes les solutions associant t_1 à q_1 ou q_3 peuvent être écartées. Donc, seules σ_5 et σ_7 peuvent être envisagées. Le terme $u = t_2 + t_1$ résultant de la réécriture est donc tel que t_2 est reconnu soit par l'état q_1 soit par l'état q_3 de l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$. Si l'on ajoute en plus que t_2 est nécessairement irréductible par rapport à $\mathcal{R}$, il doit être également reconnu par l'automate $A_{IRR(\mathcal{R})}$ que nous pouvons également calculer:

```
[] result term:
    Description of nil states q|0.nil final states q|0.nil transitions
o->q|0.s(q|0)->q|0.nil End of Description
```

  Or, le seul terme reconnu par q_1 ou q_3 et par l'automate précédent est 0. D'où $t_2 = 0$, et $u = 0 + t_1$. Puisque t_1 est également en forme normale, la seule réécriture pouvant s'appliquer sur u est appliquée au sommet. De plus la règle appliquée est nécessairement (1). On a donc $u = t_2 + t_1 \underset{in}{\longrightarrow}_\mathcal{R} t_1 = v$. Or, par la propriété de sous-terme de $>$ on a nécessairement $t > v = t_1$.

  Comme dans notre premier exemple, nous venons de montrer que pour toute chaîne de réécriture innermost avec priorité $t \underset{in}{\longrightarrow}_\mathcal{R} t_1 \underset{in}{\longrightarrow}_\mathcal{R} t_2 \ldots$ issue de $t \in \mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$, il existe un entier $i \in \mathbb{N}$ tel que $t > t_i$. De la même façon, à partir de $t > t_i$, en appliquant l'hypothèse d'induction, on obtient $P(t_i)$, i.e. t_i termine et pour tout terme $t'_i \in IRR(\mathcal{R})$ tel que $t_i \underset{in}{\longrightarrow}_\mathcal{R}^+ t'_i$, on a $t_i > t'_i$. Ainsi, comme toutes les chaînes issues de t passent par un terme qui termine, on en déduit que t termine. D'autre part, de $t > t_i$ et $t_i > t'_i$, et de la transitivité de $>$ on déduit que $t > t'_i$. D'où finalement, on a $P(t)$.

L'automatisation de ce procédé est en cours d'étude.

25. il s'agit bien ici d'un état quelconque de l'automate et pas nécessairement d'un état final, puisque nous devons montrer la propriété P pour tous les termes et sous-termes de $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A))$

8.2 Vers un environnement de preuve de terminaison

Dans les chapitres précédents, nous avons détaillé quelques-unes des principales méthodes de preuve de terminaison offrant de bonnes propriétés pour l'automatisation. Dans cette section, nous abordons le problème de la conception d'un environnement logiciel dédié à la preuve de terminaison. Tout d'abord, nous passons en revue les outils de base d'un tel environnement. Ensuite, nous étudions différentes combinaisons possibles de ces outils et nous tentons de dégager leurs intérêts relatifs en pratique.

8.2.1 Les phases et outils essentiels

En plus d'une phase de suppression des conditions des systèmes conditionnels, une preuve de terminaison peut compter jusqu'à trois phases, dont une optionelle, qui sont: une phase de préparation de la preuve par production des contraintes, une phase de simplification des contraintes (optionnelle), et enfin une phase de résolution des contraintes. Nous donnons maintenant les principaux outils que nous proposons dans notre architecture pour chaque phase.

Conditionnel → non-conditionnel Pour prouver la décroissance d'un système conditionnel, nous avons vu qu'une méthode consistait à se ramener au cas d'un système non-conditionnel dans lequel sont ajoutées des règles $l \to s_i$ et $l \to t_i$, $i = 1 \dots n$, pour toute règle conditionnelle $l \to r$ if $s_1 \downarrow t_1, \dots, s_n \downarrow t_n$. Il existe une autre méthode, bien connue, de transformation de systèmes conditionnels en systèmes non-conditionnels, utilisée notamment dans [Art97], dans laquelle des règles de réécriture décrivant la sémantique d'un opérateur if sont ajoutées et les règles conditionnelles sont exprimées grâce à cet opérateur sous la forme de règles non-conditionnelles. Ces deux méthodes seront appliquées par un outil que nous nommerons DECOND.

Préparation de la preuve La préparation de la preuve passe par la génération des contraintes d'ordre à satisfaire. Nous avons vu qu'il existait à ce jour deux méthodes de génération des contraintes: la méthode standard où, pour chaque règle $l \to r$, nous ajoutons une inégalité $l > r$, et la méthode des paires de dépendance déjà étudiée dans la section 4.3. Nous nommerons respectivement STD et DP les outils de génération des contraintes standard et avec paires de dépendance.

Simplification des contraintes Quelle que soit la méthode de génération des contraintes utilisée, il est possible de les simplifier en les normalisant par un schéma de programme récursif (ou RPS, voir section 4.3). Cependant, on a vu que ce type de simplification était essentiellement utile pour la méthodes des paires de dépendance. L'outil associé sera nommé RPS.

Résolution des contraintes Enfin, la résolution des contraintes d'ordre peut être effectuée par un outil automatique tel LPO (voir section 7.1), ou un outil semi-automatique tel GPO (voir chapitre 5), ou encore POLO [Gie95b] pour les interprétations polynomiales. L'intérêt d'un outil automatique tel LPO réside principalement dans le fait qu'il s'agit d'une procédure de décision pour l'existence d'un ordre *lpo* satisfiant les contraintes initiales. En d'autres termes, si la procédure échoue, cela signifie qu'il est possible de *définitivement* écarter *lpo* pour tenter de satisfaire ces contraintes. Ceci n'est pas le cas avec des outils tels POLO et GPO, basés sur une grande part d'interprétation, où un échec n'est pas rédhibitoire.

