Déduction avec contraintes et simplification dans les théories équationnelles

Christelle Scharff

▶ To cite this version:

Christelle Scharff. Déduction avec contraintes et simplification dans les théories équationnelles. Autre [cs.OH]. Université Henri Poincaré - Nancy 1, 1999. Français. NNT : 1999NAN10271 . tel-01747107

HAL Id: tel-01747107
https://hal.univ-lorraine.fr/tel-01747107v1

Submitted on 29 Mar 2018

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

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

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

99/1569

S: N 99/271 B

Université Henri Poincaré Nancy 1

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAE + M

DEDUCTION AVEC CONTRAINTES ET SIMPLIFICATION DANS LES THEORIES EQUATIONNELLES

THÈSE

présentée et soutenue publiquement le 24 Septembre 1999

pour l'obtention du

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

par

Christelle Scharff

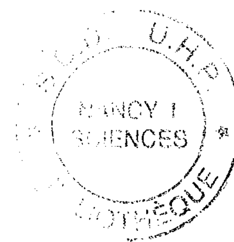
Composition du jury

Président :	Adam Cichon, Professeur, Université Henri Poincaré, Nancy 1, France
Rapporteurs :	Siva Anantharaman, Professeur, Université d'Orléans, France Maria Paola Bonacina, Professeur, Université d'Iowa, USA François Lamarche, Directeur de recherche, INRIA, France
Examinateurs :	Claude Kirchner, Directeur de recherche, INRIA, France Christopher Lynch, Professeur, Université de Clarkson, USA

BIBLIOTHEQUE SCIENCES NANCY 1
D 095 130085 7

Labora... en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.


U.H.P. NANCY I SCIENCES BIBLIOTHÈQUE

Avant-propos

Par soucis de vulgarisation, nous dirons d'abord que cette thèse est une thèse en intelligence artificielle. L'assemblage de ces deux termes en opposition « intelligence » et « artificielle » est adapté pour montrer comment créer un artifice, une illusion pour nous faire croire que les machines seraient effectivement intelligentes. On en est très loin et la pensée générale a raison de dire que l'ordinateur ne fait que ce qu'on lui dit de faire. Il n'y a alors aucune raison de penser que la machine pourrait remplacer l'homme, ses aptitudes seront toujours partielles. L'intelligence artificielle fait partie de notre vie de tous les jours. Les exemples sont nombreux. Elle est là pour suggérer un diagnostic médical, pour localiser les pièces défaillantes d'une voiture, pour jouer aux échecs et battre le champion du monde Garry Kasparov, pour guider un robot et lui faire trouver la sortie d'un labyrinthe, pour reconnaître des séquences particulières de la molécule d'ADN, pour démontrer (prouver) quelques théorèmes inédits ou des conjectures... L'ordinateur ne s'est pas doté seul de ces pouvoirs. C'est le travail en amont des informaticiens et en particulier des chercheurs en informatique qui lui a permis de développer des capacités cognitives dans des domaines précis, des capacités de reconnaissances des formes et de la parole, des capacités de vision, de planification, de jeux, des capacités de raisonnement... Pour ce faire, les techniques utilisées font intervenir les opérations ordinaires de l'informatique (opérations sur des textes, des sons, des images...), mais elles font également appel à la logique, aux mathématiques, à l'algorithmique...

L'intelligence artificielle doit faire face à des difficultés dues à la capacité des machines actuelles, à la mise au point de formalismes adéquates mais principalement aux problèmes de la combinatoire, c'est-à-dire au nombre faramineux de cas possibles et de branches à explorer avant de trouver une solution. Un exemple significatif est celui du nombre de partie d'échecs jouables qui est supérieur au nombre de particules de l'univers à savoir 10^{50}. Les chercheurs en intelligence artificielle ont alors mis au point des heuristiques basées sur la connaissance des domaines considérés en collaboration avec des spécialistes de ces domaines pour explorer les solutions les plus prometteuses et ainsi guider les machines. Les heuristiques sont donc des règles approximatives mises au point par expérimentations et raffinements successifs, qui aident à trouver les solutions de certains problèmes dans certains cas et sans réelle certitude. Toutefois, leur utilisation est très répandue et elle s'impose pour simuler les activités intellectuelles des ordinateurs.

La démonstration (preuve) automatique de théorèmes faisait d'abord partie de l'intelligence artificielle mais elle s'en est détachée lorsque l'on a commencé à disposer de formalismes, d'algorithmes et de procédures spécifiques et ainsi de réponses certaines à certains problèmes en un temps raisonnable.

Un programme de démonstration automatique (prouveur de théorèmes) donne à un ordinateur le pouvoir de raisonner, c'est-à-dire de construire des preuves à partir de propositions supposées connues.

Supposons, par exemple, que l'on veuille prouver le syllogisme d'Aristote « Les hommes sont mortels, Socrate est un homme, donc Socrate est mortel ». « Les hommes sont mortels » et

« Socrate est un homme » sont nos hypothèses initiales que nous appelons les axiomes. Nous voulons montré que ces deux propositions impliquent la proposition « Socrate est mortel ». On modélise les axiomes « Les hommes sont mortels » et « Socrate est un homme » par les clauses $homme(x) \Rightarrow mortel(x)$ qui se lit « si x est un homme alors x est mortel » où x est une variable représentant n'importe quel homme et $homme(Socrate)$ et la conclusion à prouver « Socrate est mortel » est formalisée par la proposition $mortel(Socrate)$. Pour arriver à ce syllogisme, on utilise le principe d'unification, qui consiste à remplacer une variable par un objet plus déterminé et la stratégie de résolution de J.A. Robinson, qui simule un raisonnement par l'absurde où pour démontrer qu'une proposition P, la résolution consiste à prouver que la négation de P ajouté à l'ensemble des axiomes impliquent une contradiction. Dans notre cas, la preuve s'articule alors comme suit. La négation de la conclusion que nous voulons prouver est $\neg mortel(Socrate)$. En utilisant les axiomes $homme(x) \Rightarrow mortel(x)$, où x est unifié (instancié) par $Socrate$ pour donner $homme(Socrate) \Rightarrow mortel(Socrate)$ (équivalente à $\neg homme(Socrate) \vee mortel(Socrate)$) et $\neg mortel(Socrate)$, alors nous obtenons $\neg homme(Socrate)$, ce qui est contradictoire avec l'hypothèse $homme(Socrate)$. La preuve apparaît comme une suite de propositions articulées par des déductions à partir des axiomes et également des propositions déjà démontrées. Cet exemple nous montre comment on est passé du langage naturel à une représentation formelle utilisable pour donner une preuve. Il nous informe également sur comment fonctionne notre cerveau et nous permet de repérer certains problèmes que rencontrent les ordinateurs. Parmi les déductions possibles, nous en isolons quelques unes, nous en écartons d'autres et nous maîtrisons l'explosion combinatoire. Nous raisonnons spontanément modulo la commutativité (et l'associativité) des propositions et dans les propositions. L'ordinateur n'a pas cette faculté de raisonnement naturellement, l'intelligence artificielle tente de l'imiter. Par exemple, si on demande à un programme de déduction de montrer que « 1+1=2 », l'ordinateur ne voit pas que ce problème peut se résoudre par un simple calcul mais il chercher à démontrer cette proposition en utilisation potentiellement tous les axiomes des mathématiques.

Parmi les applications de la démonstration automatique, nous citerons le domaine de la vérification de logiciels. Quelques échecs retentissants et lourds de conséquences comme par exemple, la panne du réseau téléphonique aux USA en 1989 ou la destruction du premier exemplaire de la fusée Ariane 5 en 1996 ou le bogue du processeur Pentium Pro d'Intel ont convaincu de la nécessité de vérifier certains logiciels. Parmi les systèmes dont la vérification est souhaitable, on citera les systèmes critiques, où une erreur peut avoir des conséquences catastrophiques (nucléaire, aérospatiale, transports, informatique bancaire...), les systèmes distribués particulièrement difficile à maîtriser par le cerveau humain et les systèmes réactifs, qui répondent en permanence à leur environnement (systèmes d'exploitation, matériel informatique, protocoles de communication et de télécommunication...). A l'heure actuelle, la démonstration automatique est encore loin de pouvoir répondre à toutes les questions de vérification car sa mise en oeuvre est lourde et compliquée et il faut certes vérifier mais dans un second temps, il faudra doter les machines du pouvoir de corriger automatiquement les erreurs détectées.

Le sujet d'une thèse est souvent l'étude d'un problème pointu et de l'aspect infime d'un domaine. Il est de ce fait difficile à expliquer, difficile à vulgariser. Il apparaît donc plus facile de présenter le contexte d'une thèse plutôt que le sujet en lui-même. Mais, la finalité de cette goutte d'eau dans l'océan de la connaissance est d'être intégrée, d'appeler à des comparaisons, de créer des synergies dans la communauté et d'inspirer et de faire naître de nouvelles idées... En une phrase, une thèse n'est jamais une fin en soit, seulement l'état d'une réflexion à un instant T...

Remerciements

Le développement de cette thèse a été ma préoccupation première ces quatre dernières années.

Mes motivations sous-jacentes ont mijoté dans mon esprit pendant tout mon cursus universitaire à l'Université Henri Poincaré de Nancy.

Je tiens d'abord à remercier Jocelyne Rouyer, qui a été mon professeur et qui est devenue par la suite ma collègue en enseignement à l'Ecole Supérieure d'Informatique et d'Automatique de Lorraine. Elle m'a donné le goût de la rigueur et du formel. Je remercie également Denis Lugiez, qui m'a motivée pour faire un premier stage dans l'équipe Prothéo, où s'est déroulée ensuite ma thèse. Merci à Jean-Pierre Finance d'avoir influencé ma vision de l'informatique. Je ne remercierai jamais assez Adam Cichon d'avoir accepté d'être examinateur de cette thèse et le président de mon jury de thèse. J'ai été très heureuse d'être sa première ex-étudiante de Nancy à soutenir une thèse. Je les remercie particulièrement tous les quatre, car, c'est eux qui m'ont motivée à entreprendre une thèse.

Maria-Paolo Bonacina m'a honorée en acceptant d'être rapporteur de cette thèse. Elle n'a pas hésité à venir spécialement des Etats-Unis pour participer à la soutenance. Les discussions que nous avons eues ainsi que ses remarques sur le document m'ont été très précieuses. Je la remercie également pour la confiance qu'elle a eue en mon travail.

Siva Anantharaman m'a fait l'honneur d'être rapporteur de cette thèse. J'ai été particulièrement touchée par ses commentaires sur le manuscrit et par l'intérêt de ses questions.

Je remercie aussi vivement Francois Lamarche pour avoir accepté d'être mon rapporteur interne et pour son regard extérieur.

Je tiens à remercier particulièrement Claude Kirchner, mon directeur de thèse, pour m'avoir conseillée et aidée tout au long de cette thèse. Il m'a permis de faire ma thèse dans les meilleures conditions. Son ouverture d'esprit, sa culture et ses conseils bibliographiques m'ont été fortement utiles. Nos discussions ont été très enrichissantes.

Je tiens à remercier tout spécialement Christopher Lynch, qui a co-encadré cette thèse. Il a su me communiquer sa passion de la recherche. Nos très nombreuses discussions scientifiques ont influencé ma manière de voir. Il a su également me motiver lorsque j'en avais besoin. J'ai particulièrement apprecié sa disponibilité, sa confiance, et ses conseils. Bien qu'étant à plusieurs milliers de kilométres de Nancy, nos nombreux échanges e-mails nous ont permis une collaboration solide et très enrichissante. C'est bien à lui que je dois ma passion pour la recherche et je tiens à le remercier très sincèrement et amicalement.

Merci à Carlos Castro, Peter Borovanský et Pierre-Etienne Moreau pour l'aide qu'ils m'ont apportée dans la réalisation de l'application ELAN, *ECC*.

J'en profite aussi pour remercier tous mes collègues et éléves de l'École Superieure d'Informatique et d'Automatique de Lorraine et de l'Université Nancy 2 pour avoir rendu passionnant mon travail d'enseignante. Je remercie particulièrement Jeanine Souquiéres, Dominique Mery et Francis Alexandre pour m'avoir aidée à faire mes premiers pas dans le monde de l'enseignement.

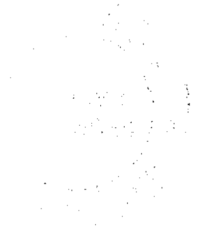
Merci à tous mes amis du laboratoire de Nancy.

Je remercie également les re-lecteurs anonymes de cette thèse pour leurs commentaires. Ils m'ont permis d'améliorer la qualité de ce manuscrit.

Enfin, je dédie cette thèse à ma famille pour m'avoir soutenue à chaque instant dans les bons moments comme dans les moments plus difficiles. Je dédie également cette thèse à mes amis de Laxou, de Nancy, de Moselle, de France, des Etats-Unis et d'Allemagne.

J'ai essayé de me souvenir de tous ceux qui ont contribué à cette thèse, mais je n'ai sans aucun doute pas réussi à les citer tous.

Christelle Scharff
Nancy
Septembre 1999



« *Sans reflexion préalable, les meilleures techniques d'écriture des applications vous permettent seulement d'arriver plus rapidement au désastre.* », Decline & Fall of the American Programmer, *Edward Yourdon*

A mon père. A ma mère.

Sommaire

Liste des figures **xi**

Liste des tables **xiii**

Introduction **1**

Chapitre 1
Cadre de travail **9**

1.1 Termes, substitutions, égalités, clauses 10
1.2 Ordres 14
1.3 Langage de contraintes 17
1.4 Résolution de contraintes 19
1.5 Formules contraintes 22
1.6 ELAN 22

Chapitre 2
Recherche de preuve et déduction en logique du premier ordre avec égalité **31**

2.1 Stratégies de recherche de preuve 32
2.2 La saturation 35
2.3 Procédures de complétion 37
2.4 Résolution et paramodulation 66
2.5 Parallélisation des procédures de recherche de preuve et de déduction 72

Chapitre 3
Une procédure de complétion close concurrente **79**

3.1 Introduction 80
3.2 Complétion basique des graphes SOUR 81
3.3 Principes de la complétion SOUR concurrente 90
3.4 Implantation des règles d'inférence de la complétion SOUR close concurrente . . 98

3.5 Calcul concurrent de l'unification 104
3.6 Calcul concurrent de l'orientation 112
3.7 Le datage de l'information 133
3.8 Terminaison de l'algorithme distribué 138
3.9 Résultats de correction et complétude 140
3.10 Expérience pratique : l'implantation CWD 147
3.11 Conclusion 149

Chapitre 4
Complétion basique avec simplification E-cycle 151

4.1 Introduction 151
4.2 Concepts de base 153
4.3 Graphe de dépendance et simplification E-cycle 158
4.4 BCES est complet 178
4.5 Comparaison avec la simplification basique 186
4.6 Expérience pratique en ELAN : l'implantation ECC 189
4.7 Stratégies de simplification incomplètes 193
4.8 Conclusion 196

Chapitre 5
Complétion basique modulo avec simplification 199

5.1 Introduction 199
5.2 Contraintes de contraintes 201
5.3 Motivations 202
5.4 Le système d'inférence $BCICS_m$ 203
5.5 $BCICS_m$ est correct et complet 213
5.6 Conclusion 217

Conclusion 219

Bibliographie 223

Liste des figures

1.1 Règles d'inférence de la logique équationnelle 12

2.1 Transitions pour la complétion de Knuth-Bendix 40
2.2 La complétion : Pour transformer une preuve en preuve en vallée 42

3.1 Etapes de la construction du graphe SOUR de départ de $E = \{f(g(a),g(a)) \approx h(g(a))\}$ 82
3.2 Sémantique des graphes SOUR 83
3.3 Sémantique des graphes SOUR - Exemple 84
3.4 Transformation SUR - cas non clos 85
3.5 Transformation RUR - cas non clos 86
3.6 Complétion de $f(f(x)) \approx g(f(x))$ 87
3.7 Transformation SUR - cas clos 88
3.8 Transformation RUR - cas clos 89
3.9 Transformation RUR-rhs - cas clos 89
3.10 Le processus maître et les processus fils 90
3.11 Graphe SOUR initial pour $E = \{f(a,a) \approx b,\ a \approx c\}$ 93
3.12 Calcul de la contrainte équationnelle c_0 de l'arc d'unification U_0 104
3.13 Implantation graphique de LPO - Cas 1 113
3.14 Implantation graphique de LPO - Cas 2 113
3.15 Implantation graphique de LPO - Cas 3 114
3.16 Preuve de l'utilité des étiquettes de temps 133
3.17 Correction : Traitement d'une configuration RUR-rhs 142
3.18 Correction : Traitement d'une configuration RUR 142
3.19 Correction : Traitement d'une configuration SUR 143
3.20 Correction : Aucun cycle d'arc S n'est créé 144

4.1 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 1 168
4.2 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 2 169
4.3 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 3 170
4.4 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 4 171
4.5 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 5 172

4.6 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 6 173
4.7 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 7 174
4.8 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 8 175
4.9 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - Etape 9 176
4.10 Graphe de dépendance final de la complétion basique avec simplification E-cycle de $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ 177
4.11 Graphe de dépendance de la complétion basique avec simplification E-cycle de $E = \{g(x) \approx f(x)\ (1), g(a) \approx b\ (2), h(f(a)) \approx b\ (3)\}$ 189
4.12 Graphe de dépendance pour la complétion basique avec simplification E-cycle de $E = \{h(f(f(a))) \approx b\ (1),\ f(x) \approx x\ (2)\}$ 196

Liste des tables

2.1 Grain de parallélisme de quelques systèmes de preuve automatique de théorèmes et de déduction automatique . . . 77
2.2 Stratégie de recherche de preuve et de déduction employée dans quelques systèmes de preuve automatique de théorèmes et de déduction automatique . . . 77
2.3 Parallélisme de partitionnement et compétitif de quelques systèmes de preuve automatique de théorèmes et de déduction automatique . . . 78

3.1 Algorithme suivi par les processus fils . . . 97
3.2 Expérience pratique avec CWD . . . 149

Résumé

Dans le cadre des méthodes de recherche de preuve et de déduction en logique du premier ordre avec égalité, la volonté de réduire l'espace de recherche et de résoudre de plus en plus de problèmes ont suggéré l'usage du parallélisme, de contraintes et de stratégies de contraction. L'objectif de cette thèse est l'exploration de ces trois voies et de leurs interactions dans le cas de la Complétion Basique (modulo).

Nous proposons une nouvelle procédure de complétion close de grain fin basée sur l'utilisation des graphes SOUR. Chaque noeud du graphe est un processus représentant un terme et les arcs sont des canaux de communication. La complétion est réalisée de façon complètement asynchrone par coopération entre les processus sans mémoire globale ni contrôle global et utilise une stratégie de contraction, ce qui est l'un des principaux avantage. Notre méthode est implantée dans CWD en C et avec PVM.

La Complétion Basique est rarement complète en combinaison avec des stratégies de contraction. Elle était connue complète seulement en combinaison avec la Simplification Basique. Nous avons développé une nouvelle stratégie de simplification, la Simplification E-cycle, basée sur l'utilisation d'un graphe de dépendance schématisant les dépendances entre égalités qui permet plus de simplifications et est plus simple à comprendre que la Simplification Basique. Elle est implantée dans ECC en ELAN.

La Simplification Basique n'est pas efficace pratiquement pour la Complétion Basique modulo. Nous proposons un nouveau système d'inférence pour la Complétion Basique modulo avec simplification, qui ne requiert pas de résoudre les contraintes et réduit l'écart entre la théorie et la pratique. Pour la complétude de notre système, nous avons besoin de réaliser des inférences dans les contraintes, ce qui prend le contre-pied des approches actuelles de la recherche de preuve et de la déduction avec contraintes.

Mots-clés: Démonstration et déduction automatique, complétion, contraintes, théories équationnelles, stratégies de contraction et de simplification.

Abstract

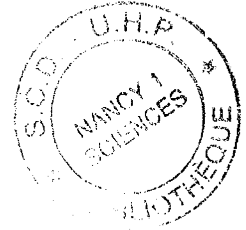
The subject of this thesis is the study of theorem proving and deduction methods in first order logic with equality. An example is the case of Basic Completion (modulo). Contraction rules are crucial in completion. This thesis investigates three ways for improving efficiency of completion : use of concurrency, of constraints and contraction strategies, and studies the interaction between these three elements.

We first propose a concurrent ground completion procedure based on the SOUR graph data structure convenient for a fine-grained approach where each node is a process representing a term and edges are communication links. Completion is done completely asynchronously by cooperation between processes without global memory or global control and uses a contraction-based deduction strategy, which is a real asset. We implement our method in CWD in C and with PVM.

Our second contribution was mainly motivated by the facts that Basic Completion combined with contraction strategies is not always complete. Basic Completion was only known to be complete with Basic Simplification for Basic Completion. We develop a new simplification strategy, E-cycle Simplification, based on the construction of a graph for dependencies between equalities that permits more simplifications and is simpler to understand than Basic Simplification and implement it in ECC in ELAN.

Basic Simplification is not practically efficient for Basic Completion modulo. We give a new inference system for Basic Completion modulo with simplification that is defined concretely and that does not require solving constraints. For completeness, we need to allow inferences into constraints, that takes an original way with respect to the current approaches of the Basic Completion community.

Keywords: Automatic deduction, theorem proving, completion, constraints, equational theories, contraction strategies, simplification strategies.



Introduction

La mise au point de programmes de démonstration automatique modernes et efficaces s'appuie sur la combinaison de trois éléments principaux : une bonne théorie, des heuristiques bien choisies mais aussi l'utilisation de techniques d'implantation efficaces (Nieuwenhuis, 1999). Ces trois éléments sont en interaction permanente.

La théorie avance dans diverses directions et produit toujours de nouvelles idées, mais rendre ces résultats théoriques applicables et les intégrer dans des prouveurs nécessitent quelques années. Du point de vue pratique, des structures de données d'indexage réutilisables dont les comportements en temps et en espace sont bien connus ont été développées pour les opérations standards comme le filtrage et l'unification.

Nous nous intéressons dans cette thèse aux challenges théoriques et pratiques concernant la construction de prouveurs de théorèmes basés sur la *Paramodulation* et la *Complétion* vues comme des méthodes par saturation.

La *Paramodulation* est une méthode de recherche de preuve réfutationnelle pour la logique du premier ordre avec égalité initialement présentée dans (Robinson et Wos, 1969). La *Complétion* est un cas particulier de la *Paramodulation*, où seul le prédicat d'égalité est utilisé. La Complétion et la Paramodulation ont subi la même évolution. La première forme de complétion a été proposée dans (Knuth et Bendix, 1970). La règle d'inférence de paramodulation est une règle très prolifique dont on doit contrôler l'application si l'on veut réduire la taille de l'espace de recherche. La *Paramodulation* a été raffinée de différentes manières au cours du temps. D. Brand a montré que l'utilisation d'axiomes de réflexivité fonctionnelle est inutile (Brand, 1975). G. Peterson a montré que les inférences à des positions de variables peuvent être évitées (Peterson, 1983). Ces deux résultats permettent de supprimer un nombre considérable d'inférences. Une seconde classe de restriction consiste à restreindre l'application des règles d'inférence à l'aide d'ordres sur les termes et les atomes (Hsiang et Rusinowitch, 1991). De plus, de nombreux mécanismes sont introduits pour simplifier et supprimer des clauses. Parmi ces mécanismes, on distingue les *stratégies de contraction* dont font partie les *stratégies de simplification.*

Un autre raffinement important est l'utilisation des contraintes (Kirchner, Kirchner et Rusinowitch, 1990), ce qui a donné naissance dans le cas de la théorie vide à la *Paramodulation Contrainte* (Vigneron, 1994a; Bachmair, Ganzinger, Lynch et Snyder, 1995; Nieuwenhuis et Rubio, 1995a). L'unificateur le plus général est sauvegardé dans une contrainte au lieu d'être appliqué à la conclusion de l'inférence comme c'est le cas dans la règle de *Paramodulation.* Les contraintes sauvegardent des restrictions d'unification mais également des restrictions d'ordre. Les deux principaux avantages sont d'une part, que les restrictions d'unification et d'ordre sont héritées lors d'une inférence et qu'une inférence ne satisfaisant pas ces critères est inutile, et d'autre part que la *Paramodulation Contrainte* contrôle l'expansion de l'espace de recherche en interdisant les inférences dans les termes provenant d'inférences réalisées précédemment dans le processus. On distingue la *Paramodulation Contrainte* de la *Paramodulation Basique* par le fait que dans le cas « basique », les contraintes sont des substitutions. Des expériences pratiques

montrent que les restrictions contrainte et basique épargnent le prouveur de théorème d'une charge non négligeable de travail, les contraintes factorisant le travail en un sens. Ces restrictions concernant l'utilisation des contraintes sont facilement intégrables dans des prouveurs existants, il suffit de marquer les positions à partir desquelles les inférences sont interdites. Par exemple, une modification d'OTTER (McCune, 1994) a été réalisée dans (Bachmair, Ganzinger, Lynch et Snyder, 1992) pour pouvoir réaliser de la *Paramodulation Basique*.

La *Paramodulation* en présence des axiomes d'associativité et de commutativité peut générer un nombre infini de versions permutées de la même clause. Pour résoudre ce problème, il convient de traiter toutes ces clauses ensemble. Cette remarque a donné lieu à la notion de théorie équationnelle intégrée (*built-in*) : les axiomes de la théorie ne sont pas impliqués dans les inférences et les calculs d'unification et de filtrage sont réalisés dans la théorie équationnelle considérée. Travailler modulo permet de séparer la déduction (les règles d'inférence) qui est en général un processus indécidable de la partie calcul (unification, filtrage) qui est décidable dans certaines théories (Dowek, Hardin et Kirchner, 1998). On obtient alors la *Paramodulation modulo une théorie équationnelle* appelée également *Paramodulation modulo* (Peterson et Stickel, 1981). Elle capture le besoin d'utiliser de plus en plus de connaissances sur les domaines des problèmes considérés (associativité, commutativité, les égalités, la logique du premier ordre...). Intégrer les axiomes entraîne une perte d'expressivité solutionnée par le recours à des extensions des clauses construites à partir des axiomes. En 1990, la *Paramodulation modulo* a donné des résultats encourageants en prouvant les égalités de Moufang dans les anneaux alternatifs (Anantharaman et Hsiang, 1990). Ces égalités avaient été prouvées « à la main » par R. Moufang en 1933, et soixante ans plus tard une preuve automatique était possible.

Une inférence de paramodulation dans le cas de la *Paramodulation modulo* déduit une clause par unificateur plus général du problème considéré. Par exemple, une inférence impliquant le problème d'unification $x + x + x + x =^?_{AC} \; y_1 + y_2 + y_3 + y_4$, où $+$ est un symbole AC, x, y_1, y_2, y_3 et y_4 sont des variables, ayant 34 359 607 481 unificateurs plus généraux minimaux (Domenjoud, 1992), entraîne la création de 34 359 607 481 clauses. La *Paramodulation avec contraintes modulo l'associativité et la commutativité* et plus généralement la *Paramodulation avec contraintes modulo* (Vigneron, 1994a; Nieuwenhuis et Rubio, 1997) a l'intérêt de ne pas résoudre cette contrainte mais de la sauver et de ne déduire qu'une seule clause. La *Paramodulation Contrainte modulo l'associativité et la commutativité* est considérée comme l'élément fondamental qui a permis de prouver la conjecture de Robbins (McCune, 1997b), conjecture qui datait des années trente.

Nous avons pour lors seulement mentionné le fait que des clauses peuvent être simplifiées ou supprimées de l'espace de recherche, mais la contraction, la simplification et la suppression sont cruciales dans toutes les formes de paramodulation. Les techniques de simplification et de suppression s'appuient sur la notion abstraite de *redondance*. Intuitivement, une clause est dite *redondante* si elle est impliquée par des clauses plus petites par rapport à un ordre donné et une inférence est redondante pour un système d'inférence $\mathcal{I}$ si elle utilise une clause redondante. Un ensemble est dit *saturé* si toutes les clauses qu'il contient ne peuvent intervenir que dans des inférences redondantes pour $\mathcal{I}$. La *saturation* est une procédure qui ajoute des clauses non redondantes à un ensemble de clauses et supprime les clauses redondantes pour $\mathcal{I}$. La saturation produit des ensembles saturés. La saturation est une technique qui a montré des avantages au niveau théorique et pratique. Elle a permis de prouver la complétude de certains systèmes d'inférence (avec des stratégies de contraction) ou de simplifier les preuves existantes. La complétude d'un système d'inférence garantit de trouver une preuve de toute clause vraie. Dans le cas de la saturation, la complétude est appelée *complétude réfutationnelle*, elle garantit qu'un modèle de l'ensemble saturé des clauses ne contenant pas la clause vide peut être construit. La *Paramodulation Basique*

modulo l'associativité-commutativité est prouvée complète dans (Vigneron, 1994b) en utilisant les arbres sémantiques transfinis, la complétude est également prouvée dans (Bachmair et al., 1995) avec la saturation. La Paramodulation Basique avec Simplification Basique a été prouvée complète dans (Bachmair et al., 1995; Nieuwenhuis et Rubio, 1992a). La renommée du prouveur SPASS (Weidenbach, Gaede et Rock, 1996) a convaincu de l'intérêt pratique de la saturation.

L'utilisation du parallélisme est également une voie à explorer pour contrôler l'expansion de l'espace de recherche. Les procédures de paramodulation et de complétion ont un grand potentiel de parallélisation (Suttner et Schumann, 1993; Bonacina et Hsiang, 1994; Bonacina, 1999b). La tendance actuelle est d'utiliser la coopération de prouveurs de théorèmes existants, pour créer des réseaux de prouveurs homogènes ou hétérogènes, chacun ayant ses propres compétences et ses propres forces (Denzinger et Dahn, 1998; Fuchs et Denzinger, 1998).

La règle d'inférence de paramodulation est une règle très prolifique, c'est pourquoi pour son utilisation en pratique les différentes possibilités d'amélioration de son application doivent être explorées en même temps. La recherche menée actuellement sur la paramodulation concerne essentiellement l'exploration de ces diverses possibilités dans le cas de la *Paramodulation avec contraintes (modulo)* vus ses résultats pratiques très prometteurs.

Les techniques actuelles de parallélisation et d'introduction de stratégies de contraction en recherche de preuve et en déduction doivent faire face à un certain nombre de problèmes. Le but de cette thèse est d'identifier les problèmes et de faire des propositions pour les résoudre dans le cas particulier de la Complétion. Le réel challenge de cette thèse est ainsi de proposer des techniques théoriques. Nous nous plaçons donc en premier lieu sur un plan théorique, dans le soucis d'améliorer les résultats actuels mais également sur un plan pratique, dans le sens où nos travaux sont illustrés par des implantations et où notre but est d'essayer de rétrécir le fossé qui existe entre théorie et pratique en proposant des techniques intéressantes pratiquement.

Parallélisme La parallélisation des procédures de preuve automatique de théorèmes ajoute aux problèmes propres du parallélisme d'autres difficultés à prendre en compte. Ces procédures font appel à un haut niveau d'abstraction et sont donc par essence complexes. Les données sont fortement dépendantes et l'espace de recherche est peu structuré et peut être infini. L'efficacité de ces procédures dépend du modèle de calcul mis au point et de l'adéquation du modèle par rapport à l'architecture choisie. De plus, l'efficacité dépend des stratégies de contraction utilisées. Ces stratégies sont difficiles à mettre au point dans le cas parallèle. Nous proposons dans ce document une procédure de complétion (close) concurrente adoptant une stratégie de contraction basée sur l'utilisation des graphes SOUR (Lynch et Strogova, 1998), dont la structure est adéquate pour une approche concurrente (Kirchner, Lynch et Scharff, 1996a).

Contraintes syntaxiques et stratégies de simplification L'utilisation simultanée de contraintes et de stratégies de contraction pose des problèmes de compatibilité vis à vis de la complétude. La *Paramodulation Basique* n'est pas complète si on utilise la *Simplification Standard* (Nieuwenhuis et Rubio, 1992a). La *Simplification Basique* est la seule stratégie de simplification complète pour la *Paramodulation Basique* (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995). Les conditions d'emploi de la règle de simplification basique sont très restrictives, c'est pourquoi pour pouvoir appliquer la règle, les contraintes sont souvent « réveillées », c'est-à-dire qu'on applique une partie de la contrainte à la clause (Nieuwenhuis et Rubio, 1995a), ce qui ne nous paraît pas satisfaisant. De nombreuses recherches dans le domaine des stratégies de contraction complètes pour la *Paramodulation Basique* sont encore à mener pour permettre

plus de simplifications. La stratégie de *Simplification E-cycle* (Lynch et Scharff, 1998; Lynch et Scharff, 1999b) que nous avons développée pour la Complétion Basique va dans ce sens, car elle autorise plus de simplifications que la *Simplification Basique* et elle est facilement intégrable aux prouveurs existants, il suffit de construire un graphe de dépendance pendant le processus de complétion et de l'utiliser pour savoir si l'on peut simplifier.

Modulo, contraintes et stratégies de simplification Les stratégies de simplification telles que la *Simplification Basique* ou la *Simplification E-cycle* ne peuvent être implantées de façon efficace dans le cas modulo. Elles impliquent de résoudre des contraintes équationnelles, ce qui est coûteux. Par exemple, le nombre d'unificateurs plus généraux des problèmes d'unification dans la théorie associative-commutative est doublement exponentiel par rapport à la taille des termes du problème (Kapur et Narendran, 1992; Domenjoud, 1992) et la théorie associative est infinitaire. Le réel défi apparaissant alors est de développer des stratégies de contraction qui n'obligent pas à résoudre les problèmes d'unification. Une étape dans ce sens est décrite dans ce document dans le chapitre 5 où nous proposons un système d'inférence pour la *Complétion Basique modulo* avec simplification, $BCICS_m$ (Lynch et Scharff, 1999a). Nous évitons de résoudre les contraintes en introduisant des contraintes de filtrage et d'ordre contraintes dont la satisfaisabilité est à tester et pour assurer la complétude du système d'inférence défini, nous réalisons des inférences dans les contraintes. Cette idée est tout à fait innovante et originale dans le domaine, car elle prend le contre-pied de toutes les approches actuelles basées sur l'utilisation des contraintes en permettant des inférences dans les contraintes.

Présentation

Premier chapitre : Cadre de travail

Cette thèse commence par la présentation des notations, concepts et notions de base nécessaires à la compréhension de ce travail.

Nous plaçons ainsi le cadre de travail de cette thèse en définissant ce que sont les termes, les substitutions, les égalités, les clauses, les ordres, les contraintes équationnelles et les formules contraintes. Nous nous intéressons aussi bien à l'aspect syntaxique qu'à l'aspect sémantique des éléments définis. Nous présentons également le langage ELAN (Borovanský, Cirstea, Dubois, Kirchner, Kirchner, Moreau, Ringeissen et Vittek, 1998) qui nous sert d'outil de prototypage pour nos applications, mais également de formalisme dans les chapitres suivants.

Deuxième chapitre : Recherche de preuve et déduction en logique du premier ordre avec égalité

Nous donnons dans ce chapitre un cadre général pour définir le domaine des preuves automatiques de théorèmes et de la déduction automatique. Nous proposons des définitions de systèmes d'inférence, de stratégies de recherche de preuve et de stratégies de déduction en accord avec la philosophie ELAN et en se basant sur les travaux dans le domaine de M.P. Bonacina et de J. Hsiang (Bonacina, 1992; Bonacina, 1999a; Bonacina, 1999b). Le cas des stratégies de contraction est traité à part, car ces stratégies sont cruciales dans toutes les méthodes de recherche de preuve ou de déduction et nous intéressent particulièrement dans cette thèse. Nous définissons les concepts et notions concernant les méthodes de recherche de preuve et de déduction par saturation.

Le cadre général est instancié par les cas des procédures de complétion, de résolution et de paramodulation. Le cas de la parallélisation de ces procédures est traité succinctement. La présentation de ces procédures constituent un état de l'art assez exhaustif, qui est le fondement des recherches menées dans cette thèse. Nos motivations y sont explicitées. Nous présentons la paramodulation, car tous les résultats développés dans cette thèse pour la complétion peuvent être étendus à la paramodulation. Les systèmes d'inférences et les stratégies de contraction de différentes formes de complétion, de résolution et de paramodulation sont donnés. Nous présentons les systèmes d'inférence de façon standard par des règles d'inférence et de transition. Nous présentons également de manière originale certaines formes de complétion par leur implantation en ELAN. Nous montrons ainsi que la transcription d'une notation à l'autre ne constitue pas une difficulté. Le langage ELAN est d'une grande expressivité et de ce fait il est adapté pour les implantations considérées. Dans ce chapitre, ELAN ne tient pas seulement lieu de formalisme, les implantations des règles d'inférence présentées sont tirées directement de l'implantation *ECC* (*Complétion E-cycle*). *ECC* implante une procédure de complétion de Knuth-Bendix, une procé-

dure de complétion standard avec simplification standard, une procédure de complétion basique avec simplification standard et une procédure de complétion basique avec simplification basique (avec et sans rétraction).

Troisième chapitre : Une procédure de complétion close concurrente

Une nouvelle procédure de complétion concurrente est proposée dans ce troisième chapitre. Elle est de grain fin de parallélisme et utilise les graphes SOUR (Lynch et Strogova, 1998).

Après avoir présenté la Complétion Basique des graphes SOUR, nous décrivons l'architecture et le fonctionnement de notre algorithme distribué réalisant la *Complétion (close) concurrente des graphes SOUR*. Nous donnons les algorithmes d'application des règles d'inférence, de calcul de l'unification, de l'orientation et de détection de la terminaison en utilisant des règles de réécriture que nous appelons des *transitions*. Nous montrons que la *Complétion (close) concurrente des graphes SOUR* est correcte et complète après avoir défini ces notions dans ce cas précis. Nous donnons enfin quelques résultats pratiques de *CWD* (*Completion Without Duplication*), l'implantation de notre modèle de la *Complétion (close) concurrente des graphes SOUR.*

Quatrième chapitre : Complétion basique avec simplification E-cycle

Ce chapitre décrit une nouvelle méthode de simplification complète en combinaison avec la complétion basique, la *Simplification E-cycle.*

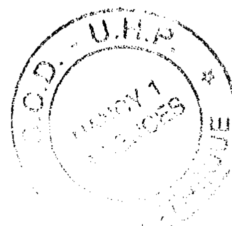
Nous montrons que la *Complétion Basique* avec *Simplification E-cycle* est complète en prouvant l'existence d'un modèle de l'ensemble saturé des égalités et qu'elle a l'avantage de réaliser plus de simplifications que la *Simplification Basique*, qui était la seule stratégie de simplification complète en combinaison avec la *Complétion Basique* connue jusqu'à présent. La *Complétion Basique avec Simplification E-cycle* est implantée dans *ECC*. Dans *ECC*, les contraintes ne sont pas simulées par le marquage des positions à partir desquelles les inférences sont interdites, mais l'expressivité d'ELAN est utilisée, les contraintes sont ainsi définies par des *sortes*. La transcription des règles d'inférence en ELAN se fait donc de façon assez naturelle.

Nous donnons également quelques éléments de réponse montrant pourquoi nous pensons que la simplification E-cycle est près de la « ligne » qui sépare les stratégies de simplification complètes des stratégies de simplification incomplètes et nous répondons au problème ouvert numéro un de R. Nieuwenhuis (Nieuwenhuis, 1999) en donnant un exemple montrant que la stratégie de simplification *avant* (*forward*) n'est pas complète.

Cinquième chapitre : Complétion basique modulo avec simplification

Ce chapitre concerne la mise au point d'un nouveau système d'inférence pour la *Complétion Basique modulo une théorie équationnelle* avec une stratégie de simplification complète, autorisant le plus de simplifications possibles et ne demandant pas de résoudre les contraintes à l'inverse de la *Simplification Basique* et de la *Simplification E-cycle* dont l'utilisation en *Complétion Basique modulo* est vouée à l'échec, car ces deux stratégies impliquent la résolution des contraintes. Nous voulons de ce fait qu'il n'y ait pas un écart insurmontable entre la théorie et la pratique. Nous

présentons le système d'inférence $BCICS_m$ (*Complétion Basique modulo E avec inférences dans les contraintes et Simplification*). Pour atteindre notre objectif, le système d'inférence réalise des inférences dans les contraintes et nous introduisons des contraintes de contraintes, les contraintes d'ordre contraintes et les contraintes de filtrage contraintes. La complétude de $BCICS_m$ est laissée pour lors en conjecture.



Chapitre 1

Cadre de travail

Sommaire

1.1 Termes, substitutions, égalités, clauses **10**
- 1.1.1 Termes et substitution 10
- 1.1.2 Egalités 11
- 1.1.3 Réécriture 13
- 1.1.4 Problème du mot 14
- 1.1.5 Clauses 14

1.2 Ordres **14**
- 1.2.1 Ordres de réduction 14
- 1.2.2 T-compatibilité 16
- 1.2.3 Ordre sur les clauses 16

1.3 Langage de contraintes **17**
- 1.3.1 Définition d'un langage de contraintes 17
- 1.3.2 Quelques langages de contraintes particuliers 18

1.4 Résolution de contraintes **19**
- 1.4.1 Ensemble des solutions d'une contrainte 19
- 1.4.2 Résolution de contraintes équationnelles et de contraintes de filtrage . . 20
- 1.4.3 Contraintes d'ordre 21

1.5 Formules contraintes **22**

1.6 ELAN **22**
- 1.6.1 Signatures 23
- 1.6.2 Règles de réécriture 25
- 1.6.3 Stratégies élémentaires 26
- 1.6.4 Modularité 28

Ce chapitre présente les notations, concepts et notions nécessaires à la compréhension de ce travail de thèse.

Il introduit dans la section 1.1 les notions concernant les termes, les substitutions et les égalités à un niveau syntaxique. La section 1.1 montre également la connexion entre la syntaxe et la sémantique basée sur des modèles pour les égalités. Elle présente de plus la logique équationnelle, la réécriture, le problème du mot et les clauses.

La section 1.2 concerne les ordres sur les termes. Elle montre la difficulté de mettre au point des ordres de simplification (totaux sur les termes clos) dans les théories équationnelles.

L'objet de cette thèse étant l'étude de procédures de déduction automatique en logique du premier ordre avec égalité en présence de contraintes, nous précisons les contraintes auxquelles nous nous intéressons dans la section 1.3 : les contraintes équationnelles, d'ordre et de filtrage.

Dans la section 1.4, nous présentons les résultats existants concernant la résolution des contraintes considérées.

Nous définissons les formules contraintes et leur sémantique dans la section 1.5. Les formules contraintes considérées sont les égalités contraintes, les règles de réécriture contraintes et les clauses contraintes. Dans le chapitre 5, nous définissons des contraintes sur des contraintes. L'idée d'utiliser des contraintes sur des contraintes semble naturelle. En effet, les contraintes sont des formules, elles peuvent donc être contraintes.

Nous décrivons succinctement dans la section 1.6 le langage ELAN et en particulier les constructions que nous utilisons dans les chapitres 2, 4 et 5. ELAN a servi dans cette thèse à implanter différentes procédures de complétion et il a également été choisi comme formalisme pour décrire certains calculs.

1.1 Termes, substitutions, égalités, clauses

1.1.1 Termes et substitution

$\mathcal{F}$ dénote un ensemble dénombrable de symboles de fonction, $\mathcal{P}$ un ensemble dénombrable de symboles de prédicat, et $\mathcal{X}$ un ensemble de variables. *arity* est une fonction de $\mathcal{F} \cup \mathcal{P}$ dans $\mathbb{N}$ qui définit l'*arité* d'un symbole de fonction ou de prédicat, c'est-à-dire le nombre de ses arguments.

Une *signature* Σ définit un ensemble de symboles de fonctions et de prédicats auxquels on associe leur arité.

L'ensemble des *termes* est $\mathcal{T}(\mathcal{F},\mathcal{X})$. Il s'agit du plus petit ensemble tel que :

- $\mathcal{X} \subseteq \mathcal{T}(\mathcal{F},\mathcal{X})$,
- $\forall f \in \mathcal{F}$, *tel que* $arity(f) = n$, $f(t_1, \dots t_n) \in \mathcal{T}(\mathcal{F},\mathcal{X})$ si $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$. f est appelé le *symbole de tête* du terme $f(t_1, \dots t_n)$ et noté $top(f(t_1, \dots t_n))$.

$\mathcal{V}(t)$ est l'ensemble des variables d'un terme t. Les termes sans variable sont appelés *clos*. L'ensemble des termes clos est $\mathcal{T}(\mathcal{F})$.

Une *position* p est une suite finie d'entiers. Si cette suite est vide, on obtient la *position vide*, notée ϵ. L'ensemble des positions d'un terme t est noté $\mathcal{P}os(t)$. Nous définissons le *sous-terme* $t_{|p}$ du terme t *à la position* p tel que :

- $t_{|\epsilon} = t$,
- si $t = f(t_1, \dots t_n)$, $p = i\ .q$ pour $1 \leq i \leq n$, alors $t_{|p} = t_{i|q}$.

La *taille* d'un terme t est notée $|t|$. La taille d'un terme t est égale au cardinal de $\mathcal{P}os(t)$.

Nous écrivons $s[t]_p$ pour indiquer que s est un terme contenant t à la position p, c'est-à-dire, $s_{|p} = t$. $s[t]$ indique que s contient t.

Une *substitution* σ est une application de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F},\mathcal{X})$, donnée en notation post-fixée, qui est l'identité partout exceptée en un nombre fini de variables. L'ensemble des substitutions est noté *Sub*. L'ensemble $Dom(\sigma) = \{x \mid \sigma(x) \neq x,\ x \in \mathcal{X}\}$ est appelé le *domaine* de la substitution σ. L'ensemble $Ran(\sigma) = \{\sigma(x) \mid \sigma(x) \neq x,\ x \in \mathcal{X}\} \subseteq \mathcal{T}(\mathcal{F},\mathcal{X})$ est appelé l'*image* de la substitution σ. Nous notons *id* la substitution telle que $Ran(id) = \emptyset$. Cette substitution est appelée l'*identité*. Une substitution σ est dite *close*, si $Ran(\sigma)$ est un ensemble de termes clos. Une *substitution de renommage* ρ est une application injective de $\mathcal{X}$ dans $\mathcal{X}$. Une substitution de renommage ρ est dite *nouvelle* (*fresh*), si les variables de $Ran(\rho)$ n'apparaissent nul part ailleurs. Une substitution σ peut être étendue en σ^* de $\mathcal{T}(\mathcal{F},\mathcal{X})$ dans $\mathcal{T}(\mathcal{F},\mathcal{X})$ telle que $\sigma^*(x) = \sigma(x)$ pour $x \in \mathcal{X}$ et $\sigma^*(f(s_1, \dots, s_n)) = f(\sigma^*(s_1), \dots, \sigma^*(s_n))$. La *composition* $\sigma\tau$ de deux substitutions est une substitution définie par $\sigma(\tau(x)) = \sigma^*(\tau(x))$. Dans la suite, pour simplifier les notations, nous ne distinguons plus σ de son extension σ^*.

1.1.2 Egalités

Le symbole $\approx$ est un symbole binaire utilisé en notation infixée pour représenter l'*égalité*. Dans la littérature, le terme *équation* est souvent employé au lieu du terme égalité. Nous faisons ici la différence entre une égalité vraie dans une algèbre et, une équation satisfaisable ou à résoudre dans une algèbre. Une équation est construite dans la suite à partir du symbole $=$. La satisfaisabilité des équations est traitée dans les sections 1.3 et 1.4.

Une égalité sur $\mathcal{T}(\mathcal{F},\mathcal{X})$ est une expression $s \approx t$, où s, $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$. s est le *membre gauche* de l'égalité et t en est le *membre droit*, ce que nous notons : $lhs(s \approx t) = s$ et $rhs(s \approx t) = t$.

L'application d'une substitution σ à une égalité $s \approx t$ est $\sigma(s) \approx \sigma(t)$, notée $\sigma(s \approx t)$. $\sigma(s \approx t)$ est une *instance* de $s \approx t$.

L'axiome $f(x,y) \approx f(y,x)$ décrit la commutativité de f, il est noté C. On dit également que l'axiome $f(f(x,y),z) \approx f(x,f(y,z))$ décrit l'associativité de f, il est noté A.

Les égalités peuvent être utilisées pour transformer des termes en d'autres termes. Par exemple, si l'on considère l'égalité $f(x,f(y,z)) \approx f(f(x,y),z)$, $f(e,f(i(e),e))$ peut être transformé en $f(f(e,i(e)),e)$.

Nous définissons alors la *relation de remplacement d'égal par égal* $\leftrightarrow_E \subseteq \mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{T}(\mathcal{F},\mathcal{X})$ pour un ensemble d'égalités E par $s \leftrightarrow_E t$ si et seulement si : $\exists\ l \approx r \in E$ (ou $r \approx l \in E$), $p \in \mathcal{P}os(s)$, $\sigma \in Sub$, tels que $(s_{|p} = \sigma(l)$ *et* $t = s[\sigma(r)]_p)$. Trouver une substitution σ telle que $t = \sigma(s)$ (si elle existe) est un *problème de filtrage.* Si σ existe, on dit que σ est un *filtre* de t dans s, que t est une *instance* de s et on écrit $s \lhd t$. Par exemple, $f(x,y) \lhd f(g(z),x)$ avec $\sigma := \{x \mapsto g(z), y \mapsto x\}$. Le filtrage est la base de la réécriture. Il est également utilisé dans des méthodes de recherche de preuves pour supprimer des égalités de l'espace de recherche. Les règles d'inférence de suppression et de simplification sont basées sur des conditions d'existence de filtres. Des exemples de règles de simplification sont données dans les chapitres 2, 4 et 5. $\leftrightarrow_E$ est appelée la *relation de preuve.*

$\overset{*}{\leftrightarrow}_E$ est la clôture transitive et réflexive de la relation symétrique $\leftrightarrow_E$. Une équivalence $\sim$ sur les termes est appelée une *congruence*, si $s \sim t$ implique que $u[s] \sim u[t]$ pour tous les termes s, t et u. La relation $\overset{*}{\leftrightarrow}_E$ est une relation de congruence. Le quotient de l'ensemble des termes $\mathcal{T}(\mathcal{F},\mathcal{X})$ par $\overset{*}{\leftrightarrow}_E$ est noté $\mathcal{T}(\mathcal{F},\mathcal{X})/E$.

$\overset{*}{\leftrightarrow}_E$ peut être décrite en utilisant la *logique équationnelle.* La logique équationnelle est décrite par les règles d'inférence de la figure 1.1. La notation utilisée $E \vdash s \approx t$ signifie que $s \approx t$ peut être obtenue à partir de E en appliquant les règles d'inférence de la figure 1.1, on dit que $s \approx t$ est une *conséquence syntaxique* de E. La *théorie équationnelle* induite par l'ensemble d'axiomes E est l'ensemble des égalités qui peuvent être déduites à partir de E en appliquant les règles d'inférence de la figure 1.1. Elle est notée $Th(E)$. Les théories équationnelles les plus communes sont la théorie vide, la théorie associative (A), la théorie commutative (C) et la théorie associative-commutative (AC).

Le théorème de Birkhoff (Birkhoff, 1935) fait la connexion entre $\overset{*}{\leftrightarrow}_E$ et $\vdash$.

Théorème 1 *(Birkhoff, 1935) Soient E un ensemble d'égalités, s et t deux termes, alors $s \overset{*}{\leftrightarrow}_E t$ si et seulement si $E \vdash s \approx t$.*

Pour donner une caractérisation sémantique de $\overset{*}{\leftrightarrow}_E$, nous utilisons les notions d'*interprétation*, d'*algèbre* et de *modèle.*

Soit une signature Σ sur un ensemble de symboles de fonction $\mathcal{F}$.

Une *interprétation* I est déterminée par :

- un domaine D_I, et

$\dfrac{}{E \vdash s \approx t}$	(1) Assomption	si $s \approx t \in E$
$\dfrac{}{E \vdash t \approx t}$	(2) Réflexivité	
$\dfrac{E \vdash s \approx t}{E \vdash t \approx s}$	(3) Symétrie	
$\dfrac{E \vdash s \approx t \quad E \vdash t \approx u}{E \vdash s \approx u}$	(4) Transitivité	
$\dfrac{E \vdash s \approx t}{E \vdash \sigma(s) \approx \sigma(u)}$	(5) Instanciation	
$\dfrac{E \vdash s_1 \approx t_1 \ldots E \vdash s_n \approx t_n}{E \vdash f(s_1,\ldots,s_n) \approx f(t_1,\ldots,t_n)}$	(6) Congruence	

FIG. 1.1 – *Règles d'inférence de la logique équationnelle*

– une interprétation des symboles de fonction f de $\mathcal{F}$ par $f_I : D_I{}^n \mapsto D_I$ où n est l'arité de f. $I(f(t_1,\ldots,t_n)) = f_I(I(t_1),\ldots,I(t_n))$.

Une *Σ-algèbre du premier ordre* $\mathcal{A} = (D_{\mathcal{A}},I(\Sigma))$ est donnée par un domaine $D_{\mathcal{A}}$ (*carrier set*) et une interprétation I des symboles de $\mathcal{F}$ dans $D_{\mathcal{A}}$. L'interprétation d'un symbole de fonction f dans $D_{\mathcal{A}}$ est noté $f_{\mathcal{A}}$ et la Σ-algèbre est notée $(D_{\mathcal{A}},\Sigma_{\mathcal{A}})$ ou $(D_{\mathcal{A}},\mathcal{F}_{\mathcal{A}})$.

Une Σ-algèbre $\mathcal{A} = (D_{\mathcal{A}},\Sigma_{\mathcal{A}})$ est *générée par les termes* (*term generated*), si pour tout élément du domaine $t \in D_{\mathcal{A}}$, il existe un terme t' dans $\mathcal{T}(\mathcal{F})$ tel que $I(t') = t$.

Exemple 1 *Pour une signature Σ sur un ensemble de symboles de fonction $\mathcal{F}$ et un ensemble de variables $\mathcal{X}$, en prenant pour $D_{\mathcal{A}}$ l'ensemble des termes $\mathcal{T}(\mathcal{F},\mathcal{X})$ et pour interprétation l'interprétation qui associe à chaque symbole de $\mathcal{F}$ le symbole lui-même, $(\mathcal{T}(\mathcal{F},\mathcal{X}),\mathcal{F})$ est une Σ-algèbre générée par les termes.*

Etant données une algèbre $\mathcal{A}$ et un ensemble de variables $\mathcal{X}$, une *valuation* ν est une application de $\mathcal{X}$ dans $\mathcal{A}$. Elle est étendue à un *morphisme* noté également ν de $\mathcal{T}(\mathcal{F},\mathcal{X})$ dans $\mathcal{A}$ par : $\nu(f(t_1,\ldots,t_n)) = f_{\mathcal{A}}(\nu(t_1),\ldots,\nu(t_n))$.

Une algèbre $\mathcal{A}$ est un *modèle* pour une égalité $s \approx t$ si pour toute valuation ν de variables dans s et t, $\nu(s) = \nu(t)$. On dit dans ce cas que $s \approx t$ est *valide* (vraie) dans E. On le note $\mathcal{A} \models s \approx t$. Une algèbre $\mathcal{A}$ est un modèle pour un ensemble d'égalités E si $\mathcal{A}$ est un modèle pour chaque égalité de E. On note $Mod(E)$ l'ensemble des modèles de E. Un ensemble d'égalités est dit *consistant* ou *satisfaisable* s'il a un modèle, sinon il est inconsistant ou *insatisfaisable*.

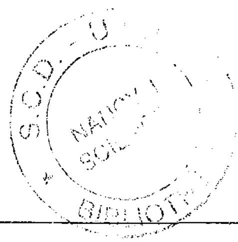
Il est facile de prouver que $(\mathcal{T}(\mathcal{F},\mathcal{X})/E,\mathcal{F})$ abrégé en $\mathcal{T}(\mathcal{F},\mathcal{X})/E$ est un modèle de E.

Le théorème de Birkhoff (Birkhoff, 1935) fait la connexion entre la syntaxe et la sémantique des égalités en montrant que la relation $\vdash$ coïncide avec la relation $\models$.

Théorème 2 *(Birkhoff, 1935) $Mod(E) \models s \approx t$ si et seulement si $\mathcal{T}(\mathcal{F},\mathcal{X})/E \models s \approx t$ si et seulement si $E \vdash s \approx t$ (si et seulement si $s \overset{*}{\leftrightarrow}_E t$).*

Nous notons $s \approx_E t$ si $Mod(E) \models s \approx t$. Le théorème 2 peut être rephrasé en utilisant cette nouvelle notation.

Parmi les algèbres qui sont des modèles de E, on distingue l'algèbre initiale $\mathcal{T}(\mathcal{F})/E$ et les algèbres qui lui sont isomorphes.


S.C.D. - U. NANCY 1 BIBLIOTHÈQUE

1.1.3 Réécriture

La réécriture permet d'orienter les égalités.

Une *règle de réécriture* est un couple (l,r) notée $l \rightarrow r$ telle que l ne soit pas une variable et $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. l est le *membre gauche* de la règle et r en est le *membre droit*, ce que nous notons : $lhs(l \rightarrow r) = l$ et $rhs(l \rightarrow r) = r$. Un *système de réécriture* R est un ensemble de règles de réécriture.

Nous définissons la *relation de réécriture* $\rightarrow_R \subseteq \mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{T}(\mathcal{F},\mathcal{X})$ pour un système de réécriture R par $s \rightarrow_R t$ si et seulement si : $\exists\, l \rightarrow r \in R$, $p \in \mathcal{P}os(s)$, $\sigma \in Sub$, $(s_{|p} = \sigma(l)$ *et* $t = s[\sigma(r)]_p)$. $\rightarrow_R$ est abrégé en $\rightarrow$ si R est évident dans le contexte.

$\overset{*}{\rightarrow}_R$ est la clôture transitive de $\rightarrow_R$. Nous écrivons $u_0 \overset{*}{\rightarrow}_R u_n$ et disons que u_0 *se réécrit* en u_n ou qu'il existe une *dérivation de réécriture* de u_0 à u_n, s'il existe un ensemble de termes $\{u_1,\ldots,u_{n-1}\}$ tels que pour tout i, $1 \leq i \leq n$, $u_{i-1} \rightarrow_R u_i$. Si la dérivation contient au moins une réécriture, on note $u_0 \overset{+}{\rightarrow}_R u_n$.

On note :

- $t \leftarrow_R s$ si et seulement si $s \rightarrow_R t$,
- $s \leftrightarrow_R t$ si et seulement si $s \rightarrow_R t$ ou $t \rightarrow_R s$,
- $t \overset{*}{\leftarrow}_R s$ si et seulement si $s \overset{*}{\rightarrow}_R t$, et
- $t \overset{+}{\leftarrow}_R s$ si et seulement si $s \overset{+}{\rightarrow}_R t$.

Pour un système de réécriture $R = \{l_i \rightarrow r_i\}_{i \in I}$, nous écrivons $\approx_R$ la théorie équationnelle générée par l'ensemble d'égalités $E = \{l_i \approx r_i\}_{i \in I}$.

Nous utilisons la terminologie suivante pour décrire la relation $\rightarrow_R$ sur $\mathcal{T}(\mathcal{F},\mathcal{X})$.

- y est une *forme normale* de x par rapport à $\rightarrow_R$ si $x \overset{*}{\rightarrow}_R y$ et si y est *irréductible*, c'est-à-dire qu'il n'existe pas de z tel que $y \overset{+}{\rightarrow}_R z$. Nous notons $y \downarrow_R$ l'ensemble des formes normales de y.
- s et t sont *joignables* s'il existe z tel que $s \overset{*}{\rightarrow}_R z \overset{*}{\leftarrow}_R t$ et dans ce cas, nous écrivons $s \downarrow_R t$.
- $\rightarrow_R$ est *Church-Rosser* si $x \overset{*}{\leftrightarrow}_R y$ implique $x \downarrow_R y$.
- $\rightarrow_R$ *termine* s'il n'existe pas de chaîne infinie $y_1 \rightarrow_R y_2 \rightarrow_R \ldots$
- $\rightarrow_R$ est *confluente* si $y_1 \overset{*}{\leftarrow}_R x \overset{*}{\rightarrow}_R y_2$ implique $y_1 \downarrow_R y_2$. Dans ce cas, chaque terme a au plus une forme normale.
- $\rightarrow_R$ est *localement confluente* si $y_1 \leftarrow_R x \rightarrow_R y_2$ implique $y_1 \downarrow_R y_2$.
  Si $\rightarrow_R$ est localement confluente et si elle termine, alors elle est confluente.
- $\rightarrow_R$ est *normalisante* si chaque élément a une *forme normale*.
  Les propriétés de normalisation et de confluence impliquent l'unicité de la forme normale.
- $\rightarrow_R$ est *convergente* si elle est confluente et si elle termine.
  La propriété de convergence implique elle aussi, par définition, l'unicité de la forme normale.

On dira que R est respectivement *terminant*, *localement confluent*, *confluent*, *Church-Rosser* et *convergent*, si $\rightarrow_R$ est respectivement terminante, localement confluente, confluente, Church-Rosser ou convergente.

Un système de réécriture R est équivalent à un système de réécriture R', noté $R \equiv R'$, si $R \models R'$ et $R' \models R$.

Théorème 3 *Soient R et R' deux systèmes de réécriture. Si $R \subseteq R'$ tel que R soit confluent et $R \equiv R'$, alors R' est confluent.*

Preuve : Supposons que $R \subseteq R'$, R est confluent, $R \equiv R'$ et $s \approx_{R'} t$. Nous prouvons que R' est confluent en prouvant que $s \approx t$ a une preuve par réécriture dans R'.
$s \approx_{R'} t$ implique $R' \models s \approx t$ d'après le théorème de Birkhoff. Comme $R \equiv R'$, $R \models s \approx t$ et d'après le théorème de Birkhoff, $s \approx_R t$. De plus, R est confluent. Ainsi, il existe une preuve de réécriture de $s \approx t$ dans R. Il existe une preuve de réécriture de $s \approx t$ dans R et $R \subseteq R'$ implique qu'il existe une preuve par réécriture de $s \approx t$ dans R qui est la même que celle dans R'. Ceci prouve que R' est confluent. $\square$

Un système de réécriture R est *linéaire-gauche* (respectivement *linéaire-droit*) si pour toute règle $l \rightarrow r \in R$, aucune variable n'est présente plus d'une fois dans l (respectivement dans r). R est *linéaire*, s'il est linéaire-gauche et linéaire-droit. R est *réduit à gauche* si pour toute règle de réécriture $l \rightarrow r$ dans R, le terme l est irréductible par $R - \{l \rightarrow r\}$. On notera en particulier qu'un système de réécriture clos, réduit à gauche et terminant est convergent.

1.1.4 Problème du mot

Le problème central du raisonnement dans les théories équationnelles est la validité des égalités dans une algèbre.

Le *problème du mot* pour E consiste à décider si l'égalité $s \approx t$ est une conséquence logique de E, c'est-à-dire si $E \models s \approx t$, pour des termes s et t arbitraires. Ce problème est en général indécidable. Toutefois, un système de réécriture R décide le problème du mot pour la théorie équationnelle E, si R est fini, convergent et si $\approx_R$ et $\approx_E$ coïncident. Si R est convergent, $E \models s \approx t$ s'il existe u tel que $s \xrightarrow{*}_R u \xleftarrow{*}_R t$. Ces étapes de réécriture constituent une *preuve en « vallée »* de $s \approx t$ dans E appelée également une *preuve par réécriture*. De plus, si E est fini et clos, alors la clôture de congruence décide le problème du mot (Kozen, 1977).

1.1.5 Clauses

Un *atome* a la forme $p(t_1, \ldots, t_n)$ où $p \in \mathcal{P}$ et pour tout $i \in \{1, \ldots, n\}$, $t_i \in \mathcal{T}(\mathcal{F},\mathcal{X})$. Un littéral est un atome (A) ou la négation d'un atome ($\neg A$). On parle alors respectivement d'atome positif ou négatif. Une *clause* C est un multi-ensemble de littéraux L_i ($i \in \{1, \ldots, n\}$) notée sous la forme d'une disjonction $L_1 \vee \ldots \vee L_n$. La clause ne contenant aucun littéral est appelée la *clause vide*, elle est notée $\square$. Une clause ne contenant que des littéraux positifs (respectivement négatifs) est dite *positive* (respectivement *négative*).

1.2 Ordres

Un *ordre* $\succ$ est une relation binaire réflexive et transitive. Dans la suite, nous utilisons un ordre partiel strict. Un *ordre partiel* est une relation réflexive, antisymétrique et transitive. Un *ordre strict* est une relation transitive et irreflexive.

