Etude des propriétés des systèmes de réécriture conditionnelle : mise en oeuvre de deux algorithmes de test de confluence sur les termes clos

Wadoud Bousdira

Centre de Recherche en Informatique de Nancy Institut National Polytechnique de Lorraine

Etude des propriétés des systèmes de réécriture conditionnelle. Mise en œuvre de deux algorithmes de test de confluence sur les termes clos.

Thèse de Docteur de l'Institut National Polytechnique de Lorraine

Option Informatique présentée par

Wadoud **BOUSDIRA**

soutenue le 19 octobre 1990
devant la commission d'examen

Président : Jean-Pierre Finance
Rapporteurs : Jean-Pierre Finance
Fernando Orejas

Examinateurs : Harald Ganzinger
Laurent Kott
Pierre Lescanne
Jean-Luc Rémy

قال رسول الله صلى الله عليه وسلم:

" من سلك طريقا يلتمس فيه علما، سهّل الله له طريقا إلى الجنّة وإنّ الملائكة لتضع أجنحتها لطالب العلم رضا بما صنع، وإن العالم ليستغفر له من في السموات ومن في الأرض حتّى الحيتان في الماء، وفضل العالم على العابد كفضل القمر على سائر الكواكب، وإن العلماء ورثة الأنبياء، وإن الأنبياء لم يورّثوا دينارا ولا درهما، إنما ورّثوا العلم فمن أخذه أخذ بحظ وافر. "

حديث شريف

Un seul régime : la démocratie

La démocratie seule est salutaire. Elle ne signifie pas l'anarchie. Elle ne signifie pas un pouvoir faible. Elle signifie : le gouvernement du peuple par le peuple. Elle signifie un Etat hiérarchisé. Une bonne Constitution doit donner la parole au peuple. Elle doit permettre la libre discussion. Cette libre discussion, loin de nuire à la discipline nationale, permettra de révéler des cadres valables et enrichira les institutions de l'Etat. Un Etat *confisqué* est un Etat mort-né.

L'indépendance confisquée
Ferhat Abbas
1962-1978

Je remercie très sincèrement tous les membres du jury d'avoir accepté de porter un avis sur cette thèse:

Jean-Pierre Finance pour avoir manifesté de l'intérêt pour mon travail et m'avoir fait l'honneur de présider ce jury. Je le remercie vivement d'avoir accepté la rude tâche d'être mon rapporteur.

Jean-Luc Rémy pour avoir dirigé mon travail de recherche. Sans sa compétence et ses conseils, ce travail n'aurait pas vu le jour. Qu'il trouve ici l'expression de ma profonde gratitude.

Fernando Orejas pour avoir bien voulu s'inéresser à ce travail et auquel je suis très reconnaisante d'avoir accepté de rédiger un rapport, malgré l'obstacle présenté par la différence de langue.

Pierre Lescanne pour m'avoir souvent fait bénéficier de ses remarques constructives, et m'avoir maintes fois prodigué des encouragements chaleureux.

Harald Ganzinger qui a accepté de porter un avis sur cette thèse et qui n'a pas hésité à se déplacer depuis Dortmund.

Laurent Kott pour avoir siégé dans ce jury. Je le remercie vivement de cette preuve d'intérêt.

Je remercie aussi Hélène Kirchner pour avoir lu attentivement une première version de cette thèse et m'avoir prodigué ses critiques et ses conseils amicaux, ainsi que Jean-Pierre Jouannaud pour s'être intéressé à mes recherches.

Je n'oublierai pas de remercier vivement tous les membres de l'équipe EURECA, plus particulièrement Eric Domenjoud dont j'ai souvent apprécié l'aide efficace, ainsi que les membres du CRIN, de l'INRIA-LORRAINE et de l'INPL qui ont su créer une agréable ambiance de travail.

Enfin que tous mes proches, parents et amis trouvent ici le témoignage de ma plus sincère gratitude pour le soutien et l'appui amical qu'ils ont su me prodiguer à tout moment.

Résumé

Définir un type abstrait algébrique par des axiomes conditionnels offre davantage de souplesse et de simplicité que dans un cadre classique, sans préconditions. Ceci nous a amenée à étudier la théorie conditionnelle de termes dans le but d'obtenir une méthode de décision fondée sur la réécriture de termes. Cependant, cette étude dans le contexte conditionnel soulève des problèmes délicats, principalement liés à la difficulté de formaliser la sémantique sous-jacente à la théorie conditionnelle et de définir des modèles.

L'objectif de cette thèse est d'étudier les propriétés des systèmes de réécriture conditionnelle. Notre contribution centrale est la mise en œuvre de deux tests de confluence sur les termes clos. Le premier test est établi pour des systèmes de réécriture *hiérarchiques*, systèmes construits par strates successives, la précondition d'une règle utilisant exclusivement des opérateurs appartenant aux strates inférieures. Le second test de confluence est établi pour des systèmes *décroissants*, dont les règles sont définies avec un principe de récurrence structurelle sur leurs composantes. Ces deux classes de systèmes ont été définies et utilisées dans des travaux ultérieurs par divers auteurs pour éviter d'engendrer des dérivations infinies de réécriture.

L'originalité du travail est l'utilisation d'un principe de raisonnement par cas, et la possibilité de réécrire une équation conditionnelle aussi bien dans sa partie conséquence que dans sa précondition. Ce processus de simplification s'est révélé souple et puissant et permet de traiter un certain nombre d'applications.

Nous abordons également dans notre étude un aspect inhérent à la théorie conditionnelle : le comportement d'une spécification conditionnelle par rapport aux booléens. Ce comportement est fidèle dès lors que les propriétés de consistance et de complétude par rapport à la sorte booléenne sont garanties, assurant de façon intuitive qu'il n'y a ni ajout de booléens, ni confusion entre les deux constantes de vérité *true* et *false*.

La propriété de complétude d'une spécification conditionnelle est également abordée. Cette propriété étant indécidable même dans la théorie classique, sans préconditions, nous proposons un algorithme fondé sur le raisonnement par cas pour tester si une spécification conditionnelle est complète par rapport à l'ensemble de ses constructeurs. Cet algorithme combine une analyse structurelle par le calcul d'un ensemble de motifs et une analyse par cas par le biais d'un ensemble de conditions utilisées dans la réécriture. Cette condition suffisante de test permet d'appréhender une sous-classe relativement étendue de définitions conditionnelles.

Table des matières

INTRODUCTION

Introduction

L'écriture et la construction de programmes se sont révélées être une activité difficile nécessitant des méthodes élaborées, des langages bien adaptés et des outils efficaces. Cette difficulté a amené l'utilisateur à créer un langage formel et rigoureux, permettant d'énoncer un problème dans un formalisme de très haut niveau, basé sur la théorie des types abstraits de données algébriques. Les spécifications algébriques sont alors des descriptions dans ce langage, et par leur rigueur, ressemblent aussi bien à des programmes qu'à des énoncés mathématiques. La spécification d'un type de données consiste en une algèbre explicitement définie. La syntaxe de la spécification correspond à la signature d'algèbres et sa sémantique est captée par l'algèbre initiale de la classe d'algèbres de la spécification. Les spécifications algébriques peuvent en outre être considérées comme une axiomatisation de la théorie du type de données qu'elles spécifient. Elles permettent par conséquent l'utilisation de prouveurs de théorèmes et donc l'automatisation de la validation de leurs propriétés formelles. Parmi les nombreux outils permettant de mener à bien cette validation, la réécriture est l'un des plus efficaces.

La réécriture est à l'origine une méthode de preuve en logique équationnelle qui a suscité depuis fort longtemps l'intérêt de nombreux chercheurs afin de résoudre le problème du mot dans l'algèbre universelle [64]. Il s'agit de décider si une égalité $s = t$ est valide dans une classe d'algèbres. La méthode classique de preuve est celle du remplacement d'égaux par des égaux qui consiste à déduire le théorème $s = t$ à partir des axiomes égalitaires. Cependant, cette méthode s'est avérée inefficace en raison des nombreux retours en arrière qu'elle suscite sur les choix d'égalités. Afin de lever cet indéterminisme, on lui a préféré la méthode de réécriture dont le principe est d'orienter les axiomes en règles de réécriture.

Dans la théorie inconditionnelle, dite classique, une règle de réécriture est une paire orientée de termes, notée $g \to d$. Réécrire un terme à l'aide d'une règle $g \to d$ consiste à remplacer, dans ce terme, une instance de g par l'instance correspondante de d. La preuve du théorème $s = t$ se ramène alors à vérifier que s et t se réécrivent en un même terme, à l'aide de l'ensemble des règles du système de réécriture.

Par ailleurs, un système de réécriture de termes doit vérifier deux propriétés essentielles, *la terminaison* appelée également *nœthérianité*, qui assure qu'un calcul termine toujours, et *la confluence* qui exprime que si le résultat de ce calcul existe, alors il est unique. Un système de réécriture confluent et nœthérien est dit canonique. Il traduit l'existence et l'unicité de tout calcul.

Cependant, il est rare qu'une spécification équationnelle transformée en système de réécriture donne directement lieu à un système canonique. Il est souvent nécessaire de recourir à la procédure de complétion de Knuth-Bendix. L'idée de base de cette procédure est de supprimer les ambiguïtés dans le calcul en forçant l'égalité de deux termes issus

de la simplification d'un même terme. Ces ambiguïtés sont plus connues sous le nom de *paires critiques*. La procédure de Knuth-Bendix définit un algorithme de semi-décision pour le problème de validité dans les théories équationnelles pour lesquelles il s'applique, et elle fournit une méthode de décision lorsque l'algorithme termine.

Les règles de réécriture peuvent être utilisées pour spécifier une partie des opérations d'un type abstrait, sous forme de définitions récursives. Elles ne sont cependant pas suffisantes dans certaines situations, notamment lorsque nous avons besoin d'exprimer des propriétés ou des définitions conditionnelles, ou encore si nous voulons définir des opérations partielles ou exprimer des propriétés restreintes à un sous-domaine. Il est alors utile de définir des axiomes conditionnels que l'on note $p \Rightarrow g = d$. La partie $g = d$ est la conséquence de l'équation et p est sa précondition.

Le cadre de cette thèse est la théorie de la réécriture conditionnelle de termes. Son objectif est l'étude des différents concepts mis en jeu pour établir la confluence des systèmes de réécriture conditionnelle dans l'algèbre initiale. Notre contribution centrale est la mise en œuvre de deux tests de confluence sur les termes clos. Le premier concerne les systèmes de réécriture hiérarchiques et le second est établi pour une classe de systèmes de réécriture conditionnelle plus souples, à savoir les systèmes décroissants.

Le plan de ce document s'articule comme suit : le premier chapitre introduit dans son premier paragraphe les bases théoriques préliminaires aux thèmes abordés dans cette thèse. Nous explicitons dans cette partie, la théorie des spécifications algébriques et dans la partie qui lui succède, la méthode de réécriture de termes. Un troisième paragraphe de ce chapitre définit la *complétude* des spécifications algébriques. Il est connu que cette propriété est indécidable. Nous rappellons les principaux résultats qui permettent d'établir des conditions suffisantes pour vérifier qu'une spécification algébrique, classique ou conditionnelle, est complète par rapport à l'ensemble de ses constructeurs. L'intérêt de cette propriété est crucial dès lors que l'on étudie la confluence d'un système de réécriture. Enfin, le quatrième paragraphe est consacré à l'étude des spécifications conditionnelles. Nous présentons dans une première partie le formalisme de la réécriture conditionnelle selon trois critères spécifiques. Il est en effet difficile d'aborder cette présentation dans un cadre général car elle dépend des objectifs visés concernant l'utilisation des systèmes conditionnels et de nombreux choix quant à la définition d'une règle de réécriture conditionnelle, sont alors possibles. Le premier critère de définition de la théorie conditionnelle caractérise la classe d'algèbres considérée, le second est lié à la terminaison de la réduction conditionnelle et le troisième spécifie le type de relation de réécriture conditionnelle que l'on choisit de définir. Nous exposons par ailleurs les problèmes que pose la généralisation du cadre classique, sans préconditions, au cadre conditionnel. Dans sa seconde partie, le paragraphe comporte une synthèse des divers travaux entrepris dans le domaine conditionnel. Ce chapitre a également pour but de fixer les notations et la terminologie utilisées dans cette thèse.

Le chapitre 2 est consacré à l'étude de la complétude des spécifications conditionnelles. La complétude d'une spécification est nécessaire dès lors qu'on s'intéresse à la propriété de confluence sur les termes clos. Elle garantit en particulier que toute instance close d'une précondition puisse être évaluée, et par conséquent, qu'un terme clos soit réécrit. L'importance de la complétude nous est apparue clairement lorsque nous avons voulu

valider nos résultats dans le cadre des spécifications hiérarchiques. En effet, la relation de réécriture hiérarchique que nous étudions dans le paragraphe suivant a un pouvoir d'expression plus restreint que la réduction récursive. Afin d'établir l'équivalence entre ces deux relations, il faut que la spécification en question soit complète. Après avoir, pendant un certain temps, supposé cette propriété acquise pour toute spécification étudiée, il nous a paru impératif d'en esquisser une étude et d'élaborer des conditions suffisantes de décision. Nous étudions dans un premier temps, la complétude dans un cadre conditionnel sans hypothèses de hiérarchie, et nous explicitons les conditions sous lesquelles cette propriété peut être décidée. Dans cette partie, nous formalisons un algorithme de test de la complétude en fonction d'une relation de réécriture appropriée. La version hiérarchique de cette relation nous permet d'appliquer, dans le chapitre suivant, une variante de cet algorithme au cadre des spécifications hiérarchiques pour obtenir une méthode de décision de la propriété de *complétude hiérarchique*.

Au chapitre 3, nous nous intéressons au caractère opérationnel de la réécriture conditionnelle. Réécrire un terme par la règle $p \Rightarrow g \rightarrow d$ en utilisant la substitution σ met en œuvre un processus qui comporte un chaînage en avant, qui consiste à simplifier le terme, et un chaînage en arrière pour justifier la précondition, i.e. vérifier que σp est vraie. Nous choisissons de considérer les préconditions des règles de réécriture comme des expressions de sorte booléenne évaluables elles aussi, par des règles de réécriture. Il est alors essentiel d'utiliser deux systèmes de réécriture différents pour éviter que tout calcul comporte une dérivation cyclique. L'idée consiste à définir une spécification conditionnelle de manière structurée, par niveaux imbriqués de hiérarchie, et à évaluer la précondition dans un sous-système afin d'assurer la terminaison d'une réduction quelconque. Ce principe nous a amenée à définir une relation hiérarchique. Cependant, de par la restriction hiérarchique qu'elle comporte dans sa définition, cette relation est appauvrie, et pour que la congruence qu'elle induit soit équivalente à la congruence induite par la réduction récursive, la spécification conditionnelle doit vérifier la propriété de complétude suffisante. La relation de réécriture hiérarchique nécessite d'être définie sur les termes clos et cela essentiellement pour deux raisons ; d'une part, pour évaluer toute l'expression σp, il faut que σp soit sans variables ; d'autre part, il n'est pas raisonnable de définir la propriété de complétude suffisante sur les termes avec variables. Afin d'élargir le domaine d'application de la réécriture hiérarchique aux termes avec variables, nous définissons par la suite un processus de réécriture contextuelle. Les conditions sont alors considérées comme des contextes et la notion de terme est généralisée à celle de *terme contextuel* qui définit un terme accompagné d'un contexte. Si une règle $p \Rightarrow g \rightarrow d$ est appliquée à un terme contextuel, plutôt que d'évaluer l'expression σp, celle-ci est ajoutée à l'ancien contexte. Sous certaines conditions, cette approche permet d'effectuer de la déduction sur les termes sans variables en comparant des *ensembles complets de formes normales contextuelles*. La complétude de ces ensembles est garantie par l'hypothèse de *bonne couverture* du système considéré. Nous étudions le processus de réécriture contextuelle dans un cadre hiérarchique et nous établissons un résultat de confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique. Par ailleurs, puisque les préconditions de règles sont des expressions booléennes, nous sommes amenée à définir le comportement des spécifications conditionnelles par rapport aux booléens à l'aide de deux propriétés essentielles, *la complétude par rapport aux booléens*, qui garantit que le support booléen d'une spécification conditionnelle est isomorphe à l'ensemble $\{true,\ false\}$, et *la consistance par rapport aux booléens*, qui assure que $true$ et $false$ sont deux constantes distinctes et évite par là-même toute

inconsistance.

L'objectif du chapitre 4 est de formaliser un processus de réécriture contextuelle défini à partir du *raisonnement par cas.* Ce processus garantit que toute simplification d'un terme est complète, puisque toutes les branches de réécriture sont envisagées. De ce fait, l'hypothèse de bonne couverture n'est plus utile et la définition par des équations conditionnelles d'un symbole d'opération, est rendue plus souple. Cependant, la formulation du raisonnement par cas que nous avons choisie dans ce chapitre, comporte dans sa définition même, une branche de réécriture infinie, appelée *branche complémentaire*, et il devient nécessaire de contrôler la terminaison de cette branche en mémorisant un certain nombre d'informations afin d'interdire la réutilisation des règles susceptibles d'engendrer une dérivation infinie. Deux solutions dans cet objectif sont envisageables; la première consiste à s'interdire l'utilisation de la règle simplificatrice sur l'équation issue de la branche complémentaire, et la seconde consiste à limiter à un nombre déterminé le nombre de simplifications dans cette branche. Pour notre part, nous avons adopté la première solution dont nous justifierons le choix. Nous établissons un test de confluence sur les termes clos des systèmes *décroissants* fondé sur des critères de convergence des paires critiques aisés à vérifier. L'étude élaborée dans ce chapitre présente de nombreux avantages, notamment en ce qui concerne la souplesse du processus de simplification et la facilité de mise en œuvre du test par rapport au travail précédent. Nous établissons par ailleurs une comparaison des deux stratégies et nous justifions les nouveaux choix.

Le rôle prépondérant des booléens dans la théorie conditionnelle nous a incitée à consacrer le dernier chapitre à l'étude des différents mécanismes de preuve dans les contextes booléens. Nous nous intéressons essentiellement à deux types de preuve et nous étudions plus particulièrement leur aspect opérationnel. Ces preuves sont soit des preuves *d'insatisfiabilité* de la forme $p = false$, ou des preuves *de bonne couverture* de la forme $p_1 \vee \ldots \vee p_n = true$. Nous nous appliquons dans une première partie, à illustrer par des exemples, la nécessité d'établir une interaction entre l'aspect algébrique du processus de réécriture conditionnelle et un traitement plus sophistiqué des booléens par des preuves du premier ordre. Ces dernières sont indispensables pour prouver l'insatisfiabilité des contextes ou encore pour montrer que la réduction d'un terme contextuel est complète et fournit un ensemble complet de formes normales contextuelles. Il est également important de mettre l'accent sur la sensibilité de ces preuves à la forme des spécifications conditionnelles et notamment, aux différentes formes que peuvent prendre les préconditions de règles. La puissance du test de confluence dépend d'un facteur essentiel: le savoir-faire dans les preuves de contextes. Il nous apparaît envisageable d'utiliser, au cours du test de confluence et éventuellement, au cours du processus de complétion, un prouveur de théorèmes pour des preuves localisées. Ce prouveur de théorèmes peut par ailleurs, être utilisé, si nécessaire, pour d'autres usages. Ce dernier chapitre se veut essentiellement prospectif. Notre intention est en effet d'exposer les divers problèmes rencontrés lors du test de convergence des paires critiques et de présenter les principales solutions adoptées par certains auteurs, particulièrement d'un point de vue implantation.

Nous concluons en résumant le travail effectué au cours de cette thèse et en indiquant d'une façon générale d'une part, les orientations que peuvent prendre les suites de ce travail et d'autre part, les problèmes encore ouverts suggérés dans le document.

CHAPITRE 1

LES SPECIFICATIONS ALGEBRIQUES ET LES SYSTEMES DE REECRITURE.

Chapitre 1

Les spécifications algébriques et les systèmes de réécriture.

1 Introduction

Ce chapitre a pour objectif de présenter le cadre de travail de cette thèse. Le premier paragraphe est consacré à des rappels de l'algèbre universelle. Il s'intègre dans le cadre des spécifications algébriques et il présente les principes de la théorie équationnelle. Le second paragraphe définit le formalisme des méthodes de réécriture de termes. Les systèmes de réécriture sont utilisés comme une base pour les spécifications algébriques de types abstraits de données. Ils fournissent également un mécanisme pour les langages de programmation fonctionnelle et leur caractéristique de base est un appel par filtrage. Ils sont inclus dans les langages actuels tels que HOPE [14], ML [15, 77], MIRANDA [103], OBJ [33, 37]. Les systèmes de réécriture de termes sont également un outil performant pour résoudre le problème du mot dans une variété d'algèbres et une méthode de décision est obtenue par l'algorithme de complétion de Knuth-Bendix sous des conditions spécifiques [64]. Un système de réécriture permet aussi de vérifier que la spécification algébrique correspondante possède certaines propriétés qui garantissent sa correction. Parmi ces propriétés, la complétude suffisante est essentielle à vérifier. Elle assure en effet que tout symbole de fonction est complètement défini par rapport aux constructeurs de la spécification. Toutefois, elle est indécidable, et les résultats connus à l'heure actuelle ont pour objectif d'établir des conditions suffisantes pour montrer qu'une spécification algébrique est suffisamment complète en utilisant son système de réécriture correspondant. Ces résultats sont pour la plupart établis dans le cadre classique, sans préconditions. Le troisième paragraphe définit la complétude suffisante d'une spécification et constitue par ailleurs un rappel des résultats connus. Ces résultats seront par la suite dans le chapitre 2, adaptés aux spécifications conditionnelles. Dans le paragraphe 4, nous nous plaçons finalement dans le cadre d'étude de cette thèse, à savoir, le cadre conditionnel et nous présentons les spécifications conditionnelles de termes. Définir un type abstrait algébrique par des axiomes conditionnels offre davantage de souplesse et de simplicité que dans un cadre classique, sans préconditions. La réécriture conditionnelle a pour objectifs de simplifier la programmation logique et d'optimiser la preuve de théorèmes. Elle permet de plus la validation des spécifications algébriques et constitue de ce fait un domaine d'étude important. Nous présentons dans une première partie de ce paragraphe la théorie de la réécriture conditionnelle sous trois aspects essentiels qui interviennent dans la définition d'une spécification conditionnelle.

Le premier critère concerne la classe d'algèbres que l'on vise à appréhender. Le second critère est un critère de terminaison qui garantit que toute réduction conditionnelle se termine et le troisième critère caractérise le type de définition que l'on adopte pour la relation de réécriture conditionnelle elle-même. Nous illustrons par ailleurs par des exemples les problèmes qui se posent lors de la généralisation des résultats de décidabilité dans la théorie classique au cadre conditionnel. Dans la seconde partie du paragraphe, nous procédons à une synthèse des divers travaux effectués dans le domaine conditionnel.

2 Spécifications algébriques et théorie équationnelle

2.1 Rappels d'algèbre universelle

Cette partie présente les principaux éléments de l'algèbre universelle sur lesquels se base la réécriture de termes. Nous nous sommes autorisée de nombreux emprunts aux travaux de Rémy et d'Hélène Kirchner [94, 63].
Par convention, nous avons choisi de terminer toute preuve par le symbole □ et tout exemple par le symbole ◇.

Signatures et F-algèbres

La théorie des algèbres universelles a pour objet l'étude des propriétés des algèbres qui sont invariantes par isomorphismes. Elle constitue de ce fait, un cadre formel d'étude pour les types abstraits de données. Il est nécessaire de considérer pour cette étude des algèbres hétérogènes, c'est-à-dire des algèbres à plusieurs sortes. De même, chaque opération est affectée d'un profil et est définie comme une application à 0 ou plusieurs arguments de diverses sortes.

Définition 1.1 Signature
Une signature $\Sigma = (S,\ F)$ est la donnée d'un ensemble S de sortes et d'un alphabet F profilé par S. Autrement dit, il existe une fonction a de F dans S^* qui, à tout symbole f de F, associe $a(f) = s_1 \ldots s_n\,; s$. $a(f)$ est le profil de f, s est son codomaine et n son arité. Un opérateur d'arité 0 est une constante.
$F_{\omega;s}$ désigne la signature composée des opérateurs de profil $\omega\,; s$ avec ω dans S^*.

Définition 1.2 F-algèbre
Une $(S,\ F)$-algèbre ou algèbre de type F, $\mathcal{A} = (A,\ F_{\mathcal{A}})$ est la donnée d'une famille $A = (A_s),\ s \in S$ d'ensembles non vides et d'une famille $F_{\mathcal{A}} = (f_{\mathcal{A}})_{f \in F}$ d'opérations dans A, indicée par F.
A est appelé le support de $\mathcal{A}$. L'application $f \rightarrow f_{\mathcal{A}}$ de F dans A^{A^*} est appelée interprétation des opérations dans $\mathcal{A}$.

Morphismes, algèbre initiale, algèbres libres

Un morphisme est une application qui permet de comparer deux F-algèbres entre elles. Plus formellement,

Définition 1.3 Morphisme
Soient $\mathcal{A} = (A,\ F_{\mathcal{A}})$ et $\mathcal{B} = (B,\ F_{\mathcal{B}})$ deux F-algèbres, une application ψ entre les supports $A = (A_s)_{s \in S}$ et $B = (B_s)_{s \in S}$ de $\mathcal{A}$ et $\mathcal{B}$ est un F-morphisme (ou morphisme de F-algèbres) ssi elle vérifie la propriété de commutativité :

$$\psi(f_{\mathcal{A}}(x_1, \ldots,\ x_n)) = f_{\mathcal{B}}(\psi(x_1, \ldots,\ \psi(x_n))$$

pour tout symbole f dans $F_{\omega;s'}$ avec $\omega = s_1 \ldots s_n$, pour tous x_1 de sorte $s_1, \ldots,\ x_n$ de sorte s_n.

Dans une variété d'algèbres universelles, une F-algèbre particulière est intéressante à étudier, la F-algèbre initiale. Cette F-algèbre est définie de la façon suivante :

Définition 1.4 F-algèbre initiale

Une F-algèbre $\mathcal{A} = (A,\ F_{\mathcal{A}})$ est dite initiale (ou minimale) si pour toute autre F-algèbre $\mathcal{B} = (B,\ F_{\mathcal{B}})$, il existe un morphisme unique noté $\psi_{\mathcal{A}}$ de A dans B.

Si l'algèbre initiale existe, elle est unique à un isomorphisme près. Cette algèbre est constituée de la F-algèbre $T(F)$ des termes clos ou, vu d'une autre façon, des arbres étiquetés par les symboles de F. On a alors le résultat suivant:

- un type abstrait algébrique est un type d'algèbre hétérogène;
- un objet du type est un élément de l'algèbre initiale;

Dans l'algèbre initiale, on considère comme différents deux termes dont on ne peut prouver l'égalité.
$T(F)$ est l'algèbre des termes clos. L'unique morphisme $\psi_{\mathcal{A}}$ interprète chaque terme clos comme un élément de $\mathcal{A}$.

Définition 1.5 Algèbre finiment engendrée

Une F-algèbre $\mathcal{A} = (A,\ F_{\mathcal{A}})$ est une algèbre finiment engendrée si le morphisme $\psi_{\mathcal{A}}$ de $T(F)$ dans $\mathcal{A}$ est surjectif.
Autrement dit, les algèbres finiment engendrées sont les images homomorphes de l'algèbre des termes. Si t est un terme clos sur $T(F)$, on note $t_{\mathcal{A}}$ son image $\psi_{\mathcal{A}}(t)$ dans $\mathcal{A}$. Ainsi,

$$A_s = \{t_{\mathcal{A}} \ /\ t \in T(F)_s\} \text{ pour } s \in S$$

Remarque: $T(F)$ n'est définie que si pour toute sorte s de S, il existe au moins un terme de sorte s. Nous ferons systématiquement cette hypothèse par la suite.

Définition 1.6 Ensemble des termes clos

La signature $\Sigma = (S,\ F)$ étant donnée, l'ensemble des termes clos, noté

$$T(F) = \{T(F)_s,\ s \in S\}$$

est défini de la façon suivante, pour tout symbole d'opération f dans F,

- *soit $f :\rightarrow s$ alors f est dans $T(F)_s$*
- *soit $f : s_1 \times \ldots \times s_n \rightarrow s$ et pour tout j dans $[1 \ldots n]$, t_j est dans $T(F)_{s_j}$, alors $f(t_1, \ldots,\ t_n)$ est dans $T(F)_s$.*

Le problème de l'existence de l'algèbre initiale est un cas particulier de celui de l'algèbre libre engendrée par une famille $X = (X_s)_{s \in S}$ d'ensembles disjoints de variables (il s'agit du cas où $X_s = \emptyset$ pour toute sorte s dans S). On peut construire une F-algèbre $T(F,\ X)$ libre sur un ensemble X de variables telle que, pour toute algèbre $\mathcal{A}$ et toute application $\nu : X \rightarrow \mathcal{A}$, il existe un morphisme unique $h_{\mathcal{A},\nu}$ de $T(F,\ X)$ dans A prolongeant ν.
Soit $X = (X_s),\ s \in S$, un ensemble de variables tel que X_s soit dénombrable, les éléments de la F-algèbre libre engendrée par X sont appelés termes.
Cette définition étant peu parlante, il est devenu classique de donner aux termes une représentation sous forme d'arbres étiquetés. Autrement dit, il s'agit de considérer un terme t comme une application d'un certain domaine d'arbres dans un alphabet, compatible avec le profil des symboles.

Définition 1.7 *Soit N_+ l'ensemble des entiers strictement positifs, N_+^* le monoïde libre engendré, ϵ le mot vide et . désignant l'opération de concaténation.*
On considère une application t de N_+^* dans $F \cup X$ et $dom(t)$ son domaine de définition, alors t est un arbre sur $F \cup X$ ssi

1. *ϵ est un élément de $dom(t)$*
2. *$dom(t)$ est clos par préfixe, i.e. $u \in dom(t)$ si $u.u' \in dom(t)$*
3. *pour tout u dans $dom(t)$, $u.i \in dom(t)$ ssi i est compris entre 1 et l'arité de $t(u)$.*

Le domaine d'un arbre t est également appelé ensemble des *occurrences* de t. Les éléments de $dom(t)$ sont les *nœuds* de l'arbre, le nœud ϵ est la *racine* (ou *sommet*) et $u.i$ est le $i^{\text{ème}}$ fils du nœud u.

Exemple 1.1 *Soit $F = \{f,\ h,\ a\}$ avec $ar(f) = 2$, $ar(h) = 1$ et $ar(a) = 0$ et $X = \{x,\ y\}$. L'application t, définie sur $\{\epsilon,\ 1,\ 2,\ 21\}$ par:*

$$t(\epsilon) = f,\ t(1) = a,\ t(2) = h \text{ et } t(21) = x$$

correspond au terme bien formé $f(a,\ h(x))$. $\diamond$

Définition 1.8 Occurrences disjointes
Deux occurrences sont disjointes ssi il n'existe pas de u" dans N_+^* tel que $u = u'.u$" ou $u' = u.u$".

Nous désignons aussi par $T(F,\ X)$ l'ensemble des arbres construits sur F et sur l'ensemble X de variables. Cela est justifié par le lemme suivant:

Lemme 1.1 *L'ensemble des arbres construits sur $F \cup X$ peut être muni d'une structure de F-algèbre, et c'est la F-algèbre engendrée par X.*

Définition 1.9 *Soit t un terme et u un élément de $dom(t)$; le sous-terme de t à l'occurrence u, noté $t_{/u}$ est le terme t' défini par $t'(u') = t(u.u')$ pour tout u' appartenant à N_+^*.*

Définition 1.10 *Soient t et t' deux termes et u une occurrence de t. L'opération de remplacement dans t du sous-terme $t_{/u}$ par t' consiste à définir un nouveau terme t" noté $t[u \leftarrow t']$ tel que*

- *$dom(t") = dom(t) - dom(t_{/u}) + u.dom(t')$*
- *$t"(u') = t(u')$ si $u' \in dom(t) - dom(t_{/u})$ ou $t'(u")$ si $u' = u.u"$*

Définition 1.11 *L'ensemble $Var(t)$ des variables d'un terme t est défini inductivement par:*

- *si t est une constante, $Var(t) = \emptyset$*
- *si t est une variable x, $Var(t) = \{x\}$*
- *si $t = f(t_1, \ldots,\ t_n)$, $Var(t) = Var(t_1) \cup \ldots \cup Var(t_n)$.*

Lorsqu'un terme t contient au plus une occurrence de chaque variable, t est dit linéaire. De même, si une variable apparaît au plus une fois dans un terme, on dit qu'elle est linéaire.

Les endomorphismes de l'ensemble des termes sont les substitutions :

Définition 1.12 Substitution
Une substitution est une application σ de X dans $T(F, X)$. Le domaine de σ, noté $Dom(\sigma)$ est le sous-ensemble de X défini par

$$Dom(\sigma) = \{x \ / \ \sigma(x) \neq x\}$$

Le caractère libre de la F-algèbre permet d'affirmer qu'une telle application se prolonge de façon unique en un endomorphisme de $T(F, X)$. Il vérifie donc pour tout f de F d'arité n, pour tous $t_1, \ldots, t_n$ de $T(F, X)$:

$$\sigma f(t_1, \ldots, t_n) = f(\sigma t_1, \ldots, \sigma t_n)$$

L'ensemble des substitutions sur $T(F, X)$ (resp sur $T(F)$) est noté $Subst(F, X)$ (resp $Subst(F)$).

Définition 1.13 *La composée de deux substitutions α et θ est la substitution notée $\alpha.\theta$ et définie pour toute variable x par $\alpha.\theta(x) = \alpha(\theta(x))$.*

Congruences :

Définition 1.14 F-congruence
Une F-congruence sur une F-algèbre $\mathcal{A}$ est une relation d'équivalence Ξ sur $\mathcal{A}$ vérifiant pour tout $f : s_1 \times \ldots \times s_n \rightarrow s$, pour tous $a_1, b_1, \ldots, a_n, b_n$ dans $\mathcal{A}$,

$$(a_1 \Xi b_1 \ \wedge \ldots \wedge \ a_n \Xi b_n) \Longrightarrow f_{\mathcal{A}}(a_1, \ldots, a_n) \Xi f_{\mathcal{A}}(b_1, \ldots, b_n)$$

Proposition 1.1 *Si Ξ est une congruence sur $\mathcal{A}$, l'ensemble $\mathcal{A}_\Xi$ appelé ensemble quotient, peut être muni d'une structure d'algèbre notée $\mathcal{A}_\Xi$ en posant pour $f : s_1 \times s_2 \times \ldots \times s_n \rightarrow s$ dans F, pour $a_1, \ldots, a_n$ dans $\mathcal{A}$,*

$$f^{\mathcal{A}_{F/\Xi}}(\bar{a}_1, \ldots, \bar{a}_n) = \overline{f^{\mathcal{A}}(a_1, \ldots, a_n)}$$

où pour tout i dans $[1 \ldots n]$, $\bar{a}_i$ est la classe d'équivalence de a_i dans l'algèbre $\mathcal{A}$.

Théorème 1.1 *Soient $\mathcal{A} = (A, F_{\mathcal{A}})$, $\mathcal{B} = (B, F_{\mathcal{B}})$ deux F-algèbres et $\psi : A \rightarrow B$ un F-morphisme, la relation Ξ_ψ définie par :*

$$a \Xi_\psi b \Longleftrightarrow \psi(a) = \psi(b) \ \forall a, \ b \in \mathcal{A}$$

est une F-congruence.
Réciproquement, si Ξ est une F-congruence sur $\mathcal{A}$, la projection

$$\pi_\Xi(a) = \bar{a}, \ \forall a \in \mathcal{A}$$

est un F-morphisme et la congruence associée à π_Ξ est la F-congruence Ξ.

Proposition 1.2 Quotients de l'algèbre initiale par des congruences
Soit Ξ une congruence sur $T(F)$; l'ensemble $T(F)/_\Xi$ des classes d'équivalence $[t]_\Xi$ de $T(F)$ peut être muni d'une structure de F-algèbre de la manière suivante :
Soient $f : s_1 \times \ldots \times s_n \rightarrow s$ dans F, t_1 dans $T(F)_{s_1}, \ldots, t_n$ dans $T(F)_{s_n}$. On pose

$$f_{T(F)/_\Xi}([t_1]_\Xi, \ldots, \ [t_n]_\Xi) = [f(t_1, \ldots, \ t_n)]_\Xi$$

2.2 Egalité des termes dans une algèbre

Soient $\mathcal{A}$ une F-algèbre et t et t' deux termes de $T(F,\ X)$; le problème est de savoir quand les termes t et t' sont égaux dans $\mathcal{A}$. Dans le cas où t et t' sont des termes clos, le problème se réduit à déterminer quand ces deux objets de même type et d'écriture différente sont en fait identiques.

Définition 1.15 *Soient $\mathcal{A}$ une F-algèbre et $t,\ t'$ deux termes de $T(F,\ X)$;*
$\mathcal{A}$ valide l'axiome $t = t'$ ssi

$$\forall \sigma : Var(t) \cup Var(t') \rightarrow \mathcal{A},\ \bar{\sigma}t = \bar{\sigma}t'$$

où $\bar{\sigma}$ est l'unique extension (à un isomorphisme près) de σ à l'ensemble $T(F,\ X)$.
On note $\mathcal{A} \models t = t'$.

$\mathcal{A}$ valide un ensemble E d'axiomes ssi $\mathcal{A}$ valide chaque axiome de E. $\mathcal{A}$ est alors une $(F,\ E)$-algèbre.

Théorème 1.2 Théorème de Birkhoff [5]
L'ensemble des congruences sur $T(F,\ X)$ compatible avec un ensemble d'axiomes E constitue un treillis complet dont l'élément minimal est la relation notée $=_E$ et appelée E-égalité ou théorie équationnelle engendrée par E.
De plus, soit $\mathcal{A}$ une $(F,\ E)$-algèbre, alors

$$\mathcal{A} \models t = t' \Longleftrightarrow t =_E t'$$

Le théorème de Birkhoff est un résultat fondamental puisqu'il ramène un problème sémantique, le problème de validité d'un axiome dans une variété équationnelle, à un problème purement syntaxique, l'égalité dans E. Cependant, il traduit le problème du mot dans une F-algèbre mais ne permet pas de le résoudre algorithmiquement. On peut même montrer qu'en général le problème de l'égalité de deux termes dans une F-algèbre est indécidable, c'est-à-dire qu'il n'existe aucun algorithme ayant pour données une F-algèbre $\mathcal{A}$ et un couple de termes $(t,\ t')$ et qui permette de répondre par oui ou par non à la question : l'égalité $t = t'$ est-elle valide dans la F-algèbre $\mathcal{A}$?

Spécifications et types abstraits.

Définition 1.16 Spécification équationnelle
Une spécification équationnelle $(\Sigma,\ E)$ est la donnée d'une signature $\Sigma = (S,\ F)$ et d'un ensemble fini E d'équations sur $T(F,\ X)$.

Une spécification équationnelle permet d'engendrer une congruence formée de toutes les égalités que l'on peut prouver à partir des axiomes du type par la méthode de remplacement d'égaux par des égaux de la façon suivante :

Définition 1.17 Théorie équationnelle
Soient $=_E$ la théorie équationnelle engendrée par un ensemble d'équations E et $t,\ t'$ deux termes de $T(F,\ X)$,
$t =_E t'$ ssi il existe une chaîne de termes $t_0,\ t_1, \ldots,\ t_n$, $0 \leq i < n$ avec $t = t_0$, $t' = t_n$ et telle que :

- *soit* $t_i = t_{i+1}$
- *soit il existe une équation* $g = d$ *dans* E, *une occurrence* u *de* t_i *et une substitution* $\sigma : X \rightarrow T(F, X)$ *telles que :*

$$t_{i/u} = \sigma g \text{ et } t_{i+1} = t_i[u \leftarrow \sigma d] \text{ ou } t_{i/u} = \sigma d \text{ et } t_{i+1} = t_i[u \leftarrow \sigma g].$$

Cette méthode permet de prouver l'égalité de deux termes. Néanmoins, elle présente un inconvénient majeur du fait qu'elle n'est pas déterministe. En effet, elle nécessite de nombreux retours en arrière sur les choix d'égalités et c'est pour lever cet indéterminisme que l'on a eu recours à la méthode de réécriture de termes.

3 Systèmes de réécriture de termes

Pour rendre décidable la congruence engendrée par un ensemble d'équations, celle-ci est transformée en un système de réécriture associant aux termes de chaque classe d'équivalence une forme normale unique. Un système de réécriture permet de prouver dans une théorie équationnelle, des égalités universellement quantifiées du type $t_1 = t_2$, où t_1, t_2 sont des termes avec variables.

Définition 1.18 Système de réécriture de termes

Un système de réécriture R *sur un ensemble de termes est un ensemble de règles de réécriture, chacune de la forme* $g \rightarrow d$, *où* g *et* d *sont des termes.* R *est en fait un ensemble* E *d'équations orientées en règles afin de rendre décidable la congruence engendrée par* E.
Les couples d'éléments de R *vérifient la condition* $Var(d) \subseteq Var(g)$.

A un système de réécriture R est associée une relation binaire $\rightarrow_R$ appelée relation de réécriture et définie de la façon suivante :

Définition 1.19 Relation de réécriture classique

Soient t *et* t' *deux termes de* $T(F, X)$, t *se réécrit en* t' *par* R *et on note* $t \rightarrow_R t'$ *s'il existe une règle* $g \rightarrow d$ *dans* R, *une occurrence* u *de* t *et une substitution* $\sigma : X \rightarrow T(F, X)$ *telles que*

$$t_{/u} = \sigma g \text{ et } t' = t[u \leftarrow \sigma d].$$

La relation de réduction $\rightarrow_R^*$ engendrée par R est la fermeture réflexive et transitive de la relation $\rightarrow_R$.

Définition 1.20 Forme normale

Un terme t *est dit irréductible ou en forme normale s'il n'existe pas de terme* t' *tel que* $t \rightarrow_R t'$.
De même, t' *est une forme normale de* t *si* $t \rightarrow_R^* t'$ *et* t' *est irréductible.*

Dans les années 70, Knuth et Bendix ont mis en évidence comment réécriture et complétion permettent de résoudre le problème du mot dans l'algèbre universelle [64]. Il s'agit d'obtenir, à partir d'un système E d'équations et par un processus de complétion, un système de réécriture R confluent et nœthérien. L'idée de base est de superposer deux membres

gauches de règles afin de créer une ambiguïté non triviale, encore appelée paire critique, de l'orienter en une règle de réécriture et de l'ajouter au système. Le processus de complétion est cyclique et lorsque l'algorithme n'engendre plus de nouvelles paires critiques et que toutes les équations sont orientées, il fournit un système final utilisé comme une procédure de décision pour l'égalité dans la variété correspondante. En effet, $t_1 =_E t_2$ si et seulement si t_1 et t_2 se réécrivent par le système R en une même forme normale $\bar{t}$. Cependant, pour que le calcul de cette forme normale se termine, il faut que la relation de réécriture $\rightarrow_R$ associée au système R soit nœthérienne, et d'autre part, afin de garantir l'unicité de $\bar{t}$, cette relation doit être confluente. Ces deux propriétés d'une relation de réécriture sont essentielles et elles se formulent de la façon suivante:

Définition 1.21 Terminaison

$\rightarrow_R$ termine ssi il n'existe pas de suite infinie de la forme

$$t_1 \rightarrow_R t_2 \rightarrow_R \ldots \rightarrow_R t_n \rightarrow_R \ldots$$

On dit encore que $\rightarrow_R$ (R) est nœthérienne (nœthérien).

Définition 1.22 Confluence, confluence sur les termes clos

$\rightarrow_R$ est confluente ssi pour tous termes t, t_1, t_2 dans $T(F, X)$, tels que $t \rightarrow_R^* t_1$ et $t \rightarrow_R^* t_2$, il existe t' dans $T(F, X)$ tel que

$$(**) \quad t_1 \rightarrow_R^* t' \text{ et } t_2 \rightarrow_R^* t'.$$

$\rightarrow_R$ est confluente sur les termes clos ssi $(**)$ est vérifiée pour tous termes clos t, t_1, t_2 dans $T(F)$ tels que $t \rightarrow_R^* t_1$ et $t \rightarrow_R^* t_2$.

La terminaison de $\rightarrow_R$ est indécidable. Cependant, cette propriété a fait l'objet de diverses études et des conditions suffisantes ont été établies pour tester la nœthérianité d'une relation de réécriture. Après quelques définitions utiles, nous formulons dans ce qui suit un théorème fondamental que nous utilisons à diverses reprises dans notre travail.

Définition 1.23 *Un ordre partiel $>$ sur un ensemble S est une relation binaire, transitive et irréflexive définie sur les éléments de S. L'ensemble S est dit partiellement ordonné.*
L'ordre $>$ est total si pour tous éléments distincts t_1, t_2 de S, soit $t_1 > t_2$ soit $t_2 > t_1$.

Définition 1.24 Ordre bien fondé

Un ordre partiel $>$ sur un ensemble S est bien fondé s'il n'existe pas de séquence infinie $t_1 > t_2 > \ldots > t_n > \ldots$ d'éléments de S.

La notion de bonne fondation de l'ordre permet de proposer la méthode suivante pour la preuve de la terminaison d'un système de réécriture:

Théorème 1.3 *$\rightarrow_R$ définie sur un ensemble de termes $T(F, X)$ termine ssi il existe un ordre $>$ bien fondé sur $T(F, X)$ tel que*

$$\forall s,\ t \in T(F, X),\ s \rightarrow_R t \Longrightarrow s > t$$

Définition 1.25 ***F*-compatibilité**

Un ordre partiel $>$ *sur l'ensemble des termes* $T(F,\ X)$ *est* F*-compatible s'il possède la propriété de remplacement définie par:*

$$s > t \Longrightarrow f(\ldots,\ s,\ldots) > f(\ldots,\ t,\ldots)$$

pour tous termes s *et* t *et pour tout symbole de fonction* f *de* F.

Nous présentons à présent le principal critère pour la preuve de terminaison :

Théorème 1.4 **[76, 18]**

$\rightarrow_R$ *termine sur un ensemble de termes* $T(F,\ X)$ *s'il existe un ordre bien fondé et* F*-compatible tel que*

$$\forall g \rightarrow d \in R,\ \sigma g > \sigma d.$$

pour toute substitution σ *dans* $T(F,\ X)$ *des variables de* g.

Un ordre bien fondé et F-compatible est encore appelé **ordre de réduction.**

Pour étudier la propriété de terminaison d'une relation de réécriture associée à un système de réécriture, un ordre appelé précédence, peut également être défini sur les symboles d'opérateurs de F. Cet ordre, total ou partiel, est étendu à l'ensemble $F \cup X$ en prenant pour convention qu'une variable n'est comparable avec aucune autre variable ni avec aucun symbole de F.

Une grande variété d'ordres a été définie dans le cadre d'étude de la terminaison, chacun d'eux étant caractérisé d'une part, par des propriétés particulières comme l'incrémentalité et la clôture par substitution et d'autre part, par leurs extensions aux ensembles et multi-ensembles ou par une extension lexicographique. Le lecteur peut se référer à [18] pour une synthèse approfondie de ces ordres.

La seconde propriété essentielle que doit vérifier une relation de réécriture est la propriété de confluence. Cette propriété est équivalente à la propriété de Church-Rosser. Soit par définition,

$$\longleftrightarrow_R^* = (\rightarrow_R^* \cup \leftarrow_R^*)^{-1}$$

La propriété de Church-Rosser est caractérisée par l'assertion suivante : pour tous termes t, t' dans $T(F,\ X)$ tels que $t \longleftrightarrow_R^* t'$, il existe $t"$ dans $T(F,\ X)$ tel que

$$t \rightarrow_R^* t" \text{ et } t' \rightarrow_R^* t".$$

$\rightarrow_R$ est canonique si elle est confluente et nœthérienne.

Dans le cadre de la réécriture classique, sans préconditions, la situation peut se résumer par le théorème de confluence de Knuth-Bendix. Dès que la relation de réécriture associée à un système de règles est nœthérienne, la confluence est équivalente à la propriété de confluence locale. La décidabilité de la canonicité d'une relation de réécriture est alors assurée puisque la confluence locale peut être vérifiée en examinant seulement un nombre fini de paires critiques et en montrant que celles-ci sont convergentes. La confluence locale se formule comme suit :

Définition 1.26 Confluence locale, confluence locale sur les termes clos
$\rightarrow_R$ est localement confluente ssi pour tous termes t, t_1, t_2 dans $T(F, X)$ tels que $t \rightarrow_R t_1$ et $t \rightarrow_R t_2$, il existe t' dans $T(F, X)$ tel que

$$(**) \quad t_1 \rightarrow_R^* t' \text{ et } t_2 \rightarrow_R^* t'.$$

$\rightarrow_R$ est localement confluente sur les termes clos ssi $(**)$ est vérifiée pour tous termes clos t, t_1, t_2 de $T(F)$ tels que $t \rightarrow_R t_1$ et $t \rightarrow_R t_2$.

Théorème 1.5 Théorème de Newman *[82]*
Si $\rightarrow_R$ termine, alors elle est confluente ssi elle est localement confluente.

Théorème 1.6 *[64]*
Une relation de réécriture $\rightarrow_R$ est localement confluente ssi toute paire critique non triviale est convergente.

On dira que la paire critique $t = t'$ est convergente s'il existe un terme $\bar{t}$ tel que $t \rightarrow_R^* \bar{t}$ et $t' \rightarrow_R^* \bar{t}$.

Nous nous sommes par ailleurs intéréssée à la propriété de complétude des spécifications conditionnelles. La complétude reste en effet une propriété essentielle pour établir la confluence sur les termes clos d'un système de réécriture. Le paragraphe qui suit constitue dans sa grande partie un rappel des résultats établis dans un cadre général, ces résultats restant valides dans le cadre des spécifications conditionnelles.

4 La complétude suffisante et les propriétés reliées

Complétude étant un terme souvent employé pour des notions différentes, nous préférons utiliser pour des raisons de clarté, le terme de complétude suffisante.
La complétude suffisante peut être en réalité formalisée à partir de deux concepts différents, exprimés l'un en fonction de la congruence algébrique et l'autre en fonction de la relation de réécriture conditionnelle. Nous établissons par la suite les conditions sous lesquelles l'équivalence de ces deux concepts peut être obtenue pour de telles spécifications.

Les définitions qui suivent utilisent les symboles $\oplus$ et $\ominus$ désignant respectivement l'union disjointe d'ensembles et la différence ensembliste.

Définition 1.27 Complétude suffisante
$SP = (S, \mathcal{C} \oplus \mathcal{D}, E)$ est suffisamment complète par rapport à $\mathcal{C}$ ssi

$$\forall t \in T(\mathcal{C} \oplus \mathcal{D}), \ \exists t_0 \in T(\mathcal{C}) \text{ tel que } t =_E t_0.$$

Définition 1.28 Complétude suffisante opérationnelle ou convertibilité
$SP = (S, \mathcal{C} \oplus \mathcal{D}, E)$ vérifie la complétude suffisante opérationnelle par rapport à $\mathcal{C}$ ssi

$$\forall t \in T(\mathcal{C} \oplus \mathcal{D}), \ \exists t_0 \in T(\mathcal{C}) \text{ tel que } t \rightarrow_R^* t_0.$$

$\mathcal{C}$ constitue l'ensemble des constructeurs de la spécification. Les termes construits uniquement à partir des constructeurs sont appelés *termes primitifs.*

Dans ce cas, nous dirons aussi que le système R est **convertible** (par rapport à $\mathcal{C}$) suivant en cela Kapur et Narendran et Zhang[59].

Complétude suffisante et convertibilité sont par conséquent deux notions distinctes. Tandis que la première est définie à partir de la congruence équationnelle, la seconde est liée à la relation de réécriture et elle est, de ce fait opérationnelle. L'exemple qui suit illustre la différence entre ces deux concepts.

Exemple 1.2 *Soit la spécification* $SP = (S,\ \mathcal{C} \oplus \mathcal{D},\ E)$ *avec:*
$S = \{elem\}$,
$\mathcal{C} = \{e :\rightarrow elem\},\ \mathcal{D} = \{a,\ b :\ \rightarrow elem\}$,
$E = \{b = a,\ b = e\}$;

Dans cette spécification, les trois constantes sont équivalentes modulo la congruence engendrée par E. En d'autres termes, $a =_E b =_E e$. Puisque e est dans $T(\mathcal{C})$, nous pouvons en conclure que la spécification est suffisamment complète par rapport à $\mathcal{C}$.
D'autre part, supposons que les équations de R soient orientées de gauche à droite et considérons la constante a. Dans le système obtenu, il n'existe aucun terme primitif $\bar{t}$ de $T(\mathcal{C})$ tel que $a \rightarrow_R^* \bar{t}$. Par conséquent, le système n'est pas convertible. $\diamond$

Définition 1.29 Clôture par $\rightarrow_R$
Un ensemble $\mathcal{T}$ de termes est clos par la relation $\rightarrow_R$ ssi pour tout terme t dans $\mathcal{T}$, s'il existe un terme t' tel que $t \rightarrow_R^* t'$, t' est aussi dans $\mathcal{T}$.

On dit encore que $\rightarrow_R$ préserve l'ensemble $\mathcal{T}$.

Cette première proposition établit les conditions sous lesquelles les deux concepts de complétude suffisante sont équivalents.

Proposition 1.3 *Soit $SP = (S,\ \mathcal{C} \oplus \mathcal{D},\ E)$ une spécification qui définit un système de réécriture R;*
si $\rightarrow_R$ est confluente sur les termes clos de $T(\mathcal{C} \oplus \mathcal{D})$ et si la clôture de $T(\mathcal{C})$ par $\rightarrow_R$ est égale à $T(\mathcal{C})$ alors
la complétude suffisante de SP par rapport à $\mathcal{C}$ est équivalente à la convertibilité de R.

Preuve:
$\Longleftarrow$: la convertibilité de R entraîne la complétude suffisante de SP puisque $\rightarrow_R^*$ est contenue dans $=_E$.

$\Longrightarrow$: Soit t un terme clos dans $T(\mathcal{C} \oplus \mathcal{D})$; il s'agit de trouver un terme clos t_0 dans $T(\mathcal{C})$ tel que $t \rightarrow_R^* t_0$.
Puisque SP est suffisamment complète par rapport à $\mathcal{C}$, il existe un terme clos t' dans $T(\mathcal{C})$ tel que $t =_E t'$.
Puisque $\rightarrow_R$ est confluente sur les termes clos, il existe un terme clos t" dans $T(\mathcal{C} \oplus \mathcal{D})$ tel que $t \rightarrow_R^* t$" et $t' \rightarrow_R^* t$".
La clôture de $T(\mathcal{C})$ par la relation $\rightarrow_R$ étant égale à $T(\mathcal{C})$, puisque t' est dans $T(\mathcal{C})$ alors t" est aussi dans $T(\mathcal{C})$. Par conséquent, $t_0 = t$" convient pour montrer que R est convertible

par rapport à C. □

La clôture de l'ensemble des termes primitifs par la relation de réécriture $\rightarrow_R$ est une propriété essentielle pour garantir l'équivalence des deux concepts de complétude suffisante. Si $\rightarrow_R$ ne préserve pas les termes primitifs, l'équivalence n'est pas toujours vérifiée même si le système est confluent sur les termes clos. Examinons à titre d'exemple, le système suivant :

Exemple 1.3 *Soit le système de réécriture conditionnelle* $(S,\ \mathcal{C} \oplus \mathcal{D},\ R)$ *avec :*

$S = \{elem\},$
$\mathcal{C} = \{c :\rightarrow elem,\ f : elem \rightarrow elem\},$
$\mathcal{D} = \{g : elem \rightarrow elem\},$
$R = \{f(x) \rightarrow g(x)\}.$

R est confluent sur les termes clos et la spécification correspondante est suffisamment complète, puisque tout terme clos contenant g est équivalent à un terme clos comportant uniquement les opérateurs f et c. Cependant, R n'est pas convertible puisqu'il n'existe aucun terme clos t_0 de $T(\mathcal{C})$ tel que $g(c) \rightarrow_R^* t_0$ parce que $\rightarrow_R$ ne préserve pas l'ensemble $T(\mathcal{C})$. ◇

La complétude suffisante d'une spécification est simple à formuler. Cependant, les différentes méthodes de décidabilité de cette propriété mettent en œuvre des procédés relativement complexes. Nos recherches, dans le but de formaliser des conditions suffisantes pour la propriété de complétude suffisante sont fortement inspirées des travaux de Kounalis [66] dans le cadre classique, sans préconditions.

De même qu'il est possible de partager l'ensemble des symboles de fonction en deux parties, il est également possible de diviser l'ensemble E des équations de la spécification en deux ensembles $\mathcal{CE}$ et $\mathcal{DE}$ où $\mathcal{CE}$ comporte des relations entre les constructeurs. Il est par conséquent uniquement formé d'équations dont tous les termes sont primitifs.
Pour notre part, nous considérons d'emblée des systèmes de réécriture pouvant comporter des relations entre les constructeurs car il est souvent utile de définir ce type de relations pour réduire les termes à leur forme normale. Cette importance est mise en évidence dans l'exemple suivant :

Exemple 1.4 **[105]**

Soit $SP = (S,\ \mathcal{C} \oplus \mathcal{D},\ \mathcal{CE} \oplus \mathcal{DE})$ *une spécification algébrique telle que :*

$S = \{bool,\ \mathcal{Z}\};$

$\mathcal{C} = \{true,\ false :\rightarrow bool,\ 0 :\rightarrow \mathcal{Z},\ succ,\ pred : \mathcal{Z} \rightarrow \mathcal{Z}\};$
$\mathcal{CE} = \{succ(pred(x)) = x,\ pred(succ(x)) = x\}$

$\mathcal{D} = \{pair : \mathcal{Z} \rightarrow bool\}$
$\mathcal{DE} = \{pair(0) = true,$
$\quad pair(succ(0)) = false,$
$\quad pair(succ(succ(x))) = pair(x),$
$\quad pair(pred(x)) = pair(succ(x))\},$

On peut prouver que le système de réécriture obtenu en orientant l'ensemble des équations de $\mathcal{CE} \oplus \mathcal{DE}$ de gauche à droite est convertible par rapport aux constructeurs. Cependant, il faut noter que si l'ensemble $\mathcal{CE}$ était vide, il n'aurait pas été possible de réduire le terme

$pair(succ(pred(0)))$ en $pair(0)$, puis en *true* qui est un terme de $T(\mathcal{C})$. Dans ce cas, R n'est pas convertible par rapport à $\mathcal{C}$. $\diamond$

Rappelons également cette proposition qui déduit la convertibilité d'un système de réécriture par rapport aux constructeurs à partir d'une caractérisation des formes normales :

Proposition 1.4 *Soit $SP = (S,\ \mathcal{C} \oplus \mathcal{D},\ E)$ une spécification qui définit un système de réécriture R ;*
Si tout terme clos en forme normale est dans $T(\mathcal{C})$, R est convertible par rapport à $\mathcal{C}$.

Preuve : soit t dans $T(F) \ominus T(BF)$; il s'agit de trouver un terme clos t_0 dans $T(\mathcal{C})$ tel que $t \rightarrow_R^* t_0$. Soit $\bar{t}$ la forme normale de t. L'existence de $\bar{t}$ est garantie par la terminaison de $\rightarrow_R$. Par hypothèse, $\bar{t} \in T(\mathcal{C})$. Par conséquent, $t_0 = \bar{t}$ convient pour montrer que R est convertible. $\square$

Une autre propriété importante liée à la convertibilité est la réductibilité inductive. Nous en donnons la définition et le principal résultat (voir [51, 59] pour plus de détails).

Définition 1.30 Réductibilité inductive
Un terme t dans $T(F,\ X)$ est inductivement réductible par $\rightarrow_R$ ssi toute instance close de t est réductible par $\rightarrow_R$.

Théorème 1.7 *Soit $(S,\ \mathcal{C} \oplus \mathcal{D},\ R)$ un système de réécriture tel que $\rightarrow_R$ soit nœthérienne ; R est convertible par rapport à $\mathcal{C}$ ssi pour tout f dans $\mathcal{D}$, $f(x_1, \ldots,\ x_n)$ est inductivement réductible par $\rightarrow_R$.*

Ce théorème fournit une méthode fondée sur un test de réductibilité inductive pour vérifier la convertibilité d'un système de réécriture par rapport à son ensemble de constructeurs. La complétude de définition formalisée dans la suite est également un concept lié à la complétude suffisante. Intuitivement, si tout opérateur est complètement défini par rapport aux constructeurs, le système est convertible.

Définition 1.31 Complétude de définition
Soit R un système de réécriture associé à une spécification $SP = (S,\ \mathcal{C} \oplus \mathcal{D},\ E)$ et soit f dans $\mathcal{D}$ de profil $s_1 \times \ldots \times s_n \rightarrow s$; f est complètement défini par rapport à $\mathcal{C}$ ssi

$$\forall t_1 \in T(\mathcal{C})_{s_1}, \ldots,\ t_n \in T(\mathcal{C})_{s_n},\ \exists \bar{t} \in T(\mathcal{C})_s \text{ tel que } f(t_1, \ldots,\ t_n) \rightarrow_R^* \bar{t}.$$

Notons que la complétude de définition est directement définie à partir du concept opérationnel de la complétude suffisante (convertibilité), c'est-à-dire en utilisant la relation de réécriture $\rightarrow_R$. En effet, seul ce concept nous est utile par la suite.

La complétude suffisante dans la théorie conditionnelle est indécidable. En effet, le problème de la décidabilité peut être ramené à celui de la théorie de preuves inductives qui est lui-même connu comme étant indécidable. Considérons à titre d'exemple la règle

$$M = N \Rightarrow f(x_1, \ldots,\ x_n) \rightarrow 0$$

où $x_1, \ldots, x_n$ sont des variables de M et N ; f est complètement définie si et seulement si l'équation $M = N$ est valide pour toutes les instances closes de ses variables et cette validité est du ressort de la preuve dans la théorie inductive.

Notre objectif dans le chapitre 2 de cette thèse est de donner des critères caractérisant une sous-classe de spécifications conditionnelles pour laquelle la propriété de convertibilité peut être vérifiée. Nous avons rappelé dans cette partie que la réductibilité inductive de tout terme non primitif de la forme $f(x_1, \ldots, x_n)$ constitue le principal test pour vérifier que le symbole f est complètement défini par rapport aux constructeurs. Dans le chapitre suivant, nous étudions plus particulièrement la complétude suffisante dans le cadre conditionnel et nous établissons une condition suffisante pour la tester dans une classe spécifique de systèmes de réécriture.

Nous nous proposons à présent de définir formellement les spécifications et les systèmes de réécriture conditionnels et de présenter les principaux travaux s'y rapportant. La question essentielle que nous nous sommes en effet posée, et à laquelle nous tentons de donner des éléments de réponse, est la suivante : existe-t-il aussi pour les théories conditionnelles, une méthode de décision pour le problème du mot dans une variété d'algèbres ? Ou encore, qu'advient-il lorsque nous appliquons l'algorithme de Knuth-Bendix aux systèmes de réécriture conditionnelle ?

5 Les spécifications conditionnelles

La raison pour laquelle nous nous intéressons aux spécifications conditionnelles est que celles-ci offrent une plus grande souplesse pour la définition d'un type abstrait algébrique. En effet, un type abstrait de données ne peut pas toujours être spécifié d'une façon simple dans un cadre algébrique pur. Des exemples de tels types sont exhibés dans [101] et [100]. L'une des méthodes pour spécifier ces types est alors de recourir à l'utilisation de fonctions cachées. Néanmoins, cette méthode complique considérablement la lisibilité et la compréhension de la spécification correspondante. La seconde méthode, plus souple et plus naturelle, consiste à définir le type de données par une spécification conditionnelle. Les équations conditionnelles sont des équations algébriques accompagnées d'une précondition qui détermine dans le cas d'une équation, la condition d'égalité des deux termes constituant cette équation et dans le cas d'une règle, la condition d'applicabilité de cette règle.

L'étude des spécifications conditionnelles a soulevé des problèmes délicats. Il s'agit essentiellement d'une part, de formaliser la sémantique sous-jacente à la théorie conditionnelle et de définir des modèles et d'autre part, d'obtenir une méthode de décision pour la preuve inductive dans la théorie conditionnelle.

Un ensemble E d'équations conditionnelles sur des termes permet de définir une congruence syntaxique. Deux termes sont équivalents si on peut passer de l'un à l'autre par une chaîne de remplacements d'égaux par des égaux, chaque remplacement étant permis si la précondition associée a été prouvée récursivement équivalente à *true*. Une règle de réécriture est appliquée en instanciant ses deux membres et en vérifiant récursivement que la précondition se réduit en vrai. Le problème est alors de traduire l'ensemble E en un système de réécriture R qui définit des formes normales uniques caractérisant les classes d'équivalence de la congruence. Il s'agit de généraliser les propriétés de confluence et de terminaison aux systèmes de réécriture conditionnelle et d'établir sous quelles conditions

la procédure de complétion de Knuth-Bendix peut être appliquée dans de tels systèmes.

Il est difficile de comparer les travaux ayant trait à la réécriture conditionnelle de termes car ils dépendent de la classe des modèles spécifiés, de la classe de théorèmes que l'on veut prouver ou encore des outils de preuve de théorèmes. Pour mener à bien cette comparaison, nous allons en premier lieu procéder à une structuration selon les différents critères intervenant dans la définition d'un système de réécriture.
Formellement, une spécification conditionnelle se définit comme suit :

Définition 1.32 Spécification conditionnelle
Une spécification conditionnelle est la donnée d'une signature Σ et d'un ensemble E d'équations conditionnelles.
Σ comporte un ensemble fini S de sortes et un ensemble fini F de symboles d'opérateurs. Les équations de E sont d'une façon générale de la forme :

$$p \Rightarrow g = d$$

où p est appelée prémisse (ou précondition) de l'équation et la partie "$g = d$" est sa conséquence.

5.1 La classe d'algèbres

Une première classification que nous pouvons faire concerne **la classe d'algèbres** qu'il est possible d'appréhender. Cette classe d'algèbres est étroitement liée à la forme syntaxique des préconditions. Généralement, la restriction à des formes particulières de préconditions est motivée par la nécessité de simplifier les preuves d'insatisfiabilité des contextes. Cette classification est principalement fondée sur le fait d'inclure ou non les booléens dans la spécification. Inclure les booléens amène à considérer la théorie du premier ordre avec toute sa puissance, tandis que le traitement d'une spécification conditionnelle ne comportant pas la sous-spécification booléenne se situe dans un cadre algébrique où tous les opérateurs sont uniquement traités par des techniques de réécriture. Dans ce cadre, quatre approches principales se sont distinguées.

L'approche algèbre libre :

Cette approche, initialement considérée par Kaplan, se situe dans un cadre algébrique. Dans [54], l'auteur montre que d'un point de vue algébrique, les résultats obtenus dans le cas purement équationnel s'étendent d'une façon naturelle aux théories conditionnelles.
Il s'agit de considérer des implications équationnelles qui ne comportent pas de fonctions à valeurs booléennes. La condition d'une équation est une conjonction d'égalités de deux termes de même sorte. Plus formellement, une équation conditionnelle est de la forme :

$$s_1 = t_1 \ \wedge \ \ldots \wedge \ s_n = t_n \Rightarrow s = t$$

Dans la littérature, des équations de ce type sont également appelées *clauses de Horn équationnelles* ou encore *clauses de Horn positives.* Dans ce cadre d'étude, le symbole d'égalité qui apparaît dans la prémisse d'une équation conditionnelle a un statut spécifique. Il est utilisé dans un but syntaxique pour relier les membres d'une même équation. Il n'est pas considéré comme un opérateur booléen car cela mènerait à toute la complexité du raisonnement logique.

Définition 1.33 Sémantique d'une égalité conditionnelle

Soient $SP = (S,\ F,\ E)$ une spécification conditionnelle, A une SP-algèbre et X un ensemble de variables;
soit e une égalité conditionnelle de la forme $s_1 = t_1\ \wedge \ldots \wedge\ s_n = t_n \Rightarrow s = t$;
A valide e et on note $A \models e$ ssi

$$\forall \sigma : X \to A,\ \forall i \in [1 \ldots n],\ \bar{\sigma} s_i = \bar{\sigma} t_i \Longrightarrow \bar{\sigma} s = \bar{\sigma} t.$$

où $\bar{\sigma}$ désigne l'unique extension de σ à l'ensemble $T(F,\ X)$.
A valide un ensemble E d'égalités conditionnelles ssi A valide toute égalité de E.
On dit alors que A est un modèle de E.

Lorsque la spécification ne comporte pas de booléens, la classe des modèles qu'elle capture est la classe de toutes les algèbres qui satisfont les axiomes conditionnels. Dans cette approche, l'existence d'algèbres libres, et par conséquent d'une algèbre initiale, est connue depuis longtemps [39].

Théorème 1.8 [54]

Soient Alg_{SP} (respectivement Gen_{SP}) la classe de tous les modèles (respectivement tous les modèles finiment engendrés) de E et soit $T(F,\ X)$ (respectivement $T(F)$) l'algèbre des termes avec variables (l'algèbre des termes clos);

1. *Alg_{SP} et Gen_{SP}, munis de leurs SP-morphismes sont des catégories non vides.*
2. *soit l'ordre $\leq$ défini sur des SP-algèbres A, B par $A \leq B$ ssi il existe un SP-morphisme $h : A \to B$.*
   Alg_{SP} et Gen_{SP} sont des treillis complets.
3. *Il existe une plus petite congruence $\equiv_E$ sur $T(F,\ X)$ engendrée par E. Cette congruence est construite en utilisant la théorie du point fixe.*
4. *Le modèle initial de Alg_{SP} et de Gen_{SP} est:*
   $I = T(F)/\equiv'_E$ où $\equiv'_E$ est la plus petite congruence engendrée par E sur $T(F)$.

Proposition 1.5 [54]

La construction de $\equiv_E$ définie sur l'algèbre $T(F,\ X)$ des termes avec variables se fait en utilisant la théorie du point fixe de la façon suivante:
Soit $Congr_F$ le treillis complet des congruences sur $T(F,\ X)$ ordonné par inclusion et soit la fonctionnelle $\theta : Congr_F \to Congr_F$ qui, à une congruence $\equiv_1$, associe la plus petite congruence $\equiv_2$ telle que:

1. *si pour tout i dans $[1 \ldots n]$, $\sigma s_i \equiv_1 \sigma t_i$ alors $\sigma s \equiv_2 \sigma t$*
2. *$\equiv_1 \subset \equiv_2$*

pour toute substitution σ dans $T(F,\ X)$ et pour toute équation conditionnelle $\wedge_{i=1}^{n}\ s_i = t_i \Rightarrow s = t$ dans E.
θ est une fonctionnelle continue et monotone sur le treillis $Congr_F$, $\equiv_E$ est son plus petit point fixe et $\equiv_E = \cup_{i \in N}\ \theta^{(n)}\ (=)$ où $=$ est l'égalité syntaxique sur $T(F,\ X)$.
On dit alors que e est déduite syntaxiquement de E et on note $E \vdash e$.

De même que dans le cas classique, sans préconditions, on obtient un résultat de complétude *à la Birkhoff*.

Théorème 1.9 [54]

Soient $SP = (\Sigma,\ E)$ une spécification conditionnelle et e une égalité conditionnelle; e est valide dans tout modèle de E ssi e peut être déduite syntaxiquement de E soit,

$$Alg_{SP} \models e \Longleftrightarrow E \vdash e$$

Nous donnons dans le théorème suivant une autre formulation pour construire explicitement la congruence sur les termes clos :

Théorème 1.10 [54]

Soit $SP = (S,\ F,\ E)$ une spécification conditionnelle. Les SP-algèbres, munies de leurs homomorphismes forment la catégorie Alg_{SP} dont l'objet initial est $T_{SP} = T(F)/\equiv'_E$ où $\equiv'_E$ est la plus petite congruence engendrée par E sur $T(F)$.
$\equiv'_E$ se construit par récurrence de la manière suivante :

- *soit $\equiv_0$ la plus petite relation de congruence qui contient la relation ϕ_0 définie par :*
  $u\ \phi_0\ v$ ssi il existe une équation $s = t$ dans E et une substitution $\sigma : X \to T(F)$ telles que $\sigma s = u$ et $\sigma t = v$.
- *soit $\equiv_{i+1}$ la plus petite congruence qui contient la relation ϕ_{i+1} sur $T(F)$ suivante :*
  $u\ \phi_{i+1}\ v$ ssi $u\ \equiv_i\ v$ ou il existe une équation conditionnelle dans E soit $\wedge_{i=1}^{n}\ s_i = t_i \Rightarrow s = t$ et une substitution $\sigma : X \to T(F)$ telles que $\sigma s = u$, $\sigma t = v$ et pour tout i dans $[1 \ldots n]$, $\sigma s_i \equiv_i\ \sigma t_i$

alors $\equiv'_E\ =\ \cup\ \{\equiv_i,\ i \geq 0\}$

Cette approche fournit une méthode de semi-décision pour la théorie équationnelle. Néanmoins, elle est insuffisante pour appréhender la théorie conditionnelle d'un ensemble d'axiomes conditionnels en ce sens qu'elle ne permet pas de considérer la classe de toutes les équations conditionnelles valides dans tous les modèles des axiomes d'une spécification conditionnelle.

L'approche *algèbre libre* présente aussi un résultat de caractérisation de la propriété de Church-Rosser par un critère de convergence de paires critiques contextuelles, valide pour des systèmes dits *réducteurs*. Ces paires critiques sont une extension naturelle aux systèmes conditionnels des paires critiques classiques. Prouver la convergence d'une paire critique contextuelle $C \Rightarrow M = N$ où C est de la forme $\wedge_{i=1}^{n}\ c_i = c'_i,\ c_i,\ c'_i \in T(F,\ X)$ consiste à montrer que, pour tout indice i dans $[1 \ldots n]$, σc_i et $\sigma c'_i$ se réduisent en un même terme. Dans cette approche, les auteurs ont recours à un algorithme d'unification conditionnelle pour mener à bien les preuves de convergence.

Cependant, l'approche *spécification sans booléens* présente trois inconvénients majeurs :

- Le test des paires critiques comporte en général un nombre infini de calculs et il est alors nécessaire de considérer des conditions suffisantes restrictives afin de mener à bien le test de convergence.
- Beaucoup de problèmes à traiter incluent les booléens et cette approche n'en tient pas compte.

- La propriété de Church-Rosser ne suffit pas pour garantir la complétude des techniques de réécriture dans la preuve de théorèmes. En effet, à la différence du cas classique, on a besoin de considérer également des preuves d'assertions conditionnelles. Un exemple qui illustre cette insuffisance est le suivant [58]:
  Soit R constitué des règles

$$a = b \Rightarrow c \rightarrow d,\ a = b \Rightarrow c \rightarrow e$$

  R est confluent puisque $I(R) = \emptyset$, mais la méthode échoue lorsque nous voulons prouver l'équation $a = b \Rightarrow d = e$ parce qu'il n'est pas possible de réduire d et e en un même terme dans le contexte $a = b$.

L'approche *Log*-algèbre:

Cette approche se situe dans un cadre algébrique avec une logique prédéfinie. Elle est due à Navarro et Orejas [81, 79] et elle considère des spécifications conditionnelles que l'auteur appelle encore *Log*-spécifications, et qui sont des extensions de la spécification des valeurs booléennes. La précondition d'une équation conditionnelle est une formule propositionnelle arbitraire pouvant comporter le symbole d'égalité. Plus formellement, une équation est de la forme:

$$c \Rightarrow s = t$$

où c est une formule propositionnelle arbitraire pouvant inclure en particulier le symbole d'égalité.
De plus, pour tout symbole d'opérateur f de profil $s_1 \times \ldots \times s_n \rightarrow s$, tel que $n \geq 1$ et f n'est pas un connecteur logique, la sorte s_i est différente de la sorte booléenne, pour tout indice i dans $[1 \ldots n]$. Cette restriction imposée par l'auteur est justifiée par son souci d'interpréter ces fonctions comme des prédicats.

L'auteur considère la spécification suivante SP_{bool} des booléens:

Définition 1.34 Spécification des booléens *[79]*
La spécification SP_{bool} des valeurs booléennes est une spécification $(\Sigma_{bool}, E_{bool})$ telle que $\Sigma_{bool} = (S_{bool}, F_{bool})$ avec:
$S_{bool} = \{bool\}$

$$\begin{aligned} F_{bool} = \{&true,\ false :\rightarrow bool,\\ &\neg : bool \rightarrow bool,\\ &\wedge,\ \vee,\ \Rightarrow,\ \Leftrightarrow : bool \times bool \rightarrow bool\} \end{aligned}$$

$$\begin{aligned} E_{bool} = \{&p \wedge true = p,\\ &p \vee false = p,\\ &p \wedge \neg p = false,\\ &p \vee \neg p = true,\\ &p \wedge q = q \wedge p,\\ &p \vee q = q \vee p,\\ &p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r),\\ &p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r),\\ &p \Rightarrow q = (\neg p) \vee q,\\ &p \Leftrightarrow q = (p \wedge q) \vee ((\neg p) \wedge (\neg q))\} \end{aligned}$$

Remarque: L'équation $c \Rightarrow s = t$ peut être vue comme une abréviation de la forme: $c = true \Rightarrow s = t$. c est le contexte de l'équation et la partie $s = t$ sa conséquence.

Définition 1.35 ***Log*-algèbre**

Soit $SP = (\Sigma,\ E)$ une Log-spécification et Alg_{SP} la classe d'algèbres associées à SP. $LogAlg_{SP}$ dénote la sous-classe d'algèbres A dans Alg_{SP} telle que

$$A_{bool} = \{\boldsymbol{true},\ \boldsymbol{false}\}$$

où A_{bool} est le support booléen de l'algèbre A et $\boldsymbol{true}$, $\boldsymbol{false}$ représentent les valeurs respectives dans A des constantes true et false.

Cette approche permet de définir un symbole de fonction eq représentant l'égalité dans une sorte quelconque avec les équations suivantes :

$$(1) : eq(x,\ x) = true, \quad (2) : eq(x,\ y) \Rightarrow x = y$$

où le symbole $=$ représente l'égalité ensembliste. En effet, considérons un modèle quelconque A ; on peut déduire à partir des équations ci-dessus que pour tous termes a, b dans A,

$$\begin{cases} eq_A(a,\ b) = true \text{ si } a = b & \text{conséquence de (1)} \\ eq_A(a,\ b) \neq true \text{ si } a \neq b & \text{contraposée de (2)} \end{cases}$$

A présent, considérons un *Log*-modèle B ; autrement dit, $B_{bool} = \{true,\ false\}$ avec $true \neq false$;

$$\forall a,\ b,\ eq_B(a,\ b) = \begin{cases} true \text{ si } a = b \\ false \text{ si } a \neq b \end{cases}$$

Dans cette approche, les modèles admissibles sont les modèles qui satisfont les équations de la spécification considérée et dont le support de sorte booléenne se limite exactement aux valeurs des deux constantes *true* et *false*. Cette approche avec booléens ne garantit pas, en général, l'existence d'une algèbre initiale, comme le montre l'exemple suivant :

Exemple 1.5 **[79]**

Soit la spécification $SP = (S,\ F,\ E)$ avec

- $S = S_{bool}$
- $F = F_{bool} \cup \{a :\rightarrow bool\}$
- $E = E_{bool}$

Cette spécification admet deux modèles ; dans le premier, l'interprétation de a est $\boldsymbol{true}$ et dans le second, l'interprétation de a est $\boldsymbol{false}$. La spécification considérée n'admet donc pas de modèle initial. ◇

Un second apport intéressant de la méthode des *Log*-algèbres est la définition d'un système de déduction L correct et complet par rapport à la classe $LogAlg_{SP}$.

Théorème 1.11 **[79]**

Soit $SP = (\Sigma,\ E)$ une Log-spécification et soit e une égalité conditionnelle de la forme $c \Rightarrow s = t$, alors

$$E \vdash_L e \iff A \models e,\ \forall A \in LogAlg_{SP}.$$

où $E \vdash_L e$ signifie que le système de déduction L valide l'égalité e.

Lemme 1.2 *La théorie définie par le système de règles d'inférence L et les axiomes de E est l'ensemble des égalités conditionnelles $c \Rightarrow t = t'$ qui sont déductibles à partir des axiomes de E en appliquant plusieurs fois les règles de déduction de L.*

Un second intérêt de la méthode est qu'elle autorise l'utilisation d'une nouvelle règle d'inférence, le raisonnement par cas, pour prouver les théorèmes. Cette règle se formule comme suit :

$$\frac{E \vdash_L \lambda X.\ t = t' \text{ si } c_1,\ E \vdash_L \lambda X.\ t = t' \text{ si } c_2}{E \vdash_L \lambda X.\ t = t' \text{ si } (c_1 \vee c_2)}$$

L'étude des spécifications paramétrées a également été effectuée dans cette approche. Deux aspects ont été abordés à savoir, la théorie de la démonstration et la correction des *Log*-spécifications paramétrées. En particulier, une caractérisation syntaxique de la correction d'une spécification paramétrée est obtenue à partir de deux propriétés : la *bool-persistance* et la *persistance* [81].

On peut voir à cette approche les inconvénients suivants :

- La méthode de déduction par le système L est semi-décidable et les problèmes de preuve dans les contextes booléens restent ouverts. Il n'existe pas en effet de méthode décidable pour montrer par exemple, que la disjonction de deux contextes, soit $c_1 \vee c_2$ est équivalente à *true* dans une *Log*-algèbre.

- L'auteur a besoin d'une spécification booléenne complète et utilise de ce fait des axiomes non-orientables tels que la commutativité du connecteur logique $\wedge$. D'un point de vue opérationnel, cette utilisation pose un problème de terminaison.

- L'existence d'un modèle initial n'est pas garantie.

L'approche algèbre initiale :

Cette approche considère aussi des spécifications conditionnelles avec booléens, mais contrairement à la précédente, elle se situe dans un cadre algébrique où les symboles de prédicats sont simplement des symboles de fonction avec des résultats de sorte booléenne. Ce choix a été adopté par Ganzinger [35] et par nous-même [8]. Une équation conditionnelle est de la forme

$$p \Rightarrow s = t$$

où le terme p est de sorte booléenne et s et t sont des termes de sorte identique. p est de la forme "$\wedge_{i=1}^{n}\ p_i = p'_i$" où pour tout i dans $[1 \ldots n]$, $p'_i \in \{true,\ false\}$.

Cette approche se place dans un cadre d'étude algébrique dont le support booléen est constitué des deux seules constantes *true* et *false*. De plus, l'hypothèse d'irréductibilité de ces deux constantes est essentielle. Ainsi, le complémentaire de *true* vaut *false* et réciproquement, celui de *false* est égal à *true*. Ceci entraîne en particulier que la réductibilité à *true* d'une précondition close se ramène à la réductibilité de t_i à t'_i pour tout i dans $[1 \ldots n]$.

Imposer que la précondition d'une règle soit de la forme précisée ci-dessus n'est pas une restriction trop forte. Cette forme syntaxique suffit en effet, à exprimer la négation d'une expression booléenne simple p, sans connecteur logique, selon le principe suivant :

$$\neg(p = true) = (p = false),\ \neg(p = false) = (p = true)$$

La correction du principe est garantie par la restriction du support booléen d'une algèbre quelconque à l'ensemble $\{\boldsymbol{true}, \boldsymbol{false}\}$. D'autre part, une équation conditionnelle de la forme

$$(P_1 \vee \ldots \vee P_n) \Rightarrow s = t$$

où $\vee$ représente le connecteur logique disjonctif et P_j est de la forme précitée est équivalente à l'ensemble suivant d'équations conditionnelles :

$$\{P_1 \Rightarrow s = t, \ldots, P_n \Rightarrow s = t\}$$

Théorème 1.12 *Soit $SP = (S, F, E)$ une spécification conditionnelle ;*
Les SP-algèbres, munies de leurs homomorphismes forment la catégorie Alg_{SP} dont l'objet initial est $T_{SP} = T(F)/\equiv_E$ où $\equiv_E$ est la plus petite congruence engendrée par E.
$\equiv_E$ se construit par récurrence de la même façon que pour l'approche algèbre libre. *Par conséquent, cette construction peut être également formulée dans cette approche par le théorème 1.7.*

Cette approche utilisant des préconditions dont la syntaxe spécifique est algébrique, garantit l'existence d'une algèbre initiale. Toutefois, manipuler des préconditions de sorte booléenne nécessite un traitement minutieux. En effet, comme le montre l'exemple qui suit, l'algèbre initiale obtenue peut ne pas être conforme au résultat attendu quant à sa partie booléenne. Ce problème se pose également au niveau de la première approche présentée, notamment lorsque les préconditions utilisent des symboles de fonction à destination booléenne, comme c'est le cas si pour un certain i dans $[1 \ldots n]$, $t_i = true$ ou $t_i = false$; néanmoins, les auteurs tels que [54] et [35] n'abordent pas ce problème dans leurs travaux.

Exemple 1.6 [58]
Soit la spécification $SP = (S, F, E)$ suivante :

- $S = \{bool\}$,
- $F = \{true, false, p, q :\rightarrow bool\}$,
- $E = \{p = false \Rightarrow q = true\}$

$\equiv_E$ est l'identité et le support de l'algèbre initiale de la spécification SP est l'ensemble $\{\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{true}, \boldsymbol{false}\}$ où $\boldsymbol{p}$, $\boldsymbol{q}$, $\boldsymbol{true}$, $\boldsymbol{false}$ sont les valeurs respectives des constantes p, q, $true$ et $false$.
Par conséquent, la partie booléenne de l'algèbre initiale n'est pas isomorphe à l'ensemble $\{\boldsymbol{true}, \boldsymbol{false}\}$. $\diamond$

Nous définissons dans la suite de nos travaux deux propriétés assurant qu'une spécification conditionnelle est fidèle par rapport à la sorte booléenne. Ces propriétés sont connues sous le nom de *complétude* et de *consistance par rapport aux booléens* et elles traduisent le comportement de la spécification par rapport au noyau $(\{bool\}, \{true, false\}, \emptyset)$.

L'approche clauses de Horn :

Cette approche a été mise en œuvre par Kounalis et Rusinowitch [67]. Les auteurs considèrent des clauses de Horn et établissent une méthode de preuve de théorèmes dans le modèle initial d'un ensemble de Horn en étendant la notion équationnelle de réductibilité inductive. La méthode est fondée sur une stratégie de réfutation complète et présente les avantages suivants :

- Elle n'échoue pas lorsqu'une équation n'est pas orientable.
- Le système peut contenir une clause dont la partie négative est plus grosse que le littéral positif. Cette situation n'entraîne pas d'échec car l'utilisation de ce type de clauses est contrôlée. En effet, il n'est pas permis par exemple, d'activer l'une de ces clauses pour la règle d'inférence dite de *paramodulation.*

Le problème du mot devient décidable dans les théories axiomatisées par des ensembles de Horn saturés et qui préservent les termes clos. Un ensemble de Horn S préserve les termes clos si tout littéral positif $s = t$ qui apparaît dans une clause de S est soit orientable, soit satisfait $Var(s) = Var(t)$ et si pour toute clause C de S, les variables de chaque littéral négatif de C sont contenues dans le littéral positif. Cette approche se situe en marge de celles présentées jusqu'à présent puisque les auteurs obtiennent de nouvelles règles d'inférence pour les systèmes conditionnels en utilisant des méthodes issues du calcul des prédicats du premier ordre. Elle est toutefois très prometteuse.