8.2.2 Les combinaisons efficaces

Dans cette section nous donnons quelques combinaisons des outils de base donnant de bons résultats en pratique. Ces résultats totalement empiriques sont le fruit de nos propres observations ainsi que des observations de T. Arts et J. Giesl dans le cadre de leurs expérimentations sur les paires de dépendances.

Ce que nous appellons ici “combinaison efficace” est une combinaison d'outils offrant un bon compromis entre

– le spectre d'application, i.e. le nombre de systèmes pouvant être traités par la combinaison, et

– la qualité d'automatisation, i.e. le temps de calcul s'il s'agit d'une combinaison d'outils automatiques, et la qualité de l'aide fournie par le calcul dans le cas d'une méthode semi-automatique.

Par exemple, nous considérons que les combinaisons STD + LPO et STD + GPO sont toutes deux efficaces, au même titre. En effet, même s'il est clair que GPO est plus puissant que LPO, LPO est une procédure de décision aux temps de calculs acceptables, alors que GPO est semi-automatique. Ceci permet entre autres de tester LPO en premier, avant de tenter de résoudre un ensemble de contraintes à l'aide de GPO. Voici quelques combinaisons efficaces des outils de base. Nous ne faisons pas apparaître les outils DECOND qui ne posent aucun problème d'efficacité puisqu'il s'agit uniquement de transformations syntaxiques.

STD + LPO Cette combinaison est la première à essayer sur tout système de réécriture. Un énorme avantage est que chacune des deux réponses possibles est informative: si la réponse est positive, alors le système termine, si la réponse est négative, alors *il n'existe aucune précédence* pour orienter ce système avec l'ordre *lpo*.

Avantages: procédure de décision. Efficace sur les systèmes simples. Génération explicite de l'ordre.

Inconvénient: spectre relativement étroit de l'ordre, systèmes simplifiants uniquement.

DP + POLO Nous avons vu que l'utilisation des paires de dépendance ramène la preuve de terminaison à la satisfiabilité d'un ensemble d'inégalités par un ordre bien fondé non nécéssairement monotone. Or, la génération de tels ordres est particulièrement aisée avec les interprétations polynomiales.

Avantages: contrôle possible de la preuve de terminaison en donnant une partie des interprétations polynomiales à la main. Bénéficie du fait que les interprétations polynomiales sont un outil relativement intuitif. Traite les systèmes non simplifiants.

Inconvénients: POLO n'est pas une procédure de décision. Les calculs sont limités dans le temps par une borne supérieure arbitraire choisie par l'utilisateur. Pas de génération explicite de l'ordre.

STD + GPO La résolution des contraintes d'ordre GPO permettent de guider l'utilisateur lors de la preuve, en extrayant la partie difficile de la preuve sur laquelle l'utilisateur doit concentrer ses efforts. Certaines parties peuvent être résolues automatiquement par des fonctions de terminaison de type précédence. Possibilité de tenter des preuves automatiques avec les instances décidables de GPO.

Avantage: bon compromis automatisation/contrôle et interaction. Génération explicite de l'ordre.

Inconvénient: GPO n'est pas une procédure de décision.

DP + RPS + LPO RPS permet de simplifier les inégalités et de les affaiblir suffisament pour que la forte monotonie du lpo ne soit plus un handicap. De plus, on profite de la décidabilité de LPO. Le seul problème reste l'énorme complexité de l'outil RPS (section 4.3)

Avantages: Traite les systèmes non simplifiants. Procédure de décision, si l'on restreint à un nombre fini de RPS à étudier.

Inconvénients: Explosion des calculs de RPS. Peu de contrôle sur la preuve. En particulier, il est difficile d'aider manuellement la résolution puisque le facteur d'explosion, le choix du RPS n'est pas toujours intuitif. Pas de génération explicite de l'ordre.

DP + GPO Identique au cas STD + GPO, mais l'affaiblissement préliminaire de la condition de terminaison par l'utilisation de DP est à l'avantage de GPO qui peut produire des instances non nécessairement monotone.

Avantage: bon compromis automatisation/contrôle et intéraction.

Inconvénient: GPO semi-automatique. Pas de génération explicite de l'ordre.

DP + RPS + GPO Cette approche combine une phase d'affaiblissement des inégalités par DP, une phase de simplification par RPS et l'extraction des parties difficiles restantes par résolution de contraintes GPO. Possibilité d'équilibrer le contrôle entre la construction du RPS et les interprétations dans GPO.

Avantage: Sans doute l'outil le plus puissant (mais pas nécessairement le plus efficace).

Inconvénient: exponentialité de RPS et indécidabilité de GPO. Pas de génération explicite de l'ordre.

Toutes ces combinaisons de méthodes visent à établir la terminaison d'un système de réécriture. Une fois les résultats établis, nous avons vu qu'il était possible de les exploiter afin de montrer la terminaison d'un système modulaire les contenant. Afin de simplifier les preuves en réutilisant les résultats de terminaison existants, un environnement de preuve de terminaison doit pouvoir vérifier les conditions classiques de modularité de la terminaison générale [Mid90, Ohl94, Der95, Gra96], ainsi que le critère de terminaison de la relation séquentielle proposé section 6.6.5.

Chapitre 9

Conclusion

Dans cette thèse, nous nous sommes principalement intéressés à la preuve de terminaison des systèmes de réécriture, mais également à ses implications dans le domaine plus général de la vérification de programmes et de processus. Afin d'automatiser les preuves, nous avons proposé deux approches complémentaires basées respectivement sur la résolution de contraintes d'ordre et la résolution de contraintes ensemblistes à l'aide d'automates d'arbres.