Les ordres sont utilisés pour orienter les règles de réécriture. L'utilisation d'*ordres* sur les termes est répandue dans les méthodes de recherche de preuves ou de déduction automatique. Ils permettent de restreindre l'application des règles d'inférence. Ce sujet est abordé dans le chapitre 2.

1.2.1 Ordres de réduction

Un ordre $\succ$ est un *ordre de réduction* si :

- $\succ$ est *bien fondé* (ou *noethérien*), c'est-à-dire qu'il n'existe pas de chaîne infinie de termes

s_1, s_2, ... telle que $s_i \succ s_{i+1}$ pour tout $i \geq 1$,
- $\succ$ est *monotone* (on dit aussi *stable par contexte*), c'est-à-dire qu'étant donnés deux termes s et t tels que $s \succ t$, alors $w[s]_p \succ w[t]_p$ pour tout terme w et toute position p dans w, et
- $\succ$ est *stable par instanciation*, c'est-à-dire que, pour toute substitution σ et pour tous termes s et t tels que $s \succ t$, alors $\sigma(s) \succ \sigma(t)$.

Exemple 2 *L'ordre $\succ$ sur les termes défini par $s \succ t$ si et seulement si $|s| > |t|$ n'est pas un ordre de réduction car il n'est pas stable par instanciation. Si l'on considère la signature $\Sigma = \{f,g,a,b\}$, les termes $f(a,b)$ et $g(y)$ et la substitution $\sigma := \{y \mapsto f(a,b)\}$, alors $f(a,b) \succ g(y)$ car $3 > 2$ mais $\sigma(f(a,b)) \not\succ \sigma(g(y))$ car $f(a,b) \not\succ g(f(a,b))$ car $3 \not> 4$. Par contre, l'ordre $\succ$ sur les termes défini par $s \succ t$ si et seulement si $|s| > |t|$ et pour tout $x \in \mathcal{V}(s) \cup \mathcal{V}(t)$, le nombre d'occurrences de x dans s est supérieur ou égal au nombre d'occurrences de x dans t est un ordre de réduction.*

Un *ordre de simplification* $\succ$ est une relation réflexive, transitive, monotone qui contient la *propriété de sous-terme*: si s est un sous-terme de t, alors $t \succeq s$.

Un ordre est *total* si pour tous termes s et t, on a soit $s \succ t$, soit $t \succ s$, soit $s = t$.

La terminaison des systèmes de réécriture est un problème indécidable (même pour un système de réécriture contenant une seule règle de réécriture avec un symbole de fonction d'arité supérieure à 1 (Dauchet, 1989; Dauchet, 1992) ou si le système n'est composé que de règles dont les symboles sont d'arité 1 (Huet et Lankford, 1978)).

Les ordres peuvent servir pour orienter les règles de réécriture. L'intérêt des ordres de réduction est apparent dans le théorème de *terminaison des systèmes de réécriture* suivant :

Théorème 4 *(Lankford, 1977) Un système de réécriture R termine si et seulement si il existe un ordre de réduction $\succ$ qui satisfait $l \succ r$ pour toute règle de réécriture $l \to r$ de R.*

Les ordres de simplifications peuvent également être utilisés pour prouver la terminaison des systèmes de réécriture (Dershowitz, 1982).

$M(t)$ est le nombre d'occurrences de t dans le multi-ensemble M. L'*extension d'un ordre $\succ$ aux multi-ensembles* est définie par : $S \succ^{mult} T$ si, (i) $S \neq T$ et (ii) quand $T(s) > S(s)$ alors il existe t tel que $t \succ s$ et $S(t) > T(t)$.

L'*extension lexicographique d'un ordre* $\succ$ est définie par : $(s_1,\ldots,s_n) \succ^{lex} (t_1,\ldots,t_n)$ si : $\exists j$, $s_j \succ t_j$ et $\forall i$, $i < j$, $s_i = t_i$.

L'un des moyens de définir un ordre de simplification est de le construire directement à partir d'un ordre bien fondé sur la signature, appelé *précédence*. L'ordre de précédence est noté $\succ_p$.

Les ordres exploitant les chemins de la structure d'arbre des termes sont appelés *ordres sur les chemins*. L'*ordre lexicographique sur les chemins* (LPO) (Kamin et Lévy, 1982) et l'*ordre récursif sur les chemins* (RPO) (Dershowitz, 1982) sont des exemples de tels ordres. Ils sont déterminés à partir d'une précédence totale.

Définition 1 *On dit que le terme $s = f(s_1,\ldots,s_n)$ est plus grand que le terme $t = g(t_1,\ldots,t_m)$ par rapport à LPO et on note : $s \succ_{lpo} t$ si et seulement si l'*une *des conditions suivantes est vraie :*
- $f \succ_p g$ *et* $\forall j$, $1 \leq j \leq m$, $s \succ_{lpo} t_j$,
- $\exists i$, $1 \leq i \leq n$, $s_i \succeq_{lpo} t$,
- $f = g$, $(s_1,\ldots,s_n) \succ^{lex}_{lpo} (t_1,\ldots,t_n)$ *et* $\forall j$, $1 \leq j \leq m$, $s \succ_{lpo} t_j$, *où* $\succ^{lex}_{lpo}$ *est l'extension lexicographique de* $\succ_{lpo}$. □

Définition 2 *On dit que le terme $s = f(s_1, \ldots, s_n)$ est plus grand que le terme $t = g(t_1, \ldots, t_m)$ par rapport à RPO et on note : $s \succ_{rpo} t$ si et seulement si l'une des conditions suivantes est vraie :*

- *$f \succ_p g$ et $\forall j$, $1 \leq j \leq m$, $s \succ_{rpo} t_j$, ou*
- *$f = g$ et $\{s_1, \ldots, s_n\} \succ_{rpo}^{mult} \{t_1, \ldots, t_n\}$, ou*
- *$\exists i$, $1 \leq i \leq n$, $s_i \succeq_{rpo} t$ ou $s_i \simeq t$, où $\simeq$ est défini par : $s \simeq t$ si $s = t \in \mathcal{X}$ ou $s = f(s_1, \ldots, s_n)$ et $t = f(t_1, \ldots, t_n)$ et il existe une permutation π de $\{1, \ldots, n\}$ telle que pour $1 \leq i \leq n$, $s_i \simeq t_{\pi(i)}$.* $\square$

LPO est un ordre de simplification total sur les termes clos, RPO est un ordre de simplification total sur les termes clos à la permutation près des arguments des symboles de fonction f considérés comme des multi-ensembles.

1.2.2 T-compatibilité

Si l'on travaille modulo une théorie équationnelle T, nous définissions la propriété de *T-compatibilité* relativement à l'ordre considéré. Un ordre $\succ$ est T-compatible, si étant donnés deux termes s et t, tels que $s \succ t$, alors pour tous termes s' et t' tels que $s \approx_T s'$ et $t \approx_T t'$, alors on a : $s' \succ t'$. La définition d'ordres de simplification totaux sur les termes clos dans une théorie T est un problème difficile. Les principaux travaux dans ce domaine se trouvent dans (Jouannaud et Muñoz, 1984; Bachmair et Plaisted, 1985; Narendran et Rusinowitch, 1996; Rubio, 1994). *RPO* est un ordre de simplification C-compatible total sur les termes clos à la permutation près des arguments des symboles de fonction f considérés comme des multi-ensembles. Tout ordre AC-compatible est A-compatible et C-compatible mais le problème est d'obtenir des ordres T-compatibles totaux sur les termes clos.

Pour la théorie AC, deux ordres de simplification AC-compatibles totaux sur les termes clos ont été proposés : l'ordre de P. Narendran et M. Rusinowitch (Narendran et Rusinowitch, 1996) et l'ordre AC-RPO de R. Nieuwenhuis et A. Rubio (Nieuwenhuis et Rubio, 1995b). Tout récemment, un ordre de ce type a été proposé dans (Nieuwenhuis et Rivero, 1999). L'ordre *APO* de L. Bachmair et D. Plaisted (Bachmair et Plaisted, 1985) est un ordre de simplification AC-compatible basé sur RPO mais qui a l'inconvénient de limiter le nombre de symboles AC et qui n'est pas total sur les termes clos.

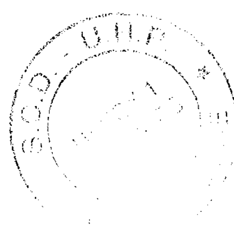
Dans la suite, $\succeq_t$ ($\succ_t$ dans sa version stricte) est un ordre de réduction bien fondé total sur les termes clos et T-compatible. $\succeq_t$ peut être instancié par LPO, RPO, ...

1.2.3 Ordre sur les clauses

Différents ordres sur les clauses ont été introduits dans la littérature. L'ordre que nous présentons ici provient de (Bachmair et al., 1995). Il est similaire à celui proposé par L. Bachmair et G. Ganzinger proposé dans (Bachmair et Ganzinger, 1994).

Dans (Bachmair et al., 1995), tout atome $p(t_1, \ldots, t_n)$ peut s'écrire sous la forme $p(t_1, \ldots, t_n) \approx \top$. La signature est étendue dans ce sens et la précédence est telle que $\top$ soit le plus petit symbole de la précédence.

Soit $\succ$ un ordre de réduction total sur les termes clos et une précédence sur les symboles de fonction et de prédicat. Pour étendre $\succ$ aux littéraux et aux clauses, un littéral positif $s \approx t$ est identifié à un multi-ensemble $\{\{s\}, \{t\}\}$ et un littéral négatif $s \not\approx t$ est identifié comme le multi-ensemble $\{\{s, t\}\}$. L'extension $(\succ^{mult})^{mult}$ est un ordre sur les littéraux noté $\succ_L$ et $((\succ^{mult})^{mult})^{mult})$ est un ordre sur les clauses noté $\succ_C$. Si $\succ_t$ est l'ordre de réduction total sur



les termes clos utilisé, et si les littéraux sont des égalités, l'ordre $\succ_L$ est noté $\succ_e$. Lorsqu'on compare un littéral avec une clause, le littéral est considéré comme une clause unitaire. En considérant les trois termes s, t et u tels que $s \succ t \succ u$, alors $s \not\approx u \succ s \approx t \succ s \approx u$.

Un littéral L (respectivement une clause C) est *maximal(e)* dans un ensemble S s'il n'existe pas de littéral L' (respectivement de clause C') dans S tel que $L' \succ_L L$ (respectivement $C' \succ_C C$). Un littéral L (respectivement une clause C) est *strictement maximal(e)* dans un ensemble S s'il n'existe pas de littéral L' (respectivement de clause C') dans $S - C$ (respectivement $S - L$) tel que $L' \succeq_L L$ (respectivement $C' \succeq_C C$).

1.3 Langage de contraintes

1.3.1 Définition d'un langage de contraintes

G. Smolka a proposé dans (Smolka, 1989) une notion générale de langage de contraintes construit à partir de symboles de fonction $\mathcal{F}$ et de prédicat $\mathcal{P}$ et d'un ensemble non vide d'interprétations. Chaque interprétation I est définie par un domaine D_I et associe à chaque contrainte l'ensemble de ses solutions dans I, qui est un sous ensemble des affectations de $V \mapsto D_I$ où V est un ensemble de variables infini dénombrable. Cette définition générale a été instanciée dans (Kirchner et al., 1990) pour prendre en compte des contraintes particulières utilisées dans le domaine de la déduction automatique. Le langage de contraintes atomiques $ACL[\mathcal{F},\mathcal{P}]$ a ainsi été formalisé. $ACL[\mathcal{F},\mathcal{P}]$ est défini par :

- un ensemble de variables $\mathcal{X}$,
- un ensemble de contraintes atomiques $\mathcal{C}$ construites à partir de termes de $\mathcal{T}(\mathcal{F},\mathcal{X})$ et de symboles de prédicat $\mathcal{P}$,
- une fonction $\mathcal{V}$ qui associe à chaque contrainte c l'ensemble de ses variables contraintes,
- et un ensemble d'interprétations I non vide.

  Chaque interprétation I est déterminée par :
  - un domaine D_I,
  - une interprétation des symboles de fonction f de $\mathcal{F}$ par $f_I : {D_I}^n \mapsto D_I$ où n est l'arité de f,
  - une interprétation des symboles de prédicat p de $\mathcal{P}$ où $p_I \subseteq {D_I}^n$, et
  - une application qui associe :
    - à chaque variable x de $\mathcal{C}$, l'ensemble des affectations $Sol_I(x) = \{\alpha : V \mapsto D_I\}$,
    - à chaque terme $t = f(t_1, \ldots, t_n)$ de $\mathcal{C}$, l'ensemble des affectations $Sol_I(t) = \{\alpha : V \mapsto D_I \mid \underline{\alpha}(t) \in D_I\}$ avec : $\underline{\alpha}(f(t_1, \ldots, t_n)) = f_I(\underline{\alpha}(t_1), \ldots, \underline{\alpha}(t_n))$ et $\underline{\alpha}(x) = \alpha(x)$, si x est une variable.
  - à chaque littéral $c = p(t_1, \ldots, t_n)$ dans $\mathcal{C}$, l'ensemble des affectations $Sol_I(l) = \{\alpha : \mathcal{V} \mapsto D_I \mid (\underline{\alpha(t_1)}, \ldots, \underline{\alpha(t_n)}) \in p_I\}$, où $\underline{\alpha}$ est défini comme précédemment.

    Chaque affectation dans $Sol_I(c)$ est une *solution* de c dans I.

Etant donnée une contrainte c, on distingue trois problèmes, le problème de la *validité* de c, le problème de la *satisfaisabilité* de c et la *résolution* de c.

Une contrainte c de $\mathcal{C}$ est *valide* dans une interprétation I, si toute affectation est une solution de c dans I. Une contrainte c de $\mathcal{C}$ est *satisfaisable*, s'il existe au moins une interprétation I dans laquelle c a une solution. La contrainte $\top$ est telle que $Sol_I(\top)$ soit l'ensemble de toutes les affectations de V dans I. La contrainte $\top$ est valide dans toutes les interprétations. La contrainte $\bot$ est telle que $Sol_I(\bot)$ soit l'ensemble vide. La contrainte $\bot$ est dite insatisfaisable.

Résoudre une contrainte c dans une interprétation consiste à énumérer toutes ses solutions $Sol(c)$ dans l'interprétation donnée. On s'intéresse dans la suite seulement à la satisfaisabilité et à la résolution d'une contrainte. Un langage de contrainte est *décidable*, si la satisfaisabilité de ses contraintes est décidable.

Exemple 3 *Supposons que $\mathcal{F} = \{o,suc\}$ et $\mathcal{P} = \{=\}$, D_I est l'ensemble des entiers naturels. o est interprété par l'entier naturel 0 et suc par la fonction de successeur. $=$ est interprété par l'égalité sur les entiers.*

Soit la contrainte atomique $c = suc(suc(x)) =^? suc(y)$ [1].

La contrainte $suc(suc(x)) =^? suc(y)$ est satisfaisable, car, pour l'interprétation sur les entiers naturels, il est possible de trouver des valuations pour x et y dans les entiers naturels tels que $suc(suc(x))$ et $suc(y)$ soient interprétés comme le même entier naturel. On prend pour x, n et pour y, $n+1$.

Le langage $ACL[\mathcal{F},\mathcal{P}]$ a ensuite été étendu pour former le langage $SL[\mathcal{F},\mathcal{P}]$ pour prendre en compte des contraintes non atomiques. La conjonction, la négation et la quantification existentielle de contraintes atomiques ont ainsi été ajoutées[2].

1.3.2 Quelques langages de contraintes particuliers

Dans cette thèse, nous utilisons des langages de contraintes basés sur les restrictions suivantes. Le domaine D_I des interprétation que nous considérons est l'ensemble des termes clos $\mathcal{T}(\mathcal{F})$. L'application associée à ces interprétations renvoie pour chaque contrainte un ensemble de substitutions closes qui représente ses solutions. Nous nous intéressons donc aux langages de contraintes $SL_T[\mathcal{F},\mathcal{P}]$, où T est une théorie équationnelle et l'ensemble des prédicats $\mathcal{P}$ contient :

- le prédicat $=$. $=_T$ est interprété comme l'égalité modulo T.
- le prédicat de filtrage $\lhd$.
  Nous étendons ici le filtrage au cas équationnel. On dit que t *filtre* s modulo la théorie équationnelle ou l'ensemble d'axiomes équationnels T, s'il existe une substitution filtre σ telle que $\sigma(s) \approx_T t$. Dans ce cas, t est une instance de s et on le note $s \lhd_T t$.
  $\lhd$ est donc interprété comme le filtrage syntaxique et $\lhd_T$ est interprété comme le filtrage modulo T.
- le prédicat d'ordre $\succ$. $\succ$ est interprété comme un ordre de simplification total sur les termes clos et $\succ_T$ est interprété comme un ordre de réduction T-compatible total sur les termes clos.

Les contraintes atomiques c que nous considérons sont :
- des équations de la forme $t =^?_T t'$,
- des contraintes de filtrage atomiques de la forme $t \lhd^?_T t'$, et
- des inéquations de la forme $t \succ^?_T t'$ [3].

Pour ces contraintes, $\mathcal{V}(c) = \mathcal{V}(t) \cup \mathcal{V}(t')$.

Les langages ainsi construits sont notés $SL_T[\mathcal{F},\mathcal{P}]$.

Une *contrainte équationnelle* est une conjonction d'équations. Une contrainte équationnelle est dans la suite également appelée un *problème équationnel* ou *problème d'unification*. Une

1. Dans les contraintes, les symboles de prédicat sont suivis de $^?$ pour souligner que l'on s'intéresse à leur satisfaisabilité ou à leur résolution.

2. La disjonction, l'implication et la quantification universelle de contraintes sont déduites directement à partir de l'utilisation des conjonctions, de la négation et la quantification existentielle des contraintes.

3. On appelle aussi inéquations les contraintes de la forme $t \succeq^?_T t'$.

contrainte d'ordre est une conjonction d'équations et d'inéquations. Une *contrainte de filtrage* est une conjonction de contrainte de filtrage atomique.

1.4 Résolution de contraintes

Nous allons énumérer les résultats concernant la résolution de contraintes équationnelles et de filtrage qui nous intéressent dans cette thèse dans la théorie vide, dans la théorie commutative, dans la théorie associative et dans la théorie associative-commutative. De plus, nous caractérisons les ensembles de solutions d'une contrainte. Ces résultats seront utilisés dans le chapitre 2 et dans le chapitre 5 pour montrer les problèmes engendrés par la résolution des contraintes lorsque l'on fait de la recherche de preuve automatique ou de la déduction.

1.4.1 Ensemble des solutions d'une contrainte

Une substitution σ est *plus générale* qu'une substitution θ modulo la théorie équationnelle T sur un ensemble de variables W, s'il existe une substitution τ telle que pour toute variable x de W, $\tau(\sigma(x)) \approx_T \theta(x)$. Les substitutions σ et θ sont, dans ce cas, *comparables* et on notera: $\sigma \leq_{s,T}^{W} \theta$. Par exemple, la substitution $\sigma := \{x \mapsto f(y)\}$ est plus générale que $\theta := \{x \mapsto f(a)\}$ pour $W = \{x\}$ et $\tau := \{y \mapsto a\}$.

$Sol_I(c)$ est l'ensemble des *solutions* de la contrainte c dans l'interprétation I.

Dans le cas où I est l'algèbre $\mathcal{T}(\mathcal{F},\mathcal{X})/T$, $Sol_I(c)$ est noté $Sol(c)$, $Sol(c)$ est un ensemble de substitutions et l'*ensemble complet des solutions* de c est un sous-ensemble de $Sol(c)$, noté $CSS(c)$. $CSS(c)$ est tel que $\forall\sigma \in CSS(C)$, $\sigma(c)$ soit valide et $\forall\theta \in Sol(c)$, $\exists\sigma \in CSS(c)$ telle que $\sigma \leq_{s,T}^{\mathcal{V}(c)} \theta$. Dans le cas où I est plus particulièrement un ensemble de termes clos ou un quotient d'un ensemble de termes clos, $Sol(c)$ est noté $Sol_G(c)$ et $Sol_G(c)$ est un ensemble de substitutions closes. $CSS(c)$ est noté $CSS_G(c)$.

Si I est l'algèbre $\mathcal{T}(\mathcal{F},\mathcal{X})/T$ et si c est une contrainte équationnelle, on parle d'*unificateurs* plutôt que de solutions.

L'unification modulo une théorie équationnelle T est le processus consistant à résoudre des équations dans l'algèbre $\mathcal{T}(\mathcal{F},\mathcal{X})/T$ (voir (Jouannaud et Kirchner, 1991; Baader et Siekmann, 1993) pour des vues d'ensemble concernant l'unification). Dans cette thèse, nous nous intéressons particulièrement à l'unification dans la théorie vide c'est-à-dire l'unification syntaxique, dans la théorie commutative, dans la théorie associative et dans la théorie associative-commutative.

On note $U_T(c)$ l'ensemble des *unificateurs* de la contrainte c modulo T. Nous nous intéressons plus particulièrement à certains ensembles générés par l'ensemble $U_T(c)$. L'*ensemble complet des unificateurs* de c est un sous-ensemble de $U_T(c)$. On le note $CSU_T(c)$. $CSU_T(c)$ est tel que $\forall\theta \in U_T(c)$, $\exists\sigma \in CSU_T(c)$ telle que $\sigma \leq_{s,T}^{\mathcal{V}(c)} \theta$ et $\forall\sigma \in CSU_T(c)$, $Dom(\sigma) \cap Ran(\sigma) = \emptyset$. $CSMGU_T(c)$ est l'ensemble complet des unificateurs plus généraux de c si $\forall\theta$, $\sigma \in CSMGU_T(c)$, $\sigma \leq_{s,T}^{\mathcal{V}(c)} \theta$ implique $\sigma = \theta$. Dans la théorie vide, l'ensemble complet des unificateurs plus généraux de la contrainte c a au plus un élément appelé unificateur plus général de c et noté $mgu(c)$. Un sous-ensemble d'unificateurs U_1 de U_2 est minimal, si $\forall\sigma_2 \in U_2$, $\exists\sigma_1 \in U_1$, telle que $\sigma_2 \leq_s \sigma_1$ et $\forall\sigma_1,\sigma_1' \in U_1$, $\sigma_1 \leq \sigma_1' \Rightarrow \sigma_1 = \sigma_1'$. Pour plus de détails concernant ces ensembles, le lecteur pourra se référer aux travaux de (Plotkin, 1972; Huet, 1976; Hullot, 1980; Jouannaud et Kirchner, 1991).

L'unification modulo T est dite *U-basée* si pour tout problème d'unification c, $CSMGU_T(c)$ existe. Elle est dite *unitaire* si elle est U-basée et si $|CSMGU_T(c)| \leq 1$ pour tout problème d'unification P. Elle est dite *finitaire* si elle est U-basée et si $|CSMGU_T(c)|$ est fini pour tout problème d'unification c. Elle est dite *infinitaire* si elle est U-basée et si elle n'est pas finitaire.

1.4.2 Résolution de contraintes équationnelles et de contraintes de filtrage

Nous allons énumérer les résultats concernant la résolution de contraintes équationnelles et de filtrage qui nous intéresse dans cette thèse.

Les résultats concernant l'unification commutative, associative et associative-commutative suivants concernent des théories ne contenant qu'un seul symbole C, A ou AC.

1.4.2.1 Unification syntaxique

L'unification modulo la théorie vide est appelée *unification syntaxique*. Elle est décidable et unitaire. Les principales références concernant l'unification syntaxique sont (Robinson, 1965; Robinson, 1971; Huet, 1976; Martelli et Montanari, 1982; Fages, 1983; Lassez, Maher et Marriot, 1986; Paterson et Wegman, 1978). En particulier, un algorithme présenté par des règles de réécriture est donné dans (Jouannaud et Kirchner, 1991).

L'unification syntaxique peut être polynomiale ou exponentielle en espace et en temps suivant la structure de donnée choisie pour représenter les termes. Si les termes sont représentés par des DAG (*Directed Acyclic Graph*) et qu'il y a donc partage de structure dans la représentation des termes, l'unification syntaxique peut être linéaire en espace et en temps (Paterson et Wegman, 1978). L'utilisation des DAG n'impliquent pas la linéarité en temps de la résolution d'une contrainte.

1.4.2.2 Unification commutative

L'unification modulo C a été étudiée dans (Siekmann, 1979; Herold, 1985; Kirchner, 1986). Elle est décidable et finitaire.

Par exemple, la contrainte équationnelle $x + y =^?_C a + b$ où $+$ est un symbole de fonction commutatif a deux unificateurs plus généraux non comparables : $\{x \mapsto a,\ y \mapsto b\}$ et $\{x \mapsto b,\ y \mapsto a\}$.

Le problème de décision de l'unification modulo C c'est-à-dire connaître la satisfaisabilité d'un problème de l'unification modulo C est NP-complet. Le nombre d'éléments de l'ensemble des unificateurs plus généraux d'une contrainte équationnelle résolue dans la théorie C peut être exponentiel par rapport à la taille des termes de la contrainte à résoudre. Par exemple, la contrainte équationnelle $(x_1 + x_2) + (x_3 + x_4) =^?_C (a + b) + (c + d)$ où $+$ est un symbole de fonction commutatif admet 4! unificateurs plus généraux.

1.4.2.3 Unification associative

L'unification modulo A a été prouvée décidable (Makanin, 1977). Elle est infinitaire (Plotkin, 1972).

Des travaux concernant l'unification modulo A se trouvent dans (Plotkin, 1972; Siekmann, 1975; Livesey et Siekmann, 1975; Pécuchet, 1981; Abdulrab et Pécuchet, 1988; Abdulrab et Pécuchet, 1990; Jaffar, 1990). En particulier, dans (Plotkin, 1972), un algorithme d'unification non complet simple est donné. Cet algorithme peut être facilement implanté d'où son utilité dans une phase de prototypage.

Considérons l'exemple suivant. La contrainte $f(x,a) =^?_A f(a,x)$ a une infinité d'unificateurs plus généraux si f est un symbole associatif. Si nous définissons une séquence de termes $t_0, \ldots, t_n$ tels que $t_0 = a$ et $t_{i+1} = f(a,t_i)$ pour $i \geq 0$. Alors, l'ensemble complet d'unificateurs plus généraux de l'équation $f(x,a) =^?_A f(a,x)$ est l'ensemble $\{x \mapsto t_i \mid i \geq 0\}$.

1.4.2.4 Unification associative-commutative

L'unification modulo AC est décidable et finitaire. Les principaux algorithmes concernant l'unification modulo AC sont dûs à (Livesey et Siekmann, 1976; Stickel, 1976; Stickel, 1981). Ils sont basés sur la transformation des équations AC en équations diophantiennes homogènes.

Le problème de la décision d'un problème d'unification modulo AC est NP-complet (Kapur et Narendran, 1986). Le nombre d'unificateurs plus généraux d'une contrainte équationnelle $s =^?_{AC} t$ est doublement exponentiel par rapport à la taille des termes de la contrainte (Kapur et Narendran, 1992; Domenjoud, 1992). Par exemple, la contrainte $x+x+x+x =^?_{AC} y_1+y_2+y_3+y_4$, où $+$ est un symbole AC, x, y_1, y_2, y_3 et y_4 sont des variables, a 34 359 607 481 unificateurs plus généraux minimaux (Domenjoud, 1992).

1.4.2.5 Filtrage

Le filtrage et l'unification sont liés. On peut, sous certaines conditions, simuler le filtrage en utilisant l'unification et en considérant toutes les variables du terme t comme des constantes (Bürckert, 1989). Toutefois, il est préférable d'implanter le filtrage séparément de l'unification pour des raisons d'efficacité.

Le filtrage modulo AC est étudié dans (Hullot, 1979). Le problème de décision du filtrage modulo AC est NP-complet (Benanav, Kapur et Narendran, 1987) et le nombre de filtres d'un problème de filtrage modulo AC est exponentiel par rapport à la taille des termes du problème. Un algorithme de filtrage modulo AC a été développé par S. Eker (Eker, 1993). Il est utilisé dans ELAN (Vittek, 1994) et MAUDE (Clavel, Durân, Eker, Lincoln et Meseguer, 1998).

1.4.3 Contraintes d'ordre

La satisfaisabilité des contraintes d'ordre a été essentiellement étudiée dans le cas de contraintes d'ordre dans la théorie vide. Les deux raisons principales sont qu'il existe peu d'ordres T-compatibles (totaux sur les termes clos) connus jusqu'à présent et que les ordres T-compatibles (totaux sur les termes clos) connus sont difficilement utilisables pour la résolution d'inéquations. L'intérêt de ces ordres est incontestable pour la déduction automatique avec contraintes. Nous verrons également l'intérêt de ces ordres dans le chapitre 5.

Les travaux dans ce domaine concernent donc principalement la théorie vide et l'utilisation de LPO ou de RPO.

La négation d'une contrainte d'ordre peut être prise en compte par des transformations de la contrainte. Par exemple, une contrainte de la forme $\neg(x \succ^? y)$ peut être transformée en $(y \succ^? x \vee x =^? y)$.

La satisfaisabilité des contraintes d'ordre où $\succ$ est interprété comme l'ordre lexicographique sur les chemins, LPO, a été prouvée décidable par H. Comon (Comon, 1990). La méthode d'H. Comon a été mise au point pour prouver la décidabilité de ce problème, elle est inefficace en pratique. L'algorithme proposé transforme la contrainte en une contrainte en forme résolue. La forme résolue d'une contrainte d'ordre est: $\bigwedge_{i\in I}(x_i =^? t_i) \bigwedge_{j\in J}(x_j \succ^? t_j) \bigwedge_{k\in K}(t_k \succ^? x_k)$ où pour tout $i \in I$, x_i apparaît seulement une seule fois dans la contrainte, pour tout $j \in J \cup K$, x_j n'apparaît pas dans t_j. Une contrainte est en forme résolue si elle est $\top$, $\bot$ ou si elle est une disjonction de formes résolues. L'algorithme a été amélioré par R. Nieuwenhuis et A. Rubio dans (Rubio, 1994; Nieuwenhuis et Rubio, 1992b; Nieuwenhuis, 1993). L'efficacité pratique de la méthode de R. Nieuwenhuis et A. Rubio a été mise en évidence par son implantation dans SATURATE (Nivela et Nieuwenhuis, 1993). La méthode précédente a également été raffinée

pour prendre en compte le prédicat $\succeq$. La satisfaisabilité des contraintes d'ordre est un problème NP-complet pour les différentes méthodes proposées.

La satisfaisabilité des contraintes d'ordre où $\succ$ est interprété comme l'ordre récursif sur les chemins, RPO, a été prouvée décidable par J. Jouannaud et M. Okada (Jouannaud et Okada, 1991).

On pourra noter un travail original sur la génération d'une précédence pour rendre une contrainte d'ordre exprimée à l'aide de LPO satisfaisable par T. Genet et I. Gnaedig (Genet et Gnaedig, 1997).

1.5 Formules contraintes

Différents types de formules contraintes sont considérées dans cette section : les égalités, les règles de réécriture et les clauses. Nous précisons la sémantique de chaque type de formule. Les modèles de ces formules auxquels nous nous intéressons sont des algèbres générées par les termes.

Une formule contrainte est une paire $F\,[\![c]\!]$ composée d'une formule (égalité $s \approx t$, règle de réécriture $l \to r$ ou clause C) et d'une contrainte c construite sur $SL_T[\mathcal{F},\mathcal{P}]$. F est appelé le *squelette* de la formule contrainte.

Nous appelons $\sigma_1(F)\,[\![c_2]\!]$ un *rétracté* de la formule contrainte $F\,[\![c]\!]$ si pour toute substitution $\sigma \in CSS(c)$ et pour toute substitution $\sigma_2 \in CSS(c_2)$, $\forall x \in Dom(\sigma)$, $\sigma(x) \approx_T \sigma_2(\sigma_1(x))$. Par exemple, $g(f(y)) \approx b\,[\![y = {}^? a]\!]$ est un rétracté de $g(x) \approx b\,[\![x = {}^? f(a)]\!]$.

L'utilisation de formules contraintes est importante dans les méthodes de recherche de preuves ou de déduction automatique. Elle a divers avantages. En effet, les contraintes permettent de mettre en place des stratégies où la résolution des contraintes est différée. Elles peuvent réduire l'espace de recherche dans le sens où elles prennent en compte le partage de structure et ont un grand pouvoir d'expressivité.

Par exemple, une égalité contrainte rend possible la représentation d'un nombre fini ou infini de formules. L'égalité contrainte $f(x) \approx x\,[\![x + y =^?_C a + b]\!]$ représente l'égalité $f(a) \approx a$ et l'égalité $f(b) \approx b$, car la contrainte équationnelle $x + y =^?_C a + b$ a deux unificateurs plus généraux $\{x \mapsto a, y \mapsto b\}$ et $\{x \mapsto b, y \mapsto a\}$.

La formule contrainte $F[\![c]\!]$ représente l'ensemble des formules $S(F[\![c]\!]) = \{\sigma(F) \mid \sigma \in Sol(c)\}$ et $\sigma(F)$ est une *instance* de la formule $F\,[\![c]\!]$. Dans le cas clos, elle représente l'ensemble des formules closes $Gr(F\,[\![c]\!]) = \{\sigma(F) \mid \sigma \in Sol_G(c)\}$. Soit E un ensemble de formules contraintes. $Gr(E) = \bigcup_{e \in E} Gr(e)$.

1.6 ELAN

Le langage ELAN a été conçu à Nancy dans l'équipe PROTHEO au début des années quatre-vingt dix. ELAN peut être vu comme un cadre logique dont le noyau est la *logique de réécriture* (Meseguer, 1992) étendue avec la notion fondamentale de *stratégies* (Borovanský, Kirchner et Kirchner, 1996). Une théorie de réécriture et une stratégie d'application des règles de réécriture forment un *système de calcul.*

Système de calcul = Règles de réécriture + Stratégie

ELAN est un outil tout à fait adapté pour le prototypage de systèmes de calcul. Sa première version est décrite dans la thèse de M. Vittek (Vittek, 1994). Une présentation de la version

actuelle est disponible dans (Borovanský, Kirchner, Kirchner, Moreau et Ringeissen, 1998). Dans la première version, il n'existait qu'un interpréteur d'ELAN. Pour des raisons d'efficacité, un compilateur d'ELAN a été développé (Moreau et Kirchner, 1997; Moreau et Kirchner, 1998; Moreau, 1999).

Parmi les caractéristiques d'ELAN, on distingue particulièrement les suivantes :

- ELAN permet de séparer le calcul réalisé par les règles de réécriture du contrôle réalisé par les stratégies. Cette caractéristique est importante du point de vue de la compréhension des programmes. En effet, cela permet de modulariser les programmes, la partie calcul et la partie contrôle étant séparées. La partie calcul est technique et est explicitée par des règles de réécriture. La partie contrôle peut, quant à elle, être exprimée par des constructions de concaténations, d'itérations et de choix, qui forment un *langage de stratégies*.
- ELAN permet de faire des calculs non-déterministes et ainsi de travailler avec des systèmes de réécriture non confluents. En effet, les stratégies permettent d'exprimer des dérivations non-déterministes et de ne considérer que des sous-ensembles de toutes les dérivations possibles.
- Le style de programmation en ELAN suit une approche similaire à celle de la programmation fonctionnelle.

ELAN a été utilisé pour développer des applications variées.

- Du point de vue programmation, ELAN a été utilisé pour implanter la programmation logique avec contraintes (Kirchner et Ringeissen, 1998).
- Du point de vue preuve, on peut citer :
  - la SLD-résolution (Vittek, 1994),
  - la procédure de complétion de Knuth-Bendix (Kirchner et Moreau, 1995),
  - la complétion sans échec et la complétion basique avec simplification standard, basique et E-cycle que nous présentons dans le chapitre 4 (Lynch et Scharff, 1999b),
  - le prouveur de prédicat B (Cirstea et Kirchner, 1998), et
  - les travaux sur la terminaison des systèmes de réécriture en utilisant la résolution de contraintes d'ordre basées sur LPO (Genet et Gnaedig, 1997) et les automates (Genet, 1998).
- Du point de vue résolution de contraintes, ELAN a été testé sur plusieurs exemples de résolveurs de contraintes :
  - sur la combinaison d'algorithmes d'unification dans des théories arbitraires (Ringeissen, 1997),
  - sur un algorithme d'unification d'ordre supérieur (Borovanský, 1995), et
  - sur le résolveur de contraintes sur les entiers et les domaines finis, COLETTE (Castro, 1998a).

Dans le reste de cette section, nous présentons brièvement le langage ELAN, sa syntaxe et sa sémantique opérationnelle. Plus de détails techniques sont disponibles dans le manuel d'ELAN (Borovanský, Cirstea, Dubois, Kirchner, Kirchner, Moreau, Ringeissen et Vittek, 1998).

D'un point de vue programmation, ELAN se veut modulaire. Il offre la possibilité de définir :

- des théories de réécriture multi-sortées décrites par signatures et des règles de réécriture,
- des stratégies d'exécution sur ces règles de réécriture.

1.6.1 Signatures

Les signatures définissent des sortes et des opérateurs.

1.6.1.1 Sortes

L'exemple 4 présente la déclaration de deux sortes : `foo` et `constraint`.

Exemple 4

```
sort
        foo constraint;
end
```

1.6.1.2 Opérateurs

Les opérateurs représentent l'ensemble des symboles de fonction de la signature. Ils sont définis en utilisant une notation *mix-fix*. La déclaration d'un opérateur comporte des informations sur sa syntaxe et sa sémantique.

- D'un point de vue syntaxique,
  - on peut définir un opérateur comme étant un *alias* d'un autre opérateur.
  - on peut spécifier la priorité syntaxique d'un opérateur par rapport à un autre à l'aide de l'option de priorité *PRI*.
  - on peut définir la visibilité d'un opérateur dans d'autres modules. L'opérateur peut ainsi être défini en *local* et n'être utilisable que dans le module où il est défini, ou en *global* et être utilisable dans d'autres modules.
  - on peut définir l'associativité syntaxique à gauche ou à droite d'un opérateur en utilisant l'option *assocLeft* ou *assocRight*.
- D'un point de vue sémantique, on peut spécifier qu'un opérateur est un symbole libre ou un symbole *AC*.

L'exemple 5 illustre la déclaration d'opérateurs avec des attributs syntaxiques et sémantiques.

Exemple 5 *A partir des sortes de l'exemple 4, nous définissons comment sont construits des termes de sortes* `foo` *et* `constraint`. *a, b, c et T sont des termes de sorte* `foo`. *On spécifie que tout terme de sorte* `foo` *est aussi un terme de sorte* `constraint` *en utilisant l'opérateur de coersion* `@`.

```
operators
global
  a       : foo;
  b       : foo;
  c       : foo;
  d       : foo;
  T       : foo;
  @       : (foo) constraint;
  @ V @   : (constraint constraint) constraint  assocLeft (AC)  pri 10;
  @ & @   : (constraint constraint) constraint  assocLeft (AC)  pri 20;
  &(@,@)  : (constraint constraint) constraint  alias @ & @:;
end
```

Ainsi, par exemple, a & b et a V b & T sont des termes de sorte `constraint`.

Une expression de la forme a & b & c V d V b représente l'expression (a & b & c) V d V b à cause des priorités.

1.6.2 Règles de réécriture

La réécriture est la base du langage ELAN. ELAN permet de définir des *règles de réécritures conditionnelles* avec des *affectations locales.* L'intérêt de ces affectations est d'affecter à une variable locale le résultat de l'application d'une stratégie sur un terme. Les règles de réécriture sont des *règles de réécritures nommées* ou des *règles de réécriture non-nommées.*

Les règles non-nommées sont utilisées pour la normalisation des termes. Leur application n'est pas contrôlée par l'utilisateur mais elles sont appliquées par une stratégie fixée du langage en profondeur d'abord et le plus à gauche (*built-in left-most inner-most*). L'ensemble des règles non-nommées est supposé être confluent et terminant.

Les règles nommées sont contrôlées par les stratégies définies par l'utilisateur. Ces stratégies définies à partir des règles nommées sont appelées *stratégies élémentaires.* L'ensemble des règles nommées n'est pas nécessairement confluent et terminant. L'utilisateur définit ainsi grâce à des stratégies, les dérivations qu'il considère. On peut noter que dans un programme ELAN, il peut y avoir plusieurs règles ayant le même nom.

La syntaxe des règles de réécriture est la suivante :

$$\begin{array}{llll} [\ell] & l & \Rightarrow & r \\ & & & ifwhere \\ \textbf{end} & & & \end{array}$$

où :

- ℓ est le nom de la règle (ℓ est vide dans le cas d'une règle non-nommée),
- l et r sont respectivement le membre gauche et le membre droit de la règle de réécriture, et
- $ifwhere ::= \{\textbf{if}\ v \mid \textbf{where}\ y\ :=\ (S)u \mid \textbf{where}\ y\ :=\ ()u\}^*$ où :
  - **if** v est une condition booléenne,
  - **where** $y\ :=\ (S)u$ est l'affectation à la variable y du résultat de l'application de la stratégie S sur le terme u, et
  - **where** $y\ :=\ ()u$ est l'affectation à la variable y du résultat de l'application de la normalisation du terme u.

L'exemple 6 présente des règles nommées en ELAN.

Exemple 6 *En utilisant la signature présentée dans les exemples 4 et 5, nous définissons les règles $Simplif1$ et $Simplif2$ qui permettent de simplifier une contrainte.*

```
rules for constraint
  x, y : constraint;
 global
[Simplif1] x V T => T
end
[Simplif2] x & y V T => T
end
 end
```

Lorsque plusieurs règles du système de réécriture décrites ont les même membres gauches, on peut factoriser les règles en utilisant la construction *try-choose.* Cette construction supprime les calculs redondants de filtrage et l'exécution de parties de code communes dans plusieurs règles.

La syntaxe des règles de réécriture dans ce cas est la suivante :

$$\begin{array}{lll} [\ell] & l \Rightarrow r & \\ & & ifwherechoose \\ \textbf{end} & & \end{array}$$

où, $ifwherechoose ::= \{ifwhere \mid \textbf{choose}\ \{\textbf{try}\ ifwherechoose\}^{+} end\}^{*}$

1.6.3 Stratégies élémentaires

La notion de stratégie est la principale originalité d'ELAN. En pratique, une stratégie définit comment les règles de réécriture sont appliquées sur un terme. L'application d'une stratégie sur un terme est la collection de toutes les dérivations issues de ce terme en utilisant la stratégie. Une stratégie énumère donc les termes qu'elle décrit, que le nombre de termes soit fini ou non. Quand l'application d'une stratégie à un terme retourne un ensemble vide de termes, on dit que la stratégie *échoue* (*fail*).

L'utilisateur peut définir des stratégies élémentaires à l'aide du *langage de stratégies* que nous décrivons ci-dessous. Toutefois, il existe également des stratégies non élémentaires. Nous pouvons citer la *stratégie de normalisation* qui est utilisée implicitement dans ELAN et qui applique les règles de réécriture non-nommées en profondeur et le plus à gauche (*left-most inner-most*) sur un terme. Il existe également les *stratégies définies* qui sont des stratégies élaborées mises au point par l'utilisateur. Dans la pratique, ce sont les stratégies élémentaires qui sont les plus naturelles à mettre au point.

Une stratégie a une sorte. La sorte d'une stratégie est de la forme $\langle s_1 \mapsto s_2 \rangle$. Cette stratégie est donc appliquée à des termes de sorte s_1 et donne des résultats de sorte s_2. On pourra noter qu'une stratégie peut s'exprimer à l'aide d'autres stratégies, que nous appelons *sous-stratégies*. Les sous-stratégies d'une stratégie sont alors de la même sorte que la stratégie.

Le langage de stratégies élémentaires comporte :

- un constructeur de stratégie *identité* : **id**,
- un constructeur de stratégie *échec* : **fail**,
- un constructeur de concaténation de stratégies introduit par le mot-clé ;,
- des opérateurs de choix de stratégies : **dk** (*Don't Know*), **dc** (*Don't Care*), **dc one**, **first** et **first one**,
- des opérateurs d'itérations de stratégies : **repeat***, **repeat+**, **iterate***, **iterate+**, et
- un opérateur de normalisation : **normalise**.

La sémantique de ces stratégies est décrite ci-dessous.

- **Stratégie id**

  La stratégie *id* correspond à l'identité, elle retourne le terme donné en entrée.
- **Stratégie fail**

  La stratégie *fail* correspond à l'échec, elle échoue toujours.
- **Concaténation de stratégies**

  La stratégie $S_1; S_2$ retourne les résultats de l'application de la stratégie S_2 aux résultats de l'application de la stratégie S_1 au terme donné en entrée.
- **Choix de stratégies**

  dk La stratégie $dk(S_1, \ldots, S_n)$ retourne tous les résultats de chacune des stratégies S_i appliquées au terme donné en entrée. Elle échoue si toutes les stratégies S_i échouent.

dk one La stratégie *dk one*$(S_1, \ldots, S_n)$ renvoie le premier des résultats de chacune de stratégies S_i appliquées au terme donné en entrée. Elle échoue si toutes les stratégies S_i échouent.

dc La stratégie $dc(S_1, \ldots, S_n)$ donne tous les résultats d'une stratégie S_i qui n'échoue pas choisie aléatoirement. Elle échoue si toutes les stratégies S_i échouent.

dc one La stratégie *dc one*$(S_1, \ldots, S_n)$ donne le premier des résultats d'une stratégie S_i qui n'échoue pas choisie aléatoirement. Elle échoue si toutes les stratégies S_i échouent.

first La stratégie $first(S_1, \ldots, S_n)$ donne tous les résultats de la première stratégie S_i qui n'échoue pas. Elle échoue si toutes les stratégies S_i échouent.

first one La stratégie *first one*$(S_1, \ldots, S_n)$ donne le premier des résultats de la première stratégie S_i qui n'échoue pas. Elle échoue si toutes les stratégies S_i échouent.

- **Itération de stratégies**

repeat* La stratégie $repeat*(S)$ itère la stratégie S sur le terme donné en entrée jusqu'à ce qu'elle échoue. Les termes résultats proviennent de la dernière stratégie itérée qui n'a pas échoué. Elle correspond à l'application de la stratégie S^i sur le terme donné en entrée, où i est tel que S^{i+1} échoue. La stratégie S^i peut également ne jamais échouer.

S^i peut être exprimée à l'aide de l'opérateur de concaténation ; par $S^i = \overbrace{S; \ldots; S}^{i\ fois}$. Cette stratégie n'échoue jamais, car elle renvoie toujours au moins le terme donné en entrée.

repeat+ La stratégie $repeat+(S)$ correspond $S; repeat*(S)$. Elle échoue si S échoue sur le terme donné en entrée.

iterate* La stratégie $iterate*(S)$ correspond à appliquer S sur le terme donné en entrée zéro, une, deux, ..., i fois. i est tel que l'application de $i+1$ fois la stratégie S sur le terme donné en entrée échoue. La stratégie peut également ne jamais échouer. Cette stratégie renvoie tous les résultats un par un. L'ensemble des résultats est ainsi l'ensemble des résultats des stratégies id, S, S^2, ..., S^i appliquées au terme donné en entrée. $iterate*(S)$ correspond à $dk(id\ ,\ S\ ,\ S;S\ ,\ S;S;S\ ,\ \ldots)$. La stratégie $repeat*(S)$ n'échoue jamais car elle renvoie toujours au moins le terme donné en entrée pour chaque sous-stratégie S^i $(i \geq 0)$.

iterate+ La stratégie $iterate+(S)$ correspond à appliquer S sur le terme donné en entrée une, deux, ..., i fois. i est tel que l'application de S^{i+1} sur le terme donné en entrée échoue. Elle renvoie tous ces résultats un par un. Cette stratégie correspond à $S; iterate*(S)$. Elle échoue si S échoue sur le terme donné en entrée.

- **Normalisation** La stratégie *normalize*(S) normalise un terme en profondeur et le plus à gauche (*left-most inner-most*) par rapport à une sous-stratégie S.

L'exemple 7 présente une stratégie en ELAN.

Exemple 7 *En utilisant les règles définies dans l'exemple 6, nous définissons la stratégie* $Simplif$*. La sous-stratégie* $first$ *one*$(Simplif1, Simplif2)$ *retourne le premier résultat de l'application de la première règle parmi* $Simplif1$ *et* $Simplif2$ *qui n'échoue pas. La stratégie* $Simplif$*, en utilisant*

l'opérateur repeat, applique cette sous-stratégie jusqu'à ce qu'elle ne puisse plus être appliquée.*

```
stratop
global
 Simplif : <constraint -> constraint> bs;
end

strategies for constraint
implicit
 [] Simplif => repeat*( first one(Simplif1, Simplif2))
 end
end
```

Le mot-clé bs *est ajouté pour montrer que la stratégie* Simplif *est définie par l'utilisateur. L'application de la stratégie Simplif sur le terme a & b V c V T & d V a, correspondant à (a & b) V c V (T & d V a), s'écrit (Simplif)a & b V c V T & d V a. Elle est évaluée à a & b V c V T & d V a. (Simplif)a V T est évaluée à T. (Simplif)a & b V T V a est aussi évaluée à T.*

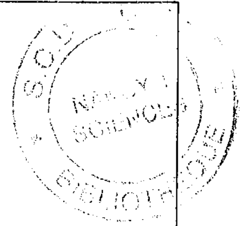
1.6.4 Modularité

ELAN est un langage modulaire. La modularité d'ELAN s'exprime à différents niveaux.

- Un programme ELAN peut être composé de plusieurs modules qui définissent des théories de réécriture. Ils sont composés de définitions de sortes, de symboles de fonctions, de règles de réécriture et de stratégies. Ces modules sont des fichiers d'extension *.eln*.
- Il est possible de restreindre la visibilité de certains éléments des modules. Un module ELAN peut importer avec la commande *import* d'autres modules et les règles de visibilité sont les suivantes. Les sortes sont toujours globales. Les opérateurs, règles de réécriture et stratégies peuvent être locaux à un module et dans ce cas, on utilise le mot-clé *local*. Ils peuvent être globaux et être utilisés dans les modules qui importent le module où ils sont déclarés et dans ce cas, on utilise le mot-clé *global*.
- ELAN possède une librairie de sortes, de fonctions et de stratégies.
- Les modules peuvent être paramétrés. L'exemple 8 illustre cette caractéristique.

Exemple 8 *Nous définissons ici le module* set.eln *qui représente un ensemble d'éléments*

de sorte X. La sorte X est le paramètre.



```
module set[X]

sort
 X
 set[X];
end

operators
global
 emptyset : set[X];
 @        : (X) set[X];
 @ u @    : (set[X] set[X]) set[X]  (AC);
end

rules for set[X]
global
[] emptyset u emptyset         => emptyset
end
end

end
```

Nous obtenons un ensemble d'entiers en instanciant la sorte X par la sorte int.

```
module setint

import
global
 set[int]
end

end
```

Chapitre 2

Recherche de preuve et déduction en logique du premier ordre avec égalité

Sommaire

2.1 Stratégies de recherche de preuve **32**
2.2 La saturation **35**
2.3 Procédures de complétion **37**
2.3.1 Complétion de Knuth-Bendix 39
2.3.2 Complétion sans échec 43
2.3.3 Complétion modulo une théorie équationnelle 49
2.3.4 Complétions contrainte et basique dans la théorie vide 51
2.3.5 Complétion contrainte et basique modulo une théorie équationnelle . . . 64
2.4 Résolution et paramodulation **66**
2.4.1 Résolution 67
2.4.2 Paramodulation 69
2.5 Parallélisation des procédures de recherche de preuve et de déduction 72
2.5.1 Grain du parallélisme 74
2.5.2 Coopération de systèmes de recherche de preuve ou de déduction 76

Dans ce chapitre, nous donnons d'abord un cadre général pour les définitions de base concernant la recherche de preuve de théorèmes et la déduction automatique en logique du premier ordre. Les notions de systèmes d'inférence, de stratégies de recherche de preuve et de stratégies de déduction sont ainsi définies. L'objet de cette thèse est l'étude des procédures de déduction automatique en logique du premier ordre basées sur l'utilisation de la saturation, nous donnons donc ici les notions fondamentales concernant la saturation. Le cadre général décrit peut être instancié: les cas des procédures de complétion, de la résolution et de la paramodulation sont détaillés. En particulier, comme nous avons implanté une procédure de complétion standard avec simplification standard, une procédure de complétion basique avec simplification standard et une procédure de complétion basique avec simplification basique (avec et sans rétraction) en ELAN dans le système *ECC* (*Complétion E-cycle*), nous utilisons au fil de ce chapitre ELAN comme formalisme pour décrire les différentes procédures de complétion implantées. Ce chapitre décrit également les principales approches utilisées pour la parallélisation des procédures de déduction automatique et de preuve automatique de théorèmes. Ceci nous permettra de comparer notre développement d'une procédure de complétion close concurrente de grain fin basée sur l'utilisation des graphes SOUR (Kirchner et al., 1996a) décrite dans le chapitre 3 aux approches existantes.

2.1 Stratégies de recherche de preuve

Dans cette section, nous donnons un cadre général suivant la terminologie de M.P. Bonacina (Bonacina, 1999a; Bonacina, 1999b) et la philosophie ELAN (Borovanský, 1998) pour les définitions de base concernant le domaine des *preuves automatiques de théorèmes.* Nous définissons ce qu'est un *prouveur de théorème.* Dans cette définition, deux aspects sont à prendre en compte. Le *système d'inférence* donne les règles de construction de preuve, mais le composant *stratégie de recherche d'une preuve* est souvent oublié. La stratégie de recherche représente la partie contrôle de l'application des règles d'inférence. Les stratégies de recherche sont classées suivant l'espace de recherche et les dérivations qu'elles génèrent.

Un *prouveur de théorèmes* a pour objectif de déterminer, étant donnés une conjecture α et un ensemble d'*hypothèses* $\mathcal{H}$, si α est une *conséquence logique* de $\mathcal{H}$ (noté $\mathcal{H} \models \alpha$), c'est-à-dire si α est valide (vraie) dans $\mathcal{H}$. Un prouveur de théorèmes peut être complètement automatique, ou interactif et nécessiter une intervention humaine ou l'intervention d'un autre prouveur de théorèmes. Nous nous plaçons ici dans le cas de prouveurs de théorèmes complètement automatiques. Nous supposons que les formules, la conjecture α et les hypothèses, sont écrites dans un langage du premier ordre basé sur une signature contenant des variables, des symboles de fonctions et des symboles de prédicats.

L'approche recherche de preuve pour ce problème est de montrer qu'il existe une *preuve* du *but* α à partir des hypothèses de $\mathcal{H}$ (noté $\mathcal{H} \vdash \alpha$) ou qu'il existe une preuve par réfutation (contradiction) de α (noté $\mathcal{H}, \neg\alpha \vdash \Box$) ou de réfuter α en trouvant un modèle de $\mathcal{H} \cup \neg\alpha$.

Nous considérons ici une preuve comme une suite de formules connectées logiquement par l'application d'une *règle d'inférence.* Les règles d'inférence sont notées :

$$\textbf{Label} \qquad \frac{\alpha_1 \ \cdots \ \alpha_n}{\alpha}$$

où, *Label* est le nom de la règle d'inférence, et si les prémisses données sont de la forme $\alpha_1, \ldots, \alpha_n$, alors on déduit une conséquence (conclusion) de la forme α.

Nous considérons dans ce document les systèmes d'inférence de complétion, de résolution et de paramodulation.

Il reste alors à faire le lien entre la validité d'une formule et une preuve présumée de cette formule. Le lien est créé par les notions de *correction* et de *complétude.*

Un système d'inférence est *correct* si toute règle d'inférence le composant est correcte, c'est-à-dire que pour toute interprétation I satisfaisant ses prémisses, alors I satisfait sa conclusion. On peut alors garantir que toute formule ayant une preuve suivant ce système d'inférence est valide. Inversement, un système d'inférence est *complet* si toute formule valide a une preuve. La notion de *complétude réfutationnelle* a été définie pour les approches de preuves basées sur les preuves par contradiction. Un système d'inférence est *réfutationnellement complet* si quand une formule est valide, alors il existe une preuve de $\Box$ à partir de l'ensemble des hypothèses et de la négation de cette formule. La correction est une propriété cruciale des systèmes d'inférence, il s'agit en effet de ne pas prouver une formule fausse. La complétude garantit de trouver une preuve d'une formule si une preuve existe. Si aucune preuve n'est trouvée, alors c'est que la formule donnée n'est pas valide. La complétude est optionnelle, elle a un coût. Il faut alors choisir entre payer ce coût ou non. Ce point est discuté dans l'introduction du chapitre 4.

Le problème est maintenant de savoir comment mener la recherche d'une preuve.

En effet, lors de la recherche d'une preuve, le deuxième composant à prendre en compte, est la *stratégie de recherche d'une preuve*, également appelée *plan de recherche d'une preuve* dans la littérature.

La construction d'une preuve peut être vue comme une *transition* d'un état dans un autre, chaque état étant décrit par des formules et pour passer d'un état dans un autre, une règle d'inférence est appliquée suivant une stratégie définie. La stratégie de recherche représente la *partie contrôle*. Elle sélectionne la prochaine règle d'inférence à appliquer et les prémisses sur lesquelles l'inférence va s'appliquer. Les transitions ont la forme suivante.

$$E \rightarrow_{(Label,Strat)} E'$$

où :

- E et E' sont des ensembles de formules,
- *Label* est le nom de la règle d'inférence appliquée dans la transition, et
- *Strat* est la stratégie de recherche de preuve utilisée pour appliquée les règles d'inférences.

On omet les étiquettes *Label* et *Strat* si elles sont évidentes dans le contexte.

Une *dérivation* est une suite (finie ou infinie) E_0, $E_1, \ldots$ telle que E_{i+1} soit obtenue de E_i par la transition $E_i \rightarrow_{(Label_i,Strat)} E_{i+1}$.

Partant d'un état et étant donnés un système d'inférence et une stratégie, différentes *dérivations* peuvent être générées. La stratégie sélectionne certaines dérivations parmi toutes les dérivations possibles. La stratégie permet donc de gérer l'*indéterminisme*.

Le couple règles d'inférence et stratégie de recherche détermine un *prouveur de théorèmes*.

Prouveur de théorèmes = Règles d'inférence + Stratégie de recherche de preuve

Cette définition montre qu'ELAN est un langage adapté pour le prototypage de prouveurs de théorèmes. En effet, en ELAN, nous avons une égalité semblable, *Système de calcul = Règles de réécriture + Stratégies*. Les règles d'inférences sont décrites par des règles de réécriture nommées et la stratégie de recherche de preuve est représentée par une stratégie élémentaire ELAN construite à partir des règles nommées.

Un système d'inférence doit être correct, voire complet. De la même manière, une stratégie de recherche doit être *équitable*. Une stratégie de recherche est *équitable* si toute preuve suivant le système d'inférence est bien générée suivant la stratégie. Un système d'inférence peut être incomplet mais être appliqué équitablement. Par contre, un système d'inférence peut être complet mais appliqué sans équité, il « perd » alors sa complétude.

Les stratégies de recherche de preuve peuvent être classées. Nous utilisons ici la classification proposée par M.P. Bonacina (Bonacina, 1992; Bonacina, 1999a; Bonacina, 1999b). On distingue le raisonnement *en avant* (*forward*) et le raisonnement *en arrière* (*backward*). Une stratégie de raisonnement *en avant* se sert principalement des hypothèses et une stratégie de raisonnement *en arrière* se sert principalement de la formule à prouver pour aboutir à une preuve. Cette classification ne suffit pas pour définir deux types de stratégies bien délimitées. Par exemple, la résolution qui utilise un *ensemble de support* (*set of support* abrégé en *sos*) a un comportement de raisonnement *en avant* si l'ensemble de support n'est composé au départ que d'hypothèses et de raisonnement *en arrière* si l'ensemble de support contient le but à prouver (voir la section 2.4.1). La classification a donc été raffinée en partant de la constatation que certaines stratégies génèrent des formules utilisables pour prouver différents buts et d'autres stratégies développent une tentative de preuve pour un but donné et en cas d'échec utilisent le retour-arrière (*backtracking*) pour passer à une autre tentative de preuve de ce but. M.P. Bonacina a ainsi défini les *stratégies d'ordre* et les *stratégies de réduction de sous-buts*.

- Les *stratégies d'ordre* sont les stratégies qui génèrent des formules utilisables pour prouver

différents buts. Elles s'appuient sur un ordre bien-fondé pour ordonner les formules et éventuellement, pour supprimer des formules de l'espace de recherche. On distingue dans cette classe les stratégies d'ordre d'*expansion* et les stratégies d'ordre de *contraction*. Une stratégie de contraction privilégie les techniques qui permettent de supprimer des formules de l'espace de recherche, à l'inverse des stratégies d'expansion.

- Une stratégie d'expansion privilégie les transitions d'expansion de la forme suivante.

$$E \cup \{\alpha\} \rightarrow_{(Label,Strat)} E \cup \{\alpha,\beta\}$$

  Dans ce cas, la règle d'inférence *Label* est appelée règle d'expansion.
- Une stratégie de contraction privilégie les transitions de contraction des deux formes suivantes.

$$E \cup \{\alpha\} \rightarrow_{(Label,Strat)} E \cup \{\beta\} \ (1)$$

  où $E \cup \{\alpha\} \succ_{mult} E \cup \{\beta\}$, et

$$E \cup \{\alpha\} \rightarrow_{(Label,Strat)} E \ (2)$$

  La règle d'inférence *Label* est appelée règle de contraction. Plus particulièrement, dans le premier cas, on a affaire à une *règle d'inférence de simplification* et dans le deuxième cas, à une *règle d'inférence de suppression* (*deletion*). La formule α est dite *redondante*. Pour un système d'inférence I complet contenant des règles d'expansion et des règles de contraction, on dit que la stratégie de contraction induite par les règles de contraction de I est complète.
  La mise au point de systèmes d'inférence complets et de stratégies de contraction complètes est un problème difficile et coûteux. L'objet de cette thèse est la mise au point de tels systèmes et de telles stratégies.

- Les *stratégies de réduction de sous-buts* développent une tentative de preuve pour un but donné et en cas d'échec utilisent le retour-arrière pour passer à une autre tentative de preuve de ce but. Ces stratégies n'utilisent pas d'ordre sur les formules et ne peuvent pas être des stratégies de contraction.

Les procédures de complétion, de paramodulation et certaines formes de résolution (résolution positive, hyper-résolution, résolution avec ensemble de support) sont des méthodes de recherche de preuve utilisant des stratégies d'ordre. Les prouveurs tels que Otter (McCune, 1994), RRL (Kapur et Zhang, 1991), REVEAL (Anantharaman et Bousdira, Fall-1992), EQP (McCune, 1997a), SPASS (Weidenbach et al., 1996), SATURATE (Nivela et Nieuwenhuis, 1993), DATAC (Vigneron, 1994a; Vigneron, 1998)...sont basées sur ces méthodes. Certaines procédures de résolution (résolution linéaire), les méthodes des tableaux et d'élimination de modèle sont des méthodes de recherche de preuve utilisant des stratégies de réduction de sous-buts. Elles sont employées dans les prouveurs de théorèmes tels que SETHEO (Letz, Mayr et Goller, 1992), METEOR (Astrachan et Loveland, 1991), PROTEIN (Baumagarter et Bruening, 1994)...

Une comparaison des stratégies d'ordre et de réduction de sous-buts ne permet pas de dire qu'une des stratégies est meilleure que l'autre, chacune a ses avantages et ses inconvénients. L'un des critères de jugement d'une stratégie est la taille de l'espace de recherche qu'elle développe. L'espace de recherche peut dans les deux cas être infini, la preuve ne représentant qu'une partie restreinte de cet espace. Dans les stratégies d'ordre, la taille de l'espace de recherche est contrôlée

par les contractions alors que dans les stratégies de réduction de sous-buts, l'espace de recherche utilisé est restreint mais, on doit prendre en considération toutes les tentatives de recherches de preuve induites par le retour-arrière. Outre cette classification, on peut ajouter d'autres éléments qui vont permettre de caractériser une stratégie, par exemple, le fait qu'une stratégie soit *orientée but*, ce qui est fondamental en recherche de preuve.

Les notions présentées concernent le domaine des preuves automatiques de théorèmes, ces même notions sont valables pour le domaine de la *déduction automatique* où le problème n'est pas la recherche d'une preuve mais seulement la *déduction* de formules à partir d'un ensemble de formules en suivant un système d'inférence et une stratégie. Chaque étape de déduction est connectée par l'application d'une règle d'inférence en suivant une *stratégie de déduction.*

2.2 La saturation

La saturation a été introduite par J.A. Robinson (Robinson, 1965). Intuitivement, la *saturation* est une procédure qui ajoute des clauses non redondantes à un ensemble de clauses et supprime les clauses redondantes pour un système d'inférence $\mathcal{I}$. La saturation produit des ensembles saturés, c'est-à-dire où il n'est plus possible ni d'ajouter ni de supprimer des clauses. L. Bachmair et H. Ganzinger (Bachmair et Ganzinger, 1994) ont introduit la notion de *saturation* d'un ensemble de formules pour la déduction mais aussi pour les méthodes de recherche de preuve basées sur la complétion, la paramodulation et leurs dérivées.