5.2 La terminaison de la réécriture conditionnelle

Une seconde classification, liée à **la propriété de nœthérianité des systèmes de réécriture** peut être établie. En effet, appliquer une règle de réécriture conditionnelle sur un terme t consiste d'une part, à remplacer l'instance du membre gauche dans t par l'instance correspondante du membre droit, et d'autre part, à évaluer la précondition instanciée à *true*. Pour garantir que toute réécriture se termine, il est nécessaire d'imposer des restrictions sur les règles. La première étape de réécriture, similaire à celle dans la réécriture classique, sans préconditions, a été largement étudiée dans ce cadre. La solution préconisée par divers auteurs est de définir des règles de réécriture de telle façon que toute instance du membre gauche soit plus grande que l'instance correspondante du membre droit par un ordre bien fondé vérifiant certaines propriétés. Le lecteur peut trouver dans [75, 18, 25] et [98], une synthèse des différents ordres utilisés dans la littérature. Quant à la seconde étape de réécriture, elle peut mener également à des dérivations infinies, comme le montre l'exemple suivant :

Exemple 1.7 *Soit R constitué de la règle*

$$pred(x) > 0 = true \Rightarrow x > 0 \rightarrow true$$

où le symbole pred désigne la fonction prédécesseur sur les entiers.
Nous nous proposons d'évaluer l'expression $-2 > 0$; ce calcul produit la dérivation infinie suivante :
$-2 > 0 \rightarrow_R true$ *si* $-3 > 0 \rightarrow_R^* true$
$-3 > 0 \rightarrow_R true$ *si* $-4 > 0 \rightarrow_R^* true$
$-4 > 0 \rightarrow_R true$ *si* .... $\diamond$

Pour éviter ce type de dérivation cyclique, deux tendances se sont distinguées, l'une est une extension naturelle du cas classique, et l'autre travaille avec des systèmes hiérarchiques.

L'approche système décroissant :

Dans cette approche, pour toute règle de réécriture $p \Rightarrow g \rightarrow d$ dans R, et pour toute substitution σ, d'une part, σd est plus petit que σg selon un ordre de réduction $>$, et

d'autre part, σp est plus petit que σg, la comparaison se faisant cette fois avec une extension de $>$ qui possède la propriété de sous-terme propre. La première propriété assure la terminaison de la réduction inconditionnelle et la seconde garantit que les termes apparaissant dans l'évaluation récursive de la précondition diminuent au fur et à mesure de taille, assurant de ce fait sa terminaison. L'idée d'établir un ordre entre le membre gauche et la précondition d'une règle conditionnelle a été proposée en premier lieu par Kaplan [55, 57]. L'auteur utilise un ordre de simplification pour comparer de façon uniforme d'une part, membre gauche et précondition et d'autre part, membre gauche et membre droit. Cette idée a donné naissance aux systèmes simplifiants. Par la suite, Jouannaud et Waldmann ont montré qu'un ordre de réduction suffit pour mener à bien cette comparaison [53] et ils ont défini les systèmes réducteurs. Récemment, Dershowitz et Okada ont généralisé la méthode en montrant que pour comparer le membre gauche avec la précondition, il suffit de définir une extension d'un ordre de réduction qui possède la propriété de sous-terme propre. Les auteurs spécifient ainsi une classe de systèmes, appelés systèmes décroissants, plus générale que celle des systèmes réducteurs ou simplifiants [19]. Un système de réécriture décroissant se définit formellement comme suit :

Définition 1.36 Système de réécriture décroissant

Soit $>$ un ordre de réduction. R est dit décroissant ssi il existe une extension bien fondée $\succ$ de $>$ qui vérifie les deux propriétés suivantes :

1. *$\succ$ contient la relation de sous-terme propre, c'est-à-dire si t est un sous-terme propre de s, alors $s \succ t$.*
2. *pour toute règle $\wedge_{i=1}^{n}\ p_i = p'_i \Rightarrow g \rightarrow d$ dans R, pour toute substitution σ,*

$$\sigma g > \sigma d,\ \sigma g \succ \sigma p_i,\ \sigma p'_i,\ \forall i \in [1 \ldots n].$$

Dans cette approche, le principal théorème établissant la terminaison de la réduction récursive s'énonce comme suit :

Théorème 1.13 [19]

Si R est un système de réécriture conditionnelle décroissant, $\rightarrow_R$ termine.

L'approche système hiérarchique :

Cette approche a en réalité vu le jour avant la précédente. Elle a été définie dans un premier temps par Pletat, Engels et Ehrich [93] puis a été formalisée par Rémy [94] et Zhang [105] dans un cadre de hiérarchie à deux niveaux. Il s'agit d'évaluer la précondition dans un sous-système de sorte que toute dérivation cyclique ne puisse être engendrée. Comme nous le verrons par la suite, cette approche est modulaire et déterministe et, la développer nous a permis d'obtenir des conditions suffisantes pour tester la confluence sur les termes clos d'un système hiérarchique. En effet, en se plaçant dans un cadre hiérarchique, il est possible de calculer les contextes sans utiliser toute la puissance potentielle des systèmes conditionnels. Ceci constitue un avantage certain car, à l'heure actuelle, combiner l'utilisation d'assertions additionnelles et de règles conditionnelles est une tâche ardue. L'approche hiérarchique est de plus très pratique de par sa modularité, pour définir des spécifications volumineuses. Cependant, elle présente l'inconvénient d'être plus restrictive. Il est difficile par exemple de spécifier les entiers relatifs avec les constructeurs **0**, **successeur** et **prédécesseur** et l'ordre $\leq$ par la méthode hiérarchique :

Exemple 1.8 [58]

1. $succ(pred(x)) \rightarrow x$
2. $pred(succ(x)) \rightarrow x$
3. $succ(x) \leq y \rightarrow x \leq pred(y)$
4. $pred(x) \leq y \rightarrow x \leq succ(y)$
5. $0 \leq 0 \rightarrow true$
6. $0 \leq pred(0) \rightarrow false$
7. $0 \leq x = true \Rightarrow 0 \leq succ(x) \rightarrow true$
8. $0 \leq x = false \Rightarrow 0 \leq pred(x) \rightarrow false.$

En effet, il serait dans ce cas naturel de définir les constructeurs 0, *succ* et *pred* au plus bas niveau (le niveau 0) et l'opérateur $\leq$ à un niveau de hiérarchie supérieur. Ceci entraîne que les deux dernières règles ne sont pas hiérarchiques puisque $\leq$ apparaît aussi dans leurs préconditions. $\diamond$

Cet exemple illustre en particulier que les systèmes décroissants et les systèmes hiérarchiques constituent des classes disjointes de systèmes de réécriture conditionnelle. Ces classes peuvent cependant être combinées pour définir un système hiérarchique dont les règles sont décroissantes.

5.3 La définition de la relation de réécriture conditionnelle

Le troisième critère permettant de structurer les résultats dans le domaine conditionnel est **la définition même de la relation de réécriture.** On distingue principalement trois définitions différentes de la relation de réécriture conditionnelle. La première est appelée réduction récursive parce que la démarche consiste à évaluer récursivement une précondition de règle avant d'appliquer celle-ci. La seconde relation est une variante de la réduction récursive. Elle est en effet, définie selon le même principe, mais comporte toutefois des conditions d'applicabilité supplémentaires liées à la hiérarchie du système. La troisième relation de réécriture consiste à réduire les termes sans tenter de vérifier la précondition de la règle, mais plutôt en ajoutant celle-ci à un certain contexte.

La réduction récursive :

Soit $p \Rightarrow g \rightarrow d$ une règle de réécriture conditionnelle. Informellement, on peut dire que, pour toute substitution σ, σg se réécrit en σd si σp se réduit en *true*. La réduction récursive est une relation transitive définie par :

Définition 1.37 Relation de réduction récursive

Soit R un système de réécriture conditionnelle ; la relation de réduction récursive est la plus petite relation réflexive et transitive $\rightarrow_R^*$ telle que, pour tous termes clos s, t, toute occurrence u de s, toute substitution σ et toute règle $p \Rightarrow g \rightarrow d$ de R,

$$s_{/u} = \sigma g,\ t = s[u \leftarrow \sigma d] \text{ et } \sigma p \rightarrow_R^* true \Longrightarrow s \rightarrow_R t$$

Rémy, dans sa thèse, a montré le résultat suivant par induction point fixe [94]

Proposition 1.6 Propriété de Church-Rosser

Si la relation $\rightarrow_R^*$ est confluente et nœthérienne, tout terme t admet une forme normale $\bar{t}$ unique. De plus,

$$\forall s,\ t \in T(F,\ X),\ s \equiv_{SP} t \Longleftrightarrow \bar{s} = \bar{t}.$$

où $\bar{s}$ et $\bar{t}$ sont les formes normales respectives de s et t dans R.

La confluence de la relation de réduction récursive ne peut pas être étudiée directement, en raison des problèmes de terminaison. Elle a été considérée par Kaplan dans un cadre de travail spécifique, dans lequel des hypothèses sont supposées vérifiées pour obtenir une méthode de décision de la propriété de confluence. L'auteur se place dans un cadre algébrique où une précondition est formée d'une conjonction d'égalités entre deux termes, soit $s_1 = t_1 \ \wedge \ldots \wedge \ s_n = t_n$. D'un point de vue opérationnel, il est alors possible d'interpréter le symbole "=" de trois façons différentes [20]. "=" peut correspondre à l'égalité syntaxique, il peut être interprété comme la fermeture symétrique et transitive de $\rightarrow_R$, soit $\leftrightarrow_R^*$, ou encore comme la relation $\downarrow_R$ définie par $s \downarrow_R t$ ssi il existe un terme u tel que $s \rightarrow_R^* u$ et $t \rightarrow_R^* u$. Le premier choix est trop restrictif car il ne permet pas de réécrire les préconditions des règles. Le second choix est le plus général et simule exactement la congruence conditionnelle. Il définit la classe des *systèmes de réécriture naturles*. Toutefois, il ne correspond pas au formalisme de la réécriture puisqu'alors les axiomes ne sont pas orientés. Le dernier choix, correspondant à la réduction récursive et définissant la classe des *systèmes de réécriture standards*, est celui adopté par Kaplan [54]. L'auteur établit des résultats d'indécidabilité de la relation dans le cas général. Dans le cadre plus restreint des systèmes réducteurs, il caractérise la propriété de Church-Rosser par la convergence des paires critiques contextuelles. Cependant, le principal inconvénient de cette caractérisation est la nécessité de considérer toutes les instances closes d'une paire critique pour tester sa convergence. Une relation plus adéquate pour travailler sur les termes avec variables est la relation de réécriture contextuelle présentée plus tard.

La relation de réécriture hiérarchique :

Elle est liée à la notion de hiérarchie d'un système de réécriture conditionnelle. La hiérarchie a été définie dans le but d'assurer la nœthérianité d'un système de réécriture conditionnelle. Elle peut être prise en compte soit dans la structure du système, soit dans la définition de la relation de réécriture. Un système hiérarchique est un système de réécriture conditionnelle dont la signature et l'ensemble des règles sont définis par palliers successifs, constituant les différents niveaux de hiérarchie. La relation hiérarchique est fondée sur le principe suivant : si un terme t est réductible par la règle $p \Rightarrow g \rightarrow d$ en utilisant la substitution σ, et si σg est un terme du i-ème niveau de hiérarchie, alors la réécriture de t n'est autorisée que si σp est de niveau strictement plus petit que i. Cette relation est a priori plus pauvre que la réduction récursive pour laquelle les instances des préconditions sont des termes quelconques. Néanmoins, un résultat essentiel peut se trouver dans [80] établissant l'équivalence entre ces deux relations lorsque la relation hiérarchique possède la propriété de complétude suffisante, c'est-à-dire lorsqu'elle permet de réduire tout terme clos d'un quelconque niveau de hiérarchie dont la sorte est dans S_i en un terme qui lui est équivalent et qui appartient à $T(F_i)$.

La relation de réécriture contextuelle:

Elle est un outil pour manipuler les termes avec variables et elle fournit une méthode effective du test de convergence des paires critiques contextuelles. Sa définition est valide si on considère des *Log*-modèles, dont la restriction à la sorte booléenne est l'algèbre booléenne à deux éléments *true* et *false*.

On peut définir plusieurs variantes de la relation de réécriture contextuelle. Historiquement, la première est celle proposée par Rémy dans sa thèse[94] où la relation dépend d'un contexte formé d'expressions booléennes. Informellement, un contexte c est un terme booléen qui implique, pour une certaine substitution σ, que la précondition p d'une règle est vraie. Si le terme booléen $(\sigma c \Rightarrow \sigma p)$ se réduit à *true*, alors la réécriture contextuelle $\sigma g \rightarrow_R \sigma d$ (σc) est autorisée. Cette relation de réécriture a été étudiée dans un cadre hiérarchique. L'existence d'ensembles complets minimaux de formes normales contextuelles est assurée et des résultats de confluence sur les termes clos sous des conditions spécifiques sont établis dans[10].

Ultérieurement, une seconde forme de réécriture contextuelle a été définie par Ganzinger[35]. Il s'agit de la réécriture sous contexte. Etant donné un ensemble d'équations inconditionnelles $C = \{c_1 = c'_1, \ldots, c_n = c'_n\}$, l'application de la règle $p \Rightarrow g \rightarrow d$, avec la substitution σ est autorisée si $\sigma p \in C\downarrow$. $C\downarrow$ dénote l'ensemble de toutes les assertions de convergence de C dérivées en utilisant les axiomes de réflexivité, de symétrie et de clôture par congruence. L'auteur utilise une technique de décomposition des paires critiques contextuelles par nécessité pour tester leur convergence. Son travail a fait l'objet d'une implantation en Prolog sur SUN, baptisée CEC[3].

Une troisième forme de réécriture contextuelle peut être définie à partir du raisonnement par cas, qui constitue une technique performante de preuve du premier ordre. Etant donné un terme t tel que t est réductible par la règle $p \Rightarrow g \rightarrow d$ avec la substitution σ, le raisonnement par cas consiste d'une façon générale, à appliquer la règle d'inférence suivante:

$$\frac{c \Rightarrow t[\sigma g]}{c \wedge \sigma p \Rightarrow t[\sigma d],\ c \wedge \neg \sigma p \Rightarrow t}$$

Cette technique est plus longuement explicitée dans le chapitre 4 de ce document.

6 Synthèse des travaux dans le cadre conditionnel

Nous présentons dans cette partie, les travaux effectués dans le cadre des spécifications conditionnelles de termes. Un certain nombre de ces travaux s'intègrent dans les trois aspects que nous avons évoqués ci-dessus:

Un premier résultat a été proposé par Pletat, Engels et Ehrich[93] puis formalisé par Rémy dans sa thèse[94] qui garantit que la justification des préconditions se termine, et qui apporte de ce fait une solution au problème de terminaison de la réduction récursive. Cette solution a consisté à définir les systèmes de réécriture hiérarchique. L'approche hiérarchique a été développée par Rémy et Zhang[105, 107] dans un cadre de hiérarchie à deux niveaux. Par la suite, les travaux ont été repris dans cette thèse (voir le chapitre 3), où l'on s'intéresse à la *Log*-algèbre initiale des spécifications conditionnelles hiérarchiques avec un nombre arbitraire de niveaux de hiérarchie. Nous obtenons des conditions suffisantes pour tester la confluence sur les termes clos d'un système hiérarchique. Par ailleurs, dans

une approche logico-algébrique, nous caractérisons le comportement d'une spécification conditionnelle pour assurer qu'elle soit fidèle par rapport aux booléens, c'est-à-dire que son support booléen soit isomorphe à l'ensemble $\{true,\ false\}$ et nous développons des conditions pour garantir cette fidélité.

Dans des travaux ultérieurs, Kaplan a proposé d'appréhender le problème de terminaison en considérant des systèmes simplifiants. Il s'agit de définir des règles de réécriture de telle sorte que le membre droit et la précondition d'une règle soient plus petits que le membre gauche correspondant, la comparaison étant établie à l'aide d'un ordre de simplification [55, 57]. Jouannaud et Waldmann ont par la suite, montré qu'il est suffisant d'utiliser un ordre de réduction pour mener à bien cette comparaison [53] et Dershowitz et Okada ont encore assoupli la méthode en considérant une famille d'ordres plus générale qui permet de définir les systèmes décroissants [19]. Les auteurs adoptent une approche algébrique et se donnent pour objectif de caractériser l'algèbre libre correspondant à la spécification considérée par une généralisation du théorème de complétude *à la Birkhoff* [55] en utilisant exclusivement la relation de réduction récursive. De ce fait, pout prouver la convergence des paires critiques, un algorithme d'unification bornée est nécessaire. Cette approche présente les inconvénients mentionnés page 26.

L'approche logico-algébrique a également été adoptée par Navarro et Orejas [79, 81]. Les auteurs considèrent des spécifications conditionnelles qui sont des extensions de la spécification des valeurs booléennes. Comme nous l'avons précédemment signalé, cette approche définit un système de déduction correct et complet par rapport à la classe $LogAlg_{SP}$ mais elle présente l'inconvénient de ne pas être opérationnelle. En effet, les auteurs ont besoin d'une spécification booléenne complète car ils utilisent des axiomes non-orientables tels que la commutativité du connecteur logique $\wedge$ et cette utilisation pose des problèmes de terminaison qui restent à l'heure actuelle encore ouverts. Navarro dans sa thèse, définit une relation de réécriture contextuelle qui comporte une branche complémentaire de réécriture. Pour réécrire un terme t, l'auteur utilise toutes les règles $p_i \Rightarrow g_i \rightarrow d_i$ telles que t contienne une instanciation de g_i et calcule le contexte de la branche complémentaire par rapport à la disjonction de la totalité des préconditions p_i. Cependant, l'auteur n'aborde pas les problèmes de terminaison inhérents à la définition de la branche complémentaire ni ceux posés par la relation de réécriture contextuelle.

Toujours dans une approche logico-algébrique mais en s'intéressant essentiellement à la confluence sur les termes clos, Ganzinger définit une relation de réécriture sous contexte pour laquelle l'applicabilité d'une règle peut être inférée à partir d'un certain contexte. Etant donné un ensemble d'équations inconditionnelles $C = \{c_1 = c'_1, \ldots,\ c_n = c'_n\}$, l'application de la règle $p \Rightarrow g \rightarrow d$, avec la substitution σ est autorisée si $\sigma p \in C{\downarrow}$. $C{\downarrow}$ dénote l'ensemble de toutes les assertions de convergence de C dérivées en utilisant les axiomes de réflexivité, de symétrie et de clôture par congruence. L'auteur utilise une technique de décomposition des paires critiques contextuelles par nécessité pour tester leur convergence. Cette technique utilise essentiellement des axiomes de recouvrement et des axiomes négatifs qui sont considérés comme des théorèmes inductifs de la spécification considérée, pouvant être prouvés par des méthodes de preuve indépendantes. Ganzinger présente une procédure de complétion dont il établit la correction par la méthode des ordres de preuve [34]. Cependant, l'auteur s'intéresse au modèle initial mais n'aborde pas les problèmes sémantiques de fidélité par rapport aux booléens. De plus, ses résultats supposent que la consistance et la complétude de la spécification conditionnelle sont acquises,

et ces propriétés sont par ailleurs difficiles à vérifier.

D'autres directions de recherche ont été explorées donnant lieu à différents travaux, que nous présentons brièvement :

Mohan et Srivas considèrent une application des systèmes de réécriture à la programmation fonctionnelle. Ils définissent un mécanisme de réécriture conditionnelle de termes appelé *Tue-Reduction* et l'utilisent pour la *Tue-Evaluation* de fonctions définies dans leur formalisme [78]. Ce formalisme consiste à considérer des règles conditionnelles pouvant contenir certains littéraux qui doivent être satisfaits et d'autres qui doivent être prouvés insatisfiables. L'objectif de leur travail est d'illustrer comment ce formalisme peut être utilisé pour construire des fonctions bien-définies.

Considérant l'approche où tous les modèles sont acceptés, Choquer et Bidoit travaillent sur les systèmes de réécriture simplifiants définis par Kaplan [57]. Ils montrent comment l'une des méthodes de preuve par récurrence, la méthode de preuve par *récurrence-surréduction*, peut être étendue de façon à permettre la preuve de propriétés conditionnelles, et ils illustrent comment ce formalisme de preuve peut être appliqué à la vérification de la correction du passage de paramètres dans une spécification paramétrée [4].

Toujours dans l'approche *algèbre libre*, Hussmann s'est intéressé au problème d'unification dans les théories conditionnelles soulevé par Kaplan [55] et il étend le mécanisme de résolution aux clauses de Horn avec égalité [49]. La méthode qu'il propose est plus générale que celle de Kaplan mais dans son approche, la propriété de confluence n'est pas traitée.

Brand et al [13] quant à eux, ont proposé les systèmes hiérarchiques, dont l'étude a été formalisée par la suite par Rémy [94] et par un certain nombre de ses collègues [96, 105], [95, 107, 10].

Bloom et Tindell [6], ont adapté des règles de réécriture inconditionnelles avec une axiomatisation du *if-then-else* dont les deux règles principales sont les suivantes :

$$if\text{–}then\text{–}else(true,\ true,\ false) = true,\ if\text{–}then\text{–}else(false,\ true,\ false) = false.$$

Par la suite, Guessarian et Messeguer [40], ont proposé une axiomatisation algébrique des opérations $if \ldots then \ldots else \ldots$ qui étend le résultat de complétude de Bloom et Tindell des algèbres discrètes aux algèbres continues, avec ou sans opérations additionnelles, et des termes finis aux termes infinis. Un système complet de preuve est alors établi avec des règles d'induction appropriées. Cependant, la notation conditionnelle $p \Rightarrow g \rightarrow d$ des règles de réécriture est plus adéquate puisqu'en considérant "$\Rightarrow$" comme un méta-symbole, elle permet d'utiliser des règles spécifiques pour manipuler la précondition p et en particulier, de traiter p par des axiomes du calcul des prédicats du premier ordre. Cette notation est par conséquent plus souple que celle préconisée dans [6] et [40] dans laquelle p perd son caractère spécifique de précondition puisqu'il est traité de la même manière que les autres termes de l'équation.

On peut également citer les travaux de Padawitz dans lesquels il s'intéresse aux *Log*-modèles [87, 84] et les travaux du groupe ADJ [39], de Drosten [23], de Goebel [36] ...

Plus récemment, Dershowitz et Plaisted ont proposé une application des systèmes de réécriture conditionnelle à la programmation logique et fonctionnelle [22].

Finalement, les systèmes de réécriture conditionnelle pouvant comporter dans les préconditions de leurs règles, aussi bien des inéquations que des équations, ont été abordés par

Kaplan [56]. L'auteur montre qu'en général, les spécifications de ce type n'admettent pas de modèle initial. Il introduit la notion de *modèle quasi-initial* comme une extension du concept de modèle initial et présente des résultats généraux d'existence et de complétude de ces modèles quasi-initiaux. De façon analogue à ses précédents travaux, il considère la classe des systèmes réducteurs et montre qu'il est possible de définir pour ces systèmes, une relation de réécriture minimale, dite *réécriture conditionnelle positive/négative*, construite comme une limite et décidable. Le principal résultat de l'auteur établit que lorsque cette relation de réécriture est confluente, l'algèbre de formes normales est un modèle quasi-initial. Enfin, un théorème fondé sur le calcul de paires critiques à la Knuth-Bendix est prouvé pour cette relation de réécriture positive/négative.

Par ailleurs, des travaux plus généraux car ayant un domaine d'application plus large que celui de la théorie conditionnelle, ont été récemment effectués par Claude et Hélène K irchner [61]. Ces travaux visent à formaliser le raisonnement équationnel avec contraintes. Ce type de raisonnement est défini de telle sorte que l'information contenue dans l'équation elle-même est exploitée, évitant ainsi autant que possible d'instancier et retardant le plus tard possible la résolution des équations. Les auteurs définissent les objets usuels tels que les termes, les équations ou encore les règles de réécriture comme des objets accompagnés d'un problème équationnel construit à partir de disjonctions et de conjonctions de problèmes élémentaires de la forme

$$u =_T v,\ u \neq_T v,\ u \ll_T v$$

où T représente une théorie quelconque. Les auteurs étudient le processus de complétion modulo la théorie T et expriment toutes les opérations de base de ce processus par la résolution d'un problème équationnel. Ils définissent la notion de réécriture contrainte et montrent qu'elle est assez puissante pour inclure strictement le formalisme de la réécriture conditionnelle. Les auteurs présentent une procédure de complétion fondée sur le raisonnement équationnel contraint et établissent sa correction par la méthode des ordres de preuve. Ils se proposent à l'avenir d'approfondir davantage la relation qui existe entre la complétion conditionnelle et la complétion contrainte.

7 Conclusion

Pour notre part, nous avons choisi de considérer la théorie conditionnelle et de travailler dans un premier temps dans un cadre hiérarchique dans le but d'étendre les travaux de Rémy à l'étude des systèmes hiérarchiques avec un nombre arbitraire de niveaux de hiérarchie. Nous nous sommes par la suite intéressée à la propriété de complétude suffisante hiérarchique car elle est une condition nécessaire pour établir l'équivalence entre la relation de réécriture hiérarchique et la réduction récursive. Le chapitre qui suit est consacré à l'étude de cette propriété dans un cadre plus général que le cadre hiérarchique. Nous appliquerons ensuite, dans le chapitre 3, nos résultats à la complétude suffisante hiérarchique.

Nous avons pu constater, dans la partie consacrée à la présentation des travaux dans le domaine de la théorie conditionnelle, qu'il est possible d'adopter diverses approches dépendant de critères spécifiques. L'un de ces critères concerne la classe des systèmes de réécriture utilisés pour pallier le problème de terminaison qui se pose lors de la justification des préconditions. Un second critère, lié à la forme des préconditions, caractérise les techniques de preuve de convergence des paires critiques. Le caractère évolutif de nos travaux de recherche nous a amenée à adopter principalement deux démarches dans le but d'établir un test de confluence sur les termes clos. En effet, dans un premier temps, nous avons choisi de considérer des systèmes de réécriture hiérarchique, dans lesquels la forme des préconditions est la suivante :

$$\wedge_{i=1}^{n}\ \epsilon_i\ q_i,$$

où pour tout i dans $[1 \ldots n]$, ϵ_i est soit le symbole de négation soit le mot vide et q_i est un atome, ne comportant aucun connecteur logique. La confluence sur les termes clos d'un système hiérarchique est alors assurée dès lors que l'on prouve l'équivalence des ensembles de formes normales contextuelles pour la totalité des paires critiques calculées au cours du test de confluence. Cette équivalence est vérifiée si les termes dans la conséquence de la paire critique sont identiques, ou si son contexte est insatisfiable. Nous avons par la suite rencontré une difficulté majeure essentiellement lorsque nous avons voulu effectuer ces preuves d'insatisfiabilité parce que ces preuves nécessitent de mettre en jeu des propriétés de prédicats que l'on ne sait pas toujours exprimer avec uniquement le système de réécriture considéré. Par ailleurs, d'un point de vue opérationnel, la solution que nous avons adoptée a consisté à utiliser le système booléen de Hsiang pour effectuer ces preuves d'insatisfiabilité. Ce système booléen, présenté dans [43], a la particularité de fournir un système canonique pour la logique propositionnelle et il calcule des formes normales uniques exprimées en fonction des connecteurs $\wedge$ et $+$ (OU exclusif). Le système de Hsiang s'est révélé toutefois insuffisant pour effectuer certains types de preuve et notamment, les preuves qui utilisent des prédicats d'ordre pour lesquels des propriétés telles que la transitivité ou la réflexivité constituent des informations essentielles pour pouvoir conclure que certains contextes sont insatisfiables. De telles propriétés nécessitent de manipuler en plus du système de Hsiang, un ensemble d'équations (les équations qui traduisent les propriétés d'associativité-commutativité et ne peuvant pas être orientées). L'utilisation dans REVEUR4 de ces propriétés comporte des procédures ad-hoc et elle n'est pas justifiée de manière rigoureuse.

Par la suite, il nous est apparu plus adéquat d'une part, de nous placer dans la classe des systèmes décroissants, dans lesquels, contrairement aux systèmes hiérarchiques, il est possible de définir un symbole de fonction de façon récursive, et d'autre part, de considérer

une forme algébrique de préconditions de règles, c'est-à-dire

$$\wedge_{i=1}^{n}\ s_i = t_i,\ t_i \in \{true,\ false\},\ \forall i \in [1 \ldots n].$$

Cette forme équationnelle est en réalité une seconde manière de coder les préconditions, mais elle présente l'avantage de faciliter les preuves d'insatisfiabilité. Ce choix est également motivé par le fait que nous disposons dans ce cadre, d'*heuristiques* relativement simples qui permettent de supprimer les équations conditionnelles triviales. D'autre part, pour cette seconde forme de préconditions, nous avons pu utiliser une stratégie de preuve dans les contextes plus efficace que dans nos précédents travaux.

Les premières études que nous avons effectuées sont présentées dans le chapitre 3 de cette thèse. C'est la raison pour laquelle ces travaux utilisent la première forme de préconditions tandis que les autres adoptent la seconde. Ces problèmes de preuves dans les contextes sont principalement développés dans le dernier chapitre du document. Nous établissons, dans ce chapitre, l'articulation entre le formalisme de la réécriture conditionnelle et la logique du premier ordre et nous illustrons la nécessité de combiner les deux concepts. Nous montrons par ailleurs que les approches adoptées par divers auteurs confortent cette nécessité.

Nous entamons cette thèse par l'étude de la complétude des spécifications conditionnelles. Un certain nombre de résultats sont connus dans le cadre classique, sans préconditions et il est à présent établi que la complétude est une propriété indécidable. Par contre, l'étude dans le contexte conditionnel a été très peu abordée. Nous visons dans le chapitre qui suit à établir une condition suffisante pour vérifier qu'une spécification définie à partir d'axiomes conditionnels est complète.

CHAPITRE 2

LA COMPLETUDE SUFFISANTE DES SPECIFICATIONS CONDITIONNELLES.

Chapitre 2

La complétude suffisante des spécifications conditionnelles.

1 Introduction

A partir de l'étude des types abstraits de données se dégage l'idée essentielle suivante : un type abstrait de données n'est pas seulement un ensemble d'objets mais aussi et principalement les opérations qui peuvent agir sur ces objets, et les propriétés qui lient ces opérations entre elles. Ces objets et ces opérations du type abstrait de données sont décrits par une spécification algébrique. Ainsi, le type abstrait constitue un cadre pour étudier des problèmes très concrets tels que la sémantique d'un programme ou celle d'un langage de programmation et le rôle de la spécification algébrique est de décrire précisément le problème considéré.

Traditionnellement, opérations et propriétés d'une spécification $SP = (S,\ F,\ E)$ sont partagées en deux parties. La première partie ou partie de base $\langle BF,\ BE\rangle$ contient respectivement l'ensemble C des *constructeurs* et les relations qui peuvent exister entre eux et la seconde contient des symboles de fonction appelés *opérations définies* de telle sorte que la spécification vérifie un principe de définition de la seconde partie par rapport à la première. Une propriété essentielle doit cependant être vérifiée par l'ensemble F des symboles de la spécification, à savoir que la définition de toute opération est complète par rapport aux constructeurs. Dans ce cas, la spécification SP est dite complète par rapport à l'ensemble C des constructeurs. La propriété de complétude équivaut à vérifier que toute forme normale d'un terme clos est un terme primitif, c'est-à-dire un terme construit uniquement à partir des constructeurs. Cette propriété est essentielle pour les spécifications algébriques car, combinée avec la propriété de *consistance*, elle assure la correction d'une instanciation de spécification paramétrée. Elle est également utile dans les aspects de preuve dans le sens où elle permet d'effectuer des preuves par induction (voir par exemple [62, 59, 52]) et également dans les preuves de confluence sur les termes clos (voir [11]).

La complétude d'une spécification n'est pas décidable en général. Cependant, un certain nombre de critères syntaxiques permet de vérifier cette propriété. La plupart de ces critères se basent sur des méthodes de réécriture mais imposent des restrictions quant à la structure des spécifications. Un travail conséquent a déjà été effectué dans le but d'étudier la complétude des spécifications algébriques inconditionnelles et des conditions suffisantes de vérification sont établies dans [41, 84, 48], [83, 17, 102], [90, 65, 66] pour certaines

classes de spécifications et plus récemment dans [59, 60] et [72, 71].

Par contre, dans le contexte conditionnel, aucune étude na va bien loin. Ceci est principalement dû à la complexité du cadre conditionnel et à l'indécidabilité du problème. Quelques idées sont suggérées par J-L. Rémy [94] fondées sur la construction de l'algèbre initiale d'une spécification partielle. Cette construction utilise des constantes d'erreurs et des prédicats de définition pour compléter la spécification. La méthode présente l'inconvénient d'être lourde car elle grossit considérablement la taille de la spécification. D'autre part, Zhang [105] a également établi des résultats de complétude principalement fondés sur la réécriture contextuelle et sur le calcul des formes normales. Son algorithme, implanté dans REVEUR4, nécessite cependant plus d'hypothèses que le nôtre, notamment la *bonne couverture* et la *hiérarchie* du système de réécriture. D'autre part, une phase importante de cet algorithme qui consiste à calculer des *ensembles tests*, est en général incorrecte du fait qu'elle ne tient pas compte du typage des opérateurs. Par la suite, Rémy a repris ses idées premières avec la collaboration d'Uhrig. Ces auteurs établissent un test de complétude suffisante des systèmes réducteurs, correspondant à des classes spécifiques de spécifications conditionnelles [104]. L'idée de considérer particulièrement la complétude suffisante par rapport à la partie booléenne est inspirée de leurs travaux. Toutefois, Rémy et Uhrig analysent le système de réécriture de manière statique et ce choix rend leur algorithme ineffectif pour certaines classes de spécifications. La méthode que nous établissons dans ce chapitre inclut strictement la leur. Elle présente de plus l'avantage d'être puissante du fait qu'elle se base sur la réécriture contextuelle et sur le raisonnement par cas.

Les conditions suffisantes que nous établissons sont effectives et nous proposons un algorithme pour les vérifier. L'idée principale sous-jacente à notre méthode est de combiner une analyse structurelle par le calcul d'un ensemble de termes qui représentent des *motifs*, et une analyse par cas par le biais d'un ensemble de conditions utilisées dans la réécriture. Cette analyse par cas permet de réduire toutes les instances d'un motif.

Une autre caractéristique de notre algorithme est qu'il n'impose pas de conditions trop restrictives quant aux systèmes de réécriture. Par exemple, nous ne traitons pas séparément les opérations utilisées dans les préconditions et celles définies par des règles conditionnelles, comme nous l'avons fait dans nos précédents travaux pour des systèmes hiérarchiques. Nous supposons simplement que les systèmes considérés sont *décroissants*. Toute règle dans un système décroissant a son membre gauche d'une part, plus grand que son membre droit selon un ordre de réduction et d'autre part, plus grand que sa précondition la comparaison étant cette fois établie par une extension de l'ordre de réduction qui possède la propriété de sous-terme strict. Cette propriété des systèmes conditionnels est assez naturelle et elle est de plus, satisfaite dans une majorité de systèmes utilisés en pratique. L'hypothèse de décroissance nous permet d'utiliser un argument inductif dans la preuve de correction de notre algorithme. De plus, structurer l'ensemble des symboles de fonctions en partitions est une opération systématique lorsqu'on définit des opérateurs à partir des constructeurs de la spécification et ne restreint pas la classe des systèmes utilisés.

Nous nous proposons d'illustrer, dans ce qui suit, le principe de notre méthode sur l'exemple des entiers naturels dont les constructeurs sont la constante **0** et l'opérateur ***succ*** :

$$pos(x) \rightarrow false,\ pos(succ(x)) \rightarrow true,\ f(0) \rightarrow 0,\ pos(x) = true \Rightarrow f(x) \rightarrow 0$$

Il est impossible de prouver la complétude suffisante simplement en examinant les règles. En effet, dans l'ensemble des règles conditionnelles, le cas $pos(x) = false$ semble avoir été omis. Il s'agit alors d'éclater le motif le plus général $f(x)$ en deux autres motifs plus spécialisés, $f(0)$ et $f(succ(x))$, définis à partir de l'ensemble des constructeurs. Le premier motif est traité par la règle inconditionnelle et le second par la règle conditionnelle puisque la précondition correspondante $pos(succ(x))$ s'évalue à *true* et ceci achève la preuve.

Ce chapitre est organisé de la façon suivante : le premier paragraphe formalise le cadre de travail et introduit les concepts de base. Il définit également le type de systèmes sur lesquels nous choisissons de travailler. Le second paragraphe définit la *convertibilité* et établit les conditions sous lesquelles cette propriété est équivalente à la *complétude suffisante*. La complétude suffisante est définie à partir de la congruence équationnelle et la convertibilité est liée au caractère opérationnel de la relation de réécriture. Le paragraphe suivant est consacré à l'étude de la convertibilité des systèmes de réécriture conditionnelle. Cette propriété étant en général indécidable, nous proposons une condition suffisante qui permet d'obtenir un algorithme de test pour certaines classes de systèmes de réécriture conditionnelle. Ce paragraphe comporte trois parties. L'ensemble de ces parties définit essentiellement les outils utilisés pour établir l'algorithme de test de la convertibilité à savoir, les notions de *système bien développé*, de *réécriture contextuelle* et de *schémas structurels*. Le quatrième paragraphe présente l'algorithme de test et sa preuve de correction. Le paragraphe qui lui succède présente un panorama d'exemples qui illustrent le fonctionnement de l'algorithme, sa portée d'utilisation et ses insuffisances. Dans cet objectif, nous résumons la situation en déterminant le type de spécifications conditionnelles, et plus précisément la forme des préconditions intervenant dans les définitions des opérateurs, pour lesquelles l'algorithme s'achève avec succès. Nous concluons dans le dernier paragraphe en envisageant un certain nombre d'extensions de ce travail permettant d'appréhender une classe plus large de spécifications algébriques.

2 Les concepts de base

Nous nous plaçons dans la classe des algèbres $\mathcal{A}$ dont le support booléen, soit A_{bool}, est constitué des seules valeurs ***true*** et ***false*** qui sont les interprétations des constantes booléennes *true* et *false* dans $\mathcal{A}$. Nous supposons de plus, que *true* et *false* sont des constantes distinctes et irréductibles dans tout système de réécriture R.

Définition 2.1 Conjonction d'équations booléennes *(CEB)*
On appelle conjonction d'équations booléennes (CEB) *une expression booléenne* $\vec{s}$ *de* $T(F,\ X)_{bool}$ *de la forme*

$$s_1 = s'_1\ \wedge \ldots \wedge\ s_n = s'_n$$

telle que pour tout $i = 1, \ldots,\ n,\ s'_i \in \{true,\ false\}$.

Nous considérons des équations conditionnelles de la forme $\vec{p} \Rightarrow g = d$ où $g,\ d$ sont des termes de $T(F,\ X)_s$ et telle que $\vec{p}$ est une CEB. Nous nous restreignons dans cette étude, à cette forme particulière de préconditions parce qu'il est, à l'heure actuelle, malaisé d'aborder le problème de la complétude suffisante dans un cadre plus général. En effet, la complétude suffisante nécessite de vérifier que toute instance close de précondition se réduit à *true* pour montrer qu'un terme clos est réductible. Ce test étant indécidable, nous adoptons cette forme de préconditions pour laquelle nous proposons des heuristiques permettant de mener à bien le test de réductibilité. La forme des CEB nous permet d'établir une condition suffisante pour tester le *recouvrement d'un ensemble de préconditions.*

Nous ne rappelons pas dans ce chapitre les définitions de congruence équationnelle, ni de relation de réécriture conditionnelle. Dans la suite, nous adopterons la convention :

- $\vec{p} =_E TT$ ssi pour tout indice $i,\ 1 \leq i \leq n,\ p_i =_E p'_i$ et
- $\vec{p} =_E FF$ ssi il existe au moins un indice i dans $[1 \ldots n]$ tel que, si $p'_i = true$ (resp $false$) alors $p_i =_E false$ (resp $true$).

Plus précisément, ceci revient à construire un enrichissement de la spécification booléenne SP_{bool} en ajoutant à l'ensemble F_{bool} les constantes booléennes $TT,\ FF$ qui sont des CEB particulières, tandis que E_{bool} est étendu par l'ajout des axiomes suivants où $\vec{p}$ désigne une CEB quelconque et où le second symbole d'égalité est assimilé à l'égalité dans l'ensemble des CEB.

1. $(true = true) = TT$
2. $(false = false) = TT$
3. $(true = false) = FF$
4. $(false = true) = FF$
5. $\vec{p} \wedge \vec{p} = \vec{p}$
6. $\vec{p} \wedge TT = \vec{p}$
7. $\vec{p} \wedge FF = FF$

et considérons désormais que tout système de réécriture est défini à partir de la signature booléenne enrichie et qu'il comporte les équations ci-dessus orientées de gauche à droite.

Les objets d'un type abstrait algébrique sont définis à partir de symboles de fonction particuliers appelés constructeurs du type. Il est alors naturel de formaliser la structure d'une spécification en scindant l'ensemble des opérateurs de la façon suivante:

Définition 2.2 *Soit $BF \oplus DF$ une partition d'un ensemble de symboles de fonction.*

- *Une BF-équation est une équation $\vec{p} \Rightarrow g = d$ dont les termes $\vec{p}$, g et d sont dans $T(BF, X)$ et telle que les variables de $\vec{p}$ apparaissent toutes soit dans g soit dans d.*
- *Une BF-règle est une BF-équation orientée $\vec{p} \Rightarrow g \rightarrow d$ telle que $Var(g)$ contient $Var(d)$ et $Var(\vec{p})$.*
- *Une DF-équation est une équation $\vec{p} \Rightarrow g = d$ dont l'un des membres g ou d est un terme de $T(F, X) \ominus T(BF, X)$ et telle que les variables de $\vec{p}$ apparaissent toutes soit dans g soit dans d.*
- *Une DF-règle est une DF-équation orientée $\vec{p} \Rightarrow g \rightarrow d$ dont le membre gauche est un terme de $T(F, X) \ominus T(BF, X)$ et telle que $Var(g)$ contient $Var(d)$ et $Var(\vec{p})$.*

Remarque: Notons que cette définition interdit de considérer des équations dont la conséquence est formée uniquement d'opérateurs de BF et dont la précondition est dans $T(F, X) \ominus T(BF, X)$.
Considérons par exemple la spécification du type abstrait $\boldsymbol{Ens(Elem)}$ avec pour constructeurs les symboles $\emptyset$, $\{\}$ et $\cup$ et supposons que nous ayons défini l'opérateur d'appartenance $\in$. La relation suivante ne peut faire partie de l'ensemble des équations de la spécification:

$$(**) \quad x \in u \Rightarrow \{x\} \cup u = u.$$

La raison en est que ce type d'équations engendre des problèmes de récursivité que nous voulons éviter. En effet, si nous considérons la relation $(**)$, celle-ci définit une relation entre les constructeurs du type à l'aide d'opérateurs définis. Ces derniers étant eux-mêmes définis à partir des constructeurs, l'utilisation de ce type de relations devient complexe et nécessite une étude plus approfondie, que nous n'abordons pas dans cette thèse.

Définition 2.3 Spécifications et systèmes structurés
Soit $F = BF \oplus DF$ un ensemble de symboles de fonctions tel que BF contient un ensemble C de constructeurs. Une spécification (S, F, E) est structurée de base (S, BF, BE) ssi $E = BE \oplus DE$ où BE est un ensemble de BF-équations et DE un ensemble de DF-équations.
Un système (S, F, R) est structuré de base (S, BF, BR) si $R = BR \oplus DR$ où BR est un ensemble de BF-règles et DR un ensemble de DF-règles.

DF est l'ensemble des *opérations définies.* Nous appelons *termes de base* les termes définis à partir des opérateurs de BF; les termes primitifs sont ceux construits uniquement à partir des constructeurs (voir page 19).
Nous supposons dans toute cette étude que les spécifications sont structurées.

Notons que nous n'interdisons pas l'utilisation de relations entre les opérateurs définis. Si

nous définissons les opérateurs ***ordered*** et ***insert*** sur le type ***liste*** muni des constructeurs [ ] et ***cons***, une relation entre ces deux opérateurs peut s'écrire:

$$ordered(insert(x,\ u)) = ordered(u)$$

Exemple 2.1 *La spécification conditionnelle* $(S,\ F,\ E)$ *suivante est une spécification structurée de base* $(S,\ BF,\ BE)$ *où* $F = BF \oplus DF,\ E = BE \oplus DE$: $S = \{bool,\ elem,\ ens\}$;

Les constructeurs:
$C = \{true,\ false :\rightarrow bool,$
$0 :\rightarrow elem,\ succ : elem \rightarrow elem,$
$\emptyset :\rightarrow ens,\ aj : elem \times ens \rightarrow ens,$

Les opérateurs de base:
$BF = C \oplus \{eg? : elem \times elem \rightarrow bool\},$

Les opérations définies:
$DF = \{Pres? : elem \times ens \rightarrow bool,$
$inclus? : ens \times ens \rightarrow bool,$
$ret : elem \times ens \rightarrow ens\},$

Les relations entre les constructeurs:
$BE = \{eg?(0,\ 0) = true,$
$eg?(0,\ succ(x)) = false,$
$eg?(succ(x),\ 0) = false,$
$eg?(succ(x),\ succ(y)) = eg?(x,\ y),$
$aj(b,\ aj(a,\ e)) = aj(a,\ aj(b,\ e)),$
$aj(a,\ aj(a,\ e)) = aj(a,\ e)\},$

Les définitions:
Définition de *Pres?*:
$DE = \{Pres?(a,\ \emptyset) = false,$
$eg?(a,\ b) = true \Rightarrow Pres?(b,\ aj(a,\ e)) = true,$
$eg?(a,\ b) = false \Rightarrow Pres?(b,\ aj(a,\ e)) = Pres?(b,\ e),$

Définition de *inclus?*:
$inclus?(\emptyset,\ y) = true,$
$Pres?(a,\ e') = true \Rightarrow inclus?(aj(a,\ e),\ e') = inclus?(e,\ e'),$
$Pres?(a,\ e') = false \Rightarrow inclus?(aj(a,\ e),\ e') = false,$

Définition de *ret*:
$ret(a,\ \emptyset) = \emptyset,$
$eg?(a,\ b) = true \Rightarrow ret(b,\ aj(a,\ e)) = aj(a,\ ret(b,\ e)),$
$eg?(a,\ b) = false \Rightarrow ret(b,\ aj(a,\ e)) = ret(b,\ e)\}.$

Remarque : La définition 2.2 étend de façon naturelle la notion de systèmes qui préservent les termes de base tels qu'ils ont été définis pas Kapur&al [60] au cadre conditionnel. Nous dirons, en effet à l'instar de ces auteurs, qu'un système de réécriture conditionnelle préserve les termes de base si toute règle de ce système dont le membre gauche est défini à partir de l'ensemble de base BF, a son membre droit également dans l'ensemble $T(BF, X)$. Ceci entraîne en particulier que la clôture de $T(BF)$ par $\rightarrow_R$ est égale à $T(BF)$. Un système structuré est un système qui préserve les termes de base comme le montre la proposition qui suit :

Proposition 2.1 *Si R est structuré de base BR alors la clôture de $T(BF)$ par la relation $\rightarrow_R$ est égale à $T(BF)$.*

Preuve par l'absurde : supposons qu'il existe un terme clos t dans $T(BF)$ réductible par une règle $\vec{p} \Rightarrow g \rightarrow d$ de $R \ominus BR$; il existe donc une occurrence u de t, une substitution close σ telles que $t_{/u} = \sigma g$, donc t contient un sous-terme t' qui appartient à $T(F) \ominus T(BF)$ ce qui contredit l'hypothèse que $t \in T(BF)$. □

Les concepts généraux de convertibilité et de complétude suffisante peuvent être trouvés dans le chapitre 1 (paragraphe 4).

Puisque dans la suite, nous considérons uniquement des systèmes de réécriture structurés, nous pouvons par conséquent affirmer le résultat suivant :

Proposition 2.2 *Soit $SP = (S, F, E)$ une spécification conditionnelle structurée de base $BSP = (S, BF, BE)$; supposons que E définisse un système de réécriture R structuré de base BR et que BR soit convertible par rapport à C.*
Si $\rightarrow_R$ est confluente sur les termes clos de $T(F)$, la complétude suffisante de SP par rapport à C est équivalente à la convertibilité de R par rapport à C.

Preuve :
$\Longleftarrow$: la convertibilité de R entraîne la complétude suffisante de SP puisque $\rightarrow_R^*$ est contenue dans $=_E$.

$\Longrightarrow$: par la proposition 2.1, la clôture de $T(BF)$ par $\rightarrow_R$ est égale à $T(BF)$. Par la proposition 1.3, nous pouvons donc affirmer que, pour tout terme clos t dans $T(F)$, il existe un terme clos t_0 dans $T(BF)$ tel que $t \rightarrow_R^* t_0$.
Puisque BR est convertible par rapport à C, il existe un terme clos $\bar{t}_0$ dans $T(C)$ tel que $t_0 \rightarrow_R^* \bar{t}_0$. Par conséquent, $t \rightarrow_R^* t_0 \rightarrow_R^* \bar{t}_0$. □

La suite de ce chapitre est consacrée à l'étude des conditions syntaxiques suffisantes permettant de vérifier la convertibilité d'un système de réécriture conditionnelle. Nous nous intéressons à la convertibilité, sachant qu'elle est équivalente à la complétude suffisante si le système de réécriture est structuré et confluent sur les termes clos. Une question qu'on peut se poser alors est la suivante : le système de réécriture obtenu après complétion d'une spécification algébrique structurée est-il lui aussi structuré? Il est probablement nécessaire pour cela, de définir un ordre tel que chaque symbole dans C soit plus petit qu'un opérateur de BF lui-même plus petit qu'un opérateur quelconque de $F \ominus BF$.

Pour pallier les problèmes de terminaison, nous choisissons de travailler sur des systèmes de réécriture décroissants. Ce type de systèmes est une généralisation des systèmes réducteurs, qui ont été largement étudiés dans le cadre conditionnel et qui sont adaptés pour diverses applications. Parmi les travaux portant sur la classe des systèmes de

réécriture réducteurs ou décroissants, nous pouvons citer [18, 53], [57, 20].

Dans ce qui suit, toute spécification conditionnelle est supposée contenir la spécification des booléens dont les seuls constructeurs sont les constantes *true* et $false$. L'ensemble des constructeurs de la sorte booléenne est noté $\mathcal{C}_{bool}$ $(= \{true,\ false\})$.

3 Une condition suffisante pour tester la convertibilité

Ce paragraphe est consacré à l'étude de la propriété de convertibilité. La vérification que toute instance close d'un terme est réductible étant une opération coûteuse et/ou ineffective, nous établissons plutôt une condition suffisante qui permet de tester la complétude suffisante dans les systèmes de réécriture structurés et décroissants. Cette condition est caractérisée à partir de trois notions présentées dans chacune des trois parties qui suivent. La première notion est celle de *système bien développé*;

3.1 Systèmes bien développés

Définition 2.4 Système bien développé

Soit C un ensemble de constructeurs; R est bien développé ssi tout terme clos de la forme

$$f(t_1, \ldots,\ t_n),\ t_i \in T(C)_{s_i},\ \forall i,\ 1 \leq i \leq n$$

est réductible par $\rightarrow_R$.

Une substitution est primitive si elle substitue les variables d'un terme par des termes primitifs, c'est-à-dire des termes construits uniquement à partir des constructeurs. Un système bien développé est alors tel que toute instanciation close primitive d'un terme de la forme $f(x_1, \ldots,\ x_n)$ est réductible par $\rightarrow_R$.

Théorème 2.1 *Soit R un système de réécriture conditionnelle structuré de base BR; supposons que BR soit convertible par rapport à C et que $\rightarrow_R$ soit nœthérienne; si R est bien développé, R est aussi convertible par rapport à C.*

Preuve : Soient t un terme clos de $T(F) \ominus T(BF)$ et $\bar{t}$ sa forme normale. $\bar{t}$ existe puisque $\rightarrow_R$ est nœthérienne. Par la proposition 2.3, il suffit de montrer que $\bar{t}$ est dans $T(C)$. Par l'absurde, supposons que $\bar{t} \notin T(C)$; puisque BR est convertible par rapport aux constructeurs, il existe un sous-terme t' de $\bar{t}$ de la forme $f(t_1, \ldots,\ t_n)$ avec f dans $F \ominus BF$ et pour tout i, $1 \leq i \leq n$, t_i est dans $T(C)$. Puisque R est bien développé, il existe une règle $\vec{p} \Rightarrow g \rightarrow d$ dans R et une substitution close σ telles que $t' = \sigma g$ et $\sigma\vec{p} \rightarrow_R^* TT$. t' est donc réductible et $\bar{t}$ également; ce qui contredit l'hypothèse que $\bar{t}$ est une forme normale. □

3.2 Réécriture contextuelle et ensemble recouvert de CEB

La difficulté du test de la réductibilité d'un terme réside dans le test $\sigma\vec{p} \rightarrow_R^* TT$ qui est indécidable. Comme nous le précisons dans la suite, ce test est pratiqué par le biais d'une condition plus forte (dans le sens où elle est suffisante mais non nécessaire) qui est la propriété de recouvrement. Ce test de recouvrement est utilisé pour tester la réductibilité par cas d'un terme de $T(F,\ X)$. Pour formaliser ces différentes notions, nous avons besoin de définir la relation de réécriture contextuelle.

Définition 2.5 Validité opérationnelle

Un ensemble de CEB closes, sans variables,

$$Cond = \{\vec{c_1}', \ldots, \vec{c_n}', \vec{c_i}' \in T(F)_{bool}, \forall i = 1, \ldots, n\}$$

est opérationnellement valide dans R ssi

$$\exists k \in [1 \ldots n] \text{ tel que } \vec{c_k}' \rightarrow_R^* TT.$$

Définition 2.6 Ensemble recouvert de *CEB*

Soit $Cond = \{\vec{c}_1, \ldots, \vec{c}_m\}$ un ensemble de CEB ; Cond est recouvert ssi la formule $\overline{Cond}$ est une tautologie dans le calcul propositionnel sur l'alphabet $\overline{At}$, où $\overline{Cond}$ et $\overline{At}$ sont définis de la façon suivante :
soit At l'ensemble des atomes booléens apparaissant dans Cond et soit $\overline{At}$ un alphabet en bijection avec At. Soit $Prop(\overline{At})$ le calcul propositionnel défini sur l'alphabet $\overline{At}$ et soit $\tilde{c}_1, \ldots, \tilde{c}_m$, $\overline{Cond}$ les formules suivantes dans $Prop(\overline{At})$

- $\overline{Cond} = \tilde{c}_1 \vee \ldots \vee \tilde{c}_m$
- *et pour tout i dans $[1 \ldots m]$, tel que $\vec{c}_i$ soit de la forme $\wedge_{i=1}^{k}\ t_i = t'_i$,*

  $$\tilde{c}_i \text{ est l'expression } \epsilon_1 \tilde{t}_1 \wedge \ldots \wedge \epsilon_k \tilde{t}_k$$

  où, pour tout j dans $[1 \ldots k]$, $\tilde{t}_j$ est le symbole correspondant à l'atome t_j et ϵ_j est le mot vide si t'_j est true et le symbole de négation $\neg$ si t'_j est false.

La notion de recouvrement d'un ensemble *Cond* de *CEB* implique que *Cond* est suffisamment complet par rapport à l'ensemble $\{true, false\}$ qui constitue l'ensemble C_{bool} des constructeurs de la sorte booléenne. Etant donné que les préconditions sont de sorte booléenne, la démarche que nous adoptons dans un premier temps, est de définir en parallèle la notion de convertibilité par rapport à un ensemble quelconque de constructeurs et par rapport à C_{bool}. Les deux notions de convertibilité seront fusionnées par la suite.

Lemme 2.1 *Soient $At = \{a_1, \ldots, a_n\}$ un ensemble d'atomes booléens et $Cond = \{\vec{c}_1, \ldots, \vec{c}_m\}$ un ensemble de CEB défini sur At ; soit σ une substitution close et soit $Cond_\sigma$ l'ensemble de CEB obtenu en instanciant toutes les composantes de Cond par σ,*

$$Cond_\sigma = \{\sigma\vec{c}_1, \ldots, \sigma\vec{c}_m\};$$

supposons que At vérifie la propriété suivante :

$$(**)\ \forall i \in [1 \ldots n],\ \exists t \in \{true, false\} \text{ tel que } \sigma a_i \rightarrow_R^* t.$$

si Cond est recouvert, $Cond_\sigma$ est opérationnellement valide.

Preuve : il s'agit de montrer qu'il existe un indice k dans $[1 \ldots m]$ tel que $\sigma\vec{c}_k \rightarrow_R^* TT$. La preuve est effectuée par l'absurde. Par hypothèse (à cause de la propriété $(**)$), nous avons, pour tout indice k dans $[1 \ldots m]$, $\sigma\vec{c}_k \rightarrow_R^* TT$ ou $\sigma\vec{c}_k \rightarrow_R^* FF$. Supposons que pour tout k dans $[1 \ldots m]$, $\sigma\vec{c}_k \rightarrow_R^* FF$.

Pour tout i dans $[1 \ldots n]$, soit b_i un terme dans $\{true, false\}$ tel que $\sigma a_i \rightarrow_R^* b_i$.

Soit $B = \{\overline{true}, \overline{false}\}$ l'algèbre des booléens et soit μ la valuation de At dans $\{true, false\}$ définie par

$$\forall i = 1, \ldots, n,\ \mu(\tilde{a}_i) = \bar{b}_i$$

Alors, $\sigma\vec{c}_k$ se réduit à FF signifie exactement que $\mu(\vec{c}_k)$ est la valeur $\overline{false}$. Puisque ceci est vérifié pour tout k, ceci entraîne que $\mu(\overline{Cond}) = \overline{false}$ et mène à une contradiction puisque $\overline{Cond}$ est par hypothèse, une tautologie. □

La relation de réécriture contextuelle est définie sur des termes contextuels ou c-termes :

Définition 2.7 Terme contextuel
Un terme contextuel ou c-terme $(\vec{c} :: t)$ *est une paire de termes telle que* t *est de sorte quelconque et* $\vec{c}$ *est une* CEB.

Définition 2.8 Réécriture contextuelle de termes, $\xrightarrow{c}_R$
Soit R *un système de réécriture conditionnelle. La relation de réécriture contextuelle de termes est notée* $\xrightarrow{c}_R$ *et définie par :*
soit $\vec{c}_1$ *de la forme* $\wedge_{j=1}^{n}\ c_{1,j} = c'_{1,j},\ n > 0$,

$(\vec{c}_1 :: t_1) \xrightarrow{c}_R (\vec{c}_2 :: t_2)$ *ssi il existe une règle de réécriture* $\vec{p} \Rightarrow g \rightarrow d$ *dans* R, *une substitution* σ *et*

- *soit une occurrence* u *de* t_1 *telle que* $t_{1/u} = \sigma g$, *alors*

$$\vec{c}_2 = \vec{c}_1 \ \wedge\ \sigma\vec{p},\ t_2 = t_1[u \leftarrow \sigma d].$$

- *soit un indice* i *dans* $[1 \ldots n]$ *et une occurrence* ω *de* $c_{1,i}$ *tels que* $c_{1,i/\omega} = \sigma g$, *alors*

$$\vec{c}_2 = (\wedge_{j=1}^{i-1}\ c_{1,j} = c'_{1,j}) \wedge (c_{1,i}[\omega \leftarrow \sigma d] = c'_{1,i}) \wedge (\wedge_{j=i+1}^{n}\ c_{1,j} = c'_{1,j}) \wedge \sigma\vec{p}$$

$$\text{et } t_2 = t_1.$$

Définition 2.9 Réductibilité par cas
Un terme t *est réductible par cas dans* R *ssi il existe un ensemble* $\{(\vec{c}_i :: t_i)\}_{i\in I}$ *de termes contextuels tel que*

- $\forall i \in I,\ (TT :: t) \xrightarrow{c}_R (\vec{c}_i :: t_i)$ *et*
- *il existe un ensemble recouvert* $\{\vec{c}'_i\}_{i\in I}$ *tel que*

$$\forall i \in I,\ \vec{c}_i \rightarrow_R^* \vec{c}'_i\ .$$

En particulier, la réductibilité classique, inconditionnelle est une réductibilité par cas, si l'on admet l'équivalence de $s \rightarrow_R t$ et de $(TT :: s) \xrightarrow{c}_R (TT :: t)$.

Remarque : Cette définition de la réductibilité par cas se base sur un seul aspect de la réécriture contextuelle, celui concernant la réécriture du terme contextuel $(TT :: t)$ dans sa partie terme. En effet, pour réduire les contextes obtenus à leur forme normale, la relation que nous utilisons est la réduction récursive.

Proposition 2.3 *Soit* t *un terme et soit* $(\vec{c}_1 :: t_1), \ldots, (\vec{c}_m :: t_m)$ *un ensemble de réductions contextuelles (en une étape) de* $(TT :: t)$ *; soit* σ *une substitution close. S'il existe un indice* i *dans* $[1 \ldots m]$ *satisfaisant* $\sigma\vec{c}_i \rightarrow_R^* TT$, σt *est réductible.*

Preuve : Elle est triviale puisque si $\sigma\vec{c_i} \rightarrow_R^* TT$, $\sigma t \rightarrow_R^* \sigma t_i$. Par conséquent, σt est réductible. □

La méthode que nous développons dans ce qui suit est inspirée d'une part, des travaux détaillés et élaborés dans un cadre inconditionnel par Kounalis dans sa thèse [66] et d'autre part, des investigations de Zhang dans un contexte conditionnel [105].

La combinaison de la proposition 2.3 et du lemme 2.1 signifie en particulier que pour vérifier qu'un opérateur f de $F \ominus BF$ est complètement défini, il suffit de vérifier que le terme $f(x_1, \ldots, x_n)$ est réductible par cas. Le test de réductibilité par cas est une condition suffisante pour montrer que le système R est convertible. En particulier, R doit aussi être convertible par rapport à l'ensemble C_{bool} c'est-à-dire à l'ensemble $\{true, false\}$. De par la définition de la validité opérationnelle, la convertibilité de R par rapport à C_{bool} entraîne la validité opérationnelle de tout ensemble de CEB. Cette propriété de convertibilité par rapport à l'ensemble des constructeurs booléens est une autre formulation de la propriété (**) supposée sur l'ensemble At des atomes qui constituent les CEB. Il devient alors paradoxal de garder cette hypothèse pour vérifier la convertibilité de tout le système considéré. Par la suite, nous donnons un résultat de convertibilité de R par rapport à C dans lequel nous ne faisons a priori aucune hypothèse sur l'ensemble d'atomes qui forment les préconditions de R.

3.3 Arbres de schémas structurels

Dans ce paragraphe, nous présentons une méthode de vérification de la convertibilité. Cette méthode est inspirée des travaux de Kounalis [65] et son idée principale consiste à construire, pour tout symbole d'opération f dans $F \ominus BF$, un arbre de motifs qui contient des termes de la forme $f(\omega_1, \ldots, \omega_n)$ avec $\omega_i \in T(C, X)_{s_i}$, pour tout i, et à tester pour chacun de ces termes la réductibilité par cas de $f(\omega_1, \ldots, \omega_n)$.

L'idée de considérer des motifs a été dans un premier temps proposée par Dershowitz [17]. Un motif est un terme dont la racine est l'opérateur f et dont les arguments sont des termes de $T(C, X)$, dont la profondeur est strictement inférieure à la profondeur maximale des règles de la spécification. La méthode de Dershowitz nécessite une étude exhaustive de l'ensemble de motifs. Kounalis dans ses travaux [66], améliore la méthode de Dershowitz en se basant sur la notion d'arbre de motifs plutôt que sur celle d'ensemble. Ce choix a pour effet de supprimer des motifs inutiles et la méthode est ainsi rendue plus efficace.

La racine de l'arbre de motifs est étiquetée par le terme $f(x_1, \ldots, x_n)$. Comme plusieurs arbres peuvent être engendrés à partir de ce terme, il s'agit de construire celui qui capture la structure du système R. Pour cela, nous utilisons les notions suivantes :

Définition 2.10 Schéma structurel

On appelle schéma structurel de sorte s_i l'ensemble des termes

$$Sh_{s_i} = \{g(x_1, \ldots, x_n),\ g \in C_{\omega, s_i}\}$$

où $(x_1, \ldots, x_n)$ est une liste (éventuellement vide si g est un symbole de constante) de variables toutes distinctes de sortes appropriées et où ω est la suite de ces sortes.

Exemple 2.2 *Si $C = \{0 :\rightarrow int,\ pred,\ succ : int \rightarrow int\}$; alors*

$$Sh_{int} = \{0,\ pred(x),\ succ(x)\}$$

Si $C = \{true,\ false : \rightarrow bool\}$;

$$Sh_{bool} = \{true,\ false\}.$$

Si $C = \{0 : \rightarrow nat,\ succ : nat \rightarrow nat,$
$[\,] : \rightarrow list,\ cons : nat \times list \rightarrow list\}$; *alors*

$$Sh_{nat} = \{0,\ succ(x)\},\ Sh_{list} = \{[\,],\ cons(x,\ u)\}.$$

◇

Définition 2.11 Motifs

Soit t un terme et soit u une occurrence de t telle que $t_{/u} = x \in X_s$; t_u^c *est le terme obtenu en remplaçant dans t la variable x par un schéma structurel c de* Sh_s *(après avoir renommé les variables).* t_u^c *est un motif de t en u.*
L'ensemble $\{t_u^c,\ c \in Sh_s\}$ *de motifs de t en u est noté* $\mathbf{Mot(t,\ u)}$, *l'opération de transformation de t en* $Mot(t,\ u)$ *est appelée greffe à l'occurrence u. Evidemment,* $Mot(t,\ u)$ *et* Sh_s *ont le même cardinal.*

Exemple 2.3 *Si* $BF = \{0 : \rightarrow int,\ succ : int \rightarrow int\}$ *et* $C = BF$;
$DF = \{+ : int \times int \rightarrow int\}$,

$Sh_{int} = \{0,\ succ(x)\}$ *et soit* $t = succ(x_1) + succ(succ(x_2))$, *alors*

$$Mot(t,\ 11) = \{succ(0) + succ(succ(x_2)),\ succ(succ(x_3)) + succ(succ(x_2))\}.$$

Si $BF = \{0 : \rightarrow nat,\ succ : nat \rightarrow nat,$
$empty\text{–}stack : \rightarrow stack,\ push : nat \times stack \rightarrow stack\}$, *et* $C = BF$;
$F \ominus BF = \{top : stack \rightarrow int\}$,

$Sh_{nat} = \{0,\ succ(x)\}$, $Sh_{stack} = \{empty\text{–}stack,\ push(x,\ u)\}$;
soit $t = top(push(x,\ u))$,

$$Mot(t,\ 11) = \{top(push(0,\ u)),\ top(push(succ(x_1),\ u))\},$$

$$Mot(t,\ 12) = \{top(push(x,\ empty\text{–}stack)),\ top(push(x,\ push(x_2,\ u_2)))\}.$$

◇

On note $Def_R(f)$ l'ensemble des règles de R dont le membre gauche est de la forme $f(\omega_1, \ldots, \omega_n)$ avec $\omega_i \in T(C,\ X)_{s_i}$. Intuitivement, c'est cet ensemble qui définit l'opérateur f.

Définition 2.12 Occurrences définies

Soit $Occ(f)$ *l'ensemble des occurrences définies de f.* $Occ(f)$ *est défini comme suit :*
$Occ(f) = \{u$ *tel qu'il existe* $\vec{p} \Rightarrow g \rightarrow d \in Def_R(f)$ *et* $g_{/u} \in F$ *ou*
$g_{/u}$ *est une occurrence de variable non linéaire dans* $g\}$.

Exemple 2.4 *Soit* $R = \{ordered([\,]) \rightarrow true,\ ordered(cons(x,\ [\,])) \rightarrow true,$
$x \leq y = true \Rightarrow ordered(cons(x,\ cons(y,\ u))) \rightarrow ordered(cons(y,\ u))$,

$$ordered(insert(x,\ u)) \rightarrow ordered(u)\}$$

$Def_R(ordered)$ est constitué des trois premières règles de R et

$$Occ(ordered) = \{\epsilon,\ 1,\ 12\}.$$

$$R = \{f(x,\ y,\ x) \rightarrow y,\ f(x,\ x,\ y) \rightarrow y,\ f(y,\ x,\ x) \rightarrow y\}$$

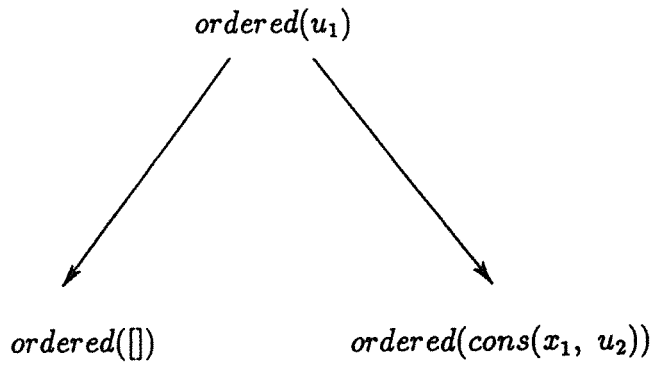
$$Def_R(f) = R \text{ et } Occ(f) = \{\epsilon,\ 1,\ 2,\ 3\}.\ \diamond$$

A partir d'un nœud quelconque de l'arbre étiqueté par le terme t de la forme $f(\omega_1, \ldots, \omega_n)$, avec $\omega_i \in T(C,\ X)_{s_i}$ pour tout $i = 1, \ldots,\ n$, on construit les fils de ce nœud en choisissant une occurrence u de t dans $Var(t) \cap Occ(f)$ et en opérant une greffe à cette occurrence. Ainsi, chaque fils est étiqueté par un élément de $Mot(t,\ u)$. Dans ce cas, on dit que t est *décomposable.* La hauteur de l'arbre de motifs est bornée car nous supposons que les règles de R ayant le symbole de fonction f en tête est en nombre fini. Ceci entraîne que l'ensemble $Occ(f)$ est fini. D'autre part, les greffes consécutives dans une même branche de l'arbre s'effectuent à des occurrences de termes de plus en plus profondes. On peut alors affirmer que l'ensemble $Var(t) \cap Occ(f)$ diminue de taille au fur et à mesure de la construction de l'arbre. Par conséquent, la hauteur de celui-ci est bornée.

Si un terme t est réductible par cas, nous pouvons affirmer qu'il ne sert à rien de développer l'arbre à partir du nœud étiqueté par ce terme, puisque par la proposition 2.4, toute instance close de ce terme est réductible. Par conséquent, nous pouvons décrire la construction de l'arbre de la façon suivante : à partir de l'arbre initialement constitué de la racine $f(x_1, \ldots,\ x_n)$, on vérifie la convertibilité en testant la réductibilité par cas de $f(x_1, \ldots,\ x_n)$. Si le terme $f(x_1, \ldots,\ x_n)$ n'est pas réductible par cas, on construit à chaque étape les fils de l'arbre en choisissant une occurrence de variable du nœud qui est aussi contenue dans $Occ(f)$, et en effectuant une opération de greffe sur le nœud à cette occurrence. L'arbre est totalement construit si chacun de ses fils soit est réductible par cas, soit n'est pas décomposable. Le calcul de $Occ(f)$ permet de prendre en compte la structure de R.

Notons en particulier qu'un terme clos n'est pas décomposable.

Exemple 2.5 *Considérons le système de l'exemple précédent; la méthode de construction d'un arbre de motifs produit l'arbre suivant :*



4 L'algorithme et sa preuve de correction

4.1 L'algorithme de convertibilité

Supposons dans toute cette partie les hypothèses suivantes :
$(S,\ F,\ R)$ est un système de réécriture structuré de base $(S,\ BF,\ BR)$, décroissant et tel que BR est convertible par rapport à C. Supposons de plus que la sorte booléenne ait pour seuls constructeurs les constantes $true$ et $false$.

R est convertible par rapport à C si tous les opérateurs de $F \ominus BF$ sont complètement définis par rapport à C.

Nous formulons l'algorithme de complétude de définition d'un opérateur f de $F \ominus BF$ par des règles d'inférence. Ces règles utilisent la structure de données suivante :

$$(Candidats,\ TC),\ \text{telle que}$$

- *Candidats* est un ensemble de termes. Ce sont les motifs de l'arbre construits au fur et à mesure que l'on effectue des greffes et desquels l'algorithme teste la réductibilité par cas.
- TC est un ensemble de termes contextuels. Si TC n'est pas vide à la fin du test de convertibilité, TC est constitué de c-termes $\{(c_i :: t_i)\}_{i \in I}$ non réductibles par cas. L'algorithme suggère alors d'examiner avec plus de soin la définition de l'opérateur f en ces termes et d'ajouter éventuellement des règles de la forme $\{c_i \Rightarrow t_i \rightarrow d_i\}_{i \in I}$.

Bien sûr, R est une composante globale de toutes les procédures qui suivent. Cependant, il ne figure pas dans les règles d'inférence puisqu'il ne varie pas au cours de ces procédures.

Soit f un symbole de fonction de $F \ominus BF$. L'algorithme de convertibilité dispose en entrée du système R et du symbole de fonction f. Le résultat est l'ensemble TC. Le test de complétude de définition de f se traduit par les règles d'inférence suivantes :

$$1) \quad Comp_1 \quad \frac{(\emptyset,\ TC)}{\text{arrêt}}$$

L'arrêt se produit lorsque l'ensemble des termes candidats est vide.
Si TC est vide, toute feuille de l'arbre de décomposition est réductible par cas. Par conséquent, le symbole d'opérateur f est complètement défini.

$$2) \quad Comp_2 \quad \frac{(Candidats \cup \{t\},\ TC)}{(Candidats,\ TC)} \ si\ t \text{ est réductible par cas}$$

Si le terme t est réductible par cas, on teste la réductibilité par cas des autres feuilles de l'arbre.

$$3) \quad Comp_3 \quad \frac{(Candidats \cup \{t\},\ TC)}{(Candidats \cup Mot(t,\ u),\ TC)} \ si \begin{cases} t \text{ n'est pas réductible par cas} \\ \exists u \in Var(t) \cap Occ(f) \end{cases}$$

Cette règle d'inférence traduit l'opération de décomposition du terme t à l'occurrence u. Elle est effectuée lorsque l'on rencontre un terme qui n'est pas réductible par cas mais qui peut être décomposé. La greffe produit les fils de t pour lesquels nous devrons tester la réductibilité par cas.

$$4) \quad Comp_4 \quad \frac{(Candidats \cup \{t\},\ TC)}{(Candidats,\ TC \cup \{\overrightarrow{c'} :: t\})} \ si \begin{cases} \forall i \in [1 \ldots m],\ (TT :: t) \xrightarrow{c}_R (\vec{c}_i :: t_i), \\ \forall i \in [1 \ldots m],\ \vec{c}_i \rightarrow_R^* \overrightarrow{c'_i}, \\ \overline{Cond} = \vec{c}'_1 \vee \ldots \vee \vec{c}'_m \\ \overline{Cond} \text{ n'est pas une tautologie} \\ Var(t) \cap Occ(f) = \emptyset \end{cases}$$

t est une feuille de l'arbre qui n'est ni réductible par cas ni décomposable. $\overline{Cond}$ est calculé suivant le principe de la définition 2.6. $\overrightarrow{c'}$ est la traduction sous forme de CEB de la formule propositionnelle correspondant à la valeur $\neg\overline{Cond}$ dans le calcul propositionnel.

L'état initial de la procédure de test de la complétude de définition d'un symbole de fonction $f \in F - BF$, de profil $s_1 \times \ldots \times s_n \rightarrow s$ est la structure $(\{f(x_1, \ldots, x_n)\}, \emptyset)$, où pour tout indice i dans $[1 \ldots n]$, les variables x_i sont de sorte s_i et sont toutes distinctes.

Procédure de test de la réductibilité par cas :
La procédure suivante teste la réductibilité par cas de t. La donnée de la procédure comporte le système de réécriture R et le terme t et son résultat est l'ensemble $Prec$ de CEB. La procédure délivre un ensemble $Prec$ non vide dans le cas où le terme t en question n'est pas réductible par cas.
Cette procédure est écrite en termes de règles d'inférence avec la structure de données $(t,\ Prec)$ où

- t est un terme de $T(F,\ X)$ pour lequel on teste la réductibilité par cas.
- $Prec$ est un ensemble de CEB.

$$1)\quad Red_1\quad \frac{(t,\ \emptyset)}{\text{échec}}\quad si\ (TT :: t) \text{ n'est pas réductible par } \xrightarrow{c}_R$$

$$2)\quad Red_2\quad \frac{(t,\ \emptyset)}{(t,\ Prec)}\quad si \begin{cases} \forall i \in [1 \ldots m],\ (TT :: t) \xrightarrow{c}_R (\vec{c}_i :: t_i) \\ \forall i \in [1 \ldots m],\ \vec{c}_i \rightarrow_R^* \vec{c}_i' \\ \text{l'ensemble } \{\vec{c}_i'\}_{i \in I} \text{ n'est pas recouvert} \\ \overline{Cond} = \vec{c}_1' \vee \ldots \vee \vec{c}_m' \\ Prec = \{\neg \overline{Cond}\} \end{cases}$$

Ces deux règles d'inférence traduisent le cas où le terme t n'est pas réductible par cas.

Dans la seconde règle d'inférence, $Prec$ contient la CEB qui doit être ajoutée à l'ensemble des contextes $\vec{c}_i'$ pour le rendre recouvert. $Prec$ est délivré par la procédure de test de tautologie dans le calcul propositionnel.

$$3)\quad Red_3\quad \frac{(t,\ \emptyset)}{\text{succès}}\quad si \begin{cases} \forall i \in I,\ (TT :: t) \xrightarrow{c}_R (\vec{c}_i :: t_i) \\ \forall i \in I,\ \vec{c}_i \rightarrow_R^* \vec{c}_i' \\ \{\vec{c}_i'\}_{i \in I} \text{ est recouvert} \end{cases}$$

Cette règle traduit le cas où t est réductible par cas.

L'état initial de la procédure est la structure $(t,\ \emptyset)$.

Le test de recouvrement d'un ensemble de CEB est effectué en vérifiant par une méthode appropriée, qu'une certaine formule $\tilde{c}$ est une tautologie dans le calcul propositionnel (voir déf 2.6). Ce test n'est pas explicité ici.

Lorsque l'ensemble $Cond$ de CEB donné en entrée et correspondant au terme $f(t_1, \ldots, t_n)$, n'est pas recouvert, il suffirait d'ajouter au système de réécriture la règle de la forme $\neg\tilde{c} \Rightarrow f(t_1, \ldots, t_n) \rightarrow d$ où d est un membre droit de règle qu'il faut définir pour $f(t_1, \ldots, t_n)$. Toutefois, étant donné que la propriété de recouvrement est une condition suffisante mais non nécessaire, si la procédure répond par la négative, cela n'entraîne pas systématiquement que l'opérateur en question n'est pas complètement défini. On peut avoir un ensemble de préconditions qui constituent un ensemble de CEB non recouvert mais néanmoins lié à un symbole de fonction complètement défini par rapport à C. De tels exemples sont exhibés dans le paragraphe suivant. Il s'agirait dans ce cas d'examiner avec davantage de soin, le terme $f(t_1, \ldots, t_n)$ dans le but de vérifier si l'incomplétude est réélle ou non. Dans le cas où l'on peut se restreindre à une classe de spécifications conditionnelles pour laquelle le test de recouvrement est une méthode de décision, la règle $\neg\tilde{c} \Rightarrow f(t_1, \ldots, t_n) \rightarrow d$ est exactement la règle qu'il faut ajouter pour que l'opérateur f soit complètement défini par rapport aux constructeurs à ce nœud de l'arbre.

4.2 Résultat principal de correction

Nous formulons à présent le théorème principal de convertibilité. Considérer un système de réécriture conditionnelle décroissant nous permet d'établir une preuve simple de ce théorème par récurrence structurelle sur les termes clos.

En termes abstraits, l'algorithme construit pour $t = f(x_1, \ldots, x_n)$, un arbre $\mathcal{T}$ de motifs tel que, pour toute feuille t_n, deux cas sont possibles:

- soit t_n est réductible par cas.
- soit t_n n'est pas décomposable.

L'algorithme termine avec succès si et seulement si chaque feuille est réductible par cas. Le résultat suivant, prouvé dans le cas non conditionnel par Kounalis [65] montre que l'arbre $\mathcal{T}$ est complet par rapport aux constructeurs.

Proposition 2.4 [65]
Soit t le terme $f(x_1, \ldots, x_n)$ avec f dans $F \ominus BF$. Soit $\mathcal{T}$ l'arbre construit par l'algorithme et soit σ une substitution close. Il existe alors une feuille t' de $\mathcal{T}$ et une substitution σ' telles que $\sigma t = \sigma' t'$.

La preuve de cette proposition est établie à partir de la méthode de construction de l'arbre de motifs. En effet, d'une part, l'arbre est construit en effectuant 0 ou plusieurs greffes au terme t et en utilisant pour chaque greffe, tous les constructeurs de la sorte correspondante; cet arbre recouvre donc l'ensemble des termes que l'on peut construire avec le symbole f en tête et des termes primitifs pour arguments. D'autre part, puisque t' est une feuille de l'arbre de motifs, t' a f pour symbole de tête et des arguments construits à partir de $C \cup X$; il existe par conséquent une instance close primitive d'une feuille de l'arbre égale à une instance spécifique σt du terme $f(x_1, \ldots, x_n)$.

La preuve du théorème qui suit est effectuée par réccurrence sur l'ordre $\succ$. Rappelons que $\succ$ est une extension bien fondée d'un ordre de réduction $>$ contenant la relation de

sous-terme propre.

Pour cette preuve, nous utiliserons les notations suivantes :
Soit $\vec{c}$ une CEB de la forme $\wedge_{i=1}^{n}\ c_i = c_i'$; si pour tout i dans $[1 \ldots n]$, $s \succ c_i$ et $s \succ c_i'$, nous écrivons aussi $s \succ\!\!\succ \vec{c}$. $\succ\!\!\succ$ est l'extension de $\succ$ aux multi-ensembles.
De même, si pour tout i dans $[1 \ldots n]$, $s > c_i$ et $s > c_i'$, on note $s \gg \vec{c}$. $\gg$ est l'extension de $>$ aux multi-ensembles.
Les ordres $\succ\!\!\succ$ et $\gg$ sont bien fondés puisque d'après [18], l'extension aux multi-ensembles d'un ordre bien fondé est aussi un ordre bien fondé.

Théorème 2.2 *Soit $(S,\ F,\ R)$ un système de réécriture conditionnelle décroissant et structuré de base $(S,\ BF,\ BR)$ tel que BR soit convertible par rapport à C. Supposons que les seuls constructeurs de sorte booléenne soient les constantes true et false.*
Si la procédure de convertibilité établie page 56 s'arrête avec succès, R est convertible par rapport à C.

Preuve : Il s'agit de prouver la propriété $\mathcal{P}$ qui s'énonce comme suit :

$$\forall \omega \in T(F),\ \exists \bar{\omega} \in T(C) \text{ tel que } \omega \rightarrow_R^* \bar{\omega}.$$

Supposons que la propriété $\mathcal{P}$ soit valide pour tous les termes ω' tels que $\omega \succ \omega'$. Si ω est dans $T(BF)$, le résultat découle de l'hypothèse de convertibilité de BR par rapport à C. Autrement, considérons un sous-terme ω" de ω de la forme $f(\omega_1, \ldots,\ \omega_n)$ avec f dans DF et $\omega_1, \ldots,\ \omega_n$ dans $T(BF)$. Nous nous proposons de prouver dans un premier temps, que ω" est réductible. Si l'un des termes ω_i est dans $T(BF) \ominus T(C)$, la réductibilité de ω" découle de la convertibilité de BR. Sinon, supposons que les termes $\omega_1, \ldots,\ \omega_n$ soient dans $T(C)$.

Soient $t = f(x_1, \ldots,\ x_n)$ et σ_0 une substitution close tels que $\omega" = \sigma_0 t$. Par la proposition 2.5, il existe une feuille t' dans l'arbre de motifs construit par l'algorithme pour t, et une substitution σ' telles que $\sigma_0 t = \sigma' t'$.

Si la procédure se termine avec succès, il existe une séquence de contextes $\vec{q}_1, \ldots,\ \vec{q}_m$ et un ensemble de termes $\alpha_1, \ldots,\ \alpha_m$ tels que

$$\forall i \in [1 \ldots m],\ (TT :: t') \xrightarrow{c}_R (\vec{q}_i :: \alpha_i),\ \text{et } \{\vec{q}_1^{\,\prime}, \ldots,\ \vec{q}_m^{\,\prime}\} \text{ est recouvert.}$$

où pour tout i, $\vec{q}_i^{\,\prime}$ est la $\rightarrow_R$- forme normale de $\vec{q}_i$.

Par définition, il existe pour tout i, une règle $\vec{p}_i \Rightarrow g \rightarrow d_i$, une occurrence v de t' et une substitution τ telles que $t'_{/v} = \tau g$ et $\vec{q}_i = \tau \vec{p}_i$. Par conséquent, τg est un sous-terme propre de t'. Donc $t' \succ \tau g \succ\!\!\succ \tau \vec{p}_i = \vec{q}_i$ puisque R est décroissant. De plus, $\vec{q}_i \gg \vec{q}_i^{\,\prime}$ puisque $\vec{q}_i \rightarrow_R^* \vec{q}_i^{\,\prime}$.

Soient à présent $a_1, \ldots,\ a_p$ les atomes apparaissant dans $\vec{q}_1^{\,\prime}, \ldots,\ \vec{q}_m^{\,\prime}$. Nous avons encore $t' \succ\!\!\succ \vec{q}_i^{\,\prime}$ et $\vec{q}_i^{\,\prime} \succ\!\!\succ a_k$ si a_k apparaît dans $\vec{q}_i^{\,\prime}$. Par conséquent, $\sigma_0 t = \sigma' t' \succ \sigma' a_k$. Par hypothèse de récurrence sur $\sigma' a_k$, il existe un terme a dans $T(C)_{bool} (= \{true,\ false\})$ tel que $\sigma' a_k \rightarrow_R^* a$.

Puisque $\{\vec{q}_1^{\,\prime}, \ldots,\ \vec{q}_m^{\,\prime}\}$ est recouvert, par le lemme 2.1, il existe un indice i dans $[1 \ldots m]$ tel que $\sigma'\ \vec{q}_i^{\,\prime} \rightarrow_R^* TT$. De plus, $\sigma' \vec{q}_i \rightarrow_R^*\ \sigma'\ \vec{q}_i^{\,\prime} \rightarrow_R^* TT$. $\sigma' t'$ est donc réductible par la

proposition 2.4, c'est-à-dire $\sigma_0 t(=\omega")$ est aussi réductible en un certain terme $t"$.

Soit à présent ω''' le terme ω où $\omega"(=\sigma_0 t)$ est remplacé par $t"$. Puisque $\omega" \rightarrow_R^* t"$, on a $\omega" > t"$ et $\omega > \omega'''$ puisque $>$ est un ordre de réduction. Par conséquent, $\omega \succ \omega'''$. Par hypothèse de récurrence sur ω''', il existe un terme $\overline{\omega'''}$ dans $T(\mathcal{C})$ tel que $\omega''' \rightarrow_R^* \overline{\omega'''}$. Par conséquent, $\omega \rightarrow_R^* \omega''' \rightarrow_R^* \overline{\omega'''}$. □

5 Un prototype expérimental

Un prototype expérimental concrétisant le travail théorique de ce chapitre et dont l'objectif est de tester la convertibilité des systèmes de réécriture conditionnelle a été implanté par A. Bouhoula, élève stagiaire de l'école d'ingénieur de Tunis, sous la direction de J.L. Rémy,[7]. Le logiciel a été conçu en CAML version 2.6.1, langage fonctionnel d'une grande souplesse [28, 29, 27, 30], sur des stations de travail SUN III, et avec Suntools, qui est un outil de développement multifenêtré.