Dans la première approche, nous avons défini un algorithme de preuve de terminaison des systèmes de réécriture résolvant des contraintes d'ordre *gpo* sur une structure de graphe OCS qui est une structure de donnée partagée représentant les règles de réécriture. Les noeuds du graphe représentent les opérateurs et les arcs représentent respectivement la relation de réécriture, la relation de sous-terme ou la relation d'ordre *gpo* entre les termes. Le dernier type d'arc est étiqueté par une formule, conditionnant la validité de l'arc, nommée obligation de preuve ($\mathcal{O}$-preuve). Nous avons donné un ensemble correct et complet de règles d'inférence assurant la construction des $\mathcal{O}$-preuves de façon polynomiale en temps et en espace. A partir d'un graphe OCS initial dépourvu d'arcs d'ordre, ces règles construisent tous les arcs d'ordre utiles, jusqu'à obtenir une forme normale. L'un des intérêts principaux de cet algorithme est qu'il évite la construction d'un grand nombre d'arcs trivialement insatisfaisables, allégeant par là même le graphe et la complexité du calcul. Une méthode de preuve de satisfaisabilité des $\mathcal{O}$-preuves a également été définie. Elle utilise une représentation des $\mathcal{O}$-preuves sous forme de DAG, un processus d'instanciation partielle des fonctions de terminaison, ainsi qu'un mécanisme de résolution basé sur la recherche de chemins minimisant les hypothèses sur les $\mathcal{O}$-preuves. Dans le cas des instances de *gpo*, pour lesquelles la satisfaisabilité de toutes les fonctions de terminaison peut être testée automatiquement, la preuve de terminaison peut être intégralement automatisée. Pour les autres cas, l'intérêt de la méthode est de focaliser l'effort de l'utilisateur sur le coeur de la preuve en démontrant de façon automatique les parties les plus simples et en extrayant les parties les plus difficiles. Des extensions de la méthode au cas particulier de l'extension lexicographique et de l'extension multi-ensemble de l'ordre *gpo* ont également été données. Dans le cas multi-ensemble, l'algorithme repose sur une nouvelle définition de l'extension multi-ensemble d'un préordre étendant la définition traditionnelle sur les ordres et plus opérationnelle que les définitions connues.

La deuxième approche établit un critère permettant de prouver la terminaison d'un système composé de modules à partir des résultats de terminaison de chacun des modules. Ce critère est basé sur le calcul des ensembles de $\mathcal{R}$-descendants, et de $\mathcal{R}$-formes normales. Ces ensembles ne sont, en général, pas des ensembles réguliers et ne sont donc pas représentables par des automates d'arbres classiques. Au lieu d'en donner une représentation exacte avec des outils plus complexes, le parti que nous avons pris ici a été d'en calculer des sur-ensembles réguliers suffisamment précis pour répondre aux besoins du critère. L'essentiel du travail a porté sur la définition d'un algorithme calculant un automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ qui reconnaît un sur-ensemble des $\mathcal{R}$-descendants de $\mathcal{L}(A)$. Cet automate d'approximation est paramètré par une fonction précisant le degré d'approximation voulu. Dans cette thèse, nous avons plus particulièrement développé le cas de la fonction d'approximation ancêtre qui, bien que simple, donne des résultats intéressants en pratique. Grâce au pliage des appels récursifs pouvant intervenir dans le calcul de $\mathcal{T}_\mathcal{R} \uparrow (A)$, l'approximation ancêtre donne, en général, de meilleurs résultats que les approximations existantes, basées sur l'approximation de la relation de réécriture. En dehors du domaine des langages réguliers, il existe une approche que nous n'avons pas étudié ici, basée sur les grammaires d'arbres synchronisées [LR97] (Tree Tuple Synchronised Grammars, ou TTSGs) qui permet de réaliser un calcul exact de l'ensemble des descendants. Cependant, ce calcul s'applique à des classes plus restreintes de systèmes de réécriture: les systèmes constructeurs linéaires confluents et la faisabilité de ce calcul n'a pas été encore réellement démontrée.

Outre les applications de ce calcul d'approximation au critère de preuve de terminaison modulaire, nous avons proposé d'autres applications, toutes dans le domaine de la vérification de programme, dont une déjà connue: la preuve de complétude suffisante, et d'autres plus originales: le test d'atteignabilité, l'approximation de co-domaine et la preuve de non-terminaison.

Grâce aux deux prototypes implantés au cours de cette thèse, nous avons démontré la faisabilité et l'intérêt pratique de l'approche contrainte pour l'automatisation des preuves de terminaison. Nous avons montré la place que pourraient prendre ces prototypes, en tant qu'outils de base, dans un démonstrateur automatique dédié à la terminaison. De plus, nous avons répertorié les autres outils potentiellement automatisables vus dans cette thèse et dégagé l'intérêt relatif de leurs combinaisons en fonction du but à atteindre: terminaison seule ou ordre décidable, procédure automatique ou avec contrôle.

D'autre part, en alliant les deux formes de contraintes, nous avons ébauché une approche prometteuse pour un problème qui nous tient en haleine depuis le début de cette thèse: la prise en compte de stratégies quelconques et d'ensembles de termes initiaux dans les preuves de terminaison. Nous utilisons, pour cela, un mécanisme de preuve par induction, sur un domaine (un ensemble de termes clos par réécriture reconnu par l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$) muni d'un ordre d'induction construit au fur et à mesure de la preuve par un mécanisme de résolution de contraintes d'ordre. Cette approche a le mérite de montrer la terminaison de problèmes situés en dehors du spectre d'application des méthodes classiques et offre ainsi de nouvelles perspectives. En outre, d'autres pistes de recherche sont possibles dans le prolongement de ce qui a été présenté dans cette thèse.