Nous donnons maintenant les notions fondamentales concernant la saturation.

Le concept de *redondance* détermine les formules inutiles dans le processus de saturation. Les formules *redondantes* sont supprimées de l'espace de recherche. Cette notion permet de mettre au point la plupart des règles de contraction. Intuitivement, une formule est redondante si elle est impliquée par d'autres formules plus petites par rapport à un ordre bien-fondé. Les formules redondantes déterminent les inférences redondantes.

Définition 3 – *Soit F une formule close et $\succ$ un ordre bien-fondé.*
F est redondante *dans $E = \{F_1, \ldots, F_n\}$ si:*

- *pour toute formule $F_i \in E$, $F_i \prec F$ et*
- *$F_1, \ldots, F_n \models F$.*

- *Une formule est redondante si toutes ses instances closes sont redondantes.*
- *Une inférence est redondante si l'une de ses prémisses ou sa conclusion est redondante.* □

La notion de redondance permet de définir un ensemble de formules *saturé.*

Définition 4 *Un ensemble de formules E est saturé si toutes les inférences impliquant des formules de E sont redondantes.* □

La notion de *persistance* découle de la définition précédente.

Définition 5 *Soit une dérivation issue de E, $E_0 = E$, E_1, ... en utilisant un système d'inférence I et une stratégie de déduction (ou de recherche de preuve) Strat.*

Une formule F est dite persistante, *s'il existe un j tel que pour tout $k \geq j$, $F \in E_k$. L'ensemble des éléments persistants de la dérivation est noté E_∞, il est appelé la* limite *de la dérivation et est défini par $E_\infty = \bigcup_j \bigcap_{k \geq j} E_k$.* □

L'ensemble E_∞ nous permet de définir la notion d'*équité*. Une stratégie de déduction (ou de recherche de preuve) est équitable si les inférences entre des formules non redondantes de E_∞ ne sont pas différées indéfiniment.

Définition 6 *Soit une dérivation issue de E, $E_0 = E$, E_1, ... en utilisant un système d'inférence I et une stratégie de déduction (ou de recherche de preuve) Strat. La stratégie Strat est* équitable, *si E_∞ est saturé.* □

La saturation d'un ensemble de formules E par rapport à un système d'inférence I et une stratégie de déduction (ou de recherche de preuve) *Strat* nous permet de définir la *complétude* de I. Nous donnons ici deux définitions de la complétude, une dans le cas de la complétion et une dans le cas de la résolution et de la paramodulation[4].

Définition 7 *Soit une dérivation issue de E où E est un ensemble de formules non contraintes, $E_0 = E$, E_1, ... en utilisant un système d'inférence I et une stratégie de déduction (ou de recherche de preuve) Strat équitable.*

- **Complétion** *Si I est un système d'inférence de complétion, alors I est* complet *si $Gr(E_\infty)$ est convergent.*
- **Résolution et paramodulation** *Si I est un système d'inférence de résolution ou de paramodulation, alors I est* réfutationnellement complet *si : si E_∞ ne contient pas la clause vide, alors il est possible de construire un modèle pour E_∞.* □

L. Bachmair et H. Ganzinger ont défini une méthode de preuve pour établir la complétude des systèmes d'inférence utilisés dans les méthodes par saturation (Bachmair et Ganzinger, 1998).

Cette méthode de preuve de complétude est basée sur les notions d'interprétation et de *modèle d'égalités de Herbrand.* Une *interprétation d'égalités de Herbrand* (*equality Herbrand interpretation*) est une congruence sur les termes clos qui satisfait l'axiome de réflexivité ($x \approx x$), l'axiome de symétrie ($x \not\approx y \vee y \approx x$), l'axiome de transitivité ($x \not\approx y \vee y \not\approx z \vee x \approx z$) et l'axiome de congruence ($x \not\approx y \vee f(\ldots,x,\ldots) \approx f(\ldots,y,\ldots)$) pour tous les symboles de fonctions f de la signature. Une formule close e est *valide* (vraie) dans l'interprétation de Herbrand $\mathcal{H}$, si, soit $A \in \mathcal{H}$ pour un atome A de e, soit $A \notin \mathcal{H}$ pour un littéral $\neg A$ de e. On dit aussi que $\mathcal{H}$ *satisfait* e. Une formule non close e est *valide* (vraie) dans $\mathcal{H}$, si toutes ses instances closes sont valides dans $\mathcal{H}$. Une interprétation de Herbrand est un modèle de Herbrand d'un ensemble de formules si elle satisfait tous les éléments de cet ensemble. Un ensemble est *satisfaisable* s'il admet un modèle sinon il est insatisfaisable. La clause vide, traditionnellement notée □, est insatisfaisable. Une formule e est une *conséquence logique* d'un ensemble de formules E, si e est valide dans tout modèle de E. Nous le notons $E \models e$.

La méthode de preuve de complétude de L. Bachmair et H. Ganzinger construit ainsi un modèle d'égalités de Herbrand $\mathcal{H}$ pour les instances closes de l'ensemble saturé $Gr(E_\infty)$. $\mathcal{H}$ est construit à partir d'un ordre de réduction total sur les termes clos $\succ$. $\mathcal{H}$ est un système de réécriture clos et convergent équivalent à $Gr(E_\infty)$, ce qui prouve que $Gr(E_\infty)$ est convergent (voir le théorème 3). Cette représentation de $\mathcal{H}$ permet de caractériser les preuves et la validité des formules de $Gr(E_\infty)$ en utilisant la réécriture. Dans cette preuve, il est nécessaire de faire le lien entre le cas clos et le cas non clos. Cette connexion est appelée *passage du cas clos au cas non clos* (*lifting*). Il faut montrer que toute formule persistante de $Gr(E_\infty)$ est une instance

4. Les systèmes d'inférence de la complétion, de la résolution et de la paramodulation n'ont pas encore été introduits, mais nous pouvons définir toutes les notions concernant la saturation sans connaître précisément les systèmes d'inférence.

close d'une formule persistante de E_∞ et que toute inférence entre éléments de $Gr(E_\infty)$ peut être obtenue par l'instanciation d'une inférence entre éléments de E_∞. Nous utilisons cette technique dans la partie 4 pour prouver la complétude du système d'inférence *BCES*.

Dans le cas de la résolution et de la paramodulation, où il s'agit de prouver la complétude réfutationnelle, cette méthode de preuve de complétude est étendue. Un modèle d'égalités de Herbrand est construit comme précédemment. Dans la preuve, il faut ajouter le test de la *réduction des contre-exemples* pour le système d'inférence considéré. Ce test consiste à vérifier que pour tous les plus petits contre-exemples C de $Gr(E_\infty)$ faux dans $\mathcal{H}$, C est réductible en un plus petit contre-exemple par les règles du système d'inférence considéré. Les systèmes d'inférence possédant cette propriété sont réfutationnellement complets.

Cette technique de preuve nécessite que l'ordre de réduction $\succ$ soit bien-fondé et total sur les termes clos. Récemment, M. Bofill, G. Godoy, R. Nieuwenhuis et A. Rubio ont étendu cette méthode en exigeant seulement de l'ordre de réduction qu'il soit bien fondé et qu'il possède la propriété de sous-terme (Bofill, Godoy, Nieuwenhuis et Rubio, 1999). La propriété de réduction est la condition minimale sur l'ordre, car, comme nous l'avons vu dans le théorème 4, un système de réécriture termine si et seulement si il est contenu dans un ordre de réduction.

2.3 Procédures de complétion

Le problème considéré ici est le problème du mot, c'est-à-dire déterminer la validité d'une égalité à partir d'un ensemble d'égalités. La complétion est utilisée pour résoudre ce problème. Elle transforme un ensemble d'égalités E en utilisant des *règles d'inférence* I et une *stratégie* $Strat$. La construction d'une *dérivation de complétion* à partir de E, I, $Strat$ et d'un ordre de réduction $\succ$ détermine une *procédure de complétion.* Les tâches élémentaires capturées par les règles d'inférence de I sont la réécriture, l'unification, l'orientation et le calcul de paires critiques obtenu par superposition de deux égalités.

On distingue deux classes de procédures de complétion. Certaines procédures de complétion ont pour but unique de produire un système de réécriture convergent ou plus généralement un ensemble d'égalités *saturé* convergent. La complétion peut être alors considérée comme la compilation d'un ensemble et l'ensemble compilé est utilisé pour prouver la validité d'un ensemble d'égalités. Dans cette thèse, nous nous plaçons dans ce cas. Mais il existe également des procédures de complétion dont l'objectif est de prouver une égalité donnée. Il est fondamental que ces procédures de complétion soient *orientées par le but.* Par exemple, dans (Hsiang et Rusinowitch, 1986; Bachmair, 1987; Rusinowitch, 1987), la *complétion sans échec* est utilisée comme prouveur de théorèmes réfutationnel. Dans (Lynch, 1997), une procédure de *complétion basique orientée but* utilisant comme structure de données les graphes SOUR est présentée. Dans un graphe SOUR, chaque sommet est étiqueté par un symbole et représente un terme. Ces graphes rendent explicites les relations de sous-termes, d'orientation, d'unification et de réécriture, puisque ces relations sont schématisées par des arcs sur le graphe. Pour faire le lien entre ces deux classes, L. Bachmair et N. Dershowitz (Bachmair et Dershowitz, 1994) décrivent les procédures de complétion comme des processus de réduction et de transformation de preuves en des *preuves en vallée* appelées également *preuves par réécriture*, qui préservent la validité des égalités.

Nous allons présenter dans cette section différentes procédures de complétion et différents systèmes d'inférence de complétion. Les systèmes d'inférence, les notions et les notations introduits sont utilisés dans les chapitres 3, 4 et 5. Cette section est nécessaire pour motiver notre travail de thèse et pour pouvoir faire des comparaisons.

La première procédure de complétion dite *complétion de Knuth-Bendix* a été mise au point au début des années soixante-dix (Knuth et Bendix, 1970). Une procédure de Knuth-Bendix est donnée dans la section 2.3.1. L'idée principale de la complétion de Knuth-Bendix est de construire incrémentalement un système de réécriture convergent à partir d'un système de réécriture donné terminant. Elle s'arrête sur échec si l'une des égalités générées ne peut pas être orientée en règle de réécriture. Si elle s'arrête sans échec, elle garantit qu'on obtient un système de réécriture convergent. Mais elle peut également ne pas s'arrêter et générer un ensemble infini d'égalités, on dit alors qu'elle diverge. Ainsi, elle peut être considérée comme une procédure de semi-décision pour le problème du mot.

La *complétion sans échec* appelée également *complétion ordonnée* ou encore *complétion standard* est une extension de la complétion de Knuth-Bendix. Elle ne requiert pas qu'il soit toujours possible d'orienter une égalité en règle de réécriture mais elle construit un ensemble d'égalités convergent s'il existe. Elle est présentée dans la section 2.3.2.

La complétion de Knuth-Bendix nécessite de partir d'un système de réécriture terminant et de pouvoir orienter les égalités en règles de réécriture. Or, certaines égalités ne peuvent être orientées en règles de réécriture et entraînent donc l'échec de la complétion de Knuth-Bendix. C'est le cas de l'axiome de commutativité $f(x,y) \approx f(y,x)$. Ce problème a motivé l'*intégration* de certaines égalités. On parle alors de théorie *modulo* ou de théorie *built-in*). On parle alors de *complétion de Knuth-Bendix modulo une théorie équationnelle.* La partie déduction de la complétion (les règles d'inférence) est alors séparée de la partie calcul (calculs concernant l'unification, le filtrage et les ordres). Ces observations sont à l'origine des travaux de (Dowek et al., 1998). Nous parlerons dans cette section et dans tout ce document principalement des théories associative, commutative et associative-commutative, car en pratique, ces théories sont souvent nécessaires.

La complétion sans échec d'un ensemble d'égalités contenant les axiomes d'associativité et de commutativité termine rarement dans la pratique. L'axiome de commutativité peut en effet générer de nombreuses versions permutées de la même égalité. Pour pallier à ce problème, les axiomes A et C sont considérés comme intégrés dans le processus de déduction et on obtient la *complétion sans échec modulo AC* et plus généralement la *complétion sans échec modulo.* La complétion sans échec modulo est décrite dans la section 2.3.3.

L'introduction de contraintes permet de sauvegarder l'unificateur le plus général dans la contrainte au lieu de l'appliquer à la conclusion d'une inférence, comme c'est le cas dans la complétion de Knuth-Bendix et la complétion standard. La *complétion contrainte* s'appuie sur ces principes. On fera la différence entre complétion contrainte et *complétion basique.* En effet, en complétion basique, les contraintes sont vues comme des substitutions[5]. Le cas des complétions contrainte et basique est détaillé dans la section 2.3.4. Le principal problème des complétions contrainte et basique est l'utilisation de stratégies de contraction, les stratégies actuelles n'étant pas satisfaisantes, car les stratégies de contraction actuelles nécessitent de résoudre les contraintes et ne permettent pas assez de simplifications. Une implantation de la complétion basique utilisant les graphes SOUR (Lynch et Strogova, 1998) est décrite dans le chapitre 3. Cette implantation est utilisée pour décrire, dans ce même chapitre, notre approche concurrente de la complétion (close).

Les complétions contrainte et basique ont été également étendues aux *complétions contrainte et basique modulo.* L'utilisation des contraintes prend toute son importance dans ce cas. Nous évoquons la complétion contrainte et basique dans la section 2.3.5. Le principal inconvénient de la complétion modulo une théorie équationnelle AX est que son efficacité est inhérente à l'efficacité des calculs d'AX-filtrage et d'AX-unification. Dans le cas de la complétion modulo

5. Les contraintes sont supposées en forme résolue.

AC, pour le calcul de paire critique, une égalité est déduite pour chaque unificateur AC de l'ensemble complet des unificateurs du problème d'unification AC considéré. On notera que $x + x + x + x =^?_{AC} y_1 + y_2 + y_3 + y_4$, où $+$ est un symbole AC, x, y_1, y_2, y_3 et y_4 sont des variables, a 34 359 607 481 unificateurs les plus généraux minimaux (Domenjoud, 1992). La résolution de cette contrainte impliquerait donc la création de 34 359 607 481 égalités. Pour remédier à cet état de fait, les AX-unificateurs ne sont plus appliqués mais sauvés dans une contrainte et les inférences sont interdites dans les contraintes.

Nous avons implanté une procédure de complétion standard, une procédure de complétion basique avec simplification standard et une procédure de complétion basique avec simplification basique (avec et sans rétraction) en ELAN dans le système *ECC*. Nous utiliserons donc ici, ELAN comme formalisme pour décrire les différentes procédures de complétion implantées.

2.3.1 Complétion de Knuth-Bendix

L'idée principale de la complétion de Knuth-Bendix (Knuth et Bendix, 1970) est de construire incrémentalement un système de réécriture convergent à partir d'un système de réécriture donné terminant et un ordre de réduction $\succ_t$ sur les termes. La complétion de Knuth-Bendix s'appuie sur la propriété de confluence locale pour un système de réécriture, car la confluence locale d'un système de réécriture terminant implique la convergence. La confluence locale est testée sur des paires de termes issues de paires de règles par superposition. S'il n'y a pas confluence locale, les paires de termes appelées *paires critiques* sont ajoutées sous forme d'égalité. Le processus est récursif et continue tant qu'il y a des paires critiques à calculer. Le processus peut s'arrêter sur échec, si une égalité déduite ne peut pas être orientée en règle de réécriture. S'il s'arrête sans échec, on obtient un système de réécriture convergent. Le processus peut également ne pas terminer et on dit qu'il diverge.

La définition d'une paire critique est donnée ici.

Définition 8 *Soient $l \to r$ et $g \to d$ deux règles de réécriture d'ensembles disjoints de variables telles qu'il existe une substitution σ telle que $\sigma = mgu(l =^? g_{|\omega})$ et $g_{|\omega}$ ne soit pas une variable. Alors $(\sigma(g[r]_\omega),\sigma(d))$ est une* paire critique. □

Exemple 9 *Si l'on considère le système de réécriture $R = \{g(f(a)) \to b\ (1),\ a \to c\ (2)\}$ orienté en utilisant LPO avec la précédence sur les symboles de fonction $g \succ_p f \succ_p a \succ_p b \succ_p c$, alors en superposant l'égalité (2) dans l'égalité (1), on obtient la paire critique $(g(f(c)),b)$ qui peut être orientée en la règle de réécriture $g(f(c)) \to b$.*

2.3.1.1 Une procédure de complétion de Knuth-Bendix

Les transitions décrivant la complétion de Knuth-Bendix sont données sur la figure 2.1. On y distingue les égalités des règles de réécriture. Ainsi, dans cette section, une transition a la forme $(P_i,R_i) \to (P_{i+1},R_{i+1})$ où P_i et P_{i+1} sont des ensembles d'égalités et R_i et R_{i+1} sont des systèmes de réécriture. P_∞ est défini par $\bigcup_{i\geq 0}\bigcap_{j\geq i} P_j$ et R_∞ est défini par $\bigcup_{i\geq 0}\bigcap_{j\geq i} R_j$. Une dérivation (P_0,R_0), $(P_1,R_1),\ldots$ échoue si une égalité ne peut être orientée en règle de réécriture.

- *Deduce* calcule les paires critiques entre des règles.
- *Orient* oriente les égalités en règles de réécriture.
- *Delete* supprime les égalités triviales $s \approx s$.
- *Compose* simplifie les membres droits des règles de réécriture en utilisant des règles de réécriture.

Deduce : $(P,R) \rightarrow (P \cup u \approx v,R)$ si (u,v) est une paire critique de R

Orient : $(P \cup s \approx t,R) \rightarrow (P; R \cup s \rightarrow t)$ si $s \succ_t t$

Delete : $(P \cup s \approx s; R) \rightarrow (P,R)$

Compose : $(P; R \cup s \rightarrow t) \rightarrow (P; R \cup s \rightarrow u)$ si $t \rightarrow_R u$

Simplifyright : $(P \cup s \approx t; R) \rightarrow (P \cup s \approx u; R)$ si $t \rightarrow_R u$

Simplifyleft : $(P \cup s \approx t; R) \rightarrow (P \cup u \approx t; R)$ si $s \rightarrow_R u$

Collapse : $(P; R \cup s \rightarrow t) \rightarrow (P \cup v \approx t; R)$ si $s \rightarrow_R v$ en utilisant la règle $l \rightarrow r$ et $s \approx t \succ_e l \approx r$

FIG. 2.1 – *Transitions pour la complétion de Knuth-Bendix*

- *Simplifyright* (respectivement *Simplifyleft*) utilise les règles pour simplifier les membres droits (respectivement gauches) d'une égalité.
- *Collapse* simplifie le membre droit d'une règle avec une règle de réécriture et transforme la forme simplifiée en égalité. Elle est appliquée à condition que la règle qui simplifie soit plus petite que la règle à simplifier.

Une autre manière de voir la règle *Deduce* est de l'exprimer sous forme de *règle d'inférence.* Nous perdons l'information concernant l'espace de recherche mais cette notation nous permet mieux de voir quelles égalités sont impliquées dans l'inférence.

$$\frac{g[l']_\omega \rightarrow d \qquad l \rightarrow r}{\sigma(g[r]_\omega \approx d)} \text{ où :}$$

- $\sigma = mgu(l,l')$.

Correction et complétude Les règles de transformation *Deduce*, *Orient*, *Delete*, *Compose*, *Simplifyright*, *Simplifyleft* et *Collapse* sont *correctes*, car elles ne changent pas l'ensemble des égalités valides. Elles sont également complètes car toute dérivation qui n'échoue pas construit un système de réécriture convergent R_∞.

Nous décrivons maintenant une procédure de complétion de Knuth-Bendix basée sur les transitions décrites précédemment et sur la stratégie suivante. L'ensemble des règles de réécriture R est divisé en deux. Il y a les règles « marquées » et les règles « non marquées ». Une règle est marquée si toutes ses paires critiques avec les autres règles de réécriture ont été calculées et ajoutées à l'ensemble des égalités P. Lorsque l'on sélectionne une règle de réécriture dans l'ensemble des règles de réécriture R, celle-ci ne doit pas être marquée. Cette méthode assure que le choix des règles de réécriture est réalisé de manière équitable. Les égalités de l'ensemble des égalités P doivent également être sélectionnée de manière équitable. De plus, l'application des règles *Compose*, *Delete*, *Simplifyleft*, *Simplifyright* et *Collapse* est prioritaire par rapport à la règle *Deduce* qui augmente la taille de l'espace de recherche.

La procédure de complétion de Knuth-Bendix considérée est décrite par le programme suivant écrit en pseudo-pascal. La fonction *SIMPLIFICATION* utilise les transitions *Compose*, *Collapse*, *Simplifyright* et *Simplifyleft*. La fonction *CRITICAL-PAIRS* utilise la règle *Deduce*.

```
Procedure KB-COMPLETION(P,R,≻t)
Si P n'est pas vide
Alors sélectionner une égalité dans P
      P  := P - {p ≈ q}
      p' := p ↓R
      q' := q ↓R
      Cas p' = q'  Alors R := KB-COMPLETION(P,R,≻t)
      Cas p' ≻t q' Alors l := p'
                             r := q'
                             (P,R) := SIMPLIFICATION(P,R,l → r)
                             R := KB-COMPLETION(P,R ∪ {l → r,≻t)
      Cas q' ≻t p' Alors l := q'
                             r := p'
                             (P,R) := SIMPLIFICATION(P,R,l → r)
                             R := KB-COMPLETION(P,R ∪ {l → r,≻t)
      Sinon échec
      fin cas
Sinon Si toutes les règles de R sont marquées
      alors retourner R
            succès
      sinon sélectionner une règle non marquée l → r équitablement
            P := CRITICAL-PAIRS(l → r, R)
            marquer la règle l → r dans R
            R := KB-COMPLETION(P,R,≻t)
      fin si
fin si
fin KB-COMPLETION
```

2.3.1.2 Preuves par réécriture

L. Bachmair et N. Dershowitz (Bachmair et Dershowitz, 1994) ont montré qu'une procédure de complétion de Knuth-Bendix peut être vue comme un processus de réduction et de transformation de preuves contenant des pics en des preuves en « vallée » appelée preuves par réécriture.

Définition 9 *Un* pic *est caractérisé par le motif* $t' \leftarrow s \rightarrow t''$ *dans une preuve.* □

Exemple 10 *Considérons le système de réécriture* $R = \{a \rightarrow b,\ a \rightarrow c,\ b \rightarrow d,\ c \rightarrow e\}$ *et la précédence* $a \succ_p b \succ_p c \succ_p d \succ_p e$.

Supposons que l'on veut prouver que $d \approx e$ *est vraie dans* R.

Une preuve équationnelle de $d \approx e$ *est* $d \leftrightarrow_E b \leftrightarrow_E a \leftrightarrow_E c \leftrightarrow_E e$. *Nous allons montrer comment cette preuve est transformée en preuve par réécriture en utilisant la complétion de Knuth-Bendix.*

Il y a un pic entre entre les égalités $a \approx b$ *et* $a \approx c$ *qui représente la paire critique* (b,c). *La paire critique* (b,c) *peut être transformée en la règle de réécriture* $b \rightarrow c$ *car* $b \succ_p c$. *On efface ce pic et on obtient la preuve équationnelle* $d \leftrightarrow_E b \leftrightarrow_E c \leftrightarrow_E e$.

Il y a un pic entre $b \approx c$ et $b \approx d$. La paire critique (d,c) est calculée et orientée en la règle de réécriture $c \rightarrow d$ car $c \succ_p d$. On obtient alors la nouvelle preuve $d \leftrightarrow_E c \leftrightarrow_E e$.

Il y a un pic entre $c \approx d$ et $c \approx e$. La paire critique (d,e) est orientée en $d \rightarrow e$.

Nous avons ainsi construit une preuve de $d \approx e$ en vallée à partir d'une preuve équationnelle.

Les transformations de la preuve sont visibles sur la figure 2.2.

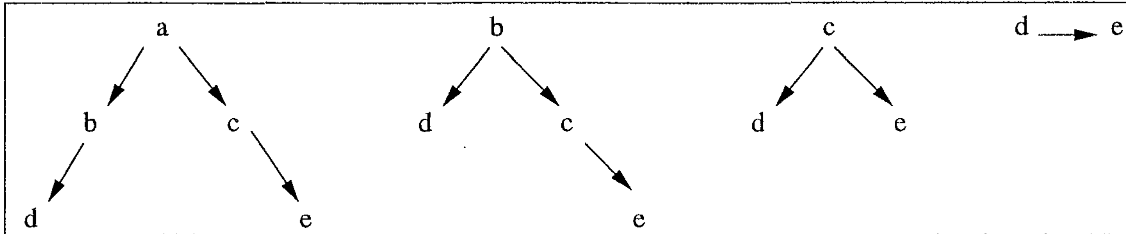


FIG. 2.2 – *La complétion : Pour transformer une preuve en preuve en vallée*

2.3.1.3 La complétion close

Dans le cas général, la complétion de Knuth-Bendix peut être exponentiellement longue (par rapport à la taille des termes de l'ensemble à compléter) et diverger. Toutefois, elle est polynomiale en temps dans le cas clos.

D. Kozen développa une méthode basée sur la clôture de congruence pour résoudre le problème du mot (Kozen, 1977). J. Gallier, P. Narendran, D. Plaisted et S. Raatz (Gallier, Narendran, Plaisted, Raatz et Snyder, 1993) ont étendu les techniques de D. Kozen pour générer un système convergent équivalent à un ensemble clos d'égalités . W. Snyder améliora le temps d'exécution de cet algorithme en $nlog(n)$ où n est le nombre d'occurrences de symboles de l'ensemble clos (Snyder, 1989). D. Plaisted et A. Sattler-Klein ont montré que, avec une stratégie particulière et avec du partage de structure, la procédure de complétion de Knuth-Bendix peut tourner en un temps polynomial pour un ensemble clos d'égalités (Plaisted et Sattler-Klein, 1996). C. Lynch a montré qu'une forme particulière de paramodulation qui n'a pas besoin de copier les termes tourne en un temps polynomial dans les cas clos (Lynch, 1995). Ce résultat a été appliqué à la complétion qui est un cas particulier de paramodulation. C. Lynch a appelé cette méthode de paramodulation, la paramodulation sans duplication (*paramodulation without duplication*). Dans le cas de la complétion, il a développé une méthode appelée la complétion des graphes SOUR (Lynch et Strogova, 1998). La complétion basique des graphes SOUR est décrite dans la section 3.2.

2.3.1.4 Limitations de la complétion de Knuth-Bendix

La complétion de Knuth-Bendix et la complétion en général peuvent générer un ensemble infini d'égalités ou de règles et ne pas terminer. La terminaison de la complétion est un problème indécidable. Les travaux (Lescanne, 1986; Hermann, 1989) proposent une condition suffisante pour prédire la divergence. L'un des éléments essentiel dont dépend la terminaison est la précédence fixée au départ et par là, l'ordre de réduction utilisé. Si l'on considère l'égalité $f(f(x)) \approx g(f(x))$ et si l'on prend la précédence $f \succ_p g$, le système de réécriture à compléter est $R = \{f(f(x)) \rightarrow g(f(x))\}$. La complétion de Knuth-Bendix diverge dans ce cas, car il y a un nombre infini de paires critiques à calculer. Par contre, si on prend comme précédence $g \succ_p f$, alors le système de réécriture à compléter est $R = \{g(f(x)) \rightarrow f(f(x))\}$. Le système de réécriture

saturé R_∞ est R. En effet, il n'y a aucune paire critique à calculer. Un autre exemple est donné dans (Kapur et Narendran, 1985). D. Kapur et P. Narendran ont montré que quelle que soit la précédence, la complétion de $R = \{f(g(f(x))) \rightarrow g(f(g(x)))\}$ ne termine pas.

La complétion de Knuth-Bendix est limitée par le fait qu'elle échoue si une égalité ne peut pas être orientée en règle de réécriture. Une extension de la complétion de Knuth-Bendix pour pallier au problème d'orientation des égalités en règle de réécriture évoqué ci-dessus est présentée dans la section 2.3.2. Il s'agit de la *complétion sans échec.*

2.3.2 Complétion sans échec

La complétion sans échec est présentée dans (Hsiang et Rusinowitch, 1987; Bachmair, Dershowitz et Plaisted, 1989). L. Bachmair a montré que la complétion de Knuth-Bendix est un sous-ensemble de la complétion sans échec dans le sens où toute dérivation de complétion de Knuth-Bendix a une dérivation de complétion sans échec correspondante (Bachmair, 1987). La complétion sans échec ne s'arrête pas avec échec sur les égalités qui ne peuvent pas être orientées en règles de réécriture, elle les laisse sous forme d'égalité. Elle émet toutefois des restrictions d'ordre sur l'emploi des égalités dans les règles d'inférence comme nous allons le voir.

La complétion sans échec est décrite par le système d'inférence *UC* (*Unfailling Completion*) composé des règles d'inférence *Paire Critique* et *Simplification Standard 1.* La règle *Paire Critique* est une règle d'expansion qui calcule une paire critique. La règle *Simplification Standard 1* est une règle de simplification. Elles sont définies comme suit.

Paire Critique

$$\frac{g[l']_\omega \approx d \qquad l \approx r}{\sigma(\mathbf{g}[\mathbf{r}]_\omega \approx \mathbf{d})} \quad \text{si :}$$

- l' n'est pas une variable,
- $\sigma = mgu(l,l')$,
- $\sigma(g[l']_\omega) \not\preceq_t \sigma(d)$, $\sigma(l) \not\preceq_t \sigma(r)$, et si $\omega = \epsilon$, alors $(\sigma(d) \succ_t \sigma(r)$ ou $\sigma(r) \not\succ_t \sigma(d))$.

Remarque 1 *La dernière condition d'application de la règle* Paire Critique *peut s'écrire* $\sigma(g[l']_\omega) \not\preceq_t \sigma(d) \wedge \sigma(l) \not\preceq_t \sigma(r) \wedge ((\omega = \epsilon \wedge (\sigma(d) \succ_t \sigma(r) \vee \sigma(r) \not\succ_t \sigma(d))) \vee \omega \neq \epsilon)$. *Cette dernière formulation est choisie plus loin pour transcrire cette règle d'inférence en* ELAN[6].

Dans cette règle d'inférence, l'égalité $g[l']_\omega \approx d$ est appelée égalité « gauche », l'égalité $l \approx r$ est appelée l'égalité « droite » et l'égalité $\sigma(g[r]_\omega \approx d)$ est appelée l'égalité « conclusion ». Ces appellations ont été choisies pour montrer dans quel terme a lieu la superposition.

La transition correspondant à cette dernière règle d'inférence est la suivante. Γ est un ensemble d'égalités.

$$\{g[l']_\omega \approx d,\, l \approx r\} \cup \Gamma \rightarrow \{g[l']_\omega \approx d,\, l \approx r,\, \sigma(g[r]_\omega \approx d)\} \cup \Gamma$$

6. Cette dernière remarque est valable pour tout le document.

Simplification Standard 1

$$\frac{g[l']_\omega \approx d \qquad l \approx r}{g[\mu(r)]_\omega \approx d} \quad \text{si :}$$

- l' n'est pas une variable,
- Il existe un filtre μ tel que, $l' = \mu(l)$, $\mu(l) \succ_t \mu(r)$ et si $\omega = \epsilon$, alors (μ n'est pas une substitution de renommage ou $d \succ_t \mu(r)$).

La transition correspondant à cette dernière règle d'inférence est la suivante. Γ est un ensemble d'égalités.

$$\{g[l']_\omega \approx d,\, l \approx r\} \cup \Gamma \to \{l \approx r,\, g[\mu(r)]_\omega \approx d\} \cup \Gamma$$

La règle *Simplification Standard 1* simplifie le membre gauche de l'égalité « gauche ». On peut également introduire une règle d'inférence qui simplifie le membre droit de l'égalité « gauche ».

L'intérêt de la règle de simplification transparaît dans l'exemple suivant.

Exemple 11 *Soient un ensemble d'égalités $E = \{f(a) \approx a,\, a \approx b\}$ et la précédence $a \succ_p f \succ_p b$. L'ordre de réduction utilisé est LPO.*

Si nous n'utilisons pas la règle de simplification, le nombre d'inférences de paires critiques à réaliser est infini. Les inférences ont lieu entre l'égalité $f(a) \approx a$ comme égalité « gauche » et l'égalité $a \approx f^n(b)$ comme égalité « droite » pour $n \geq 0$ et déduisent $a \approx f^{n+1}(b)$.

Par contre, si nous utilisons la règle de simplification, nous pouvons d'abord simplifier $f(a) \approx a$ par $a \approx b$ et nous déduisons $a \approx f(b)$. $f(a) \approx a$ est supprimée de l'espace de recherche. Puis, nous simplifions $a \approx f(b)$ par $a \approx b$ et nous déduisons $f(b) \approx b$. $f(a) \approx f(b)$ est supprimée de l'espace de recherche. Nous obtenons ainsi $E_\infty = \{a \approx b,\, f(b) \approx b\}$.

Nous avons implanté une procédure de complétion sans échec dans le système *ECC*. Cette procédure est basée sur les deux règles d'inférence précédemment décrites et sur la stratégie suivante. L'ensemble des égalités est divisé en deux parties, les égalités « traitées » et les égalités « non traitées ». Les égalités « traitées » sont dans une queue appelée *PQ* (*Processed Queue*) et les égalités non traitées dans une queue appelée *UQ* (*Unprocessed Queue*). Une égalité est dans *PQ* si toutes les paires critiques entre elle et les égalités de *PQ* ont été calculées et ajoutées à *UQ*. Les queues *PQ* et *UQ* sont triées par ordre croissant suivant la taille des égalités les composants. Les égalités sélectionnées dans *PQ* ou dans *UQ* sont les égalités en tête de ces queues. Cette stratégie de recherche est équitable. La stratégie de recherche décrite est une stratégie de contraction, appelée *stratégie de simplification standard*, car la règle de contraction (*Simplification Standard 1*) est prioritaire par rapport à la règle d'expansion (*Paire Critique*). Toute égalité *e* sélectionnée dans *UQ* de manière équitable comme décrit précédemment est d'abord simplifiée en *esimp* par les égalités de *PQ* (si possible), puis elle est utilisée pour simplifier les égalités de *PQ* (si possible). Les paires critiques entre *esimp* et les égalités de *PQ* sont alors calculées et ensuite, *esimp* est ajoutée à *PQ*.

Nous décrivons d'abord l'implantation des égalités en ELAN puis nous donnons le code ELAN correspondant à la procédure de complétion décrite. Nous avons omis tous les tests qui ne sont pas significatifs ici.

Les égalités sont implantées en ELAN par la sorte *eqconstrained*. Une égalité est déterminée par un squelette, c'est-à-dire une égalité de sorte *equality*, une contrainte de sorte *constraint* et un entier de sorte *int*, qui permet de compter les égalités de l'espace de recherche. Ce compteur

détermine le nombre d'égalités déduites pendant la complétion et nous est utile pour comparer les différentes formes de complétion. Dans cette section, nous n'utilisons pas de contraintes, mais cette représentation a été choisie pour les complétions présentées dans les sections suivantes.

Les sortes *constraint* et *eqconstrained* sont décrites ici.

```
sort
        constraint;
end

operators
 global
        /* Constructeurs d'une contrainte */
        @       : (bool) constraint;
        @ =? @  : (term term) constraint;
        @ & @   : (constraint constraint) constraint    (AC);
        (@ & @) : (constraint constraint) constraint    alias @ & @:;
end
```

```
sort
  eqconstrained;
end

operators
global
  @[@]<@> : (equality constraint int) eqconstrained;
end
```

Ainsi, par exemple, $g(x) = b[x =^? a] < 3 >$ est un élément de sorte *eqconstrained.*

La stratégie de la procédure de complétion ordonnée implantée dans *ECC* est donnée par le code ELAN suivant. La fonction *Critical_Pair* fait appel à la règle d'inférence *Paire Critique* et les fonctions *Standard_Simplification_eq_listeq* et *Standard_Simplification_listeq_eq* font appel à la règle d'inférence *Simplification Standard 1.*

```
rules for list[eqconstrained]
 counter, newcounter1, newcounter2      : int;
 e, esimp                   : eqconstrained;
 PQ, UQ, UQ0, PQ0, UQ1, UQ2s, PQ2s, res : list[eqconstrained];
 pUQPQ                      :  tuple[list[eqconstrained],list[eqconstrained]];
global

[] completion(nil, PQ, counter)         => PQ
end

[] completion(e.UQ, PQ, counter)        =>
res
   where esimp        := ()Standard_Simplification_eq_listeq(e,PQ,PQ,counter)
   where pUQPQ        := ()Standard_Simplification_listeq_eq(UQ,PQ,esimp,counter)
   where UQ0          := ()1-th (pUQPQ)
   where PQ0          := ()2-th(pUQPQ)
   /* Mise à jour du compteur d'égalités déduites */
   where newcounter :=()maxcounter(UQ0,counter)
   where UQ1          := ()Critical_Pair(esimp,esimp.PQ0,newcounter)
   where newcounter1:= ()maxcounter(UQ1,newcounter)
   /* Tri des queues UQ et PQ en fonction de la taille des égalités */
   /* qui les composent */
   where UQ2s         := ()sortinter(append(UQ0,UQ1))
   where PQ2s         := ()sortinter(append(PQ0,esimp.nil))
   where res          := ()completion(UQ2s,PQ2s,newcounter1)
end
end
```

Nous présentons maintenant en vis à vis les règles d'inférence *Paire Critique* et *Simplification Standard 1* et leurs implantations en ELAN. Nous avons utilisé divers moyens typographiques pour faire le lien entre les deux représentations. Noter que la remarque 1 est également utilisée pour permettre de faire ce lien.

Paire Critique

$$\frac{g[l']_\omega \approx d \qquad l \approx r}{\boxed{\sigma(g[r]_\omega \approx d)}}$$

si :

- l' n'est pas une variable,
- $\sigma = mgu(l,l')$
- $\sigma(g[l']_\omega) \not\preceq_t \sigma(d)$,
- $\sigma(l) \not\preceq_t \sigma(r)$, et
- si $\omega = \epsilon$,

alors $(\sigma(d) \succ_t \sigma(r)$

ou $\sigma(r) \not\succ_t \sigma(d))$.

```
[BCP] Critical_Pair
(g=d[c1]<counter1>,
l=r[c2]<counter2>, newcounter)
=>

[CPNewEq]

/*Choix d'une position non variable dans g*/
where omega := (chooseOccurence)nvocc(g)
where unif := (unifys)c1 & c2 &
 g at omega =? l
where sigma := ()constraint_to_subst(unif)
where sigmad := ()sigma(d)
where sigmag := ()sigma(g)
where sigmal := ()sigma(l)
where sigmar := ()sigma(r)
if not sigmad >lpo sigmag
if not sigmar >lpo sigmal
where posempty := ()eq_list[int](omega,nil)
choose
try
if posempty
 where cond1 := ()sigmad >lpo sigmar
 choose
 try
 if cond1
  where
  [CPNewEq := ()sigmag[sigmar] at omega=sigmad[true]<newcounter>]
 try
 if not cond1
 if not sigmar >lpo sigmad
  where
  [CPNewEq := ()sigmag[sigmar] at omega=sigmad[true]<newcounter>]
 end
try
if not posempty
 where
 [CPNewEq := ()sigmag[sigmar] at omega=sigmad[true]<newcounter>]
end
end
```

Simplification Standard 1	[SSimpLeft] **Standard_Simplification** (g=d[c1]<counter1>, l=r[c2]<counter2>, newcounter) =>
$$\frac{g[l']_\omega \approx d \qquad l \approx r}{\boxed{g[\mu(r)]_\omega \approx d}}$$	**SSimpNewEq**
si : – l' n'est pas une variable, – Il existe un filtre μ tel que : – $l' = g_{\vert\omega} = \mu(l)$,	where *omega := (chooseOccurence)nvocc(g)* where *matcha := (matchs)l =? g at omega* where mu := ()constraint_to_subst(matcha) where mul := ()mu(l) where mur := ()mu(r)
– $\mu(l) \succ_t \mu(r)$ et	*if mul >lpo mur* where posempty := ()eq_list[int](omega,nil) choose try
– Si $\omega = \epsilon$,	*if posempty* where *cond1 := ()is_renaming(mu)* where *cond2 := ()d >* lpo mur choose try
alors (μ n'est pas une substitution de renommage	*if not cond1* where **SSimpNewEq := ()g[mur] at omega= d[true]<newcounter>**
ou $d \succ_t \mu(r)$).	try if cond1 *if cond2* where **SSimpNewEq := ()g[mur] at omega= d[true]<newcounter>** try if not posempty where **SSimpNewEq := ()g[mur] at omega= d[true]<newcounter>** end end

Correction Le système d'inférence *UC* composé des règles *Paire Critique* et *Simplification Standard 1* est correct. On dit aussi que la complétion sans échec avec simplification standard est correcte.

Complétude Le système d'inférence *UC* est également complet (Bachmair et al., 1989). On dit aussi que la complétion sans échec avec simplification standard est complète.

2.3.3 Complétion modulo une théorie équationnelle

L'utilisation d'égalités intégrées a été motivée dans la complétion de Knuth-Bendix, car la complétion de Knuth-Bendix nécessite de partir d'un système de réécriture terminant et de pouvoir orienter les égalités en règles de réécriture. L'approche de G. Huet concernant la complétion de Knuth-Bendix d'un système de réécriture modulo un ensemble d'axiomes AX ne s'applique qu'à des systèmes de réécriture linéaires-gauches (Huet, 1980). Dans cette méthode, il s'agit de ne pas impliquer deux égalités de l'ensemble des égalités intégrées pour réaliser une inférence. L'approche de G. Peterson et M. Stickel (Peterson et Stickel, 1981) ne s'applique quant à elle que lorsqu'il existe un algorithme d'unification complet fini pour E et que E est linéaire. Elle nécessite l'utilisation d'*extensions* d'égalités pour pallier la perte d'expressivité due à l'utilisation des axiomes modulo. Les extensions d'égalités sont définies dans la suite. Ces principes ont été rassemblés et étendus dans (Kirchner, 1985; Jouannaud et Kirchner, 1986).

Nous nous intéressons ici plus particulièrement à la complétion sans échec modulo. La complétion sans échec termine rarement en présence des axiomes d'associativité et de commutativité. Une idée naturelle a donc été donc de définir la complétion sans échec modulo. Les principaux travaux concernant la complétion sans échec modulo sont présentés dans (Anantharaman, Hsiang et Mzali, 1989; Anantharaman et Mzali, 1989).

Dans cette section, on suppose que AX est un ensemble d'axiomes tel qu'il existe des algorithmes d'AX-unification et d'AX-filtrage. On supposera également que $\succ_t$ est un ordre de réduction total sur les termes clos AX-compatible. E est un ensemble d'égalités.

En complétion modulo, il est nécessaire d'utiliser des extensions pour les égalités de E par rapport à AX. L'exemple 12 souligne l'utilité des extensions.

Exemple 12 *Soient $E = \{f(a,b) \approx c,\ f(b,d) \approx e\}$, $AX = AC$ et la précédence $f \succ_p a \succ_p b \succ_p c \succ_p d \succ_p e$. Supposons que f est un symbole AC.*

En utilisant seulement les égalités de E, il n'est pas possible d'obtenir une preuve équationnelle de $f(c,d) \approx f(a,e)$.

Or, si on étend l'égalité $f(a,b) \approx c$ en $f(f(a,b),x) \approx f(c,x)$ en se basant sur le fait que f est un symbole AC, on obtient une preuve équationnelle : $f(c,d) \overset{*}{\leftrightarrow}_{E \cup AC} f(f(a,b),d) \overset{*}{\leftrightarrow}_{E \cup AC} f(a,f(b,d)) \overset{*}{\leftrightarrow}_{E \cup AC} f(a,e)$.

De la même manière, pour pouvoir prouver que $f(c,d) \approx f(a,e)$ par une preuve par réécriture, il est nécessaire d'utiliser des extensions des égalités de E.

Les extensions sont définies ici.

Définition 10 *Soient $g \approx d$ un axiome de AX et $l \approx r$ une égalité de E ayant des ensembles de variables disjoints.*

S'il existe une position ω dans g telle que $g_{|\omega}$ ne soit pas une variable, $g_{|\omega}$ et l sont AX-unifiables et $\sigma \in CSU(g_{|\omega} =^?_{AX} l)$ et $\sigma(r) \not\succ_t \sigma(l)$, alors $g[l]_\omega \approx g[r]_\omega$ est une extension *de l'égalité $l \approx r$ par rapport à $g \approx d$.* □

Dans le cas général, il est nécessaire de calculer les extensions des extensions et ainsi de suite, récursivement. Toutefois, dans les cas A et AC, seules deux étapes et une étape d'extension sont

nécessaires respectivement (Rubio, 1994). On peut remarquer que dans le cas C les extensions ne sont pas nécessaires.

Soient $s \approx t \in E$, x et y des variables n'apparaissant pas dans s et t.

- Si $AX = A$, les extensions nécessaires de $s \approx t$ sont :
  - $f(s,x) \approx f(t,x)$
  - $f(x,s) \approx f(x,s)$
  - $f(x,f(s,y)) \approx f(x,f(t,y))$
- Si $AX = AC$, l'extension nécessaire de $s \approx t$ est :
  - $f(s,x) \approx f(t,x)$

Les règles d'inférence de la complétion sans échec modulo sont la règle de calcul de paire critique, *Paire Critique AX* qui est une règle d'expansion et la règle de simplification, *Simplification Standard AX*, qui est une règle de contraction. Nous appelons SC_m le système d'inférence composé de ces deux règles.

Paire Critique AX

$$\frac{g \approx d \qquad l \approx r}{\sigma(\mathbf{g''[r'']_\omega \approx d''})} \quad \text{si :}$$

- $g''[l']_\omega \approx d''$ est l'égalité $g \approx d$ ou, une extension de $g \approx d$ et $\omega = \epsilon$,
- $l'' \approx r''$ est l'égalité $l \approx r$ ou une extension de $l \approx r$,
- l' n'est pas une variable,
- $\sigma \in CSU(l'' =^?_{AX} l')$,
- $\sigma(g''[l']_\omega) \not\preceq_t \sigma(d'')$, $\sigma(l'') \not\preceq_t \sigma(r'')$, et si $\omega = \epsilon$, alors $(\sigma(d'') \succ_t \sigma(r'')$ ou $\sigma(r'') \not\succ_t \sigma(d''))$.

Cette règle d'inférence s'exprime par la transition suivante. Γ est un ensemble d'égalités.

$$\{g[l']_\omega \approx d,\ l \approx r\} \cup \Gamma \rightarrow \{g[l']_\omega \approx d,\ l \approx r,\ \sigma(g''[r'']_\omega \approx d'')\} \cup \Gamma$$

Pour capturer l'utilisation des extensions, la règle *Paire Critique AX* peut également s'écrire[7] :

$$\frac{g'' \approx d'' \qquad l'' \approx r''}{\sigma(\mathbf{g''[r'']_\omega \approx d''})}$$

Exemple 13 *Dans l'exemple 12, on peut appliquer la règle* Paire Critique AC *entre l'égalité* $f(f(a,b),x) \approx f(c,x)$, *extension de* $f(a,b) \approx c$ *et l'égalité* $f(y,f(b,d)) \approx f(y,e)$, *extension de* $f(b,d) \approx e$.

$$\frac{f(f(a,b),x) \approx f(c,x) \qquad f(y,f(b,d)) \approx f(y,e)}{f(c,d) \approx f(a,e)}$$

Grâce à cette inférence, nous obtenons une preuve par réécriture de l'égalité $f(c,d) \approx f(a,e)$.

Simplification Standard AX

$$\frac{g[l']_\omega \approx d \qquad l \approx r}{g[\mu(r'')]_\omega \approx d} \quad \text{si :}$$

- $l'' \approx r''$ est l'égalité $l \approx r$ ou une extension de $l \approx r$,
- l' n'est pas une variable,
- Il existe un filtre μ tel que, $l' \overset{*}{\leftrightarrow}_{AX} \mu(l'')$, $\mu(l'') \succ_t \mu(r'')$ et si $\omega = \epsilon$, alors (μ n'est pas une substitution de renommage ou $d \succ_t \mu(r'')$).

7. Cette remarque est valable pour toutes les règles utilisant des extensions.

Cette règle d'inférence s'exprime par la transition suivante. Γ est un ensemble d'égalités.

$$\{g[l']_\omega \approx d,\, l \approx r\} \cup \Gamma \to \{l \approx r,\, g[\mu(r'')]_\omega \approx d\} \cup \Gamma$$

Correction et complétude SC_m est correct et complet. Le cas AC est prouvé dans (Bachmair et Ganzinger, 1993).

SBREVE (Anantharaman et al., 1989) et RRL (Kapur, Sivakumar et Zhang, 1986) sont deux systèmes basés sur la complétion modulo AC vue comme une méthode par réfutation. SBREVE a permis de prouver les identités de Moufang sur les anneaux alternatifs (Anantharaman et Hsiang, 1990) et RRL a permis de prouver la commutativité des anneaux associatifs (Kapur et Zhang, 1991).

Le succès de SBREVE est dû à l'utilisation de règles de *cancellations*. RRL quant à lui a dû utiliser des *sélections de paire critique* très raffinées pour pouvoir obtenir une preuve.

La règle d'inférence *Paire Critique AX* est une règle très prolifique. Elle nécessite l'utilisation de *critères de paire critique*, c'est-à-dire de critères qui nous permettent d'éviter de générer certaines paires critiques. Mais, elle justifie surtout l'utilisation des contraintes comme nous allons le voir dans les sections suivantes.

Parmi les critères de paire critique, on peut citer le *critère de paire critique bloquée* (Kapur et Zhang, 1991) qui évite de déduire une paire critique si la substitution utilisée dans l'inférence est réductible. Cette méthode nécessite toutefois de calculer toute les substitutions du problème d'AX-unification. D'autres critères essaient d'éviter d'avoir à calculer toutes les substitutions en détectant l'inutilité des paires critiques avant ce calcul. Considérons l'exemple suivant pour illustrer le critère de paire critique bloquée.

Exemple 14 *Considérons l'ensemble* $E = \{x+x+x+x+x+x \approx o,\ (x'+y')*z' \approx (x'*z') + (x'*z')\}$, *où* $+$ *est un symbole associatif-commutatif.*

Il existe une inférence de paire critique entre les deux égalités de E qui déduit 125 égalités, une pour chaque unificateur de $CSU(x'+y' =^?_{AC} x+x+x+x+x+x)$.

Toutefois, avec le critère de paire critique bloquée, 94 des 125 égalités ne sont pas déduites. En effet, considérons l'unificateur $\{x' \mapsto v+v+v+v+v+v,\ y' \mapsto u+u+u+u+u,\ x \mapsto u+v\}$ *du problème d'unification AC considéré. Cette substitution est réductible car* $\sigma(x')$ *l'est par la première règle de E. La paire critique construite à partir de cette substitution ne sera donc pas déduite.*

La recherche menée actuellement sur la complétion concerne essentiellement la complétion avec contrainte vu le caractère très prolifique de la règle de calcul de paire critique.

2.3.4 Complétions contrainte et basique dans la théorie vide

C. Kirchner, H. Kirchner et M. Rusinowitch sont à l'origine des travaux sur la déduction avec contrainte (Kirchner et al., 1990). L'idée de la complétion avec contrainte est de ne pas appliquer les unificateurs mais de sauver le problème d'unification sans le résoudre sous forme de contrainte. On teste seulement la satisfaisabilité de la contrainte d'unification. Cette restriction prend tout son sens dans le cas modulo (voir la section 2.3.5). Une autre caractéristique de la complétion avec contraintes est d'interdire les inférences dans les contraintes. La complétion avec contraintes permet par cette restriction de limiter le nombre d'égalités à déduire et ainsi de contrôler la taille de l'espace de recherche. Une autre manière de limiter la taille de l'espace de recherche est d'utiliser des règles de contraction. Les travaux sur la contraction dans la complétion contrainte n'ont abouti qu'à la mise au point de méthodes où il est nécessaire de résoudre les

contraintes ou de voir les contraintes comme des substitutions. Dans ce cas, nous parlons plutôt de complétion basique au lieu de complétion contrainte.

Dans la littérature, d'autres types de contraintes que des contraintes équationnelles ont été introduites en démonstration automatique. La complétion avec des contraintes d'appartenance a été introduite (Comon, 1992; Hintermeier, 1995). Dans (Nieuwenhuis et Rubio, 1995a), outre les contraintes équationnelles, il existe des contraintes d'ordres.

La règle d'expansion de la complétion basique dans la théorie vide est la règle de calcul de paire critique basique, *Paire Critique Basique*. Nous proposons ici différents systèmes d'inférence contenant cette règle d'expansion et différentes règles de contraction. Les règles de contraction définissent différentes stratégies de contraction. Dans le chapitre 4, nous définissons le système d'inférence *BCES* qui est complet pour la complétion basique avec simplification et sa stratégie de contraction, la stratégie de simplification E-cycle.

2.3.4.1 Paire Critique Basique

Nous présentons ici la règle de calcul de paires critiques basiques, (*Paire Critique Basique*) et son implantation en ELAN.

Paire Critique Basique

$$\frac{g[l'] \approx d\,[\![c1]\!] \qquad l \approx r\,[\![c2]\!]}{\mathbf{g[r] \approx d\,[\![c1 \ \wedge\ c2 \ \wedge\ l =^? l']\!]}} \quad \text{si :}$$

- l' n'est pas une variable,
- il existe une substitution σ telle que $\sigma \in CSU(c1 \ \wedge\ c2 \ \wedge\ l =^? l')$, $\sigma(l) \not\preceq_t \sigma(r)$ et $\sigma(g[l']_\omega) \not\preceq_t \sigma(d)$.

Cette règle d'inférence s'exprime par la transition suivante. Γ est un ensemble d'égalités.

$$\{g[l']_\omega \approx d\,[\![c1]\!],\, l \approx r\,[\![c2]\!]\} \cup \Gamma \to \{g[l']_\omega \approx d\,[\![c1]\!],\, l \approx r\,[\![c2]\!],\, g[r]_\omega \approx d\,[\![c1 \ \wedge\ c2 \ \wedge\ l =^? l']\!]\} \cup \Gamma$$

Paire Critique Basique	[BCP] **Basic_Critical_Pair** (g=d[c1]<counter1>, l=r[c2]<counter2>,Counter)
$\dfrac{g[l']_\omega \approx d\,[\![c1]\!] \qquad l \approx r\,[\![c2]\!]}{\boxed{g[r]_\omega \approx d\,[\![c1 \wedge c2 \wedge l =^? l']\!]}}$	=> **BCPNewEq**
si : – l' n'est pas une variable,	where *omega := (ChooseOccurence)nvocc(g)*
– Il existe une substitution σ telle que : – $\sigma \in CSU(c1 \ \wedge \ c2 \ \wedge \ l' =^? l)$,	where *unif:= (unifys)c1 & c2 & g at omega= ?l* where sigma := ()constraint_to_subst(unif)
– $\sigma(l) \not\preceq_t \sigma(r)$ et $\sigma(g[l']) \not\preceq_t \sigma(d)$, et	*if not sigma(d) >lpo sigma(g)* *if not sigma(r) >lpo sigma(l)*
	where posempty := ()eq_list[int](omega,nil) choose try
– Si $\omega = \epsilon$,	*if posempty* where *cond1 := ()sigma(d) >lpo sigma(r)* choose try
alors $(\sigma(d) \succ_t \sigma(r)$	*if cond1* where **BCPNewEq := ()g[r] at omega= d[unif]<Counter>** try if not cond1
ou $\sigma(r) \not\succ_t \sigma(d))$.	*if not sigma(r) >lpo sigma(d)* where **BCPNewEq := ()g[r] at omega= d[unif]<Counter>** end try if not posempty where **BCPNewEq := ()g[r] at omega= d[unif]<Counter>** end end

Parmi les avantages de la complétion basique, on peut également citer le fait que dans certains cas elle évite que la même égalité ne soit déduite plusieurs fois comme le montre l'exemple suivant, basé sur le fait que la complétion basique interdit les inférences dans la contrainte, ce qui implique un meilleur contrôle de l'expansion de l'espace de recherche.

Exemple 15 *Considérons l'ensemble d'égalités* $E = \{h(f(a)) \approx b,\ f(x) \approx g(x),\ a \approx c\}$ *et la précédence* $h \succ_p f \succ_p g \succ_p a \succ_p b \succ_p c$.

Si nous utilisons la règle Paire Critique *de la complétion sans échec et une stratégie de recherche particulière, les inférences de paire critique suivantes sont à réaliser.*

$$\frac{h(f(a)) \approx b \qquad f(x) \approx g(x)}{h(g(a)) \approx b}$$

$$\frac{h(g(a)) \approx b \qquad a \approx c}{h(g(c)) \approx b}$$

$$\frac{h(f(a)) \approx b \qquad a \approx c}{h(f(c)) \approx b}$$

$$\frac{h(f(c)) \approx b \qquad f(x) \approx g(x)}{h(g(c)) \approx b}$$

Nous obtenons $E_\infty = \{h(f(a)) \approx b,\ f(x) \approx g(x),\ h(g(a)) \approx f(b),\ a \approx c,\ h(f(c)) \approx b\}$.

L'égalité $h(g(c)) \approx b$ *est déduite de deux manières différentes. Par contre, si l'on utilise la règle* Paire Critique Basique *de la complétion basique et la même stratégie de recherche que précédemment, l'égalité n'est déduite qu'une seule fois.*

$$\frac{h(f(a)) \approx b \qquad f(x) \approx g(x)}{h(g(x)) \approx b\,[\![x \stackrel{?}{=} a]\!]}$$

$$\frac{h(f(a)) \approx b \qquad a \approx c}{h(f(c)) \approx b}$$

$$\frac{h(f(c)) \approx b \qquad f(x) \approx g(x)}{h(g(x)) \approx b\,[\![x \stackrel{?}{=} c]\!]}$$

Correction et complétude Le système d'inférence constitué uniquement de la règle *Paire Critique Basique* a été prouvé correct et complet lorsque l'ensemble d'égalités de départ n'est pas contraint dans (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995). On dit aussi que la complétion basique est correcte et complète.

Dans la suite, nous donnons des règles de contraction communément utilisées en combinaison avec la règle *Paire Critique Basique*, nous définissons différents systèmes d'inférence et différentes stratégies de contraction.

2.3.4.2 Règles de contraction : Simplification Standard

Nous étendons ici la règle *Simplification Standard 1* définie dans la section 2.3.2 pour prendre en compte les contraintes. La nouvelle règle ainsi obtenue est la règle *Simplification Standard*. La *stratégie de simplification standard* est basée sur l'utilisation de cette règle.

Le système d'inférence constitué de la règle *Paire Critique Basique* et de la règle *Simplification Standard* définie ci-dessous est appelé *BCSS*.

Simplification Standard

$$\frac{g[l'] \approx d\,[\![c1]\!] \qquad l \approx r\,[\![c2]\!]}{\mathbf{g}[\mu(\sigma_2(\mathbf{r}))] \approx \mathbf{d}\,[\![\mathbf{c1} \wedge \mu(\mathbf{c2})]\!]} \quad \text{si :}$$

- l' n'est pas une variable,
- Il existe $\sigma_1 = mgu(c1)$, $\sigma_2 = mgu(c2)$, et un filtre μ, tels que $\sigma_1(l') = \mu(\sigma_2(l))$, $\mu(\sigma_2(l)) \succ_t \mu(\sigma_2(r))$ et si $\omega = \epsilon$, alors μ n'est pas une substitution de renommage ou $\sigma_1(d) \succ_t \mu(\sigma_2(r))$.

Cette règle d'inférence s'exprime par la transition suivante. Γ est un ensemble d'égalités.

$$\{g[l']_\omega \approx d\,[\![c1]\!],\, l \approx r\,[\![c2]\!]\} \cup \Gamma \rightarrow \{l \approx r\,[\![c2]\!],\, g[\mu(\sigma_2(r))]_\omega \approx d\,[\![c1 \;\wedge\; \mu(c2)]\!]\} \cup \Gamma$$

Le code ELAN correspondant à cette règle d'inférence est donné ici.

Simplification Standard	[SSimp] **Standard_Simplification** (g=d[c1]<counter1>, l=r[c2]<counter2>, Counter)
$\dfrac{g[l']_\omega \approx d\,[\![c1]\!] \qquad l \approx r\,[\![c2]\!]}{\boxed{g[\mu(\sigma_2(r))]_\omega \approx d\,[\![c1 \wedge \mu(c2)]\!]}}$	=> **SSimpNewEq**
si : – l' n'est pas une variable,	where *omega := (chooseOccurence)nvocc(g)*
– Il existe $\sigma_1 = mgu(c1)$, $\sigma_2 = mgu(c2)$,	where *sigma1 := ()constraint_to_subst(c1)* where *sigma2 := ()constraint_to_subst(c2)*
et un filtre μ, tels que : – $\sigma_1(l') = \mu(\sigma_2(l))$,	where *matcha := (matchs)sigma2(l) =? sigma1(g at omega)* where mu := ()constraint_to_subst(matcha) where mus2l := ()mu(sigma2(l)) where mus2r := ()mu(sigma2(r))
– $\mu(\sigma_2(l)) \succ_t \mu(\sigma_2(r))$ et	if *mus2l >lpo mus2r* where posempty := ()eq_list[int](omega,nil) where mur := ()mu(r) where unif := (unifys)c1 & c2 & matcha choose
– Si $\omega = \epsilon$,	try *if posempty* where sigma1d := ()sigma1(d) where *isren := ()is_renaming(mu)* where *order := ()sigma1d >lpo mus2r* choose
alors (μ n'est pas une substitution de renommage	try *if not isren* where **SSimpNewEq := ()g[mur] at omega= d [unif]<Counter>**
ou $\sigma_1(d) \succ_t \mu(\sigma_2(r))$).	try if isren *if order* where **SSimpNewEq := ()g[mur] at omega= d [unif]<Counter>** end try if not posempty where **SSimpNewEq := ()g[mur] at omega= d[unif]<Counter>** end end

Correction et complétude *BCSS* est correct mais il n'est pas complet comme le montre le contre-exemple suivant (Nieuwenhuis et Rubio, 1992a). On dit aussi que la complétion basique avec simplification standard est correcte mais pas complète et que la stratégie de simplification standard n'est pas complète en combinaison avec la complétion basique.

Exemple 16 *Soit E l'ensemble d'égalités $\{a \approx b\ (1),\ f(g(x)) \approx g(x)\ (2),\ f(g(a)) \approx b\ (3)\}$ que nous voulons compléter en utilisant les règles d'inférence de* BCSS. *La précédence utilisée est $f \succ_p g \succ_p a \succ_p b$. Nous utilisons une stratégie de recherche particulière.*

La règle Paire Critique Basique *est d'abord appliquée.*

$$\frac{f(g(x)) \approx g(x)\ (2) \qquad f(g(a)) \approx b\ (3)}{g(x) \approx b\,[\![x =^? a]\!]\ (4)}$$

La règle Simplification Standard *est ensuite appliquée.*

$$\frac{f(g(a)) \approx b\ (3) \qquad g(x) \approx b\,[\![x =^? a]\!]\ (4)}{f(b) \approx b\ (5)}$$

L'égalité (3) est donc supprimée de l'espace de recherche.

Il n'y a alors plus d'inférence à réaliser. L'ensemble d'égalités saturé est: $E_\infty = \{a \approx b\ (1),\ f(g(x)) \approx g(x)\ (2),\ g(x) \approx b\,[\![x =^? a]\!]\ (4),\ f(b) \approx b\ (5)\}$.

BCSS *n'est pas complet, car il n'est pas possible de montrer que $g(b) \approx b$ par une preuve par réécriture alors que cette égalité est une égalité vraie. L'ensemble E_∞ n'est pas convergent.*

2.3.4.3 Règles de contraction : Blocage Standard

Le blocage consiste à enlever de l'espace de recherche les égalités ayant une contrainte réductible. La transition pour le blocage standard est la suivante.

Blocage Standard

$$\{g \approx d\,[\![\mathrm{c1}[l']]\!],\ l \approx r\,[\![\mathrm{c2}]\!]\} \cup \Gamma \rightarrow \{l \approx r\,[\![\mathrm{c2}]\!]\} \cup \Gamma$$

si :

- Γ est un ensemble d'égalités,
- l' n'est pas une variable,
- il existe $\sigma_1 = mgu(\mathrm{c1})$, $\sigma_2 = mgu(\mathrm{c2})$ et un filtre μ, tels que $\sigma_1(l') = \mu(\sigma_2(l))$.