Le test de convertibilité se base sur l'algorithme présenté dans ce travail. L'objectif du prototype est double ; en plus du test de convertibilité, il est aussi un outil d'aide à la construction de spécifications complètes. L'implantation est souple et agréable d'utilisation, puisqu'elle permet notamment d'afficher l'arbre de décomposition correspondant au système initial et également les arbres construits chaque fois que l'utilisateur ajoute de nouvelles règles de réécriture.

Etant donné un système de réécriture R dont les règles sont écrites avec l'ensemble $BF \oplus DF$ de symboles de fonction tel que BF contient les constructeurs et tel que le système BR est convertible ; lors du test de complétude de définition d'un symbole f de DF, le prototype réalisé peut réagir de différentes façons :

- soit sa réponse est affirmative. Dans ce cas, tous les nœuds de l'arbre construit sont réductibles par cas. Le système est par conséquent convertible par rapport aux constructeurs de la spécification.
- soit il rencontre des nœuds pour lesquels le test de réductibilité a échoué. Le prototype suggère alors à l'utilisateur d'ajouter des règles en lui proposant des paires sous la forme (précondition, membre gauche de règle) et demande à l'utilisateur s'il désire continuer le test de complétude.
  - si l'utilisateur veut s'arrêter à cet instant, le prototype affiche le message *je ne sais pas* et s'arrête. Dans ce cas, on ne peut rien conclure.
  - si l'utilisateur désire continuer le test, le prototype lui demande s'il juge nécessaire d'ajouter des règles au système de réécriture.
    - s'il est nécessaire d'ajouter des règles, le prototype tient compte de cet ajout et réexamine de nouveau l'arbre de décomposition pour tester la réductibilité des nœuds où il y eu problèmes. Cependant, dans ce cas, il peu arriver que le système de réécriture final définisse un symbole de fonction de manière redondante. La raison en est que les suggestions du prototype ne sont pas toujours indispensables, puisque le test est fondé sur une condition suffisante mais non nécessaire.
    - si par contre il est inutile d'enrichir le système, le prototype affiche le message *si vos hypothèses sont vraies, la spécification est opérationnellement*

complète. En effet, dans ce cas, nous jugeons que l'utilisateur a eu recours à d'autres moyens pour vérifier que la fonction est définie dans les nœuds en question et que la décision finale lui revient.

6 Exemples

L'intérêt de ce paragraphe est de présenter d'une part, l'apport de notre méthode par rapport à celle déjà existante établie par Rémy et Uhrig [104] et d'autre part, ses insuffisances. Nous illustrons ces caractéristiques par divers exemples.

Exemple 2.6 *Ce premier exemple spécifie l'opérateur logique de disjonction* $\vee$ *et contient uniquement des règles inconditionnelles. Nous utilisons le symbole de fonction* OU *pour le différencier de l'opérateur* $\vee$ *qui lui, est prédéfini dans toute spécification conditionnelle ;*

Soit $BSP = (S,\ BF,\ BE) \subset SP = (S,\ F,\ E)$ *avec :*
$S = \{bool\}$;

$BF = \{true,\ false :\rightarrow bool\},\ BR = \emptyset$ *et* $C = BF$;
$F \ominus BF = \{OU : bool \times bool \rightarrow bool\},$

$R \ominus BR = \{OU(true,\ x) \rightarrow true,\ OU(x,\ x) \rightarrow x,\ OU(x,\ true) \rightarrow true\},$

$Sh_{bool} = \{true,\ false\}$ *et* $Occ(OU) = \{\epsilon,\ 1,\ 2\},$

Soit $t = OU(x_1,\ x_2)$; t n'est pas réductible par $\xrightarrow{c}_R$. Soit $Var(t) = \{1,\ 2\}$. On choisit une occurrence u dans $Var(t) \cap Occ(OU)$, soit $u = 1$,

$$Mot(t,\ 1) = \{OU(true,\ x_2),\ OU(false,\ x_2)\}.$$

Soit $t_1 = OU(true,\ x_2)$, t_1 est réductible par cas car

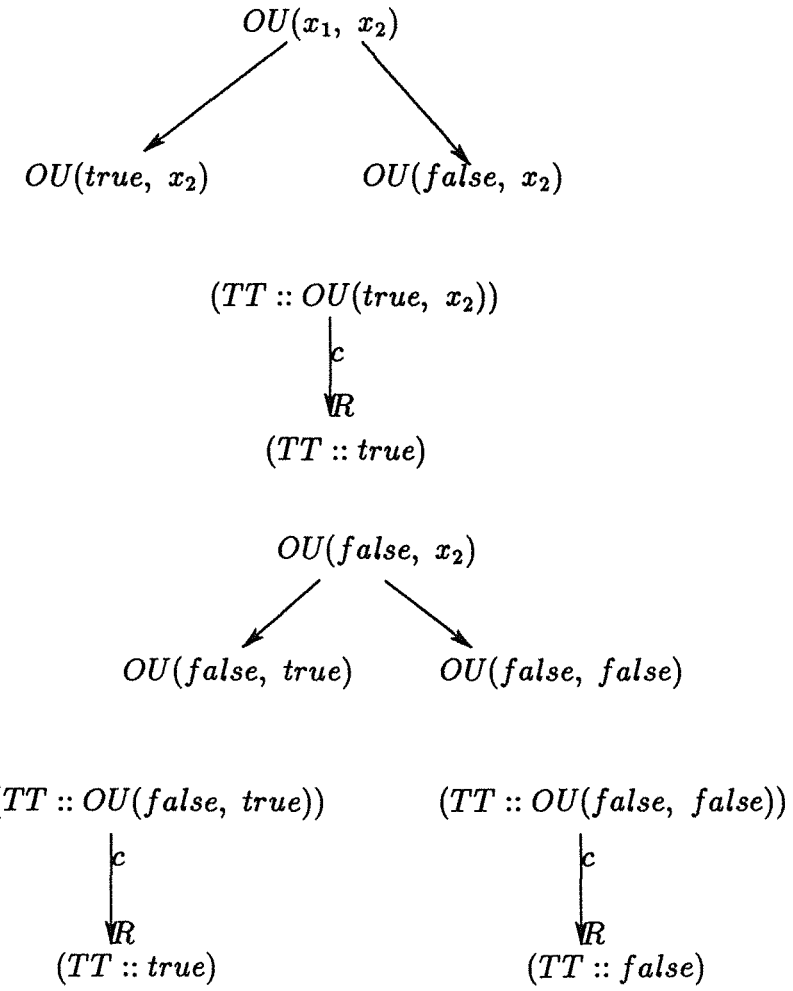
$$(TT :: t_1) \xrightarrow{c}_R (TT :: true)\,;$$

considérons $t_2 = OU(false,\ x_2)$, t_2 n'est pas réductible par $\xrightarrow{c}_R$.
$Var(t_2) = \{2\} \in Occ(OU)$. On calcule

$$Mot(t_2,\ 2) = \{OU(false,\ true),\ OU(false,\ false)\},$$

Les deux termes de $Mot(t_2,\ 2)$ sont réductibles par cas puisqu'ils sont réductibles par des règles inconditionnelles. Par conséquent, OU est complètement défini et R est convertible par rapport à C. $\diamond$

Arbre de décomposition de l'opérateur OU :



Exemple 2.7 *Considérons la spécification des listes triées avec un opérateur à résultat booléen qui permet de tester si la liste est triée.*

$S = \{bool,\ nat,\ list\}$;

$BF = \{true,\ false :\rightarrow bool,$
$\quad 0 :\rightarrow nat,\ succ : nat \rightarrow nat,\ \leq\ : nat \times nat \rightarrow bool,$
$\quad [\,] :\rightarrow list,\ cons : nat \times list \rightarrow list\}$;
$F \ominus BF = \{ordered : list \rightarrow bool\}$;

$BR = \{0 \leq x \rightarrow true,\ succ(x) \leq 0 \rightarrow false,\ succ(x) \leq succ(y) \rightarrow x \leq y\}$;
$R = BR \oplus \{ordered([\,]) \rightarrow true,\ ordered(cons(x,\ [\,])) \rightarrow true,$
$\quad (x \leq y) = true \Rightarrow ordered(cons(x,\ cons(y,\ u))) \rightarrow ordered(cons(y,\ u)),$
$\quad (x \leq y) = false \Rightarrow ordered(cons(x,\ cons(y,\ u))) \rightarrow false\}.$

Les constructeurs constituent l'ensemble $C = \{true,\ false,\ 0,\ succ,\ [\,],\ cons\}$;
$Sh_{bool} = \{true,\ false\}$, $Sh_{nat} = \{0,\ succ(x)\}$, $Sh_{list} = \{[\,],\ cons(x,\ u)\}$,
et $Occ(ordered) = \{\epsilon,\ 1,\ 12\}$.

Nous pouvons constater que l'opérateur $\leq$ est complètement défini par rapport à C.

Par convention, on utilise les variables u_i pour la sorte *list* et les variables x_i pour la sorte *nat*.

On se propose de vérifier la complétude de définition de l'opérateur *ordered*.
Soit le terme $t = ordered(u_1)$, $Var(t) = \{1\}$, t n'est pas réductible par $\xrightarrow{c}_R$; on calcule $Mot(t,\ 1) = \{ordered([\,]),\ ordered(cons(x_1,\ u_2))\}$.

Le premier terme est réductible par cas puisqu'il est réductible par une règle inconditionnelle. Considérons le terme $t' = ordered(cons(x_1,\ u_2))$.
$Var(t') = \{11,\ 12\}$, soit $u = 12 \in Var(t') \cap Occ(ordered)$;

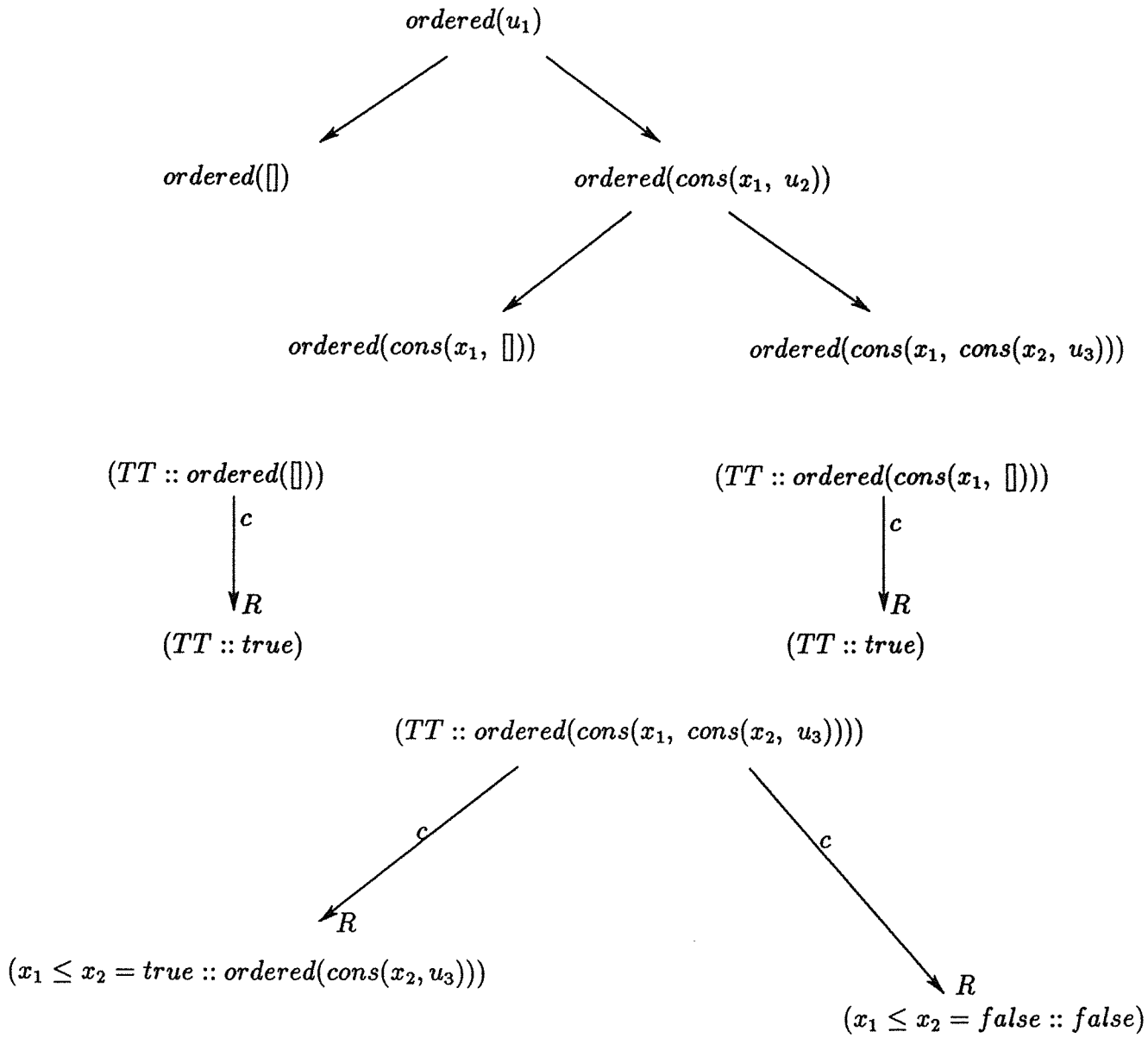
$$Mot(t',\ 12\} = \{ordered(cons(x_1, [\,])),\ ordered(cons(x_1,\ cons(x_2,\ u_3)))\}.$$

Le premier terme est réductible par cas puisqu'il est réductible par une règle inconditionnelle. Considérons le second terme, nous avons
$(TT :: ordered(cons(x_1, cons(x_2, u_3)))) \xrightarrow{c}_R (x_1 \leq x_2 = true :: ordered(cons(x_2, u_3)))$ et
$(TT :: ordered(cons(x_1, cons(x_2, u_3)))) \xrightarrow{c}_R (x_1 \leq x_2 = false :: false)$

L'ensemble $\{(x_1 \leq x_2) = true,\ (x_1 \leq x_2) = false\}$ est recouvert puisque la formule $p \vee \neg p$, où p correspond à l'atome $x_1 \leq x_2$ est une tautologie dans le calcul propositionnel donc,
$ordered(cons(x_1,\ cons(x_2,\ u_3)))$ est réductible par cas.

Finalement, nous pouvons conclure que l'opérateur *ordered* est complètement défini par rapport aux constructeurs. $\diamond$

Arbre de décomposition de l'opérateur *ordered*



Exemple 2.8 *Ajoutons à l'ensemble F l'opérateur $insert : nat \times list \rightarrow list$ et à R les règles suivantes :*

$$\begin{cases} insert(x,\ [\,]) \rightarrow cons(x,\ [\,]), \\ (x \leq y) = true \Rightarrow insert(x,\ cons(y,\ u)) \rightarrow cons(x,\ cons(y,\ u)), \\ (x \leq y) = false \Rightarrow insert(x,\ cons(y,\ u)) \rightarrow cons(y,\ insert(x,\ u)), \\ ordered(insert(x,\ u)) \rightarrow ordered(u) \end{cases}$$

$$Occ(insert) = \{\epsilon,\ 2\}$$

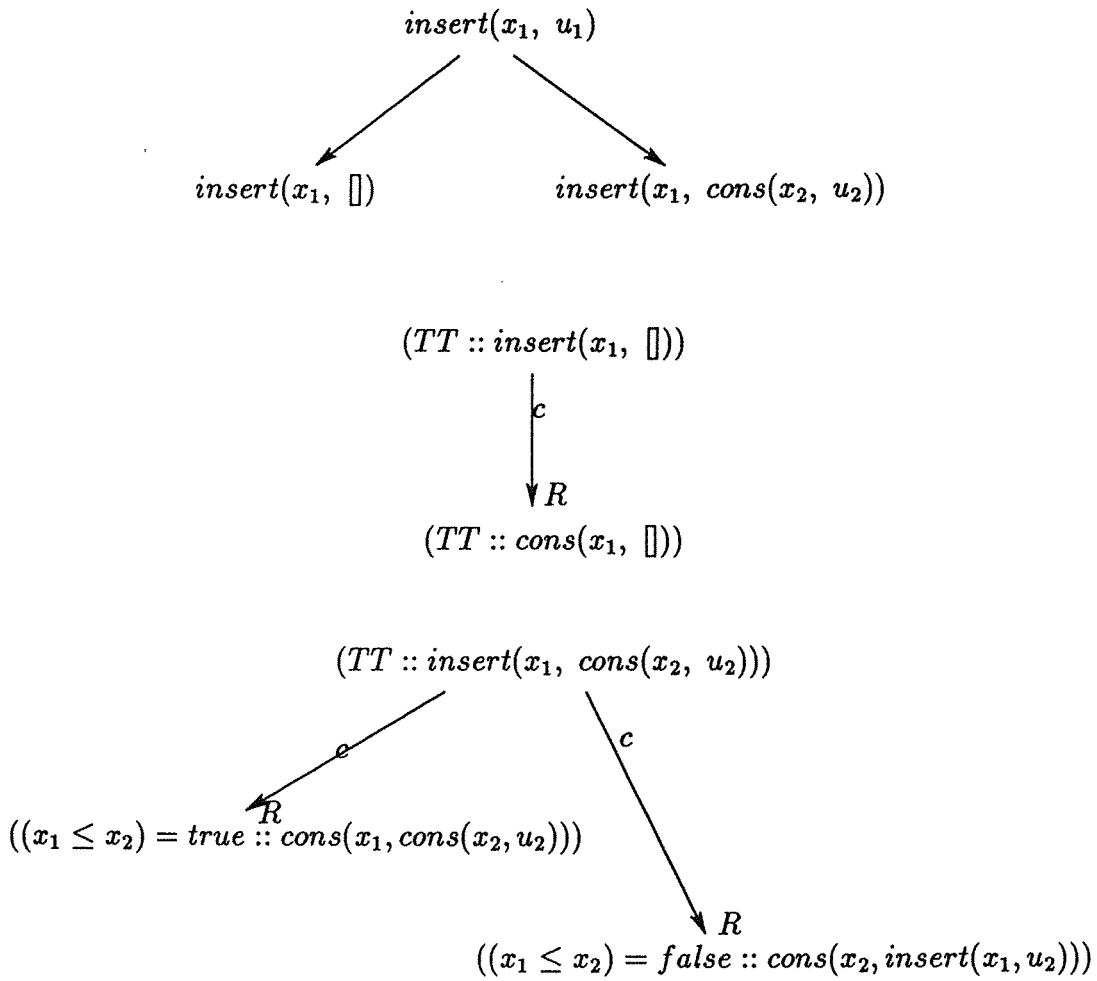
Soit le terme $t = insert(x_1,\ u_1)$, t n'est pas réductible par $\xrightarrow{c}_R$; nous allons donc construire un ensemble de motifs ; $Var(t) = \{1,\ 2\}$

soit l'occurrence 2 dans $Var(t) \cap Occ(insert)$;

$$Mot(t,\ 2) = \{insert(x_1,\ [\,]),\ insert(x_1,\ cons(x_2,\ u_2))\},$$

Ces deux termes sont réductibles par cas et on peut conclure que R est convertible par rapport à C. $\diamond$

Arbre de décomposition de l'opérateur *insert* :



Exemple 2.9 *Cet exemple simule l'axiomatisation de l'opérateur logique de conjonction par des équations conditionnelles; de même que dans le cas de l'opérateur OU, nous utilisons le symbole & pour éviter toute confusion entre ce nouvel opérateur et le connecteur logique* $\wedge$ *;*

$S = \{bool\}$;

$BF = \{true,\ false :\rightarrow bool\},\ C = BF$;
$F \ominus BF = \{\& : bool \times bool \rightarrow bool\}$;

$BR = \emptyset$;
$R = BR \oplus \{\&(true,\ true) \rightarrow true,\ \&(true,\ false) \rightarrow false,$
$x = false \Rightarrow \&(x,\ y) \rightarrow x\}$;

$Sh_{bool} = \{true,\ false\},\ Occ(\&) = \{\epsilon,\ 1,\ 2\}$;

Soit le terme $t = \&(x_1,\ x_2)$; t est réductible par la relation $\xrightarrow{c}_R$ et $(TT\ ::\ t) \xrightarrow{c}_R (x_1 = false\ ::\ x_1)$. L'ensemble $\{x_1 = false\}$ n'est pas recouvert. Par conséquent, t n'est pas réductible par cas. Toutefois, comme t a des occurrences de variables linéaires qui sont aussi des composantes de l'ensemble $Occ(\&)$, on ne peut rien conclure avant de tester la réductibilité par cas des motifs que l'on peut construire à partir de t.

On construit l'arbre à partir de $t : Var(t) = \{1,\ 2\}$;

$$Mot(t,\ 1) = \{\&(true,\ x_2),\ \&(false,\ x_2)\}.$$

Considérons le terme $t' = \&(true,\ x_2)$; on a les réductions suivantes :

$$(TT :: t') \xrightarrow{c}_R (true = false :: true) \text{ et } true = false \rightarrow_R FF.$$

$\{FF\}$ n'est pas recouvert. Donc t' n'est pas réductible par cas.

De même que pour t, il est cependant possible de construire un ensemble de motifs à partir de t'. $Var(t') = \{2\} \subset Occ(\&)$.

$$Mot(t',\ 2) = \{\&(true,\ true),\ \&(true,\ false)\}.$$

Chacun de ces termes est contextuellement réductible respectivement par les deux premières règles de R et puisque ces règles sont inconditionnelles, ils sont donc réductibles par cas.

Considérons à présent le terme $t"$ = $\&(false,\ x_2)$;

$$(TT :: t") \xrightarrow{c}_R (false = false :: false)$$

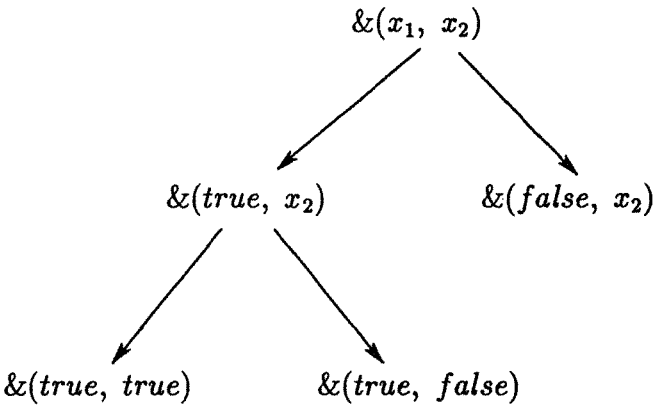
et $false = false \rightarrow_R TT$. Donc $t"$ est réductible par cas.

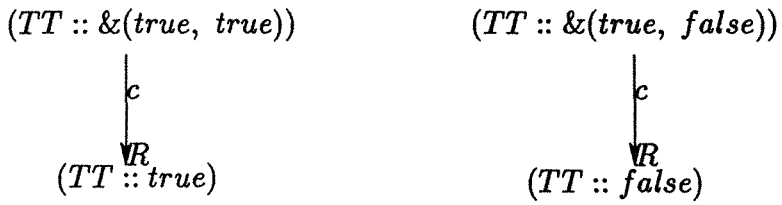
Nous avons donc construit un arbre avec les feuilles suivantes :

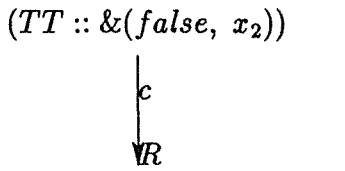
1. $\&(true,\ true)$
2. $\&(true,\ false)$
3. $\&(false,\ x_2)$

Puisque chacun de ces termes est réductible par cas, R est convertible par rapport à l'ensemble de ses constructeurs $\{true,\ false\}$. $\diamond$

Arbre de décomposition de l'opérateur &:







$false = false \rightarrow_R TT$

Cet exemple qui axiomatise l'opérateur $\wedge$ montre qu'aucune forme syntaxique n'est imposée aux préconditions du système contrairement aux travaux de Rémy et Uhrig [104]. Une hypothèse essentielle que font ces derniers est que tout symbole de fonction est défini par un ensemble de règles conditionnelles de même membre gauche et dont les préconditions sont des conjonctions complémentaires de littéraux. Une classe considérable de spécifications échappe à cette caractéristique et notamment l'exemple traité ci-dessus. Dans nos travaux, une méthode permettant d'assouplir cette rigidité de définition consiste à adopter le traitement suivant : lorsqu'un terme t dont le symbole de tête est f, dans $F \ominus BF$, n'est pas réductible par cas et qu'il possède des occurrences de variables non-linéaires contenues dans l'ensemble $Occ(f)$, nous continuons à construire l'arbre des motifs à partir de t en choisissant une occurrence appropriée pour tester la réductibilté par cas des nouveaux termes. Cette méthode permet d'avoir une forme plus souple de précondition de règle. Bien sûr, cela n'exclut pas la nécessité d'avoir un ensemble de CEB opérationnellement valide associé à tout membre gauche de règle du système de réécriture.

Exemple 2.10 *Cet exemple illustre le procédé utilisé pour compléter une définition incomplète d'un opérateur. Il spécifie la liste ordonnée d'entiers.*

Le prédicat "$x \ll u$" est vrai si x est le plus petit élément de la liste u. "ord" teste si la liste est triée et "$insert(x,\ u)$" insère l'élément x dans la liste u de telle façon que le résultat de l'insertion soit encore une liste triée.

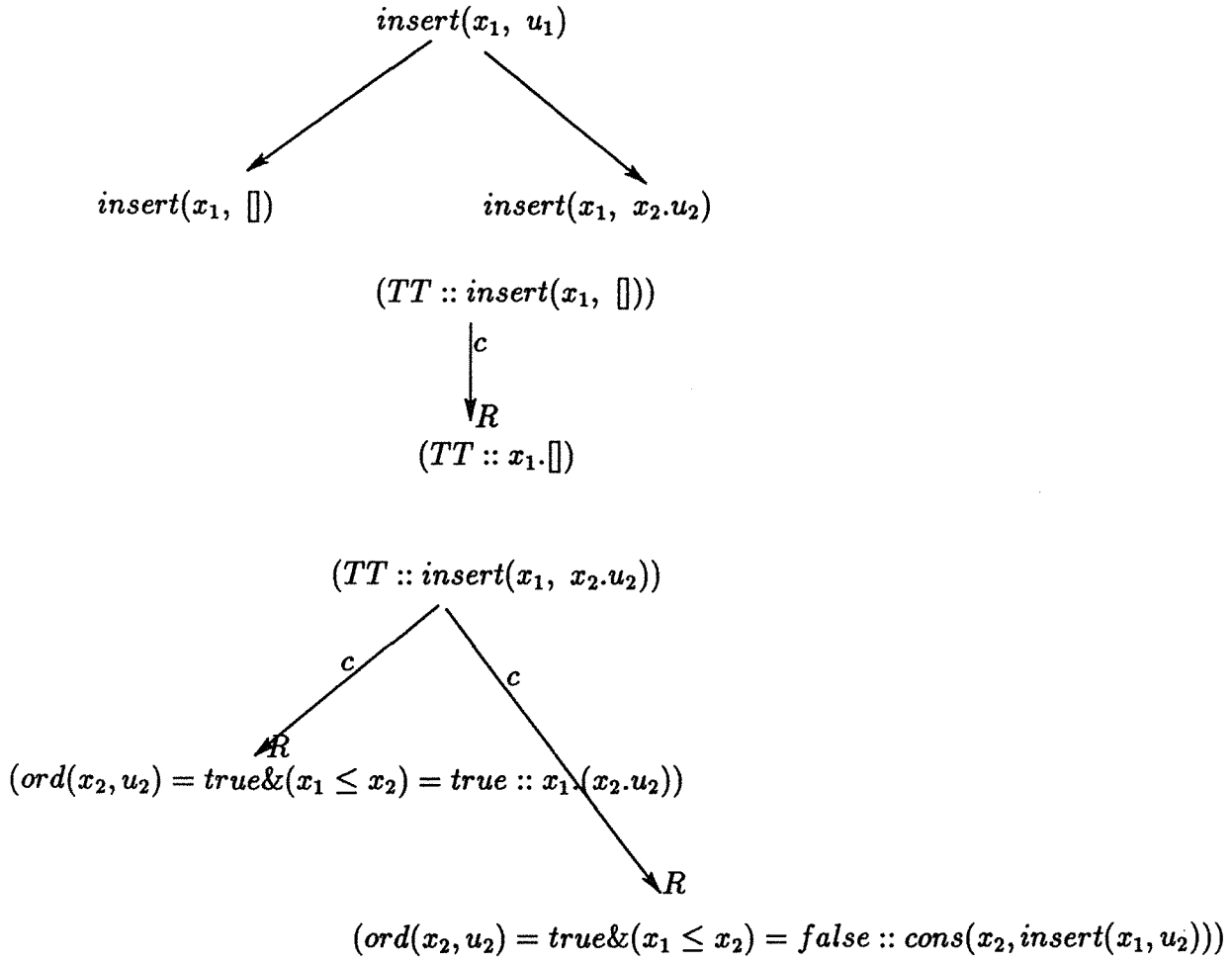
$S = \{bool,\ ent,\ list\}$;

$BF = \{true,\ false :\rightarrow bool,$
$0 :\rightarrow ent,\ succ : ent \rightarrow ent,$
$\leq : ent \times ent \rightarrow bool,$
$[\,] :\rightarrow list,\ . : ent \times list \rightarrow list,$
$\ll : ent \times list \rightarrow bool,\ ord : list \rightarrow bool\}$;
$F \ominus BF = \{insert : ent \times list \rightarrow list\}$;
et $C = \{true,\ false,\ 0,\ succ,\ [\,],\ .\}$.

$BR = \{0 \leq x \rightarrow true,\ succ(x) \leq 0 \rightarrow false,\ succ(x) \leq succ(y) \rightarrow x \leq y,$
$x \ll [\,] \rightarrow true,\ x \ll y.u \rightarrow (x \ll u)\ \wedge\ (x \leq y),$
$ord([\,]) \rightarrow true,\ ord(y.u) \rightarrow ord(u)\ \wedge\ (y \ll u)$;
$R \ominus BR = \{insert(x,\ [\,]) \rightarrow x.u,$
$ord(y.u) = true\ \wedge\ (x \leq y) = true \Rightarrow insert(x,\ y.u) \rightarrow x.(y.u),$
$ord(y.u) = true\ \wedge\ (x \leq y) = false \Rightarrow insert(x,\ y.u) \rightarrow y.insert(x,\ u)\}$.

Nous pouvons remarquer d'une part, que l'opérateur $\leq$ est complètement défini par rapport à $\{0,\ succ\}$ et d'autre part, que les opérateurs $\ll$ et ord sont complètement définis par rapport à $\{[\,],\ .\}$.
Nous nous proposons de tester la complétude de définition de l'opérateur *insert*. L'algorithme produit l'arbre de décomposition suivant :

Arbre de décomposition de l'opérateur *insert* :



Soit à tester le recouvrement de l'ensemble $Cond =$

$$\{ord(x_2.u_2) = true \wedge (x_1 \leq x_2) = true,\ ord(x_2.u_2) = true \wedge (x_1 \leq x_2) = false\}$$

correspondant au terme $t = insert(x_1,\ x_2.u_2)$. Soient l'atome a correspondant au terme $ord(x_2.u_2)$ et l'atome b correspondant au terme $x_1 \leq x_2$. Il s'agit de calculer, par le calcul propositionnel, la valeur de la formule $\tilde{c}$:

$$(a \ \wedge \ b) \ \vee \ (a \ \wedge \ \neg b)$$

et de tester si cette formule est une tautologie.

$$(a \ \wedge \ b) \ \vee \ (a \ \wedge \ \neg b) = a \ \wedge \ (b \ \vee \ \neg b) = a$$

Puisque $\tilde{c}$ n'est pas une tautologie, la règle qu'il faudrait (éventuellement) ajouter au système pour que l'opérateur *insert* soit complètement défini, est la règle

$$ord(x_2.u_2) = false \Rightarrow insert(x_1,\ x_2.u_2) \rightarrow d$$

où $ord(x_2.u_2) = false$ correspond à la traduction de $\neg a$ et où d est un membre droit que le spécifieur doit déterminer. $\diamond$

Afin d'établir une comparaison succincte de notre approche et de celle de Rémy et Uhrig, nous présentons brièvement la seconde approche ; soit

$$R_{inc} = \{g \to d \ / \ \exists \vec{p} \Rightarrow g \to d \in R\} ;$$

la méthode systématisée par les auteurs consiste à vérifier d'une part, que le système inconditionnel R_{inc} est convertible par rapport à l'ensemble de ses constructeurs par l'algorithme de Thiel [102] et d'autre part, à tester que pour tout membre gauche de règle g dans R, les préconditions associées à g forment un ensemble recouvert. L'approche de Thiel est fondée sur le principe suivant : la définition d'un opérateur f est complète si et seulement si l'ensemble des membres gauches de la définition *recouvrent* le terme $f(x_1, \ldots, x_n)$. Pour tester ce recouvrement, l'auteur utilise le processus d'unification. La méthode permet également de traiter des définitions non-linéaires de symboles de fonction et par exemple, la définition suivante qui simule la fonction *majorité* notée $+$:

$$E = \{+(x,\ x,\ y) = x,\ +(x,\ y,\ x) = x,\ +(y,\ x,\ x) = x\}.$$

Cependant, Thiel ne traite pas les spécifications qui admettent des relations entre les constructeurs. Sa méthode a été par la suite, étendue par Lazrek dans ses travaux de thèse et par Thiel et Lescanne [71, 72], pour accepter, dans le système, des relations entre les constructeurs, à condition qu'elles engendrent un système canonique.

Exemple 2.11 *Soit la spécification suivante :*

$S = \{nat\}$,

$BF = \{0 :\to nat,\ succ : nat \to nat,\ > : nat \times nat \to bool\}$,
$F \ominus BF = \{f : nat \to nat\}$

$BR = \{0 > x \to false,\ succ(x) > 0 \to true,\ succ(x) > succ(y) \to x > y\}$
$R \ominus BR = \{f(0) \to 0,\ x > 0 = true \Rightarrow f(x) \to 0\}$.

Et soit $C = \{0,\ succ\}$;

Remarquons que l'opérateur $>$ est complètement défini par rapport à l'ensemble $\{0,\ succ\}$. Il est évident que l'ensemble $\{x > 0 = true\}$ associé au terme $f(x)$ n'est pas recouvert et par conséquent, l'algorithme d'Uhrig échoue à ce niveau. Appliquons à présent notre méthode :

Soit $Occ(f) = \{\epsilon,\ 1\}$ et $Sh_{nat} = \{0,\ succ(x)\}$ et soit le terme $t = f(x_1)$.
t est réductible par R et nous avons

$$(TT :: f(x_1)) \xrightarrow{c}_R (x_1 > 0 = true :: 0)$$

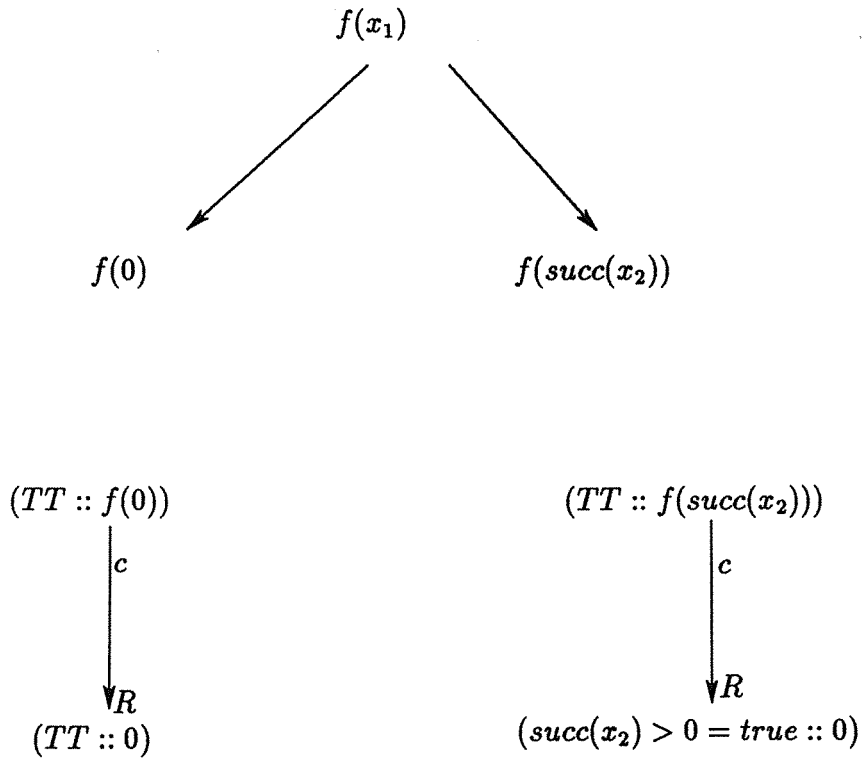
De même que précédemment, l'ensemble $\{x_1 > 0 = true\}$ n'est pas recouvert. Toutefois, puisque $Var(t) = \{1\} \subset Occ(f)$, nous procédons à une décomposition du terme t :

$$Mot(t,\ 1) = \{f(0),\ f(succ(x_2))\}$$

- $(TT :: f(0)) \xrightarrow{c}_R (TT :: 0)$ et
- $(TT :: f(succ(x_2))) \xrightarrow{c}_R (succ(x_2) > 0 = true :: 0)$ avec
  $succ(x_2) > 0 = true \to_R true = true \to_R TT$.

La réduction des termes de $Mot(t,\ 1)$ produit alors des termes primitifs et nous pouvons conclure que f est complètement défini par rapport à C. $\diamond$

Arbre de décomposition de l'opérateur f :



$succ(x_2) > 0 = true \rightarrow_R true = true \rightarrow_R TT$

Examinons la spécification suivante des listes d'entiers bornés par une constante ***entmax*** ; L'expression $pp(x,\ l)$ signifie que x est plus petit que tous les éléments de la liste l.

Exemple 2.12 *La spécification est la suivante :*

$SP = (S,\ F,\ E)\,;S = \{bool,\ nat,\ list\}\,;$

$BF = \{true,\ false :\rightarrow bool,\ 0,\ entmax :\rightarrow nat,$
$succ : nat \rightarrow nat,\ lvide :\rightarrow list,\ add : nat \times list \rightarrow list\}$
et $C = BF\,;$
$F \ominus BF = \{\leq : nat \times nat \rightarrow bool,\ min : list \rightarrow nat,\ pp : nat \times list \rightarrow bool\}.$

$BR = \{succ(entmax) \rightarrow entmax\},$
$R \ominus BR = \{0 \leq x \rightarrow true,\ succ(x) \leq 0 \rightarrow false,$
$succ(x) \leq succ(y) \rightarrow x \leq y,\ x \leq entmax \rightarrow true,$
$entmax \leq 0 \rightarrow false,\ entmax \leq succ(y) \rightarrow entmax \leq y,$

$min(lvide) \rightarrow entmax,$
$x \leq min(l) = true \Rightarrow min(add(x,\ l)) \rightarrow x,$
$x \leq min(l) = false \Rightarrow min(add(x,\ l)) \rightarrow min(l),$

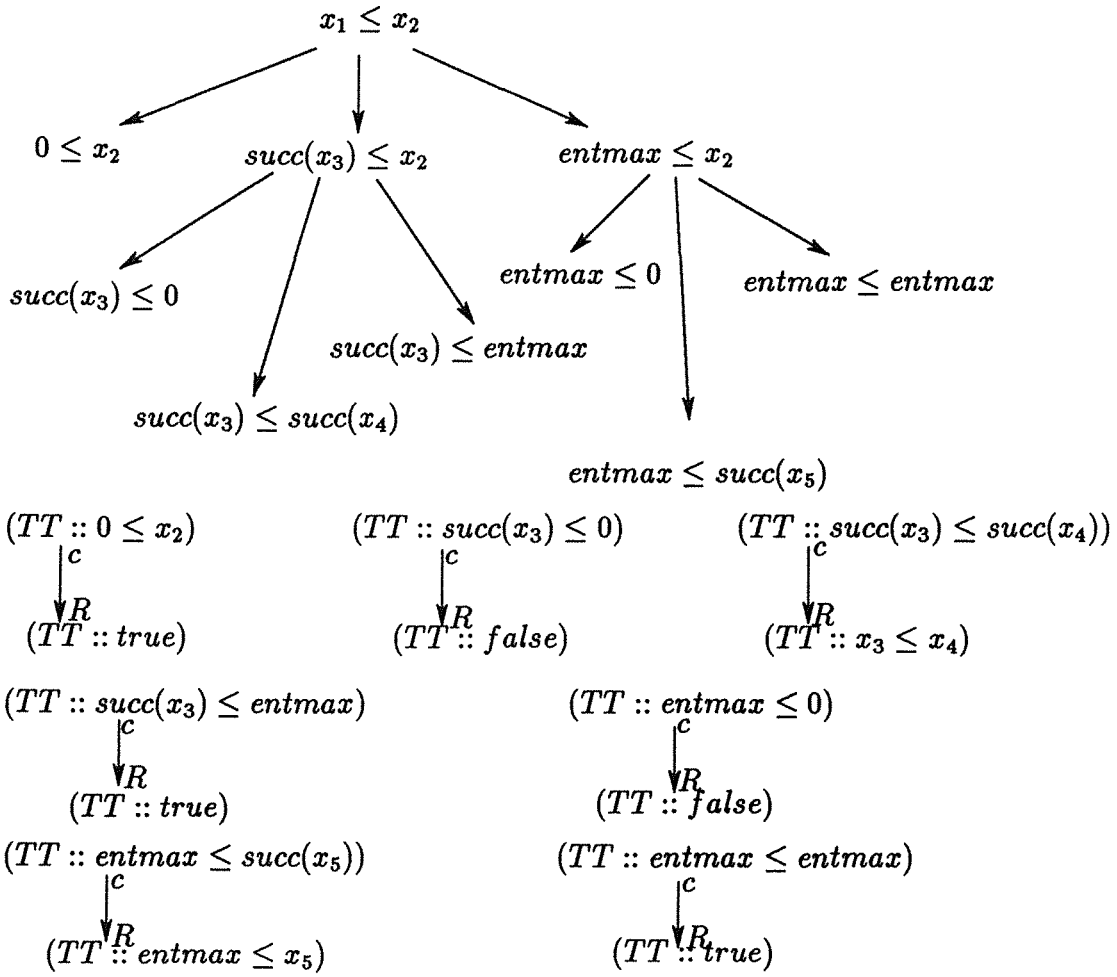
$pp(x,\ lvide) \rightarrow true,$
$x \leq y = true\ \wedge\ pp(x,\ l) = true \Rightarrow pp(x,\ add(y,\ l)) \rightarrow true,$
$x \leq y = false \Rightarrow pp(x,\ add(y,\ l)) \rightarrow false,$
$pp(x,\ l) = false \Rightarrow pp(x,\ add(y,\ l)) \rightarrow false\},$

Soient

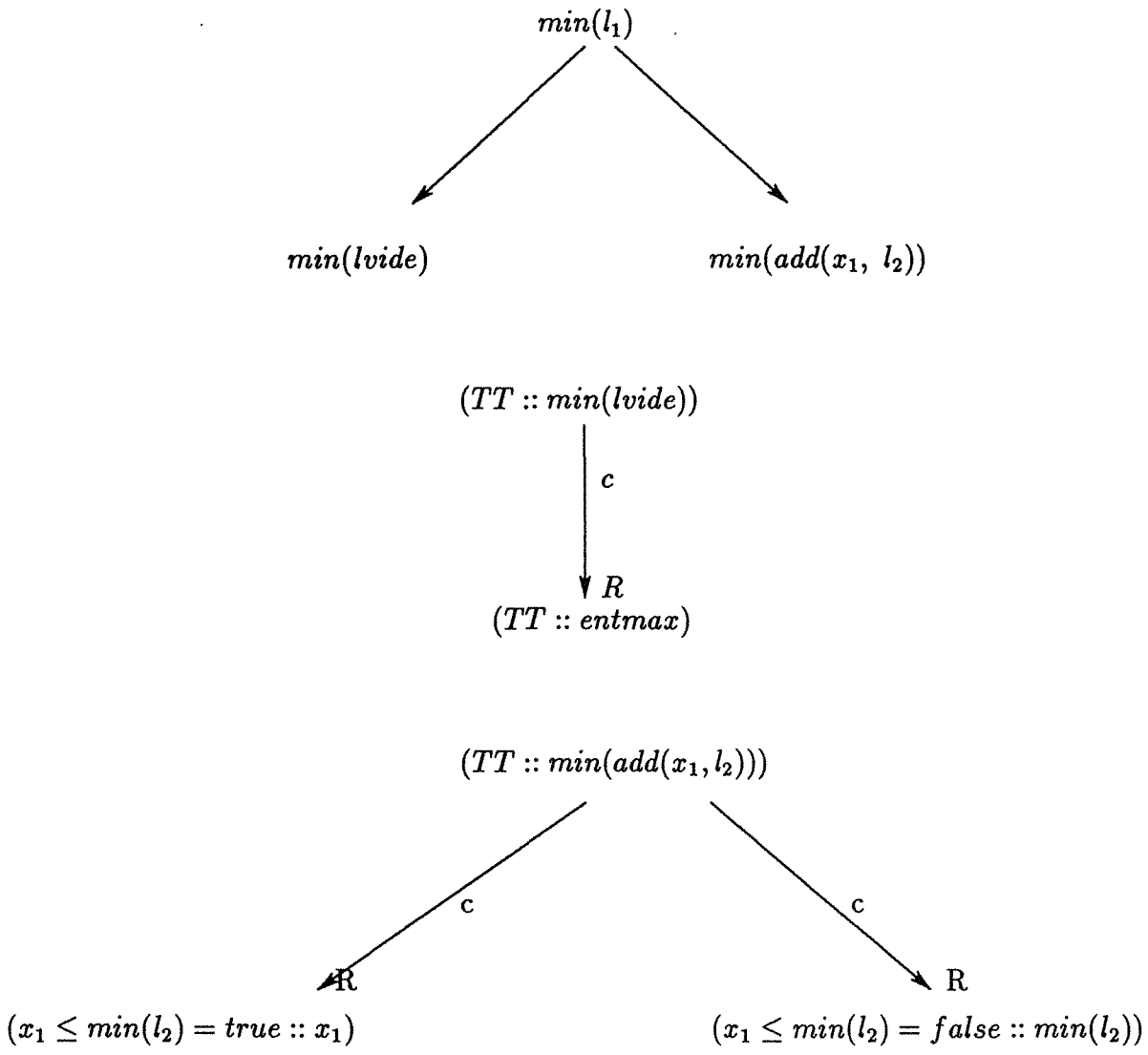
$Sh_{bool} = \{true,\ false\},\ Sh_{nat} = \{0,\ succ(x),\ entmax\},\ Sh_{list} = \{lvide,\ add(x,\ l)\}.$

Montrons successivement que les opérateurs $\leq$, min et pp sont complètement définis :

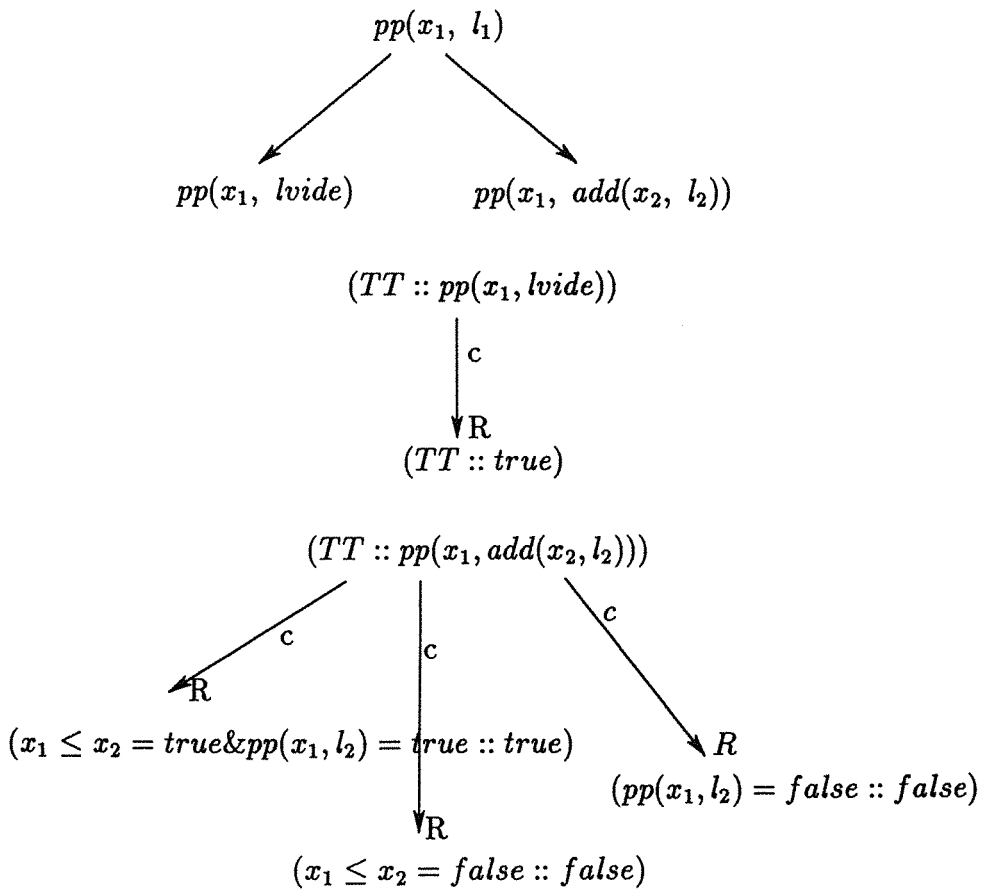
Arbre de décomposition de l'opérateur $\leq$:



Arbre de décomposition de l'opérateur *min* :



Arbre de décomposition de l'opérateur pp :



Soit $Cond$ l'ensemble de CEB correspondant à la réduction contextuelle du terme

$$(TT :: pp(x_1,\ add(x_2,\ l_2)));$$

$$Cond = \{x_1 \leq x_2 = true \ \wedge \ pp(x_1,l_2) = true, x_1 \leq x_2 = false, pp(x_1,l_2) = false\}$$

Il s'agit de coder l'ensemble des CEB de $Cond$ par des formules propositionnelles ; soit c_1 et c_2 correspondant respectivement à $x_1 \leq x_2 = true$ et $pp(x_1,\ l_2) = true$. L'expression

$$\{c_1 \ \wedge \ c_2,\ \neg c_1,\ \neg c_2\}$$

étant une tautologie dans le calcul propositionnel, nous pouvons achever la preuve de complétude de définition de l'opérateur *pp*.
Nous pouvons remarquer dans cet exemple que l'opérateur *pp* dont il s'agit de tester la complétude de définition, apparaît également dans deux des préconditions. Il est cependant possible d'utiliser un argument inductif puisque ces préconditions sont plus petites que le motif lui-même. ◇

Nous avons montré précédemment que la complétude est une notion étroitement liée à la réductibilité inductive. Certaines situations nécessitent même de compléter le test de complétude inductive par un test de réductibilité inductive de certaines feuilles de l'arbre de décomposition. Cette solution permet de pallier la restriction du test de recouvrement d'un ensemble de CEB. Une condition est cependant imposée pour valider le test de réductibilité inductive : le système de réécriture R doit être linéaire à gauche [66]. Nous ne présentons pas de résultats de manière rigoureuse. Nous développons plutôt cette idée sur un exemple. Il serait intéressant à l'avenir de creuser cette voie de recherche.

Exemple 2.13 *Il s'agit de la définition du prédicat* $0 \leq x$ *sur l'ensemble des naturels que nous notons dans l'exemple par p.*

$S = \{bool,\ nat\}$;

$BF = \{true,\ false :\rightarrow bool,\ 0 :\ \rightarrow nat,\ succ : nat \rightarrow nat\}$,
et $C = BF$;
$F \ominus BF = \{p : nat \rightarrow bool\}$;

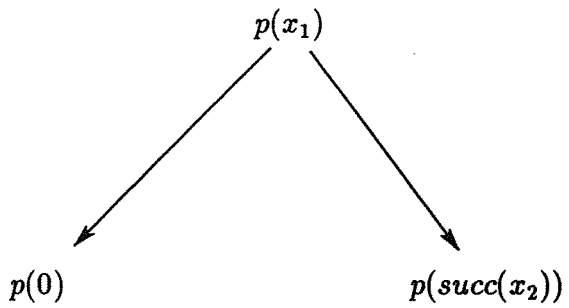
$BR = \emptyset$;
$R = BR \oplus \{p(0) \rightarrow true,\ p(x) = true \Rightarrow p(succ(x)) \rightarrow true\}$.

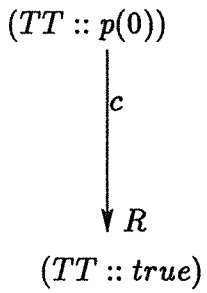
Nous calculons les ensembles suivants :

$$Sh_{bool} = \{true,\ false\},\ Sh_{nat} = \{0,\ succ(x)\},\ Occ(p) = \{\epsilon,\ 1\}.$$

L'arbre de décomposition de l'opérateur p se construit comme suit :

Arbre de décomposition de l'opérateur p:





Le terme $p(succ(x_2))$ n'est pas réductible par cas puisque

$$(TT \ :: \ p(succ(x_2))) \xrightarrow{c}_R (p(x_2) = true \ :: \ true)$$

et que l'ensemble $\{p(x_2) = true\}$ n'est pas recouvert.
D'autre part, les occurrences de variables libres du terme $p(succ(x_2))$ n'étant pas dans $Occ(p)$, il ne sert à rien de développer l'arbre à partir de ce terme. La solution que nous préconisons à ce stade du test de complétude de définition est de tester la réductibilité inductive de $p(succ(x_2))$ en adaptant la méthode proposée par Kounalis et Rusinowitch [67] pour des clauses de Horn aux règles de réécriture conditionnelle.

Tester la réductibilité inductive des clauses de Horn

Nous rappelons dans un premier temps le théorème de réductibilité inductive établi dans l'approche clauses de Horn :

Théorème 2.3 **[67]**
Soient S un ensemble de clauses de Horn linéaires qui préserve les termes clos et B un littéral tels que $P(S, B)$ soit vérifiée; B est inductivement réductible ssi tout élément de $TS(B)$ est réductible par S.

Définition 2.13 *Un ensemble S de clauses de Horn préserve les termes clos si tout littéral positif $s = t$ qui apparaît dans une clause de S est soit orientable, soit satisfait $Var(s) = Var(t)$ et si pour toute clause C de S, les variables de chaque littéral négatif de C sont contenues dans le littéral positif.*

La propriété $P(S, B)$ se définit comme suit :

Définition 2.14 *Soient S un ensemble de clauses de Horn et B un atome; S et B satisfont la propriété $P(S, B)$ si pour chaque clause $\neg A_1 \vee \ldots \vee \neg A_n \vee A$ dans S telle que A soit unifiable avec B par l'unificateur principal σ nous avons, pour tout i dans $[1 \ldots n]$,*

- *soit A_i et B sont unifiables avec un unificateur principal θ et $\sigma = \psi.\theta$ et $Q(A_i) \subset Q(A)$*
- *soit il existe au moins une clause unité C dans S telle que A_i est unifiable avec C.*

$Q(t)$ dénote l'ensemble des occurrences de non-variables de t.
$prof(t)$ désigne la profondeur de t qui est la longueur maximale des occurrences dans le terme t.
La profondeur de S, notée $prof(S)$, est la profondeur maximale des termes apparaissant dans S.
La profondeur d'une substitution σ est égale à $max\{prof(\sigma x), \ x \in Dom(\sigma)\}$.

$TS(B)$ est l'ensemble test de l'atome B et se calcule de la façon suivante :

Définition 2.15 Ensemble test [67]

Soit $niv(S) = max\{|prof(A) - prof(B)|\}$ *où* A *et* B *sont des littéraux de* S. *L'ensemble test d'un terme* B *par rapport à un ensemble de clauses* S *est l'ensemble suivant de termes clos :*

$$TS(B) = \{\sigma B \ / \ \sigma \in Subst(F) \ et \ prof(\sigma) \leq prof(S) + niv(S) + 1\}$$

Tester la réductibilité inductive de certaines feuilles de l'arbre de motifs

Si nous traduisons les règles de réécriture de R par des clauses de Horn, alors nous obtenons un ensemble de clauses de Horn qui préserve les termes clos puisque d'une part, chacune des règles conditionnelles $\vec{p} \Rightarrow g \rightarrow d$ de R vérifie $Var(\vec{p}) \subset Var(g)$ et d'autre part, la seule règle inconditionnelle de R vérifie $Var(g) = Var(d)$. De plus, la propriété $P(p(succ(x_2)))$ est valide.

La profondeur de R est égale à 1, et $niv(R) = 2$.

L'ensemble $TS(p(succ(x_2))) =$

$$\{p(succ(0)),\ p(succ^2(0)),\ p(succ^3(0)),\ p(succ^4(0))\}.$$

Chacun des termes de cet ensemble est réductible par la relation $\xrightarrow{c}_R$ sous le contexte TT. Par conséquent, nous pouvons conclure que l'opérateur p est complètement défini. $\diamond$

Il serait intéressant d'approfondir cette voie de recherche, puisqu'alors l'algorithme de convertibilité s'étendrait à une classe plus large de règles de réécriture, les règles qui définissent des prédicats d'une façon récursive.

Résumons à présent la situation :

- L'algorithme de recouvrement est décidable pour des ensembles de CEB que nous pouvons partager en deux parties, l'une de la forme $\{\vec{q} \wedge a = true\}$ et l'autre de la forme $\{\vec{q} \wedge a = false\}$ [104].
- Une amélioration de cet algorithme serait de traduire les CEB par des formules propositionnelles et de vérifier que l'ensemble ainsi obtenu forme une tautologie. Le calcul propositionnel étant décidable, l'algorithme de recouvrement devient lui-même décidable, (exemple 2.12).
- Si par l'algorithme de recouvrement, nous concluons que l'ensemble de CEB n'est pas recouvert, alors cela n'implique pas que l'opérateur f n'est pas complètement défini. Ceci provient du fait que la propriété de réductibilité par cas est une condition suffisante mais non nécessaire.

7 Conclusion

Nous avons étudié dans ce chapitre, la propriété de complétude suffisante des spécifications conditionnelles. Cette propriété a fait l'objet de nombreux travaux dans le cadre classique, mais elle est encore peu abordée dans le cadre conditionnel. La complétude et la consistance des spécifications algébriques sont des propriétés essentielles, notamment pour assurer la correction de l'instanciation de spécifications paramétrées.

Notre algorithme de test est effectif pour des systèmes de réécriture conditionnelle ayant les particularités d'être structurés et décroissants et ces deux propriétés ne sont pas restrictives et sont souvent supposées par les auteurs dans le cadre conditionnel. L'algorithme permet de vérifier qu'un symbole de fonction est complètement défini par rapport aux constructeurs d'une spécification algébrique en construisant un arbre de motifs et en testant la réductibilité inductive des feuilles par la réécriture contextuelle. Cette relation est un outil puissant car, par la réductibilité par cas, elle intègre le processus de raisonnement par cas. Cet arbre constitue en réalité l'*ensemble test* de termes dont il s'agit de vérifier la réductibilité, tel qu'il a été défini dans [51]. L'algorithme utilise un argument inductif pour traiter la complétude du calcul des préconditions aussi bien que pour traiter la complétude des autres définitions et ceci constitue un réel progrès par rapport aux traitements hiérarchiques. On peut voir à notre approche deux autres avantages, le premier est qu'elle n'interdit pas l'utilisation de relations entre les constructeurs, à condition qu'elles engendrent un système de réécriture canonique. Le second point fort est qu'elle tient compte du typage des opérateurs pour construire l'arbre de motifs et n'engendre pas de ce fait, de motifs incorrects.

Dans le cadre inconditionnel, diverses approches ont été élaborées parmi lesquelles nous pouvons citer celles de Guttag et Horning et celles d'Ehrig&al [41, 24]. La connexion entre les théories équationnelles et la réécriture de termes a été établie un peu plus tard, et les systèmes de réécriture se sont avérés être un outil performant pour vérifier les propriétés de complétude, de consistance et de correction des spécifications algébriques. Padawitz [85], a introduit la notion de *w-généricité* d'un prédicat défini par récurrence sur les constructeurs, qui exige que le système de réécriture soit linéaire à gauche. Cette restriction a été supprimée, par la suite, par Dershowitz [17], qui a implanté un algorithme fondé sur le calcul d'un ensemble test pour tester la réductibilité inductive d'un terme. Cet ensemble test est également calculé par Thiel [102], qui propose un algorithme utilisant un processus d'unification. Cependant, cet algorithme ne considère pas les relations entre les constructeurs. Les travaux de Thiel ont été repris par la suite par Lazrek [71], et avec Lescanne et Thiel [72], pour élargir le domaine d'application de la méthode des ensembles test par unification. Cette méthode permet à l'heure actuelle de traiter des systèmes de règles non linéaires à gauche, avec ou sans relations entre les constructeurs. Par ailleurs, Kounalis [66], a pour sa part élaboré des travaux dans lesquels les restrictions sur la forme syntaxique des règles de réécriture sont assouplies. Sa méthode est fondée sur le calcul d'arbres de motifs et elle constitue une amélioration appréciable de la méthode de Dershowitz. C'est la méthode que nous avons choisie d'étendre au cadre des spécifications conditionnelles.

Des améliorations de ce présent travail peuvent être envisagées. Il s'agit d'optimiser le test de recouvrement et de l'étendre de telle façon qu'il permette de traiter des classes plus larges de spécifications conditionnelles. Il serait par conséquent utile de combiner notre algorithme avec un prouveur inductif. Par exemple, si nous considérons la règle

$M = N \Rightarrow f(x_1, \ldots, x_n) \rightarrow 0$, nous savons que l'opérateur f est complètement défini si et seulement si l'équation $M = N$ est un théorème inductif de la spécification. Une autre extension serait, lors du test de réductibilité par cas, de réduire les préconditions de la même manière que l'on réécrit le terme, c'est-à-dire en utilisant la relation de réécriture contextuelle. En effet, dans ce présent travail, nous réduisons les contextes par la relation de réduction récursive, et il serait intéressant à l'avenir d'utiliser à cet effet plutôt la réécriture contextuelle et de ce fait, d'exploiter toute sa puissance sur les termes avec variables. D'autre part, nous avons montré dans les divers exemples que la méthode que nous avons établi a ses limitations, notamment lorsque la définition d'un opérateur est récursive. Il serait utile d'étudier la possibilité d'utiliser des méthodes fondées sur un principe d'induction structurelle, à l'exemple de Bidoit et Choquer, qui le font pour des preuves inductives dans des spécifications conditionnelles équationnelles [4].

Dès lors que l'on construit une spécification volumineuse $SPEC$ à partir d'une spécification plus simple $SPEC_0$ par des opérations dites d'*extensions* ou d'*enrichissements*, deux propriétés essentielles doivent être préservées, la complétude et la consistance. Ces propriétés garantissent que ni les supports ni les applications préalablement définis de $SPEC_0$ ne changent. Plus algébriquement, l'algèbre initiale de $SPEC_0$ est isomorphe à la restriction aux anciennes sortes et opérations, de l'algèbre initiale de $SPEC$.
Un type de spécifications algébriques, commode à définir par le mécanisme d'extensions ou d'enrichissements sont les spécifications hiérarchiques. L'objectif du chapitre suivant est la formulation d'un test de confluence sur les termes clos pour les systèmes de réécriture conditionnelle associés aux spécifications hiérarchiques. La consistance devient alors une propriété acquise dès lors que la confluence sur les termes clos est établie. Par contre, la complétude reste une condition nécessaire à la validation du test de confluence. Elle peut être vérifiée par l'algorithme présenté dans ce chapitre en utilisant la relation de réécriture hiérarchique appropriée. Le chapitre qui suit est consacré au cadre hiérarchique et à l'étude des différentes propriétés nécessaires pour établir la confluence sur les termes clos.

CHAPITRE 3

LES SPECIFICATIONS CONDITIONNELLES HIERARCHIQUES.

Chapitre 3

Les spécifications conditionnelles hiérarchiques.

1 Introduction

Nous abordons dans ce chapitre une approche hiérarchique de la réécriture conditionnelle qui consiste à définir une spécification par extensions successives. Cette méthode a pour but d'assurer la terminaison de l'évaluation récursive d'une précondition de règle au cours de la réécriture d'un terme. L'approche hiérarchique a été définie dans un premier temps par Pletat, Engels et Ehrich [93] puis formalisée par Rémy [94] et Zhang [105] dans un cadre de hiérarchie à deux niveaux. Les travaux de ces derniers étudient les systèmes conditionnels basés sur des systèmes primitifs comportant uniquement des règles inconditionnelles. Des résultats de confluence sur les termes clos sont établis et ont donné lieu à la réalisation d'un prototype baptisé REVEUR4. Ce logiciel a constitué le point de départ de nos travaux dans le cadre hiérarchique. Les diverses expérimentations auxquelles nous nous sommes livrée et les anomalies que nous avons pu constater nous ont motivée pour étudier de nouveau la théorie des spécifications conditionnelles hiérarchiques. Nous abordons par ailleurs cette étude avec un nombre arbitraire de niveaux de hiérarchie, contrairement aux travaux de Zhang [105]. Nous nous intéressons au modèle initial de la spécification et nous établissons des résultats de confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique. D'autre part, comme nous travaillons avec des préconditions booléennes, nous avons été amenée à définir le comportement d'une spécification conditionnelle par rapport à la sorte booléenne afin de garantir que l'algèbre initiale de la spécification soit *fidèle* par rapport aux booléens, c'est-à-dire qu'elle ne modifie en rien la sorte *booléen*, ni par ajout d'un quelconque élément, ni par confusion de $true$ avec $false$.

Ce chapitre concernant les spécifications hiérarchiques s'articule comme suit : nous définissons, dans la première partie, la relation de réduction récursive et la relation de réécriture hiérarchique sur les termes clos et nous rappelons que la propriété de complétude suffisante hiérarchique constitue la condition essentielle sous laquelle ces deux relations coïncident et calculent les mêmes formes normales closes. Le cadre d'étude hiérarchique dans lequel nous nous plaçons et la condition d'équivalence des deux relations de réécriture mettent alors en jeu différents concepts que nous abordons explicitement dans cette partie. Le second paragraphe est consacré à l'étude de la confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique. Cette étude est centrée sur la relation de réécriture hiérarchique. Cependant, nous définissons une relation de réécriture con-

textuelle afin de disposer d'un outil pour travailler sur les termes de $T(F, X)$. Dans une première partie, cette relation est formalisée dans un cadre hiérarchique afin de garantir la terminaison d'une réduction quelconque dans la partie précondition d'une règle et elle nous permet de travailler sur les termes avec variables. En effet, même si nous sommes intéressée par la réécriture sur les termes clos, nous avons besoin de travailler sur les termes avec variables. La relation de réécriture contextuelle hiérarchique dépend d'un contexte formé d'expressions booléennes et elle manipule des termes contextuels de la forme $(c :: t)$ où c est une formule booléenne. Les problèmes de terminaison nous ont également incitée à définir des systèmes de réécriture hiérarchiques. Enfin, nous établissons dans ce paragraphe le principal résultat de confluence sur les termes clos des systèmes hiérarchiques, avec un nombre arbitraire de niveaux de hiérarchie, généralisant en grande partie les travaux de Rémy et Zhang [107]. Le paragraphe qui suit comporte les preuves formelles des différents théorèmes énoncés. Le résultat de confluence sur les termes clos met en jeu un certain nombre de propriétés qui constituent les conditions d'applicabilité du théorème et que nous étudions dans le paragraphe suivant. Parmi ces propriétés, on peut citer la terminaison de la relation de réécriture contextuelle hiérarchique, la bonne couverture qui assure que toute réduction d'un terme contextuel est complète, garantissant de ce fait l'équivalence d'un terme $(c :: t)$ avec l'ensemble des termes engendrés à partir de $(c :: t)$ par simplification. Une troisième propriété nécessaire à la preuve de confluence est traduite par la complétude suffisante hiérarchique. Cette propriété garantit que la construction d'une spécification hiérarchique par paliers successifs est correcte et, combinée à la consistance relative d'une spécification hiérarchique, elle assure que le modèle initial de la spécification enrichie, restreint à la spécification de base, coïncide avec le modèle initial de la spécification de base. Enfin, la dernière propriété utile à la preuve de confluence met en jeu des preuves d'insatisfiabilité pour vérifier que deux contextes sont mutuellement exclusifs. L'exclusion mutuelle constitue un test essentiel pour montrer que deux ensembles de formes normales contextuelles sont équivalents. Par ailleurs, pour ces preuves d'insatisfiabilité, nous proposons dans le chapitre 4, des heuristiques aisées à mettre en œuvre dans un cadre spécifique. Pour toutes ces propriétés, tantôt nous suggérons des voies de recherche, tantôt nous établissons des conditions suffisantes de vérification dans un système de réécriture conditionnelle hiérarchique. Le dernier paragraphe comporte une série d'exemples présentés selon un aspect théorique pour illustrer nos résultats, et selon un aspect technique, enrichi par notre expérience avec REVEUR4. Nous concluons en résumant le travail effectué dans le cadre des spécifications hiérarchiques et en suggérant un certain nombre d'orientations que peut prendre ce travail.

2 Relations de réécriture sur les termes clos

La congruence syntaxique définie par un ensemble d'équations conditionnelles établit l'équivalence de deux termes par une chaîne de remplacements d'égaux par des égaux, chaque remplacement étant licite si la précondition associée a été prouvée récursivement équivalente à *true*. De même que pour la réécriture classique, nous nous heurtons au problème de l'indécidabilité de la congruence équationnelle. De manière analogue, pour se ramener à une situation plus déterministe, il s'agit d'orienter les équations conditionnelles en des règles de réécriture qui permettent alors de définir un système canonique. Cependant, d'un point de vue opérationnel, il n'est pas possible d'appréhender la théorie conditionnelle dans toute sa généralité. L'objectif visé dans ce paragraphe est d'étudier une classe de systèmes conditionnels dits hiérarchiques qui spécifient un type abstrait de données par niveaux imbriqués de définition. Nous étudions d'autre part deux relations de réécriture conditionnelle. La première, qui est récursive, est la plus générale. Elle pose cependant un problème de terminaison principalement dû à l'évaluation récursive de la précondition instanciée. La seconde relation est hiérarchique et consiste à restreindre la portée d'utilisation d'une règle. Cette relation, moins riche que la première lui est équivalente si l'on considère les termes clos et sous la condition de complétude suffisante.

Nous étudions dans ce chapitre les spécifications dont les préconditions sont des expressions de type booléen de la forme

$$\wedge_{i=1}^{n}\ \epsilon_i\ q_i,$$

où pour tout i dans $[1 \ldots n]$, ϵ_i est soit le symbole de négation soit le mot vide et q_i est un atome, ne comportant aucun connecteur logique. Construit uniquement à partir des connecteurs logiques $\wedge$ et $\neg$, le langage développé par les préconditions de ce type est un sous-ensemble de l'algèbre des termes de sorte booléenne et semble a priori restreint. Néanmoins, il est assez riche pour exprimer une forme normale disjonctive quelconque et d'autre part, une équation conditionnelle de la forme :

$$(p_1 \ \vee \ldots \vee \ p_n) \Rightarrow g = d$$

dans laquelle $\vee$ représente le connecteur logique de disjonction, est équivalent à l'ensemble suivant d'équations :

$$\{p_1 \Rightarrow g = d, \ldots,\ p_n \Rightarrow g = d\}$$

Dans le chapitre 1, nous avons défini une spécification conditionnelle comme la donnée d'une signature et d'un ensemble d'équations conditionnelles. Nous supposons de plus, qu'aucune sorte de la signature n'est vide. L'importance de cette condition a été mise en évidence en premier lieu par Goguen et Meseguer dans leurs travaux [38]. Ils montrent en effet que la déduction équationnelle n'est correcte que dans la classe des algèbres dont les sortes ne sont pas vides. Ceci implique en particulier qu'il existe au moins un terme clos de chaque sorte.
La proposition qui suit établit une condition assurant qu'une sorte s dans une signature quelconque est vide :

Proposition 3.1 [38]
Une sorte s est vide dans une signature $\Sigma = (S,\ F)$ ssi

1. *il n'existe pas de constante de sorte s dans S*

2. *pour tout symbole de fonction dans* F *de profil* $s_1 \times \ldots \times s_n \rightarrow s$ *et pour tout indice* i *dans* $[1 \ldots n]$, s_i *est vide.*

Etant donné que nous étudions des équations conditionnelles dont les préconditions sont des formules booléennes, toute spécification algébrique est supposée contenir une spécification de la sorte booléenne suffisamment complète pour constituer un environnement de calcul booléen. Cette hypothèse est, d'un point de vue opérationnel, difficile à réaliser. Il n'existe pas en effet de système canonique pour le calcul des prédicats. Cependant, en utilisant les connecteurs logiques $+$ (ou exclusif) et $\wedge$ (symbole de conjonction), Hsiang a formalisé un système nœthérien et confluent modulo les équations de commutativité et d'associativité de $\wedge$ et $+$. Ainsi, une expression booléenne calculée dans ce système a une forme normale unique et deux expressions booléennes sont équivalentes si elles ont la même forme normale. Ce système booléen de Hsiang a été utilisé dans l'implantation du logiciel REVEUR4 et il sera davantage explicité par la suite au cours de la présentation du prototype.

La sémantique d'une équation conditionnelle est que si p est vrai alors g et d sont égaux. L'ensemble des congruences sur $T(F, X)$ validant une famille E d'équations conditionnelles est un treillis complet qui admet une congruence minimale $=_E$. Cette congruence peut être explicitement construite en utilisant la théorie du point fixe [94]. On obtient également un résultat de complétude *à la Birkhoff* que le lecteur trouvera énoncé au chapitre 1.

Une spécification conditionnelle SP est considérée comme un système de réécriture si chaque équation dans E est orientée en règle de réécriture. Dans la suite, on note plus simplement par la suite un système de réécriture par R.

Nous étudions dans ce paragraphe des conditions pour que, dans un cadre conditionnel, le processus de réécriture et la déduction équationnelle aient la même puissance de preuve. Nous rappelons auparavant les notions usuelles utilisées dans la méthode de réécriture à savoir, les propriétés de confluence et de terminaison d'un système de réécriture. Cependant, étant donné que nous étudions plusieurs relations de réécriture, nous attacherons ces notions de confluence et de terminaison non pas à un système de réécriture, mais à la relation de réécriture elle-même.

Un premier problème évoqué au premier chapitre et lié au cadre conditionnel consiste à assurer non seulement la terminaison inconditionnelle du terme à réécrire mais aussi celle de l'évaluation récursive de la précondition. La définition du préordre $<$ sur les entiers relatifs $\mathcal{Z}$ est un exemple trivial de non-terminaison de cette évaluation.

Exemple 3.1 *Soit* (S, F, E) *la spécification suivante avec*
$S = \{\mathcal{Z}\}$;
$F = \{0 :\rightarrow \mathcal{Z},\ pred,\ succ : \mathcal{Z} \rightarrow \mathcal{Z}\}$;
et R *constitué des règles :*

1. $pred(succ(x)) \rightarrow x$
2. $0 < 0 \rightarrow false$
3. $0 < succ(x) \rightarrow true$
4. $x < pred(y) \Rightarrow x < y \rightarrow true$

Soit une constante a de sorte $\mathcal{Z}$ et différente de 0. Nous nous proposons de tester si a est une constante positive en évaluant le terme $0 < a$. Cette évaluation produit une branche

infinie de réécritures :

$0 < a \rightarrow_R true$ *si* $0 < pred(a) \rightarrow_R^* true$,
$0 < pred(a) \rightarrow_R true$ *si* $0 < pred(pred(a)) \rightarrow_R^* true$,
$0 < pred(pred(a)) \rightarrow_R true$ *si* ... $\diamond$

Nous développons dans la suite une approche de la réécriture conditionnelle adaptée dans le but d'assurer que toute réduction conditionnelle termine. L'idée consiste à définir des systèmes de réécriture hiérarchiques de telle façon que la conséquence et la précondition d'une même règle appartiennent à des sous-systèmes différents, évitant ainsi de produire des cycles au cours d'une réduction. L'utilisation d'une spécification volumineuse est en outre plus aisée car la définir de façon hiérarchique permet un traitement modulaire par niveau de hiérarchie. Nous travaillons dorénavant avec des spécifications structurées comme elles ont été définies dans le chapitre précédent et l'on parlera d'une spécification $(S,\ F,\ E)$ structurée de base $(S,\ BF,\ BE)$.
Rappelons que BF contient l'ensemble C des constructeurs de la spécification et $F \ominus BF$ l'ensemble des opérations définies. Les termes construits uniquement à partir des constructeurs (resp des symboles de BF) sont dits termes primitifs (resp termes de base).

La hiérarchie de spécifications utilise le concept de construction d'une spécification par extensions ou enrichissements successifs. Ces notions d'extension et d'enrichissement jouent un rôle important dans le mécanisme de construction des spécifications algébriques car elles permettent de construire des spécifications volumineuses à partir de spécifications plus simples. En partant d'une spécification primitive, on peut ajouter soit des opérations et des équations, il s'agit alors d'un enrichissement, soit également des sortes nouvelles et on parlera d'extension.

Définition 3.1 Extension, enrichissement

Une spécification $(S,\ F,\ E)$ *est une extension d'une spécification* $(S_0,\ F_0,\ E_0)$ *si* $S_0,\ F_0,\ E_0$ *sont respectivement inclus dans* $S,\ F,\ E$.
Un enrichissement est une extension laissant l'ensemble des sortes inchangé.

Dans la suite, $\ominus$ et $\oplus$ désignent respectivement la différence ensembliste et l'unions disjointe d'ensembles.

Définition 3.2 Spécification hiérarchique

Une spécification à $(n+1)$ *niveaux,* $SP = (SP_0, \ldots,\ SP_n)$ *est une famille de spécifications conditionnelles* $SP_i = (S_i,\ F_i,\ E_i)$ *telles que pour tout* i, $0 < i \leq n$,

1. SP_i *est soit une extension soit un enrichissement de* SP_{i-1} ;
2. $(S_i,\ F_i,\ E_i)$ *est une spécification structurée de base* $(S_i,\ BF_i,\ BE_i)$;
3. *si* $p \Rightarrow g = d \in E_i \ominus E_{i-1}$ *alors* p *est un terme de* $T(F_{i-1},\ X)$, $g,\ d \in T(F_i, X)$ *et soit* g *soit* d *est dans* $T(F_i,\ X) \ominus T(F_{i-1},\ X)$

Dans le cas d'une extension, pour tout niveau i, BF_i est constitué de F_{i-1} auquel on ajoute un ensemble de constructeurs ayant tous une nouvelle sorte, c'est-à-dire une sorte du niveau i, dans leur profil ; soit

$BF_i = F_{i-1} \oplus \{\text{constructeurs du niveau } i\}$.

Toujours dans ce cas, BE_i contient E_{i-1} et un ensemble d'équations appelées relations entre les constructeurs du niveau i, chacune d'elles mettant en jeu ces constructeurs ; soit

$BE_i = E_{i-1} \oplus \{\text{relations entre les constructeurs du niveau } i\}$.

A partir de cette définition, on convient de construire une spécification hiérarchique de la façon suivante:

- Si on étend la spécification, alors toute nouvelle sorte introduite possède un ensemble de constructeurs;
- On introduit les constructeurs d'un niveau en une seule étape, par une extension du niveau inférieur;

Ces deux conventions peuvent être exprimées formellement par la condition suivante: si $S_i \neq S_{i-1}$ alors $BF_i \neq F_{i-1}$ (il s'agit d'une extension), et si $S_i = S_{i-1}$ alors $BF_i = F_{i-1}$ (c'est un enrichissement).

Définition 3.3 Système de réécriture conditionnelle hiérarchique ($SRCH$)
Un $SRCH$ $R = (R_0, \ldots, R_n)$ est associé à une spécification hiérarchique. Plus formellement,

1. *pour tout i dans $[0 \ldots n]$, (S_i, F_i, R_i) est un système de réécriture structuré de base (S_i, BF_i, BR_i),*
2. *si $p \Rightarrow g \rightarrow d \in R_i \ominus R_{i-1}$ alors p est dans $T(F_{i-1}, X)$, $d \in T(F_i, X)$ et g est un terme de $T(F_i, X) \ominus T(F_{i-1}, X)$, $\forall i,\ 0 < i \leq n$.*

Par convention, l'ensemble $T(F_{-1}, X)$ est vide. Par conséquent, les règles de R_0 sont inconditionnelles.

Dans une spécification hiérarchique, on attribue à tout opérateur, toute règle, tout terme un niveau de hiérarchie. On dit que:

- un opérateur f est de niveau i si $f \in F_i \ominus F_{i-1}$.
- un terme t est de niveau i si t est dans $T(F_i, X) \ominus T(F_{i-1}, X)$.
- une règle est de niveau i si son membre gauche est un terme de niveau i.

Dans ce qui suit, nous supposons que toute spécification hiérarchique est construite à partir d'une extension de la spécification des booléens. Nous faisons de plus l'hypothèse suivante:

$$\forall f : s_1 \times \ldots \times s_n \rightarrow s,\ f \in F_n \ominus F_{bool} \Longrightarrow \forall i \in [1 \ldots n],\ s_i \neq bool.$$

Cette hypothèse signifie que les opérations de sorte booléenne sont vues comme des prédicats.

Lorsqu'il n'est pas utile de préciser le nombre total n de niveaux de hiérarchie, on convient de noter l'ensemble F_n par F, R_n par R et SP_n par SP.

Exemple 3.2 *On définit une spécification hiérarchique des listes ordonnées d'entiers naturels ;*

$SP_0 = SP_{bool}$;

*** * * * * Extension * * * * ***

$SP_1 = (S_1,\ F_1,\ E_1)$;
$S_1 = S_0 \oplus \{nat\},\ F_1 = BF_1 \oplus DF_1$;
$BF_1 = F_0 \oplus \{0 :\rightarrow nat,\ succ : nat \rightarrow nat\},\ DF_1 = \{\leq : nat \times nat \rightarrow bool\}$;
$E_1 = BE_1 \oplus DE_1,\ BE_1 = E_0$;
$DE_1 = \{0 \leq x = true,\ succ(x) \leq 0 = false,\ succ(x) \leq succ(y) = x \leq y\}$.

*** * * * * Extension * * * * ***

$SP_2 = (S_2,\ F_2,\ E_2),\ S_2 = S_1 \oplus \{list\},\ F_2 = BF_2 \oplus DF_2$;
$BF_2 = F_1 \oplus \{[\,] :\rightarrow list,\ cons : nat \times list \rightarrow list\}$;
$DF_2 = \{insert : nat \times list \rightarrow list\}$;
$E_2 = BE_2 \oplus DE_2,\ BE_2 = E_1$;
$DE_2 = \{insert(x,\ [\,]) = cons(x,\ [\,]),$
$\quad x \leq y \Rightarrow insert(x,\ cons(y,\ u)) = cons(x,\ cons(y,\ u)),$
$\quad \neg(x \leq y) \Rightarrow insert(x,\ cons(y,\ u)) = cons(y,\ insert(x,\ u))\}$.

*** * * * * Enrichissement * * * * ***

$SP_3 = (S_3,\ F_3,\ E_3),\ S_3 = S_2,\ F_3 = BF_3 \oplus DF_3,\ BF_3 = F_2$;
$DF_3 = \{ordered : list \rightarrow bool\}$;
$E_3 = BE_3 \oplus DE_3,\ BE_3 = E_2$;
$DE_3 = \{ordered([\,]) = true,$
$\quad ordered(cons(x,\ [\,])) = true,$
$\quad x \leq y \Rightarrow ordered(cons(x,\ cons(y,\ u))) = ordered(cons(y,\ u)),$
$\quad \neg(x \leq y) \Rightarrow ordered(cons(x,\ cons(y,\ u))) = false,$
$\quad ordered(insert(x,\ u)) = ordered(u)\}$. ◇

Outre la relation de réduction récursive, nous pouvons définir une relation de réécriture hiérarchique. La construction hiérarchique du système de réécriture permet de façon naturelle, de définir récursivement la relation hiérarchique. De ce fait, cette relation est étendue au niveau i de hiérarchie en supposant qu'elle est correctement définie au niveau $i-1$:

Définition 3.4 Relation de réécriture hiérarchique

Soient $R = (R_0, \ldots,\ R_n)$ un SRCH et t, t' deux termes clos ; pour tout i dans $[0 \ldots n]$, on définit $\rightarrow_{H,R_i}$ par $t \rightarrow_{H,R_i} t'$ ssi

- *soit $t \rightarrow_{H,R_{i-1}} t'$*
- *soit il existe une règle $p \Rightarrow g \rightarrow d$ dans $R_i \ominus R_{i-1}$, une occurrence u de t et une substitution σ tels que*

$$t_{/u} = \sigma g,\ t' = t[u \leftarrow \sigma d],\ \sigma p \rightarrow^*_{H,R_{i-1}} true \text{ et } \sigma p \in T(F_{i-1}).$$

La relation de réécriture hiérarchique $\rightarrow_{H,R}$ est définie par $\rightarrow_{H,R_n}$.

Remarques : La condition que σp soit dans $T(F_{i-1})$ est nécessaire pour que σp soit évaluable par les règles de R_{i-1}.

Il est important de noter que si t est un terme clos dans $T(F_i) \ominus T(F_{i-1})$, t peut seulement être réécrit par des règles dont le niveau de hiérarchie est inférieur ou égal à i.

Définition 3.5 Congruence hiérarchique

La congruence spécifiée par la suite $SP = (SP_0, \ldots, SP_n)$ *est celle engendrée par* SP_n.

La congruence hiérarchique engendrée par une spécification conditionnelle hiérarchique, notée $\equiv_{H,SP}$ *est la dernière composante* $\equiv_{SP_n}$ *de la plus petite suite de congruences* $(\equiv_{SP_0}, \ldots, \equiv_{SP_n})$ *définie sur* $T(F, X)$ *et satisfaisant la condition :*

$$\bar{\sigma}p \equiv_{H,SP_{i-1}} true \Longrightarrow \bar{\sigma}g \equiv_{H,SP_i} \bar{\sigma}d$$

$\forall i,\ 0 < i \leq n,\ \forall p \Rightarrow g \rightarrow d \in E_i,\ \forall \sigma : Var(g) \rightarrow T(F_i, X)$ *telle que* σp *soit un élément de* $T(F_{i-1}, X)$ *et* σg *un élément de* $T(F_i, X)$. $\bar{\sigma}$ *est l'unique* F*-homomorphisme (à un isomorphisme près) qui étend* σ *à l'ensemble* $T(F, X)$.

Exemple 3.3 *Soit* $R = (R_0, \ldots, R_3)$ *le SRCH suivant :*

$SP_0 = SP_{bool}$;

$S_1 = S_0 \oplus \{elem\}$;
$F_1 = F_0 \oplus \{q : elem \rightarrow bool,\ a :\rightarrow elem\}$;
$R_1 = R_0 \oplus \{1.\ q(a) \rightarrow true\}$.