Optimisation de la recherche des chemins minimaux

Nous avons proposé une méthode semi-automatique de preuve de satisfaisabilité des $\mathcal{O}$-preuves. Dans notre prototype, toujours dans un but d'évaluation, nous avons réalisé une implantation directe des mécanismes d'instanciation partielle gauche (avec une précédence) et de recherche des chemins minimaux tels qu'ils sont décrits dans la section 5.5. Voici un exemple de ce que l'on peut obtenir dans le cas du système définissant la factorielle d'un entier (voir section 5.4 page 72)

```
Path: [129, 126, 72, 113, 110, 49, 50, 117, 114, 52, 25, 101, 98, 30, 2, 89, 86, 7]
Precedence: [fact > p, fact > (*), (*) > (+), (+) > s]
Oproofs: T(fact(s(x)))>T(fact(p(s(x)))) T(*(s(x), y))>T(*(x, y)) T(+(x, s(y)))>T(+(x, y))

Proof done in 0.39 seconds, cpu time
```

où le chemin est donné sous la forme d'une liste d'entiers correspondant aux étiquettes des nœuds du graphe de $\mathcal{O}$-preuve[26], d'une précédence compatible et d'une liste de $\mathcal{O}$-preuves restant à démontrer. Cette première implantation montre également que, dans le cas de systèmes de taille plus importante, un algorithme naïf de recherche des chemins minimaux est insuffisant. En effet, l'efficacité du parcours complet d'un DAG de $\mathcal{O}$-preuve comptant un nombre important de chemins, est insuffisante. Cependant, là encore, il doit être possible de limiter l'explosion combinatoire en utilisant des techniques de propagation de contraintes inévitables ainsi que des contraintes relatives à la structure des graphes utilisés. Par exemple, à partir d'un chemin p qui *n'est pas* minimal il est possible de désigner et de supprimer tous les chemins "supérieurs" i.e. incluant au moins toutes les $\mathcal{O}$-preuves présentes sur le chemin p. Or, la structure des $\mathcal{O}$-preuves utilisées dans notre implantation est telle que, après une instanciation partielle gauche, par exemple, les $\mathcal{O}$-preuves restantes sont situées dans les branches droites de la disjonction. En appliquant ce type de caractérisation structurelle à partir d'un chemin non-minimal il doit être possible d'éliminer, sans les parcourir, un certain nombre de chemins inintéressants parce que non-minimaux.

La déduction de RPS

La déduction d'un schéma de programme récursif (RPS), même s'il s'agit d'un processus décidable, s'avère être parfois inutilisable en pratique même pour de petits systèmes. En effet, le nombre de RPS à considérer augmente de façon exponentielle avec le nombre de symboles de fonction. L'algorithme de déduction des RPS implanté dans le prototype de T. Arts est une recherche brutale. Une piste de recherche possible est d'optimiser la construction des RPS en utilisant un mécanisme proche de celui des SOCS. Une possibilité est également de transformer les règles de déduction pour que celles-ci, au lieu de générer simplement des $\mathcal{O}$-preuves génèrent des couples ($\mathcal{O}$-preuve, RPS) tels que la $\mathcal{O}$-preuve soit effectivement une $\mathcal{O}$-preuve de l'arc *si* l'on applique le RPS à la règle.

Application de $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ aux graphes de dépendance

Une autre piste de recherche liée aux paires de dépendance concerne la simplification des graphes de dépendance. En effet, pour déterminer s'il existe un arc entre deux paires de dépendance $\langle s, t\rangle$ et $\langle u, v\rangle$, i.e. s'il existe une substitution σ telle que $t\sigma \rightarrow^*_{\mathcal{R}} u\sigma$, la méthode communément utilisée par T. Arts et J. Giesl est l'approximation basée sur les fonction REN et CAP définies dans la section 4.3. Il existe des cas simples prenant cette approximation en défaut et un arc est ajouté alors qu'il n'a pas lieu d'être. Lorsque cela est le cas T. Arts et J. Giesl proposent d'utiliser une méthode basée sur la surréduction. Une autre approximation possible consiste à construire un automate A reconnaissant toutes les instances closes de t, de calculer $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ et de vérifier s'il existe une substitution σ telle que $u\sigma$ est reconnu par $\mathcal{T}_{\mathcal{R}} \uparrow (A)$. On peut remarquer, en outre, que lorsqu'il s'agit de montrer la terminaison innermost à l'aide des paires de dépendance, il est possible de restreindre le langage reconnu par $\mathcal{T}_{\mathcal{R}} \uparrow (A)$ avec l'hypothèse supplémentaire que les sous-termes des termes atteignables doivent être en forme normale, et donc être également reconnus par $A_{IRR(\mathcal{R})}$. Cette approximation semble plus fine que l'approximation basée sur REN et CAP, mais il reste à étudier son efficacité en pratique, par rapport à la technique à base de la surréduction de T. Arts et J. Giesl.

26. généré en même temps et donné figure 7.10

Valeur de l'approximation ancêtre dans les cas exacts

Dans le calcul de $\mathcal{T}_\mathcal{R} \uparrow (A)$, si les membres droits de toutes les règles du système $\mathcal{R}$ sont des variables ou sont de la forme $f(x_1, \ldots, x_n)$ où $x_1, \ldots, x_n$ sont des variables, alors l'approximation n'intervient pas et $\mathcal{T}_\mathcal{R} \uparrow (A)$ reconnaît exactement $\mathcal{R}^*(\mathcal{L}(A))$. En effet, toutes les transitions à ajouter sont déjà normalisée: elles sont soit de la forme $q \to q'$ soit de la forme $f(q_1, \ldots, q_n) \to q$ où $q_1, \ldots, q_n$ sont des états.