L'égalité $g \approx d\,[\![\mathrm{c1}[l']]\!]$ est dite *bloquée* de manière standard, elle est supprimée de l'espace de recherche.

Cette règle s'exprime en ELAN par deux règles de réécriture *SBlockingLeft* et *SBlokingRight*. Dans l'implantation, il est en effet nécessaire de différencier le blocage à gauche et le blocage à droite. *SBlockingLeft* permet de bloquer une égalité $g \approx d\,[\![\mathrm{c1}]\!]$ dans g et *SBlockingRight* dans d.

Blocage Standard	[SBlockingLeft] **Standard_Blocking(**
$\{g \approx d\,[\![c1[l']]\!],\ l \approx r\,[\![c2]\!]\} \cup \Gamma$ $\rightarrow$ $\{l \approx r\,[\![c_2]\!]\} \cup \Gamma$ si : – l' n'est pas une variable, – il existe $\sigma_1 = mgu(c1)$ et – il existe $\sigma_2 = mgu(c2)$ et un filtre μ, tels que $\sigma_1(l') = \mu(\sigma_2(l))$.	g=d[c1]<counter1>, l=r[c2]<counter2>, newcounter) => EqEmpty /*Retourne une égalité vide*/ /* propagate applique la contrainte c1 */ /* à l'égalite g=d */ where propgd := (propagate)propagate(g=d[c1]<counter1>) where propg := ()lhs(egalite(propgd)) where *omega :=* *(chooseOccurence)nvocc(propg)* where sigma1 := ()constraint_to_subst(c1) /* Teste si omega est une position */ /* dans la contrainte */ if isconstraintpos(g,omega,sigma1) where *sigma2 := ()constraint_ to_ subst(c2)* where *matcha := (matchs)sigma2(l) =? sigma1(g) at omega* where mu := ()constraint_to_subst(matcha) where mus2l := ()mu(sigma2(l)) where mus2r := ()mu(sigma2(r)) where unif := (unifys)c1 & mu(c2) if mus2l >lpo mus2r end

Correction et complétude Le système d'inférence *BCSB* composé des règles *Paire Critique Basique* et *Blocage Standard* est correct mais il n'est pas complet. On dit aussi que la complétion basique avec blocage standard est correcte mais pas complète que la stratégie de blocage standard n'est pas complète en combinaison avec la complétion basique.

Considérons l'exemple suivant.

Exemple 17 *Considérons l'ensemble d'égalités* $E = \{h(a,b) \approx a\ (1),\ h(x,y) \approx g(x,f(y))\ (2),\ g(a,z) \approx z\ (3),\ f(b) \approx c\ (4)\}$ *et la précédence* $h \succ_p g \succ_p a \succ_p f \succ_p b \succ_p c$. *On considère une stratégie de recherche particulière.*

Une inférence de paire critique basique est d'abord effectuée :

$$\frac{h(x,y) \approx g(x,f(y))\ (2) \qquad h(a,b) \approx a\ (1)}{g(x,f(y)) \approx a\,[\![x =^? a \wedge y =^? b]\!]\ (5)}$$

L'inférence de paire critique basique suivante a ensuite lieu :

$$\frac{g(x,f(y)) \approx a\,[\![x =^? a \wedge y =^? b]\!]\ (5) \qquad g(a,z) \approx z\ (3)}{a \approx z\,[\![z =^? f(b)]\!]\ (6)}$$

L'égalité (5) est standard bloquée par l'égalité (6), elle est donc enlevée de l'espace de recherche. L'égalité (6) est standard bloquée par l'égalité (4), elle est donc également enlevée de l'espace de recherche.

Il n'y a alors plus d'inférences à réaliser et on se retrouve avec l'ensemble saturé $E_\infty = E$.

BCSB *n'est pas complet, car il n'est pas possible de montrer que $f(b) \approx a$ est vraie par une preuve par réécriture alors que cette égalité est une égalité vraie. L'ensemble E_∞ n'est pas convergent.*

On peut également définir le système d'inférence *BCSSSB* contenant les règles *Paire Critique*, *Simplification Standard* et *Blocage Standard.* Ce système est correct mais pas complet.

2.3.4.4 Règles de contraction : Simplification Basique

La *simplification basique* est une stratégie basée sur l'utilisation d'une règle de contraction appelée *Simplification Basique.* Elle est liée à la notion de réductibilité-relativement-à introduite dans (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995). Nous appelons ici cette notion, la notion de *substitution-réductibilité-relativement-à-modulo*[8]. Il s'agit de savoir si une égalité contrainte est substitution-réduite relativement à une autre égalité contrainte modulo un filtre. Les contraintes des égalités en complétion basique sont des contraintes en forme résolue. L'élément « substitution » intervient car cette notion est basée presque exclusivement sur les unificateurs les plus généraux des contraintes des deux égalités contraintes. L'élément « filtre » intervient car cette notion est utilisée pour faire une simplification. La simplification basique contient la simplification standard, qui elle, n'est basée que sur l'existence d'un filtre.

Définition 11 *Soit $s \approx t$ une instance close de l'égalité $s' \approx t' [\![c]\!]$. $s \approx t$ est de la forme $\sigma(s') \approx \sigma(t')$, où $\sigma \in Sol_G(c)$.*

$s \approx t$ est substitution-réduite *par rapport à un système de réécriture R, si pour tout $x \in Dom(\sigma) \cap Var(s' \approx t')$, $\sigma(x)$ est irréductible par rapport à R.*

Un égalité non close $s' \approx t' [\![c]\!]$ est substitution-réduite *par rapport à un système de réécriture R, si toutes ses instances closes $\sigma(s') \approx \sigma(t')$ pour $\sigma \in Sol_G(c)$ sont substitution-réduites par rapport à R.* □

Exemple 18
- *L'égalité $f(a) \approx c$ instance close de $f(a) \approx c[\![\top]\!]$ est substitution-réduite par rapport au système de réécriture $R = \{a \to b\}$.*
- *L'égalité $f(a) \approx c$ instance close de $f(x) \approx c[\![f(x) =^? f(a)]\!]$ n'est pas substitution-réduite par rapport au système de réécriture $R = \{a \to b\}$. On en déduit que $f(x) \approx c[\![f(x) =^? f(a)]\!]$ n'est pas substitution-réduite par rapport au système de réécriture $R = \{a \to b\}$.*
- *$f(x) \approx c[\![f(x) =^? f(c)]\!]$ est substitution-réduite par rapport au système de réécriture $R = \{h(x) \to x\}$, car sa seule instance close $f(c) \approx c$ l'est.*

Définition 12 *Soient $e_1 [\![c_1]\!]$ et $e_2 [\![c_2]\!]$ deux égalités contraintes. $e_2 [\![c_2]\!]$ est* substitution-réduite relativement à *$e_1 [\![c_1]\!]$ modulo ρ, si pour tout système de réécriture R, pour toute instance close de $e_1 [\![c_1]\!]$, $\tau(\sigma_1(e_1))$ pour $\sigma_1 = mgu(c_1)$ substitution-réduite par rapport à R, $\tau(\rho(\sigma_2(e_2)))$ pour $\sigma 2 = mgu(c_2)$ est substitution-réduite par rapport à R à toutes les positions de e_2 où $x \in Dom(\sigma_2)$ apparaît.* □

Cette propriété de substitution-réductibilité-relativement-à-modulo est assez générale. La **condition suffisante** suivante implique cette propriété. $e_2 [\![c_2]\!]$ est *substitution-réduite relativement à* $e_1 [\![c_1]\!]$ modulo ρ, si pour tout x de $Dom(\sigma_2)$, il existe une variable y de $Dom(\sigma_1) \cap Var(e_1)$ telle que $\rho(\sigma_2(x))$ soit un sous-terme de $\sigma_1(y)$. Cette condition peut être transformée de la manière suivante : $e_2 [\![c_2]\!]$ est *substitution-réduite relativement à* $e_1 [\![c_1]\!]$ modulo ρ, si pour tout terme t de $Ran(\sigma_2)$, il existe une variable y de $Dom(\sigma_1) \cap Var(e_1)$ telle que $\rho(t)$ soit un sous-terme

8. Dans la suite, nous écrivons réductibilité-relativement-à au lieu de réductibilité-relativement-à-modulo pour améliorer la lisibilité.

de $\sigma_1(y)$. On voit ici que la notion de substitution-réductibilité-relativement-à est seulement liée aux unificateurs les plus généraux de c_1 et c_2 et non aux squelettes e_1 et e_2. Le test n'est qu'une inclusion des images de substitutions.

Exemple 19 – $x \approx y [\![x =^? i(y)]\!]$ *est substitution-réduite relativement à* $x \approx o [\![x =^? i(o)]\!]$ *modulo* $\rho = [y \mapsto o]$.
En effet, ici σ_2 *est* $\{y \mapsto i(y)\}$ *et* σ_1 *est* $\{x \mapsto i(o)\}$. *L'image de* σ_2 *est donc* $\{i(y)\}$. *Soit* t *le terme de la définition. On a :* $t = i(y)$. *D'où,* $\rho(t) = i(o)$. $\sigma_1(x) = i(o)$. *Donc,* $\rho(t)$ *est un sous-terme de* $\sigma_1(x)$.

– $g(x) \approx c$ *est substitution-réduite relativement à* $f(g(x)) \approx b [\![x =^? a]\!]$. *Une égalité avec une contrainte triviale est substitution-réduite relativement à n'importe quelle autre égalité.*

– $f(z) \approx b [\![x =^? h(t)]\!]$ *est substitution-réduite relativement à* $k(x) \approx c [\![x =^? f(h(y))]\!]$ *modulo* $\rho = [t \mapsto y]$.
En effet, ici σ_2 *est* $\{x \mapsto h(t)\}$ *et* σ_1 *est* $\{x \mapsto f(h(y))\}$. *L'image de* σ_2 *est donc* $\{h(t)\}$. *Soit* t' *le terme de la définition. On a :* $t' = h(t)$. *D'où,* $\rho(t') = h(y)$. $\sigma_1(x) = f(h(y))$. *Donc,* $\rho(t')$ *est un sous-terme de* $\sigma_1(x)$.

– $g(x) \approx c [\![x =^? a]\!]$ *n'est substitution-réduite relativement à* $f(g(a)) \approx b$ *modulo* $\rho = id$.
En effet, ici σ_2 *est* $\{x \mapsto a\}$ *et* σ_1 *est id. L'image de* σ_2 *est donc* $\{a\}$. *Soit* t *le terme de la définition. On a :* $t = a$. *D'où,* $\rho(t) = a$. *Il n'existe pas de variable* x *dans* $Dom(\sigma_1) \cap Var(f(g(a)) \approx b)$ *telle que* $\rho(t)$ *soit un sous-terme de* $\sigma_1(x)$.

Le test de substitution-réductibilité-relativement-à est réalisé en ELAN par la fonction *is_reduced_relative_to*.

```
rules for bool
  l, r, g, d                : term;
  counter1, counter2        : int;
  mu, sigma1, sigma2        : substitution;
  c1, c2                    : constraint;
 global
[] is_reduced_relative_to(l=r[true]<counter1>,sigma1,
   g=d[c2]<counter2>,sigma2,mu)
=> true
 end

[] is_reduced_relative_to(l=r[c1]<counter1>,sigma1,
   g=d[c2]<counter2>,sigma2,mu)
=> include_subterms(mu(ran(sigma1)),ran(sigma2))
 end
 end
```

Nous définissons maintenant la notion abstraite de *SR-redondance* (substitution-réduite-redondance). La règle de simplification basique est une application spécifique de la SR-redondance (Bachmair et al., 1995).

Définition 13 *Soit E un ensemble d'égalités et e une égalité contrainte. e est* SR-redondante[9]

9. Cette définition de la redondance n'est pas aussi compliquée que celle présentée dans (Bachmair et al., 1995). En effet, elle ne comprend pas la blocage basique. Nous la présentons ainsi pour des raisons de simplicité.

dans E si, pour toute instance close e_g de e, il existe des instances closes $e_{1_g}, \ldots, e_{n_g}$ de $e_1, \ldots, e_n$ dans E, telles que :
- *pour tout i, $e_{i_g} \prec_e e_g$,*
- *$e_{1_g}, \ldots, e_{n_g} \models e_g$, et*
- *pour tout i, e_{i_g} est substitution-réduite relativement à e_g.* □

Une égalité SR-redondante peut être supprimée de l'espace de recherche. Une inférence est *SR-redondante*, si une de ses prémisses ou sa conclusion est SR-redondante.

La règle de simplification *Simplification Basique* est la suivante.

Simplification Basique

$$\frac{g[l']_\omega \approx d\,[\![c1]\!] \qquad l \approx r\,[\![c2]\!]}{\mathbf{g}[\mu_{\min}(\sigma_2(\mathbf{r}))]_\omega \approx \mathbf{d}\,[\![\mathbf{c1} \wedge \mu(\mathbf{c2})]\!]} \quad \text{si :}$$

- l' n'est pas une variable,
- il existe $\sigma_1 = mgu(c1)$, $\sigma_2 = mgu(c2)$, et un filtre μ, tels que $\sigma_1(l') = \mu(\sigma_2(l))$, $\mu(\sigma_2(l)) \succ_t \mu(\sigma_2(r))$, et si $\omega = \epsilon$, alors μ n'est pas une substitution de renommage ou $\sigma_1(d) \succ_t \mu(\sigma_2(r))$,
- μ_{min} est calculée comme expliqué ci-dessous, et
- $l \approx r\,[\![c2]\!]$ est substitution-réduite relativement à $g[l'] \approx d\,[\![c1]\!]$ modulo μ.

La transition correspondante est la suivante. Γ est un ensemble d'égalités.

$$\{g[l']_\omega \approx d\,[\![c1]\!],\ l \approx r\,[\![c2]\!]\} \cup \Gamma \rightarrow \{l \approx r\,[\![c2]\!],\ g[\mu_{min}(\sigma_2(l))]_\omega \approx d\,[\![c1 \wedge \mu(c2)]\!]\} \cup \Gamma$$

Dans la conclusion de la règle d'inférence, on aurait pu écrire μ au lieu de μ_{min}. L'intérêt d'utiliser μ_{min} est d'instancier r au minimum dans g et donc d'optimiser l'utilisation des contraintes. Ce filtre minimum a été introduit dans (Bachmair et al., 1995). Le calcul du filtre minimum de la règle d'inférence décrite précédemment se fait de la manière suivante. Pour toute variable x de $Var(\sigma_2(l))$ aux positions $q_1, \ldots, q_m$, alors s'il existe un q_i tel que q_i soit une position de substitution dans l', alors $\mu_{min}(x) = x$, sinon $\mu_{min}(x)$ est la généralisation la plus spécifique de $l'_{|q_1}, \ldots, l'_{|q_m}$ (Huet, 1980).

Exemple 20 – *L'inférence suivante est une inférence de simplification basique.*

$$\frac{i(x) \approx o\,[\![x =^? i(o)]\!] \qquad x \approx y\,[\![x =^? i(y)]\!]}{y \approx o\,[\![y =^? i(o)]\!]}$$

En effet, toutes les conditions d'application de la règle Simplification Basique *sont reunies et en particulier, $x \approx y\,[\![x =^? i(y)]\!]$ est substitution-réduite relativement à $i(x) \approx o\,[\![x =^? i(o)]\!]$ modulo $\rho = [y \mapsto i(o)]$ (voir exemple 19).*

– *L'inférence suivante n'est pas une inférence de simplification basique.*

$$\frac{f(g(a)) \approx b \qquad g(x) \approx c\,[\![x =^? a]\!]}{f(c) \approx b}$$

En effet, toutes les conditions d'application de la règle Simplification Basique *ne sont pas reunies. $g(x) \approx c\,[\![x =^? a]\!]$ n'est substitution-réduite relativement à $f(g(a)) \approx b$ modulo $\rho = id$ (voir exemple 19).*

L'implantation en ELAN de la règle de simplification basique est donnée ici.

Simplification Basique	[BSimpLeft] **Basic_Simplification** (g=d[c1]<counter1>, l=r[c2]<counter2>, Counter)
$$\frac{g[l'] \approx d[\![c1]\!] \qquad l \approx r[\![c2]\!]}{g[\mu_{min}(\sigma_2(l))] \approx d[\![c1 \wedge \mu(c2)]\!]}$$	=> **BSimpNewEq**
si : – l' n'est pas une variable, – il existe $\sigma_1 = mgu(c1)$, $\sigma_2 = mgu(c2)$, et un filtre μ, tels que :	where *omega := (chooseOccurence)nvocc(g)* where *sigma1 := ()constraint_to_subst(c1)* where *sigma2 := ()constraint_to_subst(c2)*
– $\sigma_1(l') = \mu(\sigma_2(l))$,	where matcha := (matchs)sigma2(l) =? sigma1(g at omega) where mu := ()constraint_to_subst(matcha) where min_mu := ()min_match(g,c1 ,sigma1,l,c2,sigma2,omega,mu) where mus2l := ()mu(sigma2(l)) where mus2r := ()mu(sigma2(r))
– $\mu(\sigma_2(l)) \succ_t \mu(\sigma_2(r))$, – $l \approx r[\![c2]\!]$ est substitution-réduite relativement à $g[l'] \approx d[\![c1]\!]$ modulo μ, et	*if mus2l >lpo mus2r* if *IsReducedRelativeTo(l=r[c2]<counter2>,* *sigma2,g=d[c1]<counter1>,sigma1,mu)* where posempty := ()eq_list[int](omega,nil) where min_mur := ()min_mu(r) where unif := (unifys)c1 & c2 & matcha choose
– Si $\omega = \epsilon$,	try *if posempty* where sigma1d := ()sigma1(d) where *isren := ()is_renaming(mu)* where *order := ()sigma1d >lpo mus2r* choose
alors (μ n'est pas une substitution de renommage	try *if not isren* where **BSimpNewEq := ()g[min_mur] at omega= d[unif]<Counter>**
ou $\sigma_1(d) \succ_t \mu(\sigma_2(r))$).	try if isren if *order* where **BSimpNewEq := ()g[min_mur] at omega= d[unif]<Counter>** end try if not posempty where **BSimpNewEq := ()g[min_mur] at omega= d[unif]<Counter>** end end

Correction et complétude Le système d'inférence *BCBS* composé de la règle *Paire Critique Basique* et de la règle *Simplification Basique* est correct et complet. La complétude a été prouvé dans (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995). On dit aussi que la complétion basique avec simplification basique est correcte et complète et que la stratégie de simplification basique est complte en combinaison avec la complétion basique.

2.3.4.5 Règles de contraction : Blocage Basique

La stratégie de blocage basique est définie à partir de la règle *Blocage Basique* qui comme la règle de *Simplification Basique* est basée sur la notion de substitution-réductibilité-relative-à. Cette règle est la suivante :

Blocage Basique

$\{g \approx d\,[\![c1[l']]\!], l \approx r\,[\![c2]\!]\} \cup \Gamma \rightarrow \{l \approx r\,[\![c_2]\!]\} \cup \Gamma$ si :

- Γ est un ensemble d'égalités,
- l' n'est pas une variable,
- il existe un filtre μ, $\sigma_1 = mgu(c1)$ et $\sigma_2 = mgu(c2)$ telle que $\sigma_1(l') = \mu(\sigma_2(l))$, et
- $l \approx r\,[\![c2]\!]$ est substitution-réduite relativement à $g \approx d\,[\![c1[l']]\!]$.

L'implantation en ELAN de cette règle est similaire à celle de la règle *Blocage Standard*, nous ne la donnons donc pas ici. Il convient seulement d'ajouter la condition de substitution-réductibilité-relativement-à.

2.3.4.6 La rétraction

La rétraction est aussi appelée propagation ou réveil de la contrainte dans la littérature. Elle applique une partie de la contrainte au squelette de l'égalité.

Utiliser la rétraction est une alternative lourde de conséquences. Elle s'utilise couplée aux règles *Simplification Basique* et *Blocage Basique*. Si on peut appliquer la règle *Simplification Standard* ou la règle *Blocage Standard*, mais pas les règles *Simplification Basique* et *Blocage Basique* car les conditions ne sont pas vérifiées, c'est donc que l'égalité « droite » n'est pas substitution-réduite relativement à l'égalité « gauche ». Pour rendre ce test vrai, l'égalité « gauche » est rétractée. D'un côté, la règle de rétraction va permettre de réaliser plus de simplifications tout en préservant la complétude et de l'autre, elle enlève un point fort de la complétion basique qui est l'utilisation des contraintes.

Considérons l'exemple suivant.

Exemple 21 *Soit la précédence* $f \succ_p a \succ_p b \succ_p c$.

L'inférence suivante n'est pas une inférence de simplification basique car $f(x) \approx c\,[\![x =^? a]\!]$ *n'est pas substitution-réduite relativement à* $f(a) \approx b$. *Il s'agit donc d'une inférence de paire critique basique.*

$$\frac{f(a) \approx b \qquad f(x) \approx c\,[\![x =^? a]\!]}{b \approx c}$$

Or, si on rétracte l'égalité $f(x) \approx c\,[\![x =^? a]\!]$ *en l'égalité* $f(a) \approx c$, *on a une simplification basique.*

$$\frac{f(a) \approx b \qquad f(a) \approx c}{b \approx c}$$

C'est dans ce sens que la rétraction permet de faire plus de simplifications mais aussi qu'elle enlève l'avantage d'utiliser les contraintes.

Une méthode facilement implantable pour rétracter est de procéder comme suit.

Considérons une inférence entre les égalités $g[l']_\omega \approx d\,[\![c1]\!]$ et $l \approx r\,[\![c2]\!]$ qui est une inférence de simplification standard mais pas de simplification basique. On appelle μ le filtre de la simplification standard. L'égalité $l \approx r\,[\![c2]\!]$ n'est donc pas substitution-réduite relativement à $g[l']_\omega \approx d\,[\![c1]\!]$,

$l \approx r\,[\![c2]\!]$ est rétractée pour que le test de substitution-réductibilité-relativement-à devienne vrai. Soient $\sigma_1 = mgu(c1)$ et $\sigma_2 = mgu(c2)$. Toute position p de $\sigma_2(l)$ a une position correspondante dans $\sigma_1(l')$ à cause de l'existence du filtre μ. On procède alors comme suit. Pour toute position de substitution p dans l alors, si p n'est pas une position de substitution de l' alors on doit appliquer la contrainte ou une partie de la contrainte à la variable $x = l_{|p}$. La substitution σ_2 est appliquée au terme r.

Considérons l'exemple suivant.

Exemple 22 *Soit la précédence* $k_1 \succ_p f \succ_p g \succ_p \succ_p k \succ_p h \succ_p a \succ_p b \succ_p c$.

L'inférence suivante est une inférence de simplification standard, elle n'est pas une inférence de simplification basique car $f(x_1,t) \approx k(x_1,t)\,[\![t =^? h(h(z))]\!]$ *n'est pas substitution-réduite relativement à* $k_1(f(g(x),h(y))) \approx c\,[\![x =^? a,\ y =^? h(b)]\!]$*. Dans BCBS, il s'agit donc d'une inférence de paire critique basique.*

$$\frac{k_1(f(g(x),h(y))) \approx c\,[\![x =^? a,\ y =^? h(b)]\!] \qquad f(x_1,t) \approx k(x_1,t)\,[\![t =^? h(h(z))]\!]}{k_1(k(x_1,t)) \approx c\,[\![x_1 =^? g(a), t =^? h(h(b))]\!]}$$

Or, si on applique la partie de la contrainte $t =^? h(h(z),\ t \mapsto h(t')$*, à* $f(x_1,t) \approx k(x_1,t)\,[\![t =^? h(h(z))]\!]$ *pour obtenir l'égalité* $f(x_1,h(t')) \approx k(x_1,h(t'))\,[\![t' =^? h(z)]\!]$*, alors on a la simplification basique suivante.*

$$\frac{k_1(f(g(x),h(y))) \approx c\,[\![x =^? a,\ y =^? h(b)]\!] \qquad f(x_1,h(t')) \approx k(x_1,h(t'))\,[\![t' =^? h(z)]\!]}{k_1(k(x_1,h(t'))) \approx c\,[\![x =^? g(a),\ t' =^? h(b)]\!]}$$

La rétraction est décrite par la règle d'inférence *Rétraction* suivante :

Rétraction

$$\frac{g \approx d\,[\![c]\!]}{\sigma(\mathbf{g} \approx \mathbf{d})\,[\![\mathbf{c'}]\!]} \quad \text{si :}$$

- $\sigma(g \approx d)\,[\![c']\!]$ est un rétracté de $g \approx d\,[\![c]\!]$.

2.3.5 Complétion contrainte et basique modulo une théorie équationnelle

Dans la section précédente l'ensemble des axiomes était vide, ici, nous traitons le cas général où AX représente une théorie équationnelle différente de la théorie vide. La complétion contrainte (respectivement basique) modulo est la combinaison de la complétion contrainte (respectivement basique) et de la complétion modulo. Les complétions contrainte et basique modulo allient donc les avantages de ces deux complétions. Les principaux résultats en recherche de preuve utilisant des méthodes modulo une théorie équationnelle avec contraintes concerne la paramodulation et la superposition. Ces résultats s'appliquent à la complétion qui est un cas particulier de superposition et de paramodulation. R. Nieuwenhuis et A. Rubio ont étudié la superposition avec contraintes modulo AC dans (Nieuwenhuis et Rubio, 1994). La paramodulation contrainte et basique modulo est présentée dans (Vigneron, 1994a).

La complétion basique modulo nécessite l'utilisation d'extensions comme dans le cas sans contraintes. Les extensions sont définies ici.

Définition 14 *Soient $g \approx d$ un axiome de AX et $l \approx r \llbracket c \rrbracket$ une égalité de E ayant des ensembles de variables disjoints.*

S'il existe une position ω dans g telle que $g_{|\omega}$ ne soit pas une variable, une substitution σ telle que $\sigma \in CSU(c \wedge g_{|\omega} =^?_{AX} l)$ et $\sigma(l) \succ_t \sigma(r)$, alors $g[l]_\omega \approx g[r]_\omega \llbracket c \wedge g_{|\omega} =^?_{AX} l \rrbracket$ est une extension *de l'égalité $l \approx r$ par rapport à $g \approx d$.* □

Nous appelons BC_m le système d'inférence de complétion basique avec simplification basique composé des règles *Paire Critique Basique AX* et *Simplification Basique AX* décrites ci-dessous.

La règle d'expansion *Paire Critique Basique AX* est présentée ici.

Paire Critique Basique AX

$$\frac{g \approx d \llbracket c1 \rrbracket \qquad l \approx r \llbracket c2 \rrbracket}{\mathbf{g''[r''] \approx d \llbracket c1' \wedge c2' \wedge l'' =^?_{AX} l' \rrbracket}} \quad \text{si :}$$

- $g''[l']_\omega \approx d'' \llbracket c1' \rrbracket$ est l'égalité $g \approx d \llbracket c1 \rrbracket$ ou, une extension de $g \approx d \llbracket c1 \rrbracket$ et $\omega = \epsilon$,
- $l'' \approx r'' \llbracket c2' \rrbracket$ est l'égalité $l \approx r \llbracket c2 \rrbracket$ ou une extension de $l \approx r \llbracket c2 \rrbracket$,
- l' n'est pas une variable,
- il existe $\sigma \in CSU(l'' =^?_{AX} l' \wedge c1' \wedge c2')$ telle que :
- $\sigma(g''[l']) \not\preceq_t \sigma(d'')$, $\sigma(l'') \not\preceq_t \sigma(r'')$, et si $\omega = \epsilon$, alors $(\sigma(d'') \succ_t \sigma(r'')$ ou $\sigma(r'') \not\succ_t \sigma(d''))$.

Cette règle d'inférence s'exprime par la transition suivante. Γ est un ensemble d'égalités.

$$\{g \approx d \llbracket c1 \rrbracket, l \approx r \llbracket c2 \rrbracket\} \cup \Gamma \rightarrow \{g \approx d \llbracket c1 \rrbracket, l \approx r \llbracket c2 \rrbracket, g''[r''] \approx d \llbracket c1' \wedge c2' \wedge l'' =^?_{AX} l' \rrbracket\} \cup \Gamma$$

Exemple 23 *Dans l'exemple 12, on peut appliquer la règle* Paire Critique Basique *AC entre l'égalité $f(f(a,b),x) \approx f(c,x)$, extension de $f(a,b) \approx c$ et l'égalité $f(y,f(b,d)) \approx f(y,e)$, extension de $f(b,d) \approx e$.*

$$\frac{f(f(a,b),x) \approx f(c,x) \qquad f(y,f(b,d)) \approx f(y,e)}{f(c,x) \approx f(y,e) \llbracket f(f(a,b),x) =^?_{AC} f(y,f(b,d)) \rrbracket}$$

Grâce à cette inférence et aux extensions, nous pouvons obtenir une preuve par réécriture de l'égalité $f(c,d) \approx f(a,e)$.

Le principal problème de la complétion contrainte modulo est la mise au point de règles de contraction qui combinées à la règle *Paire Critique Basique AX* forment un système d'inférence complet. La seule règle de simplification connue satisfaisant cette condition est la règle *Simplification Basique AX*. Les travaux sur la simplification basique sont visibles dans (Bachmair et al., 1995; Vigneron, 1994b).

La règle *Simplification Basique AX* détermine la *stratégie de simplification basique dans le cas modulo*. Elle est basée sur la notion de substitution-réductibilité-relativement-à. Elle est décrite ci-dessous comme cas particulier de la règle de *Simplification Basique* décrite dans (Vigneron, 1994b). La condition suffisante de substitution-réductibilité-relative-à-modulo de la section 2.3.4 est étendue ici.

$e_2 \llbracket c_2 \rrbracket$ est *substitution-réduite relativement à* $e_1 \llbracket c_1 \rrbracket$ modulo ρ pour $\sigma_1 \in CSU(c_1)$ et $\sigma_2 \in CSU(c_2)$, si pour tout terme $t \in Ran(\sigma_2)$, il existe une variable $y \in Dom(\sigma_1) \cap Var(e_1)$ telle que $\rho(t)$ soit un sous-terme de $\sigma_1(y)$.

Simplification Basique AX

$$\frac{g[l']_\omega \approx d\,[\![c1]\!] \qquad l \approx r\,[\![c2]\!]}{\mathbf{g}[\mu_2(\sigma_2(\mathbf{r}''))]_\omega \approx \mathbf{d}\,[\![\mathbf{c1} \wedge \mu_2(\mathbf{c2}')]\!]} \quad \text{si :}$$

- $l'' \approx r''\,[\![c2']\!]$ est l'égalité $l \approx r\,[\![c2]\!]$ ou une extension de $l \approx r\,[\![c2]\!]$,
- l' n'est pas une variable,
- il existe $\sigma_2 \in MGCSU(c2)$ et un filtre μ_2, tels que pour tout $\sigma_1 \in MGCSU(c1)$, il existe un filtre μ_2' tel que $\sigma_1(l') \overset{*}{\leftrightarrow}_{AX} \mu_2(\sigma_2(l''))$, $Dom(\sigma_2) \cap Dom(\rho_2) = \emptyset$ et $\mu_2'(\mu_2(\sigma_2(l''))) \succ_t \mu_2'(\mu_2(\sigma_2(r'')))$, et $l'' \approx r''\,[\![c2']\!]$ soit substitution-réduite relativement à $g \approx d\,[\![c1]\!]$ modulo μ_2' pour σ_1 et σ_2.

Cette règle d'inférence s'exprime par la transition suivante. Γ est un ensemble d'égalités.

$$\{g \approx d\,[\![c1]\!],\ l \approx r\,[\![c2]\!]\} \cup \Gamma \rightarrow \{l \approx r,\ g[\mu_2(\sigma_2(r''))]_\omega \approx d\,[\![c1 \wedge \mu_2(c2')]\!]\} \cup \Gamma$$

L'énoncé de cette règle est très complexe. Elle différencie le filtre qui est appliqué dans l'égalité déduite, du filtre utilisé dans le test de substitution-réductibilité-relative-à-modulo. Cette règle est assez générale, car, pour une substitution $\sigma_1 \in CSU(c1)$, elle ne requiert pas d'engendrer le même filtre μ, mais seulement d'avoir une partie commune à tous les filtres μ, le filtre μ_2. La substitution μ_2' dépend de σ_1 et associe à chaque variable un terme de l'image de σ_1.

L'efficacité pratique de cette règle est compromise par différents aspects. Elle nécessite de résoudre les contraintes $c1$ et $c2$. Le test de l'existence du filtre est coûteux car il faut trouver une substitution σ_2 et vérifier les conditions données pour tous les $\sigma_1 \in CSU(c1)$. S'ajoute à ce test le test de substitution-réductibilité-relativement-à-modulo qui lui aussi utilise les substitutions solutions des contraintes $c1$ et $c2$ et les filtres μ_2'.

Du point de vue implantation, les complétions contrainte et basique modulo ont été implantées dans DATAC (Vigneron, 1994a; Vigneron, 1997; Vigneron, 1998). La règle de simplification basique a été implantée seulement en partie pour des raisons d'efficacité, le test de substitution-réductibilité-relativement-à-modulo n'a pas été implanté.

Nous revenons sur les points forts et faibles de BC_m dans l'introduction du chapitre 5. Nous proposons dans le chapitre 5, un nouveau système d'inférence pour la complétion basique modulo avec simplification $BCICS_m$ dont le but est de ne pas résoudre les contraintes mais seulement de tester leur satisfaisabilité.

2.4 Résolution et paramodulation

La notion de preuve par réfutation abordée dans la section 2.3 provient de la logique du premier ordre. Pour prouver une formule α à partir d'un ensemble d'hypothèses $\mathcal{H}$, on prouve l'*inconsistance* de l'ensemble $\neg\alpha,\mathcal{H}$. L'inconsistance est obtenue lorsqu'une *réfutation* (la clause vide, $\square$) est dérivée pendant le processus de preuve. La *résolution* et la *paramodulation* sont les deux méthodes de recherche de preuve par réfutation évoquées dans cette section. Nous nous intéressons ici principalement aux stratégies de recherche de preuve mises au point pour ces méthodes. La résolution et la paramodulation peuvent être également vues comme des méthodes de déduction par saturation.

2.4.1 Résolution

En 1965, J.A. Robinson introduit la résolution. Le système d'inférence de la résolution *RF* est composé de deux règles d'expansion, la *règle de résolution* et la *règle de factorisation*. La règle de résolution peut être vue comme la combinaison de la règle du *cut* du système de Gentzen, qui généralise la règle du *modus ponens*, et de l'utilisation de substitutions.

Résolution

$$\frac{L_1 \vee D_1 \qquad \neg L_2 \vee D_2}{\sigma(D_1) \vee \sigma(D_2)} \text{ si :}$$

- $\sigma = mgu(L_1, L_2)$.

Factorisation

$$\frac{L_1 \vee L_2 \vee D}{\sigma(L_2) \vee \sigma(D)} \text{ si :}$$

- $\sigma = mgu(L_1, L_2)$.

Correction Le système d'inférence *RF* est *correct* dans le sens où si la clause vide a été dérivée à partir de l'ensemble $\neg\alpha, \mathcal{H}$, alors l'ensemble $\neg\alpha, \mathcal{H}$ est inconsistant.

Complétude La règle de factorisation a été introduite parce que le système d'inférence de résolution composé exclusivement de la règle de résolution n'est pas complet. En effet, l'ensemble $\{P(x) \vee P(y), \neg P(u) \vee \neg P(v)\}$ est inconsistant, mais la règle de résolution ne peut pas dériver la clause vide.

Le système d'inférence *RF* est *complet* dans le sens où si $\neg\alpha, \mathcal{H}$ est inconsistant, alors on peut dériver la clause vide en utilisant les règles de *RF*.

Différentes stratégies de recherche de preuves ont été développées pour la résolution (Chang et Lee, 1973). Les stratégies de recherche de preuve concernant la résolution peuvent être vues par essence comme des stratégies de raisonnement *avant* (*forward*). Toutefois, des raisonnements *arrière* (*backward*) ont été introduits pour limiter le pouvoir générateur de la règle de résolution, qui est une règle d'expansion. Certaines autres améliorations sont dues à l'introduction de l'*élimination des tautologies* et à la *règle de subsomption*, que nous présentons brièvement.

Les stratégies SOS

Les stratégies basées sur l'utilisation d'un ensemble de support, appelées stratégies SOS, divisent l'ensemble des clauses T (hypothèses et but à prouver) en deux ensembles pendant la résolution, l'ensemble de support *sos* et l'ensemble $T \setminus sos$ tel que $T \setminus sos$ soit consistant. La règle d'inférence de la résolution ne s'applique que lorsque les deux prémisses ne sont pas dans $T \setminus sos$. La conclusion d'inférence est ajoutée à *sos*. L. Wos, D. Carson et G. Robinson proposent une stratégie SOS qui prend au départ pour *sos* la négation du but et donc pour $T \setminus sos$ l'ensemble des hypothèses de départ (Wos, Carson et Robinson, 1965). L'utilisation de cet ensemble de support de départ implique une stratégie de raisonnement *arrière*. On peut remarquer que si l'on prend pour ensemble de support de départ un sous-ensemble de $\mathcal{H}$ alors la stratégie de recherche de preuve est une stratégie *avant*. Le fait d'avoir une stratégie *avant* ou *arrière* est inhérent au choix de l'ensemble de support de départ. Les stratégies SOS appartiennent à la classe des *stratégies d'ordre*, car elles développent plusieurs tentatives de preuve.

Les stratégies SOS sont équitables dans le sens où, si T est un ensemble de clauses inconsistant et si *sos* est un sous-ensemble de T tel que $T \setminus sos$ soit consistant, alors il existe une dérivation de résolution pour déduire la clause vide à partir de T avec *sos* comme ensemble de support.

Les stratégies positives

Les stratégies positives forcent l'utilisation d'au moins une clause positive comme prémisse d'une inférence de résolution. Le cas dual est représenté par les *stratégies négatives.*

Les stratégies positives sont plus restrictives que les stratégies SOS où l'ensemble de support de départ est l'ensemble des clauses positives de départ. En effet, dans ce cas précis, les stratégies SOS autorisent des résolutions entre des prémisses non positives tant qu'une de ces prémisses au moins est dans l'ensemble de support.

Les stratégies positives et négatives font elles aussi partie des stratégies d'ordre et sont équitables.

Les stratégies d'hyper-résolution

Il existe des *stratégies d'hyper-résolution positive* et des *stratégies d'hyper-résolution négative.* La règle d'inférence de l'hyper-résolution positive a comme prémisses une clause C non positive et autant de clauses positives qu'il y a de littéraux négatifs dans C. Chaque littéral négatif de C implique une application de la règle de résolution avec une clause positive prémisse. L'avantage de cette règle est que les résultats intermédiaires des applications de la règle de résolution ne sont pas sauvegardés.

On peut remarquer que les stratégies d'hyper-résolution positive sont plus restrictives que les stratégies positives, car les stratégies positives ne garantissent pas d'avoir une clause positive conclusion lors d'une étape de résolution, alors que c'est le cas pour les stratégies d'hyper-résolution positive.

Les stratégies d'hyper-résolution font elles aussi partie des stratégies d'ordre et sont équitables.

Les stratégies linéaires

Les stratégies linéaires fonctionnent comme suit. Une *clause centrale* φ_0 est choisie dans l'ensemble des clauses de départ T telle que $S = T \setminus \{\varphi_0\}$ soit consistant. A chaque étape de dérivation, la stratégie génère une clause centrale φ_{i+1} à partir de la clause centrale du rang précédent φ_i et soit d'une clause de l'ensemble de départ, soit d'une clause centrale φ_j telle que $j < i$. La forme d'une dérivation ainsi obtenue est un *arbre linéaire.*

Les stratégies linéaires sont équitables dans le sens où, si $T = S \cup \{\varphi_0\}$ et T est consistant, alors il existe une réfutation linéaire de T.

En considérant à chaque étape la clause centrale comme le but, on voit que les stratégies linéaires sont des stratégies de réduction de sous-buts. Les stratégies linéaires cherchent une forme spécifique de preuve, une *réfutation linéaire* et elles utilisent le *retour arrière* (*backtracking*) pour y accéder. C'est dans ce sens que les stratégies linéaires sont des stratégies de réduction de sous-buts.

Elimination des tautologies et subsomption

Les stratégies de contraction en résolution reposent sur les techniques suivantes.

- L'*élimination des tautologies* passe par l'élimination des clauses contenant des paires de littéraux complémentaires. D'autres critères existent également.
  Par exemple, les clauses $C \vee \neg A \vee A$ et $C \vee s \approx s$ sont des tautologies.
- Les *stratégies de subsomption* permettent de supprimer de l'espace de recherche les clauses qui sont *subsumées* par d'autres clauses.
  Une clause C *subsume* une clause D, s'il existe une substitution σ telle que $\sigma(C) \subseteq D$ et que C n'ait pas plus de littéraux que D.
  La clause $\neg P(x) \vee Q(f(x),a)$ subsume la clause $Q(f(h(y)),a)$.

Extensions de la résolution

La résolution a été raffinée par l'utilisation de *fonctions de sélection.*

Les fonctions de sélection marquent des littéraux appelés les *littéraux sélectionnés*, des termes appelés les *termes sélectionnés* et des occurrences appelés *occurrences sélectionnées* dans une clauses. Les fonctions de sélection déterminent exactement où les inférences peuvent avoir lieu, les inférences n'ayant lieu qu'entre littéraux sélectionnés et à des positions d'occurrences sélectionnées. Nous donnons un exemple de fonction de sélection dans la section 2.4.2. L'extension de la résolution avec l'utilisation de contraintes a été proposée dans la littérature. La *résolution contrainte* a été étudiée dans le cas de la théorie vide. R. Caferra et N. Zabel (Caferra et Zabel, 1990) l'utilisent pour réfuter ou pour générer des modèles quand ils existent. W.L. Buntine et H.J. Burckert ont montré que l'utilisation des contraintes améliore considérablement l'efficacité des prouveurs de théorèmes basés sur ces techniques car ils évitent de générer des tautologies (Buntine et Bürckert, 1989; Bürckert, 1990).

Une extension évidente de la résolution à la logique du premier ordre avec égalité consiste à ajouter explicitement les *axiomes de congruence* exprimant les propriétés de l'égalité à l'ensemble des clauses. Or, la règle de résolution devient très prolifique lorsqu'on utilise ces axiomes. La *paramodulation* a été proposée comme solution aux approches de la résolution avec égalité.

2.4.2 Paramodulation

En 1969, G. Robinson et L. Wos introduisent la *paramodulation* (Robinson et Wos, 1969). La paramodulation prend en compte spécifiquement le prédicat d'égalité. Elle remplace ainsi plusieurs étapes de résolution dues à la manipulation des axiomes exprimant les propriétés de l'égalité par une étape de paramodulation composée d'une instanciation suivie d'un remplacement de sous-terme. G. Robinson et L. Wos ont prouvé la complétude réfutationnelle de leur système en présence des *axiomes de réflexivité fonctionnelle.* Les axiomes de réflexivité fonctionnelle sont les axiomes de la forme $f(x_1,\ldots,x_n) \approx f(x_1,\ldots,x_n)$ tels que les variables x_i soient différentes deux à deux et que f soit un symbole de fonction n-aire de la signature. Ils sont utilisés dans la preuve pour des inférences de paramodulation dans les variables pour permettre le passage du cas clos au cas non clos (*lifting*).

Le système d'inférence de G. Robinson et L. Wos pour la paramodulation

Nous donnons ici le système d'inférence de la paramodulation *PRF*indexPRF de G. Robinson et L. Wos. Il est composé de la *règle de paramodulation* et de la *règle de résolution équationnelle*

(*equality resolution* ou *reflexivity resolution*) et de la *règle de factorisation* décrite dans la section 2.4.1. La règle de résolution peut être codée par la règle de paramodulation et par la règle de résolution équationnelle. En effet, tout prédicat $p(s_1,\ldots,s_n)$ peut s'écrire sous la forme d'une égalité $p(s_1,\ldots,s_n) \approx \top$ comme nous l'avons vu dans le chapitre 1.

Paramodulation

$$\frac{u[s'] \approx v \vee D \qquad s \approx t \vee C}{\sigma(u[t] \approx v) \vee \sigma(D) \vee \sigma(C)} \text{ si :}$$

– $\sigma = mgu(s',s)$.

Résolution équationnelle

$$\frac{s \not\approx t \vee C}{\sigma(C)} \text{ si :}$$

– $\sigma = mgu(s,t)$.

L'exemple suivant illustre la règle de paramodulation.

Exemple 24 $$\frac{\neg P(f(x,g(y))) \vee Q(x,z) \qquad f(a,g(b)) \approx c \vee R(f(a))}{\neg P(c) \vee Q(a,z) \vee R(f(a))}$$

Au cours du temps, de nombreux raffinements de la paramodulation ont été proposés. D. Brand a montré la paramodulation est réfutationnellement complète sans utiliser les axiomes de réflexivité fonctionnelle (Brand, 1975). G. Peterson (Peterson, 1983) a montré que les inférences à des positions de variables peuvent être évitées. Ces deux résultats permettent de supprimer un nombre considérable d'inférences.

Les autres améliorations de la paramodulation concernent les domaines suivants. Des fonctions de *sélection* ont été mises au point. Divers mécanismes ont été proposés pour simplifier les clauses redondantes. L'utilisation de contraintes a été introduit dans la paramodulation et des travaux sur la paramodulation modulo ont été entrepris.

Fonctions de sélection

Nous définissons ici la fonction de sélection S_f utilisée pour définir la règle *Paramodulation basique*. S_f est définie dans (Bachmair et al., 1995). Elle généralise la fonction de sélection utilisée par L. Bachmair et H. Ganzinger dans (Bachmair et Ganzinger, 1994) pour la superposition.

La fonction de sélection S_f assigne à une clause C un ensemble d'occurrences sélectionnées non variables soumises aux conditions suivantes. Une occurrence d'un littéral de C est sélectionnée si elle contient une occurrence sélectionnée d'un terme. De plus, S_f requiert que (i) une égalité négative ou tous les littéraux maximaux soient sélectionnés et que (ii) le côté maximal d'un littéral sélectionné et tous ses sous-termes non variables soient sélectionnés. Cette fonction de sélection implique donc que si une égalité négative est maximale dans C alors elle sera sélectionnée.

D'autres types de paramodulation ont été proposés dans la littérature. On citera la *paramodulation ordonnée*. La paramodulation ordonnée permet les remplacements dans les membres les plus petits des égalités. Elle a été prouvée réfutationnellement complète dans (Hsiang et Rusinowitch, 1991). La méthode de preuve utilisé est basée sur l'utilisation d'arbres transfinis et réfère à un ordre de bien-fondé, monotone et total sur les termes clos. La *paramodulation positive* est une autre stratégie de paramodulation dont le but est d'éviter l'augmentation du nombre de littéraux négatifs.

Paramodulation basique

Comme pour la complétion, la paramodulation a été étendue à la *paramodulation basique* (Bachmair et al., 1995). La superposition de L. Bachmair et H. Ganzinger a été étendue au cas basique dans (Nieuwenhuis et Rubio, 1992a). Les contraintes permettent de contrôler la recherche en bloquant des inférences, les inférences étant interdites dans les contraintes. Les contraintes permettent donc de définir implicitement une sélection sur les termes où doivent être appliquées les inférences. L'efficacité de la paramodulation basique a été mise en évidence expérimentalement. En effet, elle a été le point clé dans la résolution du problème de Robbins par W. McCune avec EQP (McCune, 1997b).

La règle de paramodulation basique définie dans (Bachmair et al., 1995) est reportée ci-dessous. Elle utilise la fonction de sélection sur les littéraux S_f.

Paramodulation basique

$$\frac{(L[u] \vee D)\,[\![\varphi_1]\!] \qquad (s \approx t \vee C)\,[\![\varphi_2]\!]}{(L[t] \vee C \vee D)\,[\![u =^? s \wedge \varphi_1 \wedge \varphi_2]\!]} \text{ si :}$$

- u n'est pas une variable,
- $\sigma = mgu(u =^? s \wedge \varphi_1 \wedge \varphi_2)$,
- $\sigma(t) \not\succeq \sigma(s)$, $\sigma(s) \approx \sigma(t)$ est strictement maximale dans $\sigma(s) \approx \sigma(t) \vee \sigma(C)$ et $\sigma(s) \approx \sigma(t) \vee \sigma(C)$ ne contient pas d'égalité négative sélectionnée (ainsi $\sigma(s)$ est un terme sélectionné),
- $\sigma(u)$ est sélectionnée dans $\sigma(L) \vee \sigma(D)$,
- $\sigma(L) \not\preceq \sigma(s) \approx \sigma(t) \vee \sigma(D)$, et
- si $\sigma(t)$ est sélectionné et que L est un littéral négatif $u \not\approx v$, alors $\sigma(u) \approx \sigma(v) \not\succ_e \sigma(s) \approx \sigma(t)$.

Dans cette règle, la sélection permet de contrôler où ont lieu les inférences mais également d'interdire les inférences telles que la prémisse $C \vee s \approx t$ contienne des égalités négatives sélectionnées. Cette caractéristique permet d'obtenir l'effet des stratégies d'hyper-résolution et d'hyper-paramodulation (Bachmair et Ganzinger, 1994).

L'exemple suivant illustre cette règle.

Exemple 25 $$\frac{\neg P(f(x,g(y))) \vee Q(x,z) \qquad f(a,g(b)) \approx c \vee R(f(a))}{(\neg P(c) \vee Q(x,z) \vee R(f(a)))\,[\![x =^? a]\!]}$$

Le système d'inférence de paramodulation basique BP proposé dans (Bachmair et al., 1995) est composé de la *règle de paramodulation basique*, de la *règle de résolution équationnelle* (*equality resolution* ou *reflexivity resolution*) et de la *règle de factorisation* (*equality factoring*). Il a été prouvé correct et complet avec la méthode de preuve basée sur la construction de modèle décrite précédemment. Des règles de contraction ont été conçues : la *simplification standard et basique*, le *blocage standard et basique*, la *suppression standard et basique de tautologies* et la *subsomption standard et basique*. Il a été montré que la simplification standard, le blocage standard, l'élimination standard des tautologies et la subsomption standard combinées avec les règles d'expansion de la paramodulation basique ne peuvent pas former un système d'inférence complet. Par contre, les systèmes d'inférence formés à partir des règles de contraction basiques et des règles d'expansion de la paramodulation basique ont été prouvés complets. La complétude du système d'inférence de la superposition basique en présence de la simplification basique a été prouvée dans (Nieuwenhuis et Rubio, 1992a).

Paramodulation modulo

L'efficacité des prouveurs de théorèmes basés sur la paramodulation dépend également de l'amélioration d'autres facteurs. Certains axiomes peuvent être intégrés (*built-in*) dans le système d'inférence. Des exemples sont la paramodulation et la superposition modulo AC. La superposition modulo AC a été prouvée complète avec la technique de preuve des modèles et de réduction des contre-exemples dans (Bachmair et Ganzinger, 1993). La paramodulation modulo AC a été prouvée complète dans (Rusinowitch et Vigneron, 1991; Rusinowitch et Vigneron, 1995) en utilisant des arbres transfinis. Les preuves de complétude utilisent un ordre de réduction bien-fondé total sur les termes clos et AC-compatible. Le principale inconvénient de la paramodulation modulo est comme dans le cas de la complétion modulo que le nombre de clauses déduites par une inférence de paramodulation basique peut être infini. Le calcul explicite des unificateurs AC peut être évité en utilisant les contraintes. Ainsi, la paramodulation modulo AC avec contraintes a été étudiée dans (Nieuwenhuis et Rubio, 1995a; Vigneron, 1994a) et la superposition modulo AC avec contraintes dans (Nieuwenhuis et Rubio, 1994). Le principal problème de la complétion modulo avec contraintes est la mise au point de stratégies de simplification complètes en combinaison avec les règles d'expansion. La simplification basique est la seule stratégie connue complète. Elle a été développée par L. Vigneron dans le cas AC et pour les théories régulières (Vigneron, 1994a; Vigneron, 1994b).

2.5 Parallélisation des procédures de recherche de preuve et de déduction

Le calcul symbolique et en particulier la preuve automatique de théorèmes et la déduction automatique sont soumis à des explosions combinatoires. Le parallélisme semble être un moyen pour supporter des calculs intensifs et nécessitant beaucoup de mémoire et pour accélérer les temps de calcul. La parallélisation est également encouragée par la présence de réseaux de stations de travail (*workstations network*) et de machines parallèles.

La programmation parallèle est complexe.

- Le premier problème qui se pose est de développer du code parallèle. Développer du code parallèle est difficile et le déboguage l'est encore plus.
  Le modèle de calcul choisi pour réaliser une application est crucial, changer de modèle peut améliorer ou au contraire aggraver la complexité asymptotique des algorithmes.
- Le choix d'une architecture adaptée au code apparaît alors, la combinaison code-architecture étant très importante. En pratique, changer d'architecture peut affecter les performances (les accélérations) positivement ou négativement.
  On distingue les machines à mémoire distribuée (*distributed memory machines*) et les machines à mémoire partagée (*shared memory machines*). Les réseaux de stations de travail peuvent être considérés comme des machines à mémoire distribuée. On différencie également les architectures MIMD (*Multiple Instructions, Multiple Data*) des architectures SIMD (*Single Instruction Multiple Data*). Les architectures SIMD sont adaptées aux calculs dans les algèbres linéaires, mais elles ont un pouvoir limité par rapport aux architectures MIMD.

La question d'un paradigme et d'un environnement de programmation est donc d'un intérêt incontestable pour développer des applications parallèles. Les tâches parallèles pour atteindre un objectif commun en recherche de preuve comme en déduction doivent s'échanger des données et parfois se synchroniser. Un environnement de programmation parallèle idéal serait d'avoir un système d'exploitation parallèle (Tanenbaum, 1992; Tanenbaum, 1995). Ce système d'exploitation

n'existant pas, divers environnements ont été mis en place. Dans le cas des machines parallèles à mémoire distribuée, les principaux problèmes sont d'affecter des tâches à des processeurs pour équilibrer les charges de travail des processeurs et d'avoir des primitives de communication efficaces. Les systèmes PVM (Sunderam, 1990; Beguelin, Dongarra et Geist, 1994) et MPI (Gropp, Lusk et Skjellum, 1994; Snir, Otto, Huss-Lederman, Walker et Dongarra, 1996) permettent d'affecter des tâches à des processeurs, d'échanger des messages et d'exprimer des synchronisations. Dans le cas des machines à mémoire partagée, les principaux problèmes sont de faire coïncider efficacement les tâches avec les processeurs et l'architecture et d'intégrer le parallélisme dans l'environnement de programmation et le système d'exploitation. Les systèmes d'exploitation des machines parallèles à mémoire partagée fournissent un support pour la programmation parallèle à travers des *threads* de contrôle. Le système d'exploitation POSIX, par exemple, permet d'utiliser des *threads*. Un *thread* est un contexte d'exécution pour une procédure, au même titre qu'un processus est un contexte d'exécution pour un programme. Les *threads* exécutent donc des tâches à un niveau de grain fin.

La parallélisation des procédures de recherche de preuve automatique de théorèmes et de déduction automatique ajoute aux problèmes propres du parallélisme d'autres difficultés à prendre en compte. Les procédures de recherche de preuve automatique et de déduction automatique font appel à un haut niveau d'abstraction et sont donc par essence complexes. Les données sont fortement dépendantes et l'espace de recherche est peu structuré et peut être infini. L'efficacité de la recherche de preuve ou de la déduction dépend de plus des stratégies de recherche utilisées.

Néanmoins, les procédures de recherche de preuve et de déduction ont un fort potentiel de parallélisation. Les systèmes d'inférences sont naturellement parallèles à cause de l'indéterminisme dans l'application des règles d'inférence. Le parallélisme semble être le moyen adéquate pour augmenter le pouvoir de calcul et pallier aux problèmes de mémoire dans le but de résoudre un plus grand nombre de problèmes.

Les principales approches de la parallélisation de la recherche de preuve et de la déduction automatique sont répertoriées dans (Suttner et Schumann, 1993) et (Bonacina et Hsiang, 1994; Bonacina, 1999b). Nous présentons dans cette section différentes classification des procédures de recherche de preuve et de déduction. Du point de vue des implantations, on peut distinguer les systèmes par rapport aux architectures les plus adaptées, par rapport au type de communication utilisée (synchrone ou asynchrone)... On parle également de parallélisme ET et OU basé sur la notion d'arbres ET-OU empruntée à la logique. Cette classification n'est pas adaptée au système de recherche de preuve et de déduction car elle est ambiguë et toutes les opérations des systèmes de recherche de preuve et de déduction ne peuvent pas être décrites par l'utilisation d'arbres ET-OU. Nous nous intéressons particulièrement dans la suite à la classification de C.B. Suttner et J. Schumann (Suttner et Schumann, 1993) et à celle de M.P. Bonacina et J. Hsiang (Bonacina et Hsiang, 1994). La classification de M.P. Bonacina et J. Hsiang est dédiée au grain du parallélisme : grain fin, grain moyen et gros grain. La classification de C.B. Suttner et J. Schumann s'intéresse à la relation entre l'espace de recherche et les agents travaillant en parallèle. C.B. Suttner et J. Schumann se placent dans le cas d'un gros grain de parallélisme et distinguent deux types de parallélisme. La première approche considère la parallélisation basée sur le *partitionnement* de l'espace de recherche entre différents agents. Cette approche est la plus traditionnelle. La deuxième approche considère le cas où les différents agents travaillent sur le même problème mais utilisent des stratégies différentes ou des systèmes de recherche de preuve et de déduction différents. Les auteurs utilisent alors le terme de *parallélisme de compétition*. Des références à des systèmes de recherche de preuve et de déduction sont données au fil de cette section.

2.5.1 Grain du parallélisme

Dans (Bonacina, 1992; Bonacina et Hsiang, 1994), trois types de parallélisme en recherche de preuve et en déduction sont proposés, le parallélisme à grain fin (au niveau des termes), le parallélisme à grain moyen (au niveau des clauses) et le parallélisme à gros grain (au niveau de l'espace de recherche). Pour chaque type de parallélisme, nous distinguons le grain des données et le grain des opérations exécutées par des processus. Le grain des données détermine le grain de mémoire nécessaire et le grain des opérations détermine les types de tâches exécutées par des processus.

Le parallélisme à grain fin Le terme est distribué aux processus, chaque processus représentant un sous-terme. Un processus ne peut réaliser qu'une sous-tâche d'une inférence. Les processus concurrents doivent communiquer pour réaliser une inférence. Les communications entre processus sont donc très importantes.

Le filtrage parallèle, l'unification parallèle et la réécriture parallèle (Lindenstrauss, 1989; Kirchner et Viry, 1992; Alouini, 1997) sont des exemples de parallélisme à grain fin.

Nous présentons notre contribution dans le chapitre 3. Elle consiste dans le développement d'une procédure de complétion close concurrente de grain fin basée sur l'utilisation des graphes SOUR (Kirchner et al., 1996a). Le système implanté s'appelle *CWD* (*Completion Without Duplication*).

Le parallélisme à grain moyen L'ensemble des clauses est partitionné de telle sorte qu'une clause soit distribuée à chaque processus concurrent. Chaque processus peut réaliser une inférence.

Le parallélisme à gros grain Les processus concurrents sont asynchrones et cherchent en parallèle une solution (une preuve, par exemple). Quand l'un des processus a trouvé une solution, le système s'arrête avec succès sur cette solution.

L'espace de recherche est partitionné (ou non) entre les processus, chaque processus recevant une partie (ou toutes les clauses) de cet espace. Chaque processus développe sa dérivation propre et communique avec les autres processus.

Le choix de l'architecture doit être lié au grain du parallélisme de l'application. Ainsi, les parallélismes à grain fin et moyen sont plus adaptés aux architectures à mémoire partagée, mais des applications à parallélisme à gros grain peuvent aussi bien être implantées sur des architectures à mémoire partagée que distribuée.

Le grain du parallélisme est également associé aux problèmes des *conflits* entre processus concurrents. Deux processus sont en *conflit* s'ils accèdent à la même donnée en « même temps ».

Si l'on considère du parallélisme à grain moyen donc au niveau des clauses, les conflits schématisent des dépendances entre les inférences. Pour des inférences d'expansion dont les prémisses sont des ensembles disjoints, il n'y a pas de conflits entre les processus concurrents. Deux processus peuvent être amenés à lire la même clause mais deux lectures du même objet ne peuvent pas être considérées comme un conflit. Dans le cas d'inférences de contraction, trois types de conflits émergent :

Des conflits écriture-écriture-contraction Deux inférences de contraction contractent la même clause.

Ceci constitue un conflit dans le sens où si l'une des contractions est réalisée, l'autre n'a pas lieu d'être.

Des conflits lecture-écriture-contraction Une inférence de contraction contracte une clause φ qui est utilisée dans une autre inférence de contraction pour contracter une clause ψ.

Ce cas ne constitue pas un réel conflit, car la contraction de ψ par φ ne pose pas de problème même si φ n'existe plus.

Des conflits lecture-écriture-contraction-expansion Une inférence de contraction contracte une clause qui est utilisée dans une inférence d'expansion.

Ce cas de conflits est le plus critique. Deux cas se produisent. Si l'inférence d'expansion est réalisée, elle peut générer une clause redondante et si l'on suspend l'inférence d'expansion jusqu'à ce que ses prémisses soient totalement contractées, un délai important peut être introduit.

Les stratégies de contraction impliquent donc des conflits. En effet, les dépendances entre les inférence sont plus importantes pour les stratégies de contraction que pour les stratégies d'expansion ou les stratégies de réduction de sous-buts. L'espace de recherche dans ce cas est également plus dynamique dans le sens où des clauses sont ajoutées à l'espace de recherche ou enlevées de l'espace de recherche. Dans le cas de stratégies d'expansion, l'espace de recherche croît seulement monotonement par l'ajout de clauses. Mais, il est indéniable que malgré les difficultés de parallélisation, les stratégies de contraction sont les plus intéressantes dans la pratique.

Régler le problème des conflits est lié au grain du parallélisme. Pour régler certains conflits pour du parallélisme à grain moyen, l'une des approches est de passer d'un grain moyen à un grain fin. Les accès au niveau des termes peuvent alors ne pas être des conflits. Ce choix est bien sûr coûteux en communication. L'autre approche est de passer à un gros grain de parallélisme. Il n'y a pas de conflits dans le cas d'un gros grain de parallélisme, car chaque processus travaille sur un ensemble de clauses qui lui est propre. Les conflits du parallélisme à grain moyen peuvent être également réglés en introduisant des clauses dupliquées, ce qui est implicite dans le cas du parallélisme à gros grain.

A la vue de ces résultats, le grain de parallélisme apparaît donc comme un choix crucial à faire lors du développement de son modèle de calcul parallèle. Il est important de se poser la question également par rapport à la stratégie utilisée. Les stratégies de réduction de sous-buts peuvent être combinées aux trois types de parallélisme. Les stratégies d'expansion peuvent exploiter le parallélisme à grain moyen et le parallélisme à gros grain. Les stratégies de contraction sont quant à elles plus adaptées au parallélisme à gros grain, car la dépendance entre les données et la dynamicité de l'espace de recherche sont importantes pour ce type de stratégie.

Les tableaux 2.1 et 2.2 présentent la classification de quelques systèmes de recherche de preuve et de déduction en fonction de leur grain de parallélisme et de la stratégie de recherche ou de déduction utilisée.

La tendance actuelle est d'utiliser la compétition voire la coopération de systèmes existants, compétition entre différentes instances du même système, chacun utilisant une stratégie de recherche de preuve ou de déduction différentes ou de façon plus originale, compétition entre des systèmes hétérogènes. Le terme de coopération souligne le fait que les agents s'échangent des informations. Il existe des systèmes où il n'y a pas de coopération entre les agents. Les systèmes non coopératifs sont en général des systèmes basés sur des stratégies de réduction de sous-buts. Les systèmes non coopératifs sont faciles à implanter puisqu'il s'agit seulement de lancer différentes opérations en parallèle. PARTHENON, PARTHEO et METEOR sont des exemples de systèmes non coopératifs. La classification de (Suttner et Schumann, 1993) prend en compte cette idée de compétition. Nous nous intéressons plus particulièrement dans la section suivante aux systèmes coopératifs.

2.5.2 Coopération de systèmes de recherche de preuve ou de déduction

La classification de C.B. Suttner et J. Schumann concerne le parallélisme à gros grain et distingue le *parallélisme de partitionnement* du *parallélisme de compétition.*

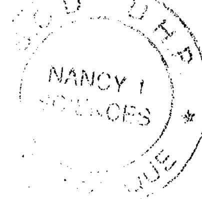
L'idée du *parallélisme de partitionnement* est de distribuer (partitionner) l'espace de recherche à différents agents qui travaillent en parallèle. Dans cette classe, on parle de *parallélisme basé complétude*, si une solution trouvée par un agent est une solution pour tout le système et de *parallélisme basé correction*, si la solution globale doit être construite à partir des solutions partielles des différents agents. La plupart des systèmes sont basés complétude plutôt que correction. Les systèmes cités dans ce document sont basés complétude.