$S_2 = S_1$;
$F_2 = F_1 \oplus \{h : elem \rightarrow elem\}$;
$R_2 = R_1 \oplus \{2.\ h(a) \rightarrow a\}$.

$S_3 = S_2$;
$F_3 = F_2 \oplus \{b :\rightarrow elem,\ f : elem \times elem \rightarrow elem\}$;
$R_3 = R_1 \oplus \{3.\ q(b) \rightarrow q(a),\ 4.\ q(x) \Rightarrow f(x, y) \rightarrow x\}$.

Soit le terme $f(h(a), b)$; par la règle 4 de R_3, $f(h(a), b) \rightarrow_{H,R} h(a)$ si $q(h(a)) \rightarrow^*_{H,R} true$ et si $q(h(a)) \in T(F_2)$.
On peut aisément vérifier la correction de la réduction suivante :

$$q(h(a)) \rightarrow_{H,R} q(a) \rightarrow_{H,R} true. \diamond$$

Dans un système de réécriture conditionnelle hiérarchique, il est en fait possible de définir au moins trois sortes de relation de réécriture chacune d'elles comportant, dans sa définition, une restriction ayant trait à la hiérarchie du système.

1. Une première relation que nous pouvons définir est celle que nous avons présentée ci-dessus. Cette relation a pour particularité de prendre en compte la hiérarchie aussi bien au niveau du système utilisé que dans sa définition même, ceci par la condition concernant le domaine d'appartenance de la précondition instanciée. Notons cette relation $\rightarrow_{R,F}$.

2. Une seconde relation qu'il est possible de définir et que nous notons $\rightarrow_{R,G}$, est une relation de réécriture tenant compte uniquement de la structure hiérarchique du système de réécriture. Si un terme est réécrit par une règle r de niveau i, alors la précondition instanciée de r est évaluée par des règles de niveau strictement inférieur à i.

3. La troisième relation est définie de manière aussi générale que la réduction récursive excepté qu'une borne maximale, correspondant au nombre de règles utilisées, est imposée au cours de la justification de la précondition. De cette façon, toute dérivation infinie est évitée. Cette borne a pour valeur le niveau de hiérarchie de la règle utilisée pour réécrire le terme. Notons cette relation $\rightarrow_{R,K}$.

Ces trois types de relation de réécriture hiérarchique ont été étudiés de manière approfondie par Navarro dans sa thèse [79]. Ces relations n'ont pas toutes la même portée d'utilisation. Plus formellement, Navarro a démontré les inclusions suivantes :

- $\rightarrow_{R,F} \subseteq \rightarrow_{R,G}$ et $\rightarrow_{R,G} \not\subseteq \rightarrow_{R,F}$.
- $\rightarrow_{R,G} \subseteq \rightarrow_{R,K}$ et $\rightarrow_{R,K} \not\subseteq \rightarrow_{R,G}$.
- $\longleftrightarrow_{R,K} \not\equiv \longleftrightarrow_{R}$.

Nous choisissons d'étudier, dans ce présent travail, la relation de réécriture $\rightarrow_{R,F}$ que nous notons $\rightarrow_{H,R}$. Il serait intéressant d'étudier à l'avenir, les conditions d'applicabilité de nos résultats de confluence sur les termes clos pour les deux autres types de relation hiérarchique.

La relation de réduction hiérarchique est définie de telle manière qu'une règle quelconque ne peut être utilisée à la fois pour réécrire un terme et pour réduire la précondition correspondante à *true*. La raison en est que ces deux réécritures se font dans des systèmes différents, l'évaluation de la précondition étant effectuée dans un système plus petit. C'est en fait ce procédé qui garantit la terminaison d'une dérivation hiérarchique. Toutefois, ceci entraîne que la congruence hiérarchique est contenue dans la congruence induite par la réduction récursive mais nous pouvons affirmer qu'elle ne lui est pas équivalente. L'exemple qui suit montre qu'en général, les deux congruences sont différentes.

Exemple 3.4 *Soit SP une spécification conditionnelle telle que*

$SP_0 = SP_{bool}$;

$S_1 = S_0 \oplus \{s\},\ F_1 = F_0 \oplus \{0 : \rightarrow s,\ a : s \rightarrow s,\ p : s \rightarrow bool\}$;
$E_1 = R_0 \oplus \{1.\ p(a(x)) = true\}$;

$S_2 = S_1,\ F_2 = F_1 \oplus \{q : s \rightarrow s,\ 1 :\rightarrow s\}$;
$E_2 = R_1 \oplus \{2.\ p(x) \Rightarrow q(x) = 0\}$.

Les deux congruences ne coïncident pas puisque

$$q(a(1)) \equiv_{SP} 0 \text{ mais } q(a(1)) \not\equiv_{H,SP} 0.$$

En effet, le terme $p(a(1))$ n'appartient pas à $T(F_1)$ et il est alors illicite d'appliquer l'équation 1 à $p(a(1))$. $\diamond$

Un résultat a cependant été prouvé dans [80] établissant l'équivalence de la congruence récursive et de la congruence hiérarchique sur les termes clos sous une condition dite de *complétude suffisante hiérarchique*. Cette propriété des spécifications hiérarchiques est à l'heure actuelle difficile à vérifier en pratique. Toutefois, elle est étroitement liée au concept de complétude de définition d'un opérateur par rapport à un ensemble de constructeurs et il est possible de ce fait, d'obtenir des conditions syntaxiques suffisantes pour établir la complétude suffisante hiérarchique d'une spécification. Ce résultat d'équivalence a en fait

constitué la motivation première qui nous a amenée à étudier la propriété de complétude suffisante en premier lieu dans un cadre non-hiérarchique, puis pour des spécifications conditionnelles hiérarchiques. Nous énonçons dans ce qui suit, le théorème d'équivalence entre les congruences induites par la réduction récursive et par la réduction hiérarchique sur les termes clos.

Définition 3.6 Complétude suffisante d'une spécification hiérarchique
Une spécification hiérarchique SP est suffisamment complète ssi

$$\forall i,\ 0 < i \leq n,\ \forall t \in T(F_i),\ \exists t' \in T(F_{i-1})\ \textit{tel que}\ t \equiv_{SP} t',$$

si la sorte de t est dans S_{i-1}.
Un $SRCH$ R est opérationnellement suffisamment complet si pour un tel terme t, $\exists t' \in T(F_{i-1})$ tel que $t \rightarrow_R^* t'$.
La seconde notion est plus simplement appelée **convertibilité.**

Théorème 3.1 [80]
Soit SP une spécification hiérarchique et soient $\longleftrightarrow_{H,R}^*$ et $\longleftrightarrow_R^*$ les fermetures réflexives symétriques et transitives correspondant respectivement aux restrictions de la relation de réécriture hiérarchique et de la relation de réduction récursive aux termes clos ;
si $\rightarrow_R$ est nœthérienne et confluente, et si SP est suffisamment complète,

$$\longleftrightarrow_{H,R}^* \text{ et } \longleftrightarrow_R^* \text{ sont équivalentes sur les termes clos.}$$

De plus, $\rightarrow_{H,R}$ est nœthérienne et confluente et pour tout terme t, les formes normales de t par les relations $\rightarrow_R$ et $\rightarrow_{H,R}$ coïncident, soit

$$\bar{t}_{H,R} = \bar{t}_R.$$

Ceci entraîne en particulier, que le $SRCH$ associé à SP est convertible.

La propriété de complétude est en général indécidable. Nous avons établi dans le chapitre 2 un algorithme de test de la complétude dans le cadre de spécifications conditionnelles sans hypothèse de hiérarchie (voir page 54). Par conséquent, a un niveau de hiérarchie i, on peut tester la complétude suffisante d'un système structuré $(S_i,\ F_i,\ R_i)$ par rapport à un système de base $(S_i,\ BF_i,\ BR_i)$. Cette technique est également applicable à des spécifications hiérarchiques puisqu'on peut aussi tester pour tout $i, 0 \leq i < n$, la complétude suffisante d'une extension $(S_{i+1},\ F_{i+1},\ R_{i+1})$ par rapport à une spécification $(S_i,\ F_i,\ R_i)$.

Dans le cadre conditionnel, des travaux ont été effectués par Zhang [105] pour tester la complétude de définition d'un opérateur défini par rapport à une spécification primitive. Ce travail a été approfondi par Rémy et Uhrig [104]. La situation est rendue plus complexe par la présence des préconditions. De plus, un autre problème délicat se pose dans le contexte conditionnel. En effet, l'hypothèse de validité du théorème 3.1 est encore trop forte du fait que l'on suppose que le système soit déjà canonique. La confluence sur les termes clos et la complétude suffisante deviennent alors des propriétés inter-dépendantes et il devient impératif d'affaiblir les conditions d'applicabilité du théorème. Pour ce faire, nous définissons la propriété de complétude d'une spécification hiérarchique non plus en fonction de la relation $\rightarrow_R$ mais en fonction de la relation hiérarchique $\rightarrow_{H,R}$. On obtient alors un lemme d'équivalence entre les deux congruences ne nécessitant plus d'avoir pour hypothèse de validité la canonicité du système de réécriture considéré.

Définition 3.7 Complétude suffisante hiérarchique, convertibilité hiérarchique
Une spécification hiérarchique SP vérifie la propriété de complétude suffisante hiérarchique si,

$$\forall i,\ 0 < i \leq n,\ \forall t \in T(F_i),\ \exists t' \in T(F_{i-1})\ \textit{tel que}\ t \equiv_{H,SP} t',$$

si la sorte de t est dans S_{i-1}.
Un $SRCH$, R est **hiérarchiquement convertible** *si $t \rightarrow^*_{H,R} t'$.*

Théorème 3.2 [80]
Soit SP une spécification hiérarchique et soient $\longleftrightarrow^*_{H,R}$ et $\longleftrightarrow^*_R$ les fermetures réflexives, symétriques et transitives correspondant respectivement aux restrictions de $\rightarrow_{H,R}$ et $\rightarrow_R$ aux termes clos;
si SP vérifie la propriété de complétude suffisante hiérarchique, alors

$$\longleftrightarrow^*_{H,R} \text{ et } \longleftrightarrow^*_R \text{ sont équivalentes sur les termes clos.}$$

De même, soit R le $SRCH$ associé à SP, si R est hiérarchiquement convertible, les relations $\rightarrow_R$ et $\rightarrow_{H,R}$ coïncident sur les termes clos et elles calculent les mêmes formes normales.

Pour la propriété de complétude suffisante, nous considérons l'ensemble des termes clos parce qu'il n'est pas envisageable de supposer les propriétés de complétude suffisante ou de complétude suffisante hiérarchique sur les termes avec variables. La complétude suffisante d'une spécification hiérarchique est l'une des propriétés essentielles qui garantissent que la construction d'une spécification hiérarchique est correcte. La correction est assurée si les extensions et les enrichissements définis ne changent ni les supports ni les applications préalablement définis. Plus algébriquement, l'algèbre initiale de départ doit être isomorphe à la restriction de la nouvelle algèbre initiale aux anciennes sortes et opérations. La seconde propriété importante qui assure cette correction, est la consistance relative d'une spécification hiérarchique définie comme suit :

Définition 3.8 Consistance relative d'une spécification hiérarchique
Une spécification hiérarchique SP est relativement consistante ssi

$$\forall i \in [1 \ldots n],\ \forall t,\ t' \in T(F_{i-1}),\ t \equiv_{SP_i} t' \Longrightarrow t \equiv_{SP_{i-1}} t'.$$

En d'autres termes, $\equiv_{SP_i} \cap\, T(F_{i-1})^2 = \equiv_{SP_{i-1}}$.

D'un point de vue opérationnel, pour étudier la confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique, nous avons besoin de travailler sur les termes avec variables. Afin d'élargir le domaine d'application de la réécriture conditionnelle à l'ensemble $T(F, X)$, nous définissons une relation de réécriture contextuelle qui permet en particulier de raisonner par cas. Pour cette relation, nous définissons des *ensembles de formes normales contextuelles* et nous établissons des résultats de confluence et de terminaison sous des conditions spécifiques. Ces ensembles de formes normales contextuelles fournissent une méthode effective pour prouver la convergence de paires critiques dites *contextuelles*. Cependant, le raisonnement par cas défini par la relation de réécriture contextuelle n'est pas lié au cadre hiérarchique. Il peut être abordé dans une approche de système décroissant. Cette étude est largement explicitée dans les chapitres qui suivent.

3 Confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique, énoncé des principaux résultats

L'objectif de ce paragraphe est d'appliquer le théorème classique de Knuth-Bendix [64] à la théorie conditionnelle pour établir la confluence des systèmes conditionnels. L'étude concerne les spécifications conditionnelles à plusieurs niveaux de hiérarchie, étendant par là-même les travaux effectués à Nancy. Pour des systèmes à deux niveaux de hiérarchie, les résultats théoriques ont été implantés dans REVEUR4 [95, 105, 107] et ont donné lieu à des expérimentations fructueuses.

Le principal apport de cette partie est un résultat de confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique qui permet de procéder à une vérification modulaire de cette confluence niveau par niveau. D'autre part, l'existence d'une algèbre initiale *fidèle* par rapport aux booléens est garantie. La propriété de fidélité par rapport aux booléens se traduit par la *consistance* et la *complétude* par rapport à la sorte booléenne que nous définissons dans la partie suivante.

Notation : Soient t_1, t_2 deux termes de $T(F, X)$; s'il existe un terme t tel que $t_1 \rightarrow_R^* t$ et $t_2 \rightarrow_R^* t$ alors on note $t_1 \downarrow_R t_2$.

La proposition qui suit prouve que la consistance relative d'une spécification hiérarchique peut être induite à partir du système de réécriture qui lui est associé.

Proposition 3.2 *Soit R un SRCH hiérarchiquement convertible ;*
si $\rightarrow_R$ est confluente sur les termes clos, la spécification SP associée à R est relativement consistante.

Preuve : Soient t_1, t_2 deux termes clos dans $T(F_i)$ tels que $t_1 \equiv_{SP_{i+1}} t_2$;
puisque $t_1 \equiv_{SP_{i+1}} t_2$, $t_1 \equiv_{SP} t_2$ puisque $\equiv_{SP_{i+1}} \subseteq \equiv_{SP}$; donc $t_1 \downarrow_R t_2$ puisque $\rightarrow_R$ est confluente sur les termes clos ; par conséquent, $t_1 \downarrow_{H,R} t_2$ par l'hypothèse de convertibilité hiérarchique ; donc $t_1 \downarrow_{R_i} t_2$ puisque t_1, t_2 ne peuvent pas être réécrits par le système $R \ominus R_i$; donc $t_1 \equiv_{SP_i} t_2$. □

Dans l'étude des types abstraits, ce résultat est essentiel puisque, comme nous le verrons par la suite, la confluence sur les termes clos est la propriété utilisée pour démontrer la consistance d'une spécification.

D'autre part, un résultat important a été établi par Navarro [79]. Ce résultat fournit une méthode de test de la confluence et de la terminaison de la relation hiérarchique $\rightarrow_{H,R}$ par niveaux successifs de hiérarchie. Il s'énonce comme suit :

Lemme 3.1 **[79]**
Soit $R = (R_0, \ldots, R_n)$ un SRCH ;
$\rightarrow_R$ est confluente et nœthérienne ssi pour tout i dans $[0 \ldots n]$, $\rightarrow_{R_i}$ est confluente et nœthérienne.

3.1 Relation de réécriture contextuelle hiérarchique

Les deux notions de réécriture précédemment définies, à savoir $\rightarrow_R$ et $\rightarrow_{H,R}$, nous permettent de travailler seulement sur les termes clos. En effet, pour toute substitution

close σ, nous supposons que σp se réduit soit en *true* soit en $false$. Ceci n'est usuellement pas le cas lorsqu'on travaille sur les termes avec variables. Afin d'élargir le domaine d'application de la réécriture conditionnelle hiérarchique aux termes avec variables, nous définissons une relation de réécriture sur des termes dits contextuels, appelée *réécriture contextuelle hiérarchique.* Un terme contextuel ou c-terme $(c :: t)$ est une paire de termes où c est une expression booléenne constituant le contexte du c-terme, et dont la syntaxe est similaire à celle des préconditions de règles. En d'autres termes, c est de la forme suivante :

$$c = \wedge_{i=1}^{n} \epsilon_i\ c_i$$

tel que pour chaque i, ϵ_i est soit le mot vide, soit le symbole de négation $\neg$, et c_i est une expression booléenne quelconque, sans aucun connecteur logique.

Exemple 3.5 *Si nous considérons de nouveau l'exemple 3.2,*

- $(\neg(x \leq y) :: insert(x,\ cons(y,\ u)))$ *et*
- $(a \leq b :: ordered(cons(a,\ cons(b,\ cons(c,\ [\ ])))))$

sont des termes contextuels. $\diamond$

Cette relation de réécriture contextuelle peut, plus généralement, être définie sans hypothèse de hiérarchie. Elle correspond alors à l'extension de la réduction récursive aux termes avec variables. Néanmoins, pour assurer sa terminaison, nous préférons en définir une version hiérarchique, formalisée à partir de la relation de réécriture hiérarchique.

Définition 3.9 Relation de réécriture contextuelle hiérarchique
Soit $R = (R_0, \ldots,\ R_n)$ *un* $SRCH$ *;*$(c :: t) \xrightarrow{c}_{H,R} (c' :: t')$*, s'il existe un indice* i *dans* $[0 \ldots n]$*, une règle* $p \Rightarrow g \rightarrow d$ *dans* $R_i \ominus R_{i-1}$ *et une substitution* σ *tels que :*

1. $\sigma p \in T(F_{i-1},\ X)$ *et*

   - *soit il existe une occurrence* u *de* t *telle que*

     $$t_{/u} = \sigma g,\ c' = c\ \wedge\ \sigma p,\ t' = t[u \leftarrow \sigma d]\,;$$

     dans ce cas, il s'agit d'une réécriture de terme et on note
     $(c :: t) \xrightarrow{c}_{H,R,t} (c' :: t')$ *;*
   - *soit il existe une occurrence* ω *dans* c *telle que*

     $$c_{/\omega} = \sigma g,\ c' = c[w \leftarrow \sigma d]\ \wedge\ \sigma p,\ t' = t\,;$$

     dans ce cas, il s'agit d'une réécriture de contexte et on note
     $(c :: t) \xrightarrow{c}_{H,R,c} (c' :: t')$ *;*

$\xrightarrow{c}{}^{*}_{R}$ (resp $\xrightarrow{c}{}^{+}_{R}$) est la fermeture transitive et réflexive (resp fermeture transitive) de la relation $\xrightarrow{c}_{R}$.
si $(c :: t) \xrightarrow{c}_{R} (c' :: t')$ en m étapes de réécriture, on écrit $(c :: t) \xrightarrow{c}{}^{m}_{R} (c' :: t')$.

Remarque : Le traitement de la partie terme et de la partie contexte est similaire. Les deux types de réécriture sont toutefois distincts puisqu'en particulier, la partie contexte est enrichie différemment dans chacun des deux cas.

3.2 Résultats de confluence sur les termes clos

Après avoir défini certaines propriétés nécessaires pour établir le principal résultat de confluence, nous présentons dans cette partie une preuve formelle et détaillée de ce résultat. La partie suivante consiste essentiellement à justifier les divers résultats intermédiaires et à étudier de manière plus approfondie les différentes notions intervenant dans les preuves.

Quelques propriétés utiles

Définition 3.10 Propriété de bonne couverture opérationnelle
Un SRCH, R est opérationnellement bien couvert ssi pour chaque sous-ensemble maximal de règles de la forme $\{p_i \Rightarrow g \rightarrow d_i,\ i \in I\}$ *dans R,*

$$\forall \sigma \in Subst(F),\ \exists i \in I \text{ tel que } \sigma p_i \rightarrow^*_{H,R} true.$$

Définition 3.11 Forme normale contextuelle
Un c-terme $(c :: t)$ *est en forme normale contextuelle ssi* $c \neq_R false$ *et s'il n'existe aucun terme contextuel* $(c' :: t')$ *tel que*

$$(c :: t) \xrightarrow{c}_{H,R} (c' :: t')$$

Ceci implique en particulier que si un terme contextuel $(c :: t)$ peut seulement être réécrit en des c-termes $(c' :: t')$ tels que $c' =_R false$, $(c :: t)$ est considéré comme une forme normale contextuelle. En effet, des réductions de ce type sont inutiles puisque, d'un point de vue déductif, comme une conséquence du lemme 3.2 établi dans la suite, $(c :: t) \xrightarrow{c}_{H,R} (c' :: t')$ tel que $c =_R false$ est équivalent à établir $t = t'$ dans le contexte $false$. De telles réductions sont dites triviales et elles ne seront pas considérées par la suite. Par analogie, un c-terme $(c :: t)$ est dit trivial si $c =_R false$.

L'existence des formes normales contextuelles pour chaque terme contextuel est garantie par la terminaison contextuelle.

Définition 3.12 Ensemble de formes normales contextuelles (*EFNC*)
Un ensemble de formes normales contextuelles d'un c-terme $(c :: t)$, *noté* $EFNC((c :: t))$, *est un ensemble*

$$\{(c_i :: t_i),\ i \in I\}$$

tel que pour chaque i dans I, $(c_i :: t_i)$ *est une forme normale contextuelle de* $(c :: t)$.

Exemple 3.6 *Soit R le SRCH dont le premier niveau de hiérarchie contient l'axiomatisation des booléens et de l'ensemble des entiers naturels enrichi de l'opérateur* $\leq$, *et dont le second niveau définit le minimum de deux entiers à l'aide des deux règles suivantes :*

$$\begin{cases} x \leq y \Rightarrow min(x,\ y) \rightarrow x \\ \neg(x \leq y) \Rightarrow min(x,\ y) \rightarrow y \end{cases}$$

Les formes normales contextuelles du c-terme $(true :: min(x,\ y))$ sont :

1. $(x \leq y :: x)$ et

2. $(\neg(x \leq y) :: y)$

Le terme clos $(true :: min(2, 3))$ est lui, réductible en la forme normale 2, (nous avons supposé pour cela que le contexte $2 \leq 3$ est réductible par une partie de R en $true$). $\diamond$

Contrairement à la réécriture classique, sans préconditions, un terme contextuel n'a pas une forme normale unique. En effet, un terme peut se réduire en plusieurs formes normales chacune d'elle dépendant d'un contexte qui constitue la condition de normalisation. De plus, ces contextes normalisés doivent avoir la particularité d'être complémentaires. Intuitivement, cette complémentarité assure que toutes les conditions de normalisation sont envisagées et par là-même, que la normalisation du terme contextuel est complète. Plus formellement,

Définition 3.13 Complétude des *EFNC*
$E = EFNC((c :: t)) = \{(c_i :: t_i),\ i \in I\}$ est complet ssi

$$\forall \sigma \in Subst(F),\ \sigma c \rightarrow^*_{H,R} true \Longrightarrow \exists i \in I \text{ tel que } \sigma c_i \rightarrow^*_{H,R} true.$$

Définition 3.14 Exclusion mutuelle opérationnelle
Soient c, c' deux termes de $T(F, X)$ de sorte booléenne ; c et c' sont opérationnellement mutuellement exclusifs ssi

$$\forall \sigma \in Subst(F),\ \sigma c \wedge \sigma c' \rightarrow^*_{H,R} false.$$

Par analogie, la notion d'égalité de deux formes normales dans le cadre inconditionnel est assimilée à la notion de cohérence de deux ensembles de formes normales contextuelles dans le cadre conditionnel. La cohérence se définit formellement comme suit :

Définition 3.15 Cohérence de deux *EFNC*
Soient $E_1 = \{(c_i :: t_i),\ i \in I\}$ et $E_2 = \{(c'_j :: t'_j),\ j \in J\}$ deux $EFNC$ d'un c-terme ; E_1 et E_2 sont cohérents ssi pour toute forme normale $(c_i :: t_i) \in E_1$, toute forme normale $(c'_j :: t'_j) \in E_2$, soit $t_i = t'_j$ soit c_i et c'_j sont opérationnellement mutuellement exclusifs.

Définition 3.16 Paire critique contextuelle
Soit R un $SRCH$ et soient r_1, r_2 deux règles de R ;
soient $r_1 : p_1 \Rightarrow g_1 \rightarrow d_1$ dans $R_i \ominus R_{i-1}$ et $r_2 : p_2 \Rightarrow g_2 \rightarrow d_2$ dans $R_j \ominus R_{j-1}$; supposons que les variables de r_1 et r_2 aient été renommées de telle façon qu'elles n'aient aucune variable commune.
Une paire critique contextuelle $< C, M, N >$, également notée $C :: M = N$, est le résultat de la superposition de r_1 et r_2 ssi il existe une occurrence u de g_1, un unificateur minimal σ de $g_{1/u}$ et g_2 tels que

$$C = \sigma p_1 \wedge \sigma p_2,\ M = \sigma g_1[u \leftarrow \sigma d_2] \text{ et } N = \sigma d_1$$

Cette paire critique contextuelle est hiérarchique si de plus, $\sigma p_1 \in T(F_{i-1}, X)$ et $\sigma p_2 \in T(F_{j-1}, X)$.

Exemple 3.7 *Une superposition entre les règles conditionnelles :*
$x \leq y \Rightarrow insert(x,\ cons(y,\ u)) \rightarrow cons(x,\ cons(y,\ u))$ *et*
$ordered(insert(x,\ u)) \rightarrow ordered(u)$
peut être calculée et produit la paire critique contextuelle suivante :

$$x \leq y \Rightarrow ordered(cons(x,\ cons(y,\ u))) = ordered(cons(y,\ u)).\ \diamond$$

Définition 3.17 Convergence d'une paire critique contextuelle hiérarchique
Soit R un $SRCH$ et soit $C :: M = N$ une paire critique contextuelle hiérarchique ; $C :: M = N$ converge sur les termes clos de $T(F)$ ssi pour toute substitution close σ dans $Subst(F)$ telle que $\sigma C \rightarrow^*_{H,R} true$,

$$\exists t' \in T(F) \text{ tel que } \sigma M \rightarrow^*_{H,R} t' \text{ et } \sigma N \rightarrow^*_{H,R} t'.$$

Un autre aspect, celui-ci concernant les booléens, nous est nécessaire pour établir notre principal résultat de confluence sur les termes clos. En effet, notre objectif étant d'appréhender l'algèbre initiale d'une spécification hiérarchique, et puisque les préconditions des règles sont des expressions booléennes, il faut alors s'assurer que cette algèbre est fidèle par rapport à la sorte booléenne. Nous sommes amené par conséquent, à caractériser le comportement de l'algèbre initiale d'une spécification conditionnelle hiérarchique par rapport à la sorte booléenne. Ce comportement est défini à l'aide de deux propriétés, la *consistance* et la *convertibilité hiérarchique par rapport aux booléens.* Plus formellement,

Définition 3.18 Convertibilité hiérarchique par rapport aux booléens
Un $SRCH$, R est hiérarchiquement convertible par rapport aux booléens ssi

$$\forall t \in T(F)_{bool},\ soit\ t \rightarrow^*_{H,R} true\ soit\ t \rightarrow^*_{H,R} false.$$

Définition 3.19 Consistance par rapport aux booléens
$E(R)$ est consistante par rapport aux booléens ssi

$$\neg(true =_{E(R)} false).$$

Une spécification consistante et convertible par rapport aux booléens est dite *fidèle* par rapport à la sorte booléenne. Cela signifie intuitivement qu'il n'y a pas d'ajout de booléen et pas de confusion entre $true$ et $false$.
L'assertion suivante est une conséquence triviale de la propriété de consistance par rapport aux booléens :

$$\text{pour tout terme booléen clos } t,\ \text{si } t =_{E(R)} true \text{ alors } t \neq_{E(R)} false.$$

où $t \neq_{E(R)} false$ signifie que l'on ne peut pas obtenir la constante $false$ à partir de t par des remplacements d'égaux par des égaux en utilisant les axiomes de E. Etant donné que nous supposons que toute spécification conditionnelle hiérarchique est une extension (ou un enrichissement) de la spécification des booléens SP_{bool}, $(SP_{bool} = SP_0)$ et que l'ensemble C_{bool} des constructeurs de SP_{bool} est uniquement constitué des constantes $true$ et $false$, irréductibles dans le système de réécriture correspondant, nous pouvons affirmer que

- si un système de réécriture conditionnelle hiérarchique est hiérarchiquement convertible, il est également hiérarchiquement convertible par rapport aux booléens. Par conséquent, dans la suite de nos travaux, nous ne citerons plus cette propriété dans les énoncés des différents résultats si l'hypothèse de convertibilité hiérarchique est supposée acquise.

- la propriété de consistance par rapport aux booléens est un cas particulier de la consistance relative (voir déf 3.8). En effet, si un terme clos booléen t vérifie $t =_{SP_i} true$ à un niveau quelconque i de hiérarchie alors, la consistance par rapport aux booléens implique $t =_{SP_j} true$ pour tout niveau de hiéarchie j dans $[0 \ldots i]$. Par le théorème 3.3 établi dans la suite, la consistance par rapport aux booléens peut également être induite à partir de la confluence sur les termes clos du système de réécriture obtenu et elle n'a donc pas besoin d'être supposée initialement.

Résultat principal

Nous sommes à présent en mesure de formuler notre résultat principal. Ce résultat établit d'une part, une condition suffisante de confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique et d'autre part, l'existence d'une algèbre initiale consistante par rapport aux booléens. La preuve de ce théorème principal est effectuée dans le paragraphe 4 qui suit.

Théorème principal de confluence sur les termes clos

Théorème 3.3 *Soit R un $SRCH$ tel que :*

1. $\xrightarrow{c}_{H,R}$ *soit nœthérienne,*
2. *R soit opérationnellement bien couvert et hiérarchiquement convertible,*
3. *$true$ et $false$ soient irréductibles dans R.*

Si toutes les paires critiques contextuelles dans R définissent des $EFNC$ complets et cohérents,

- $\rightarrow_{H,R}$ *est confluente sur les termes clos de $T(F)$ et*
- *$E(R)$ est consistante par rapport aux booléens.*

Corollaire 3.1 *Soit R un $SRCH$ vérifiant les mêmes hypothèse que dans le théorème 3.3 ;*
Si toutes les paires critiques contextuelles dans R définissent des $EFNC$ complets et cohérents,

- $\rightarrow_R$ *est confluente sur les termes clos de $T(F)$,*
- *$E(R)$ est consistante par rapport aux booléens et*
- *la relation $\rightarrow_R$ est également nœthérienne.*

Preuve : Sous les conditions du corollaire, les relations $\rightarrow_{H,R}$ et $\rightarrow_R$ sont équivalentes sur les termes clos par le théorème 3.2. D'autre part, la terminaison de $\rightarrow_R$ est induite pour les raisons suivantes : $\xrightarrow{c}_{H,R}$ est nœthérienne, donc $\rightarrow_{H,R}$ l'est aussi (par la proposition 3.3 établie dans la suite), et d'autre part, $\rightarrow_{H,R}$ et $\rightarrow_R$ sont équivalentes sur les termes clos. La preuve est par conséquent immédiate.

Il est important de remarquer que ces résultats nous permettent également d'effectuer

des preuves inductives. En effet, pour prouver la validité d'une équation e dans l'algèbre initiale d'une spécification suffisamment complète, il suffit de prouver la consistance relative de la spécification enrichie par l'équation e. Par la proposition 3.2, cette consistance est établie en montrant que la relation $\rightarrow_R$ est confluente sur les termes clos, où R est le système de réécriture conditionnelle associé à la spécification enrichie. De plus, la complétion du système de réécriture ajoute des conséquences inductives qui apparaissent comme des lemmes permettant de prouver le théorème e.

4 Preuves des principaux résultats

Cette partie comporte la preuve du théorème principal. Cette preuve présente l'avantage d'être modulaire puisqu'elle est fondée sur une récurrence sur les niveaux de hiérarchie. Il s'agit en effet de supposer la confluence sur les termes clos acquise au niveau $n-1$ de hiérarchie, et de prouver la convergence sur $T(F_n)$ de toutes les paires critiques calculées, par un test effectif portant sur les $EFNC$ correspondants. A partir de la convergence sur les termes clos de ces paires critiques, la preuve de confluence sur $T(F_n)$ de tout R_n est achevée en utilisant les hypothèses de confluence de R_{n-1} et de consistance par rapport aux booléens de $E(R_{n-1})$ pour établir la convergence des préconditions.

Nous présentons auparavant les preuves correspondant aux étapes intermédiaires et qui permettent d'établir de façon claire et concise le résultat de confluence.

4.1 Quelques résultats nécessaires

Définition 3.20 Substitution normalisée
Une substitution close σ est normalisée si pour toute variable x de sorte s dans $Dom(\sigma)$, $\sigma(x)$ appartient au premier niveau dans lequel apparaît s.

Remarque : Dans le cas particulier où une spécification hiérarchique est construite uniquement à partir d'enrichissements, σ est normalisée si $\sigma(x)$ est dans $T(F_0)$ pour toute variable x dans $Dom(\sigma)$.

Théorème 3.4 *Dans un $SRCH$, R tel que $\rightarrow_R$ soit confluente sur les termes clos et tel que les constantes $true$ et $false$ soient irréductibles, $E(R)$ est consistante par rapport aux booléens.*
Si de plus, R est hiérarchiquement convertible, la consistance de $E(R)$ peut être caractérisée par la confluence de la relation $\rightarrow_{H,R}$.

Preuve : Supposons que $E(R)$ ne soit pas consistante par rapport aux booléens, alors $true =_{E(R)} false$. Puisque $\rightarrow_R$ est confluente sur les termes clos, il existe un terme t tel que $true \rightarrow_R^* t$ et $false \rightarrow_R^* t$, ce qui contredit l'hypothèse d'irréductibilité de $true$ et $false$.

Par le théorème 3.3, la convertibilité hiérarchique de R et la confluence de $\rightarrow_R$ impliquent la confluence de $\rightarrow_{H,R}$. La preuve est alors immédiate. □

Lemme 3.2 *Soit R un $SRCH$; si $(c :: t) \stackrel{c}{\rightarrow}{}^*_{H,R} (c' :: t')$ et si $\sigma c' \rightarrow^*_{H,R} true$ pour une substitution close normalisée σ, alors $\sigma t \rightarrow^*_{H,R} \sigma t'$.*

Preuve par récurrence sur la longueur de la réécriture :
a) Si la longueur de la réécriture est 0, le cas est trivial ;

b) Supposons que la réécriture soit de longueur $m+1$ supérieure à 0 et soit $(c" :: t")$ tel que $(c :: t) \stackrel{c}{\rightarrow}^{m+1}_{H,R} (c" :: t")$,
hypothèse de récurrence : soit $(c' :: t')$ tel que $(c :: t) \stackrel{c}{\rightarrow}^{m}_{H,R} (c' :: t') \stackrel{c}{\rightarrow}_{H,R} (c" :: t")$, si $\sigma c' \rightarrow^*_{H,R} true$ alors $\sigma t \rightarrow^*_{H,R} \sigma t'$. On se propose de montrer que le lemme est valide pour $(c" :: t")$;

1. réécriture de terme :
   si $(c' :: t') \stackrel{c}{\rightarrow}_{H,R,t} (c" :: t")$, il existe un indice k dans $[1 \ldots n]$, une règle $p \Rightarrow g \rightarrow d \in R_k \ominus R_{k-1}$, une occurrence u de t et une substitution τ telles que $t'_{/u} = \tau g$, $t" = t'[u \leftarrow \tau d]$, $c" = c' \wedge \tau p$ et $\tau p \in T(F_{k-1}, X)$.
   Soit σ une substitution close normalisée telle que $\sigma c" \rightarrow^*_{H,R} true$. Par définition de $c"$, $\sigma\tau p \rightarrow^*_{H,R} true$ et $\sigma c' \rightarrow^*_{H,R} true$.

   Puisque σ est normalisée et que $\tau p \in T(F_{k-1}, X)$, $\sigma\tau p$ est dans $T(F_j)$ avec $j \leq k-1$. Par conséquent, $\sigma t' \rightarrow_{H,R} \sigma t"$ par définition de $\rightarrow_{H,R}$.

   D'autre part, $\sigma c' \rightarrow^*_{H,R} true$ donc par hypothèse de récurrence, $\sigma t \rightarrow^*_{H,R} \sigma t'$. Par conséquent, $\sigma t \rightarrow^*_{H,R} \sigma t"$.

2. réécriture de contexte :
   si $(c' :: t') \stackrel{c}{\rightarrow}_{H,R,c} (c" :: t")$, il existe une règle $p \Rightarrow g \rightarrow d \in R_k \ominus R_{k-1}$, une occurrence ω de c et une substitution τ telles que $c'_{/\omega} = \tau g$, $c" = c'[u \leftarrow \tau d] \wedge \tau p$, $t" = t'$ et $\tau p \in T(F_{k-1}, X)$.
   Soit $a = c'[u \leftarrow \tau d]$ et soit σ une substitution close normalisée telle que $\sigma c" \rightarrow^*_{H,R} true$; par conséquent, $\sigma(a \wedge \tau p) \rightarrow^*_{H,R} true$ donc, $\sigma a \rightarrow^*_{H,R} true$ et $\sigma\tau p \rightarrow^*_{H,R} true$.

   Puisque σ est normalisée, $\sigma\tau p$ est dans $T(F_j)$ avec $j \leq k-1$. Par conséquent, $\sigma c' \rightarrow_{H,R} \sigma\alpha$, par définition de $\rightarrow_{H,R}$. Par conséquent, $\sigma c' \rightarrow_{H,R} \sigma\alpha \rightarrow^*_{H,R} true$.

   Puisque $\sigma c' \rightarrow^*_{H,R} true$, par hypothèse de récurrence, nous avons $\sigma t \rightarrow^*_{H,R} \sigma t' = \sigma t"$. □

Le théorème suivant établit qu'il est suffisant de considérer les ensembles de formes normales correspondant aux paires critiques contextuelles calculées pour prouver la convergence de celles-ci. Comme pour le cas classique, sans préconditions, nous obtenons alors une méthode effective de preuve de convergence des paires critiques.

Théorème 3.5 *Soit R un $SRCH$ tel que*

1. $\stackrel{c}{\rightarrow}_{H,R}$ *soit nœthérienne,*
2. R *soit hiérarchiquement convertible,*
3. $true$, $false$ *soient irréductibles dans* R,
4. $\rightarrow_{H,R_{i-1}}$ *soit confluente sur les termes clos de* $T(F)$ *pour un entier* i *dans* $[1 \ldots n]$.

Soit $C :: M = N$ une paire critique contextuelle calculée entre R_i et R_{i-1} ou entre R_{i-1} et R_i ; si $(C :: M)$ et $(C :: N)$ admettent des $EFNC$ complets et cohérents,

σM *et* σN *convergent pour toute substitution close* σ *telle que* $\sigma C \in T(F_{i-1})$.

Preuve : Soit σ une substitution close telle que $\sigma C \rightarrow^*_{H,R} true$ et telle que $\sigma C \in T(F_{i-1})$; soient

$E_1 = \{(C_k :: M_k),\ k \in K\}$ un $EFNC$ de $(C :: M)$ et
$E_2 = \{(C'_j :: N_j),\ j \in J\}$ un $EFNC$ de $(C :: N)$.

Supposons que M soit d'un niveau de hiérarchie égal à k_1, et N d'un niveau de hiérarchie égal à k_2, $k_1 \leq i$, $k_2 \leq i$; puisque $C :: M = N$ est issue de la superposition de deux règles entre R_i et R_{i-1}, C est dans $T(F_{i-1}, X)$;
On définit σ' de la façon suivante :

- pour toute variable x dans $Var(M) \cup Var(N)$, $\sigma'(x)$ est la forme normale de $\sigma(x)$ par la relation $\rightarrow_{H,R}$
- σ' est normalisée.

et soit k' un indice dans $[1 \ldots n]$ tel que $\sigma'(x) \in T(F_{k'})$. k' est un niveau quelconque de hiérarchie tel que $0 \leq k' \leq min(k_1, k_2)$, $(i \geq k_1, k_2)$.

σ' est définie pour la raison suivante : lorsque les termes M et N sont réécrits par la relation contextuelle, les contextes C_k, $k \in K$ et C'_j, $j \in J$ obtenus dans les EFNC, contiennent généralement plus de variables que le contexte C, puisqu'à chaque étape de réécriture, un contexte est enrichi par l'ajout d'une nouvelle formule booléenne (ces variables sont toutefois toutes contenues dans les termes M et N). Il devient par conséquent nécessaire de définir une substitution σ' primitive sur les variables des contextes C_k et C'_j.

σ' existe par la propriété de convertibilité hiérarchique de R et du fait que la sorte de $\sigma(x)$ est égale à la sorte de x, contenue dans S_{k_1} ou S_{k_2}.

Nous avons $\sigma C \rightarrow^*_{H,R} \sigma' C$, $\sigma C \rightarrow^*_{H,R} true$ et $\sigma C \in T(F_{i-1})$ par hypothèse. Par conséquent, $\sigma' C \in T(F_{i-1})$. Puisque $\rightarrow_{H,R_{i-1}}$ est confluente sur les termes clos, $\sigma' C \rightarrow^*_{H,R} true$.

E_1 est complet donc par hypothèse, puisque $\sigma' C \rightarrow^*_{H,R} true$, il existe un indice k dans K tel que $\sigma' C_k \rightarrow^*_{H,R} true$ et tel que $\sigma' C_k \in T(F_{i-1})$.
Par ailleurs, $(C :: M) \overset{c}{\rightarrow}^*_{H,R} (C_k :: M_k)$. Puisque $\sigma' C_k \rightarrow^*_{H,R} true$ et que σ' est normalisée, d'après le lemme 3.2, $\sigma' M \rightarrow^*_{H,R} \sigma' M_k$.

De manière similaire, E_2 est complet et $\sigma' C \rightarrow^*_{H,R} true$. Il existe donc un indice j dans J tel que $\sigma' C'_j \rightarrow^*_{H,R} true$ et tel que $\sigma' C'_j \in T(F_{i-1})$.
En raisonnant de façon analogue, on obtient $\sigma' N \rightarrow^*_{H,R} \sigma' N_j$.

Puisque $\sigma' C_k \rightarrow^*_{H,R} true$ et $\sigma' C'_j \rightarrow^*_{H,R} true$, $\sigma' C_k \wedge \sigma' C'_j \rightarrow^*_{H,R} true$. Puisque $\rightarrow_{H,R_{i-1}}$ est confluente sur les termes clos, par le théorème 3.4, $E(R_{i-1})$ est consistante par rapport aux booléens. Par conséquent, $\sigma' C_k \wedge \sigma' C'_j \not\rightarrow^*_{H,R} false$. Les $EFNC$ étant cohérents, $M_k = N_j$ et donc $\sigma' M_k = \sigma' N_j$. Par conséquent, $\sigma' M$ et $\sigma' N$ convergent ;

D'autre part, $\sigma M \rightarrow^*_{H,R} \sigma' M$ et $\sigma N \rightarrow^*_{H,R} \sigma' N$ alors σM et σN convergent. Par conséquent, la paire critique $C :: M = N$ converge sur les termes clos. □

Théorème 3.6 *Soit R un $SRCH$ tel que*

1. $\overset{c}{\rightarrow}_{H,R}$ *soit nœthérienne,*
2. *R soit hiérarchiquement convertible et opérationnellement bien couvert,*
3. $\rightarrow_{H,R_{i-1}}$ *soit confluente sur les termes clos de $T(F)$ pour un entier i dans $[0 \ldots n]$.*

Si toutes les paires critiques entre R_i et $R_i \ominus R_{i-1}$ convergent sur les termes clos de $T(F)$, $\rightarrow_{H,R_i}$ est confluente sur les termes clos de $T(F)$.

Preuve :

1. $i = 0$, ce cas se ramène à prouver le théorème de Knuth-Bendix pour des systèmes sans préconditions [64, 47].

2. $i > 0$, montrons dans un premier temps que $\rightarrow_{H,R_i}$ est localement confluente sur les termes clos de $T(F)$;
   Soit t un terme dans $T(F)$ tel que $t \rightarrow_{H,R} t_1$ et $t \rightarrow_{H,R} t_2$;
   puisque $\rightarrow_{H,R_{i-1}}$ est confluente sur les termes clos de $T(F)$, nous pouvons supposer que l'un des pas de réécriture est de niveau i, (l'autre étant d'un niveau $j \leq i$).
   Il existe alors deux occurrences u_1 et u_2 de t, deux règles dans R, $r_1 : p_1 \Rightarrow g_1 \rightarrow d_1$ dans $R_i \ominus R_{i-1}$ et $r_2 : p_2 \Rightarrow g_2 \rightarrow d_2$ dans $R_j \ominus R_{j-1}$, et deux substitutions σ_1 et σ_2 telles que :
   $t_{/u_1} = \sigma_1 g_1$, $t_{/u_2} = \sigma_2 g_2$, $\sigma_1 p_1 \rightarrow^*_{H,R} true$, $\sigma_2 p_2 \rightarrow^*_{H,R} true$,
   $t_1 = t[u_1 \leftarrow \sigma_1 d_1]$ et $t_2 = t[u_2 \leftarrow \sigma_2 d_2]$.

   - cas des occurrences disjointes :
     Le diagramme se referme comme suit :

     $$t' = t_1[u_2 \leftarrow \sigma_2 d_2] = t_2[u_1 \leftarrow \sigma_1 d_1].$$

   - cas des occurrences préfixes :

     (a) u_2 est à une occurrence de g_1 qui ne correspond pas à une variable, soit $u_2 = u_1.u$ et $g_{1_{/u}}$ et g_2 peuvent être unifiés par $\sigma_1 \cup \sigma_2$. Soit μ leur plus général unificateur ; $\exists \tau$ tel que $\sigma_1 \cup \sigma_2 = \tau.\mu$.
     r_1 est de niveau i. Par conséquent, $\sigma_1 p_1 \in T(F_{i-1}, X)$.
     De même, r_2 a un niveau de hiérarchie égal à j. Donc $\sigma_2 p_2$ est dans $T(F_{j-1}, X)$.
     pour $k = 1, 2$, $\sigma_k p_k \rightarrow^*_{H,R} true$ alors $\tau.\mu p_k \rightarrow^*_{H,R} true$.

     Nous avons alors une paire critique :
     $C = \mu p_1 \wedge \mu p_2$, $M = t[u_1 \leftarrow \mu d_1]$, $N = t[u_2 \leftarrow \mu d_2])$;
     de plus, $\sigma_1 g_{1_{/u}} = \sigma_2 g_2$.
     $\mu p_1 \in T(F_{i-1}, X)$ et $\mu p_2 \in T(F_{j-1}, X)$ donc $C :: M = N$ est une paire critique hiérarchique.

     Par ailleurs, $C :: M = N$ converge sur les termes clos de $T(F)$; par conséquent, pour toute substitution close σ telle que $\sigma C \rightarrow^*_{H,R} true$, σM et σN convergent ; par compatibilité, t_1 et t_2 convergent aussi.

     (b) u_1 est à une occurrence de g_2 qui ne correspond pas à une variable : ce cas est symétrique au précédent.

     (c) u_2 est à une occurrence de g_1 qui correspond à une variable
     $u_2 = u_1.u$ et $u = v_1.v_2$; $g_{1_{/v_1}} = x$, $\sigma_2 g_2 = \sigma_1(x)_{/v_2}$
     soit $\sigma'_1(x) = \sigma_1(x)[v_2 \leftarrow \sigma_2 d_2]$, $\sigma'_1(y) = \sigma_1(y)$, $y \neq x$
     $\sigma_1(x) \rightarrow_R \sigma'_1(x)$; en réécrivant $t_{/u_1}$ à chaque occurrence de x, on obtient $t_{/u_1} \rightarrow^*_R \sigma'_1 g_1$.
     $r_1 \in R_i \ominus R_{i-1}$ et $r_2 \in R_j \ominus R_{j-1}$, $j \leq i$;

le problème consiste à vérifier que $\sigma'_1 p_1 \rightarrow^*_{H,R} true$ afin de réécrire $\sigma'_1 g_1$ en $\sigma'_1 d_1$; en réécrivant chaque occurrence de $\sigma_1(x)$, on obtient $\sigma_1 p_1 \rightarrow^*_R \sigma'_1 p_1$. D'une part, $\sigma_1 p_1 \rightarrow^*_{H,R} true$ et $\sigma_1 p_1 \rightarrow^*_{H,R} \sigma'_1 p_1$ et d'autre part, $\sigma_1 p_1$ et $\sigma'_1 p_1 \in T(F_{i-1}, X)$.

Puisque par hypothèse, $\rightarrow_{H,R_{i-1}}$ est confluente sur les termes clos, donc $\sigma'_1 p_1 \rightarrow^*_{H,R} true$.
Considérons le dernier cas avant de conclure ;

(d) u_1 est à une occurrence de g_2 qui correspond à une variable
$u_1 = u_2.u$ et $u = v_1.v_2$.
D'une part, soit $g_{2/v_1} = x$, $\sigma_1 g_1 = \sigma_2 x_{/v_2}$; soit $\sigma'_2(x) = \sigma_2(x)[v_2 \leftarrow \sigma_1 d_1]$,
$\sigma'_2(y) = \sigma_2(y)$, $y \neq x$
$\sigma_2(x) \rightarrow^*_{H,R} \sigma'_2(x)$.
Similairement, le problème consiste à montrer que $\sigma'_2 p_2 \rightarrow^*_{H,R} true$.
$\sigma_2 p_2 \in T(F_{j-1}, X)$ alors $\sigma_2(y) \in T(F_{j-1}, X)$ pour tout $y \in Var(p_2)$.
$\sigma_2(x)$ contient $\sigma_1 g_1$ et $g_1 \in T(F_i, X) \ominus T(F_{i-1}, X)$ donc $\sigma_2(x)$ n'est pas dans $T(F_{i-1}, X)$. Par conséquent, x n'est pas une variable de p_2 et $\sigma'_2 p_2 = \sigma_2 p_2$.
Puisque $\sigma_2 p_2 \rightarrow^*_{H,R} true$ nous avons $\sigma'_2 p_2 \rightarrow^*_{H,R} true$.

A partir de tous les cas considérés, nous pouvons conclure que $\rightarrow_{R_i}$ est localement confluente sur les termes clos de $T(F)$.

$\overset{c}{\rightarrow}_{H,R}$ est nœthérienne donc $\rightarrow_{H,R}$ est aussi nœthérienne par la proposition 3.3 établie dans le paragraphe suivant. Par conséquent, $\rightarrow_{H,R_i}$ est aussi nœthérienne.

Par le lemme de Newman [82], puisque $\rightarrow_{H,R_i}$ est nœthérienne et localement confluente sur les termes clos de $T(F)$, $\rightarrow_{H,R}$ est confluente sur les termes clos de $T(F)$. □

4.2 Preuve du théorème de confluence sur les termes clos

Après avoir présenté ces différents éléments de preuve, nous pouvons établir la preuve du théorème de confluence ;

Preuve par récurrence : Soit $i \geq 0$,

- $i = 0$: la preuve est triviale.
- $i > 0$: Nous supposons que $\forall j,\ 0 \leq j < i,\ \rightarrow_{H,R_j}$ est confluente sur les termes clos de $T(F)$ et $E(R_j)$ est consistante par rapport aux booléens ;

  puisque R est hiérarchiquement convertible, que la relation $\overset{c}{\rightarrow}_{H,R}$ est nœthérienne, que toute paire critique contextuelle hiérarchique $C :: M = N$ de R est telle que $(C :: M)$ et $(C :: N)$ définissent des $EFNC$ complets et cohérents et puisque $\rightarrow_{H,R_j}$ est confluente sur les termes clos de $T(F)$ pour $j < i$, nous pouvons affirmer par le théorème 3.5 que toute paire critique contextuelle de R converge sur les termes clos de $T(F)$.

  $\overset{c}{\rightarrow}_{H,R}$ est nœthérienne, R est hiérarchiquement convertible, opérationnellement bien couvert, $\rightarrow_{H,R_{i-1}}$ est confluente sur les termes clos. Alors, puisque toute paire critique de R, et en particulier toute paire critique entre R_i et $R_i \ominus R_{i-1}$, converge sur les termes clos de $T(F)$, par le théorème 3.6, $\rightarrow_{H,R_i}$ est confluente sur les termes

clos.

Puisque $\rightarrow_{H,R_i}$ est confluente sur les termes clos et que *true* et *false* par hypothèse sont irréductibles dans R, $E(R_i)$ est consistante par rapport aux booléens (par le théorème 3.4). □

5 Conditions d'applicabilité des résultats

L'intérêt de cette partie est de justifier l'utilisation de toutes les propriétés citées dans le paragraphe précédent. Nous étudions par ailleurs les divers procédés permettant d'établir chacune d'elles dans un système de réécriture conditionnelle hiérarchique.

5.1 La terminaison

Proposition 3.3 *Si $\xrightarrow{c}_{H,R}$ est nœthérienne, $\rightarrow_{H,R}$ est également nœthérienne et la relation de réécriture est décidable.*

Preuve : Supposons que $\xrightarrow{c}_{H,R}$ soit nœthérienne mais que $\rightarrow_{H,R}$ ne le soit pas ; il existe alors une séquence infinie de réécritures de la forme :

$$t_0 \rightarrow_{H,R} t_1 \rightarrow_{H,R} \ldots \rightarrow_{H,R} t_n \rightarrow_{H,R} \ldots$$

avec pour tout $i \geq 0$, $t_i \rightarrow_{H,R} t_{i+1}$ par une règle $p_i \Rightarrow g_i \rightarrow d_i \in T(F_{j_i}, X)$. Par conséquent, il existe pour tout i une substitution σ_i pour la séquence infinie correspondante :
$(true :: t_0) \xrightarrow{c}_{H,R} (\sigma_0 p_0 :: t_1) \xrightarrow{c}_{H,R} (\sigma_0 p_0 \wedge \sigma_1 p_1 :: t_2) \xrightarrow{c}_{H,R} \ldots \xrightarrow{c}_{H,R} (\sigma_0 p_0 \wedge \ldots \wedge \sigma_{i-1} p_{i-1} :: t_i) \xrightarrow{c}_{H,R} \ldots$ ce qui entraîne que $\xrightarrow{c}_{H,R}$ n'est pas nœthérienne.

La décidabilité de la relation de réécriture est prouvée de façon similaire. □

Afin de prouver que dans un système de réécriture hiérarchique, une dérivation quelconque termine, nous nous sommes inspirée des travaux de Jouannaud et Waldmann [53]. Nous utilisons un ordre noté $\succ$ qui est la fermeture transitive de la relation $(st \cup >)$ où st est l'ordre de sous-terme strict et $>$ est un ordre de réduction associé au système R. L'ordre $\succ$ est bien-fondé [50].

Définition 3.21 Extension de $\succ$ aux termes contextuels
Soit $\succ$ l'ordre de réduction défini ci-dessus. Cet ordre peut être étendu lexicographiquement aux termes contextuels de la façon suivante :

$$(c :: t) \succ_c (c' :: t') \Longleftrightarrow t \succ t' \text{ ou } (t = t' \text{ et } c \succ c').$$

Proposition 3.4 *Soit $>$ un ordre bien-fondé et soit $\succ_{st}$ son extension définie comme ci-dessus ; alors $\succ_{st,c}$ est aussi un ordre bien-fondé.*

Preuve : $\succ_c$ est l'extension lexicographique d'un ordre bien-fondé $\succ$. Par conséquent, il est aussi bien-fondé [18]. □

Proposition 3.5 *Soit $>$ un ordre de réduction et soit $\succ$ son extension définie comme ci-dessus ; supposons, par convention, que l'opérateur $\wedge$ soit plus petit que tout autre opérateur de F ;*

si R est un SRCH tel que toute règle de R vérifie: $Var(p) \subseteq Var(g)$, $Var(d) \subseteq Var(g)$, $\sigma g > \sigma p$ et $\sigma g > \sigma d$ pour toute substitution σ, la réécriture contextuelle est contenue dans l'extension de $>$ à $\succ_c$, i.e.

$$(c :: t) \xrightarrow{c}_{H,R} (c' :: t') \Longrightarrow (c :: t) \succ_c (c' :: t').$$

$\xrightarrow{c}_{H,R}$ est par conséquent nœthérienne.

Preuve : Soit la réécriture $(c :: t) \xrightarrow{c}_{H,R} (c' :: t')$ par la règle $p \Rightarrow g \rightarrow d \in R$; il existe par conséquent

1. soit une occurrence u de t et une substitution σ telles que $t_{/u} = \sigma g$
2. soit une occurrence u de c et une substitution σ telles que $c_{/u} = \sigma g$

Considérons le premier cas :
le résultat de la réécriture de $(c :: t)$ en $(c' :: t')$ donne $c' = c \wedge \sigma p$, $t' = t[u \leftarrow \sigma d]$.
Puisque par hypothèse, $\sigma g > \sigma d$ donc $t > t'$ (par la propriété de remplacement) ;
si l'on considère l'extension de $>$ à $\succ$, nous avons $t \succ t'$; alors $(c :: t) \succ_c (c' :: t')$.

Considérons le cas 2 :
le résultat de la réécriture de $(c :: t)$ produit les termes $c' = c[u \leftarrow \sigma d] \wedge \sigma p$ et $t' = t$.
Puisque par hypothèse, $\sigma g > \sigma p$ nous avons $c \succ \sigma p$.
D'autre part, par hypothèse, $\sigma g > \sigma d$ donc, $c = c[u \leftarrow \sigma g] > c[u \leftarrow \sigma d]$ par la propriété de remplacement ;
si l'on considère l'extension de $>$ à $\succ$, nous avons $c \succ c[u \leftarrow \sigma d]$;
$c' = c[u \leftarrow \sigma d] \wedge \sigma p$; puisque par hypothèse, $\wedge$ est plus petit que les autres opérateurs de F, nous obtenons alors $c \succ c'$;
$t = t'$ and $c \succ c'$ alors $(c :: t) \succ_c (c' :: t')$; □

Dans le but de prouver la terminaison de la relation contextuelle $\xrightarrow{c}_{H,R}$ dans un système de réécriture conditionnelle hiérarchique, nous définissons une propriété dite de *compatibilité avec la hiérarchie*. Cette propriété consiste à fixer partiellement la précédence de certains opérateurs en fonction du niveau de hiérarchie auquel ils appartiennent. Le principe est fondé sur l'assertion suivante : tout terme de niveau i est plus grand que tout autre terme de niveau plus petit. Ce procédé permet d'éviter la comparaison entre le membre gauche et la précondition d'une même règle dans un système hiérarchique. Cependant, un certain travail reste à faire pour définir formellement un ordre ayant les caractéristiques appropriées.

Définition 3.22 Compatibilité avec la hiérarchie
Soient R un SRCH et $>$ un ordre donné sur les termes ; $>$ est compatible avec la hiérarchie ssi

$$\forall s \in T(F_i, X) \ominus T(F_{i-1}, X),\ \forall t \in T(F_{i-1}, X),\ et\ Var(t) \subseteq Var(s),\ s > t.$$

Proposition 3.6 *Soit R un SRCH et soit $>$ un ordre de réduction compatible avec la hiérarchie ; si pour toute règle $p \Rightarrow g \rightarrow d$ de R telle que*

$$Var(p),\ Var(d) \subseteq Var(g),\ \forall \sigma,\ \sigma g > \sigma d,$$

alors, $\xrightarrow{c}_{H,R}$ est nœthérienne.

Preuve : Puique R est hiérarchique, pour tout i, $0 < i \leq n$, toute règle $p \Rightarrow g \rightarrow d$ dans $R_i \ominus R_{i-1}$ vérifie : $Var(p) \subseteq Var(g)$, $Var(d) \subseteq Var(g)$, $g \in T(F_i, X) \ominus T(F_{i-1}, X)$, $d \in T(F_i, X)$, $p \in T(F_{i-1}, X)$.
$>$ est compatible avec la hiérarchie donc $g > p$. Par conséquent, $\forall \sigma$, $\sigma g > \sigma p$ puisque $Var(p) \subseteq Var(g)$.
De plus, $\sigma g > \sigma d$ pour toute règle de R et pour toute substitution σ, donc par la proposition 3.5, $\xrightarrow{c}_{H,R}$ est nœthérienne. □

5.2 La propriété de bonne couverture

La propriété de bonne couverture est liée à la relation de réécriture contextuelle. Nous avons en effet, besoin de calculer, pour chaque c-terme $(c :: t)$, un ensemble complet de formes normales contextuelles, où complet signifie que la disjonction des contextes des formes normales est équivalente, dans l'algèbre initiale, au contexte c.

Définition 3.23 Validité dans l'algèbre initiale
Etant donné un ensemble de termes $\{c_i, i \in I\}$ et soit R un $SRCH$. $I(R)$ désigne l'algèbre initiale de R et $I(R) \models \vee_i c_i$ la validité dans cette algèbre initiale, de l'expression $\vee_i c_i$ définie par

$$I(R) \models \vee_i c_i \iff \forall \sigma \in Subst(F),\ \exists i \in I \text{ tel que } \sigma c_i =_R true.$$

Définition 3.24 Bonne couverture algébrique
R est algébriquement bien couvert ssi pour tout sous-ensemble maximal de règles de la forme $\{p_i \Rightarrow g \rightarrow d_i,\ i \in I\}$ dans R, $I(R) \models \vee_i p_i$.

Le concept algébrique de la propriété de bonne couverture n'implique pas, en général, le concept opérationnel. La validité opérationnelle d'une expression de la forme $\vee_{i=1}^{n} p_i$ est définie de telle façon qu'il existe un indice i dans $[1 \ldots n]$ tel que, non seulement la condition $\sigma p_i =_R true$ soit vérifiée, mais également qu'elle puisse être prouvée de manière effective en réduisant σp_i à $true$. La proposition qui suit établie à partir du lemme 3.3, montre l'équivalence des deux concepts si le système vérifie les propriétés de consistance et de convertibilité hiérarchique par rapport aux booléens.

Ces preuves de bonne couverture algébrique sont établies en utilisant des assertions disjonctives de la forme $\vee_{i=1}^{n} q_i$, ces dernières étant elles-mêmes justifiées par des méthodes de preuve indépendantes. Notons que ce problème a également été évoqué dans les travaux de Ganzinger [35].

Lemme 3.3 *Dans un $SRCH$, R hiérarchiquement convertible par rapport aux booléens, si $E(R)$ est consistante par rapport aux booléens,*

$$\forall t \in T(F)_{bool},\ t =_{E(R)} true \Longrightarrow t \rightarrow^*_{H,R} true.$$

Preuve : Soit $t =_{E(R)} true$; supposons que $t \rightarrow^*_{H,R} false$ donc $t =_{E(R)} false$ et $true =_{E(R)} false$, ce qui réfute l'hypothèse puisque $E(R)$ est consistante par rapport aux booléens. Puisque $t \not\rightarrow^*_{H,R} false$, $t \rightarrow^*_{H,R} true$ par la propriété de convertibilité hiérarchique de R par rapport aux booléens. □

Proposition 3.7 *Soit R un SRCH hiérarchiquement convertible et tel que $E(R)$ soit consistante par rapport aux booléens;*
si R est algébriquement bien couvert, R est aussi opérationnellement bien couvert.

Preuve : Soit R algébriquement bien couvert alors, pour tout sous-ensemble maximal de règles de la forme $\{p_i \Rightarrow g \rightarrow d_i,\ i \in I\}$, pour toute substitution close σ, il existe un indice i dans I tel que $\sigma p_i =_R true$.
σp_i est un terme clos de sorte booléenne donc par le lemme 3.3, si $\sigma p_i =_R true$ alors $\sigma p_i \rightarrow^*_{H,R} true$.
Par conséquent, pour tout sous-ensemble maximal de règles de la forme $\{p_i \Rightarrow g \rightarrow d_i,\ i \in I\}$ pour toute substitution close σ, il existe un indice i dans I tel que $\sigma p_i \rightarrow^*_{H,R} true$, alors R est aussi opérationnellement bien couvert. □

5.3 La complétude suffisante hiérarchique

Notre objectif, dans ce paragraphe, est d'étudier les conditions sous lesquelles les résultats établis dans le chapitre précédent, étendus à la propriété de complétude suffisante hiérarchique, s'appliquent. Il s'agit de vérifier la complétude suffisante hiérarchique d'une spécification $(S_i,\ F_i,\ E_i)$ par rapport à la spécification $(S_{i-1},\ F_{i-1},\ E_{i-1})$ de niveau inférieur. Cependant, puisque le théorème principal 3.3 de confluence sur les termes clos met en jeu la convertibilité d'un système de réécriture hiérarchique, nous allons par conséquent directement nous intéresser à la convertibilité.

La propriété de complétude suffisante hiérarchique est définie sur les termes clos. Soit le système de réécriture hiérarchique $(R = R_0,\ R_1)$; si nous considérons R_1 comme une extension de R_0, il est alors naturel de supposer que tout terme clos de $T(F_1)$ dont la sorte est dans S_0, est équivalent à un terme clos quelconque de $T(F_0)$. Si de plus, cette extension est consistante alors, en termes de modèles, le modèle initial de la spécification SP_1, restreint à l'ensemble de sortes S_0, coïncide avec le modèle initial de la spécification SP_0.

Définir la propriété de complétude suffisante hiérarchique sur les termes avec variables est en général une hypothèse trop forte car, dans la pratique, elle n'est pas souvent vérifiée. Considérons l'exemple suivant :

Exemple 3.8 *Soit $R = (R_0,\ R_1,\ R_2)$ un SRCH avec*

$SP_0 = SP_{bool}$;

$S_1 = S_0 \oplus \{nat\}$;
$F_1 = \{0 :\rightarrow nat,\ succ : nat \rightarrow nat\}$;
$R_1 = R_0$.

$S_2 = S_1$;
$F_2 = F_1 \oplus \{pair : nat \rightarrow bool\}$;
$R_2 = R_1 \oplus \{pair(0) \rightarrow true,\ pair(succ(0)) \rightarrow false,$
$\qquad pair(succ(succ(x))) \rightarrow pair(x)\}$.

R_2 est un enrichissement de R_0 hiérarchiquement convertible, puisque pour tout terme clos t dans $T(F_1)_{nat}$,
$pair(t) = true$ ou $pair(t) = false$. Cependant, le terme $pair(x)$ n'est équivalent ni à $true$ ni à $false$. ◇

Orejas et Navarro ont cependant défini un concept de complétude suffisante sur des termes

avec variables dans le cadre des spécifications paramétrées [81]. Ces termes sont toutefois restreints à des termes construits uniquement à partir de variables qui appartiennent à la sorte paramétrée, ceci dans le but d'établir des conditions suffisantes pour tester la correction d'un passage de paramètres.

Remarque : Soit $R = (R_0, \ldots, R_n)$ un $SRCH$; si nous prenons pour convention de construire R à partir d'une extension des booléens, i.e. si R_0 définit le système booléen, avec pour seuls constructeurs les constantes $true$ et $false$, la convertibilité hiérarchique par rapport aux booléens de R est alors une autre formulation de la convertibilité hiérarchique de tout R par rapport à R_0. Cette remarque implique en particulier, que le test de la convertibilité hiérarchique par rapport aux booléens peut s'inclure dans celui de la convertibilité hiérarchique.

Nous ne reprenons pas, dans ce paragraphe, les notions définies et étudiées dans le chapitre précédent. Simplement, nous pouvons affirmer que toutes ces notions peuvent être définies en fonction de la relation $\rightarrow_{H,R}$. En particulier, les notions de réductibilité forte, de validité opérationnelle, s'étendent naturellement à la relation de réécriture hiérarchique ainsi que l'algorithme de test de la convertibilité.

En effet, une condition suffisante pour tester la propriété de complétude suffisante hiérarchique peut être établie en appliquant les résultats du chapitre précédent. Puisque l'algorithme de test de la complétude suffisante d'un système de réécriture conditionnelle est défini en fonction de la relation $\xrightarrow{c}_R$ et sachant que $\xrightarrow{c}_{H,R}$ est contenue dans $\xrightarrow{c}_R$, on peut par conséquent, définir une seconde version de cet algorithme en fonction de la relation hiérarchique $\xrightarrow{c}_{H,R}$. Si on se place dans les mêmes conditions que pour le théorème 2.2, les résultats pour cette seconde version restent valides.

Théorème 3.7 *Soit R un $SRCH$ décroissant tel que les seuls constructeurs de la sorte booléenne soient les constantes true et false et soit i un indice dans $[1 \ldots n]$ tel que R_{i-1} soit confluent sur les termes clos et hiérarchiquement convertible ;*
Si l'algorithme appliqué à R_i s'arrête avec succès, R_i est hiérarchiquement convertible par rapport à F_{i-1}.

Remarque : La condition de décroissance du système garantit la terminaison de la relation $\rightarrow_R$ par le théorème 1.11, et par conséquent la terminaison de la relation $\rightarrow_{H,R}$ puisque $\rightarrow_{H,R}$ est contenue dans $\rightarrow_R$.

Combiner les classes de systèmes décroissants et hiérarchiques permet d'obtenir des résultats plus intéressants à mettre en œuvre. Nous ne savons pas, à l'heure actuelle, établir une condition suffisante pour tester la complétude suffisante d'une spécification conditionnelle hiérarchique sans avoir recours à la propriété de décroissance du système associé. Il serait utile à l'avenir, d'examiner si l'hypothèse de hiérarchie ne peut être davantage exploitée pour affaiblir les conditions d'applicabilité de l'algorithme de complétude suffisante hiérarchique. Par exemple, en établissant dans un système hiérarchique un ordre sur les opérateurs tel que tout opérateur de $F_i \ominus F_{i-1}$ soit plus grand que les opérateurs de F_{i-1}, on s'assure une décroissance partielle du système hiérarchique.

Les corollaires ci-après regroupent les résultats de complétude suffisante hiérarchique et de confluence sur les termes clos.

Corollaire 3.2 *Soit $R = (R_0, \ldots, R_n)$ un $SRCH$ décroissant, opérationnellement bien couvert et tel que true et false soient irréductibles dans R et soient les seuls con-*

structeurs de la sorte booléenne.
Pour tout indice i dans $[1 \ldots n]$, supposons que

1. *$(R_0, \ldots, R_{i-1})$ soit hiérarchiquement convertible,*
2. *$(R_0, \ldots, R_{i-1})$ soit confluent sur les termes clos ;*

si l'algorithme de convertibilité hiérarchique termine avec succès, $(R_0, \ldots, R_{i-1}, R_i)$ est hiérarchiquement convertible.
Si de plus, toutes les paires critiques contextuelles dans R_i définissent des $EFNC$ complets et cohérents,

- *$\rightarrow_{H,R_i}$ est confluente sur les termes clos,*
- *$E(R_i)$ est consistante par rapport aux booléens,*
- *$\rightarrow_{R_i}$ est confluente sur les termes clos.*

La convertibilité hiérarchique de R_i découle du théorème 3.7. La confluence sur les termes clos de $\rightarrow_{H,R_i}$ découle du théorème 3.3 et la relation $\rightarrow_{R_i}$ est également confluente sur les termes clos puisque, par le théorème 3.2, $\rightarrow_{H,R_i}$ et $\rightarrow_{R_i}$ coïncident sur les termes clos.

Corollaire 3.3 *Soit R un $SRCH$ décroissant, opérationnellement bien couvert et tel que $true$ et $false$ soient les seuls constructeurs de la sorte booléenne et soient irréductibles dans R ;*
si l'algorithme de convertibilité hiérarchique appliqué à R termine avec succès, R est hiérarchiquement convertible.
Si de plus, toutes les paires critiques dans R définissent des $EFNC$ complets et cohérents, alors d'une part, $\rightarrow_R$ est confluente sur les termes clos de $T(F)$ et d'autre part, $E(R)$ est consistante par rapport aux booléens.

La preuve du corollaire découle des théorèmes 3.3 et 3.7.

5.4 Les ensembles de formes normales contextuelles ($EFNC$)

L'hypothèse de bonne couverture est justifiée dans la proposition qui suit, car elle est une condition suffisante qui garantit la complétude des ensembles de formes normales contextuelles.

Proposition 3.8 *Soit $R = (R_0, \ldots, R_n)$ un $SRCH$ tel que $\xrightarrow{c}_{H,R}$ soit nœthérienne ;*
si R est opérationnellement bien couvert, hiérarchiquement convertible et si $E(R)$ est consistante par rapport aux booléens jusqu'au niveau $n-1$, il existe un $EFNC$ complet.

Preuve : La preuve est établie en considérant chaque pas de construction de l'$EFNC$.
Soit E_0 l'ensemble $\{(c :: t)\}$; pour tout k, soit E_k l'état de la construction de $EFNC((c :: t))$ à l'étape k, $E_k = \{(c_i :: t_i),\ i \in I\}$ et soit pour tout indice i dans I,

$$(c_i :: t_i) \xrightarrow{c}_{H,R} \{(c'_j :: t'_j),\ j \in J\} ;$$

hypothèse de récurrence :

$$\forall \sigma \in Subst(F),\ \sigma c \rightarrow^*_{H,R} true \Longrightarrow \exists i \in I \text{ tel que } \sigma c_i \rightarrow^*_{H,R} true$$

Pour tout $i \in I$, considérons les deux cas de réécriture :

1. réécriture de terme. La réécriture produit la séquence suivante :

$$(c_i :: t_i) \xrightarrow{c}_{H,R} \{(c_i \wedge \sigma_j p_j :: t'_j),\ j \in J\}.$$

Puisque R est opérationnellement bien couvert, il existe un indice j dans J tel que $\sigma\sigma_j p_j \rightarrow^*_{H,R} true$. Par conséquent, $\sigma(c_i \wedge \sigma_j p_j) \rightarrow^*_{H,R} true$.

2. réécriture de contexte. La réécriture est de la forme :

$$(c_i :: t_i) \xrightarrow{c}_{H,R} \{(c'_j \wedge \sigma_j p_j :: t_i),\ j \in J\}.$$

D'une part, puisque R est opérationnellement bien couvert, il existe un indice j dans J tel que $\sigma\sigma_j p_j \rightarrow^*_{H,R} true$.
D'autre part, par hypothèse, nous avons $\sigma c_i \rightarrow^*_{H,R} true$;
σc_i se réécrit dans le système inconditionnel en $\sigma c'_j$ et σc_i est dans $T(F_{n-1})$; par conséquent, $\sigma c'_j =_{H,R} true$;
par les propriétés de convertibilité hiérarchique et de consistance par rapport aux booléens jusqu'au niveau $n-1$, $\sigma c'_j \rightarrow^*_{H,R} true$. Par conséquent, $\sigma(c'_j \wedge \sigma_j p_j) \rightarrow^*_{H,R} true$.

Dans les deux cas de réécriture, il existe un indice l dans l'ensemble des termes contextuels tel que si $\sigma c \rightarrow^*_{H,R} true$, $\sigma c_l \rightarrow^*_{H,R} true$. Par conséquent, l'ensemble des formes normales contextuelles est encore complet à l'étape $k+1$.
Puisque $\xrightarrow{c}_{H,R}$ est nœthérienne, il existe un indice m tel que

$$E_m = E_{m+1} = EFNC((c :: t)),$$

ce qui achève la preuve. □

Cohérence de deux *EFNC*

La cohérence de deux ensembles de formes normales contextuelles E_1 et E_2 est vérifiée si pour tous termes contextuels $(c_i :: t_i) \in E_1$, et $(c'_j :: t'_j) \in E_2$, soit t_i et t'_j sont syntaxiquement égaux, soit c_i et c'_j sont opérationnellement mutuellement exclusifs. Comme pour la propriété de bonne couverture, nous définissons deux concepts d'exclusion mutuelle de contextes et nous obtenons une équivalence de ces deux concepts si le système possède les propriétés de consistance et de convertibilité hiérarchique par rapport aux booléens. Cette équivalence est obtenue par une preuve similaire que nous ne détaillerons pas.

Définition 3.25 *Soient c, c' deux termes de $T(F, X)_{bool}$, c et c' sont algébriquement mutuellement exclusifs ssi*

$$\forall \sigma \in Subst(F),\ \sigma c \wedge \sigma c' =_R false.$$

Remarque : Il est intéressant d'étudier des méthodes permettant de prouver la propriété d'exclusion mutuelle par des techniques de preuve inductives comme c'est le cas pour la propriété de bonne couverture. Le problème étant indécidable, une solution serait de définir des conditions suffisantes qui impliqueraient la propriété d'exclusion. Cette étude est abordée dans le chapitre suivant dans un cadre spécifique.

6 Exemples

Dans ce paragraphe, nous consacrons une première partie à la présentation du logiciel REVEUR4 en mettant l'accent sur ses capacités et ses insuffisances. La seconde partie comporte essentiellement des exemples de spécifications conditionnelles hiérarchiques. Nous insistons sur deux aspects, l'aspect théorique afin d'illustrer nos résultats et l'aspect technique dans lequel nous précisons comment sont résolues les difficultés dans REVEUR4.

6.1 REVEUR4

REVEUR4 est le logiciel d'implantation de la réécriture contextuelle hiérarchique à deux niveaux. Il a été écrit à Nancy en CLU sur VAX et SUN3 sous le système UNIX. Il a été développé en utilisant un certain nombre de modules de REVE, logiciel d'implantation de la réécriture classique, sans préconditions [74, 26], et il bénéficie par conséquent de la même souplesse quant à l'interface avec l'utilisateur. REVEUR4 n'a pas été étendu à un nombre quelconque de niveaux de hiérarchie (supérieur à deux), en raison de l'évolution ultérieure de nos travaux. En effet, nous avons par la suite abordé une approche différente de la relation de réécriture conditionnelle, en considérant en particulier des systèmes *décroissants*. Ce choix est motivé par divers critères que nous explicitons dans le chapitre suivant et il a donné naissance au prototype RECOND.

Actuellement, REVEUR4 assume deux principales fonctions, la preuve de confluence sur les termes clos en procédant si nécessaire, à une complétion du système et la preuve d'égalité par réécriture contextuelle [105, 95, 9].

La spécification donnée en entrée à REVEUR4 est divisée en deux parties :

- la partie *primitive* est constituée de sortes et d'opérations primitives ainsi que d'équations primitives toutes inconditionnelles. Toutes les préconditions de règles doivent être définies dans cette partie.

- la partie supérieure introduit les nouvelles opérations. Seule cette partie comporte des équations conditionnelles. Ces équations ont une précondition primitive et un membre gauche qui contient au moins un nouveau symbole d'opération, encore appelé *opérateur conditionnel.*

La déclaration des opérateurs primitifs et conditionnels est laissée à la charge de l'utilisateur. Si celui-ci ne respecte pas les conditions de hiérarchie ci-dessus, REVEUR4 ne garantit plus la validité des résultats.

De même, la spécification conditionnelle est supposée être bien couverte et aucun test de bonne couverture n'est effectué. La propriété de complétude suffisante quant à elle, est vérifiée automatiquement a posteriori par REVEUR4, mais elle comporte à l'heure actuelle des incorrections principalement parce que le typage des opérateurs n'est pas pris en compte lors du test.

Une des principales caractéristiques de REVEUR4 est qu'il contient un codage du système booléen de Hsiang dont la fonctionnalité première consiste à effectuer les preuves des contextes et des préconditions de règles [43]. En effet, certains axiomes de la logique propositionnelle ne peuvent pas être orientés et notamment, l'axiome de commutativité du connecteur de conjonction $\wedge$. En utilisant les opérateurs $\wedge$ et $+$ (qui est le OU exclusif), Hsiang obtient le système BH suivant dont une particularité importante est qu'il est

confluent et nœthérien modulo les axiomes de commutativité et d'associativité $E_{\wedge,+}$ de $\wedge$ et $+$. En particulier, toute formule propositionnelle a une forme normale unique dans ce système.

$$\begin{aligned}E_{\wedge,+} = \{&x \wedge y = y \wedge x,\\ &x \wedge (y \wedge z) = (x \wedge y) \wedge z,\\ &x + y = y + x,\\ &x + (y + z) = (x + y) + z\};\end{aligned}$$

$$\begin{aligned}BH = \{&x + x \rightarrow false,\\ &x + false \rightarrow x,\\ &x \wedge x \rightarrow x,\\ &x \wedge true \rightarrow x,\\ &x \wedge false \rightarrow false,\\ &x \wedge (y + z) \rightarrow (x \wedge y) + (x \wedge z),\\ &x \vee y \rightarrow (x + (y + (x \wedge y))),\\ &x \Longleftrightarrow y \rightarrow x + y + true,\\ &x \Longrightarrow y \rightarrow (true + (x + (x \wedge y))),\\ &\neg x \rightarrow (x + true)\}.\end{aligned}$$

Les ensembles BH et $E_{\wedge,+}$ sont en particulier essentiels pour traiter des prédicats que l'utilisateur déclare transitifs, réflexifs ou irréflexifs tels les prédicats d'ordre $\leq$ et $<$ ou encore le prédicat d'équivalence $\Longleftrightarrow$. La difficulté réside dans le fait que ces propriétés ne sont pas traduisibles par des règles de réécriture sans éviter les problèmes de terminaison. REVEUR4 comporte à cet effet des procédures spécialisées permettant d'effectuer des preuves de contextes en exploitant ce type d'information concernant les prédicats. Pour la propriété de transitivité, la méthode utilisée dans REVEUR4 consiste à saturer une proposition par toutes ses conséquences par transitivité. Si la proposition obtenue par saturation comporte le littéral $pr(x, x)$ où pr est un prédicat quelconque, la procédure remplace le terme $pr(x, x)$ par $true$ si le prédicat pr est réflexif, et par $false$ dans le cas contraire.

Exemple 3.9 *Soit le prédicat $<$ et soit la proposition*

$$p = (x < y) \wedge (y < x);$$

Puisque $<$ est transitif, la procédure de saturation calcule, par fermeture transitive, la proposition

$$p' = (x < y) \wedge (y < x) \wedge (x < x) \wedge (y < y).$$

A présent, puisque $<$ n'est pas réflexif, REVEUR4 délivre une forme normale pour p égale à $false$. $\diamond$

Pour traiter le prédicat d'équivalence, REVEUR4 fait appel à des procédures de réduction et de filtrage spécifiques, en utilisant un symbole d'opérateur particulier, noté EQ, dont l'arité est variable. Toutefois, seule une forme restreinte d'équations comportant le prédicat d'équivalence est permise. Par exemple, l'équation

$$EQ(a, b) \Rightarrow f(a) = f(b)$$

est valide dans l'algèbre initiale, mais la méthode préconisée dans REVEUR4 échoue dans cette preuve de validité.

Ces procédures spécifiques traitent certains exemples correctement. Cependant, ce sont

des procédures ad-hoc et aucune théorie correcte et complète n'est sous-jacente à leur traitement.

6.2 Des exemples de preuve de confluence sur les termes clos

Dans chaque exemple qui suit, nous supposons que la spécification correspondante comporte le système d'équations $E_{\wedge,+}$ et le système de réécriture BH.

Exemple 3.10 *Le premier exemple est une spécification à trois niveaux de hiérarchie et spécifie le type* **ensemble**. *Nous montrons dans cet exemple comment calculer un ensemble de formes normales contextuelles d'un c-terme.*
Soit $SP = (SP_0,\ SP_1,\ SP_2)$ *la spécification des ensembles ;*

$SP_0 = (S_0,\ F_0,\ E_0)$;
$S_0 = \{bool,\ ens,\ elem\}$;
$F_0 = \{true,\ false :\rightarrow bool,$
$\quad \neg : bool \rightarrow bool,\ \wedge : bool \times bool \rightarrow bool,$
$\quad 0 :\rightarrow elem,\ s : elem \rightarrow elem,$
$\quad \doteq : elem \times elem \rightarrow bool,$
$\quad \emptyset :\rightarrow ens,\ \cup : ens \times ens \rightarrow ens\}$;
$E_0 = \{(0 \doteq 0) = true,$
$\quad (0 \doteq s(x)) = false,$
$\quad (s(x) \doteq 0) = false,$
$\quad (s(x) \doteq s(y)) = (x \doteq y)\}.$

$SP_1 = (S_1,\ F_1,\ E_1)$;
$S_1 = S_0$;
$F_1 = F_0 \oplus \{\in : elem \times ens \rightarrow bool\}$;
$E_1 = E_0 \oplus \{x \in \emptyset = false,$
$\quad (x \doteq y) \Rightarrow y \in (\{x\} \cup u) = true,$
$\quad \neg(x \doteq y) \Rightarrow y \in (\{x\} \cup u) = y \in u\}.$

$SP_2 = (S_2,\ F_2,\ E_2)$;
$S_2 = S_1$;
$F_2 = F_1 \oplus \{\subset : ens \times ens \rightarrow bool\}$;
$E_2 = E_1 \oplus \{\emptyset \subset v = true,$
$\quad x \in v \Rightarrow (\{x\} \cup u) \subset v = u \subset v,$
$\quad \neg(x \in v) \Rightarrow (\{x\} \cup u) \subset v = false\}.$

Remarque : Nous utilisons deux symboles d'égalité différents, le méta-symbole $=$ pour relier deux membres d'une même équation et $\doteq$ qui représente l'égalité dans la sorte *elem*.

Pour chaque i, $1 \leq i \leq 3$, on obtient les règles de R_i en orientant les équations de gauche à droite.

Soit le terme $t = (\{x\} \cup u) \subset (\{y\} \cup v)$;
$(true :: t) \xrightarrow{c}_{H,R} (x \in (\{y\} \cup v) :: u \subset (\{y\} \cup v))$,
$(true :: t) \xrightarrow{c}_{H,R} (\neg(x \in (\{y\} \cup v)) :: false)$
$(true :: x \in (\{y\} \cup v)) \xrightarrow{c}_{H,R} ((y \doteq x) :: true)$,
$(true :: x \in (\{y\} \cup v)) \xrightarrow{c}_{H,R} (\neg(y \doteq x) :: x \in v)$.

Nous avons alors,
$(true :: t) \xrightarrow{c}_{H,R} ((y \doteq x) \wedge true :: u \subset (\{y\} \cup v))$,
$(true :: t) \xrightarrow{c}_{H,R} (\neg(y \doteq x) \wedge (x \in v) :: u \subset (\{y\} \cup v))$,
$(true :: t) \xrightarrow{c}_{H,R} ((y \doteq x) \wedge \neg true :: false)$,
$(true :: t) \xrightarrow{c}_{H,R} (\neg(y \doteq x) \wedge \neg(x \in v) :: false)\}$.

En utilisant le système BH, on obtient l'ensemble suivant de formes normales contextuelles de t :
$EFNC(t) = \{((y \doteq x) :: u \subset (\{y\} \cup v)),$
$(\neg(y \doteq x) \wedge (x \in v) :: u \subset (\{y\} \cup v)),$
$(\neg(y \doteq x) \wedge \neg(x \in v) :: false)\}$. $\diamond$

Exemple 3.11 *Cet exemple définit la fonction* ***infimum*** *sur des éléments dont la sorte est munie de prédicats d'ordre. L'intérêt de cet exemple est de présenter une preuve par réécriture contextuelle.*

Soit $SP = (SP_0, SP_1, SP_2)$ la spécification de l'infimum de deux éléments ;

$SP_0 = (S_0, F_0, E_0)$;
$S_0 = \{bool, elem\}$;
$F_0 = \{true, false :\rightarrow bool,$
$\neg : bool \rightarrow bool, \wedge : bool \times bool \rightarrow bool,$
$0 :\rightarrow elem, s : elem \rightarrow elem,$
$\doteq, \leq, < : elem \times elem \rightarrow bool\}$;
$E_0 = \{(0 \doteq 0) = true,$
$(0 \doteq s(x)) = false,$
$(s(x) \doteq 0) = false,$
$(s(x) \doteq s(y)) = (x \doteq y),$
$(x \leq y) \wedge (y \leq x) = (x \doteq y),$
$(x \leq y) \wedge (y < x) = false,$
$(x < y) \wedge (y < x) = false\}$.