Cependant, dans le cas des systèmes clos, des systèmes monadiques et semi-monadiques, si l'on calcule un automate approximation à l'aide de l'approximation ancêtre, il est parfois nécessaire de normaliser des transitions de la forme $t \to q$ où t est un terme clos de hauteur quelconque figurant dans le membre droit d'une règle $l \to r \in \mathcal{R}$. Cependant, par définition de la fonction d'approximation ancêtre, il n'y a approximation de la normalisation de ces transitions que si le terme t ou l'un de ses descendants par $\mathcal{R} \setminus \{l \to r\}$ est réécrit à nouveau par $l \to r$. Dans ce cas précis, il semble possible que l'approximation ancêtre soit exacte mais cela reste à montrer. Si tel est le cas, alors pour les systèmes clos, monadiques et semi-monadiques, nous aurons également $\mathcal{L}(\mathcal{T}_\mathcal{R} \uparrow (A)) = \mathcal{R}^*(\mathcal{L}(A)$ pour l'approximation ancêtre.

Dans, le cas des systèmes décroissants le problème est différent puisque le membre droit d'un règle peut comporter des variables n'apparaissant pas dans le membre gauche. Ceci n'est pas standard en réécriture et n'est pas réalisable par l'approximation telle qu'elle est définie dans la section 6.4.

Calculs exacts de $\mathcal{R}^*(E)$ et classes $(\mathcal{R}, E)$ régulières

Une autre voie de recherche directement liée à la précédente est la recherche de conditions sur $\mathcal{R}$ et E pour que l'ensemble des descendants $\mathcal{R}^*(E)$ soit un ensemble régulier. En analysant les sur-ensembles de $\mathcal{R}^*(\mathcal{L}(A))$ et $\mathcal{R}^!(\mathcal{L}(A))$ obtenus par l'approximation ancêtre, il est apparu que le calcul effectué était parfois exact, et ce pour des systèmes en dehors de la classe des systèmes "décroissants". Avec F. Jacquemard, nous avons tenté d'isoler une classe de systèmes soumis à des conditions orthogonales à la décroissance et de concevoir un algorithme permettant de calculer de façon exacte l'ensemble des descendants pour cette classe de systèmes. Or, il semble que si l'on n'impose aucune condition sur l'ensemble E lui-même, les conditions imposées sur $\mathcal{R}$ deviennent trop restrictives et donc impraticables pour beaucoup de systèmes de réécriture récursifs. Une piste de recherche possible est la recherche de conditions, *à la fois* sur le système de réécriture (tout en conservant une expressivité suffisante) et sur le langage de départ, i.e. $\mathcal{L}(A)$, assurant la reconnaissabilité de l'ensemble des descendants.

Raffinement de l'approximation

Deux extensions de l'algorithme de construction de $\mathcal{T}_\mathcal{R} \uparrow (A)$ sont, a priori, envisageables: extension au cas des systèmes non-linéaires à gauche et extension au cas de la réécriture innermost.

Tout d'abord, voyons sur un exemple les problèmes particuliers posés par les règles non-linéaires à gauche. Soit $\mathcal{F} = \{f : 2, g : 1, a : 0, b : 0\}$ une signature, $\mathcal{R} = \{f(x, g(x)) \to g(x)\}$ un système de réécriture et $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, tel que $\mathcal{Q} = \{q_0, q_1, q_2, q_3\}$, $\mathcal{Q}_f = \{q_0\}$ et $\Delta = \{$

$$\begin{aligned} f(q_1, q_2) &\to q_0 \\ a &\to q_1 \\ g(q_3) &\to q_2 \\ a &\to q_3 \\ b &\to q_3\} \end{aligned}$$

l'automate d'arbres reconnaissant exactement le langage: $\{f(a,g(a)), f(a,g(b))\}$. Si l'on applique l'algorithme de filtrage, défini dans la section 6.2.3, sur le problème initial $f(x,g(x)) \trianglelefteq q_0$, on obtient:

$f(x,g(x)) \trianglelefteq q_0$	règle **Configuration**
$f(x,g(x)) \trianglelefteq f(q_1,q_2)$	règle **Decompose**
$x \trianglelefteq q_1 \wedge g(x) \trianglelefteq q_2$	règle **Configuration**
$x \trianglelefteq q_1 \wedge g(x) \trianglelefteq g(q_3)$	règle **Decompose**
$x \trianglelefteq q_1 \wedge x \trianglelefteq q_3$	

Dans l'algorithme de filtrage, tel que nous l'avons défini, ceci ne donne pas une substitution satisfaisante puisqu'il est impossible d'associer à x à la fois q_1 et q_3. Ainsi, si l'on applique rigoureusement l'algorithme de construction de $\mathcal{T}_\mathcal{R} \uparrow (A)$ de la section 6.4, on ne peut ajouter aucune transition à l'automate A, d'où $\mathcal{T}_\mathcal{R} \uparrow (A) = A$ et le terme $g(a)$ n'est pas reconnu par $\mathcal{T}_\mathcal{R} \uparrow (A)$ alors qu'il appartient à $\mathcal{R}^*(\mathcal{L}(A))$. Cependant, même s'il n'existe pas directement de substitution satisfaisante, il faut bien remarquer que la réponse de l'algorithme de filtrage décrit exactement l'ensemble des instances de x satisfaisant le problème de filtrage initial. En effet, les instances t de x que nous cherchons sont telles que $t \rightarrow_\Delta^* q_1$ *et* $t \rightarrow_\Delta^* q_3$. Si l'on souhaite construire $\mathcal{T}_\mathcal{R} \uparrow (A)$ en tenant compte de ceci, une première approximation possible consiste à construire autant de membres droits qu'il existe de substitutions satisfaisantes: ajouter ici les transitions $g(q_1) \rightarrow q_0$ et $g(q_3) \rightarrow q_0$. Ainsi on couvre bien tous les termes possibles de $\mathcal{R}^*(\mathcal{L}(A))$. Cette approximation ne demande que très peu de modifications de l'algorithme général mais est, cependant, très grossière puisqu'elle ajoute à la fois $g(a)$ et $g(b)$ au sur-ensemble régulier des descendants.