Il existe un nombre de prouveurs de théorèmes ou de systèmes de déduction importants dans la littérature, chacun d'eux a ses faiblesses, ses forces et son domaine de compétences. Cette constatation est à l'origine du *parallélisme de compétition.* Un système basé sur du *parallélisme de compétition* peut être vu comme un système multi-agents ou un réseau, où les agents sont des prouveurs de théorèmes ou des systèmes de déduction existants. Chaque agent reçoit le même problème, l'espace de recherche n'est pas partitionné. Le succès d'un agent implique le succès de tout le système et implique la terminaison du système parallèle. On distingue le *parallélisme de compétition* dans un *réseau homogène* du *parallélisme de compétition* dans un *réseau hétérogène.*

Dans un réseau homogène, chaque agent est une incarnation du même système, mais chaque agent utilise une stratégie différente. La méthode TEAMWORK développée par J. Denzinger et M. Fuchs (Denzinger, 1993; Denzinger, 1994; Denzinger, 1995; Denzinger et Dahn, 1998) considère ce cas de parallélisme. Cette méthode est implantée dans DISCOUNT (Avenhaus et al., 1995).

Dans un réseau hétérogène, les agents sont des prouveurs ou des systèmes de déduction différents. TECHS (*TEams for Cooperative Heterogeneous Search*) (Denzinger et Dahn, 1998; Fuchs et Denzinger, 1998) est une méthode mise au point par J. Denzinger, M. Fuchs et I. Dahn pour prendre en compte un réseau hétérogènes de systèmes. La difficulté est dans ce cas de savoir quelles informations communiquées, car les systèmes n'utilisent pas forcément le même formalisme, la même logique. L'un des systèmes peut utiliser des clauses alors que l'autre système n'utilise que des égalités. La méthode TEAMWORK a été étendue pour prendre en compte ces difficultés.

Le tableau 2.3 présente la classification de quelques systèmes de recherche de preuve et de déduction en fonction du parallélisme utilisé.



Grain fin	Grain moyen	Gros grain
- (Bündgen, Göbel et Küchlin, 1994) - CWD (Kirchner et al., 1996a) (voir aussi le chapitre 3) - PaRedux (Bündgen, Göbel et Küchlin, 1995)	- ROO (Lusk et McCune, 1992) - (Yelick et Garland, 1992)	- DARES (Conry, MacIntosh et Meyer, 1990) - PARTHEO (Letz et Schumann, 1990) - La méthode TEAMWORK (Denzinger, 1993; Denzinger, 1994; Denzinger, 1995; Denzinger et Dahn, 1998) et son implantation dans DISCOUNT (Avenhaus, Denzinger et Fuchs, 1995) - TECHS (Denzinger et Dahn, 1998; Fuchs et Denzinger, 1998) - La méthode de diffusion de clauses (Bonacina et Hsiang, 1995a) et son implantation dans AQUARIUS (Bonacina et Hsiang, 1995b) et dans Peers (Bonacina et McCune, 1994) - La méthode de diffusion de clauses modifiée (Bonacina, 1996) et son implantation dans Peers-mcd (Bonacina, 1997a; Bonacina, 1997b) - METEOR (Astrachan et Loveland, 1991)

Table 2.1 – *Grain de parallélisme de quelques systèmes de preuve automatique de théorèmes et de déduction automatique*

Stratégie de réduction de sous-buts	Stratégie d'expansion	Stratégie de contraction
PARTHEO PARTHENON (Bose, Clarke, Long et Michaylov, 1992)	DARES	ROO DISCOUNT AQUARIUS Peers Peers-mcd TECHS

Table 2.2 – *Stratégie de recherche de preuve et de déduction employée dans quelques systèmes de preuve automatique de théorèmes et de déduction automatique*

Parallélisme de partitionnement	Parallélisme compétition
PARTHEO	DISCOUNT
PARTHENON	TECHS
METEOR	
DARES	
Peers	
Peers-mcd	
AQUARIUS	

Table 2.3 – *Parallélisme de partitionnement et compétitif de quelques systèmes de preuve automatique de théorèmes et de déduction automatique*

Chapitre 3

Une procédure de complétion close concurrente

Sommaire

3.1 Introduction **80**
3.2 Complétion basique des graphes SOUR **81**
3.3 Principes de la complétion SOUR concurrente **90**
3.3.1 Les phases de la complétion SOUR concurrente 90
3.3.2 Phase d'initialisation 91
3.3.3 Phase de complétion 92
3.3.4 Algorithmes 95
3.4 Implantation des règles d'inférence de la complétion SOUR close concurrente **98**
3.4.1 Détection des configurations 98
3.4.2 Traitement des configurations 102
3.5 Calcul concurrent de l'unification **104**
3.5.1 Principes généraux du calcul concurrent de l'unification 104
3.5.2 U-transitions 107
3.6 Calcul concurrent de l'orientation **112**
3.6.1 Arcs d'orientation 112
3.6.2 Implantation graphique de l'ordre LPO 112
3.6.3 Principes généraux du calcul concurrent de l'orientation 114
3.6.4 O-transitions 118
3.7 Le datage de l'information **133**
3.7.1 Utilité des étiquettes de temps 133
3.7.2 Définition et utilisation des étiquettes de temps 134
3.7.3 Sémantique des graphes SOUR clos avec étiquettes de temps 136
3.8 Terminaison de l'algorithme distribué **138**
3.9 Résultats de correction et complétude **140**
3.9.1 Résultats de correction 141
3.9.2 Résultats de complétude 145
3.10 Expérience pratique : l'implantation CWD **147**
3.11 Conclusion **149**

3.1 Introduction

Nous présentons dans ce chapitre une nouvelle approche concurrente de la complétion (close) qui provient de la complétion (close) des graphes SOUR (Lynch et Strogova, 1998) (voir également la section 3.2).

Les graphes SOUR capturent de bonnes propriétés pour envisager une approche concurrente de la complétion des graphes SOUR. En effet, dans la complétion des graphes SOUR, il n'y a pas de duplication du travail puisqu'il n'y a pas de copie des termes dans le graphe, car les égalités sont représentées par un DAG. Toutes les opérations sur le graphes sont locales et sans vérification de consistance. La représentation explicite des relations de sous-terme, de réécriture, d'unification et d'orientation nous permet de voir les règles d'inférence qui sont des transformations de graphe comme des règles de transitions d'un état du système (le graphe) dans un autre état (un autre graphe). Elle nous permet donc de voir la complétion SOUR comme le résultat d'opérations indépendantes asynchrones. Le parallélisme s'impose donc dans ce sens.

Les principes généraux de notre approche sont les suivants. Nous considérons chaque noeud du graphe SOUR initial représentant l'ensemble des égalités à compléter comme un processus fils représentant un terme. Le nombre des processus est fixe, il est égal au nombre de sommets du graphe SOUR initial. Les arcs sont des canaux de communication entre processus, qui schématisent des relations de sous-terme (arc S), de réécriture (arc R), d'unification (arc U) et d'orientation (arc O). Les processus fils sont lancés par un processus appelé le processus maître, qui leur envoie également de l'information concernant le noeud qu'ils représentent sur le graphe SOUR initial. Le processus maître n'intervient plus ensuite lorsque les processus fils réalisent la complétion. Chaque processus est en charge de détecter des configurations d'arcs dans le graphe SOUR pour réaliser la complétion. Le traitement de ces configurations correspond à une transformation du graphe SOUR, c'est-à-dire à l'addition et à la suppression (dans le cas clos) d'un arc de sous-terme ou de réécriture. Le traitement des configurations dépend de calculs d'unification et d'orientation. Des *contraintes équationnelle* et des *contraintes d'ordre* ont été introduites. La résolution d'une contrainte équationnelle consiste à savoir si la contrainte est égale à *True* ou *False*, car nous nous plaçons dans le cas clos dans ce chapitre. La résolution d'une contrainte d'ordre consiste à savoir si un terme est plus grand, plus petit ou égal à un terme en utilisant l'ordre lexicographique sur les chemins *LPO*. On parle alors du calcul des contraintes.

En comparant notre méthode avec les autres méthodes de recherche de preuve et de déduction utilisant le parallélisme, notre méthode apparaît comme originale. Elle est de grain fin à la différence de la plupart des systèmes de la littérature qui sont plutôt basés sur un gros grain de parallélisme (Suttner et Schumann, 1993; Bonacina et Hsiang, 1994). Nous approche est également différente de l'approche de grain fin de (Bündgen et al., 1994), où le grain fin provient de l'utilisation de *threads*. Il n'y a pas besoin de mémoire globale ni de contrôle global comme c'est le cas dans la plupart des approches de la littérature. De plus, nous utilisons une stratégie de contraction dans notre approche et ce point est un avantage indéniable de la méthode (Bonacina et Hsiang, 1994).

Dans notre algorithme distribué, vu le grain fin de parallélisme, la complétion est réalisée par coopération entre les processus par passage de messages (*message passing*). Un processus ne réagit qu'en réponse à un message indépendamment des autres processus. Les processus sont donc complètement asynchrones. Pour assurer la correction de la complétion SOUR close concurrente, comme toutes les opérations sont réalisées de façon complètement asynchrones, nous avons besoin de prendre en compte seulement l'information nouvelle arrivant d'un processus donné. Ceci est mis en oeuvre par l'utilisation du *datage de l'information* en ajoutant à chaque message une *étiquette de temps* schématisant une information de temps local du processus qui envoie le mes-

sage. La consistance globale du graphe SOUR est ainsi assurée. La complétude de la complétion SOUR close concurrente est également établie. La complétion étant réalisée de façon complètement asynchrone, la terminaison du programme est difficile à déterminer, puisqu'il faut être sûr que tous les processus sont inoccupés (*idle*) et que tous les messages envoyés ont été reçus. Nous proposons un algorithme centralisé par un processus (par le processus maître) pour détecter la terminaison du programme concurrent.

Notre approche est prometteuse contrairement aux tentatives de parallélisation de grain fin de prolog. En effet, elle s'appuie sur un concept très simple: les graphes SOUR et le nombre de processus est fixé au départ. Aucune synchronisation n'est nécessaire et chaque processus a une quantité de travail suffisante: un processus doit détecter des configurations, les traiter et il a aussi à effectuer des calculs d'unification et d'orientation. Nous avons validé notre approche en l'implantant en *C++* et *PVM* (Sunderam, 1990; Beguelin et al., 1994) dans *CWD* (*Completion Without Duplication*).

Les travaux présentés dans ce chapitre ont été publiés dans (Kirchner et al., 1996a; Kirchner, Lynch et Scharff, 1996b).

3.2 Complétion basique des graphes SOUR

Si l'on considère l'application d'une règle d'inférence de la complétion basique, toute égalité conclusion déduite est formée à partir de la structure et des contraintes de ses prémisses. Pour répondre à ce constat, les graphes SOUR (Lynch et Strogova, 1998) se proposent de représenter un état de la complétion sous forme d'un graphe acyclique orienté (*DAG*). Les sommets de ce graphe représentent des termes et les arcs différentes relations entre ces termes. Lorsqu'une inférence est réalisée, le nouvel état de complétion est déterminé à partir de l'ancien état en ajoutant un sommet et des relations entres les sommets du *DAG* schématisées par des arcs.

Les arcs de ce graphe sont de différents types. Les *arcs de sous-terme* vont d'un sommet représentant un terme vers chacun de ses sous-termes. Les *arcs de réécriture* sont entre les sommets des termes représentant les membres droit et gauche d'une égalité. Un *arc d'unification* entre deux sommets schématise un problème d'unification entre les termes représentés par ces sommets. Un *arc d'orientation* va d'un sommet représentant un terme à un sommet représentant un terme plus petit par rapport à $\succ_t$. Les arcs sont étiquetés par des substitutions de renommage et des contraintes. Le nom de graphe SOUR provient du fait qu'un graphe ainsi défini représente explicitement les relations de Sous-termes, Orientation, Unification et Réécriture. Les arcs de sous-termes sont appelés des arcs S, les arcs d'orientation sont appelés des arcs O, les arcs d'unification sont appelés des arcs U et les arcs de réécriture sont appelés des arcs R.

La complétion basique basée sur l'utilisation des graphes SOUR est appelée *Complétion Basique des Graphes SOUR* ou *Complétion SOUR*. Les inférences de la complétion basique des graphes SOUR sont effectuées par des transformations de graphe. Des motifs particuliers (*patterns*) sont recherchés sur le graphe SOUR. Chaque motif cause l'ajout d'un arc et, éventuellement, s'il s'agit d'une contraction, la suppression d'un arc sur le graphe SOUR considéré. Ces motifs particuliers sont aussi appelés des *demi-configurations* ou des *configurations* (voir les définitions 15 et 16).

En suivant les travaux de (Lynch et Strogova, 1998), nous donnons dans cette section la syntaxe et la sémantique des graphes SOUR, puis les transformations de graphes qui déterminent les règles d'inférence de la complétion basique. Nous exposons en particulier les transformations de graphes dans le cas de la complétion d'un ensemble d'égalités closes.

Graphe SOUR initial Le *graphe SOUR de départ* ou *graphe SOUR initial* est construit à partir de l'ensemble d'égalités E non contraintes que l'on désire compléter. Pour expliquer comment est construit le graphe SOUR de départ, nous définissons une fonction Sub telle que $Sub(t)$ soit l'ensemble des sous-termes d'un terme t, $Sub(s \approx t)$ est l'ensemble des sous-termes d'une égalité $s \approx t$ et $Sub(E)$ est l'ensemble des sous-termes de l'ensemble E. $Sub(t) = \{t\} \cup \bigcup_{1 \leq i \leq k} Sub(t_i)$ si $t = f(t_1, \ldots, t_k)$ et $k \geq 0$. $Sub(s \approx t) = Sub(s) \cup Sub(t)$. $Sub(E) = \bigcup_{e \in E} Sub(e)$. Le graphe SOUR de départ pour E est le graphe ayant un sommet étiqueté par $top(t)$ pour chaque élément $t \in Sub(E)$. Pour tout terme $f(t_1, \ldots, t_n)$, un arc S étiqueté par l'*index* i est ajouté du sommet représentant le terme $f(t_1, \ldots, t_k)$ aux sommets représentant les termes t_i pour $i \in \{1, \ldots, k\}$. Pour toute égalité $s \approx t \in E$, on ajoute un arc R entre le sommet représentant le terme s et le sommet représentant le terme t. Un arc U est ajouté entre des sommets étiquetés par le même symbole.

La figure 3.1 illustre la construction du graphe SOUR de départ pour $E = \{f(g(a), g(a)) \approx h(g(a))\}$ avec la précédence $f \succ_p g \succ_p h \succ_p a$.

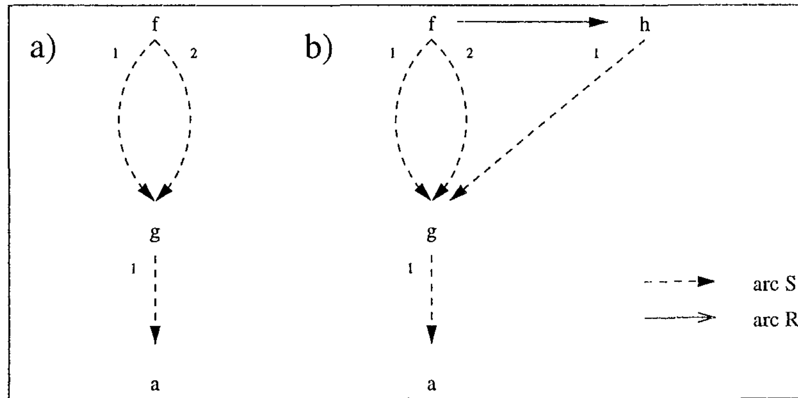


FIG. 3.1 – *Etapes de la construction du graphe SOUR de départ de* $E = \{f(g(a),g(a)) \approx h(g(a))\}$

Syntaxe des graphes SOUR Les sommets d'un graphe SOUR sont étiquetés par des constantes, des variables ou des symboles de fonctions. Nous notons $symbol(v) = f$, si f est l'étiquette du sommet v.

Les arcs S et R sont définis comme suit.

- Tout arc S e_s est étiqueté par :
  - une contrainte équationnelle notée $Constraint(e_s)$,
  - une substitution de renommage notée $Ren(e_s)$, et
  - un index noté $Ind(e_s)$.

  Les substitutions de renommage rendent explicites les renommages des égalités lors de la réalisation d'une inférence.

  S'il existe un arc S d'un sommet v_1 à un sommet v_2, on dit que v_2 est un *S-fils* de v_1 et que v_1 est un *S-père* de v_2.
- Tout arc R e_r est étiqueté par :
  - une contrainte équationnelle notée $Constraint(e_r)$ et
  - deux substitutions de renommage notées $Ren_r(e_r)$ et $Ren_l(e_r)$.

On note E_S, E_O, E_U et E_R l'ensemble des arcs S, O, U et R d'un graphe SOUR.

Sémantique des graphes SOUR La sémantique des graphes SOUR est explicitée à partir de la sémantique des sommets et des arcs S et R du graphe. Les arcs d'unification schématisent des problèmes d'unification et les arcs d'orientations sont utilisés pour orienter les arcs de réécriture.

- La sémantique d'un sommet v est l'ensemble des termes $Terms(v)$ qu'il représente. Un terme particulier de $Terms(v)$ est noté $Term(v)$.
  Soient v_f un sommet étiqueté par un symbole de fonction f d'arité n et e_i ses arcs S de v_f à v_i pour $i \in \{1,\dots,n\}$. Pour tout i, on pose $Constraint(e_i) = c_i$, $Ren(e_i) = \eta_i$, $Term(v_i) = t_i \llbracket \psi_i \rrbracket$ et τ_i est une nouvelle substitution de renommage.
  Alors $Terms(v_f) = f(\tau_1(\eta_1(t_1)),\dots,\tau_n(\eta_n(t_n))) \llbracket \bigwedge_{\{1\leq i\leq n\}} \tau_i(\tau_i(c_i) \wedge \eta_i(\psi_i)) \rrbracket$.
- Tout arc de réécriture e_r représente un ensemble d'égalités $Egalites(e_r)$.
  Soit e_r un arc de réécriture d'un sommet v_1 à un sommet v_2. Soit $t_1 \llbracket \psi_1 \rrbracket \in Terms(v_1)$ et $t_2 \llbracket \psi_2 \rrbracket \in Terms(v_2)$, $Constraint(e_r) = c$, $Ren_l(e_r) = \eta_1$ et $Ren_r(e_r) = \eta_2$.
  $\tau(\eta_1(t_1) \approx \eta_2(t_2) \llbracket \eta_1(c_1) \wedge \eta_2(c_2) \wedge c \rrbracket)$ est une égalité de $Egalites(e_r)$. Pour un graphe SOUR G, $Egalites(G) = \bigcup_{e_r \in E_R} Egalites(e_r)$.

La figure 3.2 résume la sémantique des graphes SOUR.

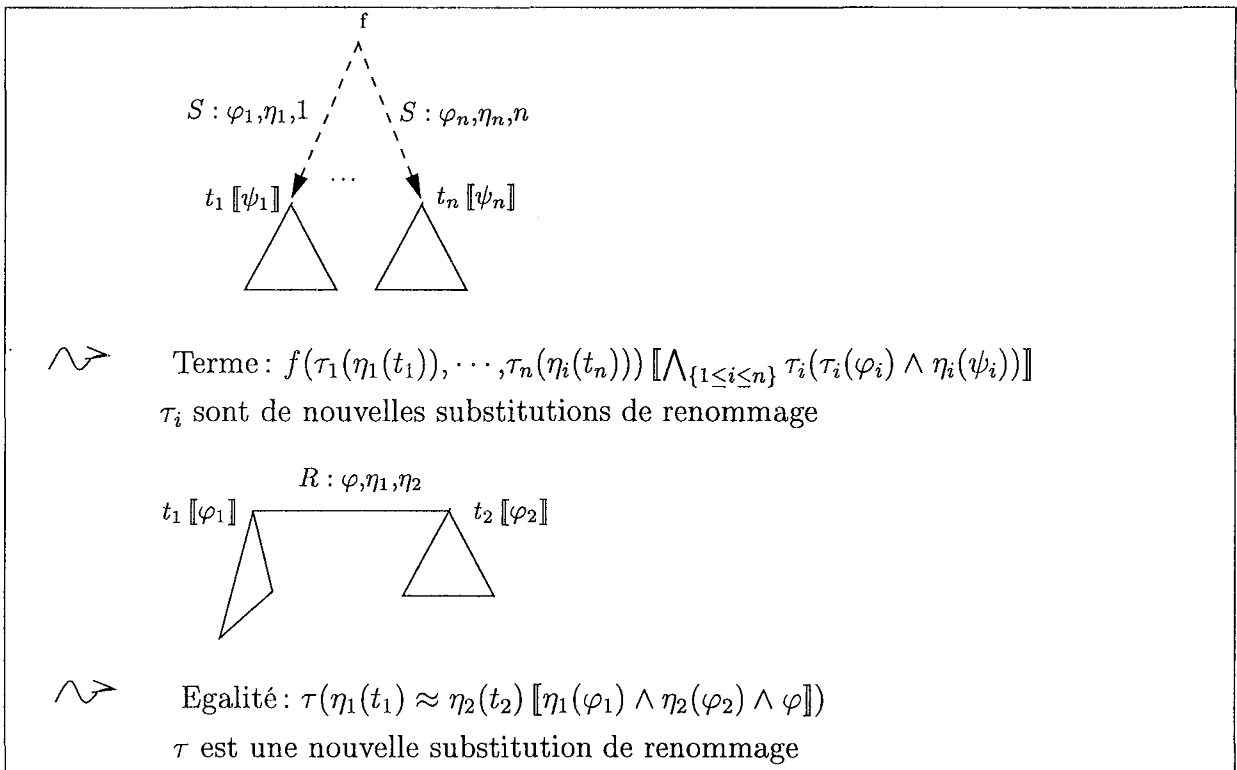


FIG. 3.2 – *Sémantique des graphes SOUR*

Considérons l'exemple suivant.

Exemple 26 *Le graphe SOUR de la figure 3.3 représente une égalité.*

Le terme représenté par le sommet étiqueté par h est : $h(\tau_1(\eta_1(z)),\tau_2(\eta_2(x))) \llbracket \tau_1(\tau_1(c_1)) \rrbracket$ *avec* τ_1 *et* τ_2 *deux nouvelles substitutions de renommage. Il s'agit donc du terme* $h(\tau_1(z_1),\tau_2(x_2)) \llbracket \tau_1(z_1) =^? a \rrbracket$.

Le terme représenté par le sommet étiqueté par a est : a.

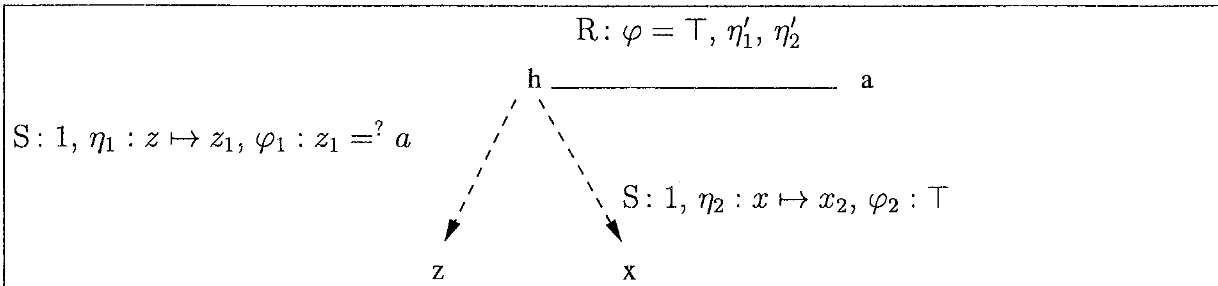


FIG. 3.3 – *Sémantique des graphes SOUR - Exemple*

L'égalité est: $\tau(h(\eta'_1(\tau_1(z_1)),\eta'_2(\tau_2(x_2))) \approx a\,[\![\eta'_1(\tau_1(z_1) =^? a) \wedge \eta'_2(c_2)]\!])$ *soit* $\tau(h(\eta'_1(\tau_1(z_1)),\eta'_2(\tau_2(x_2))) \approx a\,[\![\eta'_1(\tau_1(z_1) =^? a)]\!])$.

Transformations de graphe La complétion basique est réalisée en saturant le graphe de départ par rapport à un ensemble de transformations de graphe. Ces transformations sont basées sur la recherche des demi-configurations et des configurations sur le graphe SOUR. Nous définissons les demi-configurations et les configurations dans le cas non clos dans la définition suivante. Le cas des demi-configurations et des configurations dans le cas clos est traité dans la définition 16.

Définition 15 – *Une* demi-configuration *est un motif d'arcs composé d'un arc d'unification e_u entre des sommets v_1 et v_2 et d'un arc de réécriture e_r entre un des sommets de e_u et un sommet v_3.*
- *Une* configuration SUR *est un motif d'arcs composé d'un arc de sous-terme e_s qui va d'un sommet v_3 à un sommet v_1, d'un arc d'unification e_u entre le sommet v_1 et un sommet v_2 et d'un arc de réécriture e_r entre le sommet v_2 et un sommet v_4.*
- *Une* configuration RUR *est un motif d'arcs composé d'un arc de réécriture e_{r_1} entre un sommet v_1 et un sommet v_3, d'un arc d'unification e_u entre le sommet v_1 et un sommet v_2 et d'un arc de réécriture e_{r_2} entre le sommet v_2 et un sommet v_4.* □

1. La première transformation est appelée *transformation SUR*.
   Elle consiste à rechercher une configuration SUR sur le graphe SOUR.
   On pose $Ren(e_s) = \eta_1$, $Ren_l(e_r) = \eta_l$, $Ren_r(e_r) = \eta_r$, $Constraint(e_s) = \psi_1$, $Constraint(e_r) = \psi_2$. Supposons qu'il existe $t_1 \in Terms(v_1)$, $t_2 \in Terms(v_2)$, $t_3 \in Terms(v_3)$ et $t_4 \in Terms(v_4)$.
   Le traitement de cette configuration entraîne l'ajout d'un nouvel arc de sous-terme e_{new} de v_3 à v_4 si $\psi = \eta_1(t_1) =^? \rho(\eta_l(t_2)) \wedge \psi_1 \wedge \eta_1(c_1) \wedge \rho(\psi_2 \wedge \eta_l(c_2))$, où ρ est une nouvelle substitution de renommage, est satisfaisable.
   Ce traitement est visible sur la figure 3.4.

   Cette transformation correspond à une inférence de paire critique à une position différente de ϵ décrite ci-dessous.
   Supposons que $Symbol(v_3) = f$. D'après la sémantique d'un sommet, $f(\ldots,\tau(\eta_1(t_1)),\ldots)[\![\tau(\psi_1) \wedge \tau(\eta_1(c_1))]\!] \in Terms(v_3)$, où τ est une nouvelle substitution de renommage. On note C_{eq} une égalité du graphe contenant ce terme. C_{eq} a la forme $C[f(\ldots,\tau(\eta_1(t_1)),\ldots)][\![\eta(\tau(\psi_1)) \wedge \eta(\tau(\eta_1(c_1))) \wedge c]\!]$ où c et η sont une contrainte et une substitution de renommage apparaissant dans le graphe SOUR.

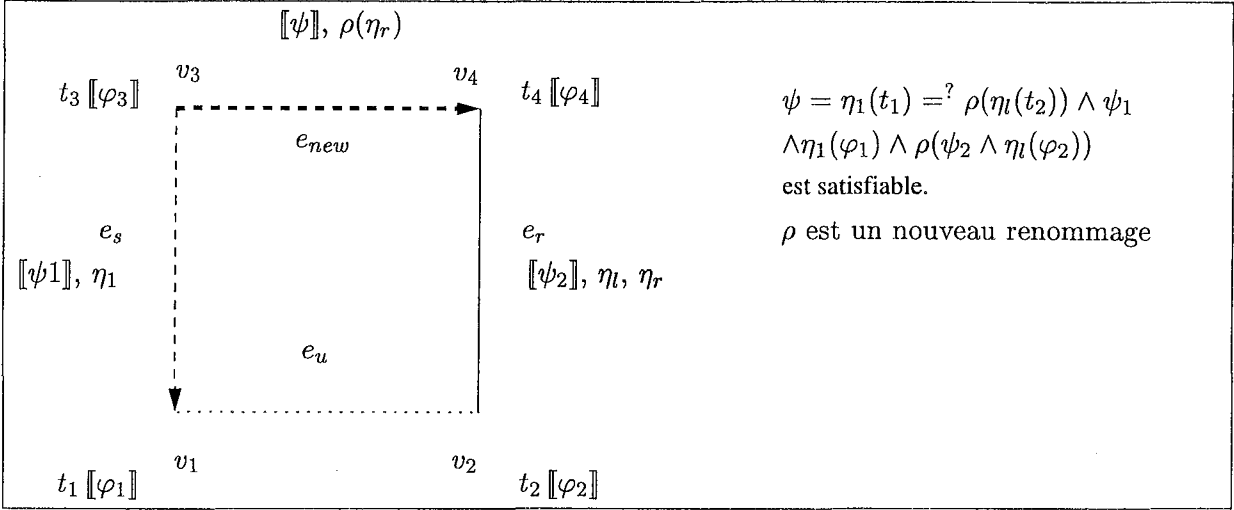


FIG. 3.4 – *Transformation SUR - cas non clos*

e_r représente l'égalité $\eta(\tau(\rho(\eta_l(t_2) \approx \eta_r(t_4) [\![\eta_l(c_2) \wedge \eta_r(c_4) \wedge \psi_2]\!])))$. $\eta\tau\rho$ est une substitution de renommage qui permet aux deux égalités impliquées dans l'inférence d'avoir des ensembles de variables disjoints.
L'inférence réalisée est la suivante.

$$\frac{C[f(\ldots,\tau(\eta_1(t_1)),\ldots)]\,[\![\eta(\tau(\psi_1)) \wedge \eta(\tau(\eta_1(c_1))) \wedge c]\!] \quad \eta(\tau(\rho(\eta_l(t_2) \approx \eta_r(t_4)\,[\![\eta_l(c_2) \wedge \eta_r(c_4) \wedge \psi_2]\!])))}{C[f(\ldots,\tau(\rho(\eta_r(t_4))),\ldots)]\,[\![\eta(\tau(c)) \wedge \eta(\tau(\eta_r(c_4))) \wedge c]\!])}$$

L'inférence doit être réalisée pour tous les termes de $Terms(v_1)$ et $Terms(v_2)$.

2. La deuxième transformation consiste à rechercher une configuration RUR sur le graphe SOUR.
On pose $Ren_l(e_{r_1}) = \eta_1$, $Ren_r(e_{r_1}) = \eta_3$, $Ren_l(e_{r_2}) = \eta_2$, $Ren_r(e_{r_2}) = \eta_4$, $Constraint(e_{r_1}) = \psi_1$. Supposons qu'il existe $Constraint(e_{r_2}) = \psi_2$. Soient $t_1 \in Terms(v_1)$, $t_2 \in Terms(v_2)$, $t_3 \in Terms(v_3)$ et $t_4 \in Terms(v_4)$.
Le traitement de cette configuration entraîne l'ajout d'un nouvel arc de sous-terme e_{new} entre v_3 et v_4 si $c = \eta_1(t_1) =^? \rho(\eta_l(t_2)) \wedge \psi_1 \wedge \eta_1(c_1) \wedge \rho(\psi_2 \wedge \eta_l(c_2))$ où ρ est une nouvelle substitution de renommage.
Ce traitement est visible sur la figure 3.5.

Cette transformation correspond à une inférence de paire critique à la position ϵ.
e_{r_1} représente l'égalité $\tau(\eta_1(t_1) \approx \eta_3(t_3)\,[\![\psi_1 \wedge \eta_1(c_1) \wedge \eta_3(c_3)]\!])$.
e_{r_2} représente l'égalité $\rho(\tau(\eta_2(t_2) \approx \eta_4(t_4)\,[\![\psi_2 \wedge \eta_2(c_2) \wedge \eta_4(c_4)]\!]))$.
L'inférence réalisée est la suivante.

$$\frac{\rho(\tau(\eta_2(t_2) \approx \eta_4(t_4)\,[\![\psi_2 \wedge \eta_2(c_2) \wedge \eta_4(c_4)]\!])) \qquad \tau(\eta_1(t_1) \approx \eta_3(t_3)\,[\![\psi_1 \wedge \eta_1(c_1) \wedge \eta_3(c_3)]\!])}{\tau(\eta_3(t_3) \approx \rho(\eta_4(t_4))\,[\![\rho(\psi) \wedge \eta_3(c_3) \wedge \rho(\eta_4(c_4))]\!])}$$

L'inférence doit être réalisée pour tous les termes de $Terms(v_1)$ et $Terms(v_3)$.

Considérons l'exemple suivant.

Exemple 27 *Considérons l'ensemble d'égalités* $E = \{f(f(x)) \to g(f(x))\}$ *et la précédence* $f \succ_p g$.

Le graphe SOUR de départ correspondant à E est visible sur la figure 3.6. On note la présence de l'arc U entre $f(x)$ *et* $f(f(x))$. $f(x)$ *et* $f(f(x))$ *sont unifiables après renommage.*

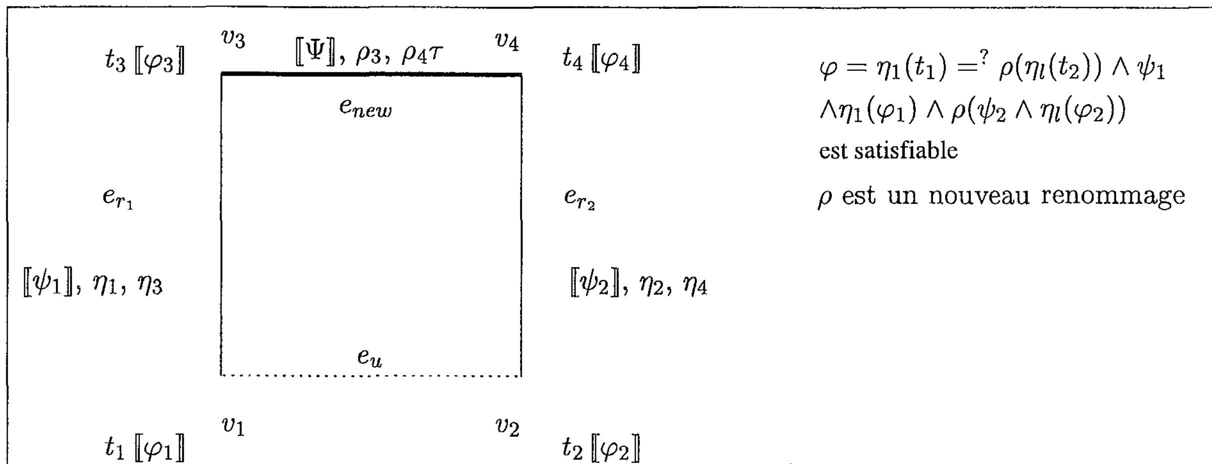


FIG. 3.5 – *Transformation RUR - cas non clos*

Il y a une configuration SUR entre les sommets représentant les termes $f(f(x))$, $f(x)$, $f(f(x))$ et $g(f(x))$. Le traitement de cette configuration entraîne l'ajout d'un arc S du sommet représentant $f(x)$ au sommet représentant $g(f(x))$. Cet arc est étiqueté par la substitution de renommage ρ_1 et par la contrainte $[\![f(f(\rho_1(x))) =^? f(x)]\!]$.

Cette transformation de graphe correspond à l'inférence :

$$\frac{f(f(\rho_1(x))) \to g(f(\rho_1(x))) \qquad f(f(x)) \to g(f(x))}{f(g(f(\rho_1(x)))) \to g(f(x))\, [\![f(f(\rho_1(x))) =^? f(x)]\!]}$$

La figure 3.6 montre comment est réalisée cette transformation.

Il y a ensuite un configuration SUR entre les sommets représentant les termes $g(f(x))$, $f(x)$, $f(f(x))$ et $g(f(x))$. Le traitement de cette configuration entraîne l'ajout d'un arc S du sommet représentant $g(f(x))$ à lui-même. Cet arc est étiqueté par la substitution de renommage ρ_2 et par la contrainte $[\![f(f(\rho_2(x))) =^? f(x)]\!]$.

$$\frac{f(g(f(\rho_1(x)))) \to g(f(x))\, [\![f(f(\rho_1(x))) =^? f(x)]\!] \qquad f(f(\tau(x))) \to g(f(\tau(x)))}{f(g(g(f(\tau(x))))) \to g(f(x))\, [\![f(f(\tau(x))) =^? f(\rho_1(x)) \wedge f(f(\rho_1(x))) =^? f(x)]\!]}$$

La figure 3.6 montre comment est réalisée cette transformation.

Correction et complétude (Lynch et Strogova, 1998) Les transformations représentent un système d'inférence correct dans le sens où si G est un graphe SOUR sur lequel on applique une transformation SUR ou RUR pour obtenir G' alors $Egalite(G) \models Egalites(G')$. Ce système d'inférence est complet dans le sens où, si eq_1 et eq_2 sont des égalités de $Egalites(G)$ telles qu'il existe une inférence les impliquant et permettant de déduire l'égalité conclusion eq_3, alors, il existe une transformation SUR ou RUR qui permet de déduite G' à partir de G et telle que $eq_3 \in Egalite(G')$.

Discussion Une limite de la complétion basique des graphes SOUR (non clos) est qu'aucune règle de contraction n'a été mise au point. Dans le cas clos par contre, toute transformation est une inférence de simplification. Le cas clos est traitée dans le paragraphe suivant.

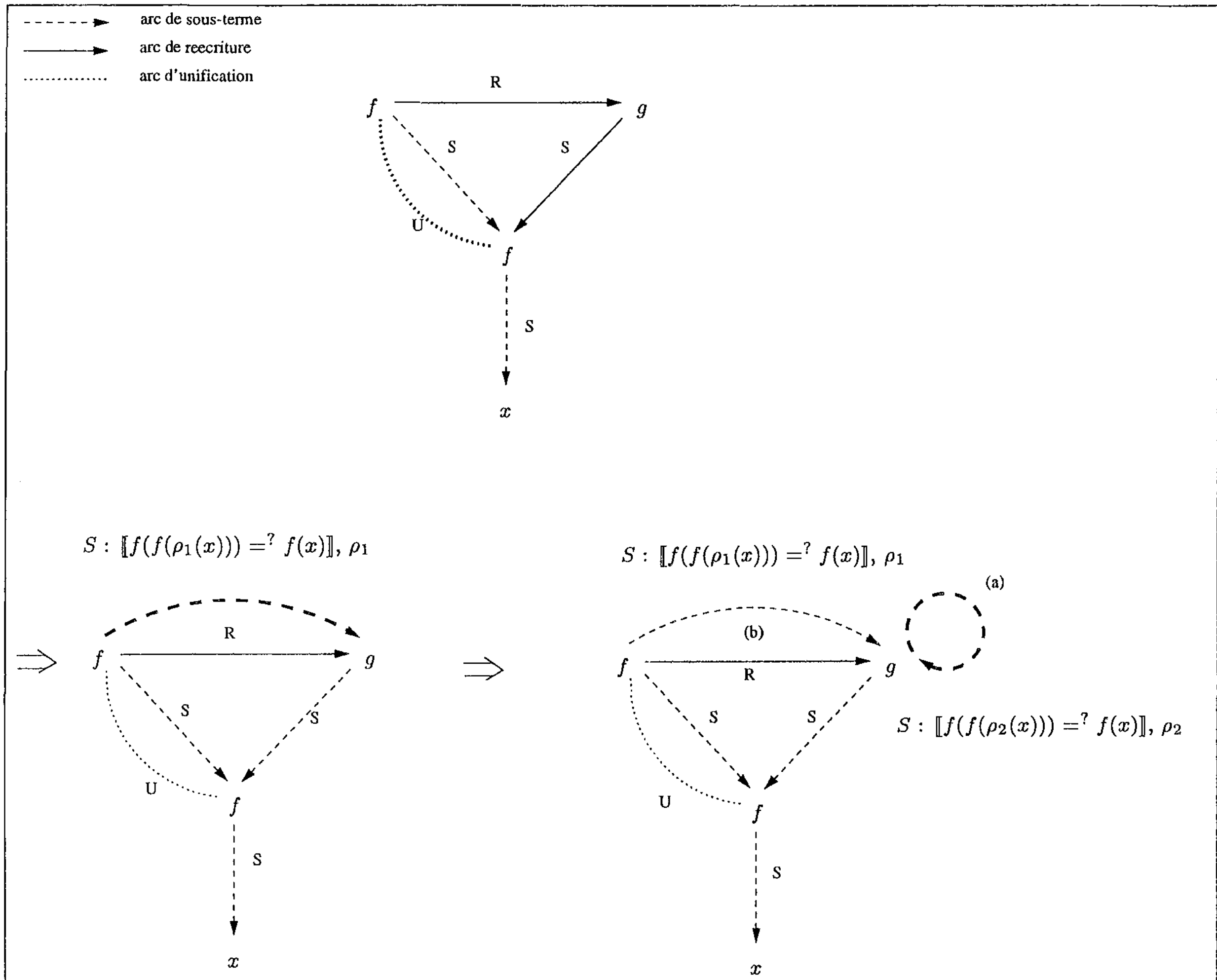


FIG. 3.6 – *Complétion de* $f(f(x)) \approx g(f(x))$

Les graphes SOUR ont été utilisés comme base pour construire d'autres procédures de complétion. Dans (Lynch, 1997), une procédure de *complétion basique orientée but* utilisant comme structure de données les graphes SOUR est présentée. Les graphes SOUR sont à l'origine de l'utilisation des graphes PATCH pour faire de la complétion dans la théorie des groupes (Lynch et Strogova, 1996). Nous présentons dans le chapitre 3 une procédure de complétion concurrente basée sur l'utilisation des graphes SOUR clos.

Le cas clos Dans le cas clos, les arcs ne sont ni étiquetés par des contraintes ni par des substitutions de renommage. Un sommet du graphe représente un unique terme. Les transformations de graphe s'expriment donc plus facilement. Toute transformation consiste à ajouter un nouvel arc et à en supprimer un ancien, la simplification étant toujours possible dans le cas clos. Les transformations sont appelées transformation SUR, transformation RUR et transformation RUR-rhs. La transformation RUR du cas non clos a été décomposée en deux transformations : la transformation RUR et la transformation RUR-rhs basées sur la recherche de configurations RUR et RUR-rhs respectivement.

La définition des demi-configurations et configurations dans le cas non clos (définitiion 15)

est spécialisée dans le cas clos comme suit.

Définition 16 – *Une* demi-configuration *est un motif d'arcs composé d'un arc d'unification e_u entre des sommets v_1 et v_2 et d'un arc de réécriture e_r d'un des sommets de e_u à un sommet v_3.*

– *Une* configuration SUR *est un motif d'arcs composé d'un arc de sous-terme e_s d'un sommet v_1 à un sommet v_2, d'un arc d'unification e_u du sommet v_2 à un sommet v_3 et d'un arc de réécriture du sommet v_3 à un sommet v_4.*

– *Une* configuration RUR *est un motif d'arcs composé d'un arc de réécriture e_{r_1} d'un sommet v_1 à un sommet v_2, d'un arc d'unification e_u du sommet v_1 à un sommet v_3 et d'un arc de réécriture e_{r_2} du sommet v_3 à un sommet v_4.*

– *Une* configuration RUR-rhs *est un motif d'arcs composé d'un arc de réécriture e_{r_1} d'un sommet v_1 à un sommet v_2, d'un arc d'unification e_u du sommet v_1 à un sommet v_3 et d'un arc de réécriture e_{r_2} du sommet v_3 à un sommet v_4.*□

Réaliser une transformation SUR consiste, comme représenté sur la figure 3.7 :

- à chercher une configuration SUR,
- à supprimer l'arc de sous-terme e_s, et
- à ajouter un arc de sous-terme de v_1 à v_4.

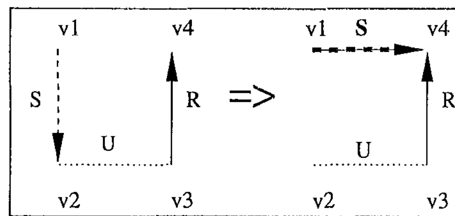


FIG. 3.7 – *Transformation SUR - cas clos*

Cette transformation correspond à l'inférence de simplification standard suivante. L'égalité $u[s]_\omega \to v$ est enlevée de l'espace de recherche.

$$\frac{u[s]_\omega \to v \qquad s \to t}{u[t]_\omega \to v}$$

Une transformation RUR consiste, comme l'indique la figure 3.8 :

- à chercher une configuration RUR,
- si $Term(v_2) \succ_t Term(v_4)$, à supprimer e_{r_1}, et à ajouter un arc de réécriture de v_2 à v_4, et
- si $Term(v_4) \succ_t Term(v_2)$, à supprimer e_{r_2}, et à ajouter un arc de réécriture de v_4 à v_2.

Cette transformation correspond à l'inférence de simplification standard suivante. $s \to t$ (respectivement $s \to t'$) est enlevée de l'espace de recherche.

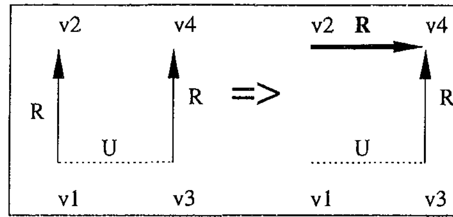


FIG. 3.8 – *Transformation RUR - cas clos*

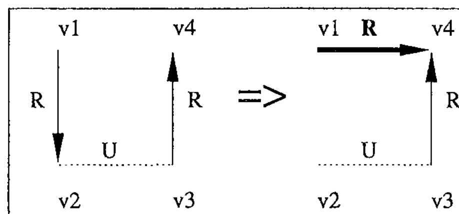
$$\frac{s \to t \quad s \to t'}{t \to t'} \text{ si } t \succ_t t'$$

ou

$$\frac{s \to t \quad s \to t'}{t' \to t} \text{ si } t' \succ_t t$$

Une transformation RUR-rhs consiste, comme le montre la figure 3.9 :

- à chercher une configuration RUR-rhs, et
- à ajouter un arc de réécriture de v_2 à v_4.



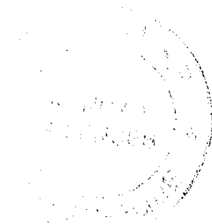


FIG. 3.9 – *Transformation RUR-rhs - cas clos*

Cette transformation correspond à l'inférence de simplification standard suivante. $s \to t$ est supprimée de l'espace de recherche.

$$\frac{s \to t \quad t \to u}{s \to u}$$

La complétion close des graphes SOUR utilise un espace polynomial. Il y a au plus $n(n+1)/2$ arcs dans un graphe SOUR de n sommets, car il n'y a jamais deux arcs de même type entre deux sommets. Elle nécessite également un temps polynomial car elle ne consiste qu'en l'ajout et en la suppression d'arcs.

Si l'on considère l'ensemble E contenant les égalités $a_o \approx f(a_1,a_1), \ldots, a_n \approx f(a_n,a_n)$, $f(a_0,a_o) \approx b$ et la précédence $a_0 \succ_p \ldots \succ_p a_n \succ_p f \succ_p b$, alors sans partage de structure, la complétion de E nécessite un espace exponentiel. Les graphes SOUR permettent le partage de structure et évite ce problème.

3.3 Principes de la complétion SOUR concurrente

3.3.1 Les phases de la complétion SOUR concurrente

L'ensemble des égalités E (closes) à compléter est représenté par un graphe SOUR, le *graphe SOUR initial*. La structure des graphes SOUR a été présentée dans la section 3.2.

L'ensemble des noeuds du graphe SOUR initial détermine le nombre de processus *fils* indépendants qui vont réaliser la complétion en utilisant les arcs comme des canaux de communication. Les processus fils sont lancés par un processus appelé le *processus maître*. Les processus sont caractérisés par un numéro d'identification appelé *tid*. On associe à chaque noeud du graphe SOUR initial un tid de processus fils.

La complétion peut être divisée en trois phases : la *phase d'initialisation*, la *phase de complétion* et la *terminaison* comme nous le décrivons ci-dessous.

Le processus maître intervient dans les trois phases de la complétion. Les processus fils interviennent uniquement dans la phase de complétion. Dans la phase d'initialisation, le processus maître lit les informations contenues sur le graphe SOUR initial et envoie à chacun des processus fils l'information locale sur le noeud qu'il représente. Les informations envoyées sont décrites en détails dans la section 3.3.2.

Dans la phase de complétion, les processus fils réalisent la complétion par coopération en s'échangeant des messages. Les processus fils envoient des messages au processus maître pour lui faire prendre connaissance des messages transitant entre les processus fils. Le processus maître rassemble l'information concernant les messages transitant entre les processus fils dans le but de détecter la terminaison de l'algorithme distribué. La terminaison de l'algorithme a lieu lorsque tous les processus sont inoccupés et que tous les messages envoyés ont été reçus. La terminaison de l'algorithme distribué correspond à la terminaison de la complétion, la complétion close des graphes SOUR terminant toujours. La terminaison est étudiée dans la section 3.8.

Les messages transitant entre les processus fils sont appelés des *messages de complétion*, les messages envoyés par le processus *maître* aux processus fils sont des *messages d'initialisation* et les messages envoyés par les processus fils aux processus *maître* sont des *messages de terminaison.* La figure 3.10 fait apparaître les différentes sortes de messages existant.

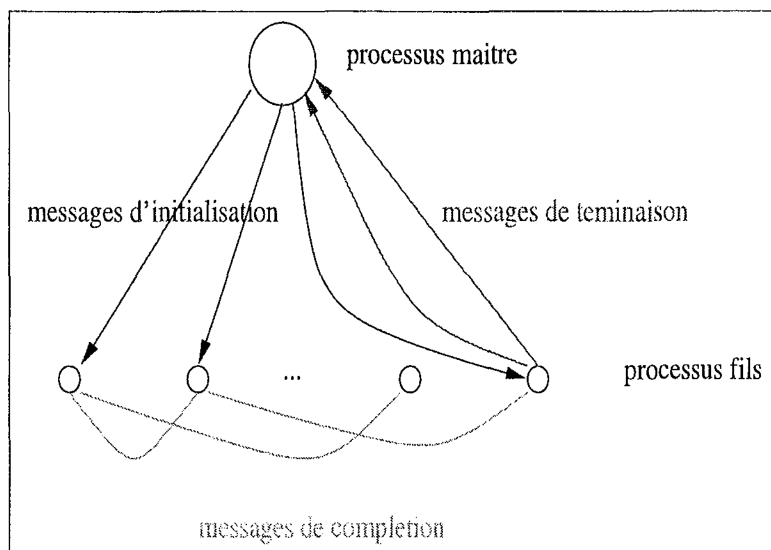


FIG. 3.10 – *Le processus maître et les processus fils*

3.3.2 Phase d'initialisation

Dans la phase d'initialisation, le processus maître lit les informations contenues sur le graphe SOUR initial et envoie à chacun des processus fils l'information locale sur le noeud qu'il représente. Le processus maître envoie donc à chaque processus fils son symbole, un dictionnaire, son nombre d'arcs de sous-terme, d'unification et de réécriture et ses arcs de sous-terme, d'unification et de réécriture.

Les envois de messages suivants sont réalisés dans la phase d'initialisation.

- Un message appelé *SYMBOL* est envoyé par le processus maître à tous les processus fils. Ce message contient le symbole du noeud représentant le processus fils.
- Le même message appelé *DICO_ ORDER_ ARITY* est envoyé par le processus maître à tous les processus fils.
  Ce message contient un dictionnaire qui à chaque symbole de fonction de l'ensemble à compléter, associe son arité et sa place dans la précédence.
- Les nombres d'arcs d'unification, de sous-terme (sortants et entrants) et de réécriture (sortants et entrants) des noeuds sont envoyés à chaque processus fils par le processus maître dans des messages appelés *INITPROCESS.*
- Des messages appelés *INITU*, *INITS* et *INITR* sont envoyés par le processus maître aux processus fils dans cet ordre pour qu'ils prennent respectivement connaissance de leurs arcs d'unification, de sous-terme et de réécriture. L'ordre est important dans la pratique (voir section 3.3.4). Un message INITS annonce l'ajout d'un arc de sous-terme dans la phase d'initialisation (comme plus loin dans la phase de complétion). Un message INITU annonce l'ajout d'un arc d'unification dans la phase d'initialisation (comme plus loin dans la phase de complétion). Un message INITR annonce l'ajout d'un arc de réécriture dans la phase d'initialisation (comme plus loin dans la phase de complétion).

  Les envois de messages *INITU*, *INITS* et *INITR* sont réalisés de la manière suivante.

  Messages INITU La même opération est réalisée pour chaque arc d'unification du graphe SOUR initial.

  Soient tid_1 et tid_2 les tids des processus fils p_1 et p_2 aux extrémités d'un arc d'unification e_u du graphe SOUR initial. La contrainte équationnelle de ces arcs d'unification est initialisée avec la valeur *False*. Un des processus à l'extrémité de l'arc d'unification e_u est désignée comme en charge du calcul de la contrainte équationnelle. Un message *INITU* est envoyé à p_1 et p_2.

  Le message *INITU* envoyé à p_1 concernant l'arc e_u contient :

  - le tid de p_2, tid_2,
  - la contrainte équationnelle de e_u initialisée avec la valeur *False*,
  - un drapeau indiquant si p_1 est en charge du calcul de la contrainte équationnelle, et
  - un drapeau indiquant si p_1 doit calculer la contrainte équationnelle. Cette notion de *nécessité* est définie dans la section 3.5.

  Le message *INITU* envoyé à p_2 concernant l'arc e_u est complémentaire au message envoyé à p_1.

  Messages INITS Le processus maître envoie un message *INITS* à chaque processus fils aux extrémités des arcs de sous-terme du graphe SOUR initial.

  Soient tid_1 et tid_2 les tids des processus fils p_1 et p_2 aux extrémités d'un arc de sous-terme e_s du graphe SOUR initial.

Le message *INITS* envoyé à p_1 concernant l'arc e_s contient :
- le tid tid_2 de p_2 et le symbole du processus fils p_2,
- un drapeau indiquant si l'arc e_s est entrant ou sortant pour p_1, et
- un index de sous-terme.

Le message *INITS* envoyé à p_2 concernant l'arc e_s est complémentaire au message envoyé à p_1.

Messages INITR Un message *INITR* est envoyé par le processus maître à chaque processus fils aux extrémités d'un arc de réécriture.

Soient tid_1 et tid_2 les tids des processus fils p_1 et p_2 aux extrémités d'un arc de réécriture e_r du graphe SOUR initial.

Le message INITR envoyé à p_1 concernant l'arc e_r contient :
- le tid tid_2 de p_2 et le symbole du processus fils p_2, et
- un drapeau indiquant si l'arc e_r est entrant ou sortant pour p_1.

Le message *INITR* envoyé à p_2 concernant l'arc e_r est complémentaire au message envoyé à p_1.

L'exemple 28 montre comment les messages d'initialisation sont envoyés.

Exemple 28 *Considérons le graphe SOUR initial de la figure 3.11 représentant l'ensemble d'égalités $E = \{f(a,a) \approx b,\ a \approx c\}$ et la précédence $f \succ_p a \succ_p b \succ_p c$.*

Le processus p_1 est représenté par le noeud de symbole f. Le processus p_2 est représenté par le noeud de symbole a. Le processus p_3 est représenté par le noeud de symbole b. Le processus p_4 est représenté par le noeud de symbole c.

Si l'on considère le processus p_1, le processus maître lui envoie les messages suivants dans la phase d'initialisation :
- *un message SYMBOL contenant f,*
- *un message DICO_ORDER_ARITY contenant l'arité et la place dans la précédence de f, a, b et c,*
- *un message INITPROCESS contenant le nombre d'arcs d'unification du noeud étiqueté par f, c'est-à-dire 0, son nombre d'arcs de sous-terme c'est-à-dire 2, et son nombre d'arcs de réécriture, c'est-à-dire 1,*
- *deux messages INITS, un par arc de sous-terme sortant, et*
- *un message INITR pour l'arc de réécriture sortant.*

3.3.3 Phase de complétion

Les processus fils sauvent l'information qu'ils ont reçue lors de la phase d'initialisation. L'information sauvée est enrichie dans la phase de complétion. En effet, la complétion est réalisée par échange de messages et grâce à la sauvegarde de l'information reçue des autres processus fils.

Définition 17 *L'ensemble de l'information sauvée par un processus est appelée son* état *et est composée des champs suivants :*
- tid *est le tid du processus fils.*
- symb *désigne le symbole du processus fils.*
- dico_order_arity *sauve le dictionnaire du processus fils qui à chaque symbole de fonction de l'ensemble à compléter, associe son arité et sa place dans la précédence.*

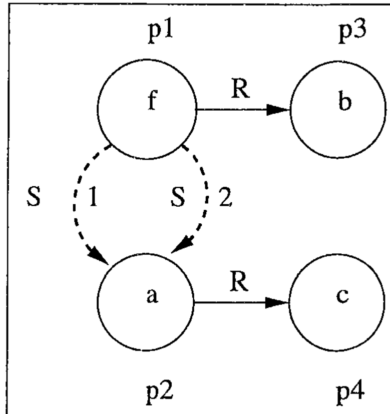


FIG. 3.11 – *Graphe SOUR initial pour* $E = \{f(a,a) \approx b,\ a \approx c\}$

- *Les arcs d'unification d'un processus fils sont sauvés dans une liste d'arcs d'unification appelée* U_list.
  Initialement, la U_list d'un processus fils contient le cycle unaire entre le noeud et lui-même, qui forme un arc d'unification avec une contrainte équationnelle de valeur True.
- *Les arcs de sous-terme entrants sont sauvés dans une liste d'arcs de sous-termes appelée* S_in_list *et les arcs de sous-terme sortants sont sauvés dans une liste d'arcs de sous-termes appelée* S_out_list.
- *De la même manière, les arcs de réécriture sont sauvés dans* R_in_list *et* R_out_list. □

Cet état sera étendu par la suite par des champs concernant les demi-configurations, des champs concernant l'unification : *list_infou*, *list_requestu* et *list_answeru* et des champs concernant l'orientation : *O_in_list*, *O_out_list*, *O_eq_list*, *calculo_list*, *requesto1_list*, *requesto2_list*, *answero1_list* et *answero2_list* qui vont être décrits dans la suite.

On distingue dans la phase de complétion les *phases élémentaires* suivantes :
- la phase de détection et de traitement des configurations notées C et P,
- la phase de calcul d'unification notée U, et
- la phase de calcul d'orientation notée O.

Chacune de ces phases fait l'objet d'une section dans ce chapitre.

Les processus fils ne travaillent que par réaction de messages ce qui permet à tout le processus de complétion d'être complètement asynchrones. Les actions effectuées par un processus à la réception d'un message sont formalisées en utilisant des règles de transitions dont la définition et la forme est donnée ci-dessous. A la réception d'un message, le processus voit son état modifié et envoie d'autres messages en réponse à son message. Notre algorithme distribué est ainsi défini par des règles de transitions.

Définition 18 *L'opérateur binaire + ajoute un élément à une liste. Il a deux arguments : une liste et un élément à ajouter à cette liste. Il est défini par :* $L + x = L$ *si* x *appartient à la liste* L*, sinon + ajoute* x *en queue de* L*. L'opérateur + peut être surchargé tel que ses deux arguments soient des listes. Dans ce cas, pour* $L + L'$*, + ajoute les éléments de la liste* L' *à la liste* L *sans répétition.*

L'opérateur binaire $-$ *enlève un élément à une liste. Il a deux arguments : une liste et un élément à enlever de cette liste. Il est défini par :* $L - x = L$ *si* x *n'appartient pas à la liste* L*, sinon* $-$ *enlève* x *de* L*. L'opérateur* $-$ *peut être surchargé tel que ses deux arguments soient des listes. Dans ce cas, pour* $L - L'$*,* $-$ *enlève tous les éléments de* L' *de* L*.* □

Définition 19 – *Une* **α-transition** *est une transformation de l'état d'un processus dans un autre état, quand il reçoit un message.*
Elle est notée : ({Mesg}, State) $\xrightarrow{\alpha}$ (Mesgset, State'), où :

- *α est une phase élémentaire de complétion, $\alpha \in \{C,D,U,O\}$.*
- *Mesg est le message reçu.*
- *State est l'état du processus avant la réception du message Mesg.*
- *Mesgset est l'ensemble de tous les messages conséquences du message Mesg, qui doivent être envoyés. Dans l'ensemble Mesgset, chaque message mesg est caractérisé par sa destination dest, et noté mesg[dest].*
- *State' est l'état du même processus où les conséquences de la réception du message Mesg ont été répercutées.*

- *Une* **C-transition** *est une α-transition pour la détection des configurations.*

  Une **P-transition***transition !P est une α-transition pour le traitement des configurations.*

  Une **U-transition***transition !U est une α-transition pour le calcul de l'unification.*

  Une **O-transition***transition !O est une α-transition pour le calcul de l'orientation.*

- *Si St est l'état d'un processus qui reçoit le message, St.tid, St.symb, St.U_list ... sont son tid, son symbole, sa liste d'arcs d'unification ...*

- *L'opérateur associatif-commutatif $\otimes$ combine les phases élémentaires de la complétion. Il est défini par : si ({Mesg}, State) $\xrightarrow{\alpha}$ (Mesgset1, State1) et ({Mesg}, State) $\xrightarrow{\beta}$ (Mesgset2, State2) alors ({Mesg}, State) $\xrightarrow{\alpha\otimes\beta}$ (Mesgset1 $\cup$ Mesgset2, State3) où State3 est défini par :*
  - *State3.tid = State.tid*
  - *State3.symb = State.symb*
  - *State3.dico_order_arity = State.dico_order_arity*
  - *State3.X = State.X + (State1.X − State.X) − (State.X − State1.X)+ (State2.X − State.X) − (State.X − State2.X)*
    pour X $\in$ {U_list,R_in_list,R_out_list,S_in_list,S_out_list,semiconfig_list, infou_list,requestu_list,answeru_list,O_in_list,O_out_list,O_eq_list,calculo_list, requesto1_list,requesto2_list, answero1_list,answero2_list}.

- *Une* **A-transition** *est une α-transition ({Mesg}, State) $\xrightarrow{A}$ (Mesgset, State'), qui exécute toutes les* D, P, U *et* O-transitions *qui peuvent être appliquées à l'état State.* $A = \alpha_1 \otimes \alpha_2 \ldots \otimes \alpha_n$ *où* $\alpha_i \in \{D,U,P,O\}$. □

3.3.4 Algorithmes

Nous précisons ici l'algorithme suivi par le processus fils.

Nous décrivons dans un premier temps la fonction *my_receive_b*.

La fonction *my_receive_b* permet une réception bloquante des messages reçus provenant de n'importe quels processus, processus maître ou processus fils. Nous considérons une réception

bloquante car les processus ne travaillent que par réception de message.

- Si l'argument de *my_receive_b* est le nom d'un message, le processus fils est bloqué tant qu'un message de ce nom n'a pas été reçu. Dès que le message voulu arrive dans le tampon du processus, il est traité. La fonction *my_receive_b* est utilisée de cette manière dans la phase d'initialisation.
- Si l'argument de *my_receive_b* est -1, le processus fils accepte de traiter tous les messages qu'il recevra. La fonction *my_receive_b* est utilisée de cette manière dans la phase de complétion.

Pour la réception et l'envoi de message, nous prenons l'hypothèse que si le processus p_1 envoie un message $mesg_1$ au processus p_2 et ensuite, un message $mesg_2$ au processus p_2, alors p_2 va recevoir $mesg_1$ avant $mesg_2$. PVM (Sunderam, 1990) qui est utilisé pour l'implantation de notre algorithme distribué garantit également cet ordre dans les messages.