Remarque : *A ce niveau, nous avons besoin de certains axiomes exprimant les propriétés des prédicats utilisés, notamment la transitivité de $\leq$ et $<$ et la réflexivité de $\leq$. Certains de ces axiomes tels que la transitivité, ne peuvent pas être orientés. C'est la raison pour laquelle nous utilisons dans REVEUR4, une méthode qui consiste à construire la fermeture transitive (resp réflexive et transitive) de toute expression*

comportant un prédicat transitif (resp réflexif et transitif).

$SP_1 = (S_1,\ F_1,\ E_1)$;
$S_1 = S_0$;
$F_1 = F_0 \oplus \{infeq : elem \times elem \rightarrow bool\}$;
$E_1 = E_0 \oplus \{(x \leq y) \Rightarrow infeq(x,\ y) = true,$
$\qquad (y\ <\ x) \Rightarrow infeq(x,\ y) = false\}.$

$SP_2 = (S_2,\ F_2,\ E_2)$;
$S_2 = S_1$;
$F_2 = F_1 \oplus \{inf : elem \times elem \rightarrow elem\}$;
$E_2 = E_1 \oplus \{infeq(x,\ y) \Rightarrow inf(x,\ y) = x,$
$\qquad \neg infeq(x,\ y) \Rightarrow inf(x,\ y) = y\}.$

Pour chaque i, $1 \leq i \leq 3$, on obtient les règles de R_i en orientant les équations de E_i de gauche à droite.

On se propose de prouver la propriété de commutativité de la fonction ***infimum***. Soit l'équation $inf(u,\ v) = inf(v,\ u)$. D'une part,

$(true\ ::\ inf(u,\ v)) \xrightarrow{c}_{H,R} (infeq(u,\ v)\ ::\ u)$,
$(true\ ::\ inf(u,\ v)) \xrightarrow{c}_{H,R} (\neg infeq(u,\ v)\ ::\ v)$;
et

$(true\ ::\ infeq(u,\ v)) \xrightarrow{c}_{H,R} (u \leq v\ ::\ true)$,
$(true\ ::\ infeq(u,\ v)) \xrightarrow{c}_{H,R} (v\ <\ u\ ::\ false)$;
alors

$(true\ ::\ inf(u,\ v)) \xrightarrow{c}_{H,R} ((u \leq v)\ \wedge\ true\ ::\ u)$
$(true\ ::\ inf(u,\ v)) \xrightarrow{c}_{H,R} ((v\ <\ u)\ \wedge\ false\ ::\ u)$
$(true\ ::\ inf(u,\ v)) \xrightarrow{c}_{H,R} ((u \leq v)\ \wedge\ \neg true\ ::\ v)$
$(true\ ::\ inf(u,\ v)) \xrightarrow{c}_{H,R} ((v\ <\ u)\ \wedge\ \neg false\ ::\ v)$.

En utilisant le système BH, on obtient l'ensemble suivant de formes normales contextuelles du c-terme $(true\ ::\ inf(u,\ v))$:

$$EFNC_1 = \{(u \leq v\ ::\ u),\ (v\ <\ u\ ::\ v)\}.$$

D'autre part, puisque u et v ont des rôles symétriques, l'ensemble de formes normales contextuelles de $(true\ ::\ inf(v,\ u))$ est :
$EFNC_2 = \{(v \leq u\ ::\ v),\ (u\ <\ v\ ::\ u)\}$.

L'opérateur inf est commutatif si les ensembles $EFNC_1$ et $EFNC_2$ sont cohérents. Montrons que cette cohérence est vérifiée :

1. En utilisant la cinquième règle de R_0, l'équation

$$(u \leq v)\ \wedge\ (v \leq u) \Rightarrow u = v$$

devient

$$u\ \doteq\ v \Rightarrow u = v$$

qui est trivialement valide (voir la remarque qui suit).

2. $(u \leq v) \wedge (u < v) \Rightarrow u = u$ est triviale puisque les deux membres de l'équation sont syntaxiquement égaux.

3. Pour la même raison, l'équation

$$(v < u) \wedge (v \leq u) \Rightarrow v = v$$

est triviale.

4. en utilisant la dernière règle de R_0, l'équation

$$(v < u) \wedge (u < v) \Rightarrow v = u$$

est triviale puisque les contextes $(v < u)$ et $(u < v)$ sont opérationnellement mutuellement exclusifs. $\diamond$

Remarque : Il est important de souligner que l'utilisation du symbole d'égalité nécessite certaines précautions. En effet, utiliser $=$ dans une expression booléenne pourrait produire des équations syntaxiquement incorrectes ; par exemple,

$$(s(x) = s(y)) = (x = y)$$

C'est pourquoi nous introduisons un symbole spécifique $\doteq$. A présent nous avons besoin d'exprimer les relations qui peuvent exister entre $\doteq$ et $=$. Ceci est traduit pas les axiomes

$$x \doteq y \Rightarrow x = y \quad et \quad (x \doteq x) = true$$

La première règle étant conditionnelle, elle ne peut donc pas être introduite dans E_0. A cet effet, un traitement spécifique est effectué dans REVEUR4. Pour les mêmes raisons, dans [79], l'auteur montre que si on se place dans une approche de *Log*-algèbres, i.e. si on considère des algèbres à deux booléens, ces axiomes suffisent à exprimer l'égalité dans une sorte quelconque.

Exemple 3.12 *Cet exemple, inspiré de [35], est intéressant. En effet, D'une part, d'un point de vue technique, il montre clairement le processus de complétion et d'autre part, d'un point de vue théorique, il illustre le procédé utilisé pour faire des preuves paramétrées.*

Soit $SP = (SP_0,\ SP_1)$ *la spécification de la liste ordonnée d'entiers;*

$SP_0 = (S_0,\ F_0,\ E_0)$;
$S_0 = \{nat,\ list\}$;
$F_0 = \{true,\ false :\rightarrow bool,$
$\neg : bool \rightarrow bool,\ \wedge : bool \times bool \rightarrow bool,$
$0 :\rightarrow nat,\ s : nat \rightarrow nat,$
$<\ ,\ \leq : nat \times nat \rightarrow bool,$
$emptyl :\rightarrow list,\ cons : nat \times list \rightarrow list\}$;

$$E_0 = \{0 \doteq 0 = true, \quad 0 \leq 0 = true, \quad 0 < 0 = false,$$
$$0 \doteq s(x) = false, \quad 0 \leq s(x) = true, \quad 0 < s(x) = true,$$
$$s(x) \doteq 0 = false, \quad s(x) \leq 0 = false, \quad s(x) < 0 = false,$$
$$s(x) \doteq s(y) = x \doteq y, \quad s(x) \leq s(y) = x \leq y, \quad s(x) < s(y) = x < y,$$
$$(x \leq y) \wedge (y < x) = false,$$
$$(x \leq y) \vee (y < x) = true\}.$$

$SP_1 = (S_1,\ F_1,\ E_1)$;
$S_1 = S_0$;
$F_1 = F_0 \oplus \{ordered : list \rightarrow bool,$
$insert : nat \times list \rightarrow list\}$;
$E_1 = E_0 \oplus \{ordered(emptyl) = true,$
$ordered(cons(x,\ emptyl)) = true,$
$(x \leq y) \Rightarrow ordered(cons(x,\ cons(y,\ u))) = ordered(cons(y,\ u)),$
$(y < x) \Rightarrow ordered(cons(x,\ cons(y,\ u))) = false,$
$insert(x,\ emptyl) = cons(x,\ emptyl),$
$(x \leq y) \Rightarrow insert(x,\ cons(y,\ u)) = cons(x,\ cons(y,\ u)),$
$(x < y) \Rightarrow insert(x,\ cons(y,\ u)) = cons(y,\ insert(x,\ u)),$
$ordered(insert(x,\ u)) = ordered(u)\}.$

Pour $i = 1,\ 2$, *on obtient les règles de* R_i *en orientant les équations de* E_i *de gauche à droite.*

Lorsque nous exécutons la procédure de Knuth-Bendix sur cet exemple, le système initial est complété par l'ajout de la règle :

$$(y < x) \Rightarrow ordered(cons(y, insert(x, u))) \rightarrow ordered(cons(y, u)).$$

Le système est confluent sur les termes clos et nous pouvons alors établir la validité inductive de l'équation :

$$ordered(insert(x, u)) = ordered(u). \diamond$$

Remarque : Cet exemple constitue une instanciation de spécification. La spécification paramétrée correspondante ne contient ni les constructeurs 0 et s ni les définitions de $\doteq$, $\leq$ et $<$ sur 0 et s. Elle est construite à partir d'un type *elem* muni de prédicats d'ordre. Sous sa forme paramétrée, l'exemple tourne sur REVEUR4 et on effectue alors une preuve de confluence paramétrée sur les termes clos. Des résultats similaires peuvent être trouvés dans [35]. Ganzinger présente une procédure de complétion des spécifications conditionnelles équationnelles, où les préconditions sont sous forme de conjonctions d'égalités. L'auteur propose une application aux spécifications paramétrées qui est néanmoins complexe à réaliser en pratique. En effet, ses résultats nécessitent des conditions de consistance et de complétude des spécifications paramétrées. Nos préoccupations se rejoignent quant à la formalisation de ces résultats dans le modèle paramétré.

Exemple 3.13 *Le dernier exemple définit la spécification du type* **pile** *qui comporte des constantes d'erreur,* $errs$ *et* $erre$*, représentant respectivement* la pile erreur *et* l'élément erreur.

Soit $SP = (SP_0,\ SP_1)$ *la spécification de la pile avec erreurs;*
$SP_0 = (S_0,\ F_0,\ E_0)$;
$S_0 = \{pile,\ elem\}$;
$F_0 = \{true,\ false :\rightarrow bool,$
$\neg : bool \rightarrow bool,\ \wedge : bool \times bool \rightarrow bool,$
$errs\ ,\ pile\text{–}vide :\rightarrow pile,\ erre :\ \rightarrow elem,$
$push : pile \times elem \rightarrow pile,$
$defs : pile \rightarrow bool,\ defe : elem \rightarrow bool\}$;
$E_0 = \{push(errs,\ x) = errs,\ push(u,\ erre) = errs,$
$defs(pile\text{–}vide) = true,$
$defs(push(u,\ x)) = defs(u)\ \wedge\ defe(x),$
$defs(errs) = false\}.$

$SP_1 = (S_1,\ F_1,\ E_1)$;
$S_1 = S_0$;
$F_1 = F_0 \oplus \{pop : pile \rightarrow pile,\ top : pile \rightarrow elem,\ is\text{–}empty : pile \rightarrow bool\}$;
$E_1 = E_0 \oplus \{defs(u)\ \wedge\ defe(x) \Rightarrow pop(push(u,\ x)) = u,$
$defs(u)\ \wedge\ defe(x) \Rightarrow top(push(u,\ x)) = x,$
$is\text{–}empty(pile\text{–}vide) = true,$
$defs(u)\ \wedge\ defe(x) \Rightarrow is\text{–}empty(push(u,\ x)) = false,$
$top(errs) = erre,$
$pop(errs) = errs,$
$is\text{–}empty(errs) = false,$
$pop(pile\text{–}vide) = errs,$
$top(pile\text{–}vide) = erre\}.$

Pour $i = 1,\ 2$*, on obtient les règles de* R_i *en orientant les équations de* E_i *de gauche à droite.*

Il est intéressant de noter que cet exemple n'est pas bien couvert. Dans ce cas, ceci ne porte pas à conséquence car les équations non bien couvertes ne sont utilisées ni pour normaliser, ni pour calculer des paires critiques. La procédure de Knuth-Bendix prouve l'équation suivante qui est la duale de la dernière équation de R_0,

$$defe(erre) = false. \diamond$$

7 Conclusion

Nous avons étudié dans ce chapitre une méthode effective de preuve de confluence sur les termes clos des systèmes de réécriture conditionnelle hiérarchique. Le cadre hiérarchique dans lequel nous nous sommes placée impose des restrictions quant à la définition d'une spécification et de ce fait, la souplesse d'utilisation d'un tel type de systèmes est altérée. Néanmoins, la hiérarchie constitue un cadre pratique pour définir des spécifications volumineuses. D'autre part, le test de confluence sur les termes clos est plus simple à mettre en pratique et la vérification est effectuée de façon modulaire, par niveaux successifs de hiérarchie.

Nous avons par ailleurs caractérisé la propriété de consistance d'une spécification conditionnelle hiérarchique par rapport à la sorte booléenne. Cette consistance est essentielle dans toute approche avec booléens, dans laquelle une spécification conditionnelle a ses équations définies à partir de préconditions de sorte booléenne. L'existence d'une algèbre initiale fidèle par rapport aux booléens est alors garantie, si la spécification est consistante et complète, au sens de la relation hiérarchique $\rightarrow_{H,R}$, par rapport aux booléens. Cette propriété de complétude est acquise dès que la complétude suffisante hiérarchique est vérifiée dans la spécification toute entière, puisqu'elle en est un cas particulier. Des conditions suffisantes sont établies pour obtenir une méthode effective pour tester cette propriété. Ces conditions sont cependant encore trop restrictives et il serait intéressant de les assouplir à l'avenir.

La propriété de confluence a également été étudiée par Ganzinger [35] et un résultat de confluence sur les termes clos a été établi par l'auteur dans les systèmes de réécriture réducteurs, qui constituent une classe plus large que celle des systèmes hiérarchiques. Les préconditions de règles ont une forme équationnelle: $\wedge_{i=1}^{n}\ t_i = t'_i,\ t'_i \in \{true,\ false\}$. L'auteur utilise une forme de réécriture contextuelle qui diffère de celle utilisée dans ce travail. En effet, cette relation est définie de telle façon que l'applicabilité d'une règle soit inférée à partir d'un ensemble C d'hypothèses de convergence et de l'ensemble de toutes les assertions de convergence dérivées par clôture réflexive et symétrique et par clôture par congruence de C. Elle est en réalité définie sur le même principe que la réduction récursive, la justification étant effectuée de façon plus souple puisqu'il est alors possible d'utiliser les assertions auxiliaires de C.
La preuve de convergence des paires critiques que le test de confluence effectue utilise une notion de bonne couverture exprimée par des prédicats de recouvrement et des assertions négatives. Ces prédicats sont des théorèmes inductifs du modèle initial de la spécification dont la validité est justifiée par des techniques de preuve indépendantes. Ces prédicats sont de deux formes, les prédicats de recouvrement, nécessaires pour éclater une paire critique en deux nouvelles équations et les assertions négatives, utiles pour prouver l'insatisfiabilité d'un contexte.
Ganzinger propose par ailleurs une application de son algorithme de confluence sur les termes clos aux spécifications paramétrées, à condition que la spécification paramétrée soit consistante et suffisamment complète. La complétude suffisante est en effet utile d'une part, pour justifier l'éclatement de la paire critique et d'autre part, combinée à la consistance, pour prouver que les paires critiques entre les règles du paramètre actuel restent convergentes après le passage de paramètres. Cependant, l'auteur ne propose aucune méthode pour tester la consistance et la complétude suffisante d'une spécification conditionnelle. Dans [35], il ne se penche pas non plus sur les problèmes de fidélité par

rapport à la sorte booléenne, que nous avons caractérisés dans ce présent travail par les propriétés de consistance et de complétude par rapport aux booléens.

Divers problèmes restent à résoudre dans le domaine de la réécriture conditionnelle. Ils consistent principalement à intégrer l'utilisation de formules du premier ordre, indispensables pour traiter les contextes et effectuer des preuves inductives au niveau inconditionnel. Ces preuves sont effectuées à l'aide d'assertions additionnelles qui peuvent être justifiées par des méthodes indépendantes. De plus, ces assertions constituent des paramètres de la spécification conditionnelle considérée et leur utilisation est à l'heure actuelle encore empirique et non-déterministe.

Une autre perspective de recherche est d'étudier les conditions d'applicabilité du théorème de confluence sur les termes clos pour les autres types de relations hiérarchiques que nous avons présentés. Il serait intéressant d'explorer cette voie de recherche car ces relations ont une portée d'utilisation plus étendue que celle que nous avons étudiée dans ce travail.

Le théorème principal de confluence sur les termes clos que nous avons établi dans ce présent chapitre nécessite cependant pour sa validité de vérifier des propriétés relativement restrictives. Ceci est principalement dû à la complexité du cadre conditionnel. Nous avons par la suite voulu assouplir ces conditions d'applicabilité en considérant essentiellement une classe plus large de systèmes de réécriture conditionnelle et une forme plus souple du processus de simplification. Ceci nous a amenée à établir un second test de confluence, différent sous plusieurs aspects de celui présenté dans ce chapitre. Ce test constitue le résultat central du chapitre qui suit.

CHAPITRE 4

UNE PREUVE DE CONFLUENCE SUR LES TERMES CLOS FONDEE SUR LE RAISONNEMENT PAR CAS.

Chapitre 4

Une preuve de confluence sur les termes clos fondée sur le raisonnement par cas.

1 Introduction

Ce chapitre est consacré à l'étude de la preuve de confluence sur les termes clos fondée sur un processus de réécriture spécifique. L'étude que nous entreprenons vient conforter une voie de recherche proposée par Kaplan et Rémy dans [58] où les auteurs suggèrent d'une part, de combiner leurs approches respectives en formalisant le processus de réécriture contextuelle dans les systèmes réducteurs et d'autre part, d'intégrer un raisonnement par cas dont la formule est plus souple que celle définie dans le chapitre 3. Pour présenter cette preuve de confluence, nous utilisons le formalisme des règles d'inférence introduit par Bachmair, Dershowitz et Hsiang [2]. Ce formalisme est utile pour présenter les concepts clés développés dans nos travaux.

Nous considérons une classe de systèmes de réécriture conditionnelle plus générale que celle des systèmes réducteurs. Ce sont les systèmes dits *décroissants* définis au chapitre1, page 32. Dans ces systèmes, le membre gauche d'une règle conditionnelle est plus grand que le membre droit, et que la précondition correspondants, la première comparaison se faisant en utilisant un ordre de réduction et la seconde en utilisant une extension de cet ordre qui contient l'ordre de sous-terme strict. Cette approche est celle de Dershowitz et Okada [19].

La confluence sur les termes clos d'un système de réécriture conditionnelle R est vérifiée dès lors que toutes les paires critiques contextuelles calculées entre les règles de R sont convergentes sur les termes clos. Cette convergence est testée par une méthode spécifique relativement aisée à mettre en œuvre. Pour chaque paire critique $C \Rightarrow M = N$, le traitement est le suivant : si la paire critique vérifie certains critères (que nous spécifierons et justifierons dans notre résultat principal), elle est convergente sur les termes clos. Dans le cas contraire, il s'agit de simplifier $C \Rightarrow M = N$ en lui appliquant successivement une ou plusieurs règles d'inférence et de tester après chaque simplification, la convergence des équations nouvelles. Ces règles de simplification sont fondées sur le processus du raisonnement par cas dont le principe est que la simplification d'un terme s'accompagne de l'enrichissement de son contexte par la condition d'applicabilité de la règle simplificatrice.

La validité d'un contexte plus riche est ainsi plus souvent facile à vérifier dans le sens où ce contexte contient plus d'informations que le contexte initial. De plus, la simplification a la particularité d'opérer aussi bien sur la conséquence que sur la condition de la paire critique, augmentant ainsi la puissance du test de convergence, notamment en ce qui concerne les paires critiques non-décroissantes.

Le plan de ce chapitre s'articule comme suit : le premier paragraphe introduit une nouvelle forme de réécriture contextuelle définie à partir du raisonnement par cas. Nous insistons particulièrement sur l'importance de ce processus de réécriture et nous mettons en évidence les problèmes de terminaison qu'il peut poser, principalement dûs à la branche complémentaire de réécriture qui produit un schéma classique de dérivation cyclique si l'on omet de prendre certaines précautions. La partie qui suit présente, sous forme de règles d'inférence, le processus de simplification contextuelle utilisé pour vérifier la convergence sur les termes clos des paires critiques contextuelles. Les règles d'inférence sont principalement partagées en deux groupes, les règles *de base*, et les règles ***simplificatrices*** qui agissent uniquement sur des équations. Dans le paragraphe suivant, nous établissons le théorème principal de confluence sur les termes clos des systèmes de réécriture décroissants, dans lesquels, les constantes de vérité *true* et *false* sont irréductibles. Ce théorème permet par ailleurs de vérifier que la spécification correspondante est consistante par rapport aux booléens, i.e. que les deux valeurs de vérité sont distinctes. La dernière partie illustre par des exemples le fonctionnement du test de confluence sur les termes clos. Elle présente par ailleurs une comparaison entre l'étude théorique élaborée dans ce chapitre et les résultats hiérarchiques du chapitre précédent. Les travaux sont en effet différents en un certain nombre de points concernant essentiellement une plus grande souplesse du processus de raisonnement par cas utilisé et une classe plus large de systèmes de réécriture conditionnelle.

2 Réécriture contextuelle et raisonnement par cas

2.1 Concepts conditionnels

Nous étudions dans ce chapitre les équations conditionnelles de la forme $p \Rightarrow g = d$ où p est une conjonction d'équations booléennes (ou CEB) de la forme

$$\forall i in[1 \ldots n],\ \wedge_{i=1}^{n}\ p_i = p'_i,\ p'_i \in \{true,\ false\}$$

p_i est un littéral de la forme $\pi(t_1, \ldots,\ t_k)$. Dans tout ce qui suit, p est noté sous la forme $\vec{p}$.

Notre objectif est d'établir une méthode de preuve de confluence sur les termes clos fondée sur le raisonnement par cas. Pour ce faire, nous visons à obtenir des systèmes de réécriture conditionnelle où les équations sont orientées afin de rendre leur utilisation plus déterministe. Nous supposons de plus, que tout système de réécriture conditionnelle R satisfait les hypothèses suivantes :

- pour toute règle de réécriture $\vec{p} \Rightarrow g \rightarrow d$, les ensembles $Var(d)$ et $Var(\vec{p})$ sont contenus dans $Var(g)$.
  $Var(\vec{p})$ correspond à l'ensemble $\cup_{i=1}^{n}\ \{Var(p_i)\}$.
- si $\vec{p} \Rightarrow g \rightarrow d$ est dans R, alors g ne contient aucun des connecteurs logiques $\wedge$, $\vee$ et $\neg$.

Remarque : Imposer que le membre gauche d'une règle conditionnelle quelconque ne contienne aucun des connecteurs logiques $\wedge$, $\vee$ et $\neg$ est dû à la raison suivante, ces symboles de fonction sont prédéfinis dans toute spécification booléenne par leur table de vérité et leur signification ne doit en aucun cas être susceptible d'être altérée. Cette restriction est étroitement liée à la propriété de fidélité par rapport aux booléens.

Nous reprenons dans cette partie les notations adaptées dans le second chapitre de ce document en ce qui concerne les préconditions de règles. Nous utilisons précisément les constantes TT et FF et la notation de CEB pour faciliter l'écriture dans certaines situations.

L'avantage de focaliser notre étude sur des préconditions booléennes sous forme de CEB est que nous disposons, pour cette forme de préconditions, de différentes heuristiques qui permettent de prouver l'insatisfiabilité d'un contexte. Notre objectif consiste en particulier, à mettre en évidence l'importance des booléens dans les spécifications conditionnelles et à établir de façon précise des conditions qui garantissent que l'algèbre initiale correspondante à la spécification conditionnelle est isomorphe à l'ensemble
$(\{bool\},\ \{true,\ false\},\ \emptyset)$. On dira alors que ladite spécification est fidèle par rapport aux booléens.
Nous choisissons de travailler avec des systèmes de réécriture décroissants tels qu'ils ont été décrits par Dershowitz et Okada (voir chapitre 1, def 1.35). Ces systèmes sont une généralisation des systèmes réducteurs et simplifiants, mais n'introduisent cependant pas une généralité excessive [21] et permettent donc d'établir les preuves nécessaires. Une autre particularité de ce travail est que l'on adopte un processus de réécriture appelé réécriture contextuelle. Ce processus est mieux adapté à la preuve d'équations conditionnelles ; il introduit les notions de contexte et de terme contextuel et il se distingue sous trois aspects :

1. La réécriture dans un contexte. Elle consiste à réécrire t en t' moyennant la validité d'un certain contexte c. Cet aspect exprime en effet, de façon fidèle le principe du raisonnement conditionnel.

2. La réécriture du contexte. Le contexte étant lui-même un terme, il est possible de le réécrire, et par là-même de faciliter la preuve de convergence des paires critiques et de simplifier les preuves du premier ordre.

3. Le raisonnement par cas. Il s'agit de réécrire t en différents termes $t_1, \ldots, t_n$, chacune de ces réductions étant respectivement valide dans les contextes $c_1, \ldots, c_n$. Le raisonnement par cas est performant en particulier pour prouver la convergence d'une paire critique contextuelle $C \Rightarrow M = N$. En effet, le contexte C est partitionné en sous-cas qui forment de nouveaux contextes dont les preuves sont souvent plus simples à établir.

L'utilisation de chacun de ces trois aspects dépend de la classe d'algèbres considérée. Le choix de cette classe d'algèbres est particulièrement important pour justifier la validité du raisonnement par cas.

Remarque : Les systèmes que nous définissons dans ce chapitre utilisent des préconditions dont la forme syntaxique est spécifique. Dans ces systèmes, il est naturel de considérer que les constantes *true* et *false* sont plus petites dans la précédence, que tout autre opérateur de la signature. Formellement, on suppose que l'assertion suivante est toujours vérifiée :

$$\forall t \in T(F)_{bool} \text{ tel que } t \neq true,\ false,\ t \succ true,\ false$$

Proposition 4.1 *Soit R un système de réécriture décroissant, soient $\succ$ une extension d'un ordre de réduction $>$ qui possède la propriété de sous-terme propre (voir déf 1.35), et $\succ\!\!\succ$ son extension aux multi-ensembles ;*
pour toute CEB, $\vec{c}$ de la forme $\wedge_{i=1}^{n}\ c_i = c'_i$

$$\forall i \in [1 \ldots n],\ t \succ\!\!\succ \vec{c} \text{ ssi } t \succ c_i$$

si pour tout terme clos t tel que $t \neq true,\ false,\ t \succ true,\ false$.

Preuve : par définition, $t \succ\!\!\succ \wedge_{i=1}^{n}\ c_i = c'_i$ ssi pour tout i dans $[1 \ldots n]$, $t \succ c_i$ et $t \succ c'_i$. La preuve est alors immédiate puisque tout terme clos distinct de *true* et *false* est plus grand par l'ordre $\succ$ que *true* et *false*. La seconde comparaison est par conséquent toujours vérifiée.

On peut voir l'opérateur de conjonction $\wedge$ comme un opérateur d'union sur les multi-ensembles, et voir une CEB, $\vec{c}$ comme un multi-ensemble de la forme

$$\vec{c} = \{c_1 = c'_1,\ c_2 = c'_2, \ldots,\ c_n = c'_n\}$$

Cette extension, par définition, valide l'assertion suivante :

$$\forall \vec{C},\ \vec{C'},\ \vec{C}",\ \vec{C} \succ\!\!\succ \vec{C'}\ \wedge\ \vec{C}" \text{ ssi } \vec{C} \succ\!\!\succ \vec{C'} \text{ et } \vec{C} \succ\!\!\succ \vec{C}".$$

Par ailleurs, une propriété importante de l'ordre $\succ\!\!\succ$ est qu'il est aussi un ordre de réduction, puiqu'il est une extension d'un ordre de réduction (voir [18]).

Une particularité de ce travail est qu'il introduit le raisonnement par cas dans le processus de réécriture conditionnelle. Dans une étape intermédiaire, le raisonnement par cas

a été étudié dans un premier temps en faisant l'hypothèse que le système considéré est bien couvert (voir chapitre 3). La propriété de bonne couverture assure que toute réécriture d'un terme contextuel est complète, en imposant que pour tout sous-ensemble maximal de règles de la forme $\{\vec{p}_i \Rightarrow g \rightarrow d_i,\ i \in I\}$ dans R, et pour toute substitution close σ, la condition

$$(**) \qquad \vee_{i \in I}\ \sigma\vec{p}_i =_R TT$$

soit satisfaite. Toutefois, cette approche impose des restrictions syntaxiques pour définir un symbole de fonction. De plus, la condition (**) n'est pas même toujours préservée au cours du processus de complétion car le calcul des paires critiques ajoute de nouvelles règles au système de réécriture. Pour ce qui concerne ce travail, nous ne faisons aucune hypothèse de bonne couverture du système initial ; nous définissons plutôt un processus de réécriture contextuelle complet appelé raisonnement par cas. Ce processus est traduit par la suite par des règles d'inférence.

Nos résultats, dans ce chapitre, se basent également sur la relation de réécriture contextuelle telle qu'elle a été définie dans le chapitre précédent. Rappelons que cette relation est définie sur des termes accompagnés d'une précondition booléenne et appelés termes contextuels. De plus, il est possible de réécrire aussi bien le terme que son contexte d'une façon similaire. Plus formellement, $(\vec{c} :: s) \xrightarrow{c}_R (\vec{c}' \wedge \sigma\vec{p} :: t)$ s'il existe une règle $\vec{p} \Rightarrow g \rightarrow d$ dans R, une substitution σ et

- soit une occurrence u de t telles que $t_{/u} = \sigma g,\ t' = t[u \leftarrow \sigma d],\ \vec{c}' = \vec{c}$,
- soit une occurrence ω de a avec $\vec{c} = (a = a') \wedge \vec{c}\,''$ telles que $a_{/\omega} = \sigma g,\ t' = t,\ \vec{c}' = (a[\omega \leftarrow \sigma d] = a') \wedge \vec{c}\,''$.

2.2 Le raisonnement par cas

Notre objectif est d'établir une preuve de confluence sur les termes clos des systèmes de réécriture conditionnelle. Cette preuve nécessite d'effectuer des preuves sur les termes clos. Cette sorte de preuve est particulièrement intéressante en programmation logique et dans un cadre de spécification algébrique. Le test de confluence comporte des règles de simplification d'équations, dans lesquelles une composante quelconque d'une équation, c'est-à-dire soit un des termes de sa conséquence, soit sa précondition, peut être simplifiée. Cette simplification est caractérisée par :

1. la réécriture contextuelle : plutôt que de vérifier la validité de la précondition instanciée avant d'appliquer la règle, la réécriture contextuelle consiste à ajouter cette instanciation au contexte initial. Ce processus de réécriture est étudié dans les chapitres précédents.

2. la réécriture à une certaine occurrence par une seule règle du système. En réalité, nous avons trois possibilités de réécrire un terme. La première, que nous avons choisi d'étudier, consiste à utiliser une seule règle à une occurrence donnée, ce qui est schématisé par la règle d'inférence suivante :

$$\frac{\vec{c} \Rightarrow s[\sigma g]}{\vec{c} \wedge \sigma\vec{p} \Rightarrow s[\sigma d]}$$

ssi il existe **une règle** $\vec{p} \Rightarrow g \rightarrow d$ dans R, une occurrence u et une substitution σ telles que $s_{/u} = \sigma g$.

La seconde possibilité utilise un paquet de règles partageant le même membre gauche et elle consiste à réécrire le terme à une seule et même occurrence

$$\frac{\vec{c} \Rightarrow s[\sigma g]}{\vec{c} \wedge \sigma\vec{p_1} \Rightarrow s[\sigma d_1], \ldots, \vec{c} \wedge \sigma\vec{p_n} \Rightarrow s[\sigma d_n]}$$

ssi il existe **un sous-ensemble maximal de règles** dans R de la forme $\{\vec{p_i} \Rightarrow g \rightarrow d_i,\ i \in I\}$, une occurrence u et une substitution σ tels que $s_{/u} = \sigma g$.

Enfin, la troisième possibilité de réécrire un terme considère toutes les règles pouvant réécrire ce terme, à toutes les occurrences possibles, sans aucune distinction :

$$\frac{\vec{c} \Rightarrow s}{\vec{c} \wedge \sigma_1\vec{p_1} \Rightarrow s_1, \ldots, \vec{c} \wedge \sigma_n\vec{p_n} \Rightarrow s_n}$$

ssi pour **toute règle** $\vec{p_i} \Rightarrow g_i \rightarrow d_i$ dans R pour laquelle il existe une occurrence u_i et une substitution σ_i telles que $s_{/u_i} = \sigma_i g_i$, alors $s_i = s[u_i \leftarrow \sigma_i d_i]$.

Nous avons choisi dans notre étude, d'adapter la première solution. Nous ne pensons pas que, dans un contexte logique, les trois méthodes soient différentes. Cependant, la différence peut être primordiale d'un point de vue opérationnel et pour des raisons d'efficacité.

3. Le troisième intérêt est l'introduction du cas complémentaire. En effet, contrairement à notre approche du chapitre 3, afin d'éviter l'hypothèse de bonne couverture du système, à partir de la simplification de l'une des composantes d'une équation conditionnelle $\vec{p} \Rightarrow g = d$ (i.e. de sa précondition ou d'une partie de sa conséquence), nous remplaçons $\vec{p} \Rightarrow g = d$ par une nouvelle équation et nous lui ajoutons un ensemble d'équations. Cet ensemble, dont la conséquence est identique à celle de l'équation simplifiée, correspond au complémentaire du résultat de la simplification. De cette manière, nous garantissons que le nouveau paquet d'équations obtenues a la disjonction de ses contextes équivalente à $true$. Cette branche de réécriture est appelée *cas complémentaire.*

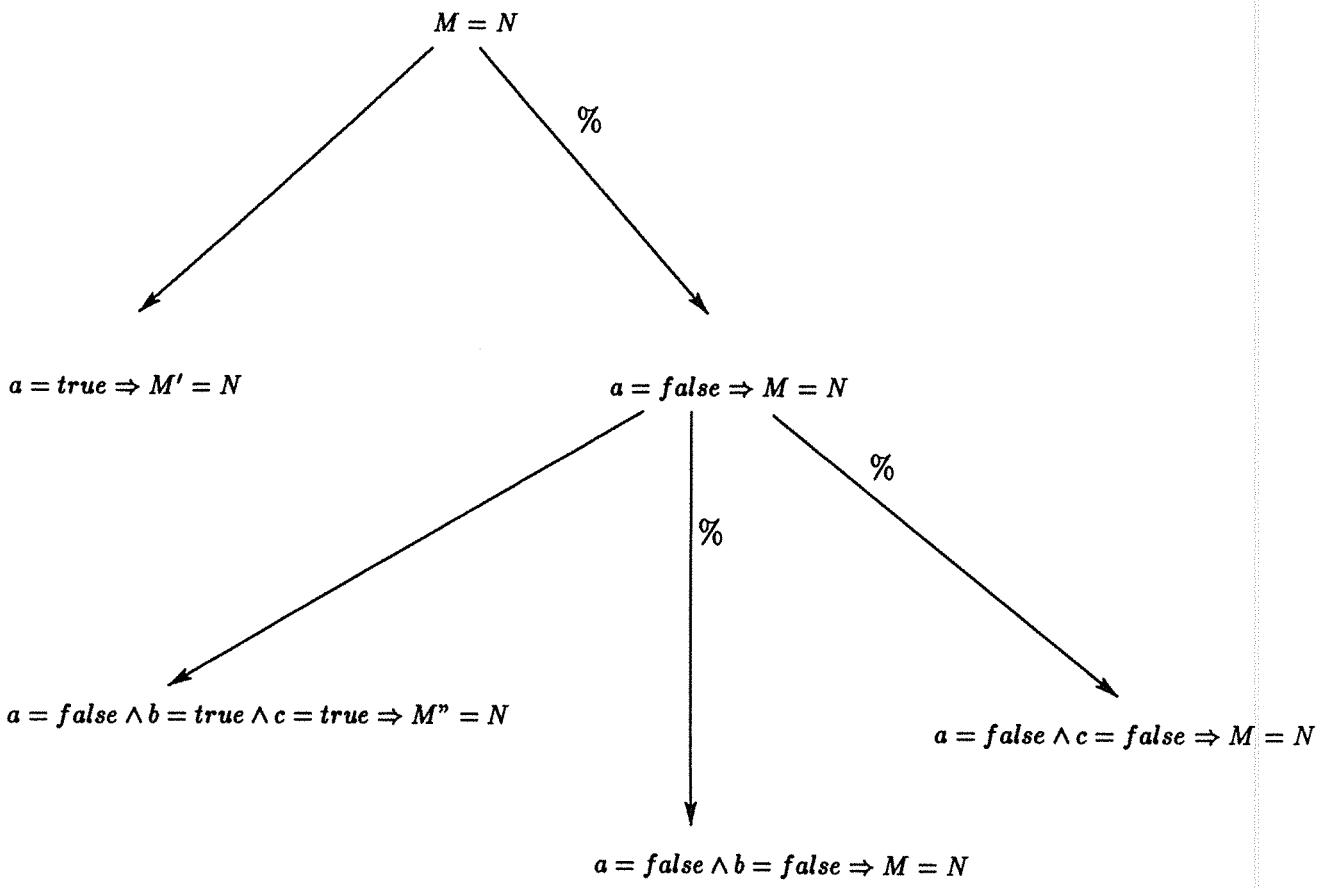
   Considérer la branche complémentaire de réécriture a également été étudié par Navarro[79]. Toutefois, l'auteur travaille sur des termes contextuels et réécrit un terme quelconque en utilisant toutes les possibilités dans la partie terme et dans la partie contexte. De plus, dans ses travaux, Navarro n'aborde pas les problèmes de terminaison inhérents à la définition de la branche complémentaire ni ceux posés par la relation de réécriture contextuelle. Ses résultats sont valides dans un cadre algébrique et aucune méthode de calculabilité n'est fournie par l'auteur.

Explicitons dans un premier temps ce processus sur un exemple simple :

Exemple 4.1 *Soit le système de réécriture R contenant les règles suivantes :*

$$\begin{cases} a = true \Rightarrow l \rightarrow r_1, \\ b = true \ \wedge \ c = true \Rightarrow l \rightarrow r_2 \end{cases}$$

et supposons qu'à l'occurrence u de M, $M_{/u} = \tau l$. La simplification est comme suit :


$M = N$
%
$a = true \Rightarrow M' = N$
$a = false \Rightarrow M = N$
%
%
%
$a = false \wedge b = true \wedge c = true \Rightarrow M'' = N$
$a = false \wedge b = false \Rightarrow M = N$
$a = false \wedge c = false \Rightarrow M = N$

où $M' = M[u \leftarrow \tau r_1]$ *et* $M'' = M[u \leftarrow \tau r_2]$. *Les branches marquées du symbole* % *sont celles qui correspondent au cas de réécriture complémentaire.*
$a = false$ *est le résultat du calcul de l'expression* $\neg(a = true)$, $b = false \vee c = false$ *celui du calcul de l'expression* $\neg(c = true \wedge b = true)$ *et l'équation* $b = false \vee c = false \Rightarrow M = N$ *est éclatée en deux.* $\diamond$

Remarque : Comme on peut le voir sur cet exemple, la branche complémentaire engendre en général, non pas une seule nouvelle équation, mais un ensemble de nouvelles équations. Ceci n'est pas surprenant puisque, de la négation d'une conjonction de contextes résulte une disjonction de nouveaux contextes.

Mise en forme canonique du contexte

L'exemple ci-dessus suscite une remarque importante. En effet, des règles additionnelles de mise en forme canonique du contexte sont nécessaires afin de s'assurer que les préconditions appartiennent toujours au langage fixé. Ces règles sont d'un autre type et nécessitent d'utiliser de nouveaux axiomes. La forme canonique n'est effectivement plus garantie notamment lorsque nous simplifions une précondition par une règle de réécriture dont le membre droit comporte un connecteur logique.

Considérons par exemple la règle $p \rightarrow q \wedge r$ et supposons que l'on veuille simplifier l'équation $p = true \Rightarrow g = d$; le résultat de la simplification produit l'équation $(q \wedge r) = true \Rightarrow g = d$, encore équivalente à l'équation

$$q = true \wedge r = true \Rightarrow g = d.$$

Supposons encore que l'on veuille calculer le complémentaire de l'expression $q_1 = true \wedge q_2 = false$. Alors

$$\neg(q_1 = true \wedge q_2 = false) = (q_1 = false) \vee (q_2 = true).$$

Pour obtenir des contextes sous la forme souhaitée, i.e. sous forme de conjonction d'équations booléennes, nous avons eu recours à des règles de transformation que nous appelons règles de *mise en forme canonique du contexte.* Ces règles mettent principalement en jeu les axiomes de distributivité de $\neg$ par rapport à $\wedge$:

$$\neg(t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n) = \neg(t_1 = t'_1) \vee \ldots \vee \neg(t_n = t'_n)$$

et les axiomes d'élimination du symbole de négation:

$$\neg(t = true) = (t = false),\ \neg(t = false) = (t = true).$$

Par la branche complémentaire, nous obtenons donc un ensemble de nouvelles équations.

$$\frac{\vec{p} \wedge \neg(t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n) \Rightarrow g = d}{\vec{p} \wedge [\neg(t_1 = t'_1) \vee \ldots \vee \neg(t_n = t'_n)] \Rightarrow g = d}$$

C'est la raison pour laquelle, dans l'écriture des règles d'inférence présentées ci-après, ce cas se traduit par l'obtention de plusieurs équations en utilisant la règle d'éclatement suivante:

$$\frac{\vec{p} \wedge (t_1 = \bar{t}'_1 \vee \ldots \vee t_n = \bar{t}'_n) \Rightarrow g = d}{\vec{p} \wedge t_1 = \bar{t}'_1 \Rightarrow g = d, \ldots, \vec{p} \wedge t_n = \bar{t}'_n \Rightarrow g = d}$$

où pour tout terme booléen $t \in \{true,\ false\}$, $\bar{t} = true$ si $t = false$ et $\bar{t} = false$ si $t = true$.

La règle d'élimination du symbole de négation $\neg$ est correcte puisque nous travaillons sur des algèbres dont le support booléen est isomorphe à l'ensemble $\{true,\ false\}$ et que ces deux constantes sont complémentaires.

Avant de présenter formellement les règles d'inférence, nous nous proposons d'illustrer par un exemple le processus de simplification.

Exemple 4.2 *Soit R un système de réécriture constitué des règles suivantes:*

$$\{q_1(x,y) = true \Rightarrow f(x,y) \rightarrow k_1(x,y),\ q_2(x,y) = true \Rightarrow f(x,y) \rightarrow k_2(x,y)\}$$

$x,\ y,\ u,\ v$ sont des variables.
L'équation $p(u,v) = true \Rightarrow g(f(u,v)) = d(u,v)$ peut être simplifiée par R et nous obtenons l'ensemble:

$$\{p(u,v) = true\ \wedge\ q_1(u,v) = true \Rightarrow g(k_1(u,v)) = d(u,v),$$

$$p(u,v) = true\ \wedge\ q_1(u,v) = false\ \wedge\ q_2(u,v) = true \Rightarrow g(k_2(u,v)) = d(u,v),$$

$$p(u,v) = true\ \wedge\ q_1(u,v) = false\ \wedge\ q_2(u,v) = false \Rightarrow g(f(u,v)) = d(u,v)\}$$

$\diamond$

2.3 Traitement de la branche complémentaire

Il est important de noter que le cas complémentaire de réécriture pose un problème de terminaison. En effet, lorsque nous simplifions par exemple le terme g de l'équation $\vec{p} \Rightarrow g = d$ par la règle $t_1 = t'_1\ \wedge\ \ldots\ \wedge\ t_n = t'_n \Rightarrow l \rightarrow r$ à l'occurrence ω avec la substitution τ, nous obtenons, par le cas complémentaire, un ensemble d'équations plus simples de la forme

$$\vec{p}\ \wedge\ t_i = \vec{t'_i} \Rightarrow g = d,\ i = 1, \ldots n$$

Il est alors possible d'appliquer de nouveau la même règle sur le même terme pour simplifier cette équation et de cette façon, nous obtenons une chaîne infinie de réductions. Dans le but d'éviter cette situation, nous préconisons la solution suivante: il s'agit de mémoriser pour chaque terme, un ensemble de radicaux correspondant aux réductions possibles sur ce terme. Formellement, pour un terme t quelconque,

$$\Re(t) = \{(\vec{\beta},\ \alpha,\ \rho) \text{ tel qu'il existe } \vec{\beta} \Rightarrow \alpha \rightarrow \lambda \in R,\ u \text{ et } \rho\alpha = t_{/u}\}.$$

Par conséquent, on mémorise le membre gauche et la précondition de toute règle susceptible de simplifier le terme, ainsi que la substitution correspondante. Contrairement au cadre inconditionnel, il est nécessaire dans notre cas de mémoriser la précondition, puisque le membre gauche d'une règle conditionnelle ne permet pas de la distinguer de façon unique.

A toute composante d'une équation, est associé l'ensemble de ses radicaux. Un radical $(\vec{\beta},\ \alpha,\ \rho)$ est supprimé de $\Re(t)$ si t a fait l'objet d'une simplification par une règle de membre gauche α et de précondition $\vec{\beta}$, avec la substitution ρ.

De cette manière, si l'équation $\vec{p} \Rightarrow g = d$ est simplifiée dans g par la règle $\wedge_{i=1}^{n}\ t_i =$

$t'_i \Rightarrow l \rightarrow r$, les équations $\{\vec{p} \wedge t_i = \vec{t}'_i \Rightarrow g = d\}_{i=1}^n$, engendrées par la branche complémentaire, voient toutes l'ensemble des redex de leur terme g diminué de la composante $(\wedge_{i=1}^n t_i = t'_i,\ l,\ \tau)$ où τ est la substitution utilisée au cours de la simplification. Ainsi, on s'interdit d'utiliser une seconde fois la même règle et la même substitution pour simplifier le terme g dans cette équation puisqu'alors, le radical $(\wedge_{i=1}^n t_i = t'_i,\ l,\ \tau)$ ne figure plus dans $\Re(g)$.

Il s'agit par conséquent au niveau du contrôle pour la branche complémentaire, de modifier l'ensemble de radicaux du terme qui a fait l'objet d'une réduction dans les autres branches, en lui ôtant le radical correspondant à la réduction.

Le traitement consiste à manipuler une structure de données liée à chaque composante t d'une équation, constituée d'un ensemble de triplets et notée $\Re(t)$. L'ensemble $\Re(t)$ est appelé ensemble des radicaux de t. Il est modifié chaque fois que le terme t est simplifié. Bien sûr, le contenu de l'ensemble $\Re(t)$ dépend fortement de l'histoire du terme t et par conséquent, de l'histoire de l'équation dans laquelle il figure. Cette association entre le terme et son ensemble de redex est notée $\Re(t) \diamond t$.

Remarque : Les termes *true* et *false* n'ont pas de radicaux puisqu'ils sont irréductibles dans R. Par conséquent, puisque nous manipulons des CEB de la forme $\wedge_{i=1}^n c_i = c'_i$ dans lesquelles les c'_i sont dans $\{true,\ false\}$, on convient d'omettre les ensembles de radicaux des termes c'_i.

Exemple 4.3 *Soit le système de réécriture suivant :*

$$R = \begin{cases} p = true \ \wedge \ q = true \Rightarrow a \rightarrow c, & r_1 \\ p = false \Rightarrow a \rightarrow d & r_2 \end{cases}$$

Notons $\vec{\beta}_1$ et $\vec{\beta}_2$ les préconditions respectives des deux règles de R. Considérons l'équation $e : f(a) = b$ et calculons les ensembles de redex de ses deux termes.

$$\Re(f(a)) = \{(\vec{\beta}_1,\ a,\ \mathcal{I}d),\ (\vec{\beta}_2,\ a,\ \mathcal{I}d)\},\ \Re(b) = \emptyset$$

où $\mathcal{I}d$ est la substitution identité.

La simplification de e par r_1 engendre les équations suivantes :

1. $\emptyset \diamond p = true \wedge \emptyset \diamond q = true \Rightarrow \emptyset \diamond f(c) = \emptyset \diamond b$,
2. $\emptyset \diamond p = false \Rightarrow \{(\vec{\beta}_2,\ a,\ \mathcal{I}d)\} \diamond f(a) = \emptyset \diamond b$,
3. $\emptyset \diamond q = false \Rightarrow \{(\vec{\beta}_2,\ a,\ \mathcal{I}d)\} \diamond f(a) = \emptyset \diamond b$.

De nouveau, il est possible de simplifier les deux dernières équations par r_2, ce qui donne les deux schémas de réduction suivants :
Simplification de la seconde équation :

1. $\emptyset \diamond p = false \Rightarrow \emptyset \diamond f(d) = \emptyset \diamond b$,
2. $\emptyset \diamond p = false \wedge \emptyset \diamond p = true \Rightarrow \emptyset \diamond f(a) = \emptyset \diamond b$,
   qui est triviale car son contexte est insatisfiable.

Simplification de la troisième équation :

1. $\emptyset \diamond q = false \wedge \emptyset \diamond p = false \Rightarrow \emptyset \diamond f(d) = \emptyset \diamond b$ *et*
2. $\emptyset \diamond q = false \wedge \emptyset \diamond p = true \Rightarrow \emptyset \diamond f(a) = \emptyset \diamond b$

Résumons à présent la situation : les quatre équations non-triviales obtenues après simplification de e par R sont comme suit :

1. $\emptyset \diamond p = true \wedge \emptyset \diamond q = true \Rightarrow \emptyset \diamond f(c) = \emptyset \diamond b$,
2. $\emptyset \diamond p = false \Rightarrow \emptyset \diamond f(d) = \emptyset \diamond b$,
3. $\emptyset \diamond q = false \wedge \emptyset \diamond p = false \Rightarrow \emptyset \diamond f(d) = \emptyset \diamond b$ *et*
4. $\emptyset \diamond q = false \wedge \emptyset \diamond p = true \Rightarrow \emptyset \diamond f(a) = \emptyset \diamond b$

On dit que ces quatre équations conditionnelles sont des formes contextuellement simplifiées de $f(a) = b$ *(que l'on peut encore écrire* $TT :: f(a) = b$*).* $\diamond$

Remarque : Notons qu'avec le type de contrôle que nous avons adopté, on s'interdit également de réduire le même terme à une occurrence différente s'il s'agit de le réduire avec la même règle et la même substitution. En effet, cela aurait pour effet d'engendrer des branches triviales. Examinons ce cas à titre d'exemple :

Exemple 4.4 *Soit R constitué de la seule règle* $r : p = true \Rightarrow m \rightarrow n$ *et soit l'équation*

$$e : \Re(f(m,m)) \diamond f(m,\ m) = \Re(a) \diamond a\ ;$$

Calculons l'ensemble $\Re(f(m,m)) = \{(p = true, m, \mathcal{I}d)\}$.

Le membre gauche de e est simplifiable par r aux occurrences 1 et 2. Simplifions dans un premier temps ce terme à l'occurrence 1 ; nous obtenons les équations suivantes :

1. $\Re(p) \diamond p = true \Rightarrow \Re(f(n,m)) \diamond f(n,m) = \Re(a) \diamond a$
   où $\Re(p) = \emptyset,\ \Re(f(n,m)) = \{(p = true, m, \mathcal{I}d)\}$ *et* $\Re(a) = \emptyset$.
   Cette équation est de nouveau simplifiable et produit :
   (a) $\Re(p) \diamond p = true \Rightarrow \Re(f(n,n)) \diamond f(n,n) = \Re(a) \diamond a$
   où $\Re(p) = \emptyset,\ \Re(f(n,n)) = \emptyset$ *et* $\Re(a) = \emptyset$.
   (b) $\Re(p) \diamond p = true \wedge \Re(p) \diamond p = false \Rightarrow \Re'(f(n,m)) \diamond f(n,m) = \Re(a) \diamond a$
   qui est triviale. On peut remarquer de plus que $\Re'(f(n,m)) = \emptyset$.
2. $\Re(p) \diamond p = false \Rightarrow \Re'(f(m,m)) \diamond f(m,m) = \Re(a) \diamond a$,
   où $\Re'(f(m,m)) = \emptyset,\ \Re(p) = \emptyset$ *et* $\Re(a) = \emptyset$.

   Montrons que la prise en compte de l'occurrence de simplification est inutile. Cette équation est en effet de nouveau simplifiable mais cette fois, à l'occurrence 2. Montrons que cette simplification ne produit que des équations inutiles. La première est triviale :

   - $\Re(p) \diamond p = false \wedge \Re(p) \diamond p = true \Rightarrow \Re(f(m,n)) \diamond f(m,n) = \Re(a) \diamond a$
   - *et la seconde est redondante :*
     $\Re(p) \diamond p = false \wedge \Re(p) \diamond p = false \Rightarrow \Re"(f(m,m)) \diamond f(m,m) = \Re(a) \diamond a$

En conclusion, les deux formes contextuellement simplifiées de e sont :

1. $\Re(p) \diamond p = true \Rightarrow \Re(f(n,\ n)) \diamond f(n,\ n) = \Re(a) \diamond a$
2. $\Re(p) \diamond p = false \Rightarrow \Re(f(m,\ m)) \diamond f(m,\ m) = \Re(a) \diamond a$

$\diamond$

Illustrons encore le traitement de la branche complémentaire sur un exemple :

Exemple 4.5 *Soit R le système de réécriture constitué de la seule règle*

$$q = true \Rightarrow m \rightarrow n,$$

et soit l'équation $f(m) = a$.

$\{(q = true, m, Id)\} \diamond f(m) = \emptyset \diamond a$ est simplifiable par R de la manière suivante :

1. *$\emptyset \diamond q = true \Rightarrow \emptyset \diamond f(n) = \emptyset \diamond a$ et*
2. *$\emptyset \diamond q = false \Rightarrow \emptyset \diamond f(m) = \emptyset \diamond a$*

Puisque pour la seconde équation, l'ensemble de redex du terme $f(m)$ est vide, ceci assure la terminaison de la branche complémentaire.

3 Règles d'inférence du test de confluence

Dans cette partie, nous présentons en premier lieu les règles de simplification (ou *règles destructrices*). Les règles de base sont données ultérieurement. Ces règles simplificatrices agissent uniquement sur des équations. Elles ont pour objectif de tester la convergence des paires critiques engendrées par les règles du système et peuvent opérer sur toutes les composantes de l'équation.

3.1 Règles de simplification

Règles d'inférence de simplification :

Notation : $(\vec{c} :: s) \xrightarrow[R]{c}{}^{[\omega,\tau,\vec{q}\Rightarrow l\rightarrow r]} (\vec{c}' :: t)$ signifie que le c-terme $(\vec{c} :: s)$ se réécrit en $(\vec{c}' :: t)$ par la règle $\vec{q} \Rightarrow l \rightarrow r$ à l'occurrence ω en appliquant la substitution τ.

Quand une règle de simplification est appliquée à une équation conditionnelle, celle-ci est soit une paire critique, soit une équation issue d'une paire critique par simplification, dont on veut prouver la convergence.

SIMPLIFIER LA CONSEQUENCE D'UNE EQUATION ($SCQE$):

$$\frac{E \cup \{\vec{p} \Rightarrow g = d\}, R}{E \cup \{\vec{p} \wedge \tau\vec{q} \Rightarrow \tilde{g} = d\} \cup \{\vec{p} \wedge (\tau q_1 = \vec{q}'_1) \Rightarrow g = d\} \cup \ldots \cup \{\vec{p} \wedge (\tau q_m = \vec{q}'_m) \Rightarrow g = d\}, R}$$

si $(\vec{p} :: g) \xrightarrow[R]{c}{}^{[\omega,\tau,\vec{q}\Rightarrow l\rightarrow r]} (\vec{p} \wedge \tau\vec{q} :: \tilde{g})$

avec $\vec{q} = (\wedge_{i=1}^{m} q_i = q'_i)$, $q'_i \in \{true, false\}$.

Le cas symétrique qui correspond à la simplification de d est couvert par une règle symétrique.

SIMPLIFIER LA CONDITION D'UNE EQUATION ($SCDE$):

$$\frac{E \cup \{\vec{p} \wedge (a = a') \Rightarrow g = d\}, R}{E \cup \{\vec{p} \wedge (\tilde{a} = a') \wedge \tau\vec{q} \Rightarrow g = d\} \cup \{\vec{p} \wedge (a = a') \wedge (\tau q_1 = \vec{q}'_1) \Rightarrow g = d\} \cup \ldots \cup \{\vec{p} \wedge (a = a') \wedge (\tau q_m = \vec{q}'_m) \Rightarrow g = d\}, R}$$

si $(TT :: a) \xrightarrow[R]{c}{}^{[\omega,\tau,\vec{q}\Rightarrow l\rightarrow r]} (\tau\vec{q} :: \tilde{a})$

avec $\vec{q} = (\wedge_{i=1}^{m} q_i = q'_i)$, $q'_i \in \{true, false\}$.

Dans ces règles d'inférence, la première équation est obtenue par réduction effective, et les m dernières sont le résultat du calcul de la branche complémentaire.

3.2 Règles d'inférence de base

Nous donnons ci-après les règles d'inférence de base nécessaires dans toute procédure de complétion.

ORIENTER UNE REGLE (OR):

$$\frac{E \cup \{\wedge_{j=1}^{n} p_j = p'_j \Rightarrow g = d\},\ R}{E,\ R \cup \{\wedge_{j=1}^{n} p_j = p'_j \Rightarrow g \rightarrow d\}}$$

si $\forall j \in [1 \ldots n],\ g \succ p_j$ et $g > d$.

Notons que cette règle d'inférence engendre également celle de l'orientation d'une règle inconditionnelle.

AJOUTER UNE CONSEQUENCE EQUATIONNELLE (ACE):

$$\frac{E,\ R \cup \{r_1,\ r_2\}}{E \cup \{(\wedge_{j=1}^{n} \sigma p_j = p'_j)\ \wedge\ (\wedge_{j=1}^{k} \sigma q_j = q'_j) \Rightarrow M = N\},\ R \cup \{r_1,\ r_2\}}$$

$$\text{où} \begin{cases} r_1 : \wedge_{j=1}^{n}\ p_j = p'_j \Rightarrow g \rightarrow d \\ r_2 : \wedge_{j=1}^{k}\ q_j = q'_j \Rightarrow l \rightarrow r \end{cases}$$

$$\text{si} \begin{cases} \text{il existe un unificateur minimal } \sigma \text{ et} \\ \text{une occurrence } u \text{ de } g \text{ telle que } \sigma g_{/u} = \sigma l \\ M = \sigma(g[u \leftarrow r]),\ N = \sigma d \end{cases}$$

Cette règle d'inférence telle qu'elle est formulée ici, est restreinte au calcul des paires critiques car nous n'avons pas besoin de l'étendre davantage. La façon dont sont calculées les paires critiques contextuelles dépend fortement du contrôle instauré sur les règles d'inférence.

SUPPRIMER UNE EQUATION TRIVIALE (SET) :

$$\frac{E \cup \{\vec{p} \Rightarrow g = d\},\ R}{E,\ R}$$

si pour toute substitution close σ, il existe une preuve de $\sigma\vec{p} =_R FF$.

Cette règle d'inférence laisse supposer en réalité l'utilisation de formules du premier ordre pour effectuer des preuves inductives, parce que sa condition d'applicabilité est indécidable. Pour appliquer cette règle, il s'agit d'obtenir des conditions suffisantes et de vérifier que le traitement est cohérent. Quelques formulations particulières de cette règle d'inférence relativement aisées à mettre en œuvre sont les suivantes :

Les deux premières portent sur une particularité de la conséquence, à savoir lorsque les deux termes qui la composent sont syntaxiquement égaux, ou lorsqu'elle est contenue dans la précondition :

$$\frac{E \cup \{\vec{p} \Rightarrow g = g\},\ R}{E,\ R}$$

$$\frac{E \cup \{\vec{p} \wedge a = a' \Rightarrow a = a'\},\ R}{E,\ R}$$

Cette dernière règle est spécifique aux équations conditionnelles dont la conséquence est formée de termes booléens et telle que $a' \in \{true,\ false\}$.

Les deux règles suivantes traduisent des cas particuliers d'insatisfiabilité du contexte :

$$\frac{E \cup \{\vec{p} \wedge true = false \Rightarrow g = d\},\ R}{E,\ R}$$

$$\frac{E \cup \{\vec{p} \wedge a = true \wedge a = false \Rightarrow g = d\},\ R}{E,\ R}$$

Chacune de ces règles préserve la congruence $\equiv_{E \cup R}$ sur les termes clos.

4 Confluence sur les termes clos

4.1 Rappels et définitions

Cette première partie a pour objectif de rappeler de façon informelle les principaux concepts utilisés dans la preuve de confluence.

La confluence sur les termes clos est définie à partir de la réduction récursive, à savoir qu'un terme clos t de $T(F)$ se réduit en un autre terme clos t' ssi

$$t = t[\sigma g],\ t' = t[\sigma d],\ \sigma\vec{p} \rightarrow_R^* TT$$

où $\vec{p} \Rightarrow g \rightarrow d$ est une règle de R et σ une substitution appropriée.

Une CEB close $\vec{c}$ de la forme $\wedge_{i=1}^{n}\ t_i = t_i'$ se réduit à TT ssi $\forall i \in [1 \ldots n],\ t_i \rightarrow_R^* t_i'$. Similairement, $\vec{c} \rightarrow_R^* FF$ ssi il existe un indice k dans $[1 \ldots n]$ tel que $t_k \rightarrow_R^* \overline{t_k'}$, où $\overline{true} = false$ et $\overline{false} = true$.

R est dit confluent sur les termes clos (resp localement confluent sur les termes clos) ssi pour tous termes clos $t,\ t_1,\ t_2$ tels que $t_1 \leftarrow_R^* t \rightarrow_R^* t_2$ (resp $t_1 \leftarrow_R t \rightarrow_R t_2$), il existe un terme clos t' tel qu'on puisse refermer le diagramme de réduction :

$$t_1 \rightarrow_R^* t' \leftarrow_R^* t_2$$

L'utilisation de systèmes de réécriture décroissants assure la terminaison de la réduction récursive. Cette terminaison est essentielle car elle garantit que toute dérivation infinie de la forme :

$$t_1 \rightarrow_R t_2 \rightarrow_R \ldots \rightarrow_R t_n \rightarrow_R \ldots$$

ne peut exister. On peut trouver ce résultat de nœthérianité dans [19].

Une spécification conditionnelle E est dite consistante par rapport aux booléens ssi $true$ et $false$ ne sont pas équivalentes dans la théorie correspondante. Autrement dit, l'égalité $true = false$ n'est pas engendrée par remplacements successifs en utilisant les axiomes de E : $\neg(true =_{E(R)} false)$.

R est convertible par rapport aux booléens ssi tout terme clos booléen t vérifie :

$$t \rightarrow_R^* true \text{ ou } t \rightarrow_R^* false$$

Rappelons que la convertibilité n'est autre que la propriété de complétude suffisante opérationnelle.

Ceci nous amène à établir le lemme des paires critiques. Le concept de paire critique contextuelle a été précédemment défini dans le chapitre 3. Rappelons simplement qu'une paire critique contextuelle $\sigma\vec{p} \wedge \sigma\vec{q} \Rightarrow \sigma(g[u \leftarrow r]) = \sigma d$ entre $\vec{p} \Rightarrow g \rightarrow d$ et $\vec{q} \Rightarrow l \rightarrow r$ est le résultat de la superposition de ces deux règles s'il existe une occurrence u de non-variable dans g telles $g_{/u}$ et l soient unifiables avec σ leur unificateur le plus général.

Définition 4.1 Forme contextuellement simplifiée

L'équation $\vec{C'} \Rightarrow M' = N'$ est une forme contextuellement simplifiée de $\vec{C} \Rightarrow M = N$ si $\vec{C'} \Rightarrow M' = N'$ est dérivée à partir de $\vec{C} \Rightarrow M = N$ par remplacements successifs en appliquant les règles d'inférence de simplification $SCQE$ ou $SCDE$.

Exemple 4.6 *Soit R le système de réécriture conditionnelle suivant :*

$$R = \begin{cases} q = true \Rightarrow m \rightarrow n, \\ q \rightarrow q' \end{cases}$$

Les équations conditionnelles suivantes sont toutes deux des formes contextuellement simplifiées de $f(m) = b$:

- $q' = true \Rightarrow f(n) = b$ obtenue en appliquant successivement les règles $SCQE$ au terme $f(m)$, puis $SCDE$ au terme q.
- $q = false \Rightarrow f(m) = b$ qui est le résultat de la branche complémentaire de l'application de $SCQE$.

Définition 4.2 Convergence sur les termes clos d'une paire critique

Soit $\vec{C} \Rightarrow M = N$ une paire critique contextuelle ; $\vec{C} \Rightarrow M = N$ converge sur les termes clos ssi pour toute substitution close σ dans $Subst(F)$ telle que $\sigma\vec{C} \rightarrow_R^* TT$ il existe un terme clos t dans $T(F)$ tel que

$$\sigma M \rightarrow_R^* t \text{ et } \sigma N \rightarrow_R^* t.$$

Soit $INF = \{SCQE,\ SCDE\}$ l'ensemble des règles d'inférence de simplification ; on définit un ensemble saturé de formes contextuellement simplifiées comme suit :

Définition 4.3 Ensemble saturé de formes contextuellement simplifiées
Soit $\mathcal{F}$ un ensemble de formes contextuellement simplifiées (EFCS) de l'équation $\vec{C} \Rightarrow M = N$;

si $\mathcal{F} = \{\vec{C} \Rightarrow M = N\}$, alors $\mathcal{F}$ est saturé.

ou soit $\mathcal{F} = \{\vec{C_i} \Rightarrow M_i = N_i\}_{i \in I}$ et soit $\vec{C}_{i_0} \Rightarrow M_{i_0} = N_{i_0}$ une équation de $\mathcal{F}$; si $\mathcal{F}$ est saturé,

$$\mathcal{F} \ominus \{\vec{C}_{i_0} \Rightarrow M_{i_0} = N_{i_0}\} \oplus \{\vec{C'_j} \Rightarrow M'_j = N'_j\}_{j \in J}$$

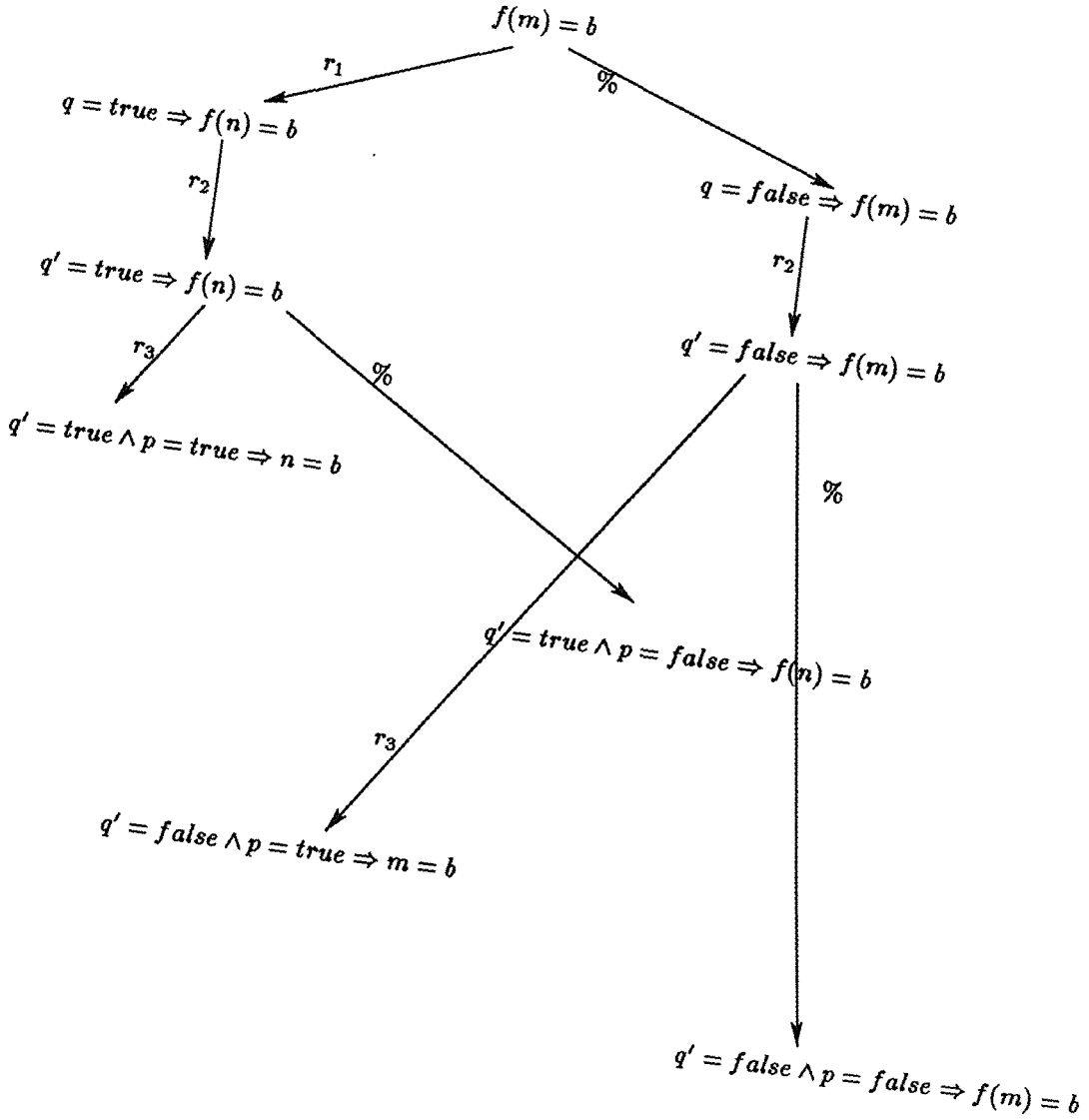
est aussi saturé.
$\{\vec{C'_j} \Rightarrow M'_j = N'_j\}_{j \in J}$ est l'ensemble de toutes les formes contextuellement simplifiées obtenues à partir de l'équation $\vec{C}_{i_0} \Rightarrow M_{i_0} = N_{i_0}$ en lui appliquant soit la règle $SCQE$ soit la règle $SCDE$.

La définition 4.3 propose en particulier une méthode de construction d'un ensemble saturé de formes contextuellement simplifiées. Soit $\vec{C} \Rightarrow M = N$ l'équation initiale ; de façon triviale, l'ensemble $\{\vec{C} \Rightarrow M = N\}$ est saturé. Si le nœud $\vec{C} \Rightarrow M = N$ est remplacé par l'ensemble des équations conditionnelles obtenues par simplification de $\vec{C} \Rightarrow M = N$ en utilisant $SCQE$ ou $SCDE$, le nouveau $EFCS$ est également saturé. D'une façon plus générale, si un $EFCS$ est saturé à l'étape i et si un élément quelconque de l'$EFCS$ est remplacé par l'ensemble de toutes les équations obtenues après une simplification, l'$EFCS$ à l'étape $i+1$ est lui aussi saturé.

Exemple 4.7 *Soit R le système de réécriture conditionnelle suivant :*

$$R = \begin{cases} r_1 : & q = true \Rightarrow m \rightarrow n, \\ r_2 : & q \rightarrow q' \\ r_3 : & p = true \Rightarrow f(x) \rightarrow x \end{cases}$$

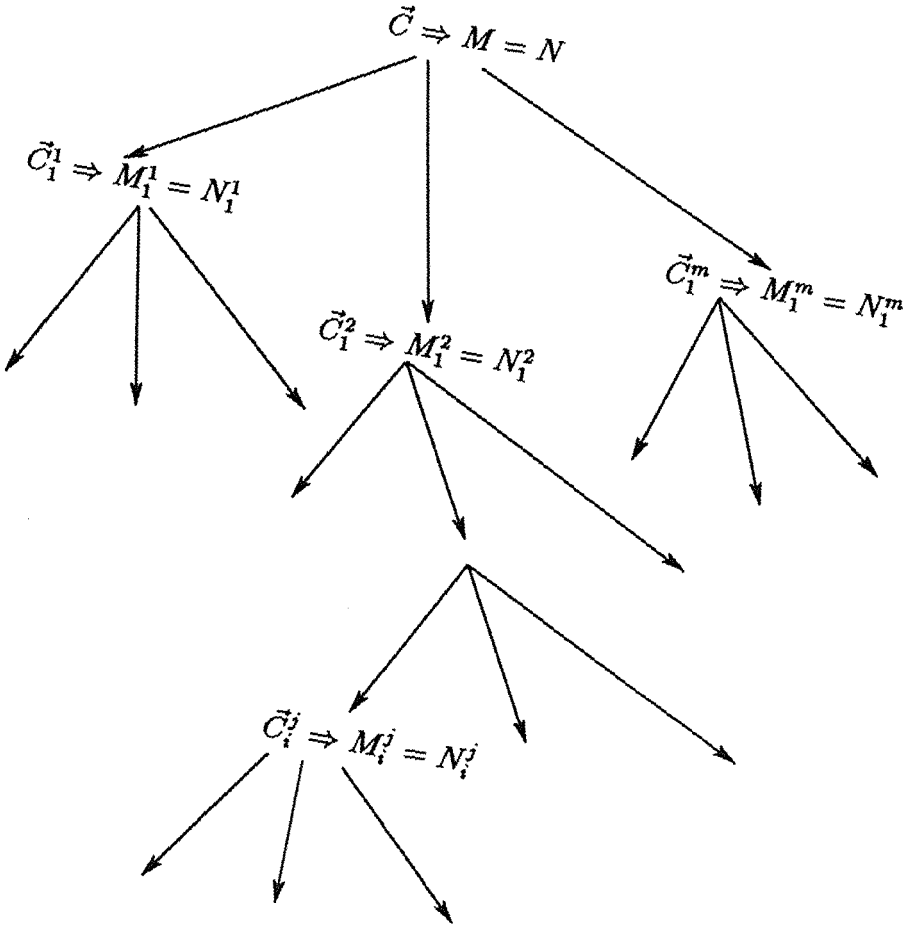
L'arbre de dérivation de l'équation $f(m) = b$ est comme suit :


$f(m) = b$
r_1
%
$q = true \Rightarrow f(n) = b$
$q = false \Rightarrow f(m) = b$
r_2
r_2
$q' = true \Rightarrow f(n) = b$
$q' = false \Rightarrow f(m) = b$
r_3
%
$q' = true \wedge p = true \Rightarrow n = b$
%
$q' = true \wedge p = false \Rightarrow f(n) = b$
r_3
$q' = false \wedge p = true \Rightarrow m = b$
$q' = false \wedge p = false \Rightarrow f(m) = b$

Les ensembles suivants sont tous des ensembles de formes contextuellement simplifiées saturés :

- $\mathcal{F}_1 = \{f(m) = b\}$,
- $\mathcal{F}_2 = \{q' = true \wedge p = true \Rightarrow n = b,\ q' = true \wedge p = false \Rightarrow f(n) = b,\ q = false \Rightarrow f(m) = b\}$
- $\mathcal{F}_3 = \{q' = true \Rightarrow f(n) = b,\ q' = false \Rightarrow f(m) = b\}$
- $\mathcal{F}_4 = \{q' = true \wedge p = true \Rightarrow n = b,\ q' = true \wedge p = false \Rightarrow f(n) = b,\ q' = false \wedge p = true \Rightarrow m = b,\ q' = false \wedge p = false \Rightarrow f(m) = b\}$

A partir d'une paire critique contextuelle $\vec{C} \Rightarrow M = N$, on construit un arbre par simplifications successives en appliquant les règles d'inférence $SCQE$ et $SCDE$. Chaque nœud est étiqueté par une équation conditionnelle. Lorsqu'on applique une règle d'inférence de simplification, nous l'appliquons à un seul nœud de l'arbre et seul ce nœud est modifié. Cette simplification engendre des formes contextuellement simplifiées du nœud en question, qui forment pour ce nœud, un ensemble saturé. Pour chacune de ces formes, on teste la convergence. Si chaque forme vérifie un des quatre critères de convergence, alors le père direct est convergent. Dans le cas contraire, on continue à réduire les formes contextuelles si possible. Ainsi, les branches de l'arbre peuvent être de longueur inégale. Pour certains nœuds, il est possible de prouver la convergence assez tôt, et pour d'autres, un certain nombre de simplifications est nécessaire.


$\vec{C} \Rightarrow M = N$
$\vec{C}_1^1 \Rightarrow M_1^1 = N_1^1$
$\vec{C}_1^2 \Rightarrow M_1^2 = N_1^2$
$\vec{C}_1^m \Rightarrow M_1^m = N_1^m$
$\vec{C}_i^j \Rightarrow M_i^j = N_i^j$

La définition 4.3 propose en particulier une méthode de construction d'un ensemble de formes contextuellement simplifiées saturé. Soit $\vec{C} \Rightarrow M = N$ l'équation initiale ; de façon triviale, l'ensemble $\{\vec{C} \Rightarrow M = N\}$ est saturé. Si le nœud $\vec{C} \Rightarrow M = N$ est remplacé par l'ensemble des équations conditionnelles obtenues par simplification de $\vec{C} \Rightarrow M = N$ en utilisant $SCQE$ ou $SCDE$, le nouveau $EFCS$ est également saturé. D'une façon plus générale, si un $EFCS$ est saturé à l'étape i et si un élément quelconque de l'$EFCS$ est remplacé par l'ensemble de toutes les équations obtenues après une simplification, l'$EFCS$ à l'étape $i+1$ est lui aussi saturé. Plus formellement,

4.2 Quelques résultats nécessaires

Lemme 4.1 *Soient t un terme clos de $T(F)$, σ une substitution close et $\vec{C} \Rightarrow M = N$ une équation conditionnelle tels que $t \succ \sigma M$, $t \succ \sigma N$, et $t \not\succcurlyeq \sigma\vec{C}$;*

si $\vec{C}_i \Rightarrow M_i = N_i$ est dérivé à partir de $\vec{C} \Rightarrow M = N$ par 0 ou plusieurs applications des règles d'inférence de INF, alors

$$t \not\succcurlyeq \sigma\vec{C}_i,\ t \succ \sigma M_i \text{ et } t \succ \sigma N_i$$

Preuve : la preuve est effectuée par récurrence sur le nombre n d'applications des règles d'inférence :

a) $n = 0$: ce cas est trivial puisque par hypothèse, $t \succ \sigma M$, σN et $t \not\succcurlyeq \sigma\vec{C}$.

b) $n > 0$:
hypothèse de récurrence : soit $\vec{C}_n \Rightarrow M_n = N_n$ dérivée à partir de $\vec{C} \Rightarrow M = N$ en appliquant n fois les règles de INF et telle que $t \not\succcurlyeq \sigma\vec{C}_n$ et $t \succ \sigma M_n$, σN_n ; considérons la dérivation

$$\frac{\vec{C}_n \Rightarrow M_n = N_n}{\vec{C}_{n+1} \Rightarrow M_{n+1} = N_{n+1}}$$

et montrons que $t \not\succcurlyeq \sigma\vec{C}_{n+1}$ et $t \succ \sigma M_{n+1}$, σN_{n+1}.

considérons les cas suivants de simplification de $\vec{C}_n \Rightarrow M_n = N_n$:

1. **réécriture effective de la partie terme :** supposons que la simplification soit effectuée dans le terme M_n (le cas où N_n est simplifié est similaire à traiter)

$$\frac{\vec{C}_n \Rightarrow M_n[\tau l] = N_n}{\vec{C}_n \wedge \tau\vec{q} \Rightarrow M_{n+1} = N_n}$$

où $M_{n+1} = M_n[\tau r]$.
Montrons que $t \succ \sigma M_{n+1}$, σN_n et $t \not\succcurlyeq \sigma(\vec{C}_n \wedge \tau\vec{q})$.

Puisque R est décroissant, $\tau l \succ \tau r$ donc, $M_n[\tau l] \succ M_n[\tau r]$, par F-compatibilité de $\succ$, donc $\sigma(M_n[\tau l]) \succ \sigma(M_n[\tau r])$, par stabilité par instanciation de $\succ$. Par conséquent, $t \succ \sigma M_n \succ \sigma M_{n+1}$.

Par hypothèse de récurrence, $t \succ \sigma N_n$.

D'autre part, $M_n[\tau l] \succeq \tau l$ (l'ordre de sous-terme strict est contenu dans $\succ$) $\not\succcurlyeq \tau\vec{q}$ puisque R est décroissant. Par conséquent, $t \succ \sigma M_n \succeq \sigma\tau l \not\succcurlyeq \sigma\tau\vec{q}$, par stabilité par instanciation de $\succ$ et $\not\succcurlyeq$. De plus, $t \not\succcurlyeq \sigma\vec{C}_n$ par hypothèse donc, $t \not\succcurlyeq \sigma(\vec{C}_n \wedge \tau\vec{q})$ par définition de l'ordre $\not\succcurlyeq$.

2. **branche complémentaire de la réécriture dans la partie terme :**

$$\frac{\vec{C}_n \Rightarrow M_n = N_n}{\vec{C}_n \wedge \tau t_1 = \bar{t}'_1 \Rightarrow M_n = N_n, \ldots, \vec{C}_n \wedge \tau t_m = \bar{t}'_m \Rightarrow M_n = N_n}$$

en supposant que $\vec{q}$ soit de la forme $\wedge_{j=1}^{m} t_j = t'_j$, $t'_j \in \{true,\ false\}$, $\forall j \in [1 \ldots m]$.

Par hypothèse, $t \succ \sigma M_n,\ \sigma N_n$.
Montrons que pour tout j dans $[1 \ldots m]$, $t \succ\!\!\!\succ \sigma(\vec{C}_n \wedge (\tau t_j = \bar{t}'_j))$.

$M_n[\tau l] \succeq \tau l$ ($\succ$ contient l'ordre de sous-terme strict) $\succ \tau t_j$, $\forall j$ (puisque τl $\succ\!\!\!\succ \tau \vec{q}$). Donc, $t \succ \sigma M_n \succ \sigma\tau t_j$, $\forall j$, puisque $\succ$ est stable par instanciation. De plus, par hypothèse de récurrence, $t \succ\!\!\!\succ \sigma\vec{C}_n$. Donc, $t \succ\!\!\!\succ \sigma(\vec{C}_n \wedge (\tau t_j = \bar{t}'_j))$, $\forall j$ par définition de $\succ\!\!\!\succ$.

3. **réécriture effective de la partie contexte :**
Soit $\vec{C}_n$ de la forme $\vec{C}\,'' \wedge (a[\tau l] = a')$

$$\frac{\vec{C}\,'' \wedge (a[\tau l] = a') \Rightarrow M_n = N_n}{\vec{C}\,'' \wedge (a[\tau r] = a') \wedge \tau\vec{q} \Rightarrow M_n = N_n}$$

Par hypothèse de récurrence, $t \succ \sigma M_n,\ \sigma N_n$.
Montrons que $t \succ\!\!\!\succ \sigma(\vec{C}\,'' \wedge (a[\tau r] = a') \wedge \tau\vec{q})$.

Par hypothèse de récurrence, $t \succ\!\!\!\succ \sigma(\vec{C}\,'' \wedge (a[\tau l] = a'))$. Puisque R est décroissant, $\tau l \succ \tau r$. Donc, $a[\tau l] \succ a[\tau r]$ par F-compatibilité de $\succ$ et $t \succ \sigma a[\tau l] \succ \sigma a[\tau r]$ car $\succ$ est stable par instanciation. D'autre part, $a[\tau l] \succeq \tau l$ (l'ordre de sous-terme strict est contenu dans $\succ$) $\succ\!\!\!\succ \tau\vec{q}$. Par conséquent, $a \succ\!\!\!\succ \tau\vec{q}$ et donc $\sigma a \succ\!\!\!\succ \sigma\tau\vec{q}$. Donc, $t \succ\!\!\!\succ \sigma(\vec{C}\,'' \wedge (a[\tau r] = a') \wedge \tau\vec{q})$ par définition de l'ordre $\succ\!\!\!\succ$.

4. **branche complémentaire de la réécriture dans la partie contexte :**

$$\frac{\vec{C}\,'' \wedge (a[\tau l] = a') \Rightarrow M_n = N_n}{\vec{C}\,'' \wedge (a[\tau l] = a') \wedge (\tau t_1 = \bar{t}'_1) \Rightarrow M_n = N_n, \ldots, \vec{C}\,'' \wedge (a[\tau l] = a') \wedge (\tau t_m = \bar{t}'_m) \Rightarrow M_n = N_n}$$

en supposant que $\vec{q}$ soit de la forme $\wedge_{j=1}^{m} t_j = t'_j$, $t'_j \in \{true,\ false\}$, $\forall j \in [1 \ldots m]$.

Par hypothèse de récurrence, $t \succ \sigma M_n,\ \sigma N_n$ (les deux premiers points sont donc prouvés). D'autre part, $t \succ\!\!\!\succ \sigma(\vec{C}\,'' \wedge (a[\tau l] = a'))$.

Puisque R est décroissant, $\forall j \in [1 \ldots m]$, $\tau l \succ \tau t_j$ puisque $\tau l \succ\!\!\!\succ \tau\vec{q}$. D'autre part, $a[\tau l] \succ \tau l$ (l'ordre de sous-terme strict est contenu dans $\succ$). Donc, $\forall j$, $a[\tau l] \succ \tau t_j$. Donc, $t \succ \sigma a \succ \sigma\tau t_j$, $\forall j$.
Par conséquent, $t \succ\!\!\!\succ \sigma(\vec{C}\,'' \wedge (a[\tau l] = a') \wedge (\tau t_j = \bar{t}'_j))$ par définition de l'ordre $\succ\!\!\!\succ$.

Tous les cas de simplification ayant été considérés, ceci achève la preuve du lemme. □

Lemme 4.2 *Soit R un système de réécriture conditionnelle décroissant tel que true et false soient R-irréductibles et tel que $\rightarrow_R$ soit convertible par rapport à l'ensemble $\{true,\ false\}$;*
Supposons que, pour toute paire critique contextuelle $\vec{C}_0 \Rightarrow M_0 = N_0$, il existe un

ensemble saturé de formes contextuellement simplifiées $\{\vec{C}_i \Rightarrow M_i = N_i\}_{i\in I}$ *et soit* σ *une substitution close telle que* $\sigma\vec{C}_0 \rightarrow_R^* TT$;
Soit t *un terme clos de* $T(F)$ *tel que* $t \succ\!\!\succ \sigma\vec{C}_0$, $t \succ \sigma M_0$, σN_0 *et tel que* R *soit confluent en tout terme clos* t' *tel que* $t \succ t'$;

il existe alors un indice k *dans* I *tel que* $\sigma\vec{C}_k \rightarrow_R^* TT$, $\sigma M_0 \rightarrow_R^* \sigma M_k$ *et* $\sigma N_0 \rightarrow_R^* \sigma N_k$.

Preuve : la preuve est effectuée par récurrence sur la construction de l'EFCS saturé de $\vec{C}_0 \Rightarrow M_0 = N_0$.

Soit $\mathcal{F} = \{\vec{C}_i \Rightarrow M_i = N_i\}_{i\in I}$ un EFCS saturé de $\vec{C}_0 \Rightarrow M_0 = N_0$ à l'étape n. Supposons que le lemme soit valide pour l'étape n de construction. Par conséquent, il existe un indice $k \in I$ tel que $\sigma\vec{C}_k \rightarrow_R^* TT$, $\sigma M_0 \rightarrow_R^* \sigma M_k$ et $\sigma N_0 \rightarrow_R^* \sigma N_k$.