Pour construire une approximation plus précise, il suffit de remarquer que les langages $\mathcal{L}(A,q_1)$ et $\mathcal{L}(A,q_3)$ sont réguliers. Il est donc possible de construire un automate $B = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$ tel que $\mathcal{Q}'_f = \{q'_0\}$, $\mathcal{Q} \cap \mathcal{Q}' = \emptyset$ et $\mathcal{L}(B,q'_0) = \mathcal{L}(A,q_1) \cap \mathcal{L}(A,q_3)$. Ensuite si l'on ajoute $\mathcal{Q}'$ à $\mathcal{Q}$ et Δ' à Δ, on peut construire une substitution solution σ qui est $\sigma = \{x \mapsto q'_0\}$. Dans notre exemple, l'automate représentant l'intersection de $\mathcal{L}(A,q_1)$ et $\mathcal{L}(A,q_3)$ est $B = \langle \mathcal{F}, \{q'_0\}, \{q'_0\}, \{a \rightarrow q'_0\} \rangle$, et la substitution solution est $\sigma = \{x \mapsto q'_0\}$. On sait que par $\mathcal{R}$: $f(q'_0, g(q'_0)) \rightarrow g(q'_0)$. En conséquence, la transition à ajouter à Δ est $g(q'_0) \rightarrow q_0$. Ainsi l'ensemble reconnu par l'automate $\mathcal{T}_\mathcal{R} \uparrow (A)$ obtenu est bien $\mathcal{L}(A) = \{f(a,g(a)), f(a,g(b)), g(a)\}$.

Cependant, dans le cas général, il reste à définir une méthode pour combiner cette technique avec une approximation qui assurera la finitude du calcul de $\mathcal{T}_\mathcal{R} \uparrow (A)$, dans tous les cas. Une des principales difficultés réside dans le fait que l'ensemble des états ajoutés n'est pas réduit aux états nécessaires pour la normalisation des transitions, mais comprend également tous les états de l'automate B reconnaissant les termes de l'intersection.

La deuxième extension envisageable de ces techniques d'approximation concerne le calcul des descendants pour la réécriture innermost, i.e. pour tout système de réécriture $\mathcal{R}$ et tout ensemble régulier de termes initiaux $E \subseteq \mathcal{T}(\mathcal{F})$, calculer un sur-ensemble régulier de $\{t \mid \exists s \in E \text{ t.q. } s \underset{in}{\longrightarrow}_\mathcal{R}^* t\}$. Le principe est très proche du cas précédent: pour chaque réécriture, nous savons que les sous-termes du membre gauche sont en forme normale. Ainsi, tout problème de filtrage de la forme $l \trianglelefteq q$ où l est un membre gauche de règle, pourra être ramené à $l \trianglelefteq q_1 \vee \ldots \vee q_n$ où $q_1, \ldots, q_n$ sont les états finaux d'un automate B reconnaissant le langage régulier $\mathcal{L}(A,q) \cap L_{inner}$ où $L_{inner} = \{f(t_1, \ldots, t_n) \mid \forall f \in \mathcal{F}^n \text{ et } \forall t_1 \ldots t_n \in IRR(\mathcal{R})\}$. Comme dans le cas précédent, en ajoutant les états et transitions de l'automate B, un des problèmes principaux reste d'assurer la finitude du calcul de $\mathcal{T}_\mathcal{R} \uparrow (A)$.

Bibliographie

[AG97a] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In M. Bidoit and M. Dauchet, editors, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, volume 1214 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1997.

[AG97b] T. Arts and J. Giesl. Proving innermost termination automatically. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer-Verlag, 1997.

[AG97c] T. Arts and J. Giesl. Termination of rewriting using dependency pairs. Technical Report IBN 97/46, Technische Hochschule Darmstadt, 1997.

[AG98] T. Arts and J. Giesl. Modularity of termination using dependency pairs. In *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 226–240. Springer-Verlag, 1998.

[Art96] T Arts. Termination by absence of infinite chains of dependency pairs. In H. Kirchner, editor, *Proceedings 21st International Colloquium on Trees in Algebra and Programming, Linköping (Sweden)*, volume 1059 of *Lecture Notes in Computer Science*, pages 196–210. Springer-Verlag, 1996.

[Art97] T. Arts. *Automatically proving termination and innermost normalisation of term rewriting systems*. PhD thesis, Universiteit Utrecht, Utrecht, 1997.

[Aza98] Ramzi Azaiez. Vérification automatique de programmes réactifs. Rapport de stage de fin d'études, INRIA Lorraine, 1998.

[BCL86] A. Ben Cherifa and P. Lescanne. An actual implementation of a procedure that mechanically proves termination of rewriting systems based on inequalities beteween polynomial interpretations. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, pages 42–51. Springer-Verlag, 1986.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[BKR92] Adel Bouhoula, E. Kounalis, and M. Rusinowitch. Spike: An automatic theorem prover. In *Proceedings of the 1st International Conference on Logic Programming* *and Automated Reasoning, St. Petersburg (Russia)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 460–462. Springer-Verlag, July 1992.

[BL90] F. Bellegarde and P. Lescanne. Termination by completion. *Applicable Algebra in Engineering, Communication and Computation*, 1(2):79–96, 1990.

[Bou94] Adel Bouhoula. *Preuves Automatiques par Récurrence dans les Théories Conditionnelles.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, March 1994.

[Bra69] Walter S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.

[BT91] B. Bogaert and S. Tison. Automata with equality test. Technical Report IT 207, Laboratoire d'Informatique Fondamentale de Lille, February 1991.

[BT92] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *Proceedings of the 9th Symposium on Theoretical Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 161–172, 1992.

[CDE+98] M. Clavel, F. Durán, S. Eker, P. Lincoln, and J. Meseguer. An Introduction to Maude (Beta Version). Technical report, SRI, March 1998.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and Tommasi. Tree automata techniques and applications. Preliminary Version, `http://l3ux02.univ-lille3.fr/tata/`, 1997.