La fonction *my_receive_b* décompacte le message reçu pour en déterminer le type. Elle utilise pour ce faire la fonction *unpack_tag*. Elle fait ensuite appel aux fonctions *traitement_MSG*. Pour chaque type de message *MSG* défini dans ce chapitre, il existe une fonction *traitement_MSG* associée qui met à jour l'état du processus suite à la réception du message *MSG* et envoie de nouveaux messages conséquences du message reçu.

```
my_receive_b(msgtag)

 /* décompactage du message mesg reçu
   obtention du type du message : mesg_tag */
mesg_tag = unpack_tag(mesg);

 cas
 cas mesgtag = SYMBOL
   alors traitement_symbol(mesg)
 cas mesgtag = DICO_ORDER_ARITY
   alors traitement_dico_order_arity(mesg)
 cas mesgtag = INITNODE
   alors traitement_dico_order_arity(mesg)
 cas mesgtag = INITU
   alors traitement_initu(mesg)
 cas mesgtag = INITS
   alors traitement_inits(mesg)
 cas mesgtag = INITR
   alors traitement_initr(mesg)
 ...
 fin cas
```

A partir de la fonction *my_receive_b*, nous décrivons l'algorithme suivi par les processus fils dans le tableau 3.1. L'algorithme se déroule comme suit. Le processus fils attend dans la phase d'initialisation les messages *SYMBOL*, *DICO_ORDER_ARITY*, *INITNODE*, *INITU*, *INITS* et *INITR* du processus maître dans cet ordre. L'ordre des messages est important car chaque message va entraîner l'envoi d'autres messages. Cet ordre assure que tous les messages conséquences d'un message seront envoyés. Le processus fils reçoit nb_u messages *INITU*, nb_s messages *INITS* et nb_r messages *INITR*. nb_u, nb_s et nb_r sont respectivement le nombre d'arcs d'unification, de sous-terme et de réécriture de ce processus fils sur le graphe SOUR initial de l'ensemble à compléter. Dans la phase de complétion, les processus fils ne filtrent plus les messages.

```
/* Phase l'initialisation */

terminaison = faux;

my_receive_b(SYMBOL);                  /* réception bloquante du message
                                       SYMBOL */

my_receive_b(DICO_ORDER_ARITY);        /* réception bloquante du message
                                       DICO_ORDER_ARITY */

my_receive_b(INTIPROCESS);             /* réception bloquante du message
                                       INITPROCESS */

nb_u = INITPROCESS.nb_u;               /* nb_u est le nombre d'arcs U
                                       du graphe SOUR initial. Il est
                                       extrait du message INITPROCESS */

nb_s = INITNODE.nb_s;                  /* nb_s est le nombre d'arcs S
                                       du graphe SOUR initial. Il est
                                       extrait du message INITPROCESS */

nb_r = INITNODE.nb_r;                  /* nb_r est le nombre d'arcs R
                                       du graphe SOUR initial. Il est
                                       extrait du message INITPROCESS */

 pour i de 1 à nb_u  faire
  my_receive_b(INITU);                 /* réception bloquante des nb_u
 fpour                                 messages INITU */

 pour i de 1 à nb_s  faire
  my_receive_b(INITS);
 fpour

 pour i de 1 à nb_r  faire
  my_receive_b(INITR);
 fpour

 /* Barrière de synchronisation */

pvm_barrier(...);

 /* Les processus fils ont toute l'information de la
  phase d'initialisation qui va leur permettre de
  réaliser la complétion */

 /* Phase de complétion */

 tant que terminaison = faux  faire
  my_receive(-1);
 fin tant que
```

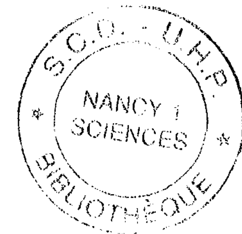

S.C.D. - U.H.P.
NANCY 1
SCIENCES
BIBLIOTHÈQUE

Table 3.1 – *Algorithme suivi par les processus fils*

3.4 Implantation des règles d'inférence de la complétion SOUR close concurrente

Nous montrons maintenant comment sont implantées les règles d'inférence de la complétion SOUR par notre algorithme distribué. Nous rappelons que ces règles d'inférence correspondent à des transformations locales du graphe SOUR. Des motifs d'arcs appelés configurations et demi-configurations sont recherchés sur le graphe SOUR et des arcs sont ajoutés et supprimés. Le problème de la détection des configurations et de leur traitement se posent donc dans le cas de notre algorithme distribué.

3.4.1 Détection des configurations

Les configurations et les demi-configurations sont définies par leurs arcs, donc par les tids des processus aux extrémités des arcs.

Pour la détection des configurations, nous utilisons le fait qu'un processus fils peut détecter une séquence :

- de deux arcs adjacents formant une configuration SUR, RUR-rhs ou RUR, ou
- deux arcs adjacents formant une demi-configuration UR.

Ces motifs de deux arcs adjacents sont détectés quand :

- un nouvel arc de sous-terme ou un nouvel arc de réécriture est ajouté, ou
- lorsque la contrainte équationnelle d'un arc d'unification devient vraie, c'est-à-dire égale à *True*.

La détection des configurations est réalisée à l'aide de messages *SEMICONF* et *CONFIG* comme décrit ci-dessous :

- Un processus p peut détecter une demi-configuration de deux manières différentes.

  (i) Supposons que p ait dans sa U_list un arc U de p à un processus p' dont la contrainte équationnelle est égale *True*. Si ce processus reçoit un message *INITR* lui signifiant l'ajout d'un arc R sortant e_r, alors p détecte une demi-configuration. Il envoie à p' un message *SEMICONF* contenant cette demi-configuration. p' est alors au courant de l'existence de cette demi-configuration.

  p détecte toutes les demi-configurations contenant e_r et les arcs U de sa U_list.

  (ii) Supposons que p ait dans sa R_out_list un arc R sortant de p à p'. Si ce processus reçoit un message *INITU* lui signifiant que la contrainte équationnelle d'un arc U e_u de p à p'' est devenue égale à *True*, alors p détecte une demi-configuration. Il envoie un message *SEMICONF* contenant cette demi-configuration à p''.

  p détecte toutes les demi-configurations contenant e_u et les arcs R de sa R_out_list.

- Un processus peut détecter une configuration SUR, RUR ou RUR-rhs composée de deux arcs adjacents.

  La détection des configurations SUR, RUR ou RUR-rhs avec un arc U est liée à la réception de messages *SEMICONF*.

  Un processus qui reçoit une demi-configuration dans un message *SEMICONF* la sauve dans une liste appelée *semiconfig_list*.

  Deux cas se présentent alors.

(i) A la réception d'un message *SEMICONF*, *p* détecte une configuration SUR par arc S de sa S_in_list, une configuration RUR par arc R de sa R_out_list et une configuration RUR-rhs par arc R de sa R_in_list.
(ii) Si *p* reçoit un message *INITS* contenant un arc S entrant, alors il détecte une configuration SUR par demi-configuration sauvée dans sa liste semiconfig_list. S'il reçoit un message *INITR* contenant un arc R sortant, il détecte un configuration RUR par demi-configuration sauvée dans sa liste semiconfig_list et s'il reçoit un message *INITR* contenant un arc R entrant, il détecte un configuration RUR-rhs par demi-configuration sauvée dans sa liste semiconfig_list.

- Les configurations sont envoyées par un message *CONFIG* à un unique processus. Le nombre de messages *CONFIG* est ainsi restreint et nous garantissons ainsi que pour une configuration détectée, il existe un unique processus qui va la traiter.

Dans ce chapitre, nous utilisons les notations suivantes :

Notation 1 *Pour un message Mesg de type CONFIG, nous écrivons CONFIG(T,S) pour montrer que la configuration contenue dans le message CONFIG est de type T (SUR, RUR ou RUR-rhs) et contient les éléments de l'ensemble S. S contient l'énumération des arcs ou l'arc et la demi-configuration formant la configuration présente dans le message.*

Nous généralisons cette notation de telle façon que pour un message Mesg, Mesg(S) signifie que le message Mesg contient les éléments de l'ensemble S. Cette notation peut être encore étendue pour définir le contenu d'un champ ou d'un arc ou d'une demi-configuration...

Nous notons les arcs de sous-terme par e_s, e'_s et e''_s, les arcs de réécriture par e_r, e'_r et e''_r, les arcs d'unification par e_u, e'_u et e''_u et les demi-configurations contenues dans un message SEMICONF par semiconf.

La détection des demi-configurations et des configurations peut être définie formellement en terme de $C - transitions$.

Les sept C-transitions qui suivent montrent comment un message *INITR* indiquant l'addition d'un nouvel arc de réécriture cause la création de demi-configurations et de configurations SUR, RUR et RUR-rhs.

(1) $(\{INITR(e_r)\}, \{St.R_out_list, St.U_list(e_u(tid))\})$
$\xrightarrow{C}$
$(\{SEMICONF(e_r,e_u)[tid]\}, \{St.R_out_list + e_r, St.U_list(e_u)\})$
si $Constraint(e_u) = True$.

(2) $(\{INITR(e_r)\}, \{St.R_out_list, St.R_in_list(e'_r(tid))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs},e_r,e'_r)[tid]\}, \{St.R_out_list + e_r, St.R_in_list(e'_r)\})$

(3) $(\{INITR(e_r(tid1))\}, \{St.R_out_list, St.R_out_list(e'_r(tid2))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR},e_r,e'_r)[tidmax]\}, \{St.R_out_list + e_r, St.R_out_list(e'_r)\})$
La configuration de type RUR est envoyée à un unique processus, le processus de tid tidmax, où tidmax = max(tid1,tid2)

(4) $(\{INITR(e_r)\}, \{St.R_out_list, St.S_in_list(e_s(tid,index))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR},e_s,e_r)[tid]\}, \{St.R_out_list + e_r, St.S_in_list(e_s)\})$

(5) $(\{INITR(e_r(tid))\}, \{St.R_in_list, St.R_out_list(e'_r)\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs},e_r,e'_r)[tid]\}, \{St.R_in_list + e_r, St.R_out_list(e'_r)\})$

(6) $(\{INITR(e_r(tid3,tid1))\}, \{St.R_out_list,$
$St.semiconfig_list(semiconf(e'_r(tid2),u(tid3,tid)))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR},e_r,semiconf)[tidmax]\}, \{St.R_out_list + e_r,$
$St.semiconfig_list(semiconf)\})$
La configuration RUR détectée n'est envoyée qu'à un seul processus. Si St.tid >tid3, elle est envoyée au processus de tid tid1 (tidmax=tid1), sinon au processus de tid tid2 ($tidmax = tid2$).

(7) $(\{INITR(e_r(tid))\}, \{St.R_in_list, St.semiconfig_list(semiconf)\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs},e_r,semiconf)[tid]\}, \{St.R_in_list + e_r,$
$St.semiconfig_list(semiconf)\})$

La C-transition suivante indique qu'une demi-configuration peut être formée, quand il existe un arc de réécriture sortant et quand la contrainte équationnelle d'un arc d'unification devient égale à *True*, c'est-à-dire quand un message *INITU* le signifiant est reçu.

(8) $(\{INITU(e_u(tid))\}, \{St.U_list, St.R_out_list(e_r)\})$
$\xrightarrow{C}$
$(\{SEMICONF(e_u,e_r)[tid]\}, \{St.U_list + e_u, St.R_out_list(e_r)\})$
$si\ Constraint(e_u) = True$

Les deux C-transitions suivantes montrent comment une configuration SUR peut être détectée suite à l'ajout d'un arc de sous-terme entrant.

(9) $(\{INITS(e_s(tid,index))\}, \{St.S_in_list, St.semiconfig_list(semiconf)\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR},e_s,semiconf)[tid]\}, \{St.S_in_list + e_s, St.semiconfig_list(semiconf)\})$

(10) $(\{INITS(e_s(tid,index))\}, \{St.S_in_list, St.R_out_list(e_r)\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR},e_s,e_r)[tid]\}, \{St.S_in_list + e_s, St.R_out_list(e_r)\})$

Les trois C-transitions suivantes montrent comment la réception d'un message *SEMICONF* entraîne la détection de configurations SUR, RUR-rhs et RUR respectivement.

(11) $(\{SEMICONF(semiconf)\}, \{St.S_in_list(e_s(tid,index)), St.semiconfig_list\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR},e_s,semiconf)[tid]\}, \{St.S_in_list(e_s), St.semiconfig_list + semiconf\})$

(12) $(\{SEMICONF(semiconf)\}, \{R_in_list(e_r(tid)), semiconfig_list\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs},e_r,semiconf)[tid]\}, \{St.R_in_list(e_r(tid)), St.semiconfig_list$
$+semiconf\})$

(13) $(\{SEMICONF(semiconf(e_u(tid1),e_r(tid1,tid2)))\}, \{St.R_out_list(e'_r(tid3)),$
$St.semiconfig_list\})$
$\xrightarrow{T}$
$(\{CONFIG(\text{RUR},e'_r,semiconf)[tidmax]\}, \{St.R_out_list(e'_r),$
$St.semiconfig_list + semiconf\})$
La configuration RUR détectée n'est envoyée qu'à un seul processus. Si St.tid >tid3, elle est envoyée au processus de tid tid1 (tidmax=tid1), sinon au processus de tid tid2 (tidmax=tid2).

3.4.2 Traitement des configurations

Le traitement d'une configuration correspond à un calcul de paire critique ou à une simplification. Dans le cas clos, il ne s'agit que de simplifications.

Le traitement d'une configuration est déclenché par la réception d'un message *CONFIG* par un processus fils. La réception d'un message *CONFIG* entraîne donc une séquence d'actions qui dépendent du type de la configuration reçue. Le processus fils qui reçoit le message *CONFIG* traite la configuration, c'est-à-dire qu'il effectue la transformation associée (addition et suppression d'un arc), et envoie des messages qui vont être utilisés pour mettre à jour l'information des autres processus fils. Ces messages sont des messages de type *INITS* et *INITR* pour annoncer qu'un nouvel arc est ajouté, et *UPDATES*, *UPDATER* et *UPDATESEMICONF* pour supprimer respectivement un arc de sous-terme, ou un arc de réécriture ou une demi-configuration de l'état d'un processus fils.

La phase élémentaire de traitement des configurations est réalisée par des P-transitions et des O-transitions. Les O-transitions ne sont visibles que dans la section 3.6.

Le traitement des configurations peut être exprimé par les P-transitions suivantes.

Les deux P-transitions suivantes montrent le traitement d'une configuration SUR, c'est-à-dire l'addition et la suppression d'un arc de sous-terme.

(14) $(\{CONFIG(\text{SUR}, e_s(tid1, index), e_r(tid2))\},\ \{St.S_out_list(e_s(tid1, index))\})$

$\xrightarrow{P}$

$(\{INITS(e'_s(tid2, index))[tid2],\ UPDATES(e_s)[tid1]\},\ \{St.S_out_list + e'_s - e_s\})$

où e'_s est un nouvel arc de sous-terme entrant et $tid2$ est le tid du processus à l'extrémité sortante de e_r.

(15) $(\{UPDATES(e_s(tid, index))\},\ \{St.S_in_list(e_s(tid, index))\})$

$\xrightarrow{P}$

$(\{\},\ \{St.S_in_list - e_s\})$

Les quatre P-transitions suivantes implantent la suppression d'un arc de réécriture dans le cas du traitement des configurations RUR et RUR-rhs. La première partie du traitement, c'est-à-dire l'addition d'un arc de réécriture, est conditionnée par le calcul de l'orientation. Elle sera donc représentée par des O-transitions (voir section 3.6).

(16) $(\{UPDATER(e_r)\}, \{St.R_out_list(e_r)\})$
$\xrightarrow{P}$
$(\{\}, \{St.R_out_list - e_r\})$

(17) $(\{UPDATER(e_r)\}, St.R_out_list(e_r), St.U_list(e_u(tid))\})$
$\xrightarrow{P}$
$(\{UPDATESEMICONF(e_r,u(tid))[tid]\}, \{St.R_out_list - e_r, St.U_list(e_u)\})$
if $Constraint(e_u) = True.$

(18) $(\{UPDATER(e_r)\}, \{St.R_in_list(e_r)\})$
$\xrightarrow{P}$
$(\{\}, \{St.R_in_list - e_r\})$

(19) $(\{UPDATESEMICONF(semiconf)\}, \{St.semiconfig_list(semiconf)\})$
$\xrightarrow{P}$
$(\{\}, \{St.semiconfig_list - semiconf\})$

3.5 Calcul concurrent de l'unification

La concurrence n'améliore pas fondamentalement le comportement de l'unification (Dwork, Kanellakis et Mitchell, 1984). Toutefois, pour le modèle de complétion que nous utilisons, nous avons besoin de mettre au point un algorithme distribué d'unification adapté.

Cet algorithme d'unification est optimisé pour réduire la quantité de travail d'un processus fils et éviter les calculs et communications inutiles. Dans une version précédente de notre algorithme d'unification, nous avons utilisé une méthode ascendante (*bottom-up*), où les messages allaient des processus S-fils vers les processus S-pères. Dans cette méthode, beaucoup de contraintes équationnelles inutiles avaient été calculées. Nous avons donc décidé d'utiliser une méthode hybride qui lie une méthode descendante (*top-down*) (des processus S-pères aux processus S-fils) et ascendante. Cette méthode peut également être appelée méthode par *demande-réponse* (*request-answer*). Elle a également été raffinée en une méthode par *demande-réponse par nécessité* (*request-answer by need*) où seules des contraintes équationnelles ciblées sont calculées. Cette méthode est décrite de façon générale dans la section 3.5.1 et de façon formelle en utilisant des U-transitions dans la section 3.5.2.

3.5.1 Principes généraux du calcul concurrent de l'unification

Calculer une contrainte équationnelle, dans le cas clos, consiste à déterminer si deux termes sont égaux et dans ce cas, la contrainte équationnelle de l'arc d'unification entre les deux processus fils représentant ces deux termes est égale à *True*. Dans le cas contraire, elle est égale à *False*. On rappelle que les arcs d'unification ne sont qu'entre des processus ayant le même symbole.

Considérons le problème d'unification de la figure 3.12. U_0 est l'arc d'unification entre le processus représentant le terme $f(s_1, \ldots, s_n)$ et le processus représentant le terme $f(t_1, \ldots, t_n)$. c_0 est la contrainte équationnelle de cet arc. Nous décrivons en utilisant la figure 3.12 comment la contrainte équationnelle c_0 est calculée.

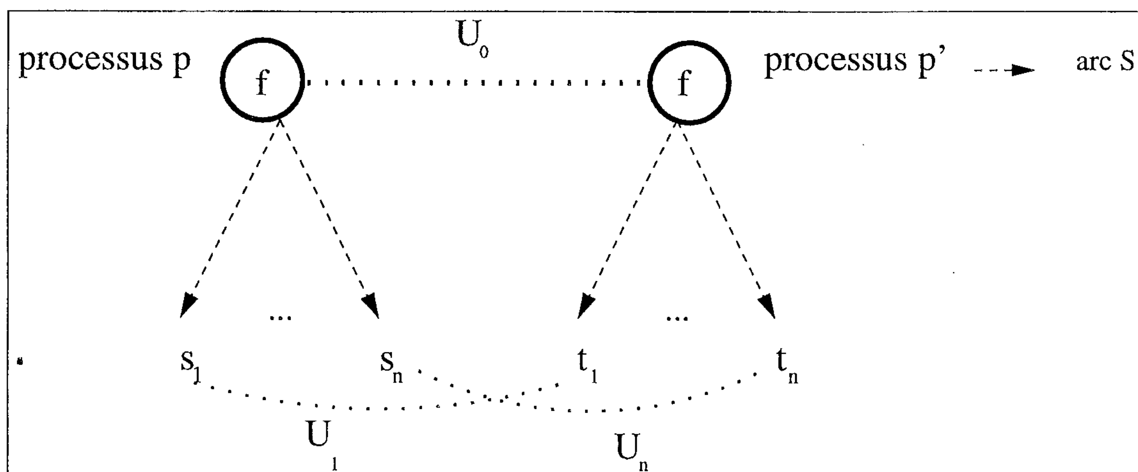


FIG. 3.12 – *Calcul de la contrainte équationnelle c_0 de l'arc d'unification U_0*

Le calcul de la contrainte équationnelle c_0 de l'arc d'unification U_0 dépend du calcul des contraintes équationnelles $c_1, \ldots, c_n$ des arcs d'unification $U_1, \ldots, U_n$. c_0 est la contrainte $c_1 \wedge \ldots \wedge c_n$. Chaque contrainte c_i représente le problème d'unification $s_i =^? t_i$.

- Initialement, toutes les contraintes équationnelles sont égales à $False$, sauf les contraintes équationnelles des arcs d'unification qui forment un cycle unaire, ces contraintes sont égales à $True$.
- Supposons que le processus p représente le terme $f(s_1,\ldots,s_n)$, que le processus p' représente le terme $f(t_1,\ldots,t_n)$ et que le processus p' soit en charge du calcul de c_0.
  Le processus p envoie au processus p' les tids des processus représentant les termes $s_1,\ldots, s_n$ pour que le processus p' sache à partir de quelles contraintes équationnelles c_0 est calculée. Ceci est réalisé avec des messages *INFOU*. Un message *INFOU* contient :
  - l'arc d'unification dont la contrainte équationnelle est à calculer, et
  - le tid et l'index d'un processus à l'extrémité d'un arc de sous-terme de p.
- L'information contenue dans les messages *INFOU* reçus par p' est sauvée dans l'état du processus p' dans une liste appelée *infou_list*. Le message *INFOU* ne contient qu'un seul tid de processus S-fils de p. Ce choix a été fait pour simplifier la mise à jour de *infou_list*.
- De plus, à la réception d'un message *INFOU* envoyé par p, p' demande alors à ses processus S-fils représentant les termes $t_1,\ldots, t_n$, si les contraintes équationnelles $c_1,\ldots, c_n$ sont égales à $True$. Il leur dit de plus qu'il calcule c_0. Ce calcul est descendant : le processus S-père (p') fait une demande à chacun de ses processus S-fils. Les demandes sont faites avec des messages *REQUESTU*.
  Lorsqu'un processus fils reçoit un message *REQUESTU*, l'information de ce message est sauvée dans son état dans une liste appelée *requestu_list*.
- Si une contrainte équationnelle c_i ($i > 0$) est égale ou devient égale à $True$ et qu'une demande issue d'un message *REQUESTU* concernant le calcul de la contrainte c_0 (utilisant c_i) a été sauvée, alors le processus fils représentant le terme t_i informe le processus fils p' que c_i est égale à $True$. Le calcul est alors ascendant : le processus S-fils envoie une réponse à son S-père. Une réponse est envoyée à l'aide d'un message *ANSWERU*.
  Les messages *ANSWERU* reçus par un processus fils S-père sont sauvés dans l'état de ce processus dans une liste appelée *answeru_list*.
- La contrainte équationnelle c_0 devient égale à $True$, si le processus p' a reçu n messages *ANSWERU* concernant le calcul de c_0, où n est l'arité du symbole du processus p'.
  Le processus p en est averti à l'aide d'un message *INITU*.
- La valeur d'une contrainte équationnelle peut changer suite à l'ajout ou à la suppression d'un arc de sous-terme.

La méthode demande-réponse du calcul d'une contrainte équationnelle a été optimisée en une méthode de calcul *demande-réponse par nécessité* (*request-answer by need*). En effet, nous avons constaté qu'il est inutile de calculer toutes les contraintes équationnelles de tous les arcs d'unification. Les contraintes équationnelles des arcs d'unification ne doivent être calculées que si ces arcs d'unification sont susceptibles de former une demi-configuration avec un arc de réécriture adjacent.

Cette observation est implantée de la manière suivante. La phase d'initialisation est considérée comme une phase de pré-traitement pour le calcul de l'unification. S'il existe un arc de réécriture sortant et un arc d'unification pour un noeud du graphe SOUR initial, alors les deux processus aux extrémités de cet arc d'unification doivent être informés qu'ils doivent calculer la contrainte équationnelle, mais seulement un des processus va faire ce calcul pour qu'il n'y ait pas redondance du travail. Dans ce cas, le processus qui n'est pas en charge du calcul de la contrainte équationnelle envoie des messages d'information *INFOU* au processus à l'autre extrémité de l'arc d'unification

pour lui permettre de calculer la contrainte équationnelle. L'optimisation réside dans le fait que l'unification n'est calculée que si c'est nécessaire. Pendant la phase de complétion, si un arc de réécriture sortant est ajouté à l'état d'un processus, et si ce processus a en plus un arc d'unification dans son état, alors, la contrainte équationnelle de cet arc d'unification doit être calculée. L'autre extrémité de l'arc d'unification doit en être informée en utilisant un message *INFOU*, si le processus à l'autre extrémité doit calculer la contrainte équationnelle, et par un message *NEEDU* dans le cas contraire. Nous définissons une fonction $Needu$ telle que, pour un arc d'unification e_u, $Needu(e_u) = True$ si la contrainte équationnelle de e_u doit être calculée comme expliqué auparavant et $Needu(e_u) = False$ sinon.

Notation 2 *Nous notons l'information contenue dans les messages INFOU par* infou *et* infou', *l'information contenue dans les messages REQUESTU par* requestu *et* requestu' *et l'information contenue dans les message ANSWERU par* answeru *et* answeru'.

3.5.2 U-transitions

L'algorithme décrit précédemment peut être formulé en utilisant les U-transitions suivantes.

Les quatre U-transitions suivantes montrent comment les structures de données *infou_list*, *requestu_list* et *answeru_list* sont mises à jour.

(20) $(\{INITU(e_u(tid))\}, \{St.U_list(e_u)\})$
$\xrightarrow{U}$
$(\{\}, \{St.U_list(e_u) - e'_u(tid) + e_u(tid)\})$
si e'_u est l'arc d'unification e_u avec une contrainte équationnelle à False et si $Constraint(e_u) = True$ dans le message INITU.

(21) $(\{INFOU(infou(e_s(tid,index)),e_u(tid))\}, \{St.infou_list, St.U_list(e_u)\})$
$\xrightarrow{U}$
$(\{\}, \{St.infou_list - infou'(index,e_u) + infou(e'_s,e_u), St.U_list(e_u)\})$
où tid est l'extrémité sortante de l'arc de sous-terme e'_s et $infou'$ est une ancienne information d'unification reçue concernant le calcul de la contrainte équationnelle de e_u en utilisant l'index *index*. Si initialement, $Needu(e_u) = False$, cette valeur est changée à $True$.

(22) $(\{REQUESTU(requestu(e_u(tid),e'_u(tid2,tid1),e'_s(tid1,tid,index)))\}, \{St.U_list(e_u), S_in_list(e'_s(tid2,index)), St.requestu_list\})$
$\xrightarrow{U}$
$(\{\}, \{St.U_list(e_u), St.S_in_list(e'_s), St.requestu_list + requestu - requestu'(e'_u,e_s)\})$
où $requestu'$ est dû à un ancien message REQUESTU reçu concernant le calcul de la contrainte de e_u et contenant *index*.

(23) $(\{ANSWERU(e'_u(tid1,tid2),e_u(St.tid,tid),index)\}, \{St.U_list(e_u(tid)), St.S_out_list(e'_s(tid1,index))\})$
$\xrightarrow{U}$
$(\{\}, \{St.U_list(e_u), St.S_out_list(e'_s), St.answeru_list - answeru' + answeru\})$
où $answeru'$ provient d'un ancien message ANSWERU reçu concernant le calcul de la contrainte de e_u et contenant *index*.

Les quatre U-transitions suivantes montrent comment l'addition d'un arc de sous-terme sortant ordonnée par un message *INITS* ou l'addition d'un arc de sous-terme sortant résultant du traitement d'une configuration SUR entraînent l'envoi de messages d'unification. En effet, les contraintes des arcs d'unification présents dans la U_list du processus ajoutant un arc de sous-terme sortant sont susceptibles d'avoir changé. Elles doivent être recalculées. Les messages envoyés sont soit des messages d'information concernant les contraintes équationnelles à calculer, c'est-à-dire des messages *INFOU*, soit des messages de demande d'information concernant les contraintes équationnelles à calculer, c'est-à-dire des messages *REQUESTU*.

(24) $(\{INITS(e_s)\}, \{St.S_out_list, St.U_list(e_u(tid))\})$
$\xrightarrow{U}$
$(\{INFOU(e_s,e_u)[tid]\}, \{St.S_out_list + e_s, St.U_list(e_u)\})$
si $Constraint(e_u) = False$, $Needu(e_u) = True$ et si le processus qui reçoit le message INITS n'est pas en charge du calcul de la contrainte équationnelle de e_u.

(25) $(\{INITS(e_s(tid,index))\}, \{St.S_out_list, St.U_list(e_u), St.infou_list(infou(e_u,index))\})$
$\xrightarrow{U}$
$(\{REQUESTU(infou,e_s)[tid]\}, \{St.S_out_list + e_s, St.U_list(e_u), St.infou_list(infou)\})$
si $Constraint(e_u) = False$ et le processus qui reçoit le message INITS
n'est pas en charge du calcul de la contrainte équationnelle de e_u.

(26) $(\{CONFIG(\mathrm{SUR},e_s(index),e_r(tid'))\}, \{St.S_out_list(e_s(index)), St.U_list(e_u(tid))\})$
$\xrightarrow{U}$
$(\{INFOU(e'_s(tid',index),e_u)[tid]\}, \{St.S_out_list - e_s + e'_s, St.U_list(e_u)\})$
si $Constraint(e_u) = False$, $Needu(e_u) = True$ et si le processus qui reçoit le message CONFIG n'est pas en charge du calcul de la contrainte équationnelle de e_u.
tid' est le tid du processus à l'extrémité sortante de e_r. e'_s est le nouvel arc de sous-terme sortant.

(27) $(\{CONFIG(\mathrm{SUR},e_s(index),e_r(tid'))\}, \{St.S_out_list(e_s(index)), St.U_list(e_u(tid)),$
$St.infou_list(infou(index,e_u(tid)))\})$
$\xrightarrow{U}$
$(\{REQUESTU(e'_s(index,tid'),infou)[tid']\}, \{St.S_out_list - e_s + e'_s, St.U_list(e_u),$
$St.infou_list(infou)\})$
si $Constraint(e_u) = False$ et si le processus qui reçoit le message CONFIG est en charge du calcul de la contrainte équationnelle de e_u.
tid' est le tid du processus à l'extrémité sortante de e_r. e'_s est le nouvel arc de sous-terme sortant.

Les quatre U-transitions suivantes montrent comment des réponses à des demandes concernant les valeurs de contraintes équationnelles remontent vers les S-pères en utilisant des messages *ANSWERU*.

(28) $(\{INITS(e_s(tid,index))\}, \{St.S_in_list, St.U_list(e_u),$
$St.requestu_list(requestu(e_u,index))\})$
$\xrightarrow{U}$
$(\{ANSWERU(e_u,e_s)[tid]\}, \{St.S_in_list + e_s, St.U_list(e_u), St.requestu_list(requestu)\})$
si $Constraint(e_u) = True$.

(29) $(\{INITU(e_u(tid))\}, \{St.U_list(e_u(tid)), St.S_in_list(e_s(tid',index)),$
$St.requestu_list(requestu(e_u,e_s))\})$
$\xrightarrow{U}$
$(\{ANSWERU(e_u,e_s)[tid']\}, \{St.U_list(e_u), St.S_in_list(e_s), St.requestu_list(requestu)\})$
si $Constraint(e_u) = False$ initialement St.U_list et $Constraint(e_u) = True$ dans le message INITU.

(30) $(\{REQUESTU(requestu(e_u,e_s(tid,index)))\}, \{St.U_list(e_u), S_in_list(e_s(tid,index)),$
$St.requestu_list\})$
$\xrightarrow{U}$
$(\{ANSWERU(e_u,e_s)[tid]\}, \{St.U_list(e_u), St.S_in_list(e_s), St.requestu_list(requestu)\})$
si $Constraint(e_u) = True$.

(31) $(\{ANSWERU(e_u(tid))\}, \{St.U_list(e_u(tid)),$
$St.requestu_list(e'_u(tid1),e_u(tid),e_s(index,tid1)), St.S_in_list(e_s)\})$
$\xrightarrow{U}$
$(\{ANSWERU(e'_u,e_s)[tid1]\}, \{St.U_list(e_u), St.requestu_list(e'_u,e_u,e_s), St.S_in_list(e_s)\})$
si le nombre de messages ANSWERU reçus pour calculer la contrainte équationnelle de e_u, initialement égale à False, est égal à l'arité de St.symbole. La contrainte équationnelle d'unification de e_u devient alors *True*.

La U-transition suivante montre comment à la réception d'un message *INFOU*, un processus envoie des demandes d'information concernant la contrainte équationnelle contenue dans le message *INFOU* en utilisant des messages *REQUESTU*.

(32) $(\{INFOU(infou(e'_s(tids,index)),e_u)\}, \{St.S_out_list(e_s(tid,index)), St.infou_list,$
$St.U_list(e_u)\})$
$\xrightarrow{U}$
$(\{REQUESTU(infou,e_s)[tid]\}, \{St.S_out_list(e_s), St.infou_list(infou), St.U_list(e_u)\})$
où *tids* est l'extrémité sortante de l'arc de sous-terme e'_s et $infou$ provient d'une ancienne information reçue dans un message INFOU concernant le calcul de la contrainte d'unification de e_u en utilisant *index*. Si initialement, $Needu(e_u) = False$, cette valeur est mise à *True*.

Les deux U-transitions suivantes montrent comment si une demande d'information concernant une contrainte équationnelle est reçue dans un message *REQUESTU*, et si aucune réponse n'est disponible, d'autres demandes sont envoyées pour rendre une réponse disponible dans le futur.

(33) $(\{REQUESTU(requestu(e_u,e_s))\}, \{St.U_list(e_u), St.S_in_list(e_s), S_out_list(e'_s(tid)), St.infou_list(infou(e_u,e''_s))\})$
$\xrightarrow{U}$
$(\{REQUESTU(e_u,e'_s,e''_s)[tid]\}, \{St.U_list(e_u), St.S_in_list(e_s), St.S_out_list(e'_s), St.infou_list(infou)\})$
si $Constraint(e_u) = False$ et si le processus qui reçoit REQUESTU est en charge du calcul de la contrainte d'unification de e_u.

(34) $(\{REQUESTU(requestu(e_u(tid),e_s))\}, \{St.U_list(e_u(tid)), St.S_in_list(e_s), S_out_list(e'_s)\})$
$\xrightarrow{U}$
$(\{INFOU(e_u,e'_s)[tid]\}, \{St.U_list(e_u), St.S_out_list(e'_s), St.S_in_list(e_s)\})$
si $Constraint(u) = False$ et si le processus qui reçoit REQUESTU n'est pas en charge du calcul de la contrainte d'unification de e_u. *requestu'* est dû à un ancien message REQUESTU reçu concernant le calcul de la contrainte de e_u et contenant *index*.

La U-transition suivante est utilisée pour décider quand une contrainte équationnelle devient égale à *True*.

(35) $(\{ANSWERU(e_u(tid))\}, \{St.U_list(e_u(tid))\})$
$\xrightarrow{U}$
$(\{INITU(e_u)[tid]\}, \{St.U_list(e_u)\})$
si le nombre de messages ANSWERU reçus pour le calcul de la contrainte d'unification de e_u, initialement fausse, est égal à l'arité de St.symbole.
La contrainte d'unification de e_u devient alors *True*.

Les U-transitions suivantes optimisent le calcul des contraintes équationnelles. Cette optimisation est basée sur le critère qu'une contrainte équationnelle doit être calculée par un processus que si ce processus a dans son état un arc d'unification et un arc de réécriture sortant. Ainsi, quand un arc de réécriture sortant est ajouté explicitement dans l'état d'un processus suite à la réception d'un message *INITR* ou implicitement quand une configuration RUR ou RUR-rhs est traitée, des calculs d'unification doivent être effectués. Dans ce dernier cas, nous utilisons des O-transitions (voir la section 3.6), car l'ajout d'un arc de réécriture suite au traitement d'une configuration RUR ou RUR-rhs est lié à l'orientation.

(36) $(\{INITR(e_r)\}, \{St.R_out_list, St.U_list(e_u(tid)), St.S_out_list(e_s(tid1))\})$
$(\{INFOU(e_u,e_s)[tid]\}, \{St.R_out_list + e_r, St.U_list(e_u), St.S_out_list(e_s)\})$
$\xrightarrow{U}$
si $Constraint(e_u) = False$ et si le processus qui reçoit INITR n'est pas en charge du calcul de la contrainte d'unification de e_u et si $Needu(e_u) = False$ initialement pour le processus qui reçoit le message INITR. Après le traitement du message INITR, $Needu(e_u)$ devient égal à *True*.

(37) $(\{INITR(e_r)\}, \{St.tid, St.R_out_list, St.U_list(e_u(tid)), St.S_out_list(e_s(tid1))\})$
$\xrightarrow{U}$
$(\{NEEDU(St.tid)[tid]\}, \{St.tid, St.R_out_list + e_r, St.U_list(e_u), St.S_out_list(e_s)\})$
si $Constraint(e_u) = False$ et si le processus qui reçoit INITR n'est pas en charge du calcul de la contrainte d'unification de e_u et si $Needu(e_u) = False$ initialement pour le processus qui reçoit INITR. Après le traitement du message INITR, $Needu(e_u)$ devient égal à *True*.

(38) $(\{NEEDU(e_u)\}, \{St.S_out_list(s(tid,index)), St.U_list(e_u(tid1))\})$
$\xrightarrow{U}$
$(\{INFOU(e_u,e_s)[tid1]\}, \{St.S_out_list(s), St.U_list(e_u)\})$
si le processus qui reçoit le message NEEDU n'est pas en charge du calcul de la contrainte d'unification de e_u. Après le traitement du message NEEDU, $Needu(e_u) = True$.

Correction et complétude Notre calcul concurrent de l'unification est correcte et complet (voir la section 3.9).

3.6 Calcul concurrent de l'orientation

Pour le calcul concurrent de l'orientation, nous utilisons un algorithme distribué de type demande-réponse comme nous l'avons fait dans le cas de l'unification dans la section 3.5. Notre algorithme est basé sur l'utilisation des arcs d'orientation et l'implantation graphique de l'ordre LPO que nous donnons dans la section 3.6.2. Nous décrivons l'algorithme de façon générale dans la section 3.6.3 et de façon formelle en utilisant de O-transitions dans la section 3.6.4.

3.6.1 Arcs d'orientation

Calculer une contrainte d'ordre consiste à déterminer si un terme s est plus grand ou plus petit qu'un terme t par rapport à l'ordre LPO ou même égal à t.

Si $s \succ_{lpo} t$, nous créons un arc d'orientation allant du processus fils représentant le terme s, p_1, au processus fils représentant le terme t, p_2. Si $t \succ_{lpo} s$, nous créons un arc d'orientation allant du processus fils p_2 au processus fils p_1. Si s est égal à t, nous créons un arc d'orientation entre p_1 et p_2. Cet arc d'orientation est un arc d'orientation non orienté en opposition aux arcs d'orientation entrants ou sortants. Quand un arc d'orientation non orienté existe entre un processus fils p_1 et un processus fils p_2, c'est qu'il existe également un arc d'unification avec une contrainte équationnelle égale à $True$ entre p_1 et p_2.

Chaque processus fils sauve ses arcs d'orientation dans son état dans des listes appelées O_in_list, O_out_list et O_eq_list, suivant le cas où, respectivement, les arcs d'orientation sont entrants, sortants ou non orientés.

Un arc d'orientation est caractérisé par :

- le tid du processus fils à l'autre extrémité de l'arc d'orientation, et
- le symbole du processus fils à l'autre extrémité de l'arc d'orientation.

3.6.2 Implantation graphique de l'ordre LPO

Nous montrons ici comment nous pouvons passer d'une définition textuelle de l'ordre LPO à une implantation graphique de LPO adaptée au calcul concurrent de l'orientation dans notre algorithme distribué de la complétion SOUR et qui est la base de notre algorithme distribué de calcul de l'orientation.

Cette implantation graphique de LPO est à l'origine des travaux sur la génération d'une précédence pour rendre une contrainte d'ordre exprimée à l'aide de LPO satisfaisable par T. Genet et I. Gnaedig (Genet et Gnaedig, 1997).

Soit p_1 le processus représentant le terme $s = f(s_1, \dots, s_n)$ et p_2 le processus représentant le terme $t = g(t_1, \dots, t_m)$. Nous voulons savoir si $s \succ_{LPO} t$, c'est-à-dire si nous pouvons ajouter un arc d'orientation O_0 allant du processus représentant le terme s au processus représentant le terme t. Les trois cas de la définition textuelle de l'ordre LPO de la section 1.2.1 peuvent être exprimés par les trois cas graphiques suivants respectivement.

1. Il y a un arc d'orientation O_0 de p_1 à p_2 , si les deux conditions suivantes sont vérifiées.
   - Le symbole du processus fils p_1 est plus grand que le symbole du processus fils p_2 dans la précédence, et
   - Le processus fils p_1 est lié à chaque processus S-fils de p_2 par un arc d'orientation sortant de p_1.

   La figure 3.13 résume ce cas.

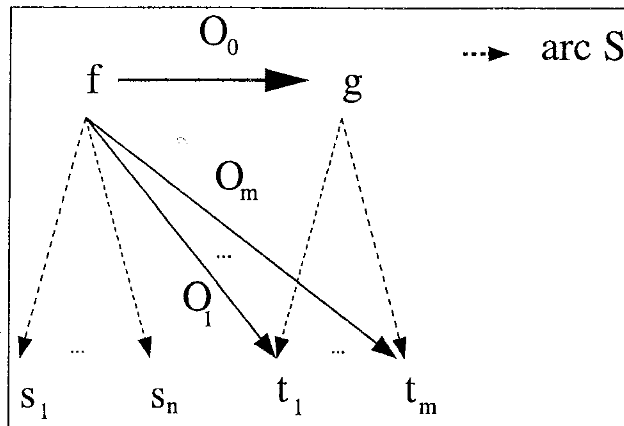


FIG. 3.13 – *Implantation graphique de LPO - Cas 1*

2. Il y a un arc d'orientation O_0 du processus fils p_1 au processus fils p_2 , s'il existe un arc d'orientation sortant ou non orienté d'un processus S-fils de p_1 à p_2.
   La figure 3.14 résume ce cas.

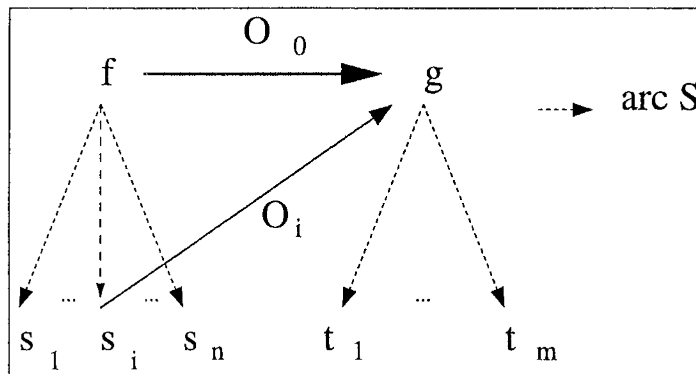


FIG. 3.14 – *Implantation graphique de LPO - Cas 2*

3. Il y a un arc d'orientation O_0 du processus fils p_1 au processus fils p_2, si les trois conditions suivantes sont vérifiées.
   - Le symbole du processus fils p_1 est égal au symbole du processus fils p_2.
   - Le processus fils p_1 est lié à chaque processus S-fils de p_2 par un arc d'orientation sortant de p_1.
   - Il existe un processus S-fils de p_1 d'index i ayant un arc d'orientation sortant vers le processus S-fils de p_2 d'index i, et il existe des arcs d'orientation non orientés des processus S-fils de p_1 d'index j tels que $j < i$ vers les processus S-fils de p_2 d'index j respectivement.

   La figure 3.15 présente le cas où le n^{ime} S-fils de p_1 est lié avec le n^{ime} S-fils de p_2 par un arc d'orientation sortant de p_1 et que les arcs d'orientation O_{n+1}, ..., O_{2n-1} sont non orientés.

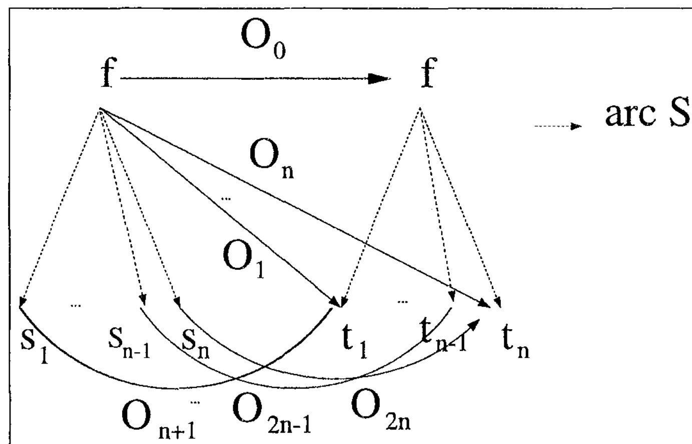


FIG. 3.15 – *Implantation graphique de LPO - Cas 3*

3.6.3 Principes généraux du calcul concurrent de l'orientation

Le calcul concurrent de l'orientation est réalisé par un algorithme distribué de type demande-réponse. Une demande consiste pour un processus fils à demander à chacun de ses processus S-fils s'ils ont des arcs d'orientation entrant d'un processus donné à eux. Les processus S-fils ne répondent que si de tels arcs existent. L'algorithme garantit que deux processus ne calculent pas la même contrainte d'orientation.

L'algorithme distribué de calcul concurrent de l'orientation que nous proposons s'appuie sur les principes décrit ci-dessous.

- Le calcul concurrent de l'orientation est basé sur la connaissance des symboles des deux processus aux extrémités de l'arc d'orientation à orienter.
  Soit $symb_1$ le symbole d'un processus fils p_1 et $symb_2$ le symbole d'un processus fils p_2. Si on a à ajouter un arc d'orientation entre p_1 et p_2, alors :
  - Si $symb_1 \succ_p symb_2$, p_2 est en charge du calcul de la contrainte d'orientation de cet arc.
  - Si $symb_1 \succ_p symb_2$, p_1 est en charge du calcul de la contrainte d'orientation de cet arc.
  - Si $symb_1 = symb_2$, p_1 et p_2 sont en charge du calcul de la contrainte d'orientation de cet arc.
- Le calcul d'une contrainte d'orientation a lieu dans deux cas.
  - Il a lieu lorsqu'une configuration RUR ou RUR-rhs est reçue par un processus pour orienter le nouvel arc de réécriture à ajouter. Les arcs d'orientation servent à orienter les arcs de réécriture. S'il existe un arc d'orientation correspondant à l'arc de réécriture à ajouter, l'arc de réécriture correspondant peut être orienté. Dans le cas contraire, des demandes d'information d'orientation sont faites pour orienter cet arc par des messages *REQUESTO1* et *REQUESTO2*.
  - Il a lieu également lorsqu'une configuration SUR est reçue. Dans ce cas, le processus p qui reçoit la configuration SUR voit le terme qu'il représente être remplacé par un terme plus petit. La direction des arcs d'orientation sortants n'est plus forcément vraie. De nouvelles demandes d'orientation sont faites par des messages *REQUESTO1* et *REQUESTO2*.

- Nous montrons ici comment procède notre algorithme distribué. Il est basé sur l'implantation graphique de LPO.
  1. Supposons que le processus p_1 ait à calculer la contrainte d'orientation d'un arc d'orientation O_0 entre lui-même et un processus fils p_2, que le processus p_1 ait pour symbole $symb_1$ et que le processus p_2 ait pour symbole $symb_2$ tels que $symb_1 \succ_p symb_2$.

     – Si $symb_2$ est une constante, un arc d'orientation sortant de p_1 peut être ajouté de p_1 à p_2 dans la O_out_list de p_1 et dans la O_in_list de p_2. Pour réaliser cette dernière opération, un message *INO* est envoyé à p_2. On peut en effet remarquer que dans le cas où au moins un des symboles des processus est une constante, le calcul de la contrainte d'orientation de O_0 est trivial en utilisant LPO.

     – Si $symb_2$ n'est pas une constante, le processus p_1 envoie un message appelé *CALCULO* au processus p_2 pour lui signifier qu'ils doivent coopérer pour calculer la contrainte d'orientation de l'arc O_0. Le message *CALCULO* contient les caractéristique des processus S-fils de p_1. Ce message signifie également à p_2 qu'il est en charge du calcul de la contrainte d'orientation (car $symb_1 \succ_p symb_2$).

     – Lorsque le processus p_2 reçoit le message *CALCULO*, il le sauve dans son état dans une structure de données appelée *calculo_ list* et envoie à chacun de ses processus S-fils une demande pour leur demander s'il y a un arc d'orientation sortant de p_1 à eux. Ceci est réalisé par des messages appelés *REQUESTO1*.

     – Considérons maintenant un processus S-fils de p_2, p_{2_i}.
     Une demande contenue dans un message *REQUESTO1* venant de p_2 n'est traitée par p_{2_i} que s'il existe un arc de sous-terme de p_2 à p_{2_i}. C'est dans ce sens que les arcs sont des canaux de communication.
     Si c'est le cas, la demande est sauvée par p_{2_i} dans une liste appelée requesto1_list et les tests suivants sont réalisés.

     - Si la O_in_list de p_{2_i} contient un arc d'orientation de p_1 à p_{2_i}, p_{2_i} envoie à p_2 une réponse lui disant qu'il existe un arc d'orientation de p_1 à p_{2_i}. Cette réponse est envoyée à l'aide d'un message *ANSWERO1*.
       Lorsque p_2 reçoit le message *ANSWERO1*, il ne le traite que s'il existe un arc de sous-terme de p_2 à p_{2_i}. Dans ce cas, la réponse contenue dans *ANSWERO1* est sauvée dans l'état de p_2 dans une liste appelée answero1_list.
     - Si la O_out_list de p_{2_i} contient un arc d'orientation sortant de p_{2_i} à p_1, le processus p_{2_i} envoie un message au processus p_2 pour lui signifier qu'il existe un arc d'orientation de p_2 à p_1 en utilisant un message *OUTO*.
       Lorsque p_2 reçoit le message *OUTO*, il ne le traite que s'il existe un arc de sous-terme de p_2 à p_{2_i}. Dans ce cas, p_2 indique au processus p_1 que l'arc d'orientation O_0 va de p_2 à p_1 en utilisant un message *INO*. Nous nous situons alors dans le cas 2 de l'implantation graphique de LPO.
     - S'il n'existe pas d'arc d'orientation entre p_{2_i} et p_1, soit un arc d'orientation peut facilement être ajouté parce que le symbole de p_{2_i} ou de p_1 est une constante ou des demandes doivent être envoyées en considérant les symboles de p_1 et p_{2_i}.

     – Si le processus p_2 reçoit n réponses *ANSWERO1* qu'il peut traiter, une de chacun de ses n S-fils p_{2_i}, lui signifiant qu'il existe un arc d'orientation de p_1 à p_{2_i}, p_2 peut décider que l'orientation de l'arc O_0 va de p_1 à p_2. Il ajoute le nouvel arc d'orientation O_0 dans sa O_in_list et envoie donc à p_1 un message *OUTO*. Si O_0 existe dans la O_out_list de p_1, cette information est mise à jour car l'état d'un processus ne peut

contenir d'information contradictoire. Nous sommes ici dans le cas 1 de l'implantation graphique de LPO.

Si nous sommes dans le cas du traitement d'une configuration, l'arc d'orientation sert à ajouter un arc de réécriture. L'arc de réécriture correspondant est donc ajouté.

2. Supposons maintenant que le processus p_1 ait besoin de calculer la contrainte d'orientation d'un arc d'orientation O_0 entre lui-même et un processus fils p_2 et que les processus p_1 et p_2 aient pour symbole *symb* d'arité n.
   - Les processus p_1 et p_2 sont tous les deux impliqués dans le calcul de la contrainte d'orientation de l'arc O_0. Ils sont tous les deux en charge du calcul de la contrainte d'orientation de O_0. Le processus p_1 envoie un message au processus p_2 pour lui signifier qu'il est en charge du calcul de la contrainte d'orientation de l'arc O_0 en lui envoyant un message *CALCULO*.
   - Lorsque le processus p_2 reçoit ce message, il le sauve dans sa liste *calculo_ list* et envoie à chacun de ses S-fils un message *REQUESTO1* comme dans le cas 1., mais il leur envoie également un message *REQUESTO2*. Un message *REQUESTO2* contient la demande : Y-a-il un arc d'orientation de p_1 entrant ou non orienté du processus S-fils de p_1 d'index i au processus S-fils de p_2 d'index i?

     De plus, le processus p_2 envoie un message appelé *CALCULO1* au processus p_1 pour lui dire qu'il doit lui aussi calculer la contrainte d'orientation correspondant à l'orientation de l'arc O_0. Ce message *CALCULO1* contient les tids et les index des processus S-fils de p_2. Les conséquences du message *CALCULO1* sont les même que celles du message *CALCULO* mis à part l'envoi du message *CALCULO1*.

     Les messages *CALCULO1* reçus par un processus sont sauvés dans la liste calculo_list de ce processus.
   - Lorsque le processus p_{2_i} S-fils du processus p_2 reçoit un message *REQUESTO1*, il sauve ce message dans requesto1_list et réalise une séquence de tests.
     - Si la O_in_list ou la O_out_list de p_{2_i} contient un arc d'orientation entre p_1 et p_{2_i}, p_{2_i} a le même comportement que celui décrit dans le cas 1 suite à la réception d'un message *REQUESTO1*.
     - S'il n'y a pas d'arc d'orientation entre p_1 et p_{2_i}, alors soit un arc d'orientation peut facilement être ajouté parce que le symbole de p_1 ou p_{2_i} est une constante, soit des demandes doivent être envoyées en considérant les symboles de p_1 et p_{2_i}.
   - Si le processus p_{2_i}, i^{ime} S-fils de p_2, reçoit un message *REQUESTO2*, il sauve ce message dans son état dans une liste appelée requesto2_list et réalise les tests suivants.
     - Si la O_in_list de p_{2_i} (respectivement sa O_eq_list) contient un arc d'orientation entre p_{1_i}, le S-fils de p_1 d'index i et p_{2_i}, alors p_{1_i} envoie au processus p_2 un message appelé *ANSWERO2* contenant le drapeau > (respectivement =).
     - S'il n'y a pas d'arc d'orientation entre p_{1_i} et p_{2_i}, alors soit un arc d'orientation peut facilement être ajouté, parce que le symbole de p_{1_i} ou de p_{2_i} est une constante, soit des demandes doivent être envoyées en considérant les symboles de p_{1_i} et p_{2_i}.
   - Si le processus p_2 reçoit n messages *ANSWERO1* à traiter, un de chacun de ses n processus S-fils, et k messages *ANSWERO2* contenant le drapeau = de

ses k premiers processus S-fils et un message *ANSWERO2* contenant le drapeau $>$ de son $k + 1^{ime}$ processus S-fils ($k \in \{1,\ldots,n-1\}$) et si tous ces messages concernent le calcul de la contrainte d'orientation de l'arc O_0, alors le processus p_2 peut décider que l'arc d'orientation O_0 va de p_1 à p_2. Le processus p_2 ajoute le nouvel arc d'orientation O_0 à sa O_in_list et envoie au processus p_1 un message *OUTO* pour lui signifier la direction de O_0.

Le processus p_1 fait le même test et peut décider que l'arc d'orientation 0_0 va de p_2 à p_1.

Notation 3 *Nous notons e_o, e'_o et e''_o les arcs d'orientation, $requesto_1$ et $requesto'_1$ l'information contenue dans un message* REQUESTO1, *$requesto_2$ et $requesto'_2$ l'information contenue dans un message* REQUESTO2, *answero1 et answero1', l'information contenue dans un message* ANSWERO1, *answero2 et answero2', l'information contenue dans un message* ANSWERO2 *et, calculo1, calculo et calculo' l'information contenue dans un message* CALCULO *ou* CALCULO1.

3.6.4 O-transitions

Le calcul distribué de l'orientation s'exprime à l'aide des O-transitions suivantes.

Les cinq O-transitions suivantes expliquent comment les structures de données calculo_list, requesto1_list, requesto2_list, answero1_list et answero2_list sont mises à jour.

(39) $(\{CALCULO(calculo(e_o(tid_1,symb_1),e_s(tid_1,tid_2,i))\},\ \{St.calculo_list\}))$

$\xrightarrow{O}$

$(\{\},\ \{St.calculo_list - calculo'(e_o,e'_s(tid_1,i)) + calculo\})$
où $calculo'$ est dû à un ancien message CALCULO reçu et sauvé concernant le calcul de la contrainte d'orientation de e_o et contenant l'index i.

(40) $(\{REQUESTO1(requesto_1(e_o(tid_1,tid_2),e'_o(St.tid,tid_1),i))\},\ \{St.tid,$ $St.S_in_list(e_s(tid_2,i)),St.requesto1_list\})$

$\xrightarrow{O}$

$(\{\},\ \{St.tid,\ St.S_in_list(e_s),\ St.requesto1_list + requesto_1\})$

(41) $(\{REQUESTO2(requesto_2(e_o(tid_1,tid_2),e'_o(St.tid,tid_3),i))\},\ \{St.tid,$ $St.S_in_list(e_s(tid_2,i)),St.requesto2_list\})$

$\xrightarrow{O}$

$(\{\},\ \{St.tid,\ St.S_in_list(e_s),\ St.requesto2_list - requesto'_2(e_o,i) + requesto_2\})$
où $requesto'_2$ est dû à un ancien message REQUESTO2 reçu et sauvé concernant le calcul de la contrainte d'orientation de e_o et contenant l'index i.
tid_3 est le tid du i^{ime} S-fils du processus de tid tid_1.

(42) $(\{ANSWERO1(answero1(e_o(tid_1,St.tid),e'_o(tid_2,tid_1),i))\},\ \{St.tid,$ $St.S_out_list(e_s(tid_2,i),answero1_list)\})$

$\xrightarrow{O}$

$(\{\},\ \{St.tid,\ St.S_out_list(e_s),\ St.answero1_list + answero1\})$

(43) $(\{ANSWERO2(answero2(e_o(tid_2,tid_1),e'_o(St.tid,tid_3),i))\},\ \{St.tid,$ $St.S_out_list(e_s(tid_2,i)),St.answero2_list\})$

$\xrightarrow{O}$

$(\{\},\{St.tid,\ St.S_out_list(e_s),\ St.answero2_list - answero2'(e_o,i) + answero2\})$
où $answero2'$ est dû à un ancien message ANSWERO2 reçu et sauvé concernant le calcul de la contrainte d'orientation de e_o et contenant l'index i.
tid_3 est le tid de i^{ime} S-fils du processus de tid tid_1.

Les cinq O-transitions suivantes montrent comment un arc de réécriture sortant est ajouté lors du traitement d'une configuration RUR. En particulier, un arc de réécriture sortant peut être ajouté :

- si un arc d'orientation sortant peut être facilement ajouté, car le symbole d'au moins un des processus aux extrémités de l'arc de réécriture à ajouter est une constante, ou
- s'il existe un arc d'orientation sortant correspondant.

(44) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2))\}, \{St.R_in_list(e_r(tid_1)),$
$St.O_out_list(e_o(tid_2)), St.R_out_list\})$
$\xrightarrow{O}$
$(\{INITR(St.tid)[tid_2]\}, \{St.R_in_list - e_r,\ St.O_out_list(e_o),\ St.R_out_list +$
$e'_r(tid_2)\})$
où tid_2 est le tid du processus à l'extrémité sortante de l'arc de réécriture e''_r.
e'_r est le nouvel arc de réécriture à ajouter à St.R_out_list.

(45) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2))\}, \{St.O_out_list(e_o(tid_2)),$
$St.R_in_list(e_r(tid_1))\})$
$\xrightarrow{O}$
$(\{UPDATER(e_r)[tid_1]\}, \{St.O_out_list(e_o),\ St.R_in_list - e_r\})$
Même conditions que pour la règle (44).

(46) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2, symb_2))\}, \{St.symb,\ St.tid,\ St.O_out_list,$
$St.R_in_list(e_r(tid_1)), St.R_out_list\})$
$\xrightarrow{O}$
$(\{INO(St.tid)[tid_2], INITR(St.tid)[tid_2]\}, \{St.symb,\ St.tid,\ St.O_out_list +$
$e_o(tid_2, symb_2), St.R_in_list - e_r,\ St.R_out_list + e'_r(tid_2, symb_2)\})$
si $symb_2$ est une constante et $St.symb > symb_2$ et il n'y a pas d'arc
d'orientation entre le processus de tid $St.tid$ et le processus
de tid_2 (tid du processus à l'extrémité sortante de e''_r). e_o est un nouvel arc d'orientation à ajouter. e'_r est le nouvel arc de réécriture à ajouter.

(47) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2, symb_2))\}, \{St.symb,\ St.tid,$
$St.R_in_list(e_r(tid_1))\})$
$\xrightarrow{O}$
$(\{UPDATER(St.tid)[tid_2]\}, \{St.symb,\ St.tid,\ St.R_in_list - e_r\})$
Même conditions que dans la règle (46).

(48) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2, symb_2))\}, \{St.symb,\ St.tid,$
$St.S_in_list(e_s(tid, i)), St.R_in_list(e_r)\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(tid, tid_2), St.tid, i, 2)[tid]\}, \{St.symb,\ St.tid,\ St.S_in_list(s_s),$
$St.R_in_list - e_r\})$
Même conditions que dans la règle (46).

Les quatre O-transitions suivantes montrent comment un arc de réécriture entrant est ajouté lors du traitement d'une configuration RUR. En particulier, un arc de réécriture entrant peut être ajouté :

- si un arc d'orientation entrant peut être facilement ajouté, car le symbole d'au moins un des processus aux extrémités de l'arc de réécriture à ajouter est une constante, ou
- s'il existe un arc d'orientation entrant correspondant.

(49) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2, tid_3), e_u(tid_1, tid_2))\}, \{St.O_in_list(e_o(tid_3)), St.R_in_list(e_r(tid_1))\})$

$\xrightarrow{O}$

$(\{INITR(St.tid)[tid_3]\}, \{St.O_in_list(e_o), St.R_in_list + e'_r(tid_3)\})$

où tid_3 est le tid du processus à l'extrémité sortante de l'arc de réécriture e''_r. e'_r est le nouvel arc de réécriture à ajouter.

(50) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2, tid_3), e_u(tid_1, tid_2))\}, \{St.O_in_list(e_o(tid_3)), St.R_in_list(e_r(tid_1))\})$

$\xrightarrow{O}$

$(\{UPDATER(e''_r(tid_3))[tid_2], UPDATER(e''_r(tid_2))[tid_3]\}, \{St.O_in_list(e_o), St.R_in_list + e'_r(tid_3)\})$

où e'_r est le nouvel arc de réécriture. tid_3 est le tid du processus à l'extrémité sortante de e''_r et les messages UPDATER contiennent l'arc de réécriture e''_r comme sortant, et entrant respectivement.

(51) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2, tid_3, symb_3), e_u(tid_1, tid_2))\}, \{St.symb, St.tid, St.O_in_list, St.R_in_list(e_r)\})$

$\xrightarrow{O}$

$(\{OUTO(e_o(St.tid), 1)[tid_3], INITR(St.tid)[tid_3]\}, \{St.symb, St.tid, St.O_in_list + e_o(tid_3), St.R_in_list + e'_r(tid_3)\})$

si $St.symb$ est une constante et $symb_3 > St.symb$ et il n'existe pas d'arc d'orientation entre le processus de tid $St.tid$ et le processus de tid tid_3 (le tid du du processus à l'extrémité sortante de e''_r). e_o est le nouvel arc d'orientation à ajouter et e'_r est le nouvel arc de réécriture à ajouter.

(52) $(\{CONFIG(\mathrm{RUR}, e_r(tid_1), e''_r(tid_2, tid_3, symb_3), e_u(tid_1, tid_2))\}, \{St.symb, St.tid, R_in_list(e_r(tid_1))\})$

$\xrightarrow{O}$

$(\{UPDATER(e''_r(tid_3))[tid_2], UPDATER(e''_r(tid_2))[tid_3]\}, \{St.symb, St.tid, St.R_in_list + e'_r(tid_3)\})$

Même conditions que dans la règle (51).

Lors du traitement d'une configuration RUR, l'information d'orientation à utiliser pour orienter l'arc de réécriture à ajouter n'est pas forcément facilement calculable ou disponible. Des demandes sont alors envoyées en considérant les symboles des processus aux extrémités de l'arc d'orientation à orienter. Les deux O-transitions suivantes montrent comment procéder.

(53) $(\{CONFIG(\mathrm{RUR},e_r(tid_1),e''_r(tid_2,symb_2))\},\ \{St.symb,\ St.tid,\ St.R_in_list(e_r)\})$
$\xrightarrow{O}$
$(\{CALCULO(e_o(St.tid,St.symb),list_sons)[tid_2]\},\ \{St.symb,St.tid,$
$St.R_in_list(e_r)\})$
si $St.symb$ et $symb_2$ sont des symboles de fonction tels que $St.symb \geq symb_2$ et qu'il n'y ait pas d'arc d'orientation entre le processus de tid $St.tid$ et le processus avec tid tid_2 (tid du processus à l'extrémité sortante de e''_r). $list_sons$ est la liste des S-fils du processus de tid $St.tid$ avec leurs caractéristiques.