Soit à présent $\mathcal{F}' = \{\vec{C'_j} \Rightarrow M'_j = N'_j\}_{j\in J}$ l'EFCS à l'étape $n+1$, obtenu à partir de $\mathcal{F}$ en simplifiant l'équation $\vec{C}_{j_0} \Rightarrow M_{j_0} = N_{j_0}$ de $\mathcal{F}$.

Notons en particulier que $\mathcal{F}'$ est aussi saturé puisque $\mathcal{F}$ l'est. Ceci est justifié par la construction de $\mathcal{F}'$ et par la définition 4.3.

Montrons que le lemme est valide à l'étape $n+1$. Deux cas peuvent se présenter :

a) soit l'indice j_0 est différent de k. Par conséquent, puisque l'équation $\vec{C}_k \Rightarrow M_k = N_k$ est aussi dans $\mathcal{F}'$, $k \in J$ et le lemme est vérifié à l'étape $n+1$.

b) soit l'indice j_0 est exactement l'indice k. Considérons alors les deux types suivants de dérivation selon que l'on a appliqué à l'équation $\vec{C}_k \Rightarrow M_k = N_k$ la règle $SCQE$ ou la règle $SCDE$:

Il s'agit de chercher un indice j dans J, i.e. dans le nouveau $EFCS$, $\mathcal{F}'$ tel que $\sigma\vec{C'_j} \rightarrow_R^* TT$, $\sigma M_0 \rightarrow_R^* \sigma M'_j$ et $\sigma N_0 \rightarrow_R^* \sigma N'_j$.

Réécriture dans la partie terme (ou application de $SCQE$) :
supposons que $SCQE$ soit appliquée à M_k (le cas où $SCQE$ est appliquée à N_k est similaire). Cette application se fait avec la règle $\vec{q} \Rightarrow l \rightarrow r$ et la substitution τ et soit $\vec{q}$ de la forme $\wedge_{i=1}^m t_i = t'_i$;

$$\frac{\vec{C}_k \Rightarrow M_k[\tau l] = N_k}{\vec{C}_k \wedge \tau\vec{q} \Rightarrow \tilde{M}_k = N_k, \vec{C}_k \wedge (\tau t_1 = \vec{t}'_1) \Rightarrow M_k[\tau l] = N_k, \ldots, \vec{C}_k \wedge (\tau t_m = \vec{t}'_m) \Rightarrow M_k[\tau l] = N_k}$$

où $\tilde{M}_k = M_k[\tau r]$.

Par hypothèse, on a $\sigma\vec{C}_k \rightarrow_R^* TT$.
Puisque R est convertible par rapport à $\{true,\ false\}$, pour tout i, $\sigma\tau t_i \rightarrow_R^* true$ ou $\sigma\tau t_i \rightarrow_R^* false$. Par conséquent, pour tout i, $\sigma\tau t_i \rightarrow_R^* t'_i$ ou $\sigma\tau t_i \rightarrow_R^* \vec{t}'_i$ puisque $\forall i$, $t'_i \in \{true,\ false\}$.

- Si pour tout i, $\sigma\tau t_i \rightarrow_R^* t'_i$, alors $\sigma\tau\vec{q} \rightarrow_R^* TT$. Donc, $\sigma\vec{C}_k \wedge \sigma\tau\vec{q} \rightarrow_R^* TT$ et $\sigma M_k \rightarrow_R^* \sigma\tilde{M}_k$. Par conséquent, $\sigma M_0 \rightarrow_R^* \sigma M_k$ (par hypothèse) $\rightarrow_R^* \sigma\tilde{M}_k$.
  On pose $\vec{C'_j} = \vec{C}_k \wedge \tau\vec{q}$, $M'_j = \tilde{M}_k$ et $N'_j = N_k$.
- S'il existe un indice i dans $[1 \ldots m]$ tel que $\sigma\tau t_i \rightarrow_R^* \vec{t}'_i$, alors $\sigma\tau t_i = \vec{t}'_i \rightarrow_R^* TT$. Par conséquent, $\sigma\vec{C}_k \wedge (\sigma\tau t_i = \vec{t}'_i) \rightarrow_R^* TT$. D'autre part, par hypothèse, $\sigma M_0 \rightarrow_R^* \sigma M_k$.
  On pose $\vec{C'_j} = \vec{C}_k \wedge (\tau t_i = \vec{t}'_i)$, $M'_j = M_k$ et $N'_j = N_k$.

Réécriture dans la partie contexte (ou application de $SCDE$):
soit $\vec{C}_k$ de la forme $\overrightarrow{C'} \wedge (a = a')$:

$$\frac{\overrightarrow{C'} \wedge (a[\tau l] = a') \Rightarrow M_k = N_k}{\overrightarrow{C'} \wedge (a[\tau r]=a') \wedge \tau\vec{q} \Rightarrow M_k=N_k, \overrightarrow{C'} \wedge (a[\tau l]=a') \wedge (\tau t_1=\vec{t}'_1) \Rightarrow M_k=N_k, \ldots, \overrightarrow{C'} \wedge (a[\tau l]=a') \wedge (\tau t_m=\vec{t}'_m) \Rightarrow M_k=N_k}$$

Par hypothèse, on a $\sigma\vec{C}_k \rightarrow_R^* TT$. Donc, $\sigma\ \overrightarrow{C'} \rightarrow_R^* TT$ et $\sigma a \rightarrow_R^* a'$.

Puisque R est convertible par rapport à $\{true,\ false\}$, $\sigma\tau t_i \rightarrow_R^* t'_i$ ou $\sigma\tau t_i \rightarrow_R^* \vec{t}'_i$ (puisque $\forall i,\ t'_i \in \{true,\ false\}$).

- Si pour tout i, $\sigma\tau t_i \rightarrow_R^* t'_i$, alors $\sigma\tau\vec{q} \rightarrow_R^* TT$ et $\sigma(a[\tau l]) \rightarrow_R^* \sigma(a[\tau r])$. D'autre part, $\sigma(a[\tau l]) \rightarrow_R^* a'$. Par le lemme 4.1, $t \gg\!\succ \sigma\vec{C}_k$. Par conséquent, $t \succ \sigma a$ puisque σa est contenu dans $\sigma\vec{C}_k$. On peut donc appliquer l'hypothèse de confluence de R en σa. Par conséquent, $\sigma(a[\tau r]) \rightarrow_R^* a'$. Donc, $\sigma\ \overrightarrow{C'} \wedge \sigma\tau\vec{q} \wedge (\sigma(a[\tau r]) = a') \rightarrow_R^* TT$.
  Par ailleurs, par hypothèse, $\sigma M_0 \rightarrow_R^* \sigma M_k$, $\sigma N_0 \rightarrow_R^* \sigma N_k$.
  On pose $\overrightarrow{C'_j}=\overrightarrow{C'} \wedge \tau\vec{q} \wedge (a[\tau r] = a')$, $M'_j = M_k$ et $N'_j = N_k$.

- S'il existe un indice i dans $[1 \ldots m]$ tel que $\sigma\tau t_i \rightarrow_R^* \vec{t}'_i$, (donc $\sigma\tau\vec{q} \rightarrow_R^* FF$) alors $(\sigma\tau t_i = \vec{t}'_i) \rightarrow_R^* TT$. Par conséquent, $\sigma\ \overrightarrow{C'} \wedge (\sigma a = a') \wedge (\sigma\tau t_i = \vec{t}'_i) \rightarrow_R^* TT$. D'autre part, par hypothèse, $\sigma M_0 \rightarrow_R^* \sigma M_k$ et $\sigma N_0 \rightarrow_R^* \sigma N_k$.
  On pose $\overrightarrow{C'_j}=\overrightarrow{C'} \wedge (a = a') \wedge (\tau t_i = \vec{t}'_i)$, $M'_j = M_k$ et $N'_j = N_k$.

□

4.3 Théorème principal de confluence sur les termes clos

Théorème 4.1 *Soit R un système de réécriture conditionnelle décroissant tel que true et false soient R-irréductibles et tel que $\rightarrow_R$ soit convertible par rapport à l'ensemble $\{true,\ false\}$;*
Supposons que, pour toute paire critique contextuelle $\vec{C} \Rightarrow M = N$, il existe un ensemble saturé de formes contextuellement simplifiées $\{\vec{C}_i \Rightarrow M_i = N_i\}_{i \in I}$ tel que, pour tout $i \in I$, une des conditions suivantes soit vérifiée:

- *$true = false$ appartient à $\vec{C}_i$ ou $false = true$ appartient à $\vec{C}_i$*
- *$p = true \wedge p = false$ appartient à $\vec{C}_i$*
- *$M_i \equiv N_i$, (où le symbole $\equiv$ représente l'égalité syntaxique)*
- *$M_i = N_i$ appartient à $\vec{C}_i$ ou $N_i = M_i$ appartient à $\vec{C}_i$*

Alors, R est confluent sur les termes clos.
De plus, $E(R)$ est aussi consistante par rapport aux booléens.

Preuve: La preuve de confluence est effectuée par récurrence sur l'ordre $\succ$. Soit t un terme clos de $T(F)$;
hypothèse de récurrence: pour tout terme clos t' tel que $t \succ t'$, R est confluent en t'.

Dans un premier temps, on montre que R est localement confluent en t ; pour cela, supposons qu'il existe deux termes clos t_1, t_2 de $T(F)$ tels que $t \rightarrow_R t_1$ et $t \rightarrow_R t_2$.

Supposons que $t \rightarrow_R t_1$ par la règle $\vec{p}_1 \Rightarrow g_1 \rightarrow d_1$ à l'occurrence u_1 avec la substitution σ_1. Par conséquent,

$$t_{/u_1} = \sigma g_1,\ t_1 = t[u_1 \leftarrow \sigma d_1],\ \sigma \vec{p}_1 \rightarrow_R^* TT$$

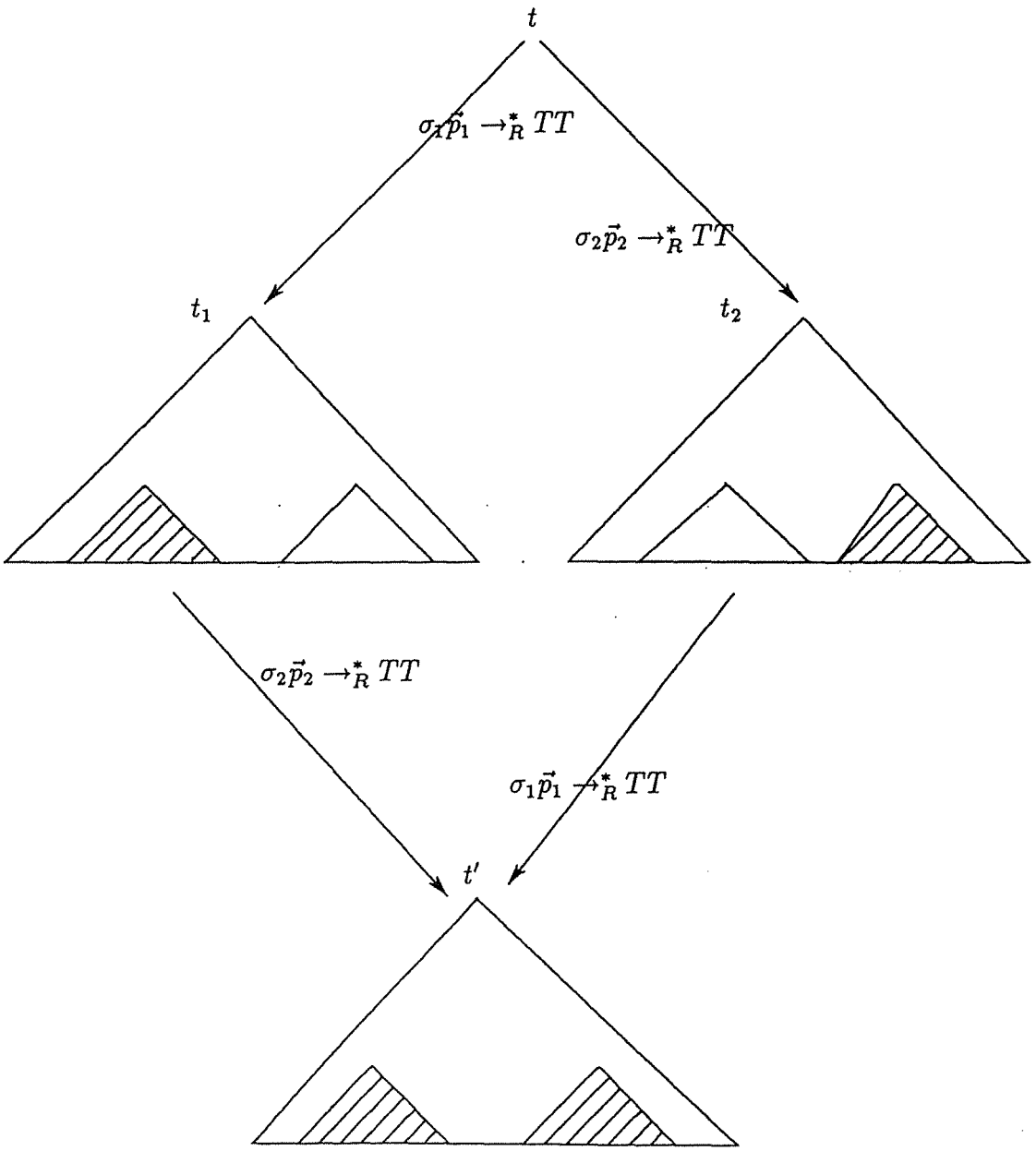
Similairement, supposons que $t \rightarrow_R t_2$ par la règle $\vec{p}_2 \Rightarrow g_2 \rightarrow d_2$. Alors,

$$t_{/u_2} = \sigma g_2,\ t_2 = t[u \leftarrow \sigma d_2],\ \sigma \vec{p}_2 \rightarrow_R^* TT$$

Différents cas doivent être considérés :

- cas des occurrences disjointes :
  Le diagramme est clos de la manière suivante :

$$t_1 \rightarrow_R t' = t_1[u_2 \leftarrow \sigma_2 d_2] = t_2[u_1 \leftarrow \sigma_1 d_1] \leftarrow_R t_2$$



- cas des occurrences préfixes:

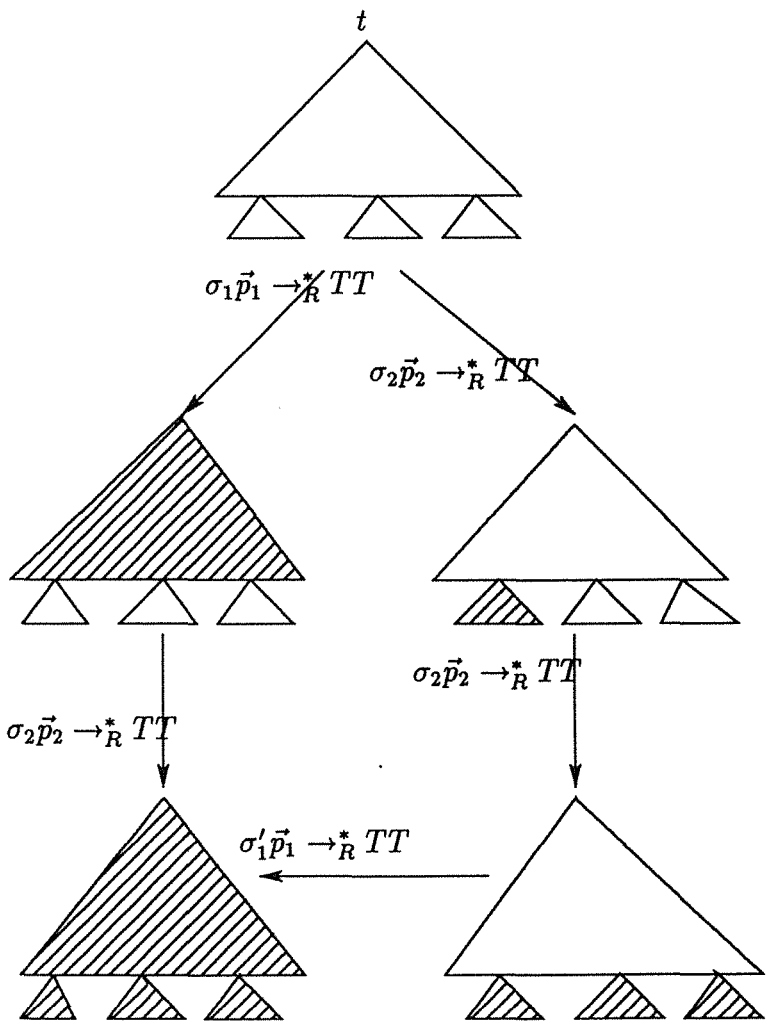
1. u_2 est à une occurrence de g_1 qui correspond à une variable,
   $u_2 = u_1.u$ et $u = v_1.v_2$; $g_{1/v_1} = x$, $\sigma_2 g_2 = \sigma_1(x)/_{v_2}$
   soit σ'_1 la substitution définie par:

$$\begin{cases} \sigma'_1(x) = \sigma_1(x)[v_2 \leftarrow \sigma_2 d_2] \\ \sigma'_1(y) = \sigma_1(y),\ \forall y \neq x \end{cases}$$

   $\sigma_1(x) \rightarrow_R \sigma'_1(x)$. En réécrivant $t_{/u_1}$ à chaque occurrence de x, on obtient $t_{/u_1} \rightarrow_R^* \sigma'_1 g_1$;
   le problème consiste à vérifier que $\sigma'_1 \vec{p}_1 \rightarrow_R^* TT$ afin de réécrire $\sigma'_1 g_1$ en $\sigma'_1 d_1$; en réécrivant chaque occurrence de $\sigma_1(x)$, on obtient $\sigma_1 \vec{p}_1 \rightarrow_R^* \sigma'_1 \vec{p}_1$.
   D'une part, $\sigma_1 \vec{p}_1 \rightarrow_R^* TT$ et d'autre part, $\sigma_1 \vec{p}_1 \rightarrow_R^* \sigma'_1 \vec{p}_1$.

   Examinons le terme $\sigma_1 \vec{p}_1$; puisque R est décroissant, $\sigma_1 g_1 \succ\!\!\succ \sigma_1 \vec{p}_1$. Pour la même raison, puisque $\succ$ contient la relation de sous-terme strict, $t \succeq \sigma_1 g_1$. Donc, $t \succ\!\!\succ \sigma_1 \vec{p}_1$.

   Par hypothèse de récurrence, R est confluent en $\sigma_1 \vec{p}_1$. Donc, $\sigma'_1 \vec{p}_1 \rightarrow_R^* TT$, ce qui permet de conclure pour ce cas.


t
$\sigma_1\vec{p}_1 \not\rightarrow_R^* TT$
$\sigma_2\vec{p}_2 \rightarrow_R^* TT$
$\sigma_2\vec{p}_2 \rightarrow_R^* TT$
$\sigma_2\vec{p}_2 \rightarrow_R^* TT$
$\sigma_1'\vec{p}_1 \rightarrow_R^* TT$

2. u_1 est à une occurrence de g_2 qui correspond à une variable ; ce cas est similaire au précédent ;
3. u_2 est à une occurrence de g_1 qui ne correspond pas à une variable, soit $u_2 = u_1.u$ et les termes $g_{1/u}$ et g_2 peuvent être unifiés par $\sigma_1 \cup \sigma_2$. Soit μ leur unificateur le plus général ; il existe τ tel que $\sigma_1 \cup \sigma_2 = \tau.\mu$.
Nous avons alors une superposition qui engendre la paire critique $\vec{C} \Rightarrow M = N$ où

$$\vec{C} = \mu\vec{p}_1 \ \wedge \ \mu\vec{p}_2, \ M = t[u_1 \leftarrow \mu d_1] \text{ et } N = t[u_2 \leftarrow \mu d_2]$$

De plus, $\sigma_1 g_{1/u} = \sigma_2 g_2$.
Nous avons alors $\tau\vec{C} \rightarrow_R^* TT$ puisque
   - $\sigma_1\vec{p}_1 \rightarrow_R^* TT$, $\sigma_2\vec{p}_2 \rightarrow_R^* TT$,
   - $\sigma_1 = \tau.\mu$, $\sigma_2 = \tau.\mu$,
   - $\vec{C} = \mu\vec{p}_1 \ \wedge \ \mu\vec{p}_2$.

   Soit $\{\vec{C}_i \Rightarrow M_i = N_i\}_{i \in I}$ un EFCS saturé de $\vec{C} \Rightarrow M = N$.

   Puisque $t \succ\!\!\succ \tau\vec{C}$ et $t \succ \tau M$, τN et que par hypothèse de récurrence, R est confluent en tout terme clos plus petit que t par l'ordre $\succ$, les conditions d'application du lemme 4.2 sont vérifiées. Par ce lemme 4.2, il existe alors un indice k dans I tel que $\tau\vec{C}_k \rightarrow_R^* TT$, $\tau M \rightarrow_R^* \tau M_k$ et $\tau N \rightarrow_R^* \tau N_k$.

   Soit $\vec{C}_k$ de la forme $\wedge_{j=1}^m c_j = c'_j$ avec $c'_j \in \{true, false\}$;
   $\tau\vec{C}_k \rightarrow_R^* TT$ signifie que pour tout j dans $[1 \ldots m]$, $\tau c_j \rightarrow_R^* c'_j$.

   Selon les conditions du théorème, 4 cas doivent être considérés :

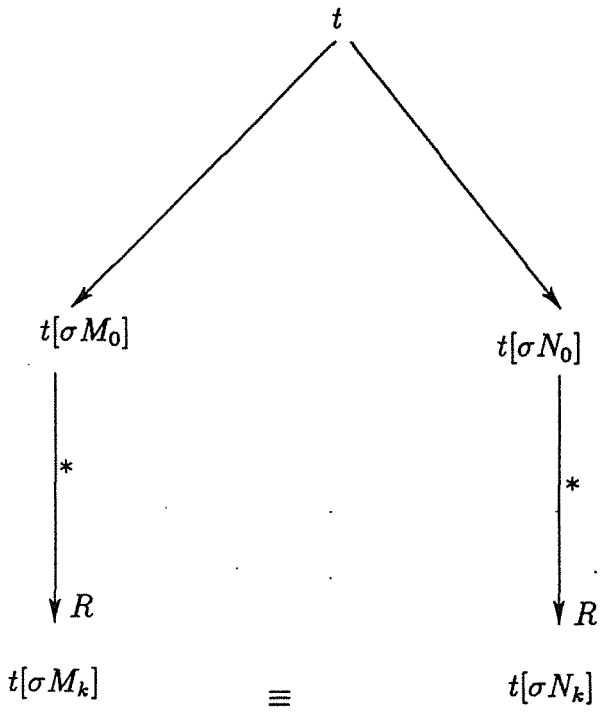
   (a) 1) soit $\vec{C}_k$ est de la forme $true = false \wedge \vec{C}'$; puisque $\tau\vec{C}_k \rightarrow_R^* TT$, donc $true \rightarrow_R^* false$, ce qui contredit l'hypothèse d'irréductibilité de la constante $true$.
   2) soit $\vec{C}_k$ est de la forme $false = true \wedge \vec{C}'$; puisque $\tau\vec{C}_k \rightarrow_R^* TT$, donc $false \rightarrow_R^* true$, ce qui contredit l'hypothèse d'irréductibilité de la constante $false$.
   Par conséquent, ce cas ne peut pas se produire, et $\vec{C}_k$ ne contient ni l'équation $true = false$ ni l'équation $false = true$.

   (b) soit $\vec{C}_k$ est de la forme $p = true \ \wedge \ p = false \wedge \vec{C}'$;

   Par hypothèse, $t \succ \tau M$, τN, et $t \succ\!\!\succ \tau\vec{C}$. D'autre part, $\vec{C}_k \Rightarrow M_k = N_k$ est dérivée à partir de $\vec{C} \Rightarrow M = N$ par 0 ou plusieurs applications de $SCQE$ et $SCDE$. Par le lemme 4.1, $t \succ\!\!\succ \tau\vec{C}_k$. Donc, $t \succ \tau p$.

   D'autre part, puisque $\tau\vec{C}_k \rightarrow_R^* TT$, par définition, $\tau p \rightarrow_R^* true$ et $\tau p \rightarrow_R^* false$. Par récurrence, R est confluent en τp. Il existe alors un terme clos t' tel que $true \rightarrow_R^* t'$ et $false \rightarrow_R^* t'$, ce qui contredit l'hypothèse d'irréductibilité des constantes $true$ et $false$.
   Par conséquent, ce cas ne peut pas non plus se produire, et $\vec{C}_k$ ne contient pas d'équation de la forme $p = true \ \wedge \ p = false$.

(c) soit $M_k \equiv N_k$. Par le lemme 4.2, $\tau M \rightarrow_R^* \tau M_k$ et $\tau N \rightarrow_R^* \tau N_k$. Par conséquent, $t[\tau M] \rightarrow_R^* t[\tau M_k] \equiv t[\tau N_k] \leftarrow_R^* t[\tau N]$, ce qui clôt le diagramme de dérivation. Dans ce cas, R est localement confluent en t.



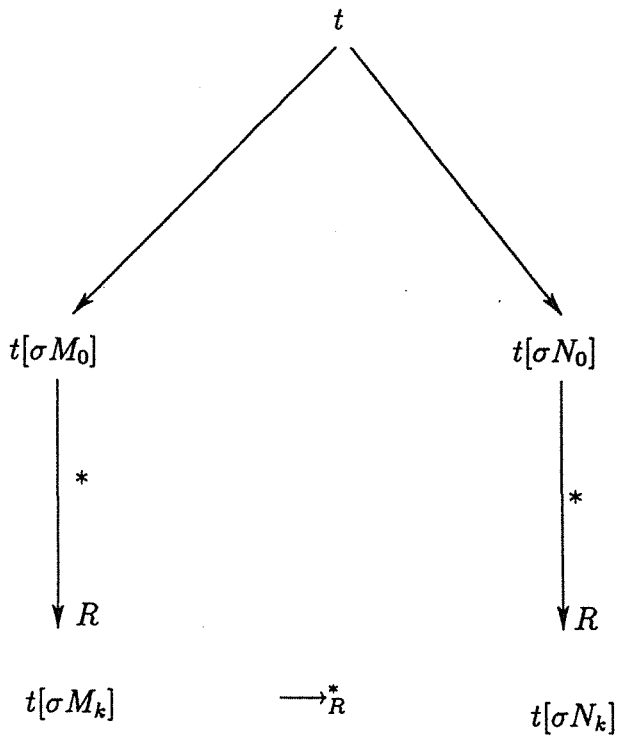
(d) 1) soit l'équation $M_k = N_k$ appartient à $\vec{C}_k$. Puisque $\tau\vec{C}_k \rightarrow_R^* TT$, soit $\tau M_k \rightarrow_R^* \tau N_k$, soit $\tau N_k \rightarrow_R^* \tau M_k$. Par le lemme 4.2, on a $\tau M \rightarrow_R^* \tau M_k$ et $\tau N \rightarrow_R^* \tau N_k$;

- Supposons que $\tau M_k \rightarrow_R^* \tau N_k$; donc $t[\tau M] \rightarrow_R^* t[\tau M_k] \rightarrow_R^* t[\tau N_k] \leftarrow_R^* t[\tau N]$.

- si $\tau N_k \rightarrow_R^* \tau M_k$, le cas est similaire : $t[\tau M] \rightarrow_R^* t[\tau M_k] \leftarrow_R^* t[\tau N_k] \leftarrow_R^* t[\tau N]$.

2) soit l'équation $N_k = M_k$ appartient à $\vec{C}_k$. Puisque $\tau\vec{C}_k \rightarrow_R^* TT$, soit $\tau M_k \rightarrow_R^* \tau N_k$, soit $\tau N_k \rightarrow_R^* \tau M_k$. Nous retrouvons les mêmes cas que ci-dessus, et le traitement est similaire.

Par conséquent, dans le cas 4, R est aussi localement confluent en t.



Dans les deux derniers cas où l'indice k peut exister, R est localement confluent en t.

4. u_1 est à une occurrence de g_2 ne correspondant pas à une variable. Ce cas est similaire au cas précédent.

A partir de l'étude exhaustive de tous les cas possibles, on peut conclure que R est localement confluent en t.

Puisque R est décroissant, la relation $\rightarrow_R$ termine. Par *une variante du* lemme de Newman [82], puisque $\rightarrow_R$ est nœthérienne et R est localement confluent en t, R est aussi confluent en t.

Par récurrence, puisque R est confluent en tout terme clos t' plus petit que t par l'ordre $\succ$, et que R est confluent en t, R est alors confluent sur tous les termes clos de $T(F)$.

Montrons à présent que $E(R)$ est consistante par rapport aux booléens.
Preuve par l'absurde : Supposons que $E(R)$ ne soit pas consistante par rapport aux booléens. Par conséquent, $true =_{E(R)} false$. Puisque R est confluent sur les termes clos, il existe un terme clos t tel que $true \rightarrow_R^* t$ et $false \rightarrow_R^* t$, ce qui réfute l'hypothèse d'irréductibilité de $true$ et $false$. □

5 Exemples

5.1 Des exemples de preuve de confluence sur les termes clos

Ce premier exemple montre que dans certains cas, les problèmes que posent les paires critiques non-décroissantes peuvent être résolus par simplification contextuelle dans la partie condition. Cet exemple inspiré de [54], spécifie les entiers avec les constructeurs 0, successeur et prédecesseur et l'opérateur d'ordre $<$.

Exemple 4.8 *Soit la spécification des entiers munie des constructeurs* **o**, ***succ*** *et* ***pred*** *pour la sorte* ***int*** *dans laquelle nous définissons la relation d'ordre* $<$ *;*
$SPEC = \{S,\ F,\ E\}$ *avec* $S = \{int,\ bool\}$ *;*
$F = \{0 :\rightarrow int,\ succ,\ pred : int \rightarrow int,$
$true,\ false :\rightarrow bool,$
$< : int \times int \rightarrow bool\}$ *;*

Equations initiales E :

Relations entre les constructeurs :

1. $succ(pred(x)) = x$
2. $pred(succ(x)) = x$

Définition de l'opérateur d'ordre $<$:

3. $0 < 0 = false$
4. $0 < succ(0) = true$
5. $succ(x) < y = x < pred(y)$
6. $pred(x) < y = x < succ(y)$
7. $y < x = true \Rightarrow y < succ(x) = true$
8. $y < x = false \Rightarrow y < pred(x) = false.$

Les règles sont obtenues en orientant les équations de E de gauche à droite.

La paire critique

$$pred(x) < y = false \Rightarrow x < succ(pred(y)) = false$$

est obtenue en superposant les règles 6 et 8. Elle sera simplifiée d'abord par les règles inconditionnelles comme suit :

$$\frac{pred(x) < y = false \Rightarrow x < succ(pred(y)) = false}{pred(x) < y = false \Rightarrow x < y = false}$$

par la règle 1.

$$\frac{pred(x) < y = false \Rightarrow x < y = false}{x < succ(y) = false \Rightarrow x < y = false}$$

par la règle 6.

Cette forme contextuelle simplifiée de la paire critique initiale ne vérifie aucun des quatre critères de convergence. On peut également remarquer qu'elle n'est pas décroissante (sa précondition n'est pas plus petite par l'ordre $\gg$ que sa conséquence). Cependant, elle peut être réduite par la règle 7, et nous obtenons :

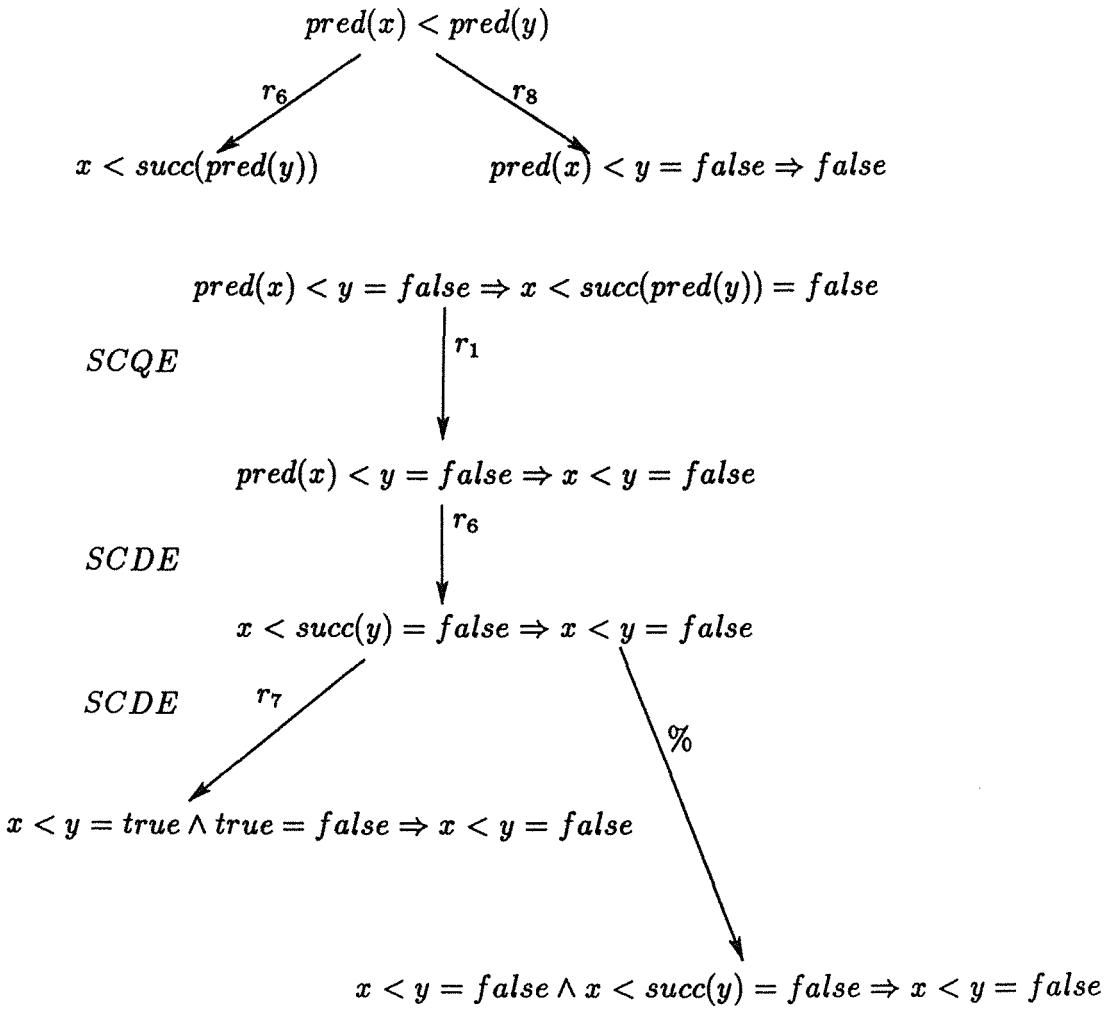
$$1.\ true = false \ \wedge \ x < y = true \Rightarrow x < y = false$$

obtenue en utilisant la règle 7 ;

$$2.\ x < succ(y) = false \ \wedge \ x < y = false \Rightarrow x < y = false$$

obtenue par la règle complémentaire de 7.

La condition de la première équation est triviale : elle répond au premier critère de convergence. La seconde équation vérifie quant à elle, le dernier critère. Nous pouvons conclure par conséquent que la paire critique calculée converge sur les termes clos.


pred(x) < pred(y)
r6
r8
x < succ(pred(y))
pred(x) < y = false ⇒ false
pred(x) < y = false ⇒ x < succ(pred(y)) = false
SCQE
r1
pred(x) < y = false ⇒ x < y = false
r6
SCDE
x < succ(y) = false ⇒ x < y = false
SCDE
r7
%
x < y = true ∧ true = false ⇒ x < y = false
x < y = false ∧ x < succ(y) = false ⇒ x < y = false

Une autre paire critique est obtenue en superposant les règles 5 et 7 :

$$succ(x) < y = true \Rightarrow x < pred(succ(y)) = true$$

Elle sera simplifiée en premier lieu par les règles inconditionnelles comme suit :

$$\frac{succ(x) < y = true \Rightarrow x < pred(succ(y)) = true}{succ(x) < y = true \Rightarrow x < y = true}$$

en utilisant la règle 2.

$$\frac{succ(x) < y = true \Rightarrow x < y = true}{x < pred(y) = true \Rightarrow x < y = true}$$

en utilisant la règle 5.

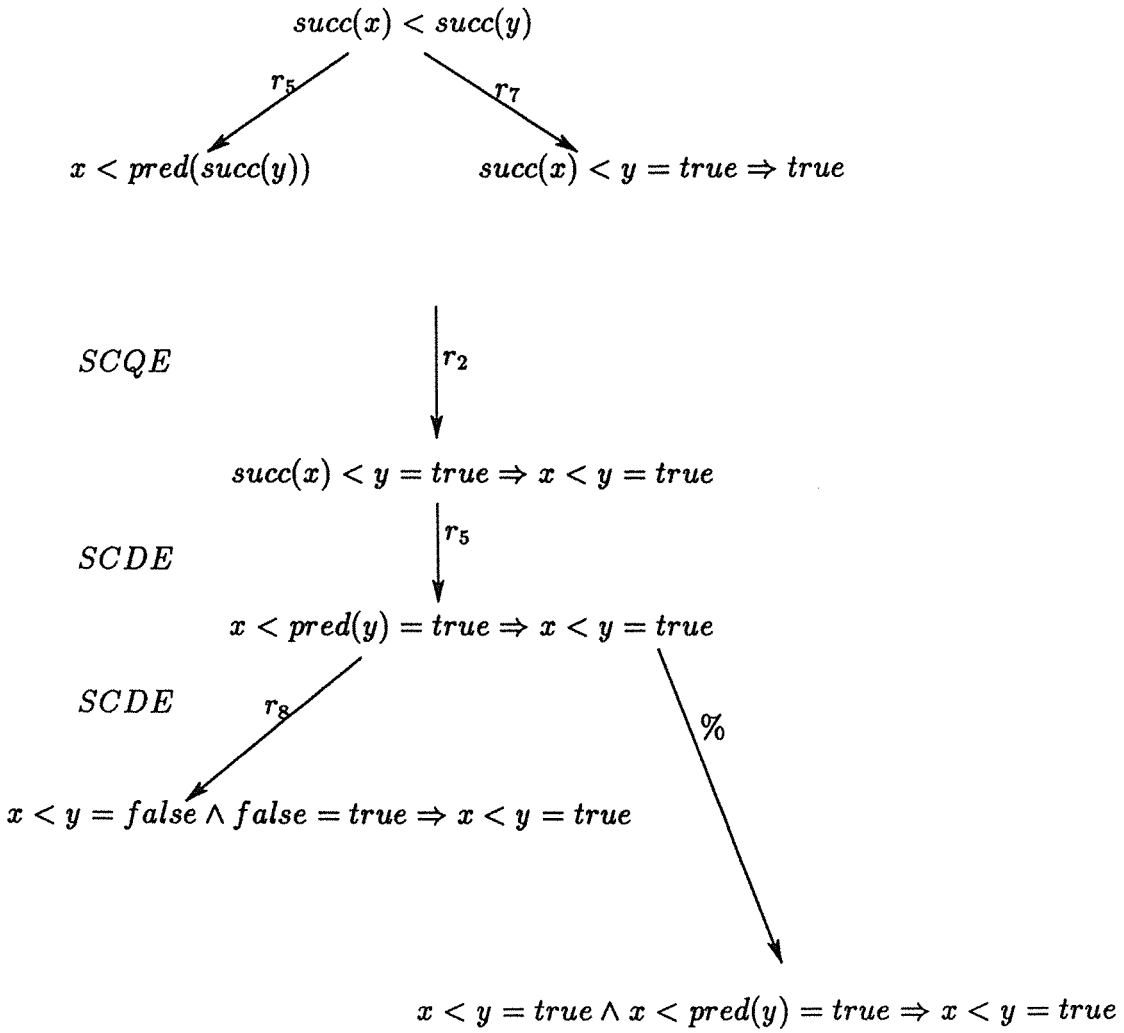
Cette forme contextuellement simplifiée ne vérifie aucun critère de convergence ; elle n'est pas décroissante non plus. Toutefois, elle peut être réduite par la règle 8 et nous obtenons :

$$1.\ false = true \ \wedge \ x < y = false \Rightarrow x < y = true$$

$$2.\ x < pred(y) = true \ \wedge \ x < y = true \Rightarrow x < y = true$$

La première équation est obtenue en utilisant la règle 8 et elle vérifie le premier critère de convergence puisque sa condition contient $false = true$. Nous obtenons la seconde équation en considérant le complémentaire de la règle 8. Cette seconde équation vérifie le dernier critère du théorème. Par conséquent, les deux formes contextuellement simplifiées de la paire critique sont triviales.

Au cours du test de confluence, seules ces deux paires critiques sont calculées. Puisqu'elles sont toutes deux convergentes, nous pouvons affirmer que le système obtenu en orientant les équations de E de gauche à droite est confluent sur les termes clos et que la spécification correspondante est consistante par rapport aux booléens. ◇


succ(x) < succ(y)
r5
r7
x < pred(succ(y))
succ(x) < y = true ⇒ true
SCQE
r2
succ(x) < y = true ⇒ x < y = true
SCDE
r5
x < pred(y) = true ⇒ x < y = true
SCDE
r8
%
x < y = false ∧ false = true ⇒ x < y = true
x < y = true ∧ x < pred(y) = true ⇒ x < y = true

L'utilisation de la règle d'inférence qui permet de supprimer une équation dont la conséquence est contenue dans la condition est primordiale. Cette règle d'inférence est égalemet utilisée sous une autre forme par Navarro [79]. Elle est en effet traduite par une règle de réécriture spécifique :

$$p \Rightarrow p \rightarrow true.$$

et il est alors essentiel de l'appliquer avec précaution et uniquement pour simplifier une condition. En effet, si cette règle est utilisée pour simplifier la partie conséquence, elle peut produire des dérivations infinies.

Ganzinger, dans ses travaux, traite cet exemple par une procédure spécifique qui consiste à superposer les règles sur les conditions des équations non-réductrices tandis que dans notre cas, le traitement est inclus dans le processus de réécriture lui-même. Il est intéressant de constater que la stratégie de simplification d'une précondition de règle par le processus de raisonnement par cas, permet d'éliminer certaines équations non-décroissantes. Nous présentons dans ce qui suit, un second exemple pour illustrer la puissance du processus de réécriture dans les préconditions de règles.

Exemple 4.9 *Cet exemple est inspiré de [34]. Il s'agit de la spécification des listes avec un opérateur de suppression* **delete** *et une notation de liste comme dans Prolog.*

1. $x \neq y = true \Rightarrow delete(x,\ [y|u]) \rightarrow [y|delete(x,\ u)]$
2. $has(x,\ u) = false \Rightarrow delete(x,\ u) \rightarrow u$
3. $x \neq y = true \Rightarrow has(x,\ [y|u]) \rightarrow has(x,\ u)$

La superposition entre (1) et (2) produit la paire critique

$$x \neq y = true \ \wedge\ has(x,\ [y|u]) = false \Rightarrow [y|delete(x,\ u)] = [y|u].$$

En utilisant la règle (3) dans la partie condition, nous obtenons

1. $x \neq y = true \ \wedge\ has(x,\ u) = false \Rightarrow [y|delete(x,\ u)] = [y|u]$

2. $x \neq y = true \ \wedge\ has(x,\ [y|u]) = false \ \wedge\ (x \neq y) = false \Rightarrow [y|delete(x,\ u)] = [y|u]$

La première équation est obtenue en utilisant la règle (3) et la seconde en considérant le cas complémentaire. Cette seconde équation vérifie le second critère de convergence. A présent, pour la première, nous pouvons utiliser la règle (2) et nous obtenons :

$$a.\ \ x \neq y = true \ \wedge\ has(x,\ u) = false \Rightarrow [y|u] = [y|u]$$

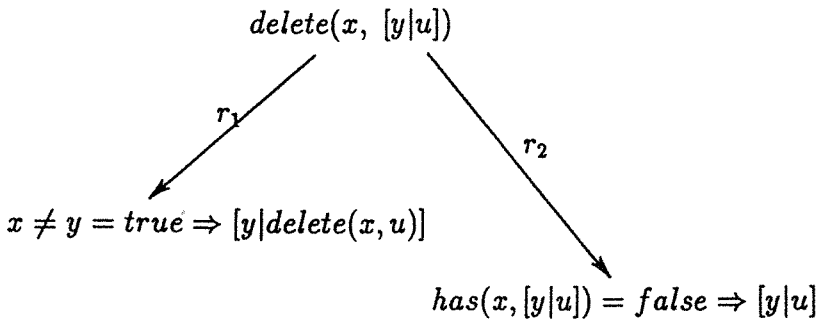
$$b.\ \ x \neq y = true \ \wedge\ has(x,\ u) = false \ \wedge\ has(x,\ u) = true \Rightarrow [y|delete(x,\ u)] = [y|u]$$

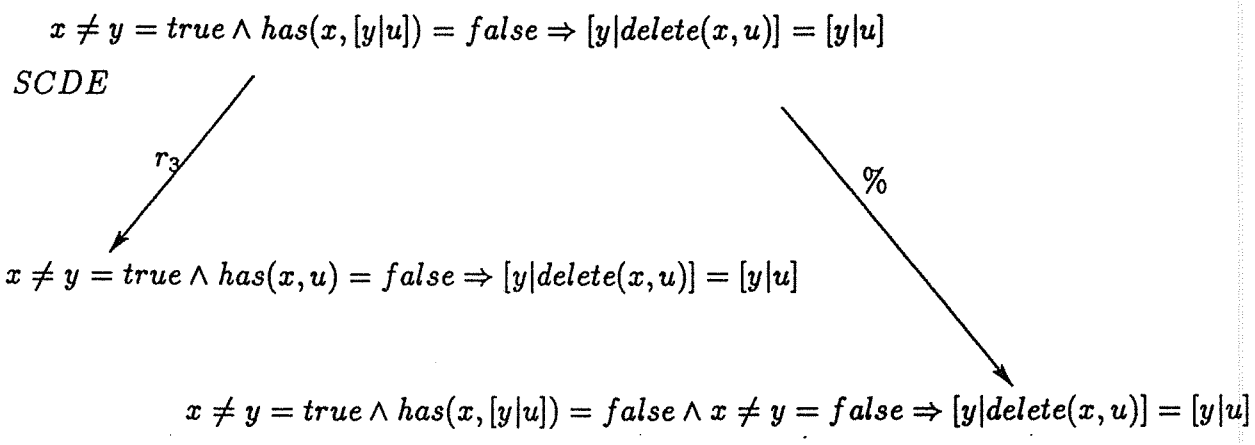
L'équation (a) est obtenue en simplifiant le premier membre de la conséquence par la règle (2) et l'équation (b) en considérant le cas complémentaire. (a) est triviale puisque les deux membres de sa conséquence sont identiques (il s'agit du troisième critère de convergence), et (b) est triviale puisqu'elle vérifie le second critère.

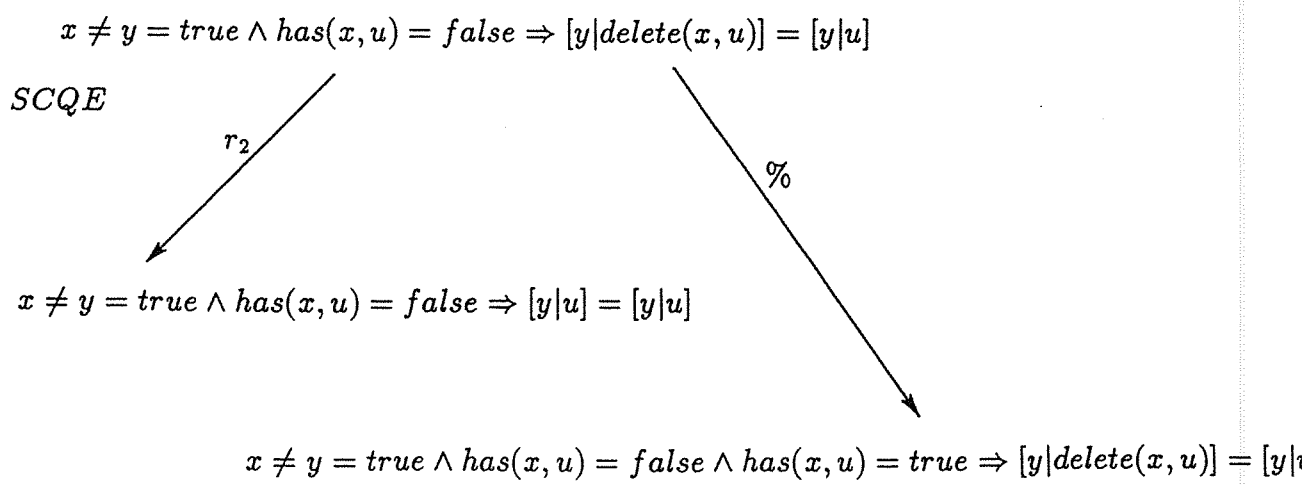
Ces vérifications permettent de conclure que la paire critique est convergente sur les termes clos.

Notons que la paire critique initiale peut également être simplifiée dans sa partie terme par la règle (2) et nous obtenons le même résultat. Cette paire critique dans les travaux de Ganzinger, est supprimée par une procédure spécifique qui consiste à réécrire un terme en

considérant les équations de la condition comme des règles additionnelles et il nécessite de vérifier certaines hypothèses. Dans nos travaux, la suppression de certaines paires critiques non-décroissantes s'inclut dans le processus naturel de réécriture, suffisant pour prouver la correction de cette suppression. ◇


delete(x, [y|u])
r1
r2
x ≠ y = true ⇒ [y|delete(x, u)]
has(x, [y|u]) = false ⇒ [y|u]


x ≠ y = true ∧ has(x, [y|u]) = false ⇒ [y|delete(x, u)] = [y|u]
SCDE
r3
%
x ≠ y = true ∧ has(x, u) = false ⇒ [y|delete(x, u)] = [y|u]
x ≠ y = true ∧ has(x, [y|u]) = false ∧ x ≠ y = false ⇒ [y|delete(x, u)] = [y|u]


x ≠ y = true ∧ has(x, u) = false ⇒ [y|delete(x, u)] = [y|u]
SCQE
r2
%
x ≠ y = true ∧ has(x, u) = false ⇒ [y|u] = [y|u]
x ≠ y = true ∧ has(x, u) = false ∧ has(x, u) = true ⇒ [y|delete(x, u)] = [y|

6 Conclusion

Nous avons étudié dans ce chapitre un processus de réécriture conditionnelle pour mettre en œuvre un test de confluence sur les termes clos d'une classe spécifique de systèmes de réécriture conditionnelle. Ce test est formalisé à partir d'un certain nombre de concepts intéressants et différents, sous plusieurs aspects, de ceux que nous adaptés dans le chapitre précédent :

- Le premier aspect se situe dans la relation de réécriture utilisée. Celle-ci est définie à partir du raisonnement par cas, dans sa forme la plus souple. En effet, il n'est pas nécessaire d'exiger que la propriété de bonne couverture soit vérifiée. On s'assure plutôt la complétude de la réécriture contextuelle en tenant compte de la branche complémentaire.

- Le test de convergence des paires critiques est en lui-même très différent, puisque nous ne procédons plus de la même manière. Dans la stratégie du chapitre précédent, il s'agissait de simplifier les termes contextuels qui composent la paire critique jusqu'à leur forme normale, de combiner deux à deux les formes normales contextuelles obtenues et de vérifier que, soit les termes sont syntaxiquement égaux, soit les contextes sont mutuellement exclusifs. C'est une stratégie que nous pouvons qualifier de *stratégie du tout ou rien.* Dans ce travail, nous choisissons plutôt de simplifier les paires critiques pas à pas, et de vérifier après chaque simplification, si la convergence est vérifiée ou non. Ce choix permet de déceler la convergence de la paire critique le plus tôt possible.

- Un autre aspect qui distingue les deux stratégies est que nous manipulons des équations conditionnelles au lieu de termes contextuels. Le cas trivial où la conséquence est une partie de la précondition est ainsi plus facile à déceler. Il peut être source de problèmes dans la stratégie du chapitre 3.
  Examinons pour cela l'exemple 4.8. L'équation

  $$x < y = true \ \wedge \ x < pred(y) = true \Rightarrow x < y = true$$

  peut être de façon triviale éliminée puisqu'elle vérifie le dernier critère de convergence du théorème principal. Par contre, si l'on considère les termes contextuels

  - $x < y = true \ \wedge \ x < pred(y) = true \ :: \ x < y$ et
  - $x < y = true \ \wedge \ x < pred(y) = true \ :: \ true$

  il est plus difficile de montrer que ces c-termes produisent une équation triviale.

- Enfin, la forme des préconditions est totalement différente. Cette forme fait que dans ce présent travail, les critères de convergence d'une paire critique sont plus restreints, mais plus simples à tester, notamment en ce qui concerne l'insatisfiabilité du contexte. Il serait toutefois intéressant de les étendre avec cette même stratégie.

- Un avantage aux résultats de ce présent travail par rapport à ceux que nous avons établi dans le précédent chapitre concerne la classe des systèmes définis. Il est certain en effet que les systèmes de réécriture décroissants sont plus souples d'utilisation que les systèmes hiérarchiques. La hiérarchie interdit par exemple des définitions récursives du type

  $$0 \leq x = true \Rightarrow 0 \leq succ(x) = true$$

puisque l'opérateur $\leq$ que l'on définit apparaît en même temps dans la conséquence et dans la précondition de la règle.

Par ailleurs, une particularité importante du processus de réécriture étudié est qu'il permet de réécrire la condition, aussi bien que la conséquence d'une équation ou d'une règle conditionnelle. Ceci simplifie en grande partie la preuve de contextes. Nous avons formulé la preuve de confluence sur les termes clos en établissant des critères bien définis de convergence des paires critiques. Il devient toutefois nécessaire d'étendre ces critères à des formes plus larges d'équations. A ce stade d'évolution de nos travaux, nous préconisons d'établir une articulation entre l'aspect algébrique de la réécriture et la théorie du premier ordre. La théorie que nous avons mise en œuvre est suffisamment applicable et l'approche que nous avons considérée est implantable. Nous disposons en effet d'un prototype expérimental réalisé par une étudiante au cours de son stage de Diplôme d'Etudes Approfondies. Ce prototype est présenté dans le chapitre suivant et des exemples d'exécution sont illustrés en annexe.

D'autres travaux ont été menés parallèlement aux nôtres dans le domaine conditionnel, et en qui concerne notre pôle d'intérêt dans ce chapitre, nous pouvons citer deux approches voisines de celle que nous avons considérée.

La première est celle de Ganzinger [35] dans laquelle l'auteur utilise des méthodes contextuelles pour formaliser un processus de réécriture conditionnelle. Etant donné un ensemble d'équations inconditionnelles C de la forme $\{c_1 = c'_1, \ldots, c_k = c'_k\}$, t se réécrit en t' par une règle conditionnelle r si la précondition instanciée de r peut être prouvée opérationnellement valide en utilisant l'ensemble $C\downarrow$. $C\downarrow$ est calculé en effectuant la fermeture de C par les axiomes de réflexivité, de congruence et de clôture par substitution. Cette relation peut être vue comme une extension de la réécriture sous contexte. En effet, tandis que cette dernière réécrit un terme sous le contexte C, la relation formalisée par Ganzinger est plus riche puisqu'elle utilise toutes les assertions de $C\downarrow$. Dans [34], l'auteur présente un algorithme de complétion conditionnelle des systèmes de réécriture réducteurs, fondé sur des règles d'inférence dont la preuve est établie par la méthode des ordres de preuve élaborée par Bachmair, Dershowitz et Hsiang [2]. Cet algorithme peut être utilisé pour tester la confluence sur les termes clos et il est enrichi par des procédures spécifiques qui consistent principalement à superposer des membres gauches de règles conditionnelles sur des préconditions d'équations non-réductrices. Dans certains cas, ce traitement permet de supprimer ce type d'équations du système. De plus, l'auteur montre que si de telles équations persistent dans le système, elles ne sont pas utiles pour effectuer des preuves. Elles sont toutefois gardées pour le cas où l'utilisateur choisirait d'enrichir la spécification. Cependant, Ganzinger considère la théorie équationnelle puisqu'il effectue seulement des preuves d'équations inconditionnelles. D'autre part, il ajoute des théorèmes inductifs à la spécification en supposant acquise leur validité opérationnelle. Ce traitement est lié au problème de validité dans le modèle initial et l'auteur n'approfondit pas la question.

Une seconde approche mise en œuvre est celle adaptée par Rusinowitch et Kounalis [67]. Les auteurs travaillent sur des clauses de Horn et obtiennent un algorithme fondé sur une stratégie de réfutation complète qui utilise un procédé de saturation d'un ensemble de clauses de Horn, et permet d'établir des preuves si l'algorithme engendre la clause vide. L'idée de superposer les membres gauches de règles sur des préconditions dans les travaux de Ganzinger est inspirée de cette méthode de saturation. Cependant, le lien entre les deux axes de travail n'est pas encore établi et les preuves des deux algorithmes sont différentes.

Les auteurs établissent des conditions suffisantes de décision pour le problème du mot par normalisation conditionnelle dans une théorie de Horn spécifique et ils montrent par ailleurs comment prouver des théorèmes dans les modèles initiaux des théories de Horn qui préservent les termes clos.

Pour cette partie de notre travail, nous envisageons d'utiliser ce procédé de réécriture étudié dans ce chapitre, pour établir une procédure de complétion conditionnelle des systèmes de réécriture décroissants. Il s'agirait d'ajouter au système d'équations les paires critiques non-convergentes en vue de les traiter comme les équations initiales de la spécification. Cette procédure de complétion dans une étape simple, effectuerait uniquement de la simplification d'équations, puis dans un second temps, ferait intervenir la simplification des règles elles-mêmes.

Il serait intéressant par ailleurs d'étendre les critères de convergence des paires critiques contextuelles et d'étudier les conditions sous lesquelles il serait possible d'établir une interaction de l'algorithme de confluence sur les termes clos avec un prouveur de théorèmes du premier ordre. Un travail conséquent reste en effet à effectuer pour utiliser de manière plus efficace les méthodes de preuve du premier ordre dans les preuves de contextes.

Cette dernière voie de recherche que nous suggérons nous semble en effet essentielle puisqu'elle permettrait de combiner l'aspect opérationnel et déterministe de la réécriture avec la puissance de preuve dans le premier ordre. Dans le chapitre qui suit, notre objectif est d'illustrer l'importance de cette idée dans un premier temps par des exemples, puis par une synthèse d'un certain nombre de travaux dans lesquels les auteurs mettent aussi en évidence cette nécessité.

CHAPITRE 5

CONFLUENCE ET COMPLETION DES SYSTEMES DE REECRITURE CONDITIONNELLE : LES PROBLEMES D'IMPLANTATION.

Chapitre 5

Confluence et complétion des systèmes de réécriture conditionnelle : les problèmes d'implantation.

1 Introduction

Dans les spécifications conditionnelles de types abstraits algébriques, la sorte booléenne joue un rôle prépondérant. Contrairement au cadre classique, sans préconditions, le caractère opérationnel de la réécriture conditionnelle n'est pas suffisant pour mener à bien les preuves de confluence. En effet, pour prouver la convergence des paires critiques calculées au cours du processus de complétion, il est indispensable d'effectuer des preuves d'insatisfiabilité pour les conditions de certaines d'entre elles. Ces preuves d'insatisfiabilité dépendent dans une large mesure, de la classe de modèles à laquelle on s'intéresse. Nous avons vu qu'il existe principalement deux approches, la première définit des spécifications conditionnelles sans booléens et elle appréhende la classe de toutes les algèbres libres qui satisfont les axiomes de la spécification. Dans cette approche, les preuves de convergence utilisent un algorithme de surréduction pour calculer l'ensemble des substitutions qui forment les solutions de la condition. Cet ensemble est souvent infini. Pour éviter les situations de non-terminaison, il est nécessaire d'avoir recours à des algorithmes de surréduction bornée, tels celui proposé par Jouannaud et Waldmann [53]. La seconde approche définit des spécifications avec booléens. Pour prouver la convergence des paires critiques, elle utilise des assertions booléennes. Des techniques de pré-traitement de ces assertions deviennent alors indispensables pour prouver la validité opérationnelle de certains contextes. La solution préconisée est d'effectuer un traitement au besoin en câblant un prouveur de théorèmes pour ce type de preuves. Notre objectif dans ce chapitre, est de nous pencher sur les difficultés que peut causer l'utilisation des booléens dans les spécifications conditionnelles. Notre intention est de mettre en évidence les différents types de problèmes rencontrés et d'établir la connexion entre le caractère opérationnel des systèmes de réécriture conditionnelle et la logique du premier ordre. Cette idée a également été suggérée par certains auteurs tels Ganzinger [35] et Plaisted [91].

Ce chapitre constitue une synthèse des divers problèmes additionnels que peut particulièrement poser le cadre de la réécriture conditionnelle aussi bien lorsqu'on veut tester la

confluence sur les termes clos d'un système que lorsqu'on veut compléter le système pour le rendre canonique. Il se veut essentiellement prospectif. Dans sa première partie, le premier paragraphe illustre par quelques exemples, le type de paires critiques susceptibles d'être engendrées par le test de confluence ou par le processus de complétion et qu'il est délicat, voire impossible, de traiter uniquement par le procédé de réécriture conditionnelle. Cette partie introduit la nécessité d'utiliser des assertions auxiliaires pour effectuer les preuves de convergence de ces paires critiques. Les assertions auxiliaires sont par la suite formellement définies dans la seconde partie du paragraphe, où l'on montre également comment exprimer et mettre à profit ce type de lemmes pour mener à bien les preuves dans les contextes. La partie qui suit fait apparaître la sensibilité de ces preuves à la forme des spécifications conditionnelles et illustre par là-même leur complexité. Ces preuves de contextes ont également été abordées par divers auteurs, certains partant d'une optique *règles de réécriture conditionnelle* et d'autres d'une optique *clauses de Horn*. Combiner la puissance du processus de réduction conditionnelle au savoir-faire dans les preuves du premier ordre s'avère une nécessité qui apparaît dans la plupart des travaux. Elle constitue de plus une issue prometteuse intéressante à explorer. Ces approches sont présentées dans le troisième paragraphe. Le paragraphe suivant explicite avec davantage de détails la règle d'inférence du test de convergence des paires critiques. L'application de cette règle est étroitement liée aux preuves dans les contextes. Nous expliquons dans cette partie le choix que nous avons adopté pour implanter cette règle d'inférence ainsi que les extensions envisageables pour augmenter la puissance de preuve. Nous présentons également un prototype qui a été réalisé au sein de l'équipe et dont l'objectif est de tester la confluence sur les termes clos. L'originalité de ce logiciel se situe essentiellement dans la preuve d'insatisfiabilité des contextes. En effet, il peut effectuer ce type de preuves en se basant uniquement sur les quatre critères de convergence du théorème 4.1 (voir chapitre 4), ou en utilisant un prouveur de théorèmes du premier ordre avec lequel il interagit. Le dernier paragraphe met finalement en évidence la sensibilité du test de confluence à la complétude de définition d'un opérateur de la signature. Nous concluons enfin en suggérant un certain nombre de directions de travail pouvant aboutir à une implantation plus efficace de la preuve de confluence sur les termes clos et de la complétion conditionnelle.

2 Problèmes de confluence et de complétion avec les méthodes de réécriture conditionnelle

Notre objectif est d'étudier la confluence sur les termes clos des systèmes de réécriture conditionnelle. La confluence sur les termes clos permet d'appréhender un modèle opérationnel pour la programmation fonctionnelle. Les appels fonctionnels peuvent alors être évalués en les réduisant à leur forme normale. De plus, la confluence sur les termes clos est suffisante dans la plupart des applications de la réécriture de termes, telles la preuve de théorèmes inductifs par complétion [32].

Afin d'obtenir un système de réécriture conditionnelle confluent sur les termes clos et de l'utiliser pour effectuer des calculs, nous avons été amenée à étendre le test de confluence [64] au cadre conditionnel. La situation devient cependant plus complexe par la présence des préconditions et certains problèmes se posent lors de l'exécution des tâches classiques de l'algorithme de confluence.

Les résultats de confluence que nous avons nous-même établis (voir chapitre 3 et chapitre 4) et ceux prouvés dans [55, 35] montrent que pour obtenir un système de réécriture conditionnelle confluent (confluent sur les termes clos), il faut prouver la convergence des paires critiques calculées au cours du test de confluence sur l'ensemble $T(F,\ X)$ (sur l'ensemble $T(F)$). Nous nous intéressons dans la suite, uniquement à la confluence sur les termes clos pour nous pencher principalement sur le test de convergence des paires critiques.

La convergence d'une paire critique $C \Rightarrow M = N$ sur les termes clos est établie si pour toute substitution close σ dans $Subst(F)$ telle que σC soit satisfaite, (i.e. telle que $\sigma C \rightarrow_R^* true$), les termes σM et σN convergent. Pour avoir une méthode effective du test de convergence, il est indispensable d'obtenir une approximation de l'ensemble infini de substitutions closes qui satisfont le contexte C. La méthode préconisée est d'utiliser des assertions auxiliaires pour mener à bien les preuves de contextes. Avant d'expliciter cette méthode, examinons les exemples suivants. Il mettent en évidence les problèmes inhérents à la théorie conditionnelle qui peuvent survenir soit au cours du test de confluence, soit au cours du processus de complétion d'un système de réécriture. Ces exemples définissent tantôt des systèmes de réécriture décroissants, tantôt des systèmes hiérarchiques. Il est en effet intéressant de constater que ces problèmes se présentent quelque soit la classe des systèmes considérée.

2.1 Quelques exemples

Exemple 5.1 *Soit* $(S,\ F,\ R)$ *le système de réécriture conditionnelle décroissant définissant le maximum de deux entiers naturels :*

$$S = \{bool\} \oplus \{nat\}\ ;$$

$$F = F_{bool} \oplus \{0 :\rightarrow nat,\ succ : nat \rightarrow nat,$$
$$\leq : nat \times nat \rightarrow bool,$$
$$max : nat \times nat \rightarrow nat\}\ ;$$

$$R = R_{bool} \oplus \{0 \leq x \rightarrow true,\ succ(x) \leq 0 \rightarrow false,\ succ(x) \leq succ(y) \rightarrow x \leq y,$$
$$1.\ x \leq y = true \Rightarrow max(x,\ y) \rightarrow y,$$
$$2.\ x \leq y = false \Rightarrow max(x,\ y) \rightarrow x,$$
$$3.\ x \leq max(x,\ y) \rightarrow true,$$
$$4.\ y \leq max(x,\ y) \rightarrow true\}.$$

Nous nous proposons de calculer les paires critiques entre les règles de R afin de mettre en évidence les diverses situations délicates que nous pouvons rencontrer.

La superposition entre les règles (1) et (3) produit la paire critique:

$$(a) \qquad x \leq y = true \Rightarrow x \leq y = true$$

Cette équation ne peut être orientée que de gauche à droite puisque nous considérons que les constantes *true* et *false* sont irréductibles dans tout système de réécriture R. De plus, il est naturel de considérer que les constantes booléennes sont plus petites dans la précédence que tout autre opérateur de F. La règle r ainsi obtenue engendre une dérivation cyclique et on se heurte à un problème de terminaison. En effet, le membre gauche et la conséquence de r étant identiques, l'évaluation de la précondition réactive de nouveau la règle elle-même et engendre de ce fait une branche infinie de réécritures. Ceci n'est pas surprenant puisque r n'est ni décroissante ni hiérarchique.

La superposition entre les règles (1) et (2) produit la paire critique:

$$(b) \qquad x \leq y = true \ \wedge\ x \leq y = false \Rightarrow x = y$$

Cette paire critique viole la condition qui établit que pour toute équation conditionnelle, les variables de la précondition doivent être toutes contenues dans l'un des membres de la conséquence. Cette condition est imposée dans le but d'éviter des situations nécessitant une surréduction. En effet, considérons à titre d'exemple la règle suivante:

$$p(x,\ y,\ z) = true \Rightarrow g(x,\ y) \rightarrow h(z)$$

Pour appliquer cette règle au terme $g(t, t')$, il est nécessaire de calculer une valeur de z pour laquelle le prédicat $p(t,\ t',\ z)$ soit vrai.
Par conséquent, l'équation obtenue ne peut pas être orientée puisque sa précondition est trop volumineuse (contient plus de variables que sa conséquence).

La superposition entre les règles (2) et (4) produit la paire critique:

$$(c) \qquad x \leq y = false \Rightarrow y \leq x = true$$

De même que la première paire critique, cette équation n'est ni décroissante ni hiérarchique et peut provoquer des dérivations infinies, notamment si un enrichissement de la spécification introduit l'équation duale $y \leq x = false \Rightarrow x \leq y = true$, (cette situation peut se produire si nous définissons le *minimum* de deux entiers naturels et si nous calculons la paire critique entre les règles
$y \leq x = false \Rightarrow min(x,\ y) \rightarrow x$ et $min(x,\ y) \leq y \rightarrow true$).

Et enfin la superposition entre les règles (1) et (4) produit la paire critique:

$$(d) \qquad x \leq y = true \Rightarrow y \leq y = true$$

Cette équation ne peut être orientée car sa précondition contient des variables qui n'apparaissent dans aucun des membres de la conséquence.

Cependant, nous sommes tentée de conclure qu'intuitivement, toutes ces paires critiques sont triviales et peuvent par conséquent être supprimées du système, pour les raisons suivantes:

- (a) est triviale puisque sa conséquence est contenue dans la précondition. D'un point de vue logique, elle traduit la tautologie $p \vee \neg p$.
- (b) a une précondition insatisfiable puisque deux entiers naturels quelconques x et y ne peuvent satisfaire simultanément les assertions $x \leq y = true$ et $x \leq y = false$. Plus généralement, un prédicat quelconque $p(\bar{x})$, où $\bar{x}$ est une suite de variables, ne peut valoir $true$ et $false$ pour les mêmes valeurs de variables de $\bar{x}$.
- (c) traduit une relation naturelle dans l'ensemble des entiers muni de l'ordre $\leq$, à savoir que pour tous entiers naturels x et y, ***si x est plus grand que y alors y est plus petit que x.***
- Finalement, la conséquence de (d) traduit la propriété de réflexivité de l'ordre $\leq$, qui est toujours vraie indépendamment de la validité de la précondition.

Il est absolument indispensable de pouvoir montrer que les quatre paires critiques que nous venons de calculer sont triviales, car il est évident que dans le cas contraire, le test de confluence échouerait pour des raisons telles que la non-terminaison (paire critique (a)) ou la taille (paires critiques (b) ou (d) qui ont plus de variables dans la précondition que dans la conséquence et qui nécessitent alors une procédure de surréduction).

Cet exemple montre qu'il peut exister deux principaux types de paires critiques triviales. Les paires critiques du premier type sont triviales sans qu'il soit nécessaire de connaître les propriétés des prédicats qui les définissent. Les équations (a) et (b) en sont des exemples. Par contre, pour le second type, il est essentiel de définir des assertions auxiliaires pour exprimer les propriétés des prédicats utilisés dans les préconditions de règles et pour permettre de ce fait, d'effectuer des preuves dans les contextes. $\diamond$

Cet exemple que nous venons d'examiner montre que, contrairement au cadre classique, sans préconditions, le caractère opérationnel de la réécriture conditionnelle ne suffit pas pour effectuer correctement les preuves dans les contextes. Il devient indispensable de définir des règles d'inférence spécifiques pour supprimer les équations triviales et même, d'utiliser des techniques de preuves du premier ordre pour prouver l'insatisfiabilité d'un contexte. Ces mêmes techniques peuvent être utilisées pour garantir qu'une étape de réécriture est complète, i.e. pour vérifier qu'une disjonction de contextes booléens équivaut à $true$. Dans la partie qui suit, nous expliquons comment ces preuves de contextes peuvent être menées à bien. Par ailleurs, ce point de vue est, à l'heure actuelle, partagé par une large communauté de chercheurs et nous montrons que divers travaux peuvent être unifiés. Nous présentons auparavant, un second exemple qui met en évidence la nécessité de combiner les propriétés des différents prédicats que l'on peut avoir à utiliser pour spécifier un type abstrait de données. L'exemple définit les listes triées d'entiers naturels (dont la sorte est notée nat), avec les prédicats $\doteq$ et $<$ qui expriment respectivement l'égalité et la relation d'ordre dans la sorte nat. Sur ces listes, nous définissons des opérations d'insertion ($insert$) et de suppression ($delete$). Nous constaterons que l'utilisation du prédicat d'égalité pose des problèmes additionnels.

Exemple 5.2 *Soit* $(S,\ F,\ E)$ *la spécification conditionnelle hiérarchique suivante :*

$SP_0 = SP_{bool}$;

$SP_1 = (S_1,\ F_1,\ E_1),\ S_1 = S_0 \oplus \{nat\}$;

$F_1 = BF_1 \oplus DF_1,\ BF_1 = F_0 \oplus C_1$;
$C_1 = \{0 : nat,\ succ\ :\ nat \rightarrow nat\}$;
$DF_1 = \{<,\ \doteq : nat \times nat \rightarrow bool\}$;

$E_1 = BE_1 \oplus DE_1,\ BE_1 = E_0$;
$DE_1 = \{$
Axiomatisation de $<$:

$x < 0 = false,\ 0 < succ(x) = true,\ succ(x) < succ(y) = x < y,$

Axiomatisation de $\doteq$:

$0 \doteq 0 = true,\ succ(x) \doteq succ(y) = x \doteq y,$
$0 \doteq succ(x) = false,\ succ(x) \doteq 0 = false\ \}.$

$SP_1 = (S_2,\ F_2,\ E_2),\ S_2 = S_1 \oplus \{liste\},$

$F_2 = BF_2 \oplus DF_2,\ BF_2 = F_1 \oplus C_2$;
$C_2 = \{[\ ] :\rightarrow liste,\ cons : nat \times liste \rightarrow liste\},$
$DF_2 = \{delete,\ insert : nat \times liste \rightarrow liste\},$

$E_2 = BE_2 \oplus DE_2,\ BE_2 = E_1,$
$DE_2 = \{$
Axiomatisation de *delete* :

1. $delete(x,\ [\ ]) = [\ ],$
2. $x < y \Rightarrow delete(y,\ cons(x,\ l)) = delete(y,\ l),$
3. $x \doteq y \Rightarrow delete(y,\ cons(x,\ l)) = l,$
4. $y < x \Rightarrow delete(y,\ cons(x,\ l)) = delete(y,\ l)$

Axiomatisation de *insert* :

5. $insert(x,\ [\ ]) = cons(x,\ [\ ]),$
6. $x < y \Rightarrow insert(x,\ cons(y,\ l)) = cons(x,\ cons(y,\ l)),$
7. $x \doteq y \Rightarrow insert(x,\ cons(y,\ l)) = cons(x,\ cons(y,\ l)),$
8. $y < x \Rightarrow insert(x,\ cons(y,\ l)) = cons(y,\ insert(x,\ l))$

Relation entre les opérateurs définis :

9. $x \doteq y \Rightarrow delete(y,\ insert(x,\ l)) = l\ \}.$

Supposons que l'ensemble des équations soit orienté de gauche à droite. Les règles ainsi obtenues forment un système de réécriture conditionnelle hiérarchique.

Examinons la paire critique suivante engendrée à partir de la superposition des règles (6) et (9):

$$x < y \ \wedge \ x \doteq z \Rightarrow \mathit{delete}(z,\ \mathit{cons}(x,\ \mathit{cons}(y,\ l))) = \mathit{cons}(y,\ l)$$

Nous nous proposons de montrer que cette paire critique est triviale. Il s'agit de calculer les formes normales contextuelles de chacun des c-termes qui constituent la paire critique et de vérifier qu'elles forment des ensembles complets de formes normales contextuelles équivalents.

Le terme contextuel

$$x < y \ \wedge \ x \doteq z \ ::\ \mathit{delete}(z,\ \mathit{cons}(x,\ \mathit{cons}(y,\ l)))$$

se réécrit en utilisant la relation $\xrightarrow{c}_{H,R}$, respectivement par les règles (2), (3) et (4) en les c-termes suivants:

1. $x < y \ \wedge \ x \doteq z \ \wedge \ x < z \ ::\ \mathit{delete}(z,\ \mathit{cons}(y,\ l))$,
2. $x < y \ \wedge \ x \doteq z \ \wedge \ x \doteq z \ ::\ \mathit{cons}(y,\ l)$,
3. $x < y \ \wedge \ x \doteq z \ \wedge \ z < x \ ::\ \mathit{delete}(z,\ \mathit{cons}(y,\ l))$.

Sachant que le prédicat $<$ est irréflexif, on peut conclure que les contextes des termes contextuels (1) et (3) sont équivalents à false (les expressions $x \doteq z \ \wedge \ x < z$ et $x \doteq z \ \wedge \ z < x$ sont équivalentes à false). Une difficulté réside cependant dans cette preuve d'insatisfiablitié. En effet, d'une part il est essentiel d'utiliser la propriété d'irréflexivité de $<$ et d'autre part, un traitement efficace de l'égalité est indispensable.

Le c-terme (2) produit une équation triviale puisqu'alors, les deux termes de l'équation sont syntaxiquement égaux. En effet, l'équation devient

$$x < y \ \wedge \ x \doteq z \ \wedge \ x \doteq z \Rightarrow \mathit{cons}(y,\ l) = \mathit{cons}(y,\ l)$$

Par ailleurs, pour achever la preuve de trivialité de la paire critique, il faut également prouver que l'assertion

$$x < z \ \vee \ x \doteq z \ \vee \ z < x$$

est valide dans l'algèbre initiale $I(R)$, pous s'assurer que la réduction du c-terme $x < y \ \wedge \ x \doteq z \ ::\ \mathit{delete}(z,\ \mathit{cons}(x,\ \mathit{cons}(y,\ l)))$ est complète. Cette preuve nécessite toutefois certaines précautions que nous explicitons dans la partie qui suit. $\diamond$

2.2 Les assertions auxiliaires et leur utilisation dans les preuves de contextes

Les assertions auxiliaires sont utiles pour traduire les propriétés de certains prédicats qui ne peuvent pas être exprimées par des règles de réécriture, en particulier parce qu'alors, ces règles engendrent des dérivations infinies. La propriété de transitivité des ordres $\leq$ ou $<$ en est un exemple. Ces assertions sont principalement de deux types:

1. **les assertions négatives**: une assertion négative est la négation d'une conjonction finie d'atomes booléens de la forme:

$$\neg(p_1 \ \wedge \ \ldots \wedge \ p_n)$$

Elle signifie qu'il n'existe aucune substitution close σ, telle que pour tout i dans $[1 \ldots n]$, les préconditions σp_i soient simultanément valides. Les assertions négatives peuvent impliquer l'insatisfiabilité de la condition d'une paire critique et permettre de ce fait, de la supprimer.

Les expressions suivantes sont toutes des assertions négatives:

$$1.\ \neg(x < y \ \wedge\ y < z \ \wedge\ z < x)$$

qui exprime la propriété de transitivité de l'ordre $<$.

$$2.\ \neg(x \leq y \ \wedge\ y < x)$$

qui exprime la relation qui existe entre les prédicats d'ordre $<$ et $\leq$ sur l'ensemble des entiers naturels, ou encore

$$3.\ \neg(\mathit{true} = \mathit{false})$$

laquelle, contenue dans l'ensemble d'axiomes d'une spécification SP, traduit la consistance de SP par rapport aux booléens.

Certaines de ces assertions peuvent être codées par des règles de réécriture. L'assertion (2) est par exemple traduite par la règle inconditionnelle $x \leq y \ \wedge\ y < x \rightarrow \mathit{false}$. D'autres par contre, ne peuvent pas être utilisées comme des règles de réécriture. L'assertion (1) exprimant la transitivité du préordre $<$ en est un exemple. En effet, (1) est traduite par l'équation

$$(x < y = \mathit{true}) \ \wedge\ (y < z = \mathit{true}) = (x < z = \mathit{true})$$

et elle viole la condition sur les variables puisque la variable y n'apparaît pas dans la conséquence.

2. **les assertions de recouvrement :** elles sont de la forme

$$p_1 \vee \ldots \vee\ p_n$$

et elles permettent d'effectuer une analyse par cas sur les paires critiques. Une assertion de recouvrement est également nécessaire pour prouver qu'un ensemble de formes normales contextuelles est complet. Supposons que le système de réécriture R contienne les règles

$$x \geq y \Rightarrow \mathit{max}(x,\ y) \rightarrow x,\ x < y \Rightarrow \mathit{max}(x,\ y) \rightarrow y$$

et considérons le terme $t = \mathit{max}(\mathit{max}(x,\ y),\ z)$. L'ensemble E de formes normales contextuelles de $(\mathit{true} :: t)$ est constitué des termes contextuels suivants:

1. $x < y \ \wedge\ y \geq z :: y$
2. $x < y \ \wedge\ y < z :: z$
3. $x \geq y \ \wedge\ x \geq z :: x$
4. $x \geq y \ \wedge\ x < z :: z$

Pour toutes variables x et y, l'assertion $x \geq y \ \vee\ x < y$ garantit la bonne couverture du système R et assure de ce fait que E est complet, puisque toutes les branches de réécriture, lors du calcul de E, ont été prises en compte. Cette assertion de recouvrement est par conséquent nécessaire même si elle n'est pas utilisée dans le processus de réécriture. En

réalité, cet axiome n'est pas essentiel pour la définition des prédicats $\geq$ et $<$ sur les termes clos si le système R est complet sur l'ensemble $T(F)$. C'est la raison pour laquelle de tels axiomes peuvent être vus comme des théorèmes inductifs. L'utilisation de ces axiomes est par ailleurs très liée aux preuves de réductibilité inductive. Considérons un système de réécriture conditionnelle R constitué des règles:

$$R = \{x \leq y \Rightarrow min(x,\ y) \rightarrow x,\ y \leq x \Rightarrow min(x,\ y) \rightarrow y\}$$

Le terme $min(x,\ y)$ est inductivement réductible si la propriété $x \leq y \ \vee \ y \leq x$ est valide dans l'algèbre initiale $I(R)$.

Revenons sur nos pas et examinons l'analyse par cas des paires critiques contextuelles telle qu'elle a été effectuée dans l'exemple 5.2. Cette analyse nécessite davantage de précautions. Considérons à titre d'exemple, le système de réécriture suivant:

Exemple 5.3 *Soit* (S, F, R) *le système de réécriture conditionnelle décroissant suivant :*

$S = \{bool\} \oplus \{nat, liste\}$;

$F = F_{bool} \oplus \{0 : nat,\ succ : nat \to nat,$
$\leq, < : nat \times nat \to bool,$
$[\,] : liste,\ cons : nat \times liste \to liste,$
$ordered : liste \to liste,$
$insert : nat \times liste \to liste\}.$

L'ensemble des constructeurs est constitué des opérateurs
$C = \{true, false, 0, succ, [\,], cons\}.$

$R = R_{bool} \oplus \{$
Axiomatisation de $\leq$
$0 \leq x \to true,\ succ(x) \leq 0 \to false,\ succ(x) \leq succ(y) \to x \leq y,$

Axiomatisation de $<$
$x < 0 \to false,\ 0 < succ(x) \to true,\ succ(x) < succ(y) \to x < y,$

Axiomatisation de *ordered*
1. $ordered([\,]) \to true,$
2. $ordered(cons(x, [\,])) \to true,$
3. $x \leq y \Rightarrow ordered(cons(x, cons(y, l))) \to ordered(cons(y, u)),$
4. $y < x \Rightarrow ordered(cons(x, cons(y, l))) \to false,$

Axiomatisation de *insert*
5. $insert(x, []) \to cons(x, [\,]),$
6. $x \leq y \Rightarrow insert(x, cons(y, l)) \to cons(x, cons(y, l)),$
7. $y < x \Rightarrow insert(x, cons(y, l)) \to cons(y, insert(x, l)),$

Relation entre les opérateurs définis :
8. $y < x \Rightarrow ordered(cons(y, insert(x, l))) \to ordered(cons(y, l))$ $\}$.

La superposition entre les règles (5) et (8) produit la paire critique :

$$(pc)\ \ y < x \Rightarrow ordered(cons(y, cons(x, [\,]))) = ordered(cons(y, [\,]))$$

Le c-terme

$$y < x \ ::\ ordered(cons(y, cons(x, [\,])))$$

est réductible par la relation $\xrightarrow{c}_R$ en utilisant les règles 3 et 4, et la réduction produit les termes contextuels suivants :

1. $y < x \ \wedge\ y \leq x \ ::\ ordered(cons(x, [\,]))$
2. $y < x \ \wedge\ x < y \ ::\ false$

Puisque l'assertion $y \leq x \ \vee\ x < y$ est logiquement équivalente à $true$, il devient possible de remplacer (pc) par l'ensemble suivant d'équations :

- $(a)\ \ y < x \ \wedge\ y \leq x \Rightarrow ordered(cons(x, [\,])) = ordered(cons(y, [\,]))$ et
- $(b)\ \ y < x \ \wedge\ x < y \Rightarrow false = ordered(cons(y, [\,]))$

Cependant, ce remplacement n'est licite que si la réduction du c-terme

$$y < x \ :: \ ordered(cons(y, \ cons(x, \ [\,])))$$

est opérationnellement complète. En d'autres termes, la validité algébrique de l'expression $y \leq x \ \vee \ x < y$ dans l'algèbre initiale ne suffit pas. Il faut également que la validité opérationnelle soit vérifiée:

$$\forall \sigma \in Subst(F), \ \sigma(y \leq x) \rightarrow_R^* true \text{ ou } \sigma(x < y) \rightarrow_R^* true$$

Si nous reprenons l'exemple, l'équation (a) est triviale puisque les deux termes de sa conséquence se réduisent tous deux en *true* par la règle 2. L'équation (b) est triviale si l'on peut prouver la validité suivante:

$$\forall \sigma \in Subst(F), \ \sigma(y < x) \ \wedge \ \sigma(x < y) \rightarrow_R^* false$$

$\diamond$

Il est par conséquent, nécessaire de définir le concept opérationnel de la validité dans l'algèbre initiale. En effet, l'équivalence logique ne suffit pas car la réduction récursive (et toutes les relations de réécriture conditionnelle qu'elle contient), est définie de telle sorte qu'une règle $p \Rightarrow g \rightarrow d$ puisse être appliquée avec une substitution σ, pas seulement si l'expression σp est satisfaite, mais peut en réalité être prouvée valide en la *réduisant* à *true*. Ces notions de validités algébrique et opérationnelle dans l'algèbre initiale $I(R)$ sont définies comme suit:

Définition 5.1 Validité d'une assertion de recouvrement dans l'algèbre initiale
$p_1 \ \vee \ldots \vee \ p_n$ *est algébriquement valide dans* $I(R)$ *et on note* $I(R) \models p_1 \ \vee \ldots \vee \ p_n$ *ssi*

$$\forall \sigma \in Subst(F), \ \exists i \in [1 \ldots n] \text{ tel que } \sigma p_i =_R true$$

La validité opérationnelle de $p_1 \ \vee \ldots \vee \ p_n$ *est traduite par*

$$\forall \sigma \in Subst(F), \ \exists i \in [1 \ldots n] \text{ tel que } \sigma p_i \rightarrow_R^* true$$

Définition 5.2 Validité d'une assertion négative dans l'algèbre initiale
$\neg(p_1 \ \wedge \ldots \wedge \ p_n)$ *est algébriquement valide dans* $I(R)$ *et on note* $I(R) \models \neg(p_1 \ \wedge \ldots \wedge \ p_n)$ *ssi*

$$\forall \sigma \in Subst(F), \ \sigma p_1 \ \wedge \ldots \wedge \ \sigma p_n =_R false$$

Similairement, $\neg(p_1 \ \wedge \ldots \wedge \ p_n)$ *est opérationnellement valide dans* $I(R)$ *ssi*

$$\forall \sigma \in Subst(F), \ \sigma p_1 \ \wedge \ldots \wedge \ \sigma p_n \rightarrow_R^* false$$

Remarque: Une assertion négative algébriquement valide est aussi opérationnellement valide.