[CDGV91] J.L. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvölgyi. Bottom-up tree pushdown automata and rewrite systems. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 287–298. Springer-Verlag, 1991.

[Com] H. Comon. About proofs by consistency. Invited talk at RTA'98.

[Com86] H. Comon. Sufficient completeness, term rewriting system and anti-unification. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer-Verlag, 1986.

[Com90] H. Comon. Solving inequations in term algebras. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 62–69, June 1990.

[CR87] H. Comon and Jean-Luc Rémy. How to characterize the language of ground normal forms. Technical Report 676, INRIA-Lorraine, 1987.

[Dau89] M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120. Springer-Verlag, April 1989.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–116, 1987. Special issue on Rewriting Techniques and Applications.

[Der95] N. Dershowitz. Hierarchical termination. In N. Dershowitz and N. Lindenstrauss, editors, *Proceedings 4th International Workshop on Conditional Term Rewriting Systems, Jerusalem (Israel)*, volume 968 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 1995.

[DH95] N. Dershowitz and C. Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, May 1995.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DOS88] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings 9th International Conference on Automated Deduction, Argonne (Ill., USA)*, volume 310 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1988.

[Dro89] K. Drosten. Termersetzungssysteme. *Informatik–Fachberichte*, 210, 1989.

[DT90] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 242–248, June 1990.

[FD85] R. Forgaard and D. Detlefs. An incremental algorithm for proving termination of term rewriting systems. In J.-P. Jouannaud, editor, *Proceedings 1st Conference on Rewriting Techniques and Applications, Dijon (France)*, pages 255–270. Springer-Verlag, 1985.

[Fer95] M.C.F. Ferreira. *Termination of Rewriting.* PhD thesis, Universiteit Utretch, Utretch, 1995.

[FG84] R. Forgaard and John V. Guttag. Reve: A term rewriting system generator with failure-resistant Knuth-Bendix. Technical report, MIT-LCS, 1984.

[Gau84] M.-C. Gaudel. A first introduction to pluss. Meteor report, LRI, 1984.

[GB85] J. H. Gallier and R. V. Book. Reductions in tree replacement systems. *Theoretical Computer Science*, 37:123–150, 1985.

[Gen97a] Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms (extended version). Technical Report RR-3325, Institut National de Recherche en Informatique et Automatique, 1997.

[Gen97b] Thomas Genet. Proving termination of sequential reduction relation using tree automata. Technical Report 97-R-091, Centre de Recherche en Informatique de Nancy, 1997.

[Gen98] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 1998.

[Geu89] O. Geupel. Overlap closure and termination of term rewriting systems. Technical Report MIP-8922, Universität Passau (Germany), 1989.

[GG97a] T. Genet and I. Gnaedig. Termination proofs using gpo ordering constraints. In M. Dauchet, editor, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, volume 1214 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1997.

[GG97b] Thomas Genet and Isabelle Gnaedig. Termination proofs using gpo ordering constraints (extended version). Technical Report R-3087, Institut National de Recherche en Informatique et Automatique, 1997.

[GHG+93] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[Gie95a] J. Giesl. Generating polynomial orderings for termination proofs. In Jieh Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Gie95b] Jurgen Giesl. Polo – a system for termination proofs using polynomial orderings. Technical Report IBN 95/24, Technische Hochschule Darmstadt, 1995.

[Gna92] I. Gnaedig. Termination of order-sorted rewriting. In Hélène Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*, volume 632 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, September 1992.

[Gra94] B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computation*, 5:131–158, 1994.

[Gra96] Bernhard Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems*. PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, January 1996.

[GS84] F. Gécseg and M. Steinby. *Tree automata*. Akadémiai Kiadó, Budapest, Hungary, 1984.

[GT95] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 333, Ravenswood Ave., Menlo Park, CA 94025, August 1988.

[HL78] G. Huet and D. S. Lankford. On the uniform halting problem for term rewriting systems. Technical Report 283, Laboria, France, 1978.

[HL86] T. Hardin and A. Laville. Proof of termination of the rewriting system SUBST on CCL. *Theoretical Computer Science*, 46:305–312, 1986.

[Hoo97] C. Hoot. *Termination of non-simple rewrite systems.* PhD thesis, University of Illinois, Urbana-Champaign, 1997.

[Jac96a] F. Jacquemard. *Automates d'arbres et réécriture de termes.* PhD thesis, Université Paris-Sud, 1996.

[Jac96b] F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, pages 362–376. Springer-Verlag, 1996.

[JL82] J.-P. Jouannaud and P. Lescanne. On multiset orderings. *Information Processing Letters*, 10:57–63, 1982.

[JSA94] P. Johann and Rolf Socher-Ambrosius. Solving simplification ordering constraints. In J.-P. Jouannaud, editor, *Proceedings of the 1st International Conference on Constraints in Computational Logics, Munich (Germany)*, volume 845 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 1994.

[JW86] J.-P. Jouannaud and B. Waldmann. Reductive conditional term rewriting systems. In M. Wirsing, editor, *Proceedings of 3rd IFIP Conf. on Formal description of Programming Concepts*, Ebberup (Denmark), 1986. Elsevier Science Publishers B. V. (North-Holland).

[Kap84] S. Kaplan. Fair conditional term rewriting systems: Unification, termination, and confluence. Technical report, University of Paris Sud, Orsay (France), Orsay, 1984.

[Kap87] S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination and confluence. *Journal of Symbolic Computation*, 4(3):295–334, December 1987.

[KB70] Donald E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KK90] M. Kurihara and I. Kaji. Modular term rewriting systems and the termination. *Information Processing Letters*, 34:1–4, February 1990.

[KL80] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Technical report, University of Illinois, 1980.

[KL82] S. Kamin and J.-J. Lévy. Attempts for generalizing the recursive path ordering. *Inria, Rocquencourt*, 1982.