(54) $(\{CONFIG(\mathrm{RUR},e_r(tid_1),e''_r(tid_2,tid_3,symb_3),e_u(tid_1,tid_2))\},\ \{St.symb,\ St.tid,$
$St.S_out_list(e_s(tid,i)),\ St.R_in_list(e_r)\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid,tid_3),e'_o(tid_3,tid),i)[tid]\},\ \{St.symb,\ St.tid,\ St.S_out_list(e_s),$
$St.R_in_list(e_r)\})$
si $St.symb$ et $symb_3$ sont des symboles de fonction, $St.symb < symb_3$ et il n'y a pas d'arc d'orientation entre le processus de tid $St.tid$ et le processus de tid tid_3 (tid du processus à l'extrémité sortante de e''_r).

S.C.D. - U.H.P. NANCY 1 SCIENCES BIBLIOTHÈQUE

Les quatre O-transitions suivantes décrivent une partie du calcul d'une contrainte d'unification. Quand un arc de réécriture sortant est ajouté suite au traitement d'une configuration RUR, les contraintes d'unification des arc d'unification de sa U_list ont besoin d'être calculées. Des messages *INFOU* ou *NEEDU* sont alors émis.

(55) $(\{CONFIG(\text{RUR}, e_r(tid_1), e_r''(tid_2))\}, \{St.R_in_list(e_r(tid_1)), St.U_list(e_u(tid_3))\})$
$\xrightarrow{O}$
$(\{NEEDU[tid_3]\}, \{St.R_in_list - e_r, St.U_list(e_u)\})$
si $Constraint(e_u) = False$ et si le processus qui reçoit le message CONFIG est en charge du calcul de la contrainte d'unification de e_u et si $Needu(e_u) = False$ initialement.
Après le traitement du message, $Needu(e_u)$ devient égal à $True$.

(56) $(\{CONFIG(\text{RUR}, e_r(tid_1), e_r''(tid_2))\}, \{St.R_in_list(e_r(tid_1)), St.U_list(e_u(tid)), St.S_out_list(e_s)\})$
$\xrightarrow{O}$
$(\{INFOU(e_u, e_s)[tid]\}, \{St.R_in_list - e_r, St.U_list(e_u), St.S_out_list(e_s)\})$
si $Constraint(e_u) = False$ et le processus qui reçoit le message CONFIG est en charge du calcul de la contrainte d'unification de e_u et $Needu(e_u) = False$ initialement.
Après le traitement du message CONFIG, la valeur de $Needu(e_u)$ est $True$.

(57) $(\{CONFIG(\text{RUR}, e_r(tid_1), e_r''(tid_2, symb_2))\}, \{St.symb, St.O_out_list, St.R_in_list(e_r), St.U_list(e_u(tid))\})$
$\xrightarrow{O}$
$(\{NEEDU[tid]\}, \{St.symb, St.O_out_list + e_o(tid_2, symb_2), St.R_in_list - e_r, St.U_list(e_u)\})$
si $Constraint(e_u) = False$, le processus qui reçoit le message CONFIG est en charge du calcul de la contrainte d'unification de e_u, $Needu(e_u) = False$ initialement, $symb_2$ est une constante, $St.symb > symb_2$ et il n'existe pas d'arc d'orientation entre le processus de tid $St.tid$ et le processus de tid tid_2 (tid du processus à l'extrémité sortante de e_r''). e_o est le nouvel arc d'orientation à ajouter. Après le traitement du message CONFIG, $Needu(e_u) = True$.

(58) $(\{CONFIG(\text{RUR}, e_r(tid_1), e_r''(tid_2, symb_2))\}, \{St.symb, St.O_out_list, St.R_in_list(e_r), St.U_list(e_u(tid)), St.S_out_list(e_s)\})$
$\xrightarrow{O}$
$(\{INFOU(e_s, e_u)[tid]\}, \{St.symb, St.O_out_list + e_o(tid_2), St.R_in_list - e_r, St.U_list(e_u), St.S_out_list(e_s)\})$
si $Constraint(e_u) = False$, le processus qui reçoit le message CONFIG n'est pas en charge du calcul de la contrainte d'unification de e_u, $Needu(e_u) = False$ initialement, $symb_2$ est une constante, $St.symb > symb_2$ et il n'existe pas d'arc d'orientation entre le processus de tid $St.tid$ et le processus de tid tid_2 (tid du processus à l'extrémité sortante de e_r'').

Des O-transitions similaires, que nous n'exprimons pas ici, existent pour le traitement de configurations RUR-rhs.

Les quatre O-transitions suivantes concernent le calcul de l'orientation dû au traitement de configurations SUR. Des demandes d'orientation *REQUESTO1* et *REQUESTO2* sont faites via le nouvel arc de sous-terme ajouté en fonction des contraintes d'orientation à calculer sauvées dans la liste calculo_list.

(59) $(\{CONFIG(\text{SUR},e_s(tid,i),e_r(tid_1,symb_1))\}, \{St.symb, St.tid, St.S_out_list(e_s(i,tid)), St.calculo_list(e_o(tid_2,symb_2))\})$

$\xrightarrow{O}$

$(\{REQUESTO1(e_o(St.tid,St.symb,tid_2,symb_2),e'_o(tid_1,tid_2),i)[tid]\}, \{St.symb,St.tid, St.S_out_list - e_s + e'_s(tid_1,symb_1), St.calculo_list(e_o)\})$

si $St.symb = symb_2$ ou $St.symb < symb_2$. La contrainte d'orientation de l'arc e_o doit être calculée. e'_s est le nouvel arc de sous-terme à ajouter.

(60) $(\{CONFIG(\text{SUR},e_s(i,tid),e_r(tid_1,symb_1))\}, \{St.symb, St.tid, St.S_out_list(e_s(i,tid)), St.calculo_list(e_o(tid_2,symb_2),i,tidsontid_2,symbsontid_2)\})$

$\xrightarrow{O}$

$(\{REQUESTO2(e_o(St.tid,St.symb,tid_2,symb_2),e'_o(tid_1,tidsontid_2,symbsontid_2),i)[tid]\}, St.symb, St.tid, St.S_out_list - e_s + e'_s(tid_1), St.calculo_list(e_o,i,tidsontid_2,symbsontid_2)\})$

si $St.symb = symb_2$. La contrainte d'orientation de l'arc e_o doit être calculée. e'_s est le nouvel arc de sous-terme à ajouter. $tidsontid_2$ et $symbsontid_2$ sont le tid et le symbole d'un S-fils du processus de tid tid_2.

(61) $(\{INITS(e_s(tid,i))\},\{St.symb,St.tid, St.calculo_list(calculo(e_o(St.tid,St.symb,tid_1,symb_1))), St.S_out_list\})$

$\xrightarrow{O}$

$(\{REQUESTO1(e_o(St.tid,St.symb,tid_1,symb_1),e'_o(tid,tidsontid_1,symbsontid_1),i)[tid]\}, St.symb,St.tid, St.calculo_list(calculo), St.S_out_list + e_s\})$

où e_o est l'arc d'orientation entre le processus de tid St.tid et le processus de tid tid_1.

(62) $(\{INITS(e_s(tid,i))\}, \{St.symb, St.tid, St.calculo_list(calculo(e_o(St.tid,St.symb,tid_1,symb_1), i,tidsontid_1,symbsontid_1)), St.S_out_list\})$

$\xrightarrow{O}$

$(\{REQUESTO2(e_o(St.tid,St.symb,tid_1,symb_1),e'_o(tid,tidsontid_1,symbsontid_1),i)[tid]\}, \{St.symb,St.tid, St.calculo_list(calculo), St.S_out_list + e_s\})$

où e_o est un arc d'orientation entre le processus de tid St.tid et le processus de tid tid_1. $tidsontid_1$ (respectivement $symbsontid_1$) est le tid (respectivement le symbole) du i^{ime} S-fils du processus de tid tid_1.

Les six O-transitions suivantes indiquent comment l'addition d'un arc de sous-terme entrant cause l'envoi de réponses d'orientation. Les demandes sauvées ayant une réponse sont alors satisfaites et des réponses sont envoyées.

(63) $(\{INITS(e_s(tid,i))\}, \{St.symb, St.tid, St.requesto1_list(requesto_1(e_o(St.tid,tid_1), e'_o(tid,tid_1),i)), St.O_in_list(e_o), St.S_in_list\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o,e'_o,i)[tid]\}, \{St.symb, St.tid, St.requesto1_list(requesto_1), St.O_in_list(e_o), St.S_in_list + e_s\})$

(64) $(\{INITS(e_s(tid,i))\}, \{St.symb, St.tid, St.requesto1_list(requesto_1(e_o(St.tid,tid_1,symb_1), e'_o(tid,tid_1),i)), St.O_in_list(e_o), St.S_in_list\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o,e'_o,i)[tid]\}, \{St.symb, St.tid, St.requesto1_list(requesto_1), St.O_in_list(e_o), St.S_in_list + e_s\})$
si $symb_1 \geq St.symb$ et $St.symb$ est une constante.

(65) $(\{INITS(e_s(tid,i))\}, \{St.symb, St.tid, St.requesto2_list(requesto_2(e_o(St.tid,tidsontid_1), e'_o(tid,tid_1),i)), St.O_in_list(e_o), St.S_in_list\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o,e'_o,i, >)[tid]\}, \{St.symb, St.tid, St.O_in_list(e_o), St.requesto2_list(requesto_2), St.S_out_list + s\})$
où $tidsontid_1$ est le tid du i^{ime} S-fils du processus de tid_1

(66) $(\{INITS(e_s(tid,i))\}, \{St.symb, St.tid, St.requesto2_list(requesto_2(e_o(St.tid,tidsontid_1, symbsontid_1),e'_o(tid,symb,tid_1),i)), St.O_in_list(e_o), St.S_in_list\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o,e'_o,i, >)[tid]\}, \{St.symb, St.tid, St.O_in_list(e_o), St.requesto2_list(requesto_2),St.S_out_list + s\})$
où $tidsontid_1$ (respectivement $symbsontid_2$) est le tid (respectivement le symbole) du i^{ime} S-fils du processus de tid tid_1,$symbsontid_1 > St.symb$, $St.symb$ est un symbole de constante et $symb = symb_1$.

(67) $(\{INITS(e_s(tid,i))\}, \{St.symb, St.tid, St.requesto2_list(requesto_2(e_o(St.tid,tidsontid_1, symbsontid_1),e'_o(tid,symb,tid_1),i)), St.O_eq_list(e_o), St.S_in_list\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o,e'_o,i, =)[tid]\}, \{St.symb, St.tid, St.O_eq_list(e_o), St.requesto2_list(requesto_2),St.S_out_list + s\})$
où $tidsontid_1$ (respectivement $symbsontid_1$) est le tid (respectivement le symbole) du i^{ime} S-fils du processus de tid tid_1, $symbsontid_1 > St.symb$, $St.symb$ est un symbole de constante et $symb = symb_1$.

(68) $(\{INITS(e_s(tid,i))\}, \{St.symb, St.tid, St.O_out_list(e_o(St.tid,tid))\})$
$\xrightarrow{O}$
$(\{OUTO(e_o,e'_o(tid_1,tid),i)[tid]\}, \{St.symb, St.tid, St.O_out_list(e_o)\})$

Les trois O-transitions suivantes concernent le traitement des messages *INITU*. La réception d'un message *INITU* entraîne la mise à jour de la liste O_eq_list du processus et la propagation ascendante des réponses d'orientation aux demandes sauvées par ce processus.

(69) $(\{INITU(e_u(tid))\}, \{St.O_eq_list\})$
$\xrightarrow{O}$
$(\{\},\{St.O_eq_list + e_o(tid)\})$
si $Constraint(e_u) = True$

(70) $(\{INITU(e_u(tid))\}, \{St.S_in_list(e_s(tid_1,i))\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(tid_1,tid),St.tid,i,2)[tid]\},\{St.S_in_list(e_s)\})$
si $Constraint(e_u) = True$

(71) $(\{INITU(e_u(tid))\}, \{St.tid, St.S_in_list(e_s(tid_2,i)),$
$St.requesto1_list(requesto_1(tid_1,tid_2,symb_1,symb_2,tid,i))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o(St.tid,tidsontid_1, >),e'_o(tid_2,tid_1),i, =)[tid_2]\},\{St.tid, St.S_in_list(e_s),$
$St.requesto1_list(requesto_1)\})$
où le processus de tid $tidsontid_1$ est le i^{ime} S-fils du processus de tid tid_1.

Les cinq O-transitions suivantes montrent comment sont traités les messages *CALCULO* et *CALCULO1*. Ces messages impliquent l'envoi de demandes *REQUESTO1* et *REQUESTO2*.

(72) $(\{CALCULO(calculo(e_o(tid_1,symb_1)))\}, \{St.symb, St.tid, St.S_out_list(e_s(tid,i))\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid,tid_1),e'_o(tid,tid_1),i)[tid]\}, \{St.symb,St.tid,St.S_out_list(e_s)\})$

(73) $(\{CALCULO(calculo(e_o(tid_1,symb_1)))\},\{St.symb, St.tid\})$
$\xrightarrow{O}$
$(\{CALCULO1(calculo1(e_o(St.tid,St,symb)))[tid_1]\},\{St.symb, St.tid\})$
si $St.symb = symb_1$

(74) $(\{CALCULO(calculo(e_o(St.tid,tid_1,symb_1),e_s(tid_1,tidsontid_1,symbsontid_1)))\}, \{St.symb, St.tid,St.S_out_list(e_s(tid,i))\})$
$\xrightarrow{O}$
$(\{REQUESTO2(e_o,e'_o(tid,tidsontid_1),i)[tid]\}, \{St.symb,St.tid,St.S_out_list(e_s)\})$
où $tidsontid_1$ (respectivement $symsontid_1$) est le tid (respectivement le symbole) du i^{ime} S-fils du processus de tid tid_1.

(75) $(\{CALCULO1(calculo(e_o(tid_1,symb_1)))\}, \{St.symb, St.tid, St.S_out_list(e_s(tid,i))\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid,tid_1),e'_o(tid,tid_1),i)[tid]\}, \{St.symb,St.tid,St.S_out_list(e_s)\})$

(76) $(\{CALCULO(calculo(e_o(St.tid,tid_1,symb_1),e_s(tid_1,tidsontid_1,symbsontid_1)))\}, \{St.symb, St.tid,St.S_out_list(e_s(tid,i))\})$
$\xrightarrow{O}$
$(\{REQUESTO2(e_o,e'_o(tid,tidsontid_1),i)[tid]\}, \{St.symb,St.tid,St.S_out_list(e_s)\})$
où $tidsontid_1$ (respectivement $symsontid_1$) est le tid (respectivement le symbole) du i^{ime} S-fils du processus de tid tid_1.

Les trois O-transitions suivantes décrivent le traitement d'un message *OUTO* dépendant du drapeau contenu dans le message.

(77) $(\{OUTO(e_o(tid,symb),listsons,3)\},\ \{St.symb,\ St.O_out_list,\ St.S_in_list(e_s(tid_1,i))\})$
$\xrightarrow{O}$
$(\{OUTO(e'_o(tid_1,tid),St.tid,i,2)\},\{St.symb,\ St.O_out_list + e_o,\ St.S_in_list(e_s)\})$
s'il existe un arc de sous-terme du processus qui reçoit le message OUTO à tous les processus dont les tids sont dans *list_sons* et si le nombre d'éléments de *list_sons* est égal à l'arité de *St.symb*.

(78) $(\{OUTO(e_o(St.tid,tid),tid_1,i,2)\},\ \{St.S_out_list(e_s(tid_1,i)),\ St.O_out_list,$ $St.S_in_list(e'_s(tid_2,i_2))\})$
$\xrightarrow{O}$
$(\{INO(St.tid)[tid],OUTO(e_o(tid_2,tid),St.tid,i,2)[tid_2]\},\{St.S_out_list(e_s),\ St.O_out_list,$ $St.S_in_list(e'_s)\})$

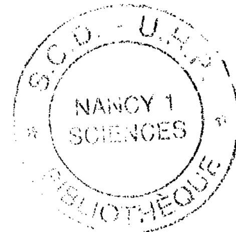
(79) $(\{OUTO(e_o(tid),1)\},\ \{St.O_out_list,\ St.S_in_list(e_s(tid_1,i))\})$
$\xrightarrow{O}$
$(\{OUTO(e'_o(tid_1,tid),St.tid,i,2)\},\{St.O_out_list + e_o,St.S_in_list(e_s)\})$

Les trois O-transitions suivantes décrivent le traitement d'un message *INO*, n particulier, la propagation ascendante des réponses quand un arc d'orientation entrant est ajouté.

(80) $(\{INO(e_o(tid_1))\},\ \{St.O_in_list\})$
$\xrightarrow{O}$
$(\{\},\{St.O_in_list + e_o(tid_1)\})$

(81) $(\{INO(e_o(tid_1))\},\ \{St.tid,\ St.requesto1_list(requesto_1(e_o(tid_1,tid_2,symb_1,symb_2),$ $e'_o(St.tid,tid_1,St.symb,St.tid),i)),\ St.S_in_list(e_s(tid_2,i))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o,e'_o,i)[tid_2]\},\{St.tid,$ $St.requesto1_list(requesto_1),\ St.S_in_list(e_s)\})$
où l'arc d'orientation e'_o est utilisé pour calculer l'arc d'orientation entre le processus de tid tid_1 et le processus de tid tid_2.

(82) $(\{INO(e_o(tid_1))\},\ \{St.tid,\ St.requesto2_list(requesto_2(e_o(tid_1,tid_2,symb_1,symb_2)),$ $e'_o(St.tid,tidsontid_1,St.symb,symbsontid_1),i),\ St.S_in_list(e_s(tid_2,i))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o,e'_o,i,>)[tid_2]\},\{St.tid,$ $St.requesto2_list(requesto_2),\ St.S_in_list(e_s)\})$
où l'arc d'orientation e'_o est utilisé pour calculer l'arc d'orientation entre le processus de tid tid_1 et le processus de tid tid_2.


NANCY 1 SCIENCES BIBLIOTHÈQUE

Les quatre O-transitions suivantes décrivent le traitement des messages *REQUESTO1*. Si le processus recevant ce message peut répondre c'est-à-dire s'il existe un arc d'orientation ou si un arc d'orientation peut facilement être ajouté, les réponses sont propagées vers le haut.

(83) $(\{REQUESTO1(requesto_1(e_o(tid_1,tid_2,symb_1,symb_2),e'_o(St.tid,tid_1,St.symb,symb_1),i))\}$, $\{St.tid,\ St.symb,\ St.O_out_list(e'_o),\ St.S_in_list(e_s(tid_2,i))\})$
$\xrightarrow{O}$
$(\{OUTO(e_o,St.tid,i,2)[tid_2]\},\ \{St.tid,\ St.symb,\ St.O_out_list(e'_o),\ St.S_in_list(e_s)\})$

(84) $(\{REQUESTO1(requesto_1(e_o(tid_1,tid_2,symb_1,symb_2),e'_o(St.tid,tid_1,St.symb,symb_1),i))\}$, $\{St.tid,St.symb,\ St.O_in_list(e'_o),\ St.S_in_list(e_s(tid_2,i))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(answero1(e_o,e'_o,i))[tid_2]\},\ \{St.tid,\ St.symb,\ St.O_in_list(e'_o),$
$St.S_in_list(e_s)\})$

(85) $(\{REQUESTO1(requesto_1(e_o(tid_1,tid_2,symb_1,symb_2),e'_o(St.tid,tid_1,St.symb,symb_1),i))\}$, $\{St.tid,St.symb,\ St.O_in_list,\ St.S_in_list(e_s(tid_2,i))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(answero1(e_o,e'_o,i))[tid_2]\},\ \{St.tid,\ St.O_in_list + e'_o,$
$St.S_in_list(e_s)\})$
si il n'y a pas d'arc d'orientation entre le processus de tid tid_1 et le processus de tid $St.tid$, $St.symb$ est une constante, $symb_1$ est un symbole de fonction ou une constante et $symb_1 > St.symb$.

(86) $(\{REQUESTO1(requesto_1(e_o(tid_1,tid_2,symb_1,symb_2),e'_o(St.tid,tid_1,St.symb,symb_1),i))\}$, $\{St.tid,St.symb,\ St.O_out_list,\ St.S_in_list(e_s(tid_2,i)),\})$
$\xrightarrow{O}$
$(\{OUTO(e_o,St.tid,i,2)[tid_2]\},\ \{St.tid,\ St.symb,\ St.O_out_list + e'_o,$
$St.S_in_list(e_s)\})$
s'il n'y a pas d'arc d'orientation entre le processus de tid $St.tid$ et le processus de tid tid_1, $symb_1$ est une constante, $St.symb$ est un symbole de fonction ou une constante et $St.symb \geq symb_1$.

Les deux O-transitions suivantes décrivent le cas où le processus recevant le message ne peut pas répondre. De nouvelles demandes d'orientation sont alors envoyées pour rendre l'information disponible.

(87) $(\{REQUESTO1(requesto_1(e_o(tid_1,tid_2,symb_1,symb_2),e'_o(St.tid,tid_1,St.symb,symb_1),i))\},$
$\{St.tid,St.symb,\ St.S_in_list(e'_s(tid_2,i)),\ St.S_out_list(e_s(tid_3,i_3))\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e'_o,e''_o(tid_1,tid_3,symb_1,symb_3),i_3)[tid_3]\},\ \{St.tid,$
$St.symb,\ St.S_in_list(e'_s),\ St.S_out_list(e_s)\})$
s'il n'y a pas d'arc d'orientation entre le processus de tid tid_1 et le processus de tid $St.tid$, $St.symb$ et $symb_1$ sont des symboles de fonction et $St.symb < symb_1$.

(88) $(\{REQUESTO1(requesto_1(e_o(tid_1,tid_2,symb_1,symb_2),e'_o(St.tid,tid_1,St.symb,symb_1),i))\},$
$\{St.tid,St.symb,\ St.S_in_list(e'_s(tid_2,i)),\ St.S_out_list(e_s(tid_3,i_3))\})$
$\xrightarrow{O}$
$(\{CALCULO(calculo(e'_o,e_s))[tid_1]\},\ \{St.tid,\ St.symb,\ St.S_in_list(e'_s),\ St.S_out_list(e_s)\})$
s'il n'y a pas d'arc d'orientation entre le processus de tid tid_1 et le processus de tid $St.tid$, $St.symb$ et $symb_1$ sont des symboles de fonction et $St.symb \geq symb_1$.

Les O-transitions sont similaires dans le cas de messages REQUESTO2. Nous ne les exprimons pas ici.

Les trois O-transitions suivantes montrent le traitement de messages *ANSWERO1*. En particulier, elles décrivent le cas 1 de l'implantation graphique de LPO.

(89) $(\{ANSWERO1(answero1(e_o(tid_1,St.tid,symb_1,St.symb),e'_o(tid_2,tid_1,symb_1,symb_2),i))\}$, $\{St.tid,St.symb,\ St.S_out_list(e_s(tid_2,i)),\ St.O_in_list\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(St.tid),1)[tid]\},\ \{St.tid,\ St.symb,\ St.S_out_list(e_s),\ St.O_in_list + e_o\})$
s'il n'existe pas d'arc d'orientation entre le processus de tid tid_1 et le processus de tid $St.tid$, si l'arc d'orientation e'_o dont on doit calculer la contrainte est entre le processus de tid tid_1 et le processus de tid $St.tid$ et le nombre de messages ANSWERO1 reçus pour le calcul de la contrainte d'orientation de l'arc d'orientation e_o est égal à l'arité de $St.symb$.

(90) $(\{ANSWERO1(answero1(e_o(tid_1,St.tid,symb_1,St.symb),e'_o(tid_2,tid_1,symb_1,symb_2),i))$, $\{St.tid,St.symb,\ St.S_out_list(e_s(tid_2,i)),\ St.S_in_list(e'_s(tid_3,i_3))$, $St.requesto1_list(requesto_1(e_o,e''_o(tid_1,tid_3,symb_1,symb_3),i_3))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o,e''_o,i_3)[tid_3]\},\{St.S_out_list(e_s),\ S_in_list(e'_s)$, $St.requesto1_list(requesto_1)\})$
Même conditions que pour la règle (89).

(91) $(\{ANSWERO1(answero1(e_o(tid_1,St.tid,symb_1,St.symb),e'_o(tid_2,tid_1,symb_1,symb_2),i))$, $\{St.tid,St.symb,\ St.S_out_list(e_s(tid_2,i)),\ St.S_in_list(e_s(tid_3,i_3))$, $St.requesto2_list(requesto_2(e_o,e''_o(tid_4,tid_3,symb_4,symb_3),i_3))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o,e''_o,i_3,>)[tid'_2]\},\{St.S_out_list(e_s),\ S_in_list(e'_s)$, $St.requesto2_list(requesto_2)\})$
Même conditions que pour la règle (89). De plus, le processus de tid tid_1 est le i_3^{ime} S-fils du processus de tid tid_4.

Les six O-transitions suivantes traitent le cas de la prise en compte de messages *ANSWERO1* et *ANSWERO2*. Elles décrivent le cas 3 de l'implantation graphique de LPO.

(92) $(\{ANSWERO2(answero2(e_o(St.tid,tid_1),e'_o(tid_2,tidsontid_1),i,>))\}, \{St.tid, St.symb, St.S_out_list(e_s(tid_2,i)), St.O_in_list\})$

$\xrightarrow{O}$

$(\{OUTO(e_o(St.tid),1)[tid_1]\},\{St.tid, St.symb, St.S_out_list(e_s), St.O_in_list + e_o\})$

si $symb_1 = St.symb$, il n'existe pas d'arc d'orientation e_o entre les processus de tid tid_1 et $St.tid$, le nombre de messages ANSWERO1 reçus est égal à l'arité du symbole $St.symb$ et le nombre des messages ANSWERO2 reçus par les k premiers S-fils du processus de tid $St.tid$ contiennent $=$ et celui reçu du $k+1^{ime}$ S-fils du processus de tid $St.tid$ contient $+$, où $k \in \{0..n-1\}$. Le processus de tid $tidsontid_1$ est le i^{ime} S-fils du processus de tid tid_1.

(93) $(\{ANSWERO2(answero2(e_o(St.tid,tid_1),e'_o(tid_2,tidsontid_1),i,>))\}, \{St.tid, St.symb, St.S_out_list(e_s(tid_2,i)), St.S_in_list(e'_s(tid_3,i_3)), St.requesto1_list(requesto_1(e_o,e''_o(tid_3,tid_1),i_3))\})$

$\xrightarrow{O}$

$(\{ANSWERO1(e_o,e''_o,i_3)[tid_3]\},\{St.tid, St.symb, St.S_out_list(e_s), St.S_in_list(e'_s), St.requesto1_list(requesto_1), St.S_in_list(e_s)\})$

Même conditions que pour la règle (92).

(94) $(\{ANSWERO2(answero2(e_o(St.tid,tid_1),e'_o(tid_2,tidsontid_1),i,>))\}, \{St.tid, St.symb, St.S_out_list(e_s(tid_2,i)), St.S_in_list(e'_s(tid_3,i_3)), St.requesto2_list(requesto_2(e_o,e''_o(tid_3,tid_5),i_3))\})$

$\xrightarrow{O}$

$(\{ANSWERO2(e_o,e''_o,i_3,>)[tid_3]\},\{St.tid, St.symb, St.S_out_list(e_s), St.S_in_list(e'_s), St.requesto2_list(requesto_2), St.S_out_list(e_s)\})$

Même conditions que pour la règle (92). De plus, le processus de tid tid_1 est le i_3^{ime} S-fils du processus de tid tid_5.

(95) $(\{ANSWERO1(answero1(e_o(St.tid,tid_1,symb_1),e'_o(tid_1,tid_2,symb_2),i))\}, \{St.tid, St.symb, St.S_out_list(e_s(tid_2,i)), St.O_in_list\})$

$\xrightarrow{O}$

$(\{OUTO(e_o(St.tid),1)[tid_1]\},\{St.tid, St.symb, St.S_out_list(e_s), St.O_in_list + e_o\})$

Même conditions que pour la règle (92).

(96) $(\{ANSWERO1(answero1(e_o(St.tid,tid_1,symb_1),e'_o(tid_1,tid_2,symb_2),i))\}, \{St.tid, St.S_out_list(e_s(tid_2,i)), St.S_in_list(e'_s(tid_3,i_3)), St.requesto1_list(requesto_1(e'_o,e''_o(tid_1,tid_3))), St.S_out_list(e_s(tid_2,i))\})$

$\xrightarrow{O}$

$(\{ANSWERO1(e'_o,e''_o(tid_3,tid_1),i_3)[tid_3]\},\{St.requesto1_list(requesto_1), St.S_out_list(e_s),S_in_list(e'_s)\})$

Même conditions que pour la règle (92).

(97) $(\{ANSWERO1(answero1(e_o(St.tid,tid_1,symb_1),e'_o(tid_1,tid_2,symb_2),i))\},$
$\{St.tid,\ St.S_out_list(e_s(tid_2,i)),\ St.S_in_list(e'_s(tid_3,i_3)),$
$St.requesto2_list(requesto_2(e'_o,e''_o(tid_3,tid_4))),\ St.S_out_list(e_s(tid_2,i))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e'_o,e''_o(tid_3,tid_4),i_3,>)[tid_3]\},\{St.requesto2_list(requesto_2),$
$St.S_out_list(e_s),S_in_list(e'_s)\})$
Même conditions que pour la règle (92). De plus, le processus de tid tid_1 est le i_3^{ime} S-fils du processus de tid_4.

(98) $(\{ANSWERO2(answero2(e_o(tid_2,tid_3),e'_o(St.tid,tid_1,St.symb,symb_1),i,=))\},\ \{St.tid,$
$St.S_out_list(e_s(tid_2,i)),\ St.O_eq_list\})$
$\xrightarrow{O}$
$(\{\},\{St.tid,\ St.S_out_list(e_s),\ St.O_eq_list + e'_o\})$
si $symb_1 = St.symb$ et le processus de tid $St.tid$ a reçu n messages ANSWERO2 contenant $=$ de ses n S-fils et n est l'arité de $St.symb$.

(99) $(\{ANSWERO2(answero2(e_o(tid_2,tid_3),e'_o(St.tid,tid_1,St.symb,symb_1),i,=))\},\ \{St.tid,$
$St.S_out_list(e_s(tid_2,i)),\ St.S_in_list(e'_s(i',tid'))\})$
$\xrightarrow{O}$
$(\{OUTO(e'_o(St.tid,tid_1),St.tid,i,2)[tid']\},\{St.tid,\ St.S_out_list(e_s),\ St.S_in_list(e'_s)\})$
Même conditions que pour la règle (98).

Correction et complétude Le calcul de l'orientation est correct et complet (voir la section 3.9).

3.7 Le datage de l'information

3.7.1 Utilité des étiquettes de temps

Dans notre approche, lorsqu'une inférence est traitée par un processus, il n'est pas nécessaire que les autres processus soient mis au courant immédiatement des conséquences de ce traitement. C'est pourquoi notre algorithme distribué est possible. Il y a toujours un délai entre l'envoi et la réception d'un message. Toutefois, nous utilisons des *étiquettes de temps* pour dater l'information contenue dans les messages pour s'assurer qu'une inférence n'est réalisée qu'en utilisant l'information la plus récente disponible.

Considérons l'exemple suivant.

Exemple 29 *Soit* $E = \{g(f(a)) \approx b,\ f(a) \approx f(b),\ a \approx c\}$ *l'ensemble à compléter. Considérons la précédence* $a \succ_g b \succ_g c \succ_g f \succ_g g$. *L'ensemble d'égalités* E *peut être représenté par le graphe* $SOUR$ *initial de la figure 3.16.*

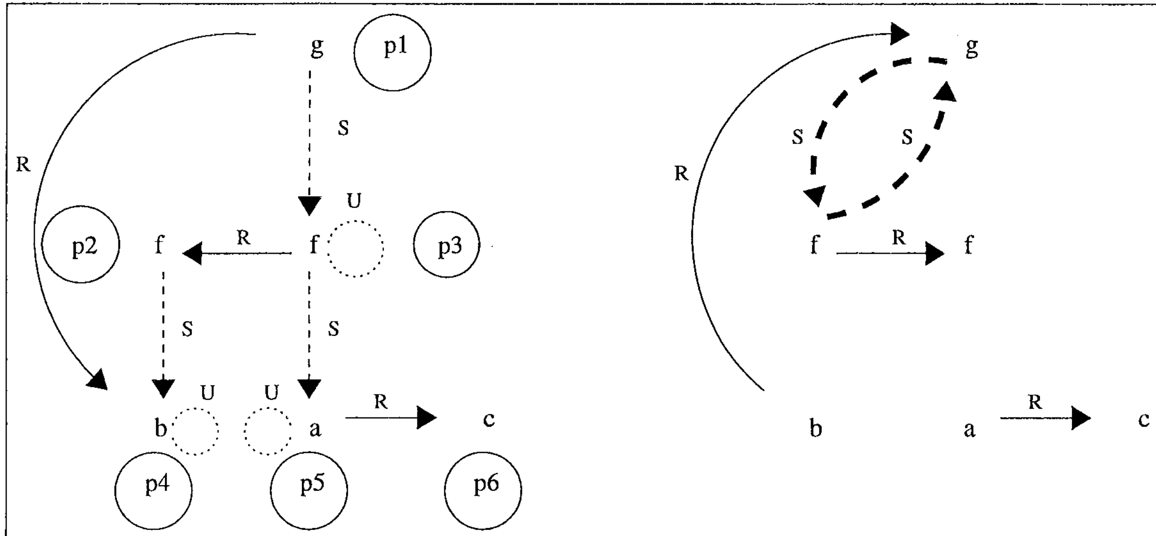


FIG. 3.16 – *Preuve de l'utilité des étiquettes de temps*

Considérons les opérations dans l'ordre suivant.

- *Le processus* p_3 *détecte la configuration* SUR $config_1$ *entre les sommets représentant les termes* $g(f(a))$, $f(a)$ *et* $f(b)$ *et l'envoie au processus* p_1.
- *Le processus* p_5 *détecte la configuration* SUR $config_2$ *entre les sommets représentant les termes* $f(a)$, a *et* c *et l'envoie au processus* p_3.
- *Le processus* p_3 *reçoit la configuration* SUR $config_2$ *et la traite. Le processus* p_3 *représente désormais le terme* $f(c)$ *et le processus* p_1 *représente le terme* $g(f(c))$.
- *Le processus* p_3 *change la direction de son arc de réécriture. Il va désormais de* p_2 *à* p_3, *car* $f(b) \succ_{lpo} f(c)$.
- *Le processus* p_1 *reçoit un message de* p_3 *pour lui dire de changer la direction de son arc de réécriture. Il va désormais de* p_4 *à* p_1, *car* $b \succ_{lpo} g(f(c))$.
- *Le processus* p_4 *détecte alors une configuration* SUR $config_3$ *entre les sommets représentant les termes* $f(b)$, b *et* $g(f(c))$ *et l'envoie à* p_2.

- *Le processus p_1 reçoit et traite la configuration SUR $config_1$. Il ajoute donc un arc de sous-terme du processus p_1 au processus p_2.*
- *Le processus p_2 reçoit et traite la configuration SUR $config_3$. Il ajoute donc un arc des sous-terme du processus p_2 au processus p_1.*

Nous notons alors la création d'un cycle d'arcs de sous-terme. Pour prévenir ce cas critique, nous utilisons des étiquettes de temps. Elles ne sont pas utilisées comme moyen de synchronisation, mais seulement pour éviter d'utiliser de l'information périmée. Dans cette exemple, lorsque p_1 traite la configuration $config_1$, cette configuration n'existe plus car l'arc de réécriture qui la compose a changé de sens.

3.7.2 Définition et utilisation des étiquettes de temps

Définition 20 *L'étiquette de temps d'un processus est un compteur initialisé à 0. Nous notons l'étiquette de temps courante d'un processus p par TS_p.* □

Tout processus p a une étiquette de temps TS_p et une table d'étiquettes de temps que nous notons TS_Table_p. Cette table a n entrées, où n est l'arité du symbole de p. Ces entrées sont initialisées à 0. Pour le processus p', $TS_Table_p[p']$ est la dernière étiquette de temps $TS_{p'}$ de p' connue par p. Quand p envoie un message à un de ses processus S-pères, il envoie son étiquette de temps courante TS_p dans le message.

Les arcs d'unification et d'orientation contiennent maintenant une nouvelle sorte d'information. Un arc d'unification entre deux processus p et p' a deux étiquettes de temps qui lui sont associées : les étiquettes de temps des processus à l'extrémité de l'arc. La même remarque s'applique pour les arcs d'orientation.

3.7.2.1 Utilisation des étiquettes de temps pour la détection et le traitement des configurations

Nous présentons dans cette section comment les étiquettes de temps sont utilisées dans la détection et le traitement des configurations.

Considérons une demi-configuration contenant un arc de réécriture e_r d'un processus p_3 à un processus p_4 et un arc d'unification e_u entre les processus p_2 et p_3.

Cette demi-configuration est envoyée à p_2 seulement si :

- le calcul de la direction de l'arc de réécriture e_r indique que e_r est de p_3 à p_4,
- le calcul de la contrainte d'unification de l'arc e_u est égale à $True$, et
- l'étiquette de temps de p_3 utilisée pour le calcul de l'unification est la même que l'étiquette de temps de p_3 utilisée pour le calcul de l'orientation.

Le message *SEMICONF* est alors envoyé à p_2 contenant l'étiquette de temps de p_2 utilisée pour calculer l'unification. Soit ts cette étiquette de temps.

Supposons que la réception du message *SEMICONF* par p_2 entraîne la détection d'une configuration. Cette configuration est envoyée à un processus p_1 seulement si $TS_{p_2} = ts$.

Si le message *CONFIG* reçu par p_1 contient une configuration SUR, p_1 traite la configuration seulement si :

- l'arc de sous-terme de la configuration du message *CONFIG* existe toujours dans la S_out_list de p_1, et

- $TS_Table_{p_1}[p_2] = ts$ ou $TS_Table_{p_1}[p_2] < ts$. Dans le cas où $TS_Table_{p_1}[p_2] < ts$, TS_{p_1} est incrémentée et $TS_Table_{p_1}$ est mis à jour tel que $TS_Table_{p_1}[p_2] = ts$. Intuitivement, TS_{p_1} est incrémenté, car le terme représentant le processus p_1 a changé.

Si le message *CONFIG* reçu par p_1 contient une configuration RUR (respectivement $RUR-rhs$), p_1 traite la configuration seulement si l'arc de réécriture entrant de p_2 à p_1 (respectivement sortant de p_1 à p_2) de la configuration du message *CONFIG* existe toujours.

3.7.2.2 Utilisation des étiquettes de temps pour le calcul de l'unification

Nous présentons dans cette section l'utilisation des étiquettes de temps dans le calcul de l'unification.

Soit un processus p_1 en charge du calcul de la contrainte d'unification d'un arc d'unification e_u entre les processus p_0 et p_1.

Supposons que le processus p_2 (k^{ime} processus S-fils de p_1) reçoive une demande $REQUESTU$ demandant s'il existe un arc d'unification avec une contrainte égale à $True$ entre les processus p_2 et p_3 (k^{ime} processus S-fils de p_0). Les réponses *ANSWERU* une fois disponibles sont envoyées à p_1, qui est en charge du calcul. Les messages de réponse *ANSWERU* contiennent les dernières étiquettes de temps de p_2 et p_3 utilisées dans le calcul de la contrainte d'unification entre p_2 et p_3, ts_2 et ts_3.

Quand le processus p_1 reçoit une réponse *ANSWERU* de p_2 contenant les étiquettes de temps ts_2 et ts_3, ce message n'est traité que si les conditions suivantes sont vérifiées :

- il y a un arc de sous-terme de p_1 à p_2 d'index k, et
- $TS_Table_{p_1}[p_2] = ts_2$ ou $TS_Table_{p_1}[p_2] < ts_2$. Dans ce dernier cas, $TS_Table_{p_1}$ est mise à jour, telle que $TS_Table_{p_1}[p_2] = ts_2$, et l'étiquette de temps courante de p_1 est incrémentée.

Si la contrainte d'unification de l'arc d'unification e_u entre p_0 et p_1 devient égale à $True$, c'est-à-dire si n messages *ANSWERU* (n étant l'arité du symbole du processus p_1) ont été reçus par p_1 concernant le calcul de la contrainte d'unification de e_u, alors un message *INITU* annonçant ce fait est envoyé à p_0. Ce message *INITU* contient l'étiquette de temps ts_3.

Quand p_0 reçoit le message *INITU* contenant l'étiquette de temps ts_3 de p_1, ce message n'est traité que si les conditions suivantes sont vérifiées :

- il y a un arc de sous-terme de p_0 à p_3 d'index k.
- $TS_Table_{p_0}[p_3] = ts_3$ ou $TS_Table_{p_0}[p_3] < ts_3$. Dans ce deuxième cas, $TS_Table_{p_0}$ est mise à jour, telle que $TS_Table_{p_0}[p_3] = ts_3$, et l'étiquette de temps courant de p_0 est incrémentée.

3.7.2.3 Utilisation des étiquettes de temps pour le calcul de l'orientation

L'utilisation des étiquettes de temps pour l'orientation est similaire à l'utilisation des étiquettes de temps pour l'unification.

Exemple 30 *Développons l'exemple 29 en utilisant des étiquettes de temps et le même ordre dans les opérations.*

- *Le processus p_3 détecte la configuration SUR $config_1$ et l'envoie au processus p_1. Le message* CONFIG *correspondant contient l'étiquette de temps $TS_{p_3}(=0)$. Soit $ts_3 = TS_{p_3}$.*
- *Le processus p_5 détecte la configuration SUR $config_2$ et l'envoie au processus p_3. Le message* CONFIG *correspondant contient l'étiquette de temps $TS_{p_5}(=0)$. Soit $ts_5 = TS_{p_5}$.*

- *Le processus* p_3 *reçoit la configuration SUR* $config_2$ *et la traite, car* $TS_Table_{p_3}[p_5] = ts_5(=0)$. *On peut noter l'étiquette de temps du processus* p_5 *ne va jamais changer. Comme* p_3 *traite une configuration SUR, son étiquette de temps est incrémentée, ainsi* $TS_{p_3} = TS_{p_3} + 1(=1)$.
- *L'arc de réécriture du processus* p_3 *change de direction. Il va maintenant de* p_2 *à* p_3, *car* $f(b) \succ_{lpo} f(c)$. p_2 *e est averti par un message* INO.
- *Le processus* p_1 *reçoit un message de* p_3 *lui indiquant que le terme qu'il représente a changé et que des arcs d'orientation sortants sont susceptibles d'avoir changé de direction. Ce message contient la dernière étiquette de temps de* p_3 *égale à* 1. *Nous avons donc* $TS_Table_{p_1}[p_3](=0) < 1$. *Ainsi, le message est traité,* $TS_Table_{p_1}[p_3] = 1$ *et* TS_{p_1} *est incrémentée de 1. La direction de l'arc de réécriture de* p_1 *change de direction il va de* p_4 *à* p_1, *car* $b \succ_{lpo} g(f(c))$.
- *Le processus* p_4 *détecte une configuration SUR* $config_3$ *qui est envoyée à* p_2. *Le message* CONFIG *correspondant est envoyé avec l'étiquette de temps* $TS_{p_4}(=0)$.
- *Le processus* p_1 *reçoit la configuration SUR* $config_1$ *contenant l'étiquette de temps* $ts_3(=0)$. *Cette configuration n'est pas traitée car l'information est périmée. En effet,* $TS_Table_{p_1}[p_3](=1) > ts_3(=0)$.
- *Le processus* p_2 *reçoit et traite la configuration SUR* $config_3$ *et ainsi, ajoute un arc de sous-terme du processus* p_2 *au processus* p_1. *Il n'apparaît ainsi pas de cycle d'arc de sous-terme. Les étiquettes de temps ont réglé ce problème.*

3.7.3 Sémantique des graphes SOUR clos avec étiquettes de temps

La sémantique des graphes SOUR clos avec étiquettes de temps est fondée sur la sémantique des termes représentés par les processus. Elle nous permet de prouver que la complétion close SOUR concurrente termine.

Définition 21 *Nous définissons la sémantique du terme représenté par le processus p par* $T(p,TS_p)$, *où* TS_p *est l'étiquette de temps du processus p.*

Nous avons : $T(p,TS_p) = f(T(p_1,ts_1),\ldots,T(p_n,ts_n))$ *où :*

- *f est le symbole du processus p.*
- *les processus* p_i *(*$i \in \{1,\ldots,n\}$*) sont les processus S-fils de p d'index i quand l'étiquette de temps de p est* TS_p.
- $ts_i = TS_Table_p[p_i]$ *est la dernière étiquette de temps de* p_i *connue par p pour* $i \in \{1,\ldots,n\}$. □

Remarque 2 *Supposons qu'à un instant donné, le processus p ait un arc de sous-terme sortant de p à* p_i, *que l'étiquette de temps de p soit* TS_p, *que* $TS_Table_p[p_i] = ts_i$ *et que l'étiquette de temps de* p_i *soit* TS_{p_i}, *alors* $ts_i \leq TS_{p_i}$. *En effet,* ts_i *est la dernière étiquette de temps de* p_i *connue par p.*

Le lemme 1 montre que le terme d'un processus est toujours remplacé par un terme plus petit.

Lemme 1 *Pour les étiquettes de temps ts et ts' telles que* $ts' < ts$, *et pour tous les processus p, nous avons :* $T(p,ts') \succ_{lpo} T(p,ts)$ *ou* $T(p,ts') = T(p,ts)$.

Preuve : Si aucune configuration SUR n'est traitée, $T(p,ts') = T(p,ts)$.

Sinon, pour prouver ce lemme, nous procédons par récurrence sur la profondeur où a lieu le traitement de configuration SUR sur le graphe SOUR.

Cas de base :

Considérons un processus p et soit $ts = TS_p$. La sémantique du terme de p est : $T(p,ts) = f(T(p_1,ts_1)), \ldots ,T(p_i,ts_i), \ldots ,T(p_n,ts_n))$, où f est le symbole de p, et pour $k \in \{1, \ldots ,n\}$, p_k est le k^{ime} processus S-fils de p et $TS_Table_p[p_k] = ts_k$.

Supposons que p reçoive une configuration *SUR* dans un message *CONFIG* et qu'il la traite. Cette configuration est décrite par un arc de sous-terme sortant du processus p au processus p_i et un arc de réécriture sortant du processus p_i au processus p_j [10].

L'étiquette de temps de p change en $TS_p = ts + 1$ suite au traitement de la configuration SUR.

La sémantique du nouveau terme de p est : $T(p,TS_p) = f(T(p_1,ts_1), \ldots ,T(p_j,ts_j), \ldots ,T(p_n,ts_n))$ où $ts_j = TS_Table_p[p_j]$.

La configuration SUR a été traitée, ainsi l'étiquette de temps contenue dans le message *CONFIG* est supérieure ou égale à $TS_Table_p[p_i]$.

Nous avons $T(p,t) \succ_{lpo} T(p_i,ts_i)$ à cause de la propriété de sous-terme et $T(p_i,ts_i) \succ_{lpo} T(p_j,ts_j)$ à cause de l'arc de réécriture.

Par définition de la sémantique d'un terme représenté par un processus et comme LPO est stable par contexte, $T(p_i,ts_i) \succ_{lpo} T(p_j,ts_j)$ implique que $T(p,TS_p) \succ_{lpo} T(p,ts)$. Ainsi, comme $ts < TS_p$ et $T(p,TS_p) \succ_{lpo} T(p,ts)$, le lemme est prouvé pour le cas de base.

Cas général :

Supposons que l'étiquette de temps de p change suite à la réception d'un message différent d'un message de type *CONFIG*. Il peut s'agir d'un message d'unification ou d'orientation. La *TS_Table* du processus p change également. La sémantique d'un sous-terme du terme représenté par le processus p a changé suite au traitement d'une configuration *SUR*, ce qui implique le changement de la sémantique du terme de p.

L'*hypothèse de récurrence* est que pour tous les processus p', tels qu'il existe une chaîne d'arcs de sous-termes de p à p', le lemme est vérifié.

Soit p_1 le processus dont le sous-terme change la sémantique du terme de p. Nous pouvons appliquer l'hypothèse de récurrence à p_1. Le terme du processus p_1 est remplacé par un terme plus petit. Ainsi, le terme de p devient plus petit, ce qui prouve le lemme. □

Le lemme précédent et la notion de persistance nous permettent de montrer que la complétion close SOUR concurrente termine.

Définition 22 *Le terme d'un processus est* persistant, *si l'étiquette de temps de ce processus est persistante.* □

Corollaire 1 *Le terme de tout processus devient persistant.*

Preuve : Soit un processus p. En utilisant le lemme 1, on a en particulier que $T(p,TS_p) \succ_{lpo} T(p,TS_p + 1)$. Comme LPO est un ordre bien-fondé, le terme de p devient plus petit et persistant. □

Théorème 5 *La complétion close SOUR concurrente termine.*

Preuve : Ce théorème est une conséquence directe du corollaire précédent. □

10. Nous considérons ici une configuration *SUR* sans arc d'unification, mais la démarche est similaire dans le cas d'une configuration *SUR* avec un arc d'unification.

3.8 Terminaison de l'algorithme distribué

La terminaison d'un programme concurrent asynchrone est difficile à déterminer, parce qu'un processus ne peut pas prédire qu'il ne va plus recevoir de messages. Nous donnons ici un algorithme de détection de la terminaison différent de l'algorithme classique présenté dans (Dijkstra, Feijen et Van Gasteren, 1983). En effet, nous n'avons pas privilégié l'utilisation de la *diffusion* (*broadcast*) pour prouver la terminaison mais plutôt une méthode par centralisation, où la terminaison est évaluée par une boîte aux lettres qui reçoit seulement des messages de terminaison. Dans la mise en oeuvre, la boîte aux lettres est gérée par le processus maître.

La définition générale de la terminaison d'un programme concurrent est donnée dans la définition 23.

Définition 23 *Un programme termine lorsque tous ses processus sont inoccupés (*idle*) et que tous les messages envoyés ont été reçus.* □

Notre algorithme de terminaison est basé sur les principes suivants.

Les messages reçus et envoyés sont notifiés à la boîte aux lettres en utilisant un message appelé *NOTIFY*. En pratique, un message peut être déclaré à la boîte aux lettres comme reçu avant d'être déclaré comme envoyé. La boîte aux lettres contient les enveloppes des messages envoyés et reçus pendant le processus de complétion.

Une enveloppe est composée :

- du tid du processus fils qui envoie le message,
- du tid du processus fils qui reçoit le message,
- d'un drapeau *SENT* ou *RECEIVED* indiquant si le message est envoyé ou reçu, et
- d'un numéro d'identification, un compteur, permettant de distinguer deux messages ayant le même processus source, le même processus destination et le même contenu.

Une enveloppe a donc l'une des deux forme suivantes : $(p_i,p_j,NB,SENT)$ pour le message de numéro NB envoyé par le processus fils p_j au processus fils p_j ou $(p_j,p_i,NB,RECEIVED)$ pour un message NB reçu par le processus fils p_i du processus fils p_j.

La boîte aux lettres consiste en deux listes :

- *list_sent* qui contient les enveloppes des messages notifiés à la boîte aux lettres comme reçus mais pas encore comme envoyés, et
- *list_received* qui contient les enveloppes des messages notifiés à la boîte aux lettres comme envoyés mais pas encore comme reçus.

Comme dans notre mise en oeuvre, la boîte aux lettres est le processus maître, nous définissons l'*état* du processus maître comme l'enregistrement composé de ces deux listes.

Les processus fils passent de l'état inoccupé (*idle*) à l'état occupé et de l'état occupé à l'état inoccupé en cycle. Un processus fils entre dans l'état occupé lorsqu'il reçoit un message. A la réception d'un message, le processus fils le traite (si les conditions sont vérifiées) et d'autres messages sont envoyés en conséquence de ce message. Dans l'état occupé, d'autres message peuvent être reçus et ces messages sont stockés dans le *tampon* des messages reçus du processus. Un processus fils entre dans l'état inoccupé lorsqu'il a fini de traiter un message. Lorsqu'il entre dans l'état inoccupé, il envoie à la boîte aux lettres un message *NOTIFY*. Le message *NOTIFY* contient une enveloppe pour le message reçu dans le cycle considéré et une enveloppe pour chaque message envoyé comme résultat de ce message reçu. Cette assertion est cruciale pour la correction de notre algorithme de terminaison. Elle sera dans la suite référencée comme *assertion (i)*.

La question principale est maintenant de détecter la terminaison. Compte tenu de la définition 23, la condition de terminaison est que *list_sent* et *list_received* soient vides.

Quand la boîte aux lettres reçoit un message *NOTIFY* contenant une enveloppe $(p_i,p_j,NB,SENT)$, l'enveloppe associée $(p_j,p_i,NB,RECEIVED)$ est cherchée dans la liste *list_received.* Si l'enveloppe $(p_j,p_i,NB,RECEIVED)$ n'est pas dans cette liste, $(p_i,p_j,NB,SENT)$ est ajoutée à *list_sent,* sinon $(p_j,p_i,NB,RECEIVED)$ est enlevée de *list_received.*

Initialement, *list_sent* contient toutes les enveloppes des messages envoyés de la phase d'initialisation. Cette hypothèse est importante pour garantir la correction de notre algorithme de terminaison. Elle sera référencée dans la suite comme *assertion (ii).*

Notre algorithme de terminaison peut être exprimé en utilisant des *T-transitions.*

Définition 24 *Une* T-transition *est une α-transition pour le calcul de la terminaison concernant la boîte aux lettres (le processus maître).* □

(100) $(\{NOTIFY((p_i,St.tid,NB,RECEIVED),(St.tid,p_j,NB',SENT),(St.tid,p_k,NB'',SENT),\ldots)\}, \{St.list_received((St.tid,p_j,NB',RECEIVED)), St.list_sent((p_i,St.tid,NB,SENT))\})$
$\xrightarrow{T}$
$(\{\},\{St.list_received - (St.tid,p_j,NB',RECEIVED), St.list_sent + (St.tid,p_k,NB'',SENT) - (St.tid,p_j,NB',SENT) - \ldots\})$
où St est l'état de la boîte aux lettres et p_i, p_j et p_k sont les tids de processus.

Dans la suite, nous nous consacrons à prouver la *correction* de notre algorithme de terminaison, c'est-à-dire que le programme termine si et seulement si *list_sent* et *list_received* sont vides.

Le passage des messages entre les processus fils peut être vu comme un arbre tel que chaque noeud soit un processus fils et que chaque arc représente le passage d'un message entre ces deux processus fils. Les arcs sont étiquetés par les noms des messages. La racine de cet arbre est un processus particulier que nous appelons le *processus principal* et qui n'apparaît qu'une seule fois dans cet arbre. Si le programme termine, cet arbre est fini sinon il est infini. Sur cet arbre, nous définissons une mesure μ. Pour un message *mesg* de cet arbre, $\mu(mesg)$ est la profondeur du noeud qui représente le processus p qui reçoit *mesg.* Par exemple, pour un message de la phase d'initialisation *mesginit*, $\mu(mesginit) = 1$.

Le lemme 2 fait le lien entre l'arbre représentant le passage des messages entre les processus fils et notre algorithme de terminaison.

Lemme 2 *Pour tout n, si* list_sent *et* list_received *contiennent seulement des messages mesg tels que $\mu(mesg) \geq n$, alors tous les messages mesg tels que $\mu(mesg) < n$ ont déjà été traités par la boîte aux lettres.*

Preuve : Pour prouver ce lemme, nous procédons par récurrence sur n.

n=2:

Supposons que *list_sent* et *list_received* ne contiennent que des messages *mesg* tels que $\mu(mesg) \geq 2$.

list_sent et *list_received* ont déjà contenus tous les messages de la phase d'initialisation *mesginit* tels que $\mu(mesginit) = 1$ (*assertion (ii)*). En effet, initialement, *list_sent* et

list_ received contiennent toutes les enveloppes des messages envoyés de la phase d'initialisation. Ainsi, tous les messages de la phase d'initialisation ont été spécifiés comme envoyés et reçus à la boîte aux lettres.
Et grâce à l'assertion (i), nous pouvons conclure que tous les messages *mesg* tels que $\mu(mesg) = 2$ ont été notifiés à la boîte au lettre comme envoyés. Donc, les messages avec une μ-mesure inférieure à 2 ont déjà été traités par la boîte aux lettres.
n>2:
L'hypothèse de récurrence est que : si *list_ sent* et *list_ received* contiennent des messages *mesg* tels que $\mu(mesg) \geq n$, alors tous les messages *mesg* tels que $\mu(mesg) < n$ ont déjà été traité par la boîte aux lettres.
Nous supposons que *list_ sent* et *list_ received* contiennent seulement des messages *mesg* tels que $\mu(mesg) \geq n + 1$ et nous allons prouver que tous les messages *mesg* tels que $\mu(mesg) < n + 1$ ont déjà été traités par la boîte aux lettres.
Si *list_ sent* et *list_ received* contiennent seulement des messages *mesg* tels que $\mu(mesg) \geq n + 1$ (donc tels que $\mu(mesg) \geq n$), alors par l'hypothèse de récurrence, tous les messages *mesg* tels que $\mu(mesg) < n$ ont déjà été traités par la boîte aux lettres.
De plus, si *list_ sent* et *list_ received* contiennent seulement des messages *mesg* tels que $\mu(mesg) \geq n + 1$, alors *list_ sent* et *list_ received* ont déjà traité tous les messages *mesg* tels que $\mu(mesg) = n$. Donc, tous les messages *mesg* envoyés tels que $\mu(mesg) = n$ sont passés dans la boîte aux lettres comme envoyés et reçus et grâce à l'*assertion (i)*, tous les messages envoyés avec une μ-mesure égale à $n + 1$ sont déjà apparus dans la boîte aux lettres. Tous les messages *mesg* tels que $\mu(mesg) < n + 1$ ont donc déjà été traités par la boîte aux lettres. □

Le théorème 6 prouve la correction de notre algorithme de terminaison.

Théorème 6 *Le programme termine si et seulement si* list_sent *et* list_received *sont vides.*

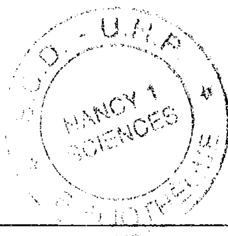
Preuve : [⇒] Il est trivial de voir que si le programme est terminé, alors *list_ sent* et *list_ received* sont vides, parce que tous les messages ont été envoyés et reçus.
[⇐] L'autre implication n'est pas triviale.
Nous procédons par contradiction. Supposons que *list_ sent* et *list_ received* sont vides mais que le programme ne termine pas. Nous considérons alors deux cas :

- Supposons qu'il y ait un message *mesg* dans le réseau tel que $\mu(mesg) = n$. En considérant le lemme 2, tous les messages tels que $\mu(mesg) = n$ ont déjà été traités. Nous rencontrons une contradiction.
- Supposons qu'il y ait un processus occupé. A la fin de son état d'occupation ce processus envoie un message *NOTIFY* à la boîte aux lettres. Supposons que ce message *NOTIFY* contienne un message *mesg* tel que $\mu(mesg) = n$. En considérant le lemme 2, tous les messages tels que $\mu(mesg) = n$ ont déjà été traités. Nous rencontrons de nouveau une contradiction.
- Les contradictions prouvent que si *list_ sent* et *list_ received* sont vides, le programme termine.

□

3.9 Résultats de correction et complétude

Dans cette section, nous prouvons la correction puis la complétude de nos calculs concurrents de l'unification et de l'orientation et de notre complétion close concurrente des graphes SOUR,



après avoir défini ces notions.

Dans la suite, E est l'ensemble d'égalités closes à compléter. Le graphe SOUR initial est construit à partir de E et détermine le nombre de processus fils impliqués dans la complétion concurrente.

3.9.1 Résultats de correction

Les définitions de correction données dans cette section sont basées sur la notion de *graphe SOUR concurrent correct.*

Définition 25 *Un* graphe SOUR concurrent *est l'ensemble des processus fils étiquetant les noeuds de ce graphe et leurs états.* □

Définition 26 *Un graphe SOUR concurrent est dit* correct *si les conditions suivantes sont vérifiées :*

- *il existe un arc d'unification de contrainte d'unification égale à True ou un arc de réécriture entre deux processus de termes s et t, alors $s =_E t$, et*
- *aucun arc de sous-terme n'est ajouté formant un cycle d'arcs de sous-termes.* □

Définition 27 *Une* dérivation de graphes SOUR concurrents *est une suite (finie ou infinie) $G_0, G_1, \ldots,$ telle que G_0 soit un graphe SOUR initial représentant un ensemble d'égalités E, G_i soit un graphe SOUR concurrent pour tout $i \in \{0, \ldots, n\}$ et G_{i+1} soit obtenu à partir de G_i par le traitement d'une configuration de G_i pour tout $i \in \{0, \ldots, n-1\}$.* □

On peut remarquer que dans le cas clos, toute dérivation de graphes SOUR concurrents est finie.

Nous définissons dans les définitions 28 et 29 ce que signifie la notion de *correction* dans le cas de notre calcul concurrent de l'unification et de la complétion close concurrente des graphes SOUR.

Définition 28 *L'unification concurrente est* correcte, *si pour toute dérivation $G_0, \ldots, G_n$ de graphes SOUR concurrents, pour tout $i \in \{0, \ldots, n\}$ et pour tous les arcs d'unifications de G_i entre des processus représentant les termes s et t avec une contrainte d'unification égale à True, nous avons $s =_E t$.* □

Définition 29 *La complétion close concurrente des graphes SOUR concurrents est* correcte *si pour toute dérivation $G_0, \ldots, G_n$ de graphes SOUR, pour tout $i \in \{0, \ldots, n\}$, G_i est un graphe SOUR concurrent correct.* □

Le lemme 3 montre que la propriété de correction des graphes SOUR clos concurrents est préservée si l'on traite une configuration.

Lemme 3 *Partant d'un graphe SOUR concurrent correct G, si un des processus de G traite une configuration, alors le graphe SOUR résultant est un graphe SOUR concurrent correct.*

Preuve : Pour prouver ce lemme, nous devons prouver que les propriétés des graphes SOUR concurrents énoncées dans la définition 26 sont vérifiées pour le graphe obtenu après traitement d'une configuration du graphe G.

1. Nous prouvons d'abord que, si un arc de réécriture est ajouté entre le processus fils p représentant le terme s et le processus fils p' représentant le terme t sur le graphe SOUR concurrent correct G, alors $s =_E t$.

Un arc de réécriture est ajouté, lorsqu'une configuration *RUR* ou *RUR-rhs* est traitée. Nous considérons ces deux cas.
– Si une configuration *RUR-rhs* est traitée, c'est-à-dire, si à un certain « moment », le schéma de la figure 3.17 se produit, un arc de réécriture est ajouté entre le processus de terme s et le processus de terme t. Nous devons prouver que $s =_E t$. Comme G est un graphe SOUR concurrent correct, alors $s =_E v$, $v =_E v'$ et $v' =_E t$. Par transitivité, nous obtenons $s =_E t$.

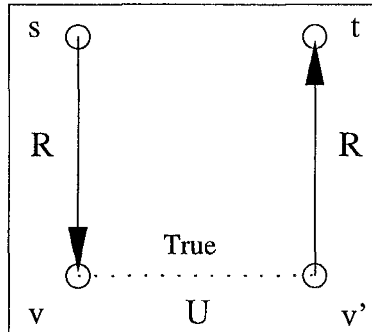


FIG. 3.17 – *Correction : Traitement d'une configuration RUR-rhs*

– Si une configuration *RUR* est traitée, c'est-à-dire, si à un certain « moment », le schéma de la figure 3.18 se produit, un arc de réécriture est ajouté entre le processus de terme s et le processus de terme t. L'orientation de cet arc dépend de la relation d'ordre entre s et t, si $s \succ_{lpo} t$ ou si $t \succ_{lpo} s$. Nous devons prouver que $s =_E t$. Comme G est un graphe SOUR concurrent correct, nous avons $v =_E s$, $v' =_E t$ et $v' =_E t$. Par transitivité, nous obtenons $s =_E t$.