En ce qui concerne les preuves de validité opérationnelle des assertions
$\neg(x \doteq z \ \wedge \ x < z)$
$\neg(x \doteq z \ \wedge \ z < x)$
$x < z \ \vee \ x \doteq z \ \vee \ z < x$

$\neg(y < x \ \wedge \ x < y)$, et
$y \leq x \ \vee \ x < y$,
un argument de hiérarchie, présenté par Ganzinger dans [35] peut être utilisé. Considérons à titre d'exemple, l'expression $y \leq x \ \vee \ x < y$. Cette propriété de recouvrement est opérationnellement valide dans l'algèbre initiale de la spécification SP si elle appartient à la théorie inductive d'une sous-spécification SP' confluente sur les termes clos, et à condition que SP soit une extension suffisamment complète de SP'.
Si on considère la sous-spécification des booléens avec les nombres naturels 0, *succ* et les prédicats $\leq$ et $<$, le système obtenu après complétion est confluent sur les termes clos. Il satisfait également la propriété $y \leq x \ \vee \ x < y$. De plus, la spécification des listes triées d'entiers naturels est une extension suffisamment complète de cette base. Ceci permet d'établir la validité opérationnelle de $y \leq x \ \vee \ x < y$ dans l'algèbre initiale $I(R)$ de SP.
Pour les autres exemples, le même argument peut être utilisé en définissant une sous-spécification des booléens avec les nombres naturels et les prédicats utilisés.

Il est désormais admis que le processus opérationnel de réécriture conditionnelle doit être complété par des preuves de contextes à l'aide de prouveurs du premier ordre. Un second aspect qui intervient dans ces preuves de contextes et qui influe sur leur efficacité est lié à la forme des règles conditionnelles et plus particulièrement, à la forme de leurs préconditions.

2.3 Influence de la forme des spécifications conditionnelles sur les preuves de contextes

Les preuves de contextes nécessitent l'introduction d'assertions (ou lemmes) qui peuvent apparaître au cours du processus de preuve pour simplifier cette preuve. La sensibilité de ces preuves à la forme des préconditions nous est apparue au cours des expériences que nous avons menées. En effet, considérons les deux définitions suivantes de la fonction *min* sur les entiers naturels :

$R_1 = \{x \leq y \Rightarrow min(x,\ y) \rightarrow x,\ y \leq x \Rightarrow min(x,\ y) \rightarrow y\}$ et

$R_2 = \{x \leq y = true \Rightarrow min(x,\ y) \rightarrow x,\ x \leq y = false \Rightarrow min(x,\ y) \rightarrow y\}$.

La paire critique qui résulte de la superposition des règles de R_1 est la suivante :

$$x \leq y \ \wedge \ y \leq x \Rightarrow x = y$$

Prouver que cette paire critique est triviale est malaisé. La preuve nécessite en effet, une utilisation complexe et une connaissance pointue des prédicats intervenant dans les préconditions, ainsi que des différentes relations qui peuvent exister entre ces prédicats.

Considérons à présent la paire critique issue de la superposition des règles de R_2 :

$$x \leq y = true \ \wedge \ x \leq y = false \Rightarrow x = y$$

Il est relativement aisé de montrer que cette équation est triviale. On peut citer trois méthodes permettant d'établir cette preuve. La première consiste à utiliser l'assertion négative

$$\neg(x \leq y = true \ \wedge \ x \leq y = false)$$

La seconde consiste à coder les préconditions dont on veut prouver l'insatisfiabilité par des clauses de Horn, et à utiliser un prouveur de théorèmes pour établir la preuve. C'est

la solution que nous détaillons par la suite car c'est celle que nous avons adoptée dans l'implantation de RECOND. Un troisième choix consiste à établir des critères spécifiques d'insatisfiabilité des contextes comme nous l'avons fait dans le théorème principal du chapitre 4. Dans ce cas précis, le critère d'insatisfiabilité permet de supprimer des équations dont la précondition est de la forme $p = true \ \wedge \ p = false \ \wedge \ c'$. $\diamond$

Nous avons soulevé un certain nombre de problèmes que l'on rencontre souvent tant au cours du test de confluence sur les termes clos qu'au cours de la procédure de complétion conditionnelle. Ces problèmes sont dans une large mesure, causés par la difficulté de définir précisément la sémantique sous-jacente à la théorie conditionnelle. Dans la partie qui suit, nous présentons les travaux des auteurs qui se sont intéréssés à cet aspect sémantique. Nos résultats rejoignent ces travaux pour une grande partie. Nous donnons également les motivations qui ont guidé nos différents choix dans ce travail de thèse.

3 Les différentes approches de preuve dans les contextes

Lorsque nous avons abordé l'aspect sémantique de la théorie conditionnelle, nous nous sommes préoccupé de trouver une réponse à la question suivante: quelle classe de modèles associer à la spécification d'un type abstrait qui mélange des équations conditionnelles et des axiomes du calcul des prédicats du premier ordre? Notre objectif dans ce paragraphe, est de dégager à partir des présentations ci-dessous, une solution à ce prolème, qui reste encore en grande partie ouvert à l'heure actuelle. Nous présentons uniquement les approches qui mettent explicitement en jeu le caractère opérationnel de la réécriture.

1. **Les travaux de Ganzinger [35]:** L'auteur aborde le problème en le posant d'une autre façon. Il choisit en effet, de considérer une spécification purement équationnelle conditionnelle, notée SP et qui admet par conséquent une algèbre initiale, notée $I(SP)$. Il s'agit par la suite, d'ajouter à cette spécification un ensemble Ax d'axiomes du calcul des prédicats du premier ordre valides dans $I(SP)$. Par conséquent, $SP \cup Ax$ admet également une algèbre initiale qui coïncide avec $I(SP)$. Les assertions auxiliaires de Ax sont de deux types, les assertions négatives et les assertions de recouvrement. Le choix des assertions effectivement utiles n'est cependant pas évident. Il dépend en grande partie de la forme de la spécification. Cette difficulté est illustrée par l'exemple suivant:

Exemple 5.4 *Soit $(S,\ F,\ R)$ le système de réécriture suivant:*

$S = \{bool,\ nat,\ list\}$;

$F = BF \oplus DF$;
$BF = C \oplus \{\leq : nat \times nat \rightarrow bool\}$;
$C = \{true,\ false :\rightarrow bool,$
$\quad 0 :\rightarrow nat,\ succ : nat \rightarrow nat,$
$\quad [\,] :\rightarrow list,\ cons : nat \times list \rightarrow list\}$;
$F \ominus BF = \{a,\ b,\ c :\rightarrow nat,$
$\quad ord : list \rightarrow bool,$
$\quad ins : nat \times list \rightarrow list\}$;

BR contient l'axiomatisation de $\leq$:
$BR = \{0 \leq x \rightarrow true,\ succ(x) \leq 0 \rightarrow false,\ succ(x) \leq succ(y) \rightarrow x \leq y\}$;
$R = BR \oplus \{1.\ x \leq y = true \Rightarrow ord(cons(x,\ cons(y,\ l))) \rightarrow ord(cons(y,\ l)),$
$\quad 2.\ b \leq c = true \Rightarrow ins(b,\ cons(c,\ l)) \rightarrow cons(b,\ cons(c,\ l)),$
$\quad 3.\ b \leq a = false \Rightarrow ord(cons(a,\ ins(b,\ l))) \rightarrow ord(cons(a,\ l))\}$.

Considérons la paire critique contextuelle suivante issue de la superposition des règles (2) et (3):

$$b \leq a = false\ \wedge\ b \leq c = true \Rightarrow ord(cons(a,\ cons(b,\ cons(c,\ l)))) = ord(cons(a,\ cons(c,\ l)))$$

et examinons sous quelles conditions cette paire critique est triviale.

Le terme contextuel

$$b \leq a = false\ \wedge\ b \leq c = true\ ::\ ord(cons(a,\ cons(b,\ cons(c,\ l))))$$

se réécrit en:

$$b \leq a = false\ \wedge\ b \leq c = true\ \wedge\ a \leq b = true\ ::\ ord(cons(b,\ cons(c,\ l)))$$

en utilisant la règle (1). Ceci nous amène à utiliser l'assertion de recouvrement $a \leq b = true\ \vee\ a \leq b = false$ et à engendrer les équations suivantes:

1. $b \leq a = false\ \wedge\ b \leq c = true\ \wedge\ a \leq b = true \Rightarrow ord(cons(b,\ cons(c,\ l))) = ord(cons(a,\ cons(c,\ l)))$ et
2. $b \leq a = false\ \wedge\ b \leq c = true\ \wedge\ a \leq b = false \Rightarrow ord(cons(a,\ cons(b,\ cons(c,\ l)))) = ord(cons(a,\ cons(c,\ l)))$

A condition que l'assertion négative

$$\neg(b \leq a = false\ \wedge\ a \leq b = false),$$

soit valide dans $I(R)$, il est alors possible de supprimer l'équation (2) du système. Examinons l'équation (1). De la même manière que lors de la première étape, cette équation peut être éclatée en deux nouvelles équations qui lui sont équivalentes, en opérant la règle (1) sur le premier membre:

1. $b \leq a = false\ \wedge\ b \leq c = true\ \wedge\ a \leq b = true\ \wedge\ b \leq c = true \Rightarrow ord(cons(c,\ l)) = ord(cons(a,\ cons(c,\ l)))$

2. $b \leq a = false \ \wedge \ b \leq c = true \ \wedge \ a \leq b = true \ \wedge \ b \leq c = false \Rightarrow ord(cons(b,\ cons(c,\ l))) = ord(cons(a,\ cons(c,\ l)))$

Cette fois, nous avons utilisé l'assertion de recouvrement

$$b \leq c = true \ \vee \ b \leq c = false.$$

La seconde équation que nous obtenons est triviale puisque

$$b \leq c = true \ \wedge \ b \leq c = false$$

est insatisfiable. Il est possible à présent d'opérer sur le second membre de l'équation restante et nous obtenons par conséquent les équations suivantes :

1. $b \leq a = false \ \wedge \ b \leq c = true \ \wedge \ a \leq b = true \ \wedge \ b \leq c = true \ \wedge \ a \leq c = true \Rightarrow ord(cons(c,\ l)) = ord(cons(c,\ l))$ et
2. $b \leq a = false \ \wedge \ b \leq c = true \ \wedge \ a \leq b = true \ \wedge \ b \leq c = true \ \wedge \ a \leq c = false \Rightarrow ord(cons(c,\ l)) = ord(cons(a,\ cons(c,\ l)))$

La première équation est triviale puisque les termes qui composent sa conséquence sont syntaxiquement égaux. Le contexte de la seconde équation est insatisfiable puisque l'opérateur d'ordre $\leq$ est transitif et, par conséquent, l'assertion

$$\neg(b \leq a = false \ \wedge \ b \leq c = true \ \wedge \ a \leq c = false)$$

est valide dans l'algèbre initiale $I(R)$. Résumons à présent la situation. Nous avons eu besoin des assertions suivantes pour établir que la paire critique calculée est triviale :

- $a \leq b = true \ \vee \ a \leq b = false$
- $\neg(b \leq a = false \ \wedge \ a \leq b = false)$
- $b \leq c = true \ \vee \ b \leq c = false$
- $b \leq c = true \ \wedge \ b \leq c = false$
- $a \leq c = true \ \vee \ a \leq c = false$
- $\neg(b \leq a = false \ \wedge \ b \leq c = true \ \wedge \ a \leq c = false)$

Le choix des assertions de recouvrement est dicté par la réductibilité des composantes de la paire critique. Quant aux assertions négatives, il n'est pas possible par contre d'avoir de guide et l'on ne peut pas savoir a priori quelle est l'assertion négative qui permettra d'achever la preuve. Considérons en effet l'assertion négative

$$\neg(b \leq a = false \ \wedge \ b \leq c = true \ \wedge \ a \leq c = false)$$

Elle traduit la transitivité de l'ordre $\leq$. Cependant, cette propriété de transitivité est également traduite par l'assertion

$$\neg(b \leq a = false \ \wedge \ b \leq c = true \ \wedge \ c \leq a = true).$$

ou encore par l'assertion

$$\neg(a \leq b = true \ \wedge \ b \leq c = true \ \wedge \ a \leq c = false)$$

De même, l'assertion

$$\neg(b \leq a = false \ \wedge \ a \leq b = false)$$

est équivalente à

$$\neg(b \leq a = true \ \wedge \ a \leq b = true)$$

et utiliser l'une plutôt que l'autre dépend des règles simplificatrices utilisées lors du processus de complétion. ◇

En adaptant cette approche, l'auteur montre qu'elle convient à l'étude des spécifications paramétrées. Les assertions auxiliaires constituent des contraintes paramètres utiles pour effectuer des preuves inductives dans la spécification paramétrée. Ces contraintes devront être vérifiées par le paramètre actuel lors de l'instanciation. Ces assertions sont par ailleurs, des formules arbitraires dont la justification dans la théorie inductive peut être établie indépendamment des méthodes de réécriture, en utilisant par exemple, un prouveur de théorèmes du premier ordre.

2. Les travaux de Navarro et Orejas [81] : Les auteurs proposent une approche alternative fondée sur les *Log*-modèles. Une équation conditionnelle, dans une *Log*-spécification SP, a une seule précondition de la forme

$$c \Rightarrow s = t$$

où c peut comporter des connecteurs booléens quelconques, dont la définition est également contenue dans SP. Les auteurs supposent également que les symboles de fonction à destination booléenne, autres que les connecteurs logiques, n'ont pas d'arguments booléens. Cette restriction est justifiée par leur souci d'interpréter les prédicats par des fonctions à valeurs booléennes. Un *Log*-modèle est une algèbre hétérogène satisfaisant les axiomes de la spécification et telle que sa restriction à la sorte booléenne comporte exactement deux constantes ***true*** et ***false*** qui sont les interprétations respectives dans la *Log*-algèbre, de *true* et *false*. En effet, Navarro et Orejas restreignent ***a priori*** la classe des modèles et l'existence d'une algèbre initiale n'est pas garantie. Pour cet aspect sémantique, le principal apport des auteurs se situe dans l'activité de la théorie des preuves. En effet, Navarro définit dans sa thèse, un système de déduction L, correct et complet par rapport aux *Log*-modèles [79]. Le système L est d'une extrême simplicité. Les sept premières règles du système formalisent le raisonnement équationnel. La règle

$$\frac{E \vdash_L c_1 \Rightarrow t = t', \ E \vdash_L c_2 \Rightarrow t = t'}{E \vdash_L c_1 \ \vee \ c_2 \Rightarrow t = t'}$$

est prévue pour le raisonnement par cas. Sa correction et sa complétude sont dues à la restriction du support booléen des *Log*-algèbres à l'ensemble $\{\boldsymbol{true}, \boldsymbol{false}\}$. Deux autres règles traitent les cas triviaux :

$$E \vdash_L false \Rightarrow t = t'$$

et

$$E \vdash_L c \Rightarrow c = true$$

Cependant, le système L est indécidable. L'approche des *Log*-algèbres s'appuie en effet, sur l'utilisation d'un système canonique des booléens et les problèmes de terminaison inhérents à ce type de systèmes ne sont pas abordés par les auteurs.

Les auteurs définissent un processus de réécriture contextuelle qui constitue, d'un point de vue théorique, une technique de preuve complète pour les *Log*-spécifications. Cependant, ces résultats ne peuvent pas actuellement être mis en pratique. Leur implantation

nécessite en effet la mise en œuvre de techniques efficaces pour la réalisation et l'utilisation au cours du processus de complétion, d'un outil de preuve de formules générales du premier ordre.

3. **Les travaux de Zhang et Kapur [106] :** L'objectif des auteurs est d'établir une méthode pour prouver automatiquement des théorèmes du premier ordre avec égalité. L'originalité de la méthode est qu'elle combine la superposition clausale sur des littéraux maximaux et la réduction de règles conditionnelles. Toute clause est transformée en une règle de réécriture conditionnelle de façon à définir un processus de réécriture conditionnelle plus puissant que la subsomption et la démodulation puisqu'alors, une clause quelconque peut être utilisée pour réécrire, qu'elle soit unitaire ou non. Cette traduction d'une clause en règle est effectuée en établissant un ordre sur les termes et sur les littéraux de façon à ce que le membre gauche de la règle corresponde au littéral maximal de la clause. Une clause unitaire est représentée par une règle inconditionnelle et une clause non-unitaire est représentée par une règle conditionnelle dont le littéral maximal est le membre gauche de la règle. Les auteurs définissent également le concept de superposition clausale comme une opération entre deux règles conditionnelles dont le résultat est une clause appelée *clause critique*. Cette superposition clausale est équivalente à la résolution ordonnée définie dans [46]. Les auteurs établissent une procédure similaire à la procédure de complétion de Knuth-Bendix et montrent que si l'ensemble de clauses en entrée est inconsistant, la clause vide, représentée par l'égalité $true = false$, est engendrée si la procédure ne diverge pas.

Les constantes sont utilisées avec leur signification habituelle. En d'autres termes, $\neg true = false$ et $\neg false = true$ où le symbole $\neg$ représente la négation. La formule $p = true$ est équivalente à p et la formule $p = false$ équivaut à $\neg p$. Enfin, l'égalité $true = false$ est équivalente à $false$, ce qui garantit la consistance par rapport aux booléens. Les axiomes d'égalité $x = x$ et $(x = y) \vee (y \neq x)$ sont systématiquement construits dans le système de preuve.

Contrairement aux précédents travaux dans [57, 35] et [53] dans lesquels la terminaison de la réécriture conditionnelle est assurée par l'utilisation exclusive de règles réductrices, ou encore aux travaux présentés dans [95, 11] où les auteurs définissent des spécifications hiérarchiques pour éviter d'engendrer des dérivations cycliques, Zhang et Kapur ont choisi d'adapter une troisième méthode pour garantir la propriété de nœthérianité. Le principe de la méthode consiste d'une part, pour assurer la terminaison de la dérivation inconditionnelle, à vérifier que toute clause se traduit en une règle dont le membre gauche est plus grand que le membre droit et d'autre part, à limiter la profondeur de chaque réécriture à un nombre raisonnable, pour garantir que la dérivation dans la partie conditionnelle ne boucle pas.

La méthode de preuve a été implantée par les auteurs dans le système RRL (Rewrite Rule Laboratory) et les auteurs illustrent, à l'aide d'un certain nombre d'exemples, la puissance du processus de réécriture par rapport aux méthodes classiques de paramodulation sur les clauses. Examinons l'exemple suivant dans lequel les clauses ont déjà été traduites par des règles de réécriture conditionnelle

Exemple 5.5 [106]

r_1 : $min(x,\ x) \rightarrow x$,
r_2 : $min(x,\ max(x,\ y)) \rightarrow x$,
r_3 : $min(x,\ y) = x \Rightarrow max(x,\ min(y,\ z)) \rightarrow min(y,\ max(x,\ z))$
et soit la clause suivante à prouver :

$$(**) \quad max(x,\ z) = x \ \vee \ \neg(min(x,\ y) = z)$$

Le littéral $max(x,\ z) = x$ est en premier lieu reformulé et donne lieu, par paramodulation, à une seconde clause équivalente

$$max(x,\ min(x,\ y)) = x$$

A présent, il est possible d'appliquer la règle r_3 pour simplifier la clause puisque la précondition instanciée $min(x,\ x) = x$ est satisfaite par la règle r_1. La clause devient :

$$min(x,\ max(x,\ y)) = x$$

qui à son tour devient

$$x = x$$

après simplification par la règle r_2. Cette clause étant triviale, ceci achève la preuve de la clause initiale $(**)$. $\diamond$

Les auteurs montrent pour cet exemple, que ni la démodulation ni la subsomption ne peuvent supprimer la clause $(**)$ [106].

Par rapport aux autres mécanismes de réduction étudiés par la communauté de réécriture conditionnelle, il est important de remarquer que le processus de simplification proposé par Zhang et Kapur est davantage apparenté aux travaux de Kaplan [57] et de Jouannaud et Waldmann [53] qu'aux travaux de Ganzinger [35] et aux nôtres. En effet, une simplification n'opère que si la précondition instanciée de la règle simplificatrice est prouvée satisfiable tandis que le principe de la réécriture contextuelle quant à lui diffère puisque la simplification est effectuée systématiquement en ajoutant la précondition au contexte existant. Dans ce cas, la validité de la précondition instanciée est vérifiée ultérieurement lorsque le produit de la simplification est en forme normale.

Finalement, partant d'une optique différente puisque, contrairement à nos travaux, les auteurs se placent en premier lieu dans l'approche *clauses de Horn*, ces derniers voient également la nécessité de combiner l'utilisation des clauses avec le processus de réécriture conditionnelle pour augmenter la puissance de preuve. Leur mécanisme de réécriture est simple, mais peut cependant être raffiné en intégrant par exemple du raisonnement par cas.

4. **Les travaux de Plaisted [90] :** L'auteur présente une méthode sémantique pour prouver la confluence sur les termes clos d'un système de réécriture conditionnelle ou encore pour compléter ce système afin de le rendre confluent sur les termes clos. Cette méthode sémantique nécessite plus d'interaction avec l'utilisateur que les extensions de la procédure de Knuth-Bendix au cadre conditionnel qui sont des méthodes purement syntaxiques. La particularité de la méthode de confluence sémantique proposée par l'auteur réside dans le principe de séparer la théorie du système de réécriture. Dans les approches adaptées jusqu'à présent, la théorie T et le système de réécriture R sont confondus. Plaisted quant

à lui, s'applique à obtenir des systèmes de réécriture qui sont des procédures de décision pour diverses théories du premier ordre et théories initiales. La théorie T est supposée consistante. Elle n'a cependant pas besoin d'être restreinte à une logique équationnelle ou à une logique du premier ordre. Elle peut contenir des clauses plus générales et il n'est pas non plus nécessaire qu'elle admette un modèle initial, ce qui constitue le principal avantage de la méthode. La confluence sémantique utilise la validité dans une théorie T pour montrer la confluence d'une système de réécriture. La méthode est fondée sur la notion de E-séparation de T sur un ensemble S de termes définie par $t =_E s$ ssi $t = s$ est valide dans la théorie T pour tous termes s et t de S. Pour expliciter la méthode de l'auteur, nous nous proposons d'examiner l'exemple suivant:

Exemple 5.6 *Soit* $SPEC = (S,\ F,\ E)$ *où*

$$S = \{int^+,\ bool,\ list+\};$$

$$\begin{aligned} F = \{&0,\ \infty :\rightarrow int^+,\ succ:\ int^+ \rightarrow int^+,\\ &\leq : int^+ \times int^+ \rightarrow bool,\\ &nil :\rightarrow list^+,\\ &min : list^+ \rightarrow int^+,\\ &cons : int^+ \times list^+ \rightarrow list^+\}; \end{aligned}$$

$$\begin{aligned} E = \{&min(nil) = \infty,\\ &min(cons(x,\ nil)) = x,\\ &x \leq min(l) \Rightarrow min(cons(x,\ nil)) = x,\\ &\neg(x \leq min(l) \Rightarrow min(cons(x,\ nil)) = min(l)\}. \end{aligned}$$

Dans cet exemple, la sorte int^+ *correspond à la sorte des entiers notée int à laquelle est ajoutée la constante* ∞.

Toutes les occurrences de *min* dans les conditions ont des arguments plus petits que celles dans les membres gauches. Par conséquent, pour les termes clos, les occurrences de *min* peuvent être évaluées récursivement en utilisant le même ensemble d'équations. Pour les termes clos, ceci réduit les conditions à la forme $x \leq y$ ou $\neg(x \leq y)$ où x et y sont des entiers, et qui peuvent être évaluées dans la théorie T par une procédure de décision spécialisée. ◇

L'auteur illustre la relation qui existe entre la confluence sémantique et les approches algébriques. Par exemple, vérifier que certains opérateurs, tels que l'opérateur *min* dans l'exemple, peuvent être supprimés est similaire à la propriété de complétude de définition. En effet, l'auteur propose d'effectuer une analyse par cas pour montrer que tous les termes clos de la forme $min(l)$ sont réductibles si l n'a pas d'occurrences de *min*. Une variable l quelconque est soit la constante *nil*, et dans ce cas, elle se réduit à ∞, soit de la forme $cons(x,\ m)$ et elle se réduit en x si $x \leq min(m)$ ou en $min(m)$ si $\neg(x \leq min(m))$. Il est alors possible de vérifier, en utilisant un prouveur de théorèmes d'une part, que la disjonction des deux cas $x \leq min(m)$ et $\neg(x \leq min(m))$ est vraie et d'autre part, que les deux cas sont mutuellement exclusifs ($x \leq min(m) \wedge \neg(x \leq min(m)) = false$). Il est de plus nécessaire de prouver que les termes *nil* et $cons(x,\ m)$ sont toujours différents, ce qui peut être établi en considérant la théorie initiale des structures de listes. Par conséquent, tous les termes clos de la forme $min(cons(x,l))$ sont réductibles et partant, *min* ne peut pas apparaître dans un terme en forme normale.

Dans [90], Plaisted établit une condition suffisante pour vérifier la propriété de E-séparation d'une théorie T sur un ensemble de termes. La méthode de confluence sémantique est plus

générale que les preuves de confluence qui étendent la procédure de Knuth-Bendix, mais nécessite cependant une interaction importante avec l'utilisateur. Elle est par conséquent plus difficile à mettre en œuvre. Il est par ailleurs également intéressant de remarquer le souci de l'auteur d'utiliser un prouveur de théorèmes pour effectuer des preuves de contextes. Ces preuves sont soit des preuves de bonne couverture, soit des preuves d'insatisfiabilité permettant d'établir l'exclusion mutuelle de contextes.

5. **Les travaux de Peterson** [88] : Pour manipuler des équations telles que celles qui traduisent la commutativité et qui ne peuvent donc être orientées, l'auteur introduit le concept de réduction contrainte. La contrainte est attachée à une équation quelconque et une réduction est appliquée à un terme clos seulement si la contrainte est satisfaite. Cette approche évite d'utiliser des algorithmes d'unification spéciaux tels que l'unification associative-commutative [89, 50]. Par ailleurs, par son aspect opérationnel, elle est similaire aux travaux effectués dans le cadre conditionnel, mais elle reste totalement différente au niveau sémantique pour les raisons suivantes : dans les systèmes définis par Peterson, les contraintes n'affectent jamais la validité d'une équation, mais seulement son applicabilité. De plus, les contraintes utilisées sont limitées à des conjonctions d'atomes dont le symbole de prédicat est l'opérateur $>$. Il nous a paru cependant intéressant d'évoquer cette approche parce qu'à un niveau opérationnel, l'auteur se heurte aux mêmes difficultés que celles que l'on rencontre dans le cadre conditionnel, et il propose une méthode originale. Il développe un test de complétude pour un ensemble de réductions contraintes équivalent au test de convergence des paires critiques de Knuth-Bendix [64]. Cette convergence est établie par cas, de façon similaire à nos travaux dans le chapitre 4.

L'équation $e = (s = t \text{ if } c)$ est partiellement réductible par $(\lambda \to \rho \text{ if } c_1)$ en e_1 avec le reste e_2 s'il existe une substitution σ et une occurrence u de $(s = t)$ telles que $(s = t)_{/u} = \sigma\lambda$, $(c \wedge \sigma c_1) \not\equiv \mathit{false}$,

$$e_1 = ((s = t)[u \leftarrow \sigma\rho] \text{ if } (c \wedge \sigma c_1)) \text{ et } e_2 = (s = t \text{ if } (c \wedge \neg\sigma c_1))$$

A titre d'exemple, l'ensemble suivant est un ensemble complet de réductions équivalent à la théorie associative-commutative :

$$\begin{cases} (x.y).z \to x.(y.z) \\ x.y \to y.x \text{ if } x > y \\ x.(y.z) \to y.(x.z) \text{ if } x > y \end{cases}$$

Les résultats de Peterson consistent à exhiber un ensemble complet de réductions contraintes pour des théories algébriques R. Pour prouver la complétude de ces ensembles, l'auteur vérifie que toute paire critique e est convergente sur les termes clos. Pour effectuer ce test, il s'agit de calculer en premier lieu la forme normale de la contrainte de e. L'auteur propose à cet effet un algorithme spécifique à l'utilisation exclusive du prédicat $>$ dans les contraintes. La seconde opération consiste à réduire la conséquence de e, ce qui engendre en général un arbre de réductions dont les branches dépendent de la contrainte de e et de celles des éléments de R puisque l'auteur effectue des réductions par cas. Finalement, il faut ensuite vérifier l'égalité syntaxique des deux membres de chaque équation réduite.

L'avantage de l'approche contrainte est que les contraintes permettent à toute équation d'être orientée. Par contre, elle peut échouer dans la recherche infinie d'ensembles complets de réductions contraintes. Par ailleurs, l'auteur exhibe des ensembles complets de réductions contraintes pour diverses théories équationnelles.

4 La preuve de convergence des paires critiques

Cette partie étant consacrée aux problèmes inhérents à l'utilisation des booléens au cours du test de confluence sur les termes clos, nous étudions dans ce qui suit, la règle d'inférence dans laquelle intervient cette utilisation. Nous illustrons également la nécessité d'un certain savoir-faire dans les preuves du premier ordre pour le succès du test de confluence. Cette règle d'inférence concerne la suppression des équations triviales lorsqu'il s'agit principalement de prouver que le contexte est équivalent à *false*. Nous illustrons d'une part, les diverses formulations qu'on peut donner de cette règle, produisant dans certains cas, une combinaison performante et d'autre part, les choix que nous avons adoptés pour réaliser un prototype souple d'utilisation et capable de traiter un certain nombre d'applications.

4.1 Les équations triviales

Intuitivement, une équation conditionnelle est triviale si les deux membres de sa conséquence sont identiques ou encore si sa précondition est insatisfiable. Le premier cas ne pose aucun problème puisqu'il s'agit de tester l'égalité syntaxique de deux termes de $T(F,\ X)$. Par contre, éliminer le second type d'équations triviales constitue sans nul doute, l'une des étapes les plus délicates du test de confluence ou encore du processus de complétion. Cette étape dépend en effet de différents facteurs de choix, pouvant être d'une importance cruciale pour l'efficacité de la complétion.

Supprimer une équation triviale dont la précondition est insatisfiable se traduit formellement par la règle d'inférence suivante :

$$\frac{E \cup \{p \Rightarrow g = d\},\ R}{E \cup R} \quad \text{si } \forall \sigma \in Subst(F),\ \sigma p =_R false$$

Le test $\sigma p =_R false$ étant indécidable dans sa généralité, il s'agit de mettre en œuvre

des *heuristiques* correctes permettant d'éliminer certains contextes insatisfiables. Ces heuristiques dépendent en grande partie de la classe des algèbres que l'on veut appréhender et de la forme de préconditions adaptée. Nous nous plaçons d'emblée dans la classe des *Log*-algèbres et nous nous intéressons à la *Log*-algèbre initiale. Ce choix est assez naturel parce que d'une part, l'étude des spécifications algébriques nécessite seulement de considérer l'algèbre des termes clos et d'autre part, se restreindre à un support booléen isomorphe à l'ensemble $\{true,\ false\}$ nous place correctement dans l'algèbre booléenne avec uniquement deux valeurs de vérité. A partir de là, plusieurs méthodes sont envisageables.

L'utilisation de formules du premier ordre dans toute leur généralité est, à l'heure actuelle difficile à mettre en pratique. En effet, nous ne disposons pas d'outils de preuve pour une classe de formules aussi générale. Nous dirons simplement que dans ce contexte, il est possible d'intégrer, à chaque système de règles de réécriture conditionnelle, un système de réécriture canonique pour le calcul propositionnel tels que le système de Hsiang explicité dans [43], mais ce choix s'est avéré inefficace. En effet, traiter les préconditions uniquement par le processus de réécriture est en général insuffisant pour mener à bien les preuves dans les contextes. L'exemple de REVEUR4 [10] dans lequel la nécessité d'avoir recours à des procédures ad-hoc pour exploiter les propriétés de certains prédicats telles que la

transitivité et la réflexivité de $\leq$ ou encore pour traiter le prédicat d'égalité, illustre la difficulté actuelle d'adapter un tel choix.

Une seconde solution consiste à adopter une forme équationnelle pour les préconditions de règles. Une précondition p, notée $\vec{p}$, est par conséquent de la forme:

$$p_1 = p'_1 \ \wedge \ \ldots \ \wedge \ p_n = p'_n$$

Toutefois, pour conserver un cadre d'étude booléen et pouvoir utiliser toute la puissance de la logique du premier ordre, nous avons envisagé de restreindre la sorte des termes apparaissant dans chaque équation d'une précondition à la sorte booléenne. La précondition p vérifie alors la condition suivante:

$$\forall i \in [1 \ldots n],\ p'_i \in \{true,\ false\}.$$

L'avantage de ce choix est que nous disposons pour cette forme de préconditions, d'un certain nombre d'heuristiques relativement aisées à justifier dans la classe des *Log*-algèbres et à mettre en œuvre. Ces heuristiques sont principalement de trois types:

1. Le premier type porte uniquement sur la précondition de l'équation et ne nécessite aucune connaissance concernant les propriétés que possèderaient éventuellement les prédicats utilisés:

$$\frac{E \cup \{\vec{q} \ \wedge \ a = true \ \wedge \ a = false \Rightarrow g = d\},\ R}{E,\ R}$$

$$\frac{E \cup \{\vec{q} \ \wedge \ true = false \Rightarrow g = d\},\ R}{E,\ R}$$

2. Le second type d'heuristiques ne dépend pas non plus des prédicats intervenant dans la précondition mais porte sur toute l'équation. Le cas le plus général est celui où la conclusion de l'équation est une conséquence logique de la précondition. Dans sa forme la plus simple, cette heuristique exprime l'appartenance de la conséquence à la partie précondition:

$$\frac{E \cup \{\vec{q} \ \wedge \ g = d \Rightarrow g = d\},\ R}{E,\ R}$$

3. Le troisième type d'heuristiques fait intervenir les propriétés des prédicats utilisés. A titre d'exemple, l'équation qui suit est triviale :

$$x \leq y = true \ \wedge \ y \leq z = true \ \wedge \ x \leq z = false \Rightarrow g = d$$

si les variables x, y et z sont de type *entier* et si nous prenons en compte la propriété de transitivité de la relation d'ordre $\leq$ sur les entiers.

Dans ce qui suit, nous explicitons la méthode choisie pour effectuer les preuves d'insatisfiabilité du contexte afin de supprimer les équations triviales. Par ailleurs, cette méthode est en partie implantée dans le prototype RECOND.

Pour prouver l'insatisfiablité d'un contexte, une première solution consiste à adapter la procédure de Knuth-Bendix sans échec (UKB)[1]. Cette procédure de complétion est exécutée avec en entrée, l'ensemble des équations booléennes qui forment le contexte. Si la procédure engendre l'égalité $true = false$, le contexte est insatisfiable. Le principal inconvénient de cette méthode est qu'elle procède à l'orientation des équations en règles de réécriture. Si l'un des prédicats intervenant dans le contexte est transitif, la procédure s'arrête en échec puisque l'axiome de transitivité est une équation conditionnelle non orientable. Notre choix s'est fixé de ce fait, sur une représentation des préconditions et des différentes propriétés des prédicats sous forme de clauses de Horn. Dans ce cadre de travail, plusieurs stratégies de preuve d'inconsistance ont été mises en œuvre dont[45], [46],[43],[44], [98]... Nous avons choisi d'adapter la stratégie de paramodulation ordonnée établie par Rusinowitch dans sa thèse[98].

4.2 Preuve d'insatisfiabilité du contexte

La stratégie de paramodulation ordonnée

La stratégie de paramodulation ordonnée est semi-décidable. Il s'agit par conséquent, d'étudier les conditions d'arrêt de la procédure, pour éviter des calculs infinis. Pour un contexte mis sous forme d'une clause de Horn, la stratégie s'applique à montrer que ce contexte est insatisfiable. Dans ce qui suit, nous formalisons les divers concepts;

Définition 5.3 Interprétation de Herbrand, Assignation

Soient $\beta : \mathcal{R} \rightarrow N$ une signature relationnelle, $\alpha : F \rightarrow N$ une signature fonctionnelle, et X un ensemble de variables. Une interprétation de Herbrand $\mathcal{I}$ est la donnée

- *pour chaque symbole de relation R de $\mathcal{R}$, d'une application $\mathcal{I}(R) : T(F)_{\mathcal{I}}^{\beta(R)} \rightarrow \{true,\ false\}$*
- *pour chaque symbole f de F d'arité n, d'une application $\mathcal{I}(f) : T(F)^n \rightarrow T(F)$ qui à $(h_1, \ldots, h_n)$ associe $f(h_1, \ldots, h_n)$.*

Une assignation μ de $\mathcal{I}$ est une application $\mu : X \rightarrow T(F)_{\mathcal{I}}$ que l'on étend par morphisme à $T(F, X)$ par les règles suivantes :

- $\mu(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mu(t_1, \ldots, \mu(t_n))$
- $\mu(R(t_1, \ldots, t_n)) = \mathcal{I}(R)(\mu(t_1, \ldots, \mu(t_n))$

Définition 5.4 Ensemble inconsistant de clauses

Un ensemble $C = \{c_1, \ldots, c_n\}$ de clauses est dit inconsistant s'il existe une clause c_i dans C telle que, pour toute interprétation de Herbrand $\mathcal{I}$, et toute assignation μ de $\mathcal{I}$, $\mu(c_i) = false$.

Cette définition constitue la philosophie de la stratégie par réfutation. Pour montrer qu'une formule quelconque c est insatisfiable dans un ensemble Ax d'axiomes, il s'agit de tester l'inconsistance de l'ensemble $Ax \cup Clause(c)$, i.e. d'engendrer la clause vide par réfutation. $Clause(c)$ est l'ensemble de clauses de Horn qui traduisent la formule c. Dans ce qui suit, nous présentons la méthode choisie pour tester l'insatisfiablité des contextes et par là-même, pour éliminer des équations triviales du système. Cette méthode a été implantée par une étudiante au cours de son stage d'Etudes Approfondies, dans un prototype expérimental baptisé RECOND [42].

Codage des formules

Il s'agit dans un premier temps de coder, par des clauses de Horn, l'ensemble Ax d'axiomes traduisant les diverses propriétés des prédicats, l'ensemble des règles de réécriture conditionnelle ainsi que la précondition p dont on veut tester l'insatisfiabi lité. Cette opération est fondée sur le principe suivant:

$\vec{c}$ est une CEB de la forme $\wedge_{i=1}^{n}\ \pi(x_1, \ldots,\ x_m) = c'_i$ telle que pour tout i dans $[1 \ldots n]$, $c'_i \in \{true,\ false\}$ et où $\pi(x_1, \ldots,\ x_m)$ est un littéral quelconque. On peut alors traduire $\vec{c}$ sous forme de clauses de Horn en procédant de la manière suivante:

$$c = \wedge_{i=1}^{n}\ \epsilon_i\ \pi(x_1, \ldots,\ x_m)$$

où ϵ_i est soit le symbole vide si $c'_i = true$, soit le symbole de négation si $c'_i = false$.
La partie précondition étant implicitement close existentiellement par rapport aux variables contenues dans c, nous pouvons écrire c comme suit:

$$c = \exists x_1,\ \ldots,\ \exists x_m\ \wedge_{i=1}^{n}\ \epsilon_i\ \pi(x_1, \ldots,\ x_m)$$

En appliquant une skolémisation à la formule c, on obtient l'ensemble de clauses $Clause(c)$ suivant:

$$\{\epsilon_i\ \pi_i(\bar{x}_1,\ \ldots,\ \bar{x}_m),\ i \in [1 \ldots n]\}$$

Cette opération est de la même manière effectuée pour l'ensemble des axiomes Ax, pour les règles de R, et pour la précondition p. L'ensemble donné en entrée à la stratégie de réfutation est:

$$Clause(Ax) \cup Clause(R) \cup Clause(p)$$

Les propriétés de l'ensemble Ax doivent être valides dans l'algèbre initiale $I(R)$. De plus, l'ensemble des équations contenues dans R est supposé être complètement défini indépendamment des axiomes de Ax.

Un ordre de comparaison sur les clauses

La stratégie de paramodulation ordonnée est fondée sur le principe que l'application des règles d'inférence doit toujours concerner les littéraux maximaux de la (ou des) clause(s) parente(s). Il est de ce fait utile d'établir un ordre sur les clauses. L'idée d'ordonner les littéraux d'une clause a été en premier lieu utilisée dans [68, 12, 70], [69, 31] et plus récemment dans [46]. Boyer définit un ordre en assignant des poids arbitraires aux littéraux sans variables, et Lankford et Ballantyne étendent cette méthode de la résolution à la paramodulation. Fribourg combine la résolution close avec une forme restreinte de paramodulation dans laquelle la paramodulation est exécutée seulement sur le plus grand argument du prédicat d'égalité. Hsiang et Rusinowitch définissent la résolution ordonnée comme une résolution binaire où les deux littéraux unifiés sont maximaux. Ils raffinent également la paramodulation restreinte définie par Fribourg en définissant la paramodulation ordonnée, exécutée sur le plus grand argument dans l'atome égalitaire seulement si cet atome est maximal dans la clause.

L'ordre sur les littéraux est défini à partir d'un ordre de simplification $>$, complet sur les termes de $T(F,\ X)$ de la façon suivante:

- $P(t_1, \ldots,\ t_n) > Q(t'_1, \ldots,\ t'_m)$ ssi P a une précédence supérieure à celle de Q.
- $P(t_1, \ldots,\ t_n) > P(t'_1, \ldots,\ t'_n)$ ssi $t_1 \ldots t_n >_{lex} t'_1 \ldots t'_n$.

- $t_1 = t_2 > t'_1 = t'_2$ ssi $\{t_1,\ t_2\} \gg \{t'_1,\ t'_2\}$.

où $>_{lex}$ est l'extension lexicographique de $>$ et $\gg$ son extension aux multi-ensembles.

Règles d'inférence de la stratégie de paramodulation ordonnée

Soit $>$ un ordre de simplification complet. On définit l'ordre $\not\preceq$ sur les clauses comme suit :

$$C \not\preceq C' \text{ ssi } C > C' \text{ ou } C \text{ et } C' \text{ sont incomparables.}$$

L'expression $\sigma = mgu(t,\ t')$ signifie que la substitution σ est l'unificateur le plus général des termes t et t'. En d'autres termes, d'une part, $\sigma t = \sigma t'$ et d'autre part, pour toute autre substitution τ telle que $\tau t = \tau t'$, il existe une substitution ρ telle que $\rho\sigma = \tau$.

Les règles d'inférence de la paramodulation ordonnée sont les suivantes [98] :
1. La résolution ordonnée :

$$\frac{L_1 \vee D_1 \qquad \neg L_2 \vee D_2}{\sigma(D_1 \vee D_2)} \quad si \begin{cases} \sigma = mgu(L_1,\ L_2) \\ \sigma L_1 \not\preceq \sigma A,\ \forall A \in D_1 \\ \sigma L_2 \not\preceq \sigma A,\ \forall A \in D_2 \end{cases}$$

2. La paramodulation ordonnée :

$$\frac{(s = t) \vee D_1 \qquad L[a] \vee D_2}{\sigma(D_1 \vee L[t] \vee D_2)} \quad si \begin{cases} \sigma = mgu(s,\ a) \\ \sigma s \not\preceq \sigma t \\ \sigma(s = t) \not\preceq \sigma A,\ \forall A \in D_1 \\ \sigma L \not\preceq \sigma A,\ \forall A \in D_2 \\ a \textit{ est un sous terme non variable de } L \end{cases}$$

3. La factorisation ordonnée :

$$\frac{L_1 \vee L_2 \vee D_1}{\sigma(L_1 \vee D_1)} \quad si \begin{cases} \sigma = mgu(L_1,\ L_2) \\ \sigma L_1 \not\preceq \sigma A,\ \forall A \in D_1 \end{cases}$$

4. La réfutation réflexive :

$$\frac{s \neq t \vee D}{\sigma D} \quad si \begin{cases} \sigma = mgu(s,\ t) \end{cases}$$

Les deux dernières règles d'inférence servent à simplifier les clauses. En particulier, la règle de réfutation réflexive évite d'ajouter la clause $(x = x)$ à l'ensemble initial de clauses.

Le contrôle du prouveur

Etant donnés un ensemble S de clauses, et deux clauses C_1 et C_2, le contrôle instauré consiste à privilégier l'utilisation de la règle de résolution ordonnée sur les clauses C_1 et C_2, puis en cas d'échec de cette règle, à appliquer la paramodulation ordonnée sur ces mêmes clauses. Dans ces deux cas, la clause résultante subit séquentiellement les règles de factorisation ordonnée et de réfutation réflexive pour être simplifiée. Soit Inf l'ensemble des règles d'inférence présentées ci-dessus ; dans [98], l'auteur prouve en utilisant les arbres sémantiques transfinis, que Inf forme une stratégie correcte et complète pour le calcul du premier ordre :

Théorème 5.1 [98]
Soit S un ensemble de clauses de Horn et soit S^* la clôture de S par l'application des règles de Inf.
Si S est un ensemble inconsistant de clauses, S^* contient la clause vide $\square$.

Nous pouvons alors résumer le comportement du prouveur de la façon suivante

1. Si chaque clause de S contient un littéral négatif, le prouveur s'arrête avec S consistant.
2. Autrement, si un cycle est détecté lors de l'application des règles de Inf, i.e. si toute clause engendrée existe déjà dans S, S est consistant et le prouveur s'arrête.
3. Si la clause vide $\square$ est engendrée, le prouveur s'arrête et S est inconsistant.
4. Si aucun des cas précédents n'est vérifié, le processus de preuve diverge. Il faut de ce fait provoquer l'arrêt du prouveur, et il n'est par conséquent pas possible de démontrer l'inconsistance de la clause candidate. L'équation conditionnelle dont la précondition correspond à cette clause est par conséquent gardée dans le système.

L'arrêt du prouveur est provoqué lorsque le nombre de clauses contenues dans S^* est supérieur à une certaine valeur K fixée au départ. Le choix de K est délicat et l'expérience avec RECOND a montré qu'il est tout à fait aléatoire.
L'ordre implanté dans RECOND pour comparer les clauses est défini à partir de l'ordre récursif $>$ sur les chemins avec un statut gauche-droite (RPO with status [16, 18]).

Exemple 5.7 *Soit la précondition p suivante :*

$$(a = b) = true \ \wedge \ (c = b) = true \ \wedge \ (a = c) = false$$

Nous nous proposons de prouver l'insatisfiabilité de p en utilisant les axiomes de transitivité et de symétrie du prédicat $=$. Ces axiomes sont respectivement traduits par les clauses suivantes qui constituent l'ensemble $Clause(\mathcal{A}x)$:

$c_1 \quad \neg(x = y) \ \vee \ \neg(y = z) \ \vee \ (x = z)$
$c_2 \quad \neg(x = y) \ \vee \ (y = x)$

La traduction de p par des clauses de Horn produit l'ensemble $Clause(p)$ suivant :

$c_3 \quad a = b$
$c_4 \quad c = b$
$c_5 \quad \neg(a = c)$

La procédure de preuve prend en entrée l'ensemble $S = Clause(R) \cup Clause(p) \cup Clause(\mathcal{A}x)$ et se déroule comme suit :

c_6	$\neg(b = z) \ \vee \ (a = z)$	résolution ordonnée entre c_1 et c_3
c_7	$\neg(b = c)$	résolution ordonnée entre c_5 et c_6
c_8	$\neg(c = b)$	résolution ordonnée entre c_2 et c_7
	$\square$	résolution ordonnée entre c_4 et c_8 $\diamond$

Cet exemple a été expérimenté dans RECOND et donne des résultats significatifs quant à l'importance que peut prendre le choix de la variable K. En effet, lorsqu'aucune précédence

n'est fixée entre les constantes a, b et c, i.e. si a, b et c sont incomparables, le processus de preuve engendre 62 clauses pour montrer que l'ensemble S est inconsistant. Par contre, si la précédence est définie de telle façon que $b > a > c$, le processus engendre seulement 11 clauses pour mener à bien la preuve d'inconsistance. Il serait utile à l'avenir d'établir des heuristiques pour aider l'utilisateur à déterminer la valeur de K.

Il est important de remarquer que cette stratégie de réfutation peut également être utilisée pour la seconde règle d'élimination des équations triviales, à savoir

$$\frac{E \cup \{\vec{q} \wedge g = d \Rightarrow g = d\},\ R}{E,\ R}$$

Il suffit de traduire la négation de toute l'expression $\vec{q} \wedge g = d \Rightarrow g = d$, soit l'ensemble $Clause(p)$ résultant, et de tester l'inconsistance de l'ensemble $Clause(\mathcal{A}x) \cup Clause(R) \cup Clause(p)$.

4.3 RECOND

RECOND est un prototype du raisonnement par cas réalisé au laboratoire par une étudiante qui a préparé son Diplôme d'Etudes Approfondies sous la direction de J.L. Rémy [42]. Ce logiciel est écrit en CAML version 2.6.1 [28, 29, 27, 30]. Il s'agit d'un prototype simplifié car il ne comporte pas les règles d'inférence de la simplification d'une règle de réécriture. Le prototype est cependant assez souple pour permettre l'intégration de ces règles d'inférence qui n'ont pas été prises en compte dans un premier temps.

Le théorème 4.1 constitue la base théorique du travail de réalisation.

Une précondition de règle est une conjonction d'égalités booléennes de la forme $\wedge_{i=1}^{n} t_i = t'_i$ avec $t'_i \in \{true,\ false\}$ pour tout i dans $[1 \ldots n]$. De plus, membre gauche et membre droit ne doivent pas contenir de connecteur logique en tête. Un test explicite est effectué concernant l'appartenance des variables de la précondition et du membre droit au membre gauche de la règle.

Explicitement, les règles d'inférence de la complétion implantées sont les règles OR (orientation d'une règle), ACE (ajout d'une conséquence équationnelle), SET (simplification d'une équation triviale), $SCQE$ et $SCDE$ qui correspondent respectivement à la simplification de la conséquence et de la précondition d'une équation.

La procédure de complétion choisie est celle de la S-complétion et elle est implantée en CAML en termes de *règles d'inférence* + *contrôle*. Ce type d'implantation utilise un ensemble de *règles de transition* qui sont les opérations de base de la complétion, une *structure de données*, souvent appelée *univers*, sur laquelle opèrent les règles de transition, un *contrôle* décrivant la façon dont les règles de transition sont invoquées et un *jeu d'outils*, partagé par les diverses procédures de complétion. Cette description produit des programmes élégants lorsqu'elle est utilisée comme une méthode de programmation. Le contrôle instauré dans RECOND est celui proposé par Lescanne dans le projet ORME [75].

La stratégie de simplification a été choisie de telle manière à favoriser la simplification de la conséquence d'une équation par rapport à celle de sa précondition lorsque les deux types de simplification sont simultanément possibles. D'autre part, l'implantation actuelle privilégie l'invocation du prouveur par rapport à la simplification par l'ensemble des règles du système.

L'originalité de ce prototype par rapport à REVEUR4 est l'implantation du raison-

nement par cas. En effet, le processus de simplification engendre systématiquement la branche complémentaire après l'utilisation d'une seule règle conditionnelle.

Des règles de mise en forme canonique du contexte ont été implantées dans des modules de transformation.

La mise en œuvre de la règle d'élimination des équations triviales comporte le test d'insatisfiablité du contexte. Ce problème étant indécidable, RECOND met en jeu des heuristiques qui sont principalement de trois types :

1. le premier type d'heuristiques concerne les cas triviaux :

$$\vec{c} \wedge true = false,\ \vec{c} \wedge p = true \wedge p = false$$

2. le second type correspond aux équations dans lesquelles la conséquence est une partie de la précondition.

3. le dernier type est plus complexe et met en jeu des axiomes du premier ordre qui traduisent certaines propriétés relatives aux prédicats utilisés.

La stratégie de preuve implantée est la stratégie de paramodulation orientée, formalisée par Rusinowitch dans sa thèse [98]. La stratégie est correcte et complète. Elle est cependant semi-décidable et il faut alors prévoir un test d'arrêt du prouveur, test sans lequel le processus de preuve engendre des clauses indéfiniment.

Les règles d'inférence du prouveur sont les suivantes : la résolution ordonnée, la paramodulation ordonnée, la factorisation ordonnée et la réfutation réflexive.

Pour tester l'insatisfiabilité d'un contexte $\vec{c}$ dans un ensemble d'axiomes Ax, le processus implanté traduit dans un premier temps $\vec{c}$ sous forme de clauses. Soit cl la clause correspondant à cette traduction ; par des applications successives des règles d'inférence du prouveur sur l'ensemble $Ax \cup \{cl\}$, le prouveur s'applique à montrer que cet ensemble est inconsistant, i.e. à engendrer la clause vide par réfutation.

L'utilisation de cette méthode de preuve est correcte à condition que les propriétés de Ax soient valides dans l'algèbre initiale. Ce test de validité n'est pas effectué dans l'implantation actuelle. Il faut de plus que l'ensemble des équations soit complètement défini indépendamment des propriétés qui caractérisent les symboles de prédicats dans Ax.

L'arrêt du prouveur est contrôlé en fixant une valeur maximale au nombre de clauses engendrées au cours du déroulement du processus de preuve.

Le problème de divergence de la branche complémentaire est traité de la façon suivante : chaque fois qu'un terme est réécrit par une règle conditionnelle, le triplet (précondition, membre gauche, occurrence de simplification) est mémorisé, car il caractérise une simplification de façon unique. Ce triplet accompagne alors le terme contextuel issu de la branche complémentaire. Une réécriture du même terme, utilisant ce même triplet, est par la suite interdite.

Divers exemples ont été déroulés sur REVEUR4 et RECOND en vue d'une comparaison des deux prototypes. Nous pouvons faire les remarques suivantes :

- Dans RECOND, les membres gauches des règles obtenues par complétion sont contextuellement irréductibles, ce qui n'est pas toujours le cas dans REVEUR4. Il

est donc plus facile d'obtenir des systèmes inter-réduits par RECOND que par REVEUR4.

- Le temps d'exécution peut avantager REVEUR4 par rapport à RECOND. Cependant, la théorie de preuve est plus fiable dans RECOND puisque celui-ci utilise un prouveur de théorèmes fondé sur une stratégie correcte et complète, tandis que REVEUR4 intègre des procédures ad-hoc pour effectuer des preuves de contextes.

RECOND est à l'heure actuelle un prototype portable, souple d'utilisation et dans lequel il est possible d'intégrer de nouvelles règles d'inférence, aussi bien pour le processus de complétion que pour le prouveur de théorèmes. Certaines optimisations sont également envisageables pour le rendre plus performant. Il est par conséquent intéressant, à l'avenir, de le développer davantage.

Il est important de noter que RECOND réalise actuellement deux fonctions principales: il effectue un test de confluence sur les termes clos, et il peut également procéder à la complétion d'un système de réécriture conditionnelle décroissant. Cependant, d'un point de vue théorique, nous ne savons à l'heure actuelle justifier nos expérimentations que sous les conditions suivantes:

- si RECOND montre qu'un système de réécriture conditionnelle est confluent sur les termes clos sans avoir ajouté de nouvelles équations au système, i.e. s'il effectue simplement un test de confluence, et si au cours de ce test, il a prouvé la convergence des paires critiques calculées en utilisant uniquement les quatre critères du théorème 4.1 (voir chapitre 4), alors on peut affirmer que le système est confluent sur les termes clos et que la spécification correspondante est consistante par rapport aux booléens.

- si par contre, RECOND a complété le système de réécriture par l'ajout de nouvelles règles, et si pour prouver la convergence des paires critiques contextuelles, il a uniquement utilisé les quatre critères de convergence du théorème 4.1 au cours de la complétion, nous ne pouvons rien dire à propos du système initial d'équations. Par contre, le système final lui, est confluent sur les termes clos. Il suffit de transformer les règles obtenues en équations, et d'effectuer à nouveau le test de confluence sur ces équations pour nous retrouver dans les conditions d'applicabilité du théorème 4.1.

- si le test de confluence a fait appel au prouveur de théorèmes avec lequel RECOND interagit, que RECOND ait ou non ajouté de nouvelles équations au système, nous ne savons pas encore à l'heure actuelle, justifier cet aspect de nos travaux.

Tableau comparatif des prototypes

	REVEUR4	RECOND
Fonctionnalités :	*preuve de confluence sur les termes clos *preuve de consistance *complétion d'un système de réécriture hiérarchique	*preuve de confluence sur les termes clos *preuve de consistance
Forme des préconditions :	formule du premier ordre $p = \wedge_{i=1}^{n} \epsilon_i \ \pi_i(t_1, \ldots, t_n)$ ϵ_i est soit le mot vide, soit le symbole de négation $\neg$	conjonction d'équations booléennes $p = \wedge_{i=1}^{n} \pi_i(t_1, \ldots, t_m) = p'_i$ $p'_i \in \{true, false\}, \forall\ i \in [1 \ldots n]$
Preuves dans les contextes :	preuves d'insatisfiabilité preuves de bonne couverture	preuves d'insatisfiabilité
Système de réécriture utilisé :	$R = R_{SP} \cup R_{bool}$ R_{bool} représenté par R_{Hsiang} canonique pour le calcul propositionnel	$R = R_{SP}$ modulo les axiomes de mise en forme canonique
Stratégie de preuve :	réécriture de p par $\rightarrow_{H,R}$ et utilisation de procédures ad-hoc	traduction de p par un ensemble de clauses de Horn $Clause(p)$ et stratégie de réfutation en utilisant $Clause(Ax) \cup Clause(R) \cup Clause(p)$
Stratégie de simplification :	calcul d'ensembles complets de formes normales contextuelles	simplification pas à pas par application des règles d'inférence

	REVEUR4	RECOND
Preuve de convergence :	test d'équivalence des ECFNC	vérification des 4 critères du théorème principal
Raisonnement par cas :	hypothèse de bonne couverture de R	prise en compte de la branche complémentaire de réécriture
Terminaison :	R hiérarchique	R réducteur
Complétude suffisante :	hiérarchique	non-hiérarchique

Nous achevons enfin ce chapitre par une série d'exemples qui illustrent la sensibilité du test de confluence et du processus de complétion à la forme des spécifications conditionnelles. En particulier, la propriété de complétude de définition d'un opérateur a également son importance pour obtenir un bon comportement du test de confluence.

5 L'influence de la complétude de définition sur le test de confluence

Exemple 5.8 *Les exemples décrivent le déroulement de RECOND. Ils sont inspirés des travaux de Plaisted[90] et ils spécifient les listes d'entiers munies d'un opérateur min qui calcule le plus petit élément d'une liste :*

$SP = (S,\ F,\ E)$ *avec*
$S = \{bool,\ ent,\ list\}$;

$C = \{true,\ false :\rightarrow bool,$
$\quad 0 :\rightarrow ent,\ succ : ent \rightarrow ent,$
$\quad nil :\rightarrow list,\ cons : ent \times list \rightarrow list\}$;
$BF \ominus C = \{\leq : ent \times ent \rightarrow bool\}$;
$F = BF \oplus \{min : list \rightarrow ent\}$.

$BE = \{0 \leq x = true,\ succ(x) \leq 0 = false,\ succ(x) \leq succ(y) = x \leq y\},$
$E = BE \oplus \{1.\ min(cons(x,\ nil)) = x,$
$\quad 2.\ x \leq min(l) = true \Rightarrow min(cons(x,\ l)) = x,$
$\quad 3.\ x \leq min(l) = false \Rightarrow min(cons(x,\ l)) = min(l)\}.$

Le test de confluence effectue les étapes suivantes :
1. Orientation de (1).
2. Orientation de (2).
3. Simplification de (3) par (2) :

$$\frac{x \leq min(l) = false \Rightarrow min(cons(x,l)) = min(l)}{x \leq min(l){=}false \wedge x \leq min(l){=}true \Rightarrow x{=}min(l), x \leq min(l){=}false \Rightarrow min(cons(x,l)){=}min(l)}$$

La première équation obtenue est triviale puisque son contexte est insatisfiable (il contient une expression de la forme $p = true \wedge p = false$). La seconde équation n'est autre que l'équation (3).

4. Orientation de (3).

A cette étape, les règles du système sont les suivantes :

$$\begin{cases} 1.\ min(cons(x,\ nil)) \rightarrow true \\ 2.\ x \leq min(l) = true \Rightarrow min(cons(x,\ l)) \rightarrow x \\ 3.\ x \leq min(l) = false \Rightarrow min(cons(x,\ l)) \rightarrow min(l) \end{cases}$$

5. Calcul des paires critiques :
5.1 Calcul de la paire critique entre (1) et (2) :

$$(a) \quad x \leq min(nil) = true \Rightarrow x = x$$

5.2 Calcul de la paire critique entre (1) et (3):

$$(b) \quad x \leq min(nil) = false \Rightarrow min(nil) = x$$

5.3 Calcul de la paire critique entre (2) et (3):

$$(c) \quad (x \leq min(l) = true) \ \wedge \ (x \leq min(l) = false) \Rightarrow min(l) = x$$

(a) est triviale puisque les deux membres de sa conséquence sont identiques.
(c) est triviale puisque son contexte est insatisfiable.
(b) n'est ni triviale ni convergente. Le test de confluence conclut alors que le système n'est pas confluent sur les termes clos car aucune autre règle de R ne peut simplifier la paire critique (b) et aucune nouvelle paire critique ne peut être calculée. $\diamond$

Remarque : L'opérateur *min* n'est pas complètement défini par rapport aux constructeurs ***nil*** et *cons* de la sorte *list*. En effet, le terme $min(nil)$ n'est réductible par aucune règle de R et il n'est par conséquent équivalent à aucun terme clos primitif.

Exemple 5.9 *Reprenons l'exemple précédent en considérant la sorte des entiers munie d'une constante ∞. Cette nouvelle sorte est notée ent^+. L'ensemble F des opérateurs devient $F' = F \cup \{\infty : \rightarrow ent^+\}$, et l'ensemble E d'équations est enrichi par l'ajout de l'axiome :*

$$E' = E \cup \{4. \ \ min(nil) = \infty\}.$$

Examinons de nouveau le comportement du test de confluence sur les termes clos ; celui-ci effectue les étapes suivantes :

1. Orientation de (4).
2. Orientation de (1).
3. Orientation de (2).
4. Simplification de (3) par (2):

$$\frac{x \leq min(l) = false \Rightarrow min(cons(x,l)) = min(l)}{x \leq min(l){=}false \wedge x \leq min(l){=}true \Rightarrow x{=}min(l), x \leq min(l){=}false \Rightarrow min(cons(x,l)){=}min(l)}$$

La première équation est triviale et la seconde est exactement l'équation (3).

5. Orientation de (3).
6. Calcul des paires critiques :
6.1 Calcul de la paire critique entre (1) et (2):

$$(a) \quad x \leq min(nil) = true \Rightarrow x = x$$

6.2 Calcul de la paire critique entre (1) et (3):

$$(b) \quad x \leq min(nil) = false \Rightarrow min(nil) = x$$

6.3 Calcul de la paire critique entre (2) et (3):

$$(c) \quad (x \leq min(l) = true) \ \wedge \ (x \leq min(l) = false) \Rightarrow min(l) = x$$

(a) est triviale puisque les deux membres de sa conséquence sont identiques.
(c) est triviale puisque son contexte est insatisfiable.
(b) est simplifiable par (4):

$$\frac{x \leq min(nil) = false \Rightarrow min(nil) = x}{x \leq \infty = false \Rightarrow \infty = x}$$

De même que pour l'exemple précédent, l'équation (d) obtenue n'est ni triviale, ni convergente. De plus, aucune nouvelle paire critique ne pouvant être calculée, le processus se termine sans pouvoir éliminer (d). Ce résultat n'est pas surprenant puisquà présent, l'opérateur $\leq$ n'est pas complètement défini par rapport à la sorte ent^+. $\diamond$

Si on ajoute au système la règle inconditionnelle

$$x \leq \infty \rightarrow true$$

le processus est alors capable de supprimer (d) et il s'arrête avec succès, en montrant que le système ainsi obtenu:

$$\begin{cases} (a) & x \leq \infty \rightarrow true \\ (b) & min(nil) \rightarrow \infty \\ (c) & min(cons(x,\ nil)) \rightarrow x \\ (d) & x \leq min(l) = true \Rightarrow min(cons(x,\ l)) \rightarrow x \\ (e) & x \leq min(l) = false \Rightarrow min(cons(x,\ l)) \rightarrow min(l) \end{cases}$$

est confluent sur les termes clos. $\diamond$

Remarque : La définition de l'opérateur *min* est complète et redondante. En effet, pour une instanciation close (e) de la variable x, le terme $min(cons(e,\ nil))$ est réductible par les règles (c) et (d) puisqu'alors, la validité de la précondition $e \leq min(nil)$ est vérifiée en la réduisant en *true* par la règle $x \leq \infty \rightarrow true$. C'est cette redondance dans la définition qui pose justement un problème puisque la paire critique (b) est issue de la superposition de (1) (l'équation redondante) et de (3).

Exemple 5.10 *Reprenons encore l'exemple précédent en supprimant cette fois l'équation (1) i.e. en considérant le système d'équations suivant:*

$$E = BE \oplus \{1.\ min(nil) = \infty,$$
$$2.\ x \leq min(l) = true \Rightarrow min(cons(x,\ l)) = x,$$
$$3.\ x \leq min(l) = false \Rightarrow min(cons(x,\ l)) = min(l)\}.$$

Le processus de complétion effectue les étapes suivantes :
1. Orientation de (1).
2. Orientation de (2).
3. Simplification de (3) par (2):

$$(x \leq min(l) = true) \ \wedge \ (x \leq min(l) = false) \Rightarrow min(l) = x$$

Cette équation est triviale.

4. Orientation de (3).
5. Calcul des paires critiques:
5.1 Calcul de la paire critique entre (2) et (3):

$$(x \leq min(l) = true) \ \wedge \ (x \leq min(l) = false) \Rightarrow min(l) = x$$

Cette paire critique est triviale puisque son contexte est insatisfiable. Aucune nouvelle paire critique n'est calculable et aucune autre simplification ne peut être effectuée. Le processus s'arrête avec succès en produisant un système de règles confluent sur les termes clos. ◇

Remarque : La définition de l'opérateur *min* est complète et non redondante.

6 Conclusion

Nous avons voulu mettre en évidence dans ce chapitre, les difficultés que soulève l'implantation de la procédure de complétion conditionnelle, particulièrement en ce qui concerne le traitement des préconditions. Nous proposons également des solutions à ces problèmes. L'existence des préconditions nécessite notamment de compléter le caractère opérationnel de la réécriture par un traitement logique des contextes. Aborder à l'heure actuelle ce traitement en adoptant une forme générale de préconditions est malaisé. Notre sentiment est qu'il est plus intéressant dans un premier temps, de restreindre cette forme afin d'obtenir des résultats plus approfondis.

La nécessité d'établir des preuves dans les contextes appraît en particulier lorsqu'il s'agit de prouver la convergence de paires critiques contextuelles ou de calculer les formes normales contextuelles d'un terme. Ces preuves sont principalement de deux types, les preuves de bonne couverture et les preuves d'insatisfiabilité. Intégrer le raisonnement par cas dans le processus de réécriture conditionnelle permet d'éviter le premier type de preuves. Seules les preuves d'insatisfiabilité sont encore nécessaires. Une autre amélioration de ce processus de réécriture consiste à utiliser l'information contenue dans le contexte pour réécrire plutôt que d'effectuer systématiquement une analyse par cas. Cette amélioration a l'avantage de diminuer le nombre de preuves d'insatisfiabilité à établir.

Le problème crucial qu'il a fallu par la suite résoudre a été d'adapter une méthode de preuve dans les contextes. Une approche intéressante, établie par Ganzinger [35], consiste à introduire des assertions auxiliaires dans la partie paramètre formel de la spécification conditionnelle et de les utiliser pour effectuer les preuves dans les contextes. Cependant, comme nous l'avons illustré par un exemple, le choix de ces lemmes additionnels n'est pas évident et nécessite une connaissance avisée du déroulement de la procédure de complétion. Nous avons choisi pour notre part, d'intégrer un prouveur de théorèmes du premier ordre, indépendent de la procédure de complétion et qui peut être utilisé au besoin. Ce prouveur peut de ce fait servir à d'autres buts. L'implantation actuelle du prototype est par ailleurs suffisamment souple pour être enrichie par de nouvelles règles d'inférence de stratégie par réfutation. Il est cependant important de rappeler que le comportement de RECOND est justifié par nos résultats théoriques du chapitre 4 d'une part, seulement lorsqu'il teste la confluence sur les termes clos d'un système de réécriture conditionnelle, et d'autre part, en effectuant des preuves de convergence uniquement fondées sur les quatre critères du théorème 4.1. Il est cependant utile pour nos expérimentations et nous suggère principalement deux voies de recherche. La première concerne l'étude de la complétion conditionnelle fondée sur le raisonnement par cas, et la seconde la justification de l'interaction d'un prototype tel que RECOND avec un prouveur de théorèmes du premier ordre.

CONCLUSION

Conclusion

Le travail de cette thèse se situe dans le cadre des spécifications conditionnelles de types abstraits algébriques. Ce type de spécifications est utile dans la plupart des applications informatiques parmi lesquelles nous pouvons citer les langages de programmation qui utilisent des instructions telles que *if-then-else* ou *if-then* ou encore les systèmes de déduction qui considèrent des règles d'inférence avec préconditions représentant des actions contrôlées par des conditions. Dans les applications de contrôle de processus, il est nécessaire d'établir des tables de décision régies elles aussi par des commandes conditionnelles. Enfin, dans la théorie des types abstraits algébriques, l'étude des opérations partiellement définies et des preuves d'implantation nécessitent aussi de prouver des assertions conditionnelles... Cela nous a conduite à mettre l'accent sur la théorie conditionnelle et plus particulièrement, sur les systèmes de réécriture opérationnels dans cette théorie, qui constituent un outil performant pour étudier les types abstraits algébriques et effectuer des preuves conditionnelles.

Une large proportion de travaux a été consacrée à l'étude du raisonnement équationnel et des systèmes de réécriture de termes classiques, sans préconditions. Ces travaux ont donné lieu à des applications intéressantes dans divers domaines. Comparativement, peu de recherches ont été menées dans le cadre conditionnel. De plus, il est malaisé de comparer les différentes approches adaptées par les auteurs car celles-ci dépendent du choix d'un certain nombre de critères permettant d'appréhender une classe spécifique de modèles.

Nous nous sommes intéressée, dans ce travail, à deux catégories de systèmes de réécriture conditionnelle, les systèmes hiérarchiques et les systèmes décroissants, et nous avons étudié un certain nombre de leurs propriétés en considérant pour chacune d'elles, l'algèbre initiale. Ces deux catégories de systèmes ont été définies dans le but déviter la génération de dérivations infinies. Nous avons consacré une partie de notre recherche aux systèmes de réécriture hiérarchique avec des règles dont les préconditions sont des formules propositionnelles arbitraires. Un système de réécriture hiérarchique quelconque peut être vu comme une extension d'un système de base dans lequel les prédicats doivent être complètement définis. L'idée, formalisée par Rémy [94], a émergé à cause de la nécessité de définir totalement les préconditions des opérations qui servent de prémisses aux assertions que l'on prouver. Nous avons abordé l'étude hiérarchique en considérant des algèbres avec booléens dont le principal intérêt est d'autoriser l'utilisation d'une nouvelle règle d'inférence qui traduit le raisonnemment par cas. L'approche hiérarchique présente l'avantage de faciliter la définition de spécifications conditionnelles volumineuses et permet d'établir une méthode effective pour calculer les formes normales contextuelles d'un terme. La hiérarchie que nous avons adaptée porte aussi bien sur la signature de la spécification que sur la définition de la relation de réécriture. Il est cependant possible de définir différents degrés de hiérarchie pour les deux concepts. Certaines de ces définitions sont

plus souples que celle que nous avons choisie et il conviendrait de les étudier à l'avenir. Pour cette classe de systèmes hiérarchiques, nous avons établi des résultats de confluence sur les termes clos et des résultats de consistance relative de la spécification conditionnelle associée qui permettent alors d'effectuer des preuves inductives. Ces preuves sont aussi connues sous le nom de *preuves par consistance.*

Pour mettre au point nos résultats concernant les systèmes de réécriture hiérarchique, nous avons été amenée par ailleurs, à étudier quelques propriétés, et plus généralement à donner pour chacune d'elles, des conditions suffisantes de décidabilité. Une propriété essentielle à laquelle nous avons consacré le chapitre 2, est la complétude suffisante des spécifications conditionnelles pour laquelle nous établissons un algorithme de test fondé sur le raisonnement par cas. L'usage combiné des motifs structurels et des préconditions rend le problème du test plus complexe mais permet de prendre en compte des définitions impossibles à traiter uniquement avec des motifs structurels. Il n'est pas possible de proposer une réponse universelle, le problème étant indécidable dans sa pleine généralité, et il s'agit donc de considérer des sous-classes de définitions pour lesquelles le problème admet des solutions. Nous proposons également dans ce chapitre, des extensions et des améliorations de notre travail, aussi bien dans le cadre général, sans hypothèse de hiérarchie, que dans un cadre hiérarchique.

Une étape cruciale dans la preuve de confluence sur les termes clos des systèmes hiérarchiques consiste à montrer l'équivalence de contextes dans le système de base. Ce point intervient lors de la comparaison de deux ensembles de formes normales contextuelles. Une première solution, adaptée par Zhang dans sa thèse [105], a été d'utiliser pour ces preuves, le système de réécriture canonique proposé par Hsiang [43] pour le calcul propositionnel. Cette solution a donné lieu à la réalisation du prototype REVEUR4 [95]. Pour mener à bien ces preuves de contextes, l'idée est d'introduire des axiomes sur les prédicats tels que des propriétés de transitivité, d'antisymétrie. Cependant, le système de Hsiang s'étant révélé insuffisant dans ce cadre, nous avons par la suite, envisagé d'utiliser une technique de réfutation et de l'intégrer au test de confluence. Cette technique consiste à engendrer l'équation $true = false$ en utilisant un algorithme de complétion tel que l'algorithme formalisé par Rusinowitch fondé sur la paramodulation orientée [98].

Nous avons choisi d'implanter cette seconde solution dans les systèmes de réécriture conditionnelle décroissants. Un système décroissant se caractérise par des règles dont le membre droit et la précondition sont plus petites que le membre gauche correspondant selon un ordre spécifique sur les termes. D'autre part, dans le souci de faciliter les preuves de contextes, nous avons également choisi d'adapter une forme équationnelle pour les préconditions de règles. Dans ce cadre, nous avons formalisé une méthode de réécriture conditionnelle fondée sur le raisonnement par cas qui s'est révélée être un outil puissant pour établir les preuves de confluence sur les termes clos des systèmes décroissants. Ces preuves de confluence se basent sur des critères de convergence des paires critiques contextuelles simples à vérifier. Un prototype expérimental mettant en œuvre le processus de réécriture étudié a été réalisé au sein du laboratoire. Il intègre les règles d'inférence de simplification des équations et contient, à côté du module de gestion de ces règles, un prouveur de calcul des prédicats du premier ordre utilisable, si nécessaire, pour d'autres usages. Ce prototype est constitué de modules aussi simples que possible de manière à pouvoir l'étendre et le modifier à l'avenir, en fonction de nos résultats théoriques.

Par ailleurs, étant donné que nous avons principalement considéré l'algèbre des ter-

mes clos, nous nous sommes préoccupée tout au long de cette thèse, du comportement de cette algèbre par rapport à la sorte booléenne. Nous avons essentiellement illustré dans le dernier chapitre, la nécessité d'établir une interaction entre les systèmes de réécriture avec leur aspect algébrique, et les preuves du premier ordre. Le raisonnement logico-algébrique consiste à faire coopérer les règles d'inférence de la logique du premier ordre et celles de l'algèbre universelle, les premières étant orientées vers la déduction et les secondes vers la simplification. Cependant, le traitement préconisé est délicat car les diverses expériences menées sur les spécifications algébriques conditionnelles pour valider des propriétés de celles-ci, ont mis en évidence l'extrême sensibilité des preuves à la forme des-dites spécifications. Typiquement, de telles preuves nécessitent l'introduction de lemmes et, en fonction de la forme initiale de la spécification, ces lemmes pourront ou non être d'une grande utilité au cours du processus de preuve. Câbler un prouveur de théorèmes du premier ordre tels que le prouveur de Stickel [99], ou de Plaisted [92] permet de développer des méthodes effectives pour valider les spécifications algébriques et ceci est d'une importance cruciale pour l'avènement de systèmes sûrs de construction de programmes. Un effort doit par conséquent être fait pour l'automatisation de ces preuves.

Une étude ayant pour objet l'application des résultats obtenus dans ce travail, notamment ceux concernant l'utilisation des méthodes de réécriture contextuelle, aux spécifications paramétrées serait intéressante à mener. Cette étude présenterait bien sûr un degré de difficulté supplémentaire par rapport au cadre non paramétré.

Nous avons soulevé divers problèmes dans les conclusions de chaque chapitre de ce document et nous espérons que des travaux futurs viendront continuer et étendre nos résultats.

NOTATIONS

Notations

Σ	signature d'algèbre	page 11
S	ensemble de sortes	page 11
F	ensemble de symboles de fonction ou alphabet	page 11
$T(F)$	algèbre des termes clos	page 12
X	ensemble de variables	page 12
$dom(t)$	domaine de définition du terme t	page 13
ϵ	le mot vide	page 13
$T(F,\ X)$	algèbre libre engendrée par X ou ensemble des termes avec variables	page 13
$t_{/u}$	sous-terme de t à l'occurrence u	page 13
$Var(t)$	ensemble de variables du terme t	page 13
$Dom(\sigma)$	domaine de la substitution σ	page 14
$Subst(F,\ X)$	ensemble des substitutions définies sur l'ensemble X des variables	page 14
$Subst(F)$	ensemble des substitutions closes	page 14
$\models$	validité dans une algèbre	page 15
E	ensemble d'équations (conditionnelles)	page 15
$=_E$	E-égalité ou théorie équationnelle engendrée par E	page 15
R	système de réécriture (conditionnelle)	page 16
$\rightarrow_R$	réduction récursive	page 16
$\rightarrow_R^*$	fermeture réflexive et transitive de $\rightarrow_R$	page 16
$\bar{t}$	forme normale du terme t	page 16
$>$	ordre de comparaison de termes	page 17
$\longleftrightarrow_R^*$	fermeture réflexive, symétrique et transitive de $\rightarrow_R$	page 18
$\oplus$	union disjointe d'ensembles	page 19
$\ominus$	différence ensembliste	page 19
C	ensemble des constructeurs	page 19
$T(C,\ X)$	ensemble des termes primitifs	page 19
$T(C)$	ensemble des termes clos primitifs	page 19
CE	ensemble des relations entre les constructeurs	page 21
DE	définitions des opérateurs de la spécification	page 21

Alg_{SP}	classe de tous les modèles satisfaisant les équations de SP	page 24
Gen_{SP}	classe de toutes les algèbres finiment engendrées de SP	page 24
$Congr_F$	treillis complet des congruences sur $T(F,\ X)$	page 25
SP_{bool}	spécification des booléens	page 26
$LogAlg_{SP}$	classe des *Log*-algèbres	page 27
A_{bool}	support booléen de l'algèbre A	page 27
L	système de déduction défini dans $LogAlg_{SP}$	page 28
$\vdash_L$	validité dans L	page 28
T_{SP}	objet initial de la catégorie Alg_{SP} des algèbres	page 29
$\succ$	extension d'un ordre de réduction $>$ possédant la relation de sous-terme strict	page 31
CEB	conjonction d'équations booléennes	page 44
$\vec{p}$	l'expression booléenne p est une CEB	page 44
TT	abréviation de $true = true,\ false = false$	page 44
FF	abréviation de $true = false,\ false = true$	page 44
BF	ensemble des opérateurs de base	page 45
$T(BF,\ X)$	ensemble des termes de base	page 45
$T(BF)$	ensemble des termes clos de base	page 45
$DF,\ F \ominus BF$	ensemble des opérateurs définis	page 45
$BE,\ BR$	resp ensemble d'équations et de règles de réécriture de base	page 46
C_{bool}	constructeurs de sorte booléenne	page 47
At	ensemble d'atomes booléens	page 49
$(\vec{c} :: t)$	terme contextuel ou $\vec{c}$ est une CEB	page 50
$\xrightarrow{c}_R$	relation de réécriture contextuelle	page 50
Sh_s	schéma structurel de sorte s	page 51
t^c_u	motif du terme t à l'occurrence u	page 52
$Mot(t,\ u)$	ensemble des motifs de t à l'occurrence u	page 52
$Occ(f)$	ensemble des occurrences définies de f	page 52
$\gg$	extension aux multi-ensembles de $>$	page 57
$\succ\!\!\!\succ$	extension aux multi-ensembles de $\succ$	page 57

R_{inc}	partie inconditionnelle d'un système conditionnel	page 71
$prof(S)$	profondeur d'un ensemble S de clauses de Horn	page 79
$TS(A)$	ensemble test de l'atome A	page 79
$SRCH$	système de réécriture conditionnelle hiérarchique	page 90
$\rightarrow_{H,R}$	relation de réécriture hiérarchique	page 91
$\rightarrow^*_{H,R}$	fermeture réflexive et transitive de $\rightarrow_{H,R}$	page 91
$\equiv_{H,SP}$	congruence hiérarchique engendrée par une spécification conditionnelle hiérarchique	page 92
$\longleftrightarrow^*_{H,R}$	fermeture réflexive, symétrique et transitive de $\rightarrow_{H,R}$	page 94
$\downarrow_R$	exprime la réductibilité dans R de deux termes en un même	page 96
$(c :: t)$	terme contextuel ou c-terme	page 97
$\xrightarrow{c}_{H,R}$	relation de réécriture contextuelle hiérarchique	page 97
$\xrightarrow{c}_{H,R,t}$	réécriture contextuelle hiérarchique dans la partie terme	page 97
$\xrightarrow{c}_{H,R,c}$	réécriture contextuelle hiérarchique dans la partie contexte	page 98
$EFNC$	ensemble de formes normales contextuelles	page 99
$\succ_c$	extension de $\succ$ aux termes contextuels	page 107
$I(R)$	algèbre initiale de R	page 109
BH	système booléen de Hsiang	page 114
$\Re(t)$	ensemble des redex du terme t	page 133
$SCQE$	simplification d'une équation dans sa partie terme	page 137
$SCDE$	simplification d'une équation dans sa précondition	page 137
OR	orientation d'une règle	page 138
ACE	ajout d'une conséquence équationnelle	page 138
SET	suppression d'une équation triviale	page 139
$EFCS$	ensemble de formes contextuellement simplifiées	page 141
$\mathcal{I}$	interprétation de Herbrand	page 189
$>_{lex}$	extension lexicographique de $>$	page 190

INDEX

Index

BIBLIOGRAPHIE

Bibliographie

[1] S. Anantharaman, J. Hsiang, and J. Mzali. Sbreve2- a term rewriting laboratory with (ac-)unfailing completion. In N. Dershowitz, editor, *Proc. of Rewriting Techniques and Applications*, pages 533–537. Springer-Verlag, 1989.

[2] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for Equational Proofs. In *DC, IEEE Computer Society*, pages 346–357, Cambridge, Massachusetts USA, 1986. 1st Symposium on Logic in Computer Science.

[3] H. Bertling, H. Ganzinger, and R. Schäfers. CEC: A System for Conditional Equational Completion. user manual (version 1.4). Technical Report PROSPECTRA report, Universität Dortmund, West Germany, 1988.

[4] M. Bidoit and M. Choquer. Preuve de Formules Conditionnelles par Récurrence. In *Lecture Notes in Computer Science*, volume 1, pages 329–344, Grenoble, France, 1985. 5th of AFCET Conference Reconnaissance des Formes et Intelligence Artificielle, Springer-Verlag.

[5] G. Birkhoff. On the Structure of Abstract Algebras. In *Proceedings Cambridge Phil. Soc. 31*, pages 433–454, 1935.

[6] S. Bloom and R. Tindell. Varieties of "IF-THEN-ELSE". *S.I.A.M. J. on Computing*, 12(4):677–707, 1983.

[7] A. Bouhoula. Implantation d'un algorithme de test de complétude suffisante d'une spécification conditionnelle et d'aide à la construction de définitions structurelles complètes. Rapport de Stage de Fin d'Etudes, Centre de Recherche en Informatique de Nancy, 1990.

[8] W. Bousdira. Case Reasoning in Completion of Conditional Rewriting Systems. 89-r-015, Centre de Recherche en Informatique de Nancy, 1989.

[9] W. Bousdira and J-L. Rémy. Complétion des Systèmes de Réécriture Conditionnelle. *revue BIGRE+GLOBULE*, 55, 1987.

[10] W. Bousdira and J-L. Rémy. REVEUR4 : A Laboratory for Conditional Rewriting. In F.J. Branderburg, G. Vidal-Naquet, and M. Wirsing, editors, *Lecture Notes in Computer Science*, pages 472–473, Passau RFA, 1987. 4th Symposium on Theoretical Aspects of Computer Science, Springer-Verlag.

[11] W. Bousdira and J-L. Rémy. Hierarchical Contextual Rewriting with Several Levels. In R. Cori and M. Wirsing, editors, *Lecture Notes in Computer Science*, volume 294, pages 193–206, Bordeaux, France, 1988. 5th Synposium on Theoretical Aspects of Computer Science, Springer-Verlag.

[12] R.S. Boyer. Locking: A restriction of resolution, 1971. Ph.D Thesis.

[13] B. Brand, J.A. Darringer, and W.H. Joyner. Completeness of Conditional Reductions. *Proceedings Symposium on Automatic Deduction*, 1979.

[14] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. HOPE : An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, 1980.

[15] G. Cousineau, L. Paulson, G. Huet, R. Milner, M. Gordon, and C. Wadsworth. *The ML Handbook.* INRIA, Rocquencourt, May 1985.

[16] N. Dershowitz. Orderings for Term-Rewriting Systems. *Theoretical Computer Science*, 17:279–301, 1982.

[17] N. Dershowitz. Computing with Term Rewriting Systems. In *Proceedings of An NSF Workshop On The Rewrite Rule Laboratory*, 1984.

[18] N. Dershowitz. Termination. In J-P. Jouannaud, editor, *Lecture Notes in Computer Science*, volume 202, Dijon, France, 1985. 1st Conference on Rewriting Techniques and Applications, Springer-Verlag. Also in Journal os jsc 3, special issue on Rewriting Techniques and Applications, 69-115, 1987.

[19] N. Dershowitz and M. Okada. Conditional Equational Programming and the Theory of Conditional Term Rewriting. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 337–346, Tokyo, Japan, November 1988. ICOT.

[20] N. Dershowitz and M. Okada. Proof-Theoretic Techniques and the Theory of Rewriting. In *Proceedings of the third Symposium on Logic in Computer Science*, pages 104–111, Edinburgh, Scotland, July 1988. IEEE.

[21] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical Conditional Rewrite Systems. In *Ninth Conference on Automated Deduction*, pages 538–549, Argonne, May 1988. Available as Vol. 310, lncs, Springer, Berlin.

[22] N. Dershowitz and D. Plaisted. Logic Programming cum Applicative Programming. In *Proceedings of the IEEE Symposium on Logic Programming*, 1985.