[Kli93] P. Klint. The ASF+SDF Meta-environment User's Guide. Technical report, CWI, 1993.

[KN85] M. S. Krishnamoorthy and P. Narendran. A note on recursive path ordering. In *Theoretical Computer Science*, volume 40, pages 323–328, 1985.

[KNZ87] D. Kapur, P. Narendran, and H. Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–415, 1987.

[KO91] M. Kurihara and A. Ohuchi. Modular term rewriting systems with shared constructors. *Journal of Information Processing of Japan*, 14(3):357–358, 1991.

[Kou85] E. Kounalis. Completeness in data type specifications. In B. Buchberger, editor, *Proceedings EUROCAL Conference, Linz (Austria)*, volume 204 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, 1985.

[Kru60] J. B. Kruskal. Well-quasi ordering, the tree theorem and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95:210–225, 1960.

[Lan75] D. S. Lankford. Canonical algebraic simplifications. Technical report, Louisiana Tech. University, 1975.

[Lan79] D. S. Lankford. On proving term rewriting systems are noetherian. Technical report, Louisiana Tech. University, Mathematics Dept., Ruston LA, 1979.

[Les83] P. Lescanne. Computer experiments with the REVE term rewriting systems generator. In *Proceedings of 10th ACM Symposium on Principles of Programming Languages*, pages 99–108. ACM, 1983.

[Les84] P. Lescanne. Uniform termination of term rewriting systems. Recursive decomposition ordering with status. In B. Courcelle, editor, *Proceedings 9th Colloquium on Trees in Algebra and Programming*, pages 182–194. Cambridge University Press, 1984.

[LM78] D. S. Lankford and D. R. Musser. A finite termination criterion. Unpublished draft, Information Sciences Institute, University of Southern California, Marina–del–Rey, CA, 1978.

[LR97] S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars. In M. Dauchet, editor, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, volume 1214 of *Lecture Notes in Computer Science*, pages 429–440. Springer-Verlag, 1997.

[LS95] Christopher Lynch and Polina Strogova. Sour graphs for efficient completion. Technical Report 95-R-343, CRIN, 1995.

[MG94] A. Middeldorp and B. Gramlich. Simple termination is difficult. *Applicable Algebra in Engineering, Communication and Computation*, 1994. To appear.

[Mid89] A. Middeldorp. A sufficient condition for the termination of the direct sum of term rewriting systems. In *Proceedings 4th IEEE Symposium on Logic in Computer Science, Pacific Grove*, pages 396–401, 1989.

[Mid90] A. Middeldorp. *Modular Properties of Term Rewriting Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[MN70] Z. Manna and S. Ness. On the termination of markov algorithms. In *Third Hawai International Conference on System Sciences*, pages 789–792, 1970.

[Mon81] J. Mongy. Transformations de noyaux reconnaissables d'arbres. forêt rateg. 1981. Thèse d'Université de Lille 1.

[Mor98] Pierre Etienne Moreau. A choice-point library for backtrack programming. In *JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic*, 1998.

[MS+93] M. Meier, J. Schimpf, et al. ECLiPSe User Manual. Technical report ECRC-93-6, ECRC, Munich (Germany), 1993.

[New42] M. H. A. Newman. On theories with a combinatorial definition of equivalence. In *Annals of Math*, volume 43, pages 223–243, 1942.

[Nie93] R. Nieuwenhuis. Simple lpo constraint solving methods. *Information Processing Letters*, 47(2), 1993.

[NR92] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In D. Kapur, editor, *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs (N.Y., USA)*, volume 607 of *Lecture Notes in Computer Science*, pages 477–491. Springer-Verlag, 1992.

[NW83] T. Nipkow and G. Weikum. A decidability result about sufficient completeness of axiomatically specified abstract data types. In *6th GI Conference*, volume 145 of *Lecture Notes in Computer Science*, pages 257–268. Springer-Verlag, 1983.

[OCL96] Peter Ölveszky Csaba and Olav Lysne. Order-sorted termination: The unsorted way. In Michael Hanus and Mario Hanus, Rodríguez-Artalejo, editors, *Proceedings 5th International Conference on Algebraic and Logic Programming, Aachen (Germany)*, volume 1139 of *Lecture Notes in Computer Science*, pages 92–106. Springer-Verlag, 1996.

[Ohl94] E. Ohlebusch. *Modular Properties of Composable Term Rewriting Systems.* PhD thesis, Universität Bielefeld, Bielefeld, 1994.

[Pla93] D. Plaisted. Polynomial time termination and constraint satisfaction tests. In C. Kirchner, editor, *Proceedings 5th Conference on Rewriting Techniques and Applications, Montreal (Canada)*, volume 690 of *Lecture Notes in Computer Science*, pages 405–420, Montreal (Québec, Canada), June 1993. Springer-Verlag.

[Roc97] C. Rocques. *Modularité dans les spécifications algébriques.* PhD thesis, Université d'Orsay, Orsay, 1997.

[Rus87] M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26(2):65–70, 1987.

[Sal88] K. Salomaa. Deterministic Tree Pushdown Automata and Monadic Tree Rewriting Systems. *Journal of Computer and System Sciences*, 37:367–394, 1988.

[SK93] J. Steinbach and U. Kühler. Check your ordering – termination proofs and open problems. Technical report, Universität Kaiserslautern, 1993.

[Ste89] J. Steinbach. Extensions and comparisons of simplification orderings. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 434–448. Springer-Verlag, April 1989.

[Ste94] J. Steinbach. Generating polynomial orderings. *Information Processing Letters*, 49:85–93, 1994.

[Ste95] J. Steinbach. Simplification orderings: history of results. *Fundamenta Informaticae*, 24:47–87, 1995.

[Toy86] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, May 1986.

[Wal94] C. Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1), 1994.

[Zan94] H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.

[Zan95] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

Index