[23] K. Drosten. Toward Executable Specifications using Conditional Axioms. Technical Report A.C.M 29, Institut fur Informatik, Braunschweig, 1982.

[24] H. Ehrig, H. Kreowsky, and P. Padawitz. Stepwise Specifications and Implementations of ADT's. In *Lecture Notes in Computer Science*, volume 62, pages 205–226. Springer-Verlag, 1978.

[25] R. Forgaard and D. Detlefs. An Incremental Algorithm for Proving Termination of Term Rewriting Systems. In J-P. Jouannaud, editor, *Proceedings 1st International Conference on Rewriting Techniques and Applications*, pages 255–270. Springer-Verlag, 1985.

[26] R. Forgaard and J. Guttag. REVE : A Term Rewriting System Generator with failure-resistant Knuth-Bendix. Technical report, MIT-LCS, 1984.

[27] Projet FORMEL. The CAML Primer. Technical report, INRIA LIENS, 1987.

[28] Projet FORMEL. CAML: the reference manuel. Technical report, INRIA-ENS, March 1987.

[29] Projet FORMEL. *The CAML Anthology*, July 1987. Internal Document.

[30] Projet FORMEL. *The CAML Reference Manual*, March 1989. Version 2.6.

[31] L. Fribourg. A superposition oriented theorem prover. *Theoretical Computer Science*, 35:129–164, 1985.

[32] L. Fribourg. A strong restriction of the inductive completion procedure. In *Lecture Notes in Computer Science*, pages 105–115. Proceedings 13th International Colloquium on Automata, Languages and Programming, Springer-Verlag, 1986.

[33] K. Futatsugi, J. Goguen, J-P. Jouannaud, and J. Meseguer. Principles of OBJ-2. In B. Reid, editor, *Proceedings 12th ACM Symp. on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[34] H. Ganzinger. A Completion Procedure for Conditional Equations. In S. Kaplan and J-P. Jouannaud, editors, *Lecture Notes in Computer Science*, volume 308, pages 62–83, Orsay, France, 1987. 1st International Workshop on Conditional Term Rewriting Systems, Springer-Verlag. Submitted to jsc, special issue on Theorem Proving.

[35] H. Ganzinger. Ground Term Confluence in Parametric Conditional Equational Specifications. In F.J. Brandenburg, G. Vidal, and M. Wirsing, editors, *Lecture Notes in Computer Science*, volume 247, pages 286–298, Passau RFA, 1987. 4th annual Symposium on Theoretial Aspects of Computer Science, Springer-Verlag.

[36] R. Goebel. Rewrite Rules with Conditions for Algebraic Specifications. In M. Broy and M. Wirsing, editors, *Proceedings of 2nd Workshop on Theory and Applications of Abstract Data Types*, Passau, West Germany, 1983.

[37] J. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, J. Meseguer, and T. Winkler. An Introduction to OBJ-3. In J-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems*, volume 308. Springer-Verlag, June 1988. Also as internal report CRIN: 88-R-001.

[38] J.A. Goguen and J. Meseguer. Completeness of Many-Sorted Equational Logic. Technical Report CSLI-84-15, Center for the Study of Language and Information Stanford University, 1984.

[39] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology IV : Data Structuring*, volume 4, pages 80–144. Englewood Cliffs, Prentice Hall, 1978.

[40] I. Guessarian and J. Meseguer. On the axiomatization of "if-then-else". *SIAM journal of Computing*, 16:332–357, 1986.

[41] J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.

[42] A. Habacha Hamada. Implantation d'un Algorithme de Complétion Conditionnelle basé sur le Raisonnement par Cas. Rapport interne CRIN 90-R-014, Centre de Recherche en Informatique de Nancy, 1990.

[43] J. Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25(1):255–300, 1985.

[44] J. Hsiang. Rewrite methods for theorem provig in first order theory with equality. *Journal of Symbolic Computation*, 3:133–151, 1987.

[45] J. Hsiang and N. Dershowitz. Rewrite Methods for Clausal and Non-clausal Theorem Proving. In *Proceedings of 10th International Colloquium on Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 331–346, Barcelona, Spain, 1983. Springer-Verlag.

[46] J. Hsiang and M. Rusinowitch. A new method for establishing refutational completeness in theorem proving. In J. Siekmann, editor, *Proceedings 8th Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 141–152. Springer-Verlag, 1986.

[47] G. Huet. A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm. *Journal of Computer Systems and Sciences*, 23:11–21, 1981.

[48] G. Huet and J-M. Hullot. Proofs by Induction in Equational Theories with Constructors. *Journal of Computer and System Sciences*, 25(2):239–266, October 1982. Preliminary version in Proceedings 21st Symposium on Foundations of Computer Science, IEEE, 1980.

[49] Hussmann. Unification in Conditional Equational Theories. In B.F. Caviness, editor, *Lecture Notes in Computer Science*. European Computer Algebra Conference, Springer-Verlag, 1985.

[50] J-P. Jouannaud and H. Kirchner. Completion of a Set of Rules modulo a Set of Equations. *Proceedings 11th ACM Symposium on Principles Of Programming Languages*, 1984.

[51] J-P. Jouannaud and E. Kounalis. Automatic Proofs by Induction in Theories without Constructors. Technical Report 295, Université de Paris-Sud, Centre d'Orsay, September 1986.

[52] J-P. Jouannaud and E. Kounalis. Proof by induction in Equational Theories without Constructors. In *Proceedings 1st Symposium on Logic in Computer Science*, pages 358–366, Boston, USA, 1986.

[53] J-P. Jouannaud and B. Waldmann. Reductive Conditional Term Rewriting Systems. In M. Wirsing, editor, *Elsevier Science Publishers*, pages 223–244, Ebberup, Denmark, 1986. 3rd IFIP Working Conference on Formal Description of Programming Concepts.

[54] S. Kaplan. Conditional Rewrite Rules. *Theoretical Computer Science*, 3:175–193, 1984.

[55] S. Kaplan. Fair Conditional Term Rewriting Systems : Unification, Termination and Confluence. Technical Report 194, University of Paris Sud, Orsay, 1984.

[56] S. Kaplan. Positive/Negative Conditional Rewriting. In R. Cori and M. Wirsing, editors, *Lecture Notes in Computer Science*, pages 129–143, Orsay, France, 1987. 1st International Workshop on Conditional Term Rewriting Systems, Springer-Verlag.

[57] S. Kaplan. Simplifying Conditional Term Rewriting Systems : Unification, Termination and Confluence. *Journal of Symbolic Computation*, 1987.

[58] S. Kaplan and J-L. Rémy. Completion Algorithms for Conditional Rewriting Systems. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, Austin, Texas, 1987. MCC-INRIA Colloquium, Academic Press, INC.

[59] D. Kapur, P. Narendran, and H. Zhang. Proof by induction using test sets. In *Proceedings of 8th International Conference on Automated Deduction*, pages 99–117. Springer-Verlag, 1986. Lecture Notes in Computer Science, volume 230.

[60] D. Kapur, P. Narendran, and H. Zhang. On Sufficient Completeness and Related Properties of Term Rewriting Systems. *Acta Informatica*, 24:395–415, 1987.

[61] C. Kirchner and H. Kirchner. Constrained Equational Reasoning. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 382–389. ACM Press, July 1989.

[62] H. Kirchner. A General Inductive Algorithm and Application to Abstract Data Types. In R. Shostak, editor, *Proceedings 7th International Conference on Automated Deduction*, pages 282–302. Springer-Verlag, Lecture Notes in Computer Science, 1984.

[63] H. Kirchner. Preuves par Complétion dans les Variétés d'Algèbres, 1985. Thèse de Doctorat d'Etat.

[64] D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[65] E. Kounalis. Completeness in Data Type Specifications. In B. Buchberger, editor, *Proceedings EUROCAL Conference*, volume 204 of *Lecture Notes in Computer Science*, Linz (Austria), 1985. Springer-Verlag.

[66] E. Kounalis. Validation des Spécifications Algébriques par Complétion Inductive, 1985. Thèse de l'Université de Nancy 1.

[67] E. Kounalis and M. Rusinowitch. On Word Problems in Horn Logic. In E. Lusk and R. Overbeek, editors, *Lecture Notes in Computer Science*, volume 310, Argonne, Illinois, 1988. 9th International Conference on Automated Deduction, Springer-Verlag.

[68] R. Kowalski and J-P. Hayes. Semantic trees in automatic theorem proving. *Machine Intelligence*, 5, 1969.

[69] D. Lankford and A.M. Ballantyne. The refutational completeness of blocked permutative narrowing and resolution. In *4th Conference on Automated Deduction*, Austin, Texas, 1979.

[70] D.S. Lankford. Canonical Inference. Technical report, Louisiana Tech. University, 1975.

[71] A. Lazrek. Etude et Réalisation de Méthodes de Preuve par Récurrence en Logique Equationnelle, décembre 1988. Thèse de l'Institut National Polytechnique de Lorraine.

[72] A. Lazrek, P. Lescanne, and J-J. Thiel. Tools for Proving Inductive Equalities, Relative Completeness and ω-Completeness. Technical Report 88-R-131, Centre de Recherche en Informatique de Nancy, 1988. to be published in Information and Computation.

[73] P. Lescanne. Some Properties of the Decomposition Ordering, a Simplification Ordering to Prove Termination of Rewriting Systems. *RAIRO Informatique Théorique / Theoretical Informatics*, 16(4):331–347, 1982.

[74] P. Lescanne. Computer Experiments with the REVE Term Rewriting Systems Generator. In *Proceedings, 10th ACM Symposium on Principles of Programming Languages*, pages 99–108. Association for Computing Machinery, 1983.

[75] P. Lescanne. Completion Procedures as Transition Rules + Control. In M. Diaz and F. Orejas, editors, *TAPSOFT'89*, volume 351, pages 28–41. Springer-Verlag, Lecture Notes in Computer Science, 1989.

[76] Z. Manna and S. Ness. On the Termination of Markov algorithms. In *Proceedings of the Third Hawaii International Cobference on System Science*, pages 789–792, Honolulu, Hawaii, 1970.

[77] R. Milner. A proposal for standard ML. In *Proceedings ACM Conference on LISP and Functional Programming*, 1984.

[78] C.K. Mohan and M.K. Srivas. Function Definitions in Term Rewriting and Applicative Programming. *Information and Control*, 3:186–217, 1986.

[79] M. Navarro. Técnicas de Reescritura para Especificaciones Condicionales, 1987.

[80] M. Navarro and F. Orejas. On the Equivalence of Hierarchical and Non-hierarchical Rewriting on Conditional Term Rewriting Systems. In J. Fitch, editor, *Lecture Notes in Computer Science*, volume 174, Oxford, 1984. Eurosam Conference, Springer-Verlag.

[81] M. Navarro and F. Orejas. Parameterized Horn Clause Specifications : Proof Theory and Correctness. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Lecture Notes in Computer Science*, volume 249, pages 202–216, Pisa Italy, 1984. TAPSOFT'87, Springer-Verlag.

[82] M.H.A. Newman. On Theories with a Combinatorial Definition of Equivalence. In *Annals of Math.*, volume 43, pages 223–243, 1942.

[83] T. Nipkow and G. Weikum. A Decidability Result about Sufficient Completeness of Axiomatically Specified Abstract Data Types. In *6th GI Conference*, volume 145, pages 257–268. Springer-Verlag, Lecture Notes in Computer Science, 1983.

[84] P. Padawitz. New Results on Completeness and Consistency of Abstract Data Types. In *Lecture Notes in Computer Science*, volume 88, pages 460–473. Proceedings 9th Symposium on Mathematical Foundations of Computer Science, Springer-Verlag, 1980.

[85] P. Padawitz. Correctness, Completeness and Consistency of Equational Data Type Specifications. Technical Report Bericht Nr 83-15, Technische Universitat, Berlin, West Germany, 1983.

[86] P. Padawitz. *Parameter Passing Data Type Specifications*, volume 1 of *Lecture Notes in Computer Science*, pages 323–341. Springer-Verlag, Berlin, 1985.

[87] P. Padawitz. *Computing in Horn Clause Theories.* Springer-Verlag, 1988.

[88] G.E. Peterson. Complete Sets of Reductions with Constraints. In *Proceedings 10th International Conference on Automated Deduction*, Kaiserslautern, RFA, 1990. Springer-Verlag. Lecture Notes in Computer Science.

[89] G.E. Peterson and M. Stickel. Complete Sets of Reductions for Some Equational Theories. *Journal of the Association for Computing Machinery*, 28:233–264, 1981.

[90] D. Plaisted. Semantic Confluence Tests and Completion Methods. In *Journal Information and Control 65*, pages 182–215, 1985.

[91] D.A. Plaisted. A Logic for Conditional Rewrite Rules. In S. Kaplan and J-P. Jouannaud, editors, *Lecture Notes in Computer Science*, volume 308, pages 62–83, Orsay, France, 1987. 1st International Workshop on Conditional Term Rewriting Systems, Springer-Verlag.

[92] D.A. Plaisted. Non-Horn Clause Logic Programming without Contrapositives. *Journal of Automated Reasoning*, 4(3):287–325, 1988.

[93] U. Pletat, G. Engels, and H.D. Ehrig. Operational Semantics of Algebraic Specifications with Conditional Equations. In A. Arnold and M. Dauchet, editors, *Lecture Notes in Computer Science*, Lille, France, 1982. 7th CAAP Conference, Springer-Verlag.

[94] J-L. Rémy. Etude des Systèmes de Réécriture Conditionnelle et Applications aux Types Abstraits Algébriques, 1982. Thèse d'Etat.

[95] J-L. Rémy and H. Zhang. REVEUR4 : a System for Validating Conditional Algebraic Specifications of Abstract Data Types. In *ECAI*, pages 563–572, Pisa, Italy, 1984. 6th ECAI Conference.

[96] M. Rice. The construction of Complete Minimal Set of Contextual Normal Forms. In J.A. Van Hulzen, editor, *Lecture Notes in Computer Science*, pages 255–266, London, England, 1983. Proceedings of EUROCAL'83, European Computer Algebra Conference, Springer-Verlag.

[97] M. Rusinowitch. Path of Subterm Ordring and Recursive Decomposition Ordering Revisited. In J-P. Jouannaud, editor, *Lecture Notes in Computer Science*, volume 202, pages 117–131, Dijon, France, 1985. 1st Conference on Rewriting Techniques and Applications, Springer-Verlag.

[98] M. Rusinowitch. Démonstration Automatique par des Techniques de Réécriture, 1987. Thèse d'Etat.

[99] M. Stickel. A Prolog Technology Theorem Prover : Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

[100] J.W. Thatcher and E.G. Wagner. Specification of Abstract Data Types using Conditional Axioms. *IBM T.J.Watson Res Center Rep RC-6214*, 1976.

[101] J.W. Thatcher and E.G. Wagner. Data Type Specification : Parametrization and the Power of Specification Techniques. In *ACM TOPLAS 4,4*, pages 711–732, 1982.

[102] J-J. Thiel. Stop loosing sleep over Incomplete Data Type Specifications. In *Proceeding 11th ACM Symp. on Principles of Programming Languages*, pages 76–82. Association for Computing Machinery, 1984.

[103] D. A. Turner. MIRANDA: A non-strict Functional Language with Polymorphic Types. In J-P. Jouannaud, editor, *Proceedings 2nd Conf. on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[104] S. Uhrig and J-L. Rémy. An Algorithm for Testing Suffisent Completeness of a Simple Class of Conditional Specifications. Technical Report 88-R-155, Centre de Recherche en Informatique de Nancy, 1988.

[105] H. Zhang. REVEUR4 : Etude et mise en œuvre de la Réécriture Conditionnelle, 1984. Thèse de 3ème cycle, Université de Nancy 1.

[106] H. Zhang and D. Kapur. First-order theorem proving using conditional rewrite rules. In E. Lusk and R. Overbeek, editors, *Proceedings 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1988.

[107] H. Zhang and J-L. Rémy. Contextual Rewriting. In J-P. Jouannaud, editor, *Lecture Notes in Computer Science*, volume 202, pages 46–62, Dijon France, 1985. 1st Conference on Rewriting Techniques and Applications, Springer-Verlag.

ANNEXE A : REVEUR4

Annexe A : REVEUR4

```
%% 1) Spe'cification des listes trie'es (voir chapitre3, page 120) %%


-> read harald2


User equations:

 1.   ((x =< y) & (y < x)) == false
 2.   ((x =< y) # (y < x)) == true
 3.   ((y < x) & (x < y)) == false
 4.   ((z < y) & ((y =< x) & (x =< z))) == false
 5.   ordered(empty_l) == true
 6.   ordered(cons(x, empty_l)) == true
 7.   (x =< y) :: ordered(cons(x, cons(y, u))) == ordered(cons(y, u))
 8.   (y < x) :: ordered(cons(x, cons(y, u))) == false
 9.   insert(x, empty_l) == cons(x, empty_l)
10.   (x =< y) :: insert(x, cons(y, u)) == cons(x, cons(y, u))
11.   (y < x) :: insert(x, cons(y, u)) == cons(y, insert(x, u))
12.   ordered(insert(x, u)) == ordered(u)

No critical pair equations.

No rewrite rules.

Note that the following identifiers were parsed as nullary operators:
     false     true      empty_l

%% Les pre'dicats =< et < sont de'clare's transitifs %%
-> transitive =< <

Among the above operators, what is reflexive one ? Please tape in them.
Operators : =<

%% On de'clare les ope'rateurs du niveau 1 de hie'rarchie %%
-> conditional insert ordered


%% On declare les constructeurs de la specification (les constantes true et
false sont implicitement les seuls constructeurs de la sorte boole'enne) %%
-> constructors cons empty_l


Precedence extended.

%% La comple'tion est ensuite de'clenche'e %%
-> kb

Starting Knuth-Bendix...

Ordered the equation:
    ((y < x) & (x =< y)) == false
    into the rewrite rule:
    ((y < x) & (x =< y)) -> false

Ordered the equation:
    ((y < x) # (x =< y)) == true
```

```
    into the rewrite rule:
    ((y < x) # (x =< y)) -> true

Ordered the equation:
    ordered(empty_l) == true
    into the rewrite rule:
    ordered(empty_l) -> true

Ordered the equation:
    ordered(cons(x, empty_l)) == true
    into the rewrite rule:
    ordered(cons(x, empty_l)) -> true

Ordered the equation:
    (x =< y) :: ordered(cons(x, cons(y, u))) == ordered(cons(y, u))
    into the rewrite rule:
    (x =< y) :: ordered(cons(x, cons(y, u))) -> ordered(cons(y, u))

Ordered the equation:
    (y < x) :: ordered(cons(x, cons(y, u))) == false
    into the rewrite rule:
    (y < x) :: ordered(cons(x, cons(y, u))) -> false

Ordered the equation:
    insert(x, empty_l) == cons(x, empty_l)
    into the rewrite rule:
    insert(x, empty_l) -> cons(x, empty_l)

Ordered the equation:
    (x =< y) :: insert(x, cons(y, u)) == cons(x, cons(y, u))
    into the rewrite rule:
    (x =< y) :: insert(x, cons(y, u)) -> cons(x, cons(y, u))

Ordered the equation:
    (y < x) :: insert(x, cons(y, u)) == cons(y, insert(x, u))
    into the rewrite rule:
    (y < x) :: insert(x, cons(y, u)) -> cons(y, insert(x, u))

Ordered the equation:
    ordered(insert(x, u)) == ordered(u)
    into the rewrite rule:
    ordered(insert(x, u)) -> ordered(u)

There are currently 10 rules and 0 equations in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    ordered(insert(x, u)) -> ordered(u)
    and the rule:
    insert(x, empty_l) -> cons(x, empty_l)
        are as follows:
        ordered(cons(x, empty_l)) == ordered(empty_l)
        (they are added to the system as kb-equations.)

There are currently 10 rules and 1 equation in the system.


Starting to reduce and order equations...
```

```
There are currently 10 rules and 0 equations in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    (x =< y) :: insert(x, cons(y, u)) -> cons(x, cons(y, u))
    and the rule:
    ordered(insert(x, u)) -> ordered(u)
        are as follows:
        (x =< y) :: ordered(cons(x, cons(y, u))) == ordered(cons(y, u))
        (they are added to the system as kb-equations.)

There are currently 10 rules and 1 equation in the system.


Starting to reduce and order equations...

There are currently 10 rules and 0 equations in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    (y < x) :: insert(x, cons(y, u)) -> cons(y, insert(x, u))
    and the rule:
    ordered(insert(x, u)) -> ordered(u)
        are as follows:
        (y < x) :: ordered(cons(y, insert(x, u))) == ordered(cons(y, u))
        (they are added to the system as kb-equations.)

There are currently 10 rules and 1 equation in the system.


Starting to reduce and order equations...

There are currently 10 rules and 1 equation in the system.

Ordered the equation:
    (y < x) :: ordered(cons(y, insert(x, u))) == ordered(cons(y, u))
    into the rewrite rule:
    (y < x) :: ordered(cons(y, insert(x, u))) -> ordered(cons(y, u))

There are currently 11 rules and 0 equations in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    (y < x) :: ordered(cons(y, insert(x, u))) -> ordered(cons(y, u))
    and the rule:
    insert(x, empty_l) -> cons(x, empty_l)
        are as follows:
        (y < x) :: ordered(cons(y, cons(x, empty_l))) == ordered(cons(y,
                                                                 empty_l))
        (they are added to the system as kb-equations.)
    and the rule:
    (x =< y) :: insert(x, cons(y, u)) -> cons(x, cons(y, u))
        are as follows:
```

```
        ((x =< y) & (z < x)) :: ordered(cons(z, cons(x, cons(y, u))))
                        == ordered(cons(z, cons(y, u)))
        (they are added to the system as kb-equations.)
    and the rule:
    (y < x) :: insert(x, cons(y, u)) -> cons(y, insert(x, u))
        are as follows:
        ((y < x) & (z < x)) :: ordered(cons(z, cons(y, insert(x, u))))
                        == ordered(cons(z, cons(y, u)))
        (they are added to the system as kb-equations.)

There are currently 11 rules and 3 equations in the system.


Starting to reduce and order equations...

There are currently 11 rules and 0 equations in the system.

No user equations.

No critical pair equations.

Rewrite rules:

 1.   ordered(empty_l) -> true
 2.   ordered(cons(x, empty_l)) -> true
 3.   insert(x, empty_l) -> cons(x, empty_l)
 4.   ordered(insert(x, u)) -> ordered(u)
 5.   ((y < x) & (x =< y)) -> false
 6.   ((y < x) # (x =< y)) -> true
 7.   (y < x) :: ordered(cons(x, cons(y, u))) -> false
 8.   (x =< y) :: ordered(cons(x, cons(y, u))) -> ordered(cons(y, u))
 9.   (x =< y) :: insert(x, cons(y, u)) -> cons(x, cons(y, u))
10.   (y < x) :: insert(x, cons(y, u)) -> cons(y, insert(x, u))
11.   (y < x) :: ordered(cons(y, insert(x, u))) -> ordered(cons(y, u))

Your system is complete!
The current system is contextually confluent.
Knuth-Bendix ran for 5 seconds.

%% Le syste`me est comple'te' par l'ajout de la re`gle 11 %%

%% L'e'quation initiale 3 a e'te' supprime'e parce que < e'tant transitif
et irre'flexif, elle devient triviale. %%

%% L'e'quation initiale 4 est e'galement supprime'e par transitivite' de =<
(qui ajoute le terme y =< z) et par la re`gle 1. %%
```

```
% 2) Exemple de preuve par re'e'criture contextuelle %

-> read inf


User equations:

1.    ((x =< y) & (y =< x)) == (x = y)
2.    ((x < y) & (y =< x)) == false
3.    ((x < y) | (y =< x)) == true
4.    (x < y) :: inf(x, y) == x
5.    (y =< x) :: inf(x, y) == y

No critical pair equations.

No rewrite rules.

Note that the following identifiers were parsed as nullary operators:
     false  true

%% Les pre'dicats =< et < sont de'clare's transitifs %%
-> tr =< <

Among the above operators, what is reflexive one ? Please tape in them.
Operators : =<

%% On de'clare les ope'rateurs conditionnels %%
-> condi inf


%% Mise en route de la proce'dure de comple'tion %%
-> k

Starting Knuth-Bendix...

Ordered the equation:
    ((x =< y) & (y =< x)) == EQ(y, x)
    into the rewrite rule:
    ((x =< y) & (y =< x)) -> EQ(y, x)

Ordered the equation:
    ((x < y) & (y =< x)) == false
    into the rewrite rule:
    ((x < y) & (y =< x)) -> false

Ordered the equation:
    ((x < y) # (y =< x)) == true
    into the rewrite rule:
    ((x < y) # (y =< x)) -> true

Ordered the equation:
    (x < y) :: inf(x, y) == x
    into the rewrite rule:
    (x < y) :: inf(x, y) -> x

Ordered the equation:
    (y =< x) :: inf(x, y) == y
    into the rewrite rule:
    (y =< x) :: inf(x, y) -> y
```

```
There are currently 5 rules and 0 equations in the system.


Starting to compute critical pairs...


Starting to reduce and order equations...

No user equations.

No critical pair equations.

Rewrite rules:

1.    ((x < y) & (y =< x)) -> false
2.    ((x < y) # (y =< x)) -> true
3.    ((x =< y) & (y =< x)) -> EQ(y, x)
4.    (x < y) :: inf(x, y) -> x
5.    (y =< x) :: inf(x, y) -> y

Your system is complete!
The current system is contextually confluent.
Knuth-Bendix ran for 0 seconds.

%% preuve de la commutativite' de "inf" %%
-> prove

Please enter the equation you would like proved, terminated with <ESC>:
inf(x,y) == inf(y,x)

Your system is already complete!

Following equation:
    inf(x, y) == inf(y, x)
    has been proven because the (C-)identity of normal form(s):
        The left handside is(are):
        (x < y) :: x
        (y =< x) :: y
        The right handside is(are):
        (y < x) :: y
        (x =< y) :: x

Your equation has been proved by contextual rewriting in your system.

%% preuve de l'associativite' de "inf" %%
-> prove

Please enter the equation you would like proved, terminated with <ESC>:
inf(x,inf(y,z)) == inf(inf(x,y),z)

Your system is already complete!

Following equation:
    inf(x, inf(y, z)) == inf(inf(x, y), z)
    has been proven because the (C-)identity of normal form(s):
        The left handside is(are):
        (((x < y) & ((x < z) & (y < z))) # ((x < z) & (z =< y))) :: x
        ((y =< x) & (y < z)) :: y
        ((z =< x) & (z =< y)) :: z
        The right handside is(are):
```

```
((x < y) & (x < z)) :: x
(((y =< x) & ((z =< x) & (z =< y))) # ((x < y) & (z =< x))) :: z
((y =< x) & (y < z)) :: y
```

Your equation has been proved by contextual rewriting in your system.

```
%% 3) Spe'cification de la pile avec erreurs %%

-> read stack1_err


User equations:

 1.   push(errs, x) == errs
 2.   push(u, erre) == errs
 3.   defs(empty_stack) == true
 4.   defs(push(u, x)) == (defs(u) & defe(x))
 5.   defs(errs) == false
 6.   (defs(u) & defe(x)) :: pop(push(u, x)) == u
 7.   (defs(u) & defe(x)) :: top(push(u, x)) == x
 8.   is_empty(empty_stack) == true
 9.   (defs(u) & defe(x)) :: is_empty(push(u, x)) == false
10.   top(errs) == erre
11.   pop(errs) == errs
12.   is_empty(errs) == false
13.   pop(empty_stack) == errs
14.   top(empty_stack) == erre

No critical pair equations.

No rewrite rules.

Note that the following identifiers were parsed as nullary operators:
     errs         erre         empty_stack  true         false

-> conditional pop top is_empty


%% on de'clare les constructeurs de la spe'cification, la pre'ce'dence est
par conse'quent mise a` jour %%
-> constructors push empty_stack


Precedence extended.

-> kb

Starting Knuth-Bendix...

Ordered the equation:
    push(errs, x) == errs
    into the rewrite rule:
    push(errs, x) -> errs

Ordered the equation:
    defs(empty_stack) == true
    into the rewrite rule:
    defs(empty_stack) -> true

Ordered the equation:
    defs(errs) == false
    into the rewrite rule:
    defs(errs) -> false

Ordered the equation:
    (defs(u) & defe(x)) :: pop(push(u, x)) == u
```

```
    into the rewrite rule:
    (defs(u) & defe(x)) :: pop(push(u, x)) -> u

Ordered the equation:
    (defs(u) & defe(x)) :: top(push(u, x)) == x
    into the rewrite rule:
    (defs(u) & defe(x)) :: top(push(u, x)) -> x

Ordered the equation:
    is_empty(empty_stack) == true
    into the rewrite rule:
    is_empty(empty_stack) -> true

Ordered the equation:
    (defs(u) & defe(x)) :: is_empty(push(u, x)) == false
    into the rewrite rule:
    (defs(u) & defe(x)) :: is_empty(push(u, x)) -> false

Ordered the equation:
    pop(errs) == errs
    into the rewrite rule:
    pop(errs) -> errs

Ordered the equation:
    is_empty(errs) == false
    into the rewrite rule:
    is_empty(errs) -> false

There are currently 9 rules and 5 equations in the system.


Starting to compute critical pairs...


Starting to reduce and order equations...

Consider the following equation:

top(errs) == erre

The following precedence suggestions may allow the equation to be ordered:

    1.  errs > erre
    2.  top > erre

ALL (orders), #, REVERSE, STATUS, DIVIDE, POSTPONE, MANUAL,
OPERATORS, INTERRUPT, or UNDO: 2


Ordered the equation:
    top(errs) == erre
    into the rewrite rule:
    top(errs) -> erre

Ordered the equation:
    top(empty_stack) == erre
    into the rewrite rule:
    top(empty_stack) -> erre

There are currently 11 rules and 3 equations in the system.
```

```
Starting to compute critical pairs...


Starting to reduce and order equations...

Consider the following equation:

pop(empty_stack) == errs

The following precedence suggestions may allow the equation to be ordered:

    1.  pop > errs

ALL (orders), #, REVERSE, STATUS, DIVIDE, POSTPONE, MANUAL,
OPERATORS, INTERRUPT, or UNDO: 1


Ordered the equation:
    pop(empty_stack) == errs
    into the rewrite rule:
    pop(empty_stack) -> errs

There are currently 12 rules and 2 equations in the system.


Starting to compute critical pairs...


Starting to reduce and order equations...

Consider the following equation:

push(u, erre) == errs

The following precedence suggestions may allow the equation to be ordered:

    1.  erre > errs

ALL (orders), #, STATUS, DIVIDE, POSTPONE, MANUAL, OPERATORS,
INTERRUPT, or UNDO: 1


Ordered the equation:
    push(u, erre) == errs
    into the rewrite rule:
    push(u, erre) -> errs

There are currently 13 rules and 1 equation in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    push(u, erre) -> errs
    and the rule:
    (defs(u) & defe(x)) :: pop(push(u, x)) -> u
        are as follows:
        (defe(erre) & defs(u)) :: pop(errs) == u
```

```
        (they are added to the system as kb-equations.)
    and the rule:
    (defs(u) & defe(x)) :: top(push(u, x)) -> x
        are as follows:
        (defe(erre) & defs(u)) :: top(errs) == erre
        (they are added to the system as kb-equations.)
    and the rule:
    (defs(u) & defe(x)) :: is_empty(push(u, x)) -> false
        are as follows:
        (defe(erre) & defs(u)) :: is_empty(errs) == false
        (they are added to the system as kb-equations.)

There are currently 13 rules and 4 equations in the system.


Starting to reduce and order equations...

There are currently 13 rules and 2 equations in the system.

Consider the following equation:

defs(push(u, x)) == (defs(u) & defe(x))

The following precedence suggestions may allow the equation to be ordered:

    1.  defs > &
    2.  defs > defe

ALL (orders), #, REVERSE, STATUS, DIVIDE, POSTPONE, MANUAL,
OPERATORS, INTERRUPT, or UNDO: a


All precedence suggestions accepted.  Equation ordered.

Ordered the equation:
    defs(push(u, x)) == (defs(u) & defe(x))
    into the rewrite rule:
    defs(push(u, x)) -> (defs(u) & defe(x))

There are currently 14 rules and 1 equation in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    defs(push(u, x)) -> (defs(u) & defe(x))
    and the rule:
    push(errs, x) -> errs
        are as follows:
        defs(errs) == (defs(errs) & defe(x))
        (they are added to the system as kb-equations.)
    and the rule:
    push(u, erre) -> errs
        are as follows:
        defs(errs) == (defs(u) & defe(erre))
        (they are added to the system as kb-equations.)

There are currently 14 rules and 3 equations in the system.
```

```
Starting to reduce and order equations...

Ordered the equation:
    (defe(erre) & defs(u)) == false
    into the rewrite rule:
    (defe(erre) & defs(u)) -> false

There are currently 15 rules and 0 equations in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    (defe(erre) & defs(u)) -> false
    and the rule:
    defs(empty_stack) -> true
        are as follows:
        (defe(erre) & true) == false
        (they are added to the system as kb-equations.)
    and the rule:
    defs(errs) -> false
        are as follows:
        (defe(erre) & false) == false
        (they are added to the system as kb-equations.)
    and the rule:
    defs(push(u, x)) -> (defs(u) & defe(x))
        are as follows:
        (defe(erre) & (defs(u) & defe(x))) == false
        (they are added to the system as kb-equations.)

There are currently 15 rules and 3 equations in the system.


Starting to reduce and order equations...

Ordered the equation:
    defe(erre) == false
    into the rewrite rule:
    defe(erre) -> false

        Left-hand side reduced:
        (defe(erre) & defs(u)) -> false
            became:
            (false & defs(u)) == false

There are currently 15 rules and 0 equations in the system.


Starting to compute critical pairs...


Starting to reduce and order equations...

No user equations.

No critical pair equations.

Rewrite rules:

 1.   defs(empty_stack) -> true
```

```
 2.   defs(errs) -> false
 3.   is_empty(empty_stack) -> true
 4.   pop(errs) -> errs
 5.   is_empty(errs) -> false
 6.   push(errs, x) -> errs
 7.   top(errs) -> erre
 8.   top(empty_stack) -> erre
 9.   pop(empty_stack) -> errs
10.   push(u, erre) -> errs
11.   defs(push(u, x)) -> (defs(u) & defe(x))
12.   defe(erre) -> false
13.   (defs(u) & defe(x)) :: pop(push(u, x)) -> u
14.   (defs(u) & defe(x)) :: top(push(u, x)) -> x
15.   (defs(u) & defe(x)) :: is_empty(push(u, x)) -> false

Your system is complete!
The current system is contextually confluent.
Knuth-Bendix ran for 4 seconds.

%% Le syste`me est comple'te' par l'ajout de la re`gle 12 qui est la duale
de la re`gle 2. %%
```

```
%% 4) Spe'cification des files avec pointeurs %%

-> read examen


User equations:

1.    insere(x, ajoute(y, xl)) == ajoute(y, insere(x, xl))
2.    adroite(lvide) == true
3.    adroite(ajoute(x, xl)) == false
4.    adroite(insere(x, xl)) == adroite(xl)
5.    adroite(xl) :: avance(xl) == xl
6.    not(adroite(xl)) :: avance(ajoute(x, xl)) == ajoute(x, avance(xl))
7.    adroite(xl) :: avance(ajoute(x, xl)) == insere(x, xl)

No critical pair equations.

No rewrite rules.

Note that the following identifiers were parsed as nullary operators:
     lvide  true   false

0.820000
-> condi avance


0.060000
-> k

Starting Knuth-Bendix...

Ordered the equation:
    adroite(lvide) == true
    into the rewrite rule:
    adroite(lvide) -> true

Ordered the equation:
    adroite(ajoute(x, xl)) == false
    into the rewrite rule:
    adroite(ajoute(x, xl)) -> false

Ordered the equation:
    adroite(insere(x, xl)) == adroite(xl)
    into the rewrite rule:
    adroite(insere(x, xl)) -> adroite(xl)

Ordered the equation:
    adroite(xl) :: avance(xl) == xl
    into the rewrite rule:
    adroite(xl) :: avance(xl) -> xl

There are currently 4 rules and 3 equations in the system.


Starting to compute critical pairs...


Starting to reduce and order equations...

Consider the following equation:
```

```
insere(x, ajoute(y, xl)) == ajoute(y, insere(x, xl))

The following precedence suggestions may allow the equation to be ordered:

    1.  insere > ajoute
    2.  ajoute > insere

#, REVERSE, STATUS, DIVIDE, POSTPONE, MANUAL, OPERATORS,
INTERRUPT, or UNDO: 1


Ordered the equation:
    insere(x, ajoute(y, xl)) == ajoute(y, insere(x, xl))
    into the rewrite rule:
    insere(x, ajoute(y, xl)) -> ajoute(y, insere(x, xl))

There are currently 5 rules and 2 equations in the system.


Starting to compute critical pairs...

Critical pairs between the rule:
    insere(x, ajoute(y, xl)) -> ajoute(y, insere(x, xl))
    and the rule:
    adroite(insere(x, xl)) -> adroite(xl)
        are as follows:
        adroite(ajoute(y, insere(x, xl))) == adroite(ajoute(y, xl))
        (they are added to the system as kb-equations.)

There are currently 5 rules and 3 equations in the system.


Starting to reduce and order equations...

There are currently 5 rules and 2 equations in the system.

Consider the following equation:

adroite(xl) :: avance(ajoute(x, xl)) == insere(x, xl)

The following precedence suggestions may allow the equation to be ordered:

    1.  avance > insere

ALL (orders), #, REVERSE, STATUS, DIVIDE, POSTPONE, MANUAL,
OPERATORS, INTERRUPT, or UNDO: 1


Ordered the equation:
    adroite(xl) :: avance(ajoute(x, xl)) == insere(x, xl)
    into the rewrite rule:
    adroite(xl) :: avance(ajoute(x, xl)) -> insere(x, xl)

Ordered the equation:
    (true # adroite(xl)) :: avance(ajoute(x, xl)) == ajoute(x, avance(xl))
    into the rewrite rule:
    (true # adroite(xl)) :: avance(ajoute(x, xl)) -> ajoute(x, avance(xl))

There are currently 7 rules and 0 equations in the system.
```

```
Starting to compute critical pairs...


Starting to reduce and order equations...

No user equations.

No critical pair equations.

Rewrite rules:

1.   adroite(lvide) -> true
2.   adroite(ajoute(x, xl)) -> false
3.   adroite(insere(x, xl)) -> adroite(xl)
4.   insere(x, ajoute(y, xl)) -> ajoute(y, insere(x, xl))
5.   adroite(xl) :: avance(xl) -> xl
6.   adroite(xl) :: avance(ajoute(x, xl)) -> insere(x, xl)
7.   (true # adroite(xl)) :: avance(ajoute(x, xl)) -> ajoute(x, avance(xl))

Your system is complete!
The current system is contextually confluent.
Knuth-Bendix ran for 3 seconds.
```

ANNEXE B : RECOND

Annexe B : RECOND

```
%% R est l'ensemble des re`gles conditionnelles dont toutes les paires
critiques entre ces re`gles ont e'te' calcule'es. C'est l'ensemble des
re`gles marque'es. %%

%% T est l'ensemble des re`gles de`ja` utilise'es pour simplifier le syste`me.
Aucune paire critique entre ces re`gles, ni avec les re`gles de R n'a e'te'
calcule'e. %%

%% S contient la re`gle courante de simplification. %%

%% E est un ensemble d'e'quations en attente d'e^tre oriente'es. %%

%% 1) Spe'cification de "min", voir chapitre 5, page 202 %%

() : unit


() : unit

(*** min(nil) = infini ***)
Value eq1 =
      ([],(Term ("min",[(Term ("Nil",[]))])),[],(Term ("Infini",[])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** x1 =< min(x2) = true ==> min(cons(x1, x2)) = x1) ***)
Value eq2 =
      ([((Term ("inf",[(Var 1); (Term ("min",[(Var 2)]))])),[],
         (Term ("True",[])),[])],
       (Term ("min",[(Term ("cons",[(Var 1); (Var 2)]))])),[],(
       Var 1),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** x1 =< min(x2) = False => min(cons(x1, x2)) = min(x2) ***)
Value eq3 =
      ([((Term ("inf",[(Var 1); (Term ("min",[(Var 2)]))])),[],
         (Term ("False",[])),[])],
       (Term ("min",[(Term ("cons",[(Var 1); (Var 2)]))])),[],
       (Term ("min",[(Var 2)])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

Value E =
      [([],(Term ("min",[(Term ("Nil",[]))])),[],(Term ("Infini",[])),[]);
       ([((Term ("inf",[(Var 1); (Term ("min",[(Var 2)]))])),[],
          (Term ("True",[])),[])],
        (Term ("min",[(Term ("cons",[(Var 1); (Var 2)]))])),[],(
        Var 1),[]);
       ([((Term ("inf",[(Var 1); (Term ("min",[(Var 2)]))])),[],
          (Term ("False",[])),[])],
        (Term ("min",[(Term ("cons",[(Var 1); (Var 2)]))])),[],
        (Term ("min",[(Var 2)])),[])] :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list) list

() : unit
```

```
  Ensemble des equations condionnelles
E=={
 (min(Nil) = Infini);
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}

==================================
----          TRAITEMENT de E----
R=={}

T=={}

S=={}

E=={
 (min(Nil) = Infini);
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}


==================================
    Elimination des equations triviales
E=={
 (min(Nil) = Infini);
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
     Simplification
E=={
 (min(Nil) = Infini);
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
     Elimination des equations triviales
E=={
 (min(Nil) = Infini);
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
----Orientation des identites ----
R=={}

T=={}

S=={}

E=={
 (min(Nil) = Infini);
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (min(Nil) -> Infini) }

S=={}
```

plainted3.

```
E=={
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}


==================================
    Elimination des equations triviales
E=={
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
     Simplification
E=={
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
     Elimination des equations triviales
E=={
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
----Orientation des identites ----
R=={}

T=={
 (min(Nil) -> Infini) }

S=={}

E=={
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) = x1);
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (min(Nil) -> Infini) ;
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) }

S=={}

E=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}


==================================
    Elimination des equations triviales
E=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
     Simplification
E=={
 ((inf(x1,min(x2))=False) & (inf(x1,min(x2))=True)) => (x1 = min(x2));
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
     Elimination des equations triviales
```

```
E=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}
==================================
----Orientation des identites ----
R=={}

T=={
 (min(Nil) -> Infini) ;
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) }

S=={}

E=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) = min(x2))}


----Calcul des CCP----
R=={}

T=={
 (min(Nil) -> Infini) ;
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) ;
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) -> min(x2)) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (min(Nil) -> Infini) }

T=={
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) ;
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) -> min(x2)) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) ;
 (min(Nil) -> Infini) }

T=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) -> min(x2)) }

S=={}

E=={}


==================================
----          TRAITEMENT de E----
R=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) -> min(x2)) ;
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) ;
```

```
 (min(Nil) -> Infini) }

T=={}

S=={}

E=={
 ((inf(x1,min(x2))=True) & (inf(x1,min(x2))=False)) => (x1 = min(x2))}


==================================
    Elimination des equations triviales
E=={}
==================================
     Simplification
E=={}
==================================
     Elimination des equations triviales
E=={}
==================================
----Orientation des identites ----
R=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) -> min(x2)) ;
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) ;
 (min(Nil) -> Infini) }

T=={}

S=={}

E=={}


========================================
 Succes de la completion conditionnelle
========================================
   Le systeme de reecriture equivalent a E est le suivant:
R=={
 (inf(x1,min(x2))=False) => (min(cons(x1,x2)) -> min(x2)) ;
 (inf(x1,min(x2))=True) => (min(cons(x1,x2)) -> x1) ;
 (min(Nil) -> Infini) }
```

```
%% 2) Spe'cification de la file avec pointeurs %%

() : unit

(*** insere(x,ajoute(y,l)) = ajoute(y,insere(x,l)) ***)
let eq1=interface_YACC<<Insere(x,Ajoute(y,l))=Ajoute(y,Insere(x,l))>>;;

(*** adroite([]) = true ***)
let eq2=interface_YACC<<Adroite(Lvide)=True>>;;

(*** adroite(ajoute(x,l)) = false ***)
let eq3=interface_YACC<<Adroite(Ajoute(x,l))=False>>;;

(*** adroite(insere(x,l)) = adroite(l) ***)
let eq4=interface_YACC<<Adroite(Insere(x,l))=Adroite(l)>>;;

(*** adroite(l) = true ==> avance(l) = l ***)
let eq5=interface_YACC<<(Adroite(l)=True) =>Avance(l)=l>>;;

(*** adroite(l) = false ==> avance(ajoute(x, l)) = ajoute(x,avance(l)) ***)
let eq6=interface_YACC<<(Adroite(l)=False) =>Avance(Ajoute(x,l))=
                                             Ajoute(x,Avance(l))>>;;

(*** adroite(l) = true ==> avance(ajoute(x,l)) = insere(x,l) ***)
let eq7=interface_YACC<<(Adroite(l)=True) =>Avance(Ajoute(x,l))=
                                             Insere(x,l)>>;;

let E=[eq1;eq2;eq3;eq4;eq5;eq6;eq7];;

(*
let RES=RECOND ord_predicat rpo precedence E;;
*)


  Ensemble des equations conditionnelles
E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(Lvide) = True);
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))}

==================================
----           TRAITEMENT de E----
R=={}

T=={}

S=={}

E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(Lvide) = True);
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))}
```

```
==================================
    Elimination des equations triviales
E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(Lvide) = True);
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))}
==================================
     Simplification
E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(Lvide) = True);
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))}
==================================
     Elimination des equations triviales
E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(Lvide) = True);
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))}
==================================
----Orientation des identites ----
R=={}

T=={}

S=={}

E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(Lvide) = True);
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (Adroite(Lvide) -> True) }

S=={}

E=={
```

```
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


====================================
    Elimination des equations triviales
E=={
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
====================================
     Simplification
E=={
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
====================================
     Elimination des equations triviales
E=={
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
====================================
----Orientation des identites ----
R=={}

T=={
 (Adroite(Lvide) -> True) }

S=={}

E=={
 (Adroite(Ajoute(x1,x2)) = False);
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


====================================
----          TRAITEMENT de E----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) }
```

```
S=={}

E=={
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


==================================
    Elimination des equations triviales
E=={
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Simplification
E=={
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Elimination des equations triviales
E=={
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
----Orientation des identites ----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) }

S=={}

E=={
 (Adroite(insere(x1,x2)) = Adroite(x2));
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
```

```
 (Adroite(insere(x1,x2)) -> Adroite(x2)) }

S=={}

E=={
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


==================================
    Elimination des equations triviales
E=={
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Simplification
E=={
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Elimination des equations triviales
E=={
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
----Orientation des identites ----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) }

S=={}

E=={
 (Adroite(x1)=True) => (Avance(x1) = x1);
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) }
```

examen

```
S=={}

E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


==================================
    Elimination des equations triviales
E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Simplification
E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 ((Adroite(x1)=True) &(False=True)) => (Ajoute(x2,x1) = Insere(x2,x1));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 ((Adroite(x1)=False) & (False=True) & (Adroite(x1)=True)) => (Ajoute(x2,x1)
 = Ajoute(x2,x1));
 ((Adroite(x1)=False) & (False=True)) => (Ajoute(x2,x1) = Ajoute(x2,Avance(x1)
));
 ((Adroite(x1)=False) & (Adroite(x1)=True)) => (Avance(Ajoute(x2,x1))
= Ajoute(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Elimination des equations triviales

********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))

********************************
  ARRET provoque:  Equation consideree comme non triviale
NOMBRE de CLAUSES CALCULEES: 60

********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))

********************************
  Toutes les clauses contiennent des litteraux negatifs
 Equation non triviale E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
----Orientation des identites ----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) }

S=={}
```

```
E=={
 (Insere(x1,Ajoute(x2,x3)) = Ajoute(x2,Insere(x1,x3)));
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


====================================
----          TRAITEMENT de E----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) }

S=={}

E=={
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


====================================
    Elimination des equations triviales

**********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))

**********************************
  ARRET provoque:  Equation consideree comme non triviale
NOMBRE de CLAUSES CALCULEES: 60

**********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))

**********************************
  Toutes les clauses contiennent des litteraux negatifs
 Equation non triviale E=={
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
====================================
     Simplification
E=={
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
====================================
     Elimination des equations triviales

**********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1))

**********************************
  ARRET provoque:  Equation consideree comme non triviale
NOMBRE de CLAUSES CALCULEES: 60
```

```
**********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))

**********************************
  Toutes les clauses contiennent des litteraux negatifs
 Equation non triviale E=={
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
----Orientation des identites ----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) }

S=={}

E=={
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) = Insere(x2,x1));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) }

S=={}

E=={
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


==================================
    Elimination des equations triviales

**********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))

**********************************
  Toutes les clauses contiennent des litteraux negatifs
 Equation non triviale E=={
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Simplification
E=={
 ((Adroite(x1)=False) & (Adroite(x1)=True)) => (Insere(x2,x1) =
```

```
Ajoute(x2,Avance(x1)));
 ((False=True) & (Adroite(x1)=False)) => (Avance(Ajoute(x2,x1)) =
Ajoute(x2,Avance(x1)));
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
     Elimination des equations triviales

**********************************
          PROUVEUR APPELE avec :
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))

**********************************
  Toutes les clauses contiennent des litteraux negatifs
 Equation non triviale E=={
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}
==================================
----Orientation des identites ----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) }

S=={}

E=={
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) = Ajoute(x2,Avance(x1)))}


----Calcul des CCP----
R=={}

T=={
 (Adroite(Lvide) -> True) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) ;
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) -> Ajoute(x2,Avance(x1))) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (Adroite(Lvide) -> True) }

T=={
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) ;
```

```
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) -> Ajoute(x2,Avance(x1))) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(Lvide) -> True) }

T=={
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) ;
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) -> Ajoute(x2,Avance(x1))) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(Lvide) -> True) }

T=={
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) ;
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) -> Ajoute(x2,Avance(x1))) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(Lvide) -> True) }

T=={
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) ;
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) -> Ajoute(x2,Avance(x1))) }

S=={}

E=={}


----Calcul des CCP----
R=={
```

```
= Ajoute(x2,Avance(x1)))

*********************************
  ARRET provoque:  Equation consideree comme non triviale
NOMBRE de CLAUSES CALCULEES: 60
E=={
 ((Adroite(Ajoute(x2,x1))=True) & (Adroite(x1)=False)) => (Ajoute(x2,x1)
= Ajoute(x2,Avance(x1)))}
====================================
     Simplification
E=={
 ((False=True) & (Adroite(x1)=False) & (Adroite(x1)=True)) => (Ajoute(x2,x1)
= Ajoute(x2,x1));
 ((False=True) & (Adroite(x1)=False)) => (Ajoute(x2,x1) =
Ajoute(x2,Avance(x1)))}
====================================
     Elimination des equations triviales
E=={}
====================================
----Orientation des identites ----
R=={
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) -> Ajoute(x2,Avance(x1))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(Lvide) -> True) }

T=={}

S=={}

E=={}


========================================
 Succes de la completion conditionnelle
========================================
   Le systeme de reecriture equivalent a E est le suivant:
R=={
 (Adroite(x1)=False) => (Avance(Ajoute(x2,x1)) -> Ajoute(x2,Avance(x1))) ;
 (Adroite(x1)=True) => (Avance(Ajoute(x2,x1)) -> Insere(x2,x1)) ;
 (Insere(x1,Ajoute(x2,x3)) -> Ajoute(x2,Insere(x1,x3))) ;
 (Adroite(x1)=True) => (Avance(x1) -> x1) ;
 (Adroite(insere(x1,x2)) -> Adroite(x2)) ;
 (Adroite(Ajoute(x1,x2)) -> False) ;
 (Adroite(Lvide) -> True) }
```

```
3) Spe'cification du pre'dicat "=<" dans l'ensemble des entiers relatifs. %%

() : unit

(*** (succ(pred(x1)) = x1) ***)
Value eq1 = ([],(Term ("s",[(Term ("p",[(Var 1)]))])),[],(Var 1),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** (pred(succ(x1)) = x1) ***)
Value eq2 = ([],(Term ("p",[(Term ("s",[(Var 1)]))])),[],(Var 1),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** 0 =< 0 = true ***)
Value eq3 =
      ([],(Term ("inf",[(Term ("0",[])); (Term ("0",[]))])),[],
       (Term ("True",[])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(***  0 =< pred(0) = false ***)
Value eq4 =
      ([],
       (Term ("inf",[(Term ("0",[])); (Term ("p",[(Term ("0",[]))]))])),
       [],(Term ("False",[])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** 0 =< x1 = true ==> 0 =< succ(x1) = true ***)
Value eq5 =
      ([((Term ("inf",[(Term ("0",[])); (Var 1)])),[],
         (Term ("True",[])),[])],
       (Term ("inf",[(Term ("0",[])); (Term ("s",[(Var 1)]))])),[],
       (Term ("True",[])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** 0 =< x1 = false ==> 0 =< pred(x1) = false ***)
Value eq6 =
      ([((Term ("inf",[(Term ("0",[])); (Var 1)])),[],
         (Term ("False",[])),[])],
       (Term ("inf",[(Term ("0",[])); (Term ("p",[(Var 1)]))])),[],
       (Term ("False",[])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** succ(x1) =< x2 = x1 =< pred(x2) ***)
Value eq7 =
      ([],(Term ("inf",[(Term ("s",[(Var 1)])); (Var 2)])),[],
       (Term ("inf",[(Var 1); (Term ("p",[(Var 2)]))])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)

(*** pred(x1) =< x2 = x1 =< succ(x2) ***)
Value eq8 =
      ([],(Term ("inf",[(Term ("p",[(Var 1)])); (Var 2)])),[],
       (Term ("inf",[(Var 1); (Term ("s",[(Var 2)]))])),[]) :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list)
```

```
Value E =
      [([],(Term ("s",[(Term ("p",[(Var 1)]))])),[],(Var 1),[]);
       ([],(Term ("p",[(Term ("s",[(Var 1)]))])),[],(Var 1),[]);
       ([],(Term ("inf",[(Term ("0",[])); (Term ("0",[]))])),[],
        (Term ("True",[])),[]);
       ([],
        (Term ("inf",[(Term ("0",[])); (Term ("p",[(Term ("0",[]))]))])),
        [],(Term ("False",[])),[]);
       ([((Term ("inf",[(Term ("0",[])); (Var 1)])),[],
          (Term ("True",[])),[])],
        (Term ("inf",[(Term ("0",[])); (Term ("s",[(Var 1)]))])),[],
        (Term ("True",[])),[]);
       ([((Term ("inf",[(Term ("0",[])); (Var 1)])),[],
          (Term ("False",[])),[])],
        (Term ("inf",[(Term ("0",[])); (Term ("p",[(Var 1)]))])),[],
        (Term ("False",[])),[]);
       ([],(Term ("inf",[(Term ("s",[(Var 1)])); (Var 2)])),[],
        (Term ("inf",[(Var 1); (Term ("p",[(Var 2)]))])),[]);
       ([],(Term ("inf",[(Term ("p",[(Var 1)])); (Var 2)])),[],
        (Term ("inf",[(Var 1); (Term ("s",[(Var 2)]))])),[])] :
      ((term * 'a list * term * 'b list) list * term * 'c list * term *
       'd list) list

() : unit

  Ensemble des equations conditionnelles
E=={
 (s(p(x1)) = x1);
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)))}

==================================
----           TRAITEMENT de E----
R=={}

T=={}

S=={}

E=={
 (s(p(x1)) = x1);
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)))}


==================================
    Elimination des equations triviales
E=={
 (s(p(x1)) = x1);
```

```
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)))}
==================================
     Simplification
E=={
 (s(p(x1)) = x1);
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)))}
==================================
     Elimination des equations triviales
E=={
 (s(p(x1)) = x1);
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)))}
==================================
----Orientation des identites ----
R=={}

T=={}

S=={}

E=={
 (s(p(x1)) = x1);
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)))}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (s(p(x1)) -> x1) }

S=={}

E=={
 (p(s(x1)) = x1);
 (inf(0,0) = True);
```

```
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
    Elimination des equations triviales
E=={
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Simplification
E=={
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Elimination des equations triviales
E=={
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
----Orientation des identites ----
R=={}

T=={
 (s(p(x1)) -> x1) }

S=={}

E=={
 (p(s(x1)) = x1);
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
----           TRAITEMENT de E----
R=={}
```

```
T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) }

S=={}

E=={
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
    Elimination des equations triviales
E=={
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Simplification
E=={
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Elimination des equations triviales
E=={
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
----Orientation des identites ----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) }

S=={}

E=={
 (inf(0,0) = True);
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
```

```
==================================
----          TRAITEMENT de E----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) }

S=={}

E=={
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
    Elimination des equations triviales
E=={
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Simplification
E=={
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Elimination des equations triviales
E=={
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
----Orientation des identites ----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) }

S=={}

E=={
 (inf(0,p(0)) = False);
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
```

```
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
----          TRAITEMENT de E----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) }

S=={}

E=={
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
    Elimination des equations triviales
E=={
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Simplification
E=={
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Elimination des equations triviales
E=={
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
----Orientation des identites ----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) }

S=={}

E=={
 (inf(s(x1),x2) = inf(x1,p(x2)));
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
```

```
==================================
----          TRAITEMENT de E----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) }

S=={}

E=={
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
    Elimination des equations triviales
E=={
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Simplification
E=={
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Elimination des equations triviales
E=={
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
----Orientation des identites ----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) }

S=={}

E=={
 (inf(p(x1),x2) = inf(x1,s(x2)));
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
----          TRAITEMENT de E----
```

```
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) }

S=={}

E=={
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


====================================
    Elimination des equations triviales
E=={
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
====================================
     Simplification
E=={
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
====================================
     Elimination des equations triviales
E=={
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
====================================
----Orientation des identites ----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) }

S=={}

E=={
 (inf(0,x1)=True) => (inf(0,s(x1)) = True);
 (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


====================================
----          TRAITEMENT de E----
R=={}

T=={
 (s(p(x1)) -> x1) ;
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
```

```
  (inf(s(x1),x2) -> inf(x1,p(x2))) ;
  (inf(p(x1),x2) -> inf(x1,s(x2))) ;
  (inf(0,x1)=True) => (inf(0,s(x1)) -> True) }

S=={}

E=={
  (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


==================================
    Elimination des equations triviales
E=={
  (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Simplification
E=={
  (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
     Elimination des equations triviales
E=={
  (inf(0,x1)=False) => (inf(0,p(x1)) = False)}
==================================
----Orientation des identites ----
R=={}

T=={
  (s(p(x1)) -> x1) ;
  (p(s(x1)) -> x1) ;
  (inf(0,0) -> True) ;
  (inf(0,p(0)) -> False) ;
  (inf(s(x1),x2) -> inf(x1,p(x2))) ;
  (inf(p(x1),x2) -> inf(x1,s(x2))) ;
  (inf(0,x1)=True) => (inf(0,s(x1)) -> True) }

S=={}

E=={
  (inf(0,x1)=False) => (inf(0,p(x1)) = False)}


----Calcul des CCP----
R=={}

T=={
  (s(p(x1)) -> x1) ;
  (p(s(x1)) -> x1) ;
  (inf(0,0) -> True) ;
  (inf(0,p(0)) -> False) ;
  (inf(s(x1),x2) -> inf(x1,p(x2))) ;
  (inf(p(x1),x2) -> inf(x1,s(x2))) ;
  (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
  (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


----Calcul des CCP----
```

```
R=={
 (s(p(x1)) -> x1) }

T=={
 (p(s(x1)) -> x1) ;
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


====================================
----           TRAITEMENT de E----
R=={
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={
 (s(x1) = s(x1));
 (p(x2) = p(x2))}


====================================
    Elimination des equations triviales
E=={}
====================================
     Simplification
E=={}
====================================
     Elimination des equations triviales
E=={}
====================================
----Orientation des identites ----
R=={
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }
```

```
S=={}

E=={}


----Calcul des CCP----
R=={
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,0) -> True) ;
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,p(0)) -> False) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}
```

```
==================================
----          TRAITEMENT de E----
R=={
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={
 (inf(x3,x2) = inf(p(x3),p(x2)))}


==================================
    Elimination des equations triviales
E=={
 (inf(x3,x2) = inf(p(x3),p(x2)))}
==================================
     Simplification
E=={
 (inf(x3,x2) = inf(x3,x2))}
==================================
     Elimination des equations triviales
E=={}
==================================
----Orientation des identites ----
R=={
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
```

```
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


==================================
----          TRAITEMENT de E----
R=={
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={
 (inf(x3,x2) = inf(s(x3),s(x2)))}


==================================
    Elimination des equations triviales
E=={
 (inf(x3,x2) = inf(s(x3),s(x2)))}
==================================
     Simplification
E=={
 (inf(x3,x2) = inf(x3,x2))}
==================================
     Elimination des equations triviales
E=={}
==================================
----Orientation des identites ----
R=={
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


----Calcul des CCP----
R=={
```

```
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


==================================
----          TRAITEMENT de E----
R=={
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={
 (inf(0,p(x2))=True) => (inf(0,x2) = True)}


==================================
    Elimination des equations triviales
E=={
 (inf(0,p(x2))=True) => (inf(0,x2) = True)}
==================================
     Simplification
E=={
 ((False=True) & (inf(0,x2)=False)) => (inf(0,x2) = True);
 ((inf(0,x2)=True) & (inf(0,p(x2))=True)) => (inf(0,x2) = True)}
==================================
     Elimination des equations triviales
E=={}
==================================
----Orientation des identites ----
R=={
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
```

```
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


----Calcul des CCP----
R=={
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) }

S=={}

E=={}


====================================
----          TRAITEMENT de E----
R=={
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={}

S=={}

E=={
 (inf(0,0)=False) => (False = False);
 (inf(0,s(x2))=False) => (inf(0,x2) = False)}


====================================
    Elimination des equations triviales
E=={
 (inf(0,s(x2))=False) => (inf(0,x2) = False)}
====================================
     Simplification
E=={
 ((True=False) & (inf(0,x2)=True)) => (inf(0,x2) = False);
 ((inf(0,x2)=False) & (inf(0,s(x2))=False)) => (inf(0,x2) = False)}
====================================
     Elimination des equations triviales
E=={}
====================================
----Orientation des identites ----
```

```
R=={
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }

T=={}

S=={}

E=={}


========================================
 Succes de la completion conditionnelle
========================================
   Le systeme de reecriture equivalent a E est le suivant:
R=={
 (inf(0,x1)=False) => (inf(0,p(x1)) -> False) ;
 (inf(0,x1)=True) => (inf(0,s(x1)) -> True) ;
 (inf(p(x1),x2) -> inf(x1,s(x2))) ;
 (inf(s(x1),x2) -> inf(x1,p(x2))) ;
 (inf(0,p(0)) -> False) ;
 (inf(0,0) -> True) ;
 (p(s(x1)) -> x1) ;
 (s(p(x1)) -> x1) }
```

ANNEXE C

LE TEST DE COMPLETUDE SUFFISANTE.

Annexe C : comple'tude suffisante

%% 1) spe'cification avec la constante "entmax" (voir chapitre2, page 75) %%

```
(* ---------- Re`gles inconditionnelles--------------------*)
(*** succ(entmax) -> entmax ***)
let reg1=interface_YACC<<Succ(Entmax)->Entmax>>;;
(*** 0 =< x -> true ***)
let reg2=interface_YACC<<Inf(Zero,x)->True>>;;
(*** succ(x) =< 0 -> false ***)
let reg3=interface_YACC<<Inf(Succ(x),Zero)->False>>;;
(*** succ(x) =< succ(y) -> x =< y ***)
let reg4=interface_YACC<<Inf(Succ(x),Succ(y))->Inf(x,y)>>;;
(*** x =< entmax -> true ***)
let reg5=interface_YACC<<Inf(x,Entmax)->True>>;;
(*** entmax =< 0 -> false ***)
let reg6=interface_YACC<<Inf(Entmax,Zero)->False>>;;
(*** entmax =< succ(y) -> entmax =< y ***)
let reg7=interface_YACC<<Inf(Entmax,Succ(y))->Inf(Entmax,y)>>;;
(*** min([]) -> entmax ***)
let reg8=interface_YACC<<Min(Lvide)->Entmax>>;;
(*** pp(x,[]) -> true ***)
let reg9=interface_YACC<<PP(x,Lvide)->True>>;;
(*** Ri est l'ensemble des regles inconditionnelles ***)
let Ri=[reg1;reg2;reg3;reg4;reg5;reg6;reg7;reg8;reg9];;
(* ---------- Re`gles conditionnelles-----------------------*)
(*** x =< min(l) = true ==> min(add(x,l)) -> x ***)
let reg10=interface_YACC<<(Inf(x,Min(l))=True)=>Min(Add(x,l))->x>>;;
(*** x =< min(l) = false ==> min(add(x,l)) -> min(l) ***)
let reg11=interface_YACC<<(Inf(x,Min(l))=False)=>Min(Add(x,l))->Min(l)>>;;
(*** x =< y = true & pp(x,l) = true ==> pp(x,add(y,l)) -> true ***)
let reg12=interface_YACC<<((Inf(x,y)=True) & (PP(x,l)=True))=>
```

```
PP(x,Add(y,l))->True>>;;

(*** x =< y = false ==> pp(x,add(y,l))->false ***)

let reg13=interface_YACC<<(Inf(x,y)=False)=>PP(x,Add(y,l))->False>>;;

(*** pp(x,l) = false ==> pp(x,add(y,l))->false ***)

let reg14=interface_YACC<<(PP(x,l)=False)=>PP(x,Add(y,l))->False>>;;

let Rc=[reg10;reg11;reg12;reg13;reg14];;

(* ---------- Fonctions a` tester ----------------------------*)

(*** Inf : nat * nat --> bool, min : list --> nat, ***)
(*** pp : nat * list --> bool ***)

let Fonc_a_tester=[ ["Inf";"nat";"nat";"bool"];
                    ["Min";"list";"nat"];
                    ["PP";"nat";"list";"bool"] ];;

(*---------- Fonctions supposees Completes ----------------*)

let Fonc_complet =[  ];;

(* ---------- Constructeurs ---------------------------------*)

(***
    Pour le type nat on a les Constructeurs suivants :

              - 0 : --> nat
              - succ : nat --> nat
              - entmax : --> nat
  ***)

(***
    Pour le type list on a les Constructeurs suivants :

              - [] : --> list
              - Add : nat * list --> list
  ***)


let Const=[ [["nat"];["Zero"];["Succ";"nat"];["Entmax"]] ;
[["list"];["Lvide"];["Add";"nat";"list"]] ];;

(* ---------- Appel de la fonction "test_specification" --------*)

let init=rappel Rc Ri
 in
  test_specification (Rc,Ri,Const,Fonc_a_tester,Fonc_complet,false);;


 ================================================================
 >>>      TEST DE COMPLETUDE D'UNE SPECIFICATION            <<<
 ================================================================

                >>> Rappel des regles introduites <<<
```

```
        >>> Regles inconditionnelles <<<

{Succ(Entmax) --> Entmax }
 {Inf(Zero,x1) --> True }
 {Inf(Succ(x1),Zero) --> False }
 {Inf(Succ(x1),Succ(x2)) --> Inf(x1,x2) }
 {Inf(x1,Entmax) --> True }
 {Inf(Entmax,Zero) --> False }
 {Inf(Entmax,Succ(x1)) --> Inf(Entmax,x1) }
 {Min(Lvide) --> Entmax }
 {PP(x1,Lvide) --> True }

 >>> Tapez  Return pour continuer <<<

        >>> Regles conditionnelles <<<

 { (Inf(x1,Min(x2))=True) ==> Min(Add(x1,x2)) --> x1 }
 { (Inf(x1,Min(x2))=False) ==> Min(Add(x1,x2)) --> Min(x2) }
 { ((Inf(x1,x2)=True)&(PP(x1,x3)=True)) ==> PP(x1,Add(x2,x3)) --> True }
 { (Inf(x1,x2)=False) ==> PP(x1,Add(x2,x3)) --> False }
 { (PP(x1,x2)=False) ==> PP(x1,Add(x3,x2)) --> False }

 >>> Tapez  Return pour continuer <<<

          >>> Affichage de l'arbre de motif de Inf  <<<

 Inf(x1,x2)
   Inf(Zero,x2)  ok
   Inf(Succ(x3),x2)
     Inf(Succ(x3),Zero)  ok
     Inf(Succ(x3),Succ(x1))  ok
     Inf(Succ(x3),Entmax)  ok
   Inf(Entmax,x2)
     Inf(Entmax,Zero)  ok
     Inf(Entmax,Succ(x1))  ok
     Inf(Entmax,Entmax)  ok

 >>> Tapez Return pour continuer  <<<

          >>> Affichage de l'arbre de motif de Min  <<<

 Min(x1)
   Min(Lvide)  ok
   Min(Add(x2,x3))  ok
     (Inf(x2,Min(x3))=True)::x2
     (Inf(x2,Min(x3))=False)::Min(x3)

 >>> Tapez Return pour continuer  <<<
```

```
        >>> Affichage de l'arbre de motif de PP  <<<

PP(x1,x2)
  PP(x1,Lvide)  ok
  PP(x1,Add(x3,x4))  ok
    ((Inf(x1,x3)=True)&(PP(x1,x4)=True))::True
    (Inf(x1,x3)=False)::False
    (PP(x1,x4)=False)::False

>>> Tapez Return pour continuer  <<<

        >>> La specification est complete <<<

Consultation de la liste definitives des regles <O/N> : O

      >>> Consultation des regles inconditionnelles <<<

{Succ(Entmax) --> Entmax }
{Inf(Zero,x1) --> True }
{Inf(Succ(x1),Zero) --> False }
{Inf(Succ(x1),Succ(x2)) --> Inf(x1,x2) }
{Inf(x1,Entmax) --> True }
{Inf(Entmax,Zero) --> False }
{Inf(Entmax,Succ(x1)) --> Inf(Entmax,x1) }
{Min(Lvide) --> Entmax }
{PP(x1,Lvide) --> True }

>>> Tapez  Return pour continuer <<<

      >>> Consultation des regles conditionnelles <<<

{ (Inf(x1,Min(x2))=True) ==> Min(Add(x1,x2)) --> x1 }
{ (Inf(x1,Min(x2))=False) ==> Min(Add(x1,x2)) --> Min(x2) }
{ ((Inf(x1,x2)=True)&(PP(x1,x3)=True)) ==> PP(x1,Add(x2,x3)) --> True }
{ (Inf(x1,x2)=False) ==> PP(x1,Add(x2,x3)) --> False }
{ (PP(x1,x2)=False) ==> PP(x1,Add(x3,x2)) --> False }

>>> Tapez Return pour continuer <<<
```

```
%% 2) listes trie'es avec un ope'rateur qui range un e'le'ment dans une liste
a` la bonne place %%


%% cet exemple est incomplet, il doit e^tre enrichi par l'ajout d'une re`gle
conditionnelle. %%

(* ---------- Re`gles inconditionnelles--------------------*)

(*** 0 =< x -> true ***)

let reg1=interface_YACC<<Inf(Zero,x)->True>>;;

(*** succ(x) =< 0 -> false ***)

let reg2=interface_YACC<<Inf(Succ(x),Zero)->False>>;;

(*** succ(x) =< succ(y) -> x =< y ***)

let reg3=interface_YACC<<Inf(Succ(x),Succ(y))->Inf(x,y)>>;;

(*** insert(x,[]) -> cons(x,[]) ***)

let reg4=interface_YACC<<Insert(x,Lvide)->Cons(x,Lvide)>>;;

(*** ordered(insert(x,u)) -> ordered(u) ***)

let reg5=interface_YACC<<Ordered(Insert(x,u))->Ordered(u)>>;;

(*** Ri est l'ensemble des regles inconditionnelles ***)

let Ri=[reg1;reg2;reg3;reg4;reg5];;

(* ---------- Re`gles conditionnelles----------------------*)

(*** x =< y = true ==> insert(x,cons(y,u)) -> cons(x,cons(y,u)) ***)

let reg6=interface_YACC<<(Inf(x,y)=True)=>Insert(x,Cons(y,u))->
Cons(x,Cons(y,u))>>;;

let Rc=[reg6];;

(* ---------- Fonctions a` tester ------------------------*)

(*** insert : nat * list --> list ***)

let Fonc_a_tester=[ ["Insert";"nat";"list";"list"] ];;

(*---------- Fonctions supposees Completes ----------------*)

let Fonc_complet =[ ["Inf";"nat";"nat";"bool"] ;
                    ["Ordered";"list";"bool"] ];;

(* ---------- Constructeurs ----------------------------------*)

(***
    Pour le type nat on a les Constructeurs suivants :

              - Zero : --> nat
              - Succ : nat --> nat
```

```
  ***)

(***
    Pour le type list on a les Constructeurs suivants :

              - Lvide : --> list
              - Cons : nat * list --> list
  ***)


let Const=[ [["nat"];["Zero"];["Succ";"nat"]] ;
[["list"];["Lvide"];["Cons";"nat";"list"]] ];;

(* ----------- Appel de la fonction "test_specification" --------*)

let init=rappel Rc Ri
 in
  test_specification (Rc,Ri,Const,Fonc_a_tester,Fonc_complet,false);;


 ================================================================
 >>>      TEST DE COMPLETUDE D'UNE SPECIFICATION            <<<
 ================================================================


                >>> Rappel des regles introduites <<<


       _____________________________________
       >>> Regles inconditionnelles <<<
       _____________________________________

 {Inf(Zero,x1) --> True }
 {Inf(Succ(x1),Zero) --> False }
 {Inf(Succ(x1),Succ(x2)) --> Inf(x1,x2) }
 {Insert(x1,Lvide) --> Cons(x1,Lvide) }
 {Ordered(Insert(x1,x2)) --> Ordered(x2) }

 >>> Tapez  Return pour continuer <<<


       _____________________________________
       >>> Regles conditionnelles <<<
       _____________________________________


 { (Inf(x1,x2)=True) ==> Insert(x1,Cons(x2,x3)) --> Cons(x1,Cons(x2,x3)) }

 >>> Tapez  Return pour continuer <<<


          ___________________________________________________
          >>> Affichage de l'arbre de motif de Insert  <<<
          ___________________________________________________


 Insert(x1,x2)
   Insert(x1,Lvide)  ok
   Insert(x1,Cons(x3,x4))  non ok
     (Inf(x1,x3)=True)::Cons(x1,Cons(x3,x4))
```

```
>>> Tapez Return pour continuer  <<<

        ------------------------------------------
        >>> La specification peut etre incomplete <<<
        ------------------------------------------

>>> Je pense qu'il manque des regles de la forme suivante <<<

{ Not ( (Inf(x1,x3)=True) ) ==> Insert(x1,Cons(x3,x4))--> ...}

Voulez-vous avoir une simplification des preconditions
pour les regles conditionnelles <O/N> :O

{(Inf(x1,x3)=False) ==> Insert(x1,Cons(x3,x4)) --> ... }


Voulez-vous continuer le test <O/N> :O


Est-ce que vous pensez qu'il est utile d'ajouter des regles  ?
Reponse <O/N> :O

                >>> Ajout de regles <<<

Les regles doivent etre introduites de la maniere  suivante :

** Les regles inconditionnelles :  g->d
** Les regles conditionnelles   :  ((u0=v0)&(u1=v1)&...)=>g->d

Ajout de regles inconditionnelles <O/N> : N
Ajout de regles conditionnelles   <O/N> : O
** (Inf(x,y)=False)=>Insert(x,Cons(y,u))->Cons(y,Insert(x,u))
Autres Regles <O/N> :N


        ------------------------------------------
        >>> Affichage de l'arbre de motif de Insert  <<<
        ------------------------------------------


Insert(x1,x2)
  Insert(x1,Lvide)  ok
  Insert(x1,Cons(x3,x4))  ok
    (Inf(x1,x3)=True)::Cons(x1,Cons(x3,x4))
    (Inf(x1,x3)=False)::Cons(x3,Insert(x1,x4))

>>> Tapez Return pour continuer  <<<

        ------------------------------------------
        >>>  La specification  est  complete    <<<
        ------------------------------------------


Consultation de la liste definitives des regles <O/N> : O

        ------------------------------------------
```

>>> Consultation des regles inconditionnelles <<<

```
{Inf(Zero,x1) --> True }
{Inf(Succ(x1),Zero) --> False }
{Inf(Succ(x1),Succ(x2)) --> Inf(x1,x2) }
{Insert(x1,Lvide) --> Cons(x1,Lvide) }
{Ordered(Insert(x1,x2)) --> Ordered(x2) }
```

>>> Tapez Return pour continuer <<<

>>> Consultation des regles conditionnelles <<<

```
{ (Inf(x1,x2)=True) ==> Insert(x1,Cons(x2,x3)) --> Cons(x1,Cons(x2,x3)) }
{ (Inf(x1,x2)=False) ==> Insert(x1,Cons(x2,x3)) --> Cons(x2,Insert(x1,x3)) }
```

>>> Tapez Return pour continuer <<<

3) Exemple qui de'finit l'ope'rateur "f" (voir chapitre 2, page 73). %%

```
(* ---------- Re`gles inconditionnelles---------------------*)
(*** 0 > x  -> false ***)
let reg1=interface_YACC<<Greater(Zero,x)->False>>;;
(*** Succ(x) > 0 -> true ***)
let reg2=interface_YACC<<Greater(Succ(x),Zero)->True>>;;
(*** succ(x) > succ(y) -> x > y ***)
let reg3=interface_YACC<<Greater(Succ(x),Succ(y))->Greater(x,y)>>;;
(*** f(0) -> 0 ***)
let reg4=interface_YACC<<F(Zero) -> Zero>>;;
(*** Ri est l'ensemble des regles inconditionnelles ***)
let Ri=[reg1;reg2;reg3;reg4];;
(* ---------- Re`gles conditionnelles-----------------------*)
(*** x > 0 = true ==> f(x) -> 0 ***)
let reg5=interface_YACC<<(Greater(x,Zero)=True)=>F(x)->Zero>>;;
let Rc=[reg5];;
(* ---------- Fonctions a` tester --------------------------*)
(*** F : nat --> nat ***)
let Fonc_a_tester=[ ["F";"nat";"nat"] ];;
(*---------- Fonctions supposees Completes ----------------*)
let Fonc_complet =[ ["Greater";"nat";"nat";"bool"] ];;
(* ---------- Constructeurs ---------------------------------*)
(***
    Pour le type nat on a les Constructeurs suivants :

              - Zero : --> nat
              - Succ : nat --> nat
  ***)
let Const=[ [["nat"];["Zero"];["Succ";"nat"]] ];;
(* ---------- Appel de la fonction "test_specification" --------*)
let init=rappel Rc Ri
 in
  test_specification (Rc,Ri,Const,Fonc_a_tester,Fonc_complet,false);;
```

```
================================================================
>>>      TEST DE COMPLETUDE D'UNE SPECIFICATION              <<<
================================================================

              >>> Rappel des regles introduites <<<

      ___________________________________
      >>> Regles inconditionnelles <<<
      ___________________________________

{Greater(Zero,x1) --> False }
{Greater(Succ(x1),Zero) --> True }
{Greater(Succ(x1),Succ(x2)) --> Greater(x1,x2) }
{F(Zero) --> Zero }

>>> Tapez  Return pour continuer <<<

      ___________________________________
      >>> Regles conditionnelles <<<
      ___________________________________

{ (Greater(x1,Zero)=True) ==> F(x1) --> Zero }

>>> Tapez  Return pour continuer <<<

        ____________________________________________
        >>> Affichage de l'arbre de motif de F  <<<
        ____________________________________________

F(x1)
  F(Zero)  ok
  F(Succ(x2))  ok

>>> Tapez Return pour continuer  <<<

        ____________________________________
        >>>  La specification  est  complete    <<<
        ____________________________________

Consultation de la liste definitives des regles <O/N> : O

      ____________________________________________
      >>> Consultation des regles inconditionnelles <<<
      ____________________________________________

{Greater(Zero,x1) --> False }
{Greater(Succ(x1),Zero) --> True }
{Greater(Succ(x1),Succ(x2)) --> Greater(x1,x2) }
{F(Zero) --> Zero }
```

>>> Tapez Return pour continuer <<<

>>> Consultation des regles conditionnelles <<<

{ (Greater(x1,Zero)=True) ==> F(x1) --> Zero }

>>> Tapez Return pour continuer <<<

```
%% 4) listes trie'es avec un ope'rateur boole'en qui teste si la liste est
trie'e ou non (voir chapitre2, page 63). %%

%% Les re`gles sont partage'es en deux ensembles : les re`gles sans
pre'conditions, et les re`gles conditionnelles %%

(* ---------- Re`gles inconditionnelles----------------------*)

(*** 0 =< x -> true ***)

let reg1=interface_YACC<<Inf(Zero,x)->True>>;;

(*** succ(x) =< 0 -> false ***)

let reg2=interface_YACC<<Inf(Succ(x),Zero)->False>>;;

(*** succ(x) =< succ(y) -> x =< y ***)

let reg3=interface_YACC<<Inf(Succ(x),Succ(y))->Inf(x,y)>>;;

(*** ordered([]) -> true ***)

let reg4=interface_YACC<<Ordered(Lvide)->True>>;;

(*** ordered(cons(x,[])) -> true ***)

let reg5=interface_YACC<<Ordered(Cons(x,Lvide))->True>>;;

(*** Ri est l'ensemble des regles inconditionnelles ***)

let Ri=[reg1;reg2;reg3;reg4;reg5];;

(* ---------- Re`gles conditionnelles----------------------*)

(*** x =< y = true ==> ordered(cons(x,cons(y,u))) -> ordered(cons(y,u)) ***)

let reg6=interface_YACC<<(Inf(x,y)=True)=>Ordered(Cons(x,Cons(y,u)))->
Ordered(cons(y,u))>>;;

(*** x =< y = false ==> ordered(cons(x,cons(y,u))) -> false ***)

let reg7=interface_YACC<<(Inf(x,y)=False)=>Ordered(Cons(x,Cons(y,u)))->
False>>;;

let Rc=[reg6;reg7];;

(* ---------- Fonctions a` tester ---------------------------*)

(*** Ordered : list --> bool ***)

let Fonc_a_tester=[ ["Ordered";"list";"bool"] ];;

(*---------- Fonctions supposees Completes -----------------*)

let Fonc_complet =[ ["Inf";"nat";"nat";"bool"] ];;

(* ---------- Constructeurs ---------------------------------*)

(***
    Pour le type nat on a les Constructeurs suivants :
```

```
                - Zero : --> nat
                - Succ : nat --> nat
  ***)

(***
    Pour le type list on a les Constructeurs suivants :

                - Lvide : --> list
                - Cons : nat * list --> list
  ***)


let Const=[ [["nat"];["Zero"];["Succ";"nat"]] ;
[["list"];["Lvide"];["Cons";"nat";"list"]] ];;

(* ------------ Appel de la fonction "test_specification" --------*)

let init=rappel Rc Ri
 in
  test_specification (Rc,Ri,Const,Fonc_a_tester,Fonc_complet,false);;

%% Le test de convertibilite' se de'roule de la manie`re suivante : %%

                    >>> Rappel des regles introduites <<<

        _____________________________________
        >>> Regles inconditionnelles <<<
        _____________________________________

 {Inf(Zero,x1) --> True }
 {Inf(Succ(x1),Zero) --> False }
 {Inf(Succ(x1),Succ(x2)) --> Inf(x1,x2) }
 {Ordered(Lvide) --> True }
 {Ordered(Cons(x1,Lvide)) --> True }

 >>> Tapez  Return pour continuer <<<


        _____________________________________
        >>> Regles conditionnelles <<<
        _____________________________________

 { (Inf(x1,x2)=True) ==> Ordered(Cons(x1,Cons(x2,x3))) -->
Ordered(cons(x2,x3)) }
 { (Inf(x1,x2)=False) ==> Ordered(Cons(x1,Cons(x2,x3))) --> False }

 >>> Tapez  Return pour continuer <<<


          ___________________________________________________
          >>> Affichage de l'arbre de motif de Ordered  <<<
          ___________________________________________________

 Ordered(x1)
   Ordered(Lvide)  ok
   Ordered(Cons(x2,x3))
     Ordered(Cons(x2,Lvide))  ok
     Ordered(Cons(x2,Cons(x1,x4)))  ok
```

```
      (Inf(x2,x1)=True)::Ordered(cons(x1,x4))
      (Inf(x2,x1)=False)::False

>>> Tapez Return pour continuer  <<<


          >>>  La specification  est  complete    <<<


Consultation de la liste definitives des regles <O/N> :O


      >>> Consultation des regles inconditionnelles <<<


{Inf(Zero,x1) --> True }
{Inf(Succ(x1),Zero) --> False }
{Inf(Succ(x1),Succ(x2)) --> Inf(x1,x2) }
{Ordered(Lvide) --> True }
{Ordered(Cons(x1,Lvide)) --> True }

>>> Tapez  Return pour continuer <<<


      >>> Consultation des regles conditionnelles <<<


 { (Inf(x1,x2)=True) ==> Ordered(Cons(x1,Cons(x2,x3))) -->
Ordered(cons(x2,x3)) }
 { (Inf(x1,x2)=False) ==> Ordered(Cons(x1,Cons(x2,x3))) --> False }

 >>> Tapez Return pour continuer <<<
```

Résumé

Définir un type abstrait algébrique par des axiomes conditionnels offre davantage souplesse et de simplicité que dans un cadre classique, sans préconditions. Ceci nous amenée à étudier la théorie conditionnelle de termes dans le but d'obtenir une méthode de décision fondée sur la réécriture de termes. Cependant, cette étude dans le contexte conditionnel soulève des problèmes délicats, principalement liés à la difficulté de formaliser la sémantique sous-jacente à la théorie conditionnelle et de définir des modèles.

L'objectif de cette thèse est d'étudier les propriétés des systèmes de réécriture conditionnelle. Notre contribution centrale est la mise en œuvre de deux tests de confluence sur les termes clos. Le premier test est établi pour des systèmes de réécriture *hiérarchiques*, systèmes construits par strates successives, la précondition d'une règle utilisant exclusivement des opérateurs appartenant aux strates inférieures. Le second test de confluence est établi pour des systèmes *décroissants*, dont les règles sont définies avec un principe de récurrence structurelle sur leurs composantes. Ces deux classes de systèmes ont été définies et utilisées dans des travaux ultérieurs par divers auteurs pour éviter d'engendrer des dérivations infinies de réécriture.

L'originalité du travail est l'utilisation d'un principe de raisonnement par cas, et la possibilité de réécrire une équation conditionnelle aussi bien dans sa partie conséquence que dans sa précondition. Ce processus de simplification s'est révélé souple et puissant et permet de traiter un certain nombre d'applications.

Nous abordons également dans notre étude un aspect inhérent à la théorie conditionnelle : le comportement d'une spécification conditionnelle par rapport aux booléens. Ce comportement est fidèle dès lors que les propriétés de consistance et de complétude par rapport à la sorte booléenne sont garanties, assurant de façon intuitive qu'il n'y a ni ajout de booléens, ni confusion entre les deux constantes de vérité *true* et *false*.

La propriété de complétude d'une spécification conditionnelle est également abordée. Cette propriété étant indécidable même dans la théorie classique, sans préconditions, nous proposons un algorithme fondé sur le raisonnement par cas pour tester si une spécification conditionnelle est complète par rapport à l'ensemble de ses constructeurs. Cet algorithme combine une analyse structurelle par le calcul d'un ensemble de motifs et une analyse par cas par le biais d'un ensemble de conditions utilisées dans la réécriture. Cette condition suffisante de test permet d'appréhender une sous-classe relativement étendue de définitions conditionnelles.