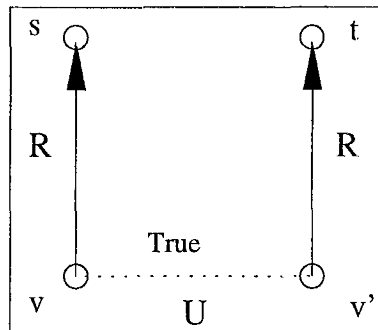


FIG. 3.18 – *Correction : Traitement d'une configuration RUR*

2. Nous prouvons maintenant que si le terme d'un processus fils p change en s' suite à l'ajout d'un arc de sous-terme et s'il y a un arc de réécriture entre le processus p et le processus de terme t alors $s' =_E t$.
   Si une configuration *SUR* est traitée, c'est-à-dire si à un certain « moment », le schéma de la figure 3.19 se produit, un arc de sous-terme est ajouté entre le processus de terme w' et le processus de terme w. Nous devons prouver que $s[w'[w]] =_E t$.
   Comme G est un graphe SOUR clos concurrent correct, nous avons $v' =_E w$, $s[w'[v]] =_E t$ et $v' =_E v'$. Par transitivité, nous obtenons $s[w'[w]] =_E t$.

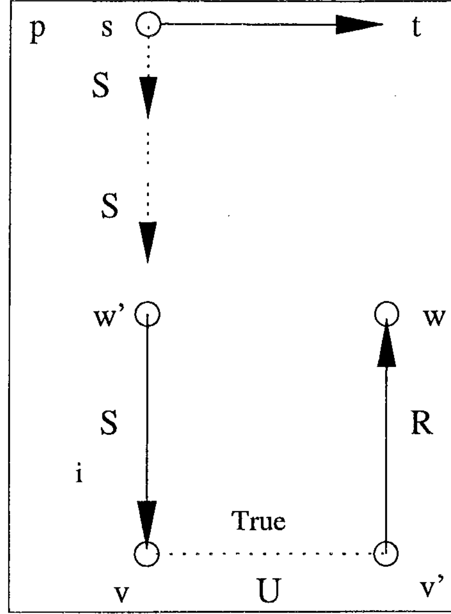


FIG. 3.19 – *Correction : Traitement d'une configuration SUR*

3. On montre comme précédemment que si le terme d'un processus fils p change en s' suite à l'ajout d'un arc de sous-terme et s'il y a un arc d'unification entre le processus p et le processus de terme t alors $s' =_E t$.
4. Nous prouvons maintenant que si un arc d'unification avec une contrainte d'unification égale à $True$ est ajouté entre deux processus fils de termes s et t sur le graphe SOUR concurrent correct G, alors $s =_E t$.

   Supposons qu'il existe un arc d'unification avec une contrainte d'unification égale à $True$ entre le processus de terme $f(s_1,\ldots,s_n)$ et le processus de terme $f(t_1,\ldots,t_n)$. Un de ces processus a l'information qu'il y a des arcs d'unification de contrainte d'unification $True$ entre s_i et t_i pour $i \in \{1,\ldots,n\}$ (voir les U-transitions (28) à (31)). Comme G est correct, $s_i =_E t_i$ pour $i \in \{1,\ldots,n\}$, donc $f(s_1,\ldots,s_n) =_E f(t_1,\ldots,t_n)$ par la propriété de congruence de $=_E$.
5. Nous prouvons maintenant qu'aucun arc de sous-terme n'est ajouté sur le graphe SOUR clos concurrent correct G tel qu'un cycle d'arcs de sous-terme soit créé. Nous nous basons sur la figure 3.20. En effet, le traitement d'une configuration *SUR* est le seul moyen de créer un cycle d'arcs de sous-terme. Ici, nous considérons le cas où la configuration *SUR* ne contient pas d'arc d'unification mais la preuve est similaire lorsque la configuration *SUR* en contient un.

   En utilisant la figure 3.20, nous allons prouver qu'il est impossible d'ajouter un arc de sous-terme de p_2 à p_n.

   Dans la figure 3.20, TS_{p_n}, $TS_{p_{n-1}}$, ..., TS_{p_2} et TS_{p_1} sont les étiquettes de temps des processus p_n, p_{n-1}, ..., p_2 et p_1 dans le graphe SOUR clos concurrent correct G.

   Soit $t_1 = TS_Table_{p_2}[p_1], \ldots, t_{n-2} = TS_Table_{p_{n-1}}[p_{n-2}]$, $t_{n-1} = TS_Table_{p_n}[p_{n-1}]$.

   Sur la figure 3.20, nous avons : $T(p_n,t_n) \succ_{lpo} T(p_{n-1},t_{n-1}) \succ_{lpo} \cdots \succ_{lpo} T(p_2,t_{p_2}) \succ_{lpo} T(p_1,t_1)$ **(1)**.

   Nous prouvons que $\forall i \in \{1,\ldots,n-1\}$, $T(p_{i+1},TS_{p_{i+1}}) \succ_{lpo} T(p_i,TS_{p_i})$.

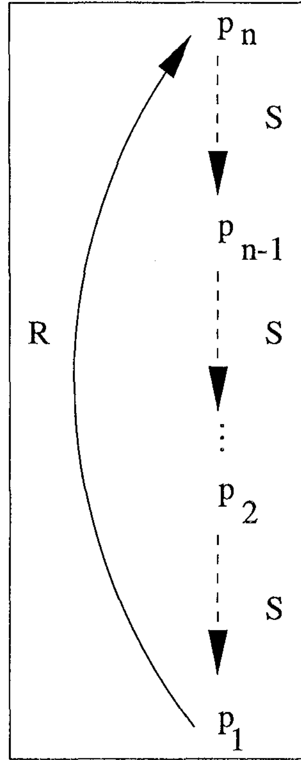


FIG. 3.20 – *Correction : Aucun cycle d'arc S n'est créé*

Nous avons $T(p_{i+1},TS_{p_{i+1}}) \succ_{lpo} T(p_i,t_i)$ par la propriété de sous-terme. De plus, en utilisant le lemme 1, comme $TS_{p_i} \geq t_i$, alors $T(p_i,t_i) \succ_{lpo} T(p_i,TS_{p_i})$ et par transitivité, $T(p_{i+1},TS_{p_{i+1}}) \succ_{lpo} T(p_i,TS_{p_i})$
Une configuration *SUR*, contenant un arc de sous-terme d'un processus p_2 à un processus p_1 et un arc de réécriture d'un processus p_1 à un processus p_n, est détectée par le processus p_1 et envoyée au processus p_2. Quand le processus p_2 reçoit cette configuration, et si il la traite, par la propriété de sous-terme, $T(p_2,TS_{p_2}) \succ_{lpo} T(p_1,t_1)$. De plus, à cause de l'arc de réécriture et si nous gelons le système, nous avons : $\exists t$, $T(p_2,TS_{p_2}) \succ_{lpo} T(p_n,t)$ tel que $t = TS_Table_{p_1}[p_n]$ quand l'étiquette de temps de p_1 est $TS_Table_{p_2}[p_1]$ et l'étiquette de temps de p_2 est TS_{p_2}. Si TS_{p_n} est l'étiquette de temps courante de p_n, alors la configuration *SUR* est reçue et $TS_{p_n} \geq t$ et en utilisant le lemme 1, nous pouvons déduire que : $T(p_n,t) \succeq_{lpo} T(p_n,TS_{p_n})$.
Comme $T(p_2,TS_{p_2}) \succ_{lpo} T(p_n,t)$ et $T(p_n,t) \succeq_{lpo} T(p_n,TS_{p_n})$, nous avons par transitivité : $T(p_2,TS_{p_2}) \succ_{lpo} T(p_n,TS_{p_n})$ **(2)**, qui est en contradiction avec **(1)**. Donc, il est impossible de créer un cycle d'arcs de sous-terme.

□

Du lemme 3, nous déduisons la correction de notre complétion close concurrente basée sur l'utilisation des graphes SOUR et la correction de notre calcul concurrent de l'unification.

Théorème 7 *La complétion close concurrente des graphes SOUR est correcte.*

Preuve : La preuve se fait par récurrence. Le cas de base est trivial, puisque l'on part

d'un graphe SOUR clos concurrent correct et l'étape de récurrence suit immédiatement le lemme 3. □

Corollaire 2 *L'unification concurrente est correcte.*

Preuve : Comme pour le théorème 7, la preuve se fait par récurrence. Le cas de base est trivial, puisque l'on part d'un graphe SOUR clos concurrent correct et l'étape de récurrence suit immédiatement le lemme 3. □

3.9.2 Résultats de complétude

Cette section contient les définitions et les preuves de complétude de l'unification concurrente, de l'orientation concurrente et de notre complétion close concurrente des graphes SOUR.

Définition 30 *L'unification concurrente est* complète, *si quand s et t sont des termes persistants et si s et t sont unifiables, alors à un certain « moment », l'arc d'unification entre les processus représentant les termes s et t a une contrainte d'unification égale à True, qui ne change plus.* □

Définition 31 *Le calcul concurrent de l'orientation est* complet, *si quand s et t sont deux termes persistants et $s \succ_{lpo} t$, alors, à un certain « moment », l'arc d'orientation entre les processus représentant les termes s et t va de s à t et ne change plus.* □

Définition 32 *La complétion close concurrente des graphes SOUR est* complète, *s'il n'y a pas de configuration entre des noeuds du graphe dont les termes sont persistants.* □

Nous prouvons par récurrence et en utilisant les U-transitions et les O-transitions que les calculs concurrents de l'unification et de l'orientation sont complets.

Théorème 8 *L'unification concurrente est complète.*

Preuve : Pour prouver la complétude de l'unification concurrente, nous procédons par induction structurelle sur des paires de termes du graphe SOUR.

- **Cas de base** Si nous considérons les termes s et t comme des constantes, le résultat est trivial.
- **Cas général** Supposons que les termes $s = f(s_1, \ldots, s_n)$ et $t = f(t_1, \ldots, t_n)$ soient persistants et unifiables. Alors, s_i et t_i sont persistants et unifiables pour $i \in \{1, \ldots, n\}$. L'hypothèse de récurrence est que pour toutes les paires de termes structurellement moins complexes que $f(s_1, \ldots, s_n)$ et $f(t_1, \ldots, t_n)$, l'unification concurrente est complète.
  D'après l'hypothèse de récurrence, il y a donc des arcs d'unification avec des contraintes d'unification égales à *True* entre s_i et t_i pour $i \in \{1, \ldots, n\}$. En passant cette information vers le haut à l'aide de messages *ANSWERU* (voir les U-transitions (28) à (31)), la contrainte de l'arc d'unification entre s et t devient égale à *True*.
  Il est clair que l'information provenant des processus S-fils ne va pas être ignorée, parce que les étiquettes de temps des processus S-fils sont persistantes. En effet, l'information des *TS_tables* des processus recevant l'information de leurs processus S-fils n'est pas en contradiction avec les étiquettes de temps des messages passés vers le haut.

□

Théorème 9 *Le calcul concurrent de l'orientation est complet.*

Preuve : Pour prouver la complétude de notre calcul concurrent de l'orientation, nous procédons par induction structurelle sur des paires de termes du graphe SOUR.

Cas de base Si nous considérons les termes s et t comme des constantes, le résultat est trivial (voir les règles de transition (64) et (85) par exemple).

Cas général L'hypothèse d'induction est que pour tout les termes structurellement moins complexes que $s = f(s_1, \dots, s_n)$ et $t = g(t_1, \dots, t_m)$, la complétude est vérifiée.

La preuve s'articule autour des trois points caractérisant l'implantation graphique de LPO.

- Supposons que les termes $s = f(s_1, \dots, s_n)$ et $t = g(t_1, \dots, t_m)$ soient persistants et que $s \succ_{lpo} t$ tels que $f \succ_p g$ et $\forall j \in \{1, \dots, m\}$, $s \succ_{lpo} t_j$.

  Chaque terme t_j est persistant pour $j \in \{1, \dots, m\}$. Par l'hypothèse d'induction, à un certain « moment » pour $j \in \{1, \dots, m\}$, l'arc d'orientation entre les processus représentant les termes s et t_j va de s à t_j. En remontant l'information (voir les règles de transition (64) et (85) par exemple), le processus représentant le terme t va connaître toute cette information et en utilisant la règle de transition (89), il va ajouter un nouvel arc d'orientation du processus représentant le terme s au processus représentant le terme t.
- Supposons que les termes $s = f(s_1, \dots, s_n)$ et $t = g(t_1, \dots, t_m)$ soient persistants et que $s \succ_{lpo} t$ tel que $\exists i \in \{1, \dots, n\}$, $s_i \succ_{lpo} t$.

  Chaque terme s_i est persistant. Par l'hypothèse d'induction, à un certain moment, l'arc d'orientation entre les processus représentant les termes s_i et t va aller de s_i à t. En remontant l'information (voir les règles de transition (63) et (78) par exemple), le processus représentant le terme s va ajouter un nouvel arc d'orientation du processus représentant le terme s au processus représentant le terme t.
- Supposons que les termes $s = f(s_1, \dots, s_n)$ et $t = g(t_1, \dots, t_m)$ soient persistants et que $s \succ_{lpo} t$ tel que $f = g$, $n = m$, $\forall i \in \{1, \dots, n\}$, $s \succ_{lpo} t_i$, et $(s_1, \dots, s_n) \succ_{lex} (t_1, \dots, t_n)$ tel que $s_j \succ_{lpo} t_j$ et $\forall k < j$, $s_k = t_k$.

  Ainsi, s_i et t_i sont persistants pour $i \in \{1, \dots, n\}$. Par l'hypothèse d'induction, à un certain moment, pour $k \in \{1, \dots, n\}$, les arcs d'orientation entre les processus représentant les termes s et t_k vont de s à t_k, les arcs d'orientation entre les processus représentant les termes s_j et t_j vont de s_j à t_j, et l'arc d'orientation entre les processus représentant les termes s_i et t_i pour $i \in \{1, \dots, j-1\}$ est un arc d'orientation "égal". En remontant l'information, (voir les règles de transitions (64), (66) et (67) par exemple), le processus représentant le terme s va connaître toute cette information et en utilisant les règles de transition (92) et (99), il va ajouter un nouvel arc d'orientation du processus représentant le terme s au processus représentant le terme t.

Il est clair que l'information des processus S-fils ne va pas être ignorée pour les même raisons que précédemment dans le théorème 8. □

complète par une preuve pas l'absurde.

Théorème 10 *La complétion close concurrente des graphes SOUR est complète.*

Preuve : Nous procédons par l'absurde. Supposons que les noeuds n_1 et n_2 aient des termes persistants, et que le noeud n_1 ait une configuration SUR à traiter. Le terme du noeud n_1 n'est alors pas persistant. La contradiction prouve la complétude. □

3.10 Expérience pratique : l'implantation CWD

Nous appelons *CWD* (*Completion Without Duplication*) le système réalisant la complétion close concurrente des graphes SOUR suivant le modèle décrit dans ce chapitre. *CWD* implante les transitions D,P,U,O et T présentées.

Nous résumons d'abord les différentes optimisations d'implantation que nous avons réalisées.

- Le nombre de messages a été limité. Il faut en effet limiter les communications, car les communications sont coûteuses en temps.
  Les messages identiques (*REQUESTU*, *REQUESTO1* et *REQUESTO2*), les configurations identiques (*CONFIG*) et les messages *INFOU* ne sont envoyés qu'une seule fois.
- L'unification et l'orientation sont évaluées par une méthode de demande-réponse, ainsi seulement l'information jugée nécessaire est demandée.
- L'unification n'est évaluée que lorsque c'est nécessaire, c'est-à-dire quand il existe un arc d'unification adjacent à un arc de réécriture sortant.
- Les messages dont l'étiquette de temps ne vérifie pas les conditions requises ne sont pas traités.

CWD est implanté en *C++* (environ 15000 lignes de C++ commentées) et en utilisant *LEDA* (Library of Efficient Data types) (Naher, 1993) et *PVM* (Parallel Virtual Machine) (Sunderam, 1990; Beguelin et al., 1994). Le système a été testé sur un réseau de stations de travail Sun4 sous solaris (Sparc5, Sparc10, Sparc20), et des stations de travail Sgi Indy (processeur R4400, 200 MHz, 64 Mb) et sur le Power Challenge Array (PCA) (8 processeurs R8000 à 90 MHz et 160 Mb). Le PCA est une machine parallèle à architecture distribuée et mémoire partagée.

CWD a été testé sur un large éventail d'exemples. Nous nous intéressons ici aux trois exemples suivants : *Counter5*, Expf6 et *Data*.

Exemple 31 *counter5 (Gallier et al., 1993)*
Ensemble d'égalités à compléter :

$$E = \begin{cases} f(c) \approx g(c)\ , \\ f(g(c)) \approx g(f(c)), \\ f(g(g(c))) \approx g(f(f(c))), \\ f(g(g(g(c)))) \approx g(f(f(f(c)))), \\ f(g(g(g(g(c))))) \approx g(f(f(f(f(c))))) \end{cases}$$

Précédence : $f \succ_g g \succ_g c$.
Nombre de processus fils (nombre de sommets du graphe SOUR initial représentant l'ensemble des égalités à compléter) : 17
*L'intérêt de cet exemple est que pour certaines stratégies de complétion il peut tourner exponentiellement longtemps par rapport à la taille des termes, même si on utilise le partage de structure (*structure sharing*).*
Le résultat de la complétion de E *est l'ensemble d'égalités* E_∞.

$$E_\infty = \begin{cases} f(c) \approx g(c) \\ f(g(c)) \approx g(g(c)) \\ f(g(g(c))) \approx g(g(g(c))) \\ f(g(g(g(c)))) \approx g(g(g(g(c)))) \\ f(g(g(g(g(c))))) \approx g(g(g(g(g(c))))) \end{cases}$$

Exemple 32 *Expf6*

$$E = \begin{cases} aa \approx f(ab,ab,ab,ab,ab,ab), \\ ab \approx f(ac,ac,ac,ac,ac,ac), \\ ac \approx f(ad,ad,ad,ad,ad,ad), \\ ad \approx f(ae,ae,ae,ae,ae,ae), \\ ae \approx f(af,af,af,af,af,af), \\ af \approx f(ag,ag,ag,ag,ag,ag), \\ f(aa,aa,aa,aa,aa,aa) \approx b \end{cases}$$

Précédence : $aa \succ_g ab \succ_g ac \succ_g ad \succ_g ae \succ_g af \succ_g ag \succ_g f \succ_g b$.

Nombre de processus fils : 15

Cet exemple est intéressant, parce que les termes résultant des égalités sont de taille exponentielle.

Sur cet exemple, OTTER-3.0 *est « hors mémoire »* (out of memory*) après 20 minutes d'exécution sur un serveur Digital avec 2 CPUs et 256 Mb de mémoire principale.* PaReDux *ne donne pas de résultat non plus à cause du nombre de cellules réclamées.*

Par contre, grâce au partage de structure, nous obtenons le résultat de ce probème. E_∞ *n'est pas fournit dans ce manuscrit pour des raisons de lisibilité.*

Exemple 33 *data*

$$E = \begin{cases} a \approx b \\ a \approx c \\ f(a) \approx d \\ g(a,a) \approx h(b,c) \\ k(a,c,d) \approx k(f(a),d,g(a,a)) \\ j(f(b)) \approx j(g(a,b)) \\ f(c) \approx g(f(a),g(b,c)) \\ l(a,b) \approx e \\ l(g(f(a),f(b)),a) \approx e \end{cases}$$

Précédence : $a \succ_p b \succ_p c \succ_p d \succ_p e \succ_p f \succ_p g \succ_p h \succ_p j \succ_p k \succ_p l$

Nombre de processus fils : 20

Le résultat de la complétion de E *est l'ensemble d'égalités* E_∞.

$$E_\infty = \begin{cases} k(d,d,h(c,c)) \approx k(c,c,d) \\ j(h(c,c)) \approx j(d) \\ l(c,c) \approx e \\ l(g(d,d),c) \approx e \\ g(c,c) \approx h(c,c) \\ a \approx c \\ b \approx c \\ f(c) \approx d \\ g(d,h(c,c)) \approx d \end{cases}$$

Des mesures variées ont été réalisées sur ces exemples. La table 3.2 résume les résultats obtenus. Elle présente pour chaque exemple et pour différentes plate-formes, le nombre total de messages de complétion envoyés entre les processus fils, le nombre total de messages *NOTIFY* pour détecter la terminaison et le temps réel d'exécution de l'exemple donné en secondes.

Problème	Plate-forme	Messages de complétion	Message de terminaison	Temps réel
Counter5	1 Sun4	582	627	13.00 s
Counter5	5 Sun4	470	514	4.00 s
Counter5	10 Sun4	648	696	3.62 s
Counter5	15 Sun4	616	663	3.23 s
Counter5	5 Sgi	500	545	4.77 s
Counter5	PCA	594	639	1.88 s
Expf6	1 Sun4	724	822	11.00 s
Expf6	5 Sun4	737	835	5.52 s
Expf6	10 Sun4	734	832	3.00 s
Expf6	15 Sun4	729	827	3.00 s
Expf6	5 Sgi	715	813	4.56 s
Expf6	PCA	722	820	2.32 s
Data	1 Sun4	1023	1099	10s
Data	5 Sun4	850	925	5s
Data	10 Sun4	937	1012	4s
Data	15 Sun4	893	968	3s
Data	5 Sgi	889	964	7s
Data	PCA	975	1051	7s

Table 3.2 – *Expérience pratique avec CWD*

Par lecture de la table 3.2, nous pouvons voir que les résultats sont meilleurs quand on exécute *CWD* sur le *PCA*. Ceci est dû à l'architecture du *PCA*. L'architecture du *PCA* est une architecture distribuée avec un composant de mémoire partagée. Le passage de messages est implanté à travers la mémoire partagée. Dans le cas des réseaux de stations de travail SUN4 et SGI, les accélérations (*speeds ups*) sont entre 3 et 4. Les accélérations dépendent des exemples, de leur séquentialité et des charges de communication. Comme les graphes SOUR sont adaptés à une approche concurrente, nous pensons que l'implantation peut être améliorée. En particulier, le nombre de messages de terminaison peut être réduit en raffinant l'algorithme de terminaison et pour équilibrer la charge de travail entre les processus, une possibilité est de faire faire un travail de fond aux processus fils.

3.11 Conclusion

Nous avons présenté un algorithme distribué pour la complétion close des graphes SOUR.

Dans notre approche, chaque noeud du graphe SOUR initial représentant l'ensemble des égalités à compléter est un processus représentant un terme. Notre approche est donc de grain fin. Le nombre des processus est fixe, il est égal au nombre de sommets du graphe SOUR initial. A notre connaissance, il s'agit de la seule complétion concurrente et procédure concurrente de preuve automatique de grain fin, dans le sens où un processus représente un terme. Les arcs sont des canaux de communication entre processus, qui schématisent des relations de sous-terme (arc S), de réécriture (arc R), d'unification (arc U) et d'orientation (arc O). Les processus opèrent totalement indépendamment les uns des autres et de manière asynchrone. Nous n'avons pas besoin de mémoire globale ou de contrôle global comme dans la plupart des approches de la littérature, notre algorithme fonctionne donc de manière locale et nous pensons que c'est un élément important. Aucune vérification globale de consistance de l'information reçue par les

processus n'est nécessaire. Deux processus ne réalisent pas le même travail, il n'y a donc pas de travail redondant. Les stratégies basées contraction sont les plus efficaces, mais elles sont les plus difficiles à paralléliser. Leur parallélisation demande presque toujours un contrôle global ou une mémoire globale pour que l'information de simplification soit propagée. Toutefois, notre approche utilise une stratégie de contraction sans mémoire globale et sans contrôle global. Nous pensons que cette approche est fondamentalement nouvelle, car il n'avait jamais été imaginé de combiner le grain fin et des opérations complètement locales, et que de plus cette approche est prometteuse.

Nous disposons grâce à notre approche et de façon indépendante de différents algorithmes distribués : un algorithme qui réalise la complétion des graphes SOUR, un algorithme basé sur l'utilisation des graphes SOUR permettant de déterminer si deux termes sont unifiables (égaux), un algorithme basé sur l'utilisation des graphes SOUR et LPO permettant de déterminer si un terme est plus, grand plus petit ou égal à un autre terme et un algorithme pour détecter la terminaison d'un programme concurrent où les processus ne travaille que par réception de messages et de manière asynchrone.

Nous avons validé notre approche par une implantation, *CWD*, écrite en *C++* et *PVM*. *CWD* tourne aussi bien sur un réseau de stations de travails que sur le Power Challenge Array (PCA), qui est une machine parallèle à architecture distribuée et mémoire partagée. Cette implantation nous a permis de nous confronter aux difficultés et aux efforts à fournir pour développer et déboguer des applications parallèles. L'implantation peut être améliorée, le nombre de messages de terminaison peut être réduit en raffinant l'algorithme de terminaison et pour équilibrer la charge de travail entre les processus, une possibilité est de faire faire un travail de fond aux processus fils.

Nous nous sommes focalisés dans ce chapitre sur le cas clos. Mais, les principes généraux de l'algorithme distribué pour la complétion close des graphes SOUR restent valables dans le cas non clos. Dans le cas non clos, les arcs des graphes SOUR sont étiquetés par des contraintes et des substitutions de renommage. La détection et le traitement des configurations, les calculs d'unification et d'orientation doivent prendre en compte ces nouveaux éléments. Par exemple, les messages d'unification ne contiennent plus la contrainte équationnelle triviale *True*, mais des contraintes équationnelles satisfaisable. Les messages *CONFIG* contiennent beaucoup plus d'information. Toutefois, nous avons abandonné l'idée d'étendre le cas clos au cas non clos pour deux raisons. Le cas non clos est très complexe séquentiellement à cause de l'utilisation forcée de renommage et de plus, aucune règle de contraction n'a été mise au point pour la complétion basique des graphes SOUR non clos.

Chapitre 4

Complétion basique avec simplification E-cycle

Sommaire

4.1 Introduction **151**
4.2 Concepts de base **153**
4.2.1 Contraintes de réductibilité 153
4.2.2 Graphe de dépendance 157
4.2.3 E-chemins et E-cycles 157
4.3 Graphe de dépendance et simplification E-cycle **158**
4.3.1 Le système d'inférence BCES 158
4.3.2 Les transitions de graphes 160
4.3.3 Propriétés des graphes de dépendance 166
4.3.4 Un exemple 167
4.4 BCES est complet **178**
4.4.1 Graphe de dépendance clos 178
4.4.2 Preuve de complétude 181
4.5 Comparaison avec la simplification basique **186**
4.5.1 La simplification E-cycle n'est pas locale 186
4.5.2 La simplification basique est contenue dans la simplification E-cycle . . 187
4.6 Expérience pratique en ELAN : l'implantation ECC **189**
4.7 Stratégies de simplification incomplètes **193**
4.7.1 Stratégies S_1, S_2 et S_3 194
4.7.2 Incomplétude de la stratégie S_1 194
4.7.3 Incomplétude de la stratégie S_2 194
4.7.4 Incomplétude de la stratégie S_3 195
4.8 Conclusion **196**

4.1 Introduction

Les stratégies de contraction sont cruciales en recherche de preuve et en déduction. Elles permettent de supprimer des formules de l'espace de recherche. Toutefois, elles posent le problème de la complétude du système d'inférence résultant de l'addition des règles de contraction. Il est important de savoir si ce système d'inférence est complet ou non. En effet, la complétude d'un

système d'inférence garantit de trouver une preuve si elle existe. Si aucune preuve n'a été trouvée, alors on peut en conclure que la formule est fausse.

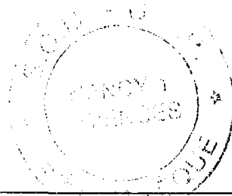
La complétude est une propriété importante qui a un coût, notamment si l'on considère la taille de l'espace de recherche. C'est pourquoi, l'utilisation de stratégies de contraction incomplètes est souvent privilégiée en pratique pour résoudre des problèmes ouverts. L'espace de recherche est alors « moins grand ». Dans (Bachmair et al., 1992), OTTER (McCune, 1994) est modifié pour réaliser de la paramodulation basique. La stratégie de contraction incomplète utilisée est la *simplification standard*. Bien que le système d'inférence composé de la paramodulation basique et de la simplification standard soit incomplet, tous les théorèmes donnés en entrée ont été prouvés. Plus récemment, W. McCune a développé un prouveur de théorèmes, *EQP* (McCune, 1997a), utilisant une stratégie de contraction incomplète, qui a résolu le problème de Robbins concernant les algèbres booléens (Kolata, 1996; McCune, 1997c). Outre la stratégie de simplification, l'utilisation de la paramodulation basique est le point clé dans la résolution de ce problème (McCune, 1997b).

Nous nous intéressons dans ce chapitre à la complétion basique et à la complétude des systèmes d'inférence basés sur la complétion basique. La complétion basique a été prouvée complète dans (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995). A. Rubio ont montré que la complétion basique avec *simplification standard* est incomplète (Nieuwenhuis et Rubio, 1992a). La complétion basique avec *blocage standard* (*standard blocking*) est également incomplète (Bachmair et al., 1995) (voir également la section 2.3.4.3). chapitre 4.7 de cette partie, d'autres simplification incomplètes pour la complétion données. La seule stratégie de simplification complète en combinaison avec la complétion basique connue jusqu'à présent était la *simplification basique* (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995). Ces résultats de complétude nous ont amenés à analyser la combinaison de la complétion basique et des stratégies de contraction, le but étant de comprendre quelles contractions détruisent la complétude et quelles contractions ne la détruisent pas. Cette analyse a ainsi entraîné la mise au point d'une nouvelle stratégie de simplification complète pour la complétion basique, la *simplification E-cycle* et du système d'inférence *BCES* (*Complétion Basique avec Simplification E-cycle*).

La simplification E-cycle est basée sur la construction d'un graphe de dépendance pendant la complétion. Le graphe de dépendance est un graphe étiqueté et orienté. Il nous permet de connaître les égalités à conserver pour assurer la complétude du système d'inférence. Les noeuds de ce graphe sont des égalités. Les arcs représentent les dépendances entre les égalités et déterminent également les ancêtres d'une égalité. Quand une nouvelle égalité est déduite lors d'une inférence, un nouveau noeud et de nouveaux arcs sont ajoutés. On associe des contraintes, appelée contraintes de réductibilité, aux arcs. Les arcs sont ainsi en conflit les uns avec les autres. Les E-chemins et les E-cycles sont des chemins et des cycles sans conflit entre les contraintes de réductibilité associées aux arcs les composant. On interdit la création de E-cycles dans le graphe de dépendance. L'apparition d'un E-cycle dans le graphe de dépendance représente une simplification non autorisée d'une égalité par un de ses ancêtres et seule une simplification peut provoquer la création d'un tel cycle.

Nous avons prouvé que la complétion basique avec cette stratégie de simplification est complète. Notre preuve est basée sur la technique de construction d'un modèle de Herbrand de l'ensemble des égalités saturé (Bachmair et Ganzinger, 1994; Bachmair et al., 1995; Nieuwenhuis et Rubio, 1992a; Nieuwenhuis et Rubio, 1995a) explicitée dans la section 2.2. Notre preuve utilise un ordre construit directement à partir du graphe de dépendance. Cet ordre est noté $\succ_g$.

Cette stratégie de simplification autorise plus de simplifications que la simplification basique, dans le sens suivant : si la simplification basique autorise une simplification alors la simplification E-cycle aussi et il existe des cas où, la simplification E-cycle permet de simplifier et la simplifica-



tion basique ne le permet pas. Nous avons ainsi montré que la simplification basique est contenue strictement dans la simplification E-cycle.

La complétion basique avec simplification E-cycle a été implantée dans le système *ECC* (*Complétion E-cycle*) en utilisant le langage ELAN. La complétion basique avec simplification basique est également implantée dans *ECC*. L'implantation nous a permis de comparer la simplification basique et la simplification E-cycle pratiquement sur des exemples standards de complétion. L'intérêt pratique de la simplification E-cycle a été mis en évidence sur ces exemples.

Les résultats de ce chapitre ont été présentés à la conférence AISC'98 (Lynch et Scharff, 1998) et vont paraître dans (Lynch et Scharff, 1999b).

4.2 Concepts de base

Nous allons, dans cette section, définir les concepts de base nécessaires dans la suite de ce travail concernant la complétion basique avec simplification E-cycle.

4.2.1 Contraintes de réductibilité

Les premières instances des contraintes de réductibilité se trouvent dans (Peterson, 1994) et dans (Lynch et Snyder, 1995). La raison technique d'existence de telles contraintes est que lors du processus de complétion, les égalités avec des contraintes réductibles peuvent être enlevées de l'espace de recherche. Les contraintes de réductibilités rendent cela explicite.

Nous définissions un symbole de prédicat *Red*, qui a deux arguments, un terme t, et une égalité close e ou le symbole $\top$. Le second paramètre de *Red* sera utilisé seulement dans la section 4.4 dans la plate-forme abstraite utilisée pour prouver la complétude de *BCES*.

Définition 33 *Soit E un ensemble d'égalités. Une* contrainte de réductibilité *est :*
- $\top$ *dénotant la conjonction vide et la contrainte de réductibilité vraie, ou*
- $\bot$ *dénotant la contrainte de réductibilité fausse, ou*
- *de la forme* $c_{r_1} \wedge \ldots \wedge c_{r_n}$*, où* c_{r_i} *est, soit de la forme* $(\bigvee_j Red(t_j,e_j))$*, soit de la forme* $\neg Red(t_j,e_j)$ *ou* $\top$*, où* $t_j \in \mathcal{T}$ *et* $e_j \in Gr(E) \cup \{\top\}$. □

Par exemple, si l'on omet le deuxième argument du prédicat *Red*, $(Red(a) \vee Red(b)) \wedge \neg Reb(a) \wedge \neg Red(c)$, où a, b et c sont des constantes, est une contrainte de réductibilité.

En ELAN, les contraintes de réductibilité sont définies par des objets de sorte *redconstraint* de la manière suivante :

```
operators
global
 T        : redconstraint;
 F        : redconstraint;
 R(@)     : (term) redconstraint;      /* R pour Red */
 nR(@)    : (term) redconstraint;      /* nR pour ¬ Red */
          /* & (et) est un opérateur AC de priorité supérieure à V */
 @ & @    : (redconstraint redconstraint) redconstraint (AC) pri 20;
          /* V (ou) est un opérateur AC */
 @ V @    : (redconstraint redconstraint) redconstraint (AC) pri 40;
end
```

Définition 34 *Une* contrainte de réductibilité close *est une contrainte de réductibilité telle que le premier paramètre du prédicat Red soit un terme clos.* □

La définition suivante définit la satisfaisabilité d'une contrainte de réductibilité close dans un système de réécriture clos R.

Définition 35 *Soit c_r une contrainte de réductibilité close, et R un système de réécriture clos. Alors, c_r est* satisfaisable dans R, *si et seulement si l'une des conditions suivantes est vraie :*

- $c_r = \top$.
- $c_r = Red(t,\top)$ *et t est réductible dans R.*
- $c_r = Red(t,e)$ *et t est réductible par une égalité e' de R telle que $e' \prec_e e$.*
- $c_r = \neg c'_r$ *et c'_r ne soit pas satisfaisable dans R.*
- $c_r = c'_r \wedge c''_r$ *et c'_r et c''_r soient satisfaisables dans R.*
- $c_r = c'_r \vee c''_r$ *et c'_r ou c''_r soit satisfaisable dans R.* □

Nous pouvons remarquer que pour tout système de réécriture clos R et pour toute égalité e close avec t plus petit que le terme maximal de e, alors $Red(t,e)$ est satisfaisable dans R, si et seulement si $Red(t,\top)$ est satisfaisable dans R. Nous abrégeons $Red(t,e)$ par $Red(t)$ dans ce cas. De plus, nous abrégeons $Red(t,\top)$ par $Red(t)$.

Définition 36 *Une contrainte de réductibilité c_r est* satisfaisable *si et seulement si il existe un système de réécriture clos R et une substitution close σ, tels que $\sigma(c_r)$ soit satisfaisable dans R.* □

La satisfaisabilité est une notion sémantique. Dans notre mécanisme de recherche de preuve, nous avons besoin d'une notion syntaxique. C'est pourquoi, nous définissons la notion de *contradiction syntaxique.*

Définition 37 *Une contrainte de réductibilité c_r est* contradictoire syntaxiquement *si et seulement si l'une des conditions suivantes est vraie :*

- $c_r = \bot$.
- *il existe des termes $u_1[\sigma_1(t_1)]$, ..., $u_n[\sigma_n(t_n)]$, des égalités e_1, ..., e_n et e'_1, ..., e'_n tels que, pour $i \in \{1,\dots,n\}$, σ_i soit une substitution et $(\bigvee_{i\in\{1,\dots,n\}} Red(t_i,e_i))$ apparaisse dans c_r et $\neg Red(u_i[\sigma_i(t_i)],e'_i)$ apparaisse dans c_r et $e_i \preceq_e e'_i$.*
- *il existe des termes $u_1[\sigma_1(t_1)]$, ..., $u_n[\sigma_n(t_n)]$, des égalités e_1, ..., e_n et $v_1 \approx w_1$, ..., $v_n \approx w_n$ tels que, pour $i \in \{1,\dots,n\}$, σ_i soit une substitution et $(\bigvee_{i\in\{1,\dots,n\}} Red(t_i,e_i))$ apparaisse dans c_r et $\neg Red(u_i[\sigma_i(t_i)],v_i \approx w_i)$ apparaisse dans c_r et $v_i \succ_t t_i$.* □

Par exemple, si l'on omet le deuxième argument du prédicat *Red* pour éviter les lourdeurs d'écriture, $(Red(a) \vee Red(b)) \wedge \neg Red(a) \wedge \neg Red(c)$ est une contrainte de réductibilité non contradictoire syntaxiquement, mais $(Red(a) \vee Red(b)) \wedge \neg Red(a) \wedge \neg Red(b)$ l'est, car elle est équivalente à $(Red(a) \vee Red(b)) \wedge \neg(Red(a) \vee Red(b))$.

Il existe une relation entre la contradiction syntaxique et l'insatisfaisabilité. Nous prouvons dans le théorème 11 qu'une contrainte de réductibilité satisfaisable n'est pas contradictoire syntaxiquement.

Théorème 11 *Soit c_r une contrainte de réductibilité. Si c_r est satisfaisable, alors c_r n'est pas contradictoire syntaxiquement.*

Preuve : Nous allons prouver la contraposée. Soit c_r une contrainte de réductibilité contradictoire syntaxiquement. Montrons que c_r est insatisfaisable.

Soient $\sigma(c_r)$ une instance close de c_r et R un système de réécriture clos.

c_r satisfait l'une des conditions de la définition 37.

- Si elle satisfait la première condition, alors $c_r = \bot$ et donc $\sigma(c_r) = \bot$. Ainsi, $\sigma(c_r)$ est insatisfaisable par définition, car elle ne satisfait aucune des conditions de la définition 35.

- Supposons que la condition 2 ou la condition 3 de la définition 37 soit vraie pour c_r. Alors, c_r contient $(\bigvee_{i\in\{1,\ldots,n\}} Red(t_i,e_i))$. Nous supposons que $\sigma(c_r)$ est satisfaisable dans R et nous allons obtenir une contradiction.
  $\sigma(c_r)$ contient $(\bigvee_{i\in\{1,\ldots,n\}} Red(\sigma(t_i),\sigma(e_i)))$. Ainsi, comme $\sigma(c_r)$ est satisfaisable, pour $i \in \{1,\ldots,n\}$, $\sigma(t_i)$ est réductible par une égalité e''_i plus petite que $\sigma(e_i)$ $(e''_i \prec_e \sigma(e_i))$.
  Si la condition 2 de la définition 37 est vraie, c_r contient, pour $i \in \{1,\ldots,n\}$, $\neg Red(u_i[t_i],e'_i)$, où $e_i \preceq_e e'_i$. Ainsi, $\sigma(c_r)$ contient $\neg Red(\sigma(u_i[t_i]),\sigma(e'_i))$ et $\sigma(e_i) \preceq_e \sigma(e'_i)$. Comme $\sigma(t_i)$ est réductible par e''_i, alors $\sigma(u_i[t_i])$ est aussi réductible par e''_i. Et comme $e''_i \prec_e \sigma(e_i)$, nous avons : $e''_i \preceq_e \sigma(e'_i)$. Ainsi, d'après le cas 5 de la définition 35, $\sigma(c_r)$ n'est pas satisfaisable.
  Si la condition 3 de la définition 37 est vraie, c_r contient, pour $i \in \{1,\ldots,n\}$, $\neg Red(u_i[t_i],v_i \approx w_i)$, où $v_i \succ_t t_i$. Ainsi, $\sigma(c_r)$ contient, pour $i \in \{1,\ldots,n\}$, $\neg Red(\sigma(u_i[t_i]), \sigma(v_i) \approx \sigma(w_i))$ et $\sigma(v_i) \succ_t \sigma(t_i)$. Comme $\sigma(t_i)$ est réductible par e''_i, alors $\sigma(u_i[t_i])$ est aussi réductible par e''_i. Comme e''_i réduit $\sigma(t_i)$, alors e''_i est de la forme $v'_i \to w'_i$, où $v'_i \preceq_t \sigma(t_i)$. Ainsi, $v'_i \prec_t \sigma(v_i)$, ce qui implique que $e''_i \prec_e (\sigma(v_i) \approx \sigma(w_i))$. Donc, $\sigma(c_r)$ n'est pas satisfaisable d'après le cas 5 de la définition 35.

Dans les trois cas, la contradiction montre que $\sigma(c_r)$ n'est pas satisfaisable, donc que c_r également. Le théorème est donc prouvé. □

On peut remarquer qu'il est simple de tester si une contrainte de réductibilité est contradictoire syntaxiquement en utilisant la définition 37. Nous allons décrire ici, comment le test de la contradiction syntaxique d'une contrainte de réductibilité est réalisé en ELAN. Les contraintes de réductibilité étant des conjonctions de disjonctions, nous avons utilisé la même technique d'implantation que celle développée dans (Castro, 1998b). Cette technique peut être utilisée pour toute contrainte représentée par une conjonction de disjonctions.

Nous définissons l'objet R pour tester la contradiction syntaxique des contraintes de réductibilité. Cet objet est une paire $R[C1,C2]$ où :

- $C1$ est une contrainte de réductibilité conjonctive dont on doit tester si elle est contradictoire syntaxiquement, et
- $C2$ est une contrainte de réductibilité disjonctive dont on doit tester si elle est contradictoire syntaxiquement.

```
operators
global
 R@ : (tuple[redconstraint,redconstraint]) tuple[redconstraint,redconstraint];
end
```

Les règles de transformation de l'objet R sont les règles *PostElementaryConjunct*, *ContradictionForEC1*, *ContradictionForEC2* et *IsSolution*.

- La règle *PostElementaryConjunct* ajoute une contrainte élémentaire, c'est-à-dire une contrainte qui n'est formée que de conjonctions et sans disjonction, à la contrainte C qui collecte la contrainte composée seulement de conjonctions.
- Les règle *ContradictionForEC1* et *ContradictionForEC2* vérifient qu'il n'y a pas de contradiction syntaxique locale dans une contrainte élémentaire. Le test réalisé consiste à vérifier qu'aucune des conditions de la définition 37 n'est vérifiée. Ce test est réalisé dans la fonction *IsContradictoire.*
- La règle *IsSolution* renvoie une structure R contenant une contrainte de réductibilité élémentaire $C11$ qui n'est pas contradictoire syntaxiquement.

```
rules for tuple[redconstraint,redconstraint]
 C, CC, c1, c2, C11, C22 : redconstraint;
global

[PostElementaryConjunct] R[C, c1 V c2 & CC] => R[c1 & C, CC]
    if is_elementary_redconstraint(c1) and c1 != END
end

[PostElementaryConjunct] R[C, c1 & CC] => R[c1 & C, CC]
   if is_elementary_redconstraint(c1) and c1 != END
end

[ContradictionForEC1] R[C11,C22] => R[C11,C22]
     if IsContradictoire(C11)
end

[ContradictionForEC2] R[C11,C22] => R[F,C22]
end

[IsSolution] R[C11,C22] => R[C11,C22]
     if C11 != F and C22 != F
end

end
```

Nous décrivons maintenant la stratégie *ContradictionForECandDC* appliquée sur les règles précédentes en ELAN.

```
stratop
global
 ContradictionForECandDC   : <tuple[redconstraint,redconstraint]> bs;
end

strategies for tuple[redconstraint,redconstraint]
implicit
[] ContradictionForECandDC =>
first one(
first one(ContradictionForEC1, ContradictionForEC2) ;
repeat* (
        dk(PostElementaryConjunct);
        first one (ContradictionForEC1, ContradictionForEC2)
       )
;first one (IsSolution)
)
end
end
```

Les règles *ContradictionForEC1* et *ContradictionForEC2* sont d'abord essayées sur l'objet R donné en entrée. Le résultat, à cause du *first one*, est le résultat de la première de ces règles

qui est applicable. On applique ensuite la règle *PostElementaryConjunct*. Après l'application de cette règle, on obtient un ou plusieurs résultats à cause du *dk*. On applique la première des règles applicables parmi *ContradictionForEC1* et *ContradictionForEC2* sur le premier résultat obtenu par le *dk*. Les résultats obtenus sont les problèmes d'entrée pour la nouvelle itération du *repeat*. De cette façon, à chaque itération de la boucle *repeat*, on crée un point de choix et on peut ainsi simuler le retour-arrière (*backtracking*). Le *backtracking* intervient si l'application *first one(IsSolution)* échoue sur l'objet obtenu après le *repeat*. Sinon, on a trouvé une solution et la stratégie s'arrête sur la première contrainte de réductibilité non contradictoire syntaxiquement trouvée. Si la stratégie *ContradictionForECandDC* échoue, la contrainte de réductibilité est contradictoire syntaxiquement, sinon elle ne l'est pas.

La définition 38 nous montre comment transformer une contrainte équationnelle en une contrainte de réductibilité en utilisant la fonction *EqToRedConst*. Les contraintes de réductibilité construites de cette façon étiquettent les arcs du graphe de dépendance.

Définition 38 *Soit c une contrainte équationnelle. Nous définissons* $EqToRedConst(c)$ *comme étant la contrainte de réductibilité* $\bigvee\{Red(t) \mid t \text{ est un sous} - \text{terme de } \sigma(x) \text{ pour } x \in Dom(\sigma) \text{ et } \sigma = mgs(c)\}$. *En particulier,* $EqToRedConst(\top) = \bot$. □

4.2.2 Graphe de dépendance

Un graphe G est décrit par $G = (V,ED)$, où V est un ensemble de sommets et ED est un ensemble d'arcs.

Le *graphe de dépendance* de la complétion d'un ensemble d'égalités E est un graphe orienté construit pendant la complétion de E. Il schématise les dépendances entre les égalités et permet de contrôler la complétude.

Les sommets du graphe de dépendance sont étiquetés par des égalités. Soit *label* une fonction telle que, pour un sommet v, $label(v)$ soit l'égalité e étiquetant v. On associe un ensemble de sommets $Cancestor(v)$ à chaque sommet qui détermine les ancêtres de $label(v)$.

Il existe trois sortes d'arcs dans le graphe de dépendance : des arcs de contraintes appelés arcs C, des arcs d'inférence appelés arcs I et des arcs de simplification appelés arcs S. Chaque arc a une contrainte de réductibilité qui lui est associée, déterminée par le type d'arc (C, I ou S) et les contraintes des égalités étiquetant les sommets aux extrémités de l'arc. Soit e_d un arc du sommet v_1 étiqueté par l'égalité $e_1 [\![c_1]\!]$ au sommet v_2 étiqueté par l'égalité $e_2 [\![c_2]\!]$ dans le graphe de dépendance.

- Si e_d est un arc C, alors la contrainte de réductibilité associée à e_d est $EqToRedConst(c_1)$. e_d est noté par (v_1,v_2,C).
- Si e_d est un arc I, alors la contrainte de réductibilité associée à e_d est $\neg EqToRedConst(c_2)$. e_d est noté par (v_1,v_2,I).
- Si e_d est un arc S, alors la contrainte de réductibilité associée à e_d est $\top$. e_d est noté par (v_1,v_2,S).

4.2.3 E-chemins et E-cycles

Nous donnons ici la définition d'un E-chemin et d'un E-cycle et une propriété des E-chemins et des E-cycles.

Définition 39 *Un* E-chemin *est un chemin d'arcs C, I et S dans le graphe de dépendance tels que la conjonction des contraintes de réductibilité associées à ces arcs ne soit pas contradictoire syntaxiquement.* □

Définition 40 *Un* E-cycle *est un E-chemin qui commence et finit au même sommet et qui contient au moins un arc C et un arc S.* □

Le problème de trouver si un chemin ou un cycle est un E-chemin ou un E-cycle dans le graphe de dépendance est un problème NP-complet (Hermann, 1998). En effet, ce problème peut être réduit par une réduction polynomiale en un problème de satisfaisabilité de formules.

Le théorème suivant fournit une propriété des E-chemins et des E-cycles qui sera utilisée dans la suite. En effet, dans un E-chemin ou un E-cycle, un arc I ne peut pas être suivi par un arc C. La preuve de cette propriété provient directement de la caractérisation des contraintes de réductibilité des arcs I et C.

Théorème 12 *Un E-chemin ne contient pas d'arc I suivi d'un arc C.*

Preuve : Considérons un arc I e_{inf} dans un E-chemin $\mathcal{P}$. e_{inf} est du sommet v étiqueté par $e\,[\![c]\!]$ à un sommet v' étiqueté par $e'\,[\![c']\!]$. La contrainte de réductibilité associée à e_{inf} est ainsi $\neg EqToRedConst(c')$.
Si e_{inf} est suivi par un arc C, cet arc C a comme contrainte de réductibilité $EqToRedConst(c')$.
Un E-chemin ne peut pas contenir ces deux arcs, car la contrainte de réductibilité $EqToRedConst(c') \wedge \neg EqToRedConst(c')$ est contradictoire syntaxiquement.
Ainsi, un arc I ne peut pas être suivi par un arc C. □

4.3 Graphe de dépendance et simplification E-cycle

Dans cette section, nous décrivons la construction du graphe de dépendance pendant la complétion basique avec simplification E-cycle de façon informelle et de façon formelle en utilisant des transitions de graphes. Nous présentons le système *BCES* qui nous permet de faire de la complétion basique avec simplification E-cycle.

4.3.1 Le système d'inférence BCES

Au début du processus de la complétion basique, l'ensemble des égalités E à compléter est représenté par le *graphe de dépendance initial* G_{init}.

Chaque égalité de l'ensemble E à compléter est l'étiquette d'un sommet de $G_{init} = (V_{init}, ED_{init})$ avec $ED_{init} = \emptyset$. $Cancestor(v) = \{v\}$ pour tout $v \in V_{init}$.

Quand une inférence de *BCES* est réalisée, le graphe de dépendance est mis à jour. Un nouveau sommet étiqueté par la conclusion de l'inférence est ajouté au graphe et des arcs sont ajoutés.

Nous présentons maintenant comment les règles de *BCES* mettent à jour le graphe de dépendance en considérant les différentes règles de *BCES*:

- la règle *Paire Critique Basique*,
- la règle *Simplification E-cycle*,
- la règle de suppression, *Blocage E-cycle*, et
- la règle *Rétraction E-cycle*.

Les règles *Simplification E-cycle* et *Blocage E-cycle* déterminent la *stratégie de simplification E-cycle*.

4.3.1.1 Paire Critique Basique

La règle *Paire Critique Basique* est la règle standard de (Kirchner et al., 1990) présentée dans la section 2.3.4.

Paire Critique Basique

$$\frac{g[l']_\omega \approx d\,[\![c_1]\!] \quad l \approx r\,[\![c_2]\!]}{\mathbf{g[r] \approx d\,[\![l =^? l' \wedge c_1 \wedge c_2]\!]}}$$

- l' n'est pas une variable,
- il existe une substitution σ telle que $\sigma \in CSU((l =^? l') \wedge c_1 \wedge c_2)$, $\sigma(l) \not\leq_t \sigma(r)$ et $\sigma(g[l']) \not\leq_t \sigma(d)$.

Le calcul d'une paire critique basique nécessite une mise à jour du graphe de dépendance. Soient $g[l'] \approx d\,[\![c_1]\!]$ l'étiquette du sommet v_1 et $l \approx r\,[\![c_2]\!]$ l'étiquette du sommet v_2.

- v_3 est un nouveau sommet ajouté au graphe de dépendance tel que $label(v_3) = g[r] \approx d\,[\![l =^? l' \wedge c_1 \wedge c_2]\!]$.
- Si $l =^? l' \wedge c_1 \wedge c_2$ n'est pas la contrainte équationnelle triviale $\top$, nous ajoutons des arcs C du sommet v_3 à tous les sommets v tels que $v \in Cancestor(v_1) \cup Cancestor(v_2)$. Sinon, aucun arc C n'est ajouté.
  Dans les deux cas, $Cancestor(v_3)$ est mis à jour. $Cancestor(v_3) = Cancestor(v_1) \cup Cancestor(v_2)$.
- Un arc I est ajouté du sommet v_1 au sommet v_3.

4.3.1.2 Simplification E-cycle

La règle *Simplification E-cycle* est la règle de simplification standard donnée dans la section 2.3.4.2 à laquelle on a ajouté une condition pour éviter la création de E-cycles dans le graphe de dépendance.

Simplification E-cycle

$$\frac{g[l'] \approx d\,[\![c_1]\!] \quad l \approx r\,[\![c_2]\!]}{\mathbf{g[\mu_{min}(\sigma_2(r))] \approx d\,[\![c_1 \wedge \mu(c_2)]\!]}} \quad \text{si :}$$

- $g[l'] \approx d\,[\![c_1]\!]$ peut être standard simplifiée par $l \approx r\,[\![c_2]\!]$, et
- l'addition d'arcs S de la prémisse « gauche » à la prémisse « droite » et de la prémisse « gauche » à l'égalité conclusion de l'inférence ne crée pas de E-cycle.

Soient $g[l'] \approx d\,[\![c_1]\!]$ l'étiquette du sommet v_1 et $l \approx r\,[\![c_2]\!]$ l'étiquette du sommet v_2. L'égalité $g[l'] \approx d\,[\![c_1]\!]$ est supprimée de l'espace de recherche.

- v_3 est un nouveau sommet du graphe de dépendance tel que $label(v_3) = g[\mu_{min}(\sigma_2(r))] \approx d\,[\![c_1 \wedge \mu(c_2)]\!]$.
- Nous ajoutons des arcs C et mettons à jour $Cancestor(v_3)$ comme décrit dans la section 4.3.1.1.
- Aucun arc I n'est ajouté.
- Nous ajoutons des arcs S du sommet v_1 au sommet v_2 et du sommet v_1 au sommet v_3.

4.3.1.3 Blocage E-cycle

La règle *Blocage E-cycle* est la règle de blocage standard donnée dans la section 2.3.4.3 à laquelle on a ajouté une condition pour éviter la création de E-cycles dans le graphe de dépendance. Le blocage permet de supprimer une égalité dont la contrainte est réductible.

Blocage E-cycle

$\{g \approx d [\![c_1[l']]\!], l \approx r [\![c_2]\!]\} \cup \Gamma \rightarrow \{l \approx r [\![c_2]\!]\} \cup \Gamma$ si :

- Γ est un ensemble d'égalités,
- l' n'est pas une variable,
- il existe un filtre μ et $\sigma_2 = mgu(c_2)$ tels que $l' = \mu(\sigma_2(s))$, et
- l'addition d'un arc S du sommet étiqueté par $g \approx d[\![c_1[l']]\!]$ au sommet étiqueté par $l \approx r[\![c_2]\!]$ ne crée pas de E-cycle.

L'égalité $g \approx d [\![c_1[l']]\!]$ est supprimée de l'espace de recherche.

- Nous ajoutons un arc S du sommet étiqueté par $g \approx d [\![c_1[l']]\!]$ au sommet étiqueté par $l \approx r [\![c_2]\!]$.

4.3.1.4 Rétraction E-cycle

Rétraction E-cycle

$$\frac{g \approx d [\![c]\!]}{\sigma(\mathbf{g} \approx \mathbf{d})} \quad \text{si :}$$

- $\sigma = mgu(c)$.
- Soit v le sommet d'étiquette $g \approx d [\![c]\!]$. L'étiquette de v devient $\sigma(g \approx d)$.

La rétraction est utilisée pour permettre de faire plus de simplifications en appliquant la contrainte. L'avantage de la complétion basique à savoir l'utilisation des contraintes est alors supprimé. L'application de cette règle est toutefois optionnelle. Dans le cas de la complétion basique avec simplification E-cycle, pour faire plus de simplifications E-cycle, il faut éviter de créer des E-cycles. Ceci ne peut être réaliser qu'en limitant le nombre d'arcs C sur le graphe de dépendance, c'est-à-dire en rétractant les égalités au maximum de façon à obtenir des égalités non contraintes. Ainsi, tous les arcs C partant de v sont supprimés du graphe de dépendance. L'ensemble $Cancestor(v)$ est inchangé. Si l'on ne rétracte pas au maximum, aucun arc C n'est supprimé et dans ce cas, la rétraction n'a pas d'intérêt pour la complétion basique avec simplification E-cycle. Pour conclure, la rétraction n'est pas intéressante dans le cas de la simplification E-cycle.

Correction La correction du système d'inférence *BCES* est évidente vue la forme des règles d'inférence le composant.

4.3.2 Les transitions de graphes

Nous présentons maintenant de façon formelle comment une règle d'expansion (règle de calcul de paire critique basique), comment une règle de contraction (règle de suppression ou de simplification) et comment une rétraction mettent à jour le graphe de dépendance en utilisant

des *transitions de graphes*. On rappelle qu'une suppression supprime une égalité et qu'une simplification consiste en une inférence de paire critique et en une suppression.

Définition 41 *Une* transition de graphes *est notée* $(E_i,G_i) \rightarrow (E_{i+1},G_{i+1})$*, où* E_i *et* E_{i+1} *sont des ensembles d'égalités tels que* E_{i+1} *soit obtenu à partir de* E_i *en réalisant une inférence de paire critique basique, une inférence de suppression ou une rétraction et* $G_i = (V_i,ED_i)$ *et* $G_{i+1} = (V_{i+1},ED_{i+1})$ *soient des graphes de dépendance tels que* G_{i+1} *soit obtenu de* G_i *par :*

- *une inférence de paire critique basique :*
  Nous avons la transition de graphes suivante : $(\{e_0,e_1\} \cup \Gamma,G_i) \rightarrow (\{e_0,e_1,e_2\} \cup \Gamma,G_{i+1})$*, où* e_0 *est l'égalité « gauche », * e_1 *est l'égalité « droite » et* e_2 *est l'égalité conclusion de l'inférence de paire critique basique.*
  Soient e_0 *l'étiquette du sommet* v_0 *et* e_1 *l'étiquette du sommet* v_1*.*
  - $V_{i+1} = V_i \cup \{v_2\}$ *tel que* $label(v_2) = e_2$
  - $ED_{i+1} = ED_i \cup E_C \cup E_I$ *où :*
    - *Si* e_2 *est une égalité contrainte, alors* $E_C = \emptyset$*, sinon* $E_C = \bigcup_{v \in Cancestor(v_0)}(v_2,v,C) \cup \bigcup_{v \in Cancestor(v_1)}(v_2,v,C)$.
    - $Cancestor(v_2) = Cancestor(v_0) \cup Cancestor(v_1)$.
    - $E_I = \{(v_0,v_2,I)\}$
- *une inférence de suppression :*
  Nous avons la transition de graphes suivante : $(\{e_0,e_1,\ldots,e_n\} \cup \Gamma,G_i) \rightarrow (\{e_1,e_2,\ldots,e_n\} \cup \Gamma,G_{i+1})$*, où l'égalité* e_0 *est supprimée à cause de* $e_1, \ldots, e_n$.
  Soit e_i *l'étiquette du sommet* v_i *pour* $i \in \{0,\ldots,n\}$.
  - $V_{i+1} = V_i$
  - $ED_{i+1} = ED_i \cup E_S$ *où :*
    $E_S = \bigcup_{i \in \{1,\ldots,n\}}(v_0,v_i,S)$.
- *une rétraction :*
  Nous avons la transition de graphes suivante : $(\{e_1\} \cup \Gamma,G_i) \rightarrow (\{e'_1\},G_{i+1})$*, où* e'_1 *est un rétracte de* e_1.
  Soit e_1 *l'étiquette du sommet* v.
  - $V_{i+1} = V_i$ *où l'étiquette de* v *est changé en* e'_1.
  - $ED_{i+1} = ED_i - E_C$ *où :*
    - $E_C = \{e_d \mid e_d$ *est un arc* C *sortant de* v $\}$*, si* e'_1 *est une égalité non contrainte, sinon* $E_C = \emptyset$. □

Nous pouvons résumer la définition 41 comme suit.

- Un arc C est créé d'une égalité contrainte à ses ancêtres originaux, les égalités initiales de l'ensemble E à compléter. La dépendance schématisée par les arcs de contraintes n'est utile que si la contrainte de la conclusion est réductible. Nous avons besoin d'ajouter cette dépendance sur le graphe de dépendance pour pouvoir créer une version réduite de l'égalité contrainte.
- Un arc I est ajouté du sommet étiqueté par la prémisse « gauche » d'une inférence au sommet étiqueté par la conclusion d'une inférence. Ceci indique que la prémisse « gauche » dépend de la conclusion. L'arc I indique que l'inférence n'est utile que si la conclusion de l'inférence a une contrainte irréductible. En effet, dans le cas contraire, l'inférence n'a pas besoin d'être réalisée.
- Un arc S est ajouté de l'égalité à simplifier à l'égalité simplificatrice et de l'égalité à simplifier à l'égalité conclusion de la simplification. Ceci indique que l'égalité simplifiée dépend des deux autres.

Dans le graphe de dépendance, il n'y a pas de E-cycle. La construction évite la création de tels cycles. En fait, seule l'addition d'arc S peut créer un E-cycle (voir théorème 12) et les règles susceptibles d'ajouter des arcs S telles que la simplification E-cycle et le blocage E-cycle interdisent la création de E-cycles.

Définition 42 *Soit une séquence de graphes* G_0, G_1, ... *où* $G_i = (V_i, ED_i)$, *pour tout* i, *la* limite *est* G_∞ *avec* $G_\infty = (V_\infty, ED_\infty)$, *où* $V_\infty = \bigcup_i \bigcap_{j \geq i} V_j$, $label(v) = \bigcup_i \bigcap_{j \geq i} label(v_j)$ *pour tout* $v \in V_\infty$, *et* $ED_\infty = \bigcup_i \bigcap_{j \geq i} ED_j$. □

Définition 43
- *Une* dérivation de transitions de graphes *de* E *est une dérivation qui peut être infinie* $(E_0 = E, G_0 = G_{init}) \rightarrow (E_1, G_1) \rightarrow \ldots$, *où pour tout* i, $(E_i, G_i) \rightarrow (E_{i+1}, G_{i+1})$ *est une transition de graphes. La* limite de transition *est notée* $T_\infty = (E_\infty, G_\infty)$.
- *Une dérivation de transitions de graphes* $(E_0, G_0) \rightarrow (E_1, G_1) \rightarrow \ldots$ *est* correcte, *si la dérivation de complétion* $E_0 \vdash E_1 \vdash \ldots$ *est correcte.*
- *Une dérivation de transitions de graphes* $(E_0, G_0) \rightarrow (E_1, G_1) \rightarrow \ldots$ *est* équitable, *si la dérivation de complétion* $E_0 \vdash E_1 \vdash \ldots$ *est équitable.* □

Nous montrons maintenant comment les transitions de graphe sont implantées en ELAN. Nous présentons dans l'ordre l'implantation des règles *Paire Critique Basique*, *Simplification E-cycle* et *Blocage E-cycle*.

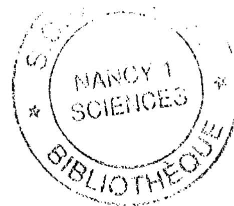
Paire Critique Basique

```
[BCP] Basic_Critical_Pair(g=d[c1]<counter1><Cancestor1>,
l=r[c2]<counter2><Cancestor2>,omega,Graph_init,counter)
=>
 BCPNewEqG
     /* omega est une position non variable de g */
     /* Calcul de la contrainte de l'égalité conclusion */
where unif        := (unifys)c1 & c2 & g at omega =? l
where sigma       := ()constraint_to_subst(unif)
where sigmad      := ()sigma(d)
where sigmag      := ()sigma(g)
where sigmal      := ()sigma(l)
where sigmar      := ()sigma(r)
     /* Conditions d'ordre */
if not sigmad >lpo sigmag
if not sigmar >lpo sigmal
     /* Calcul de Cancestor pour l'égalité conclusion */
where Cancestor3:= ()simple_append(Cancestor1,Cancestor2)
where NewEq:= ()g[r] at omega = d[unif]<counter><Cancestor3>
     /* Ajout d'un arc I avec comme contrainte de */
     /* réductibilité redi au graphe de dépendance*/
where redi        := ()EqToRedConst(I,NewEq)
where Graph0     := ()add_Iedge([counter1,counter,I,redi],Graph_init)
     /* Si l'égalité contrainte est contrainte, on ajoute des arcs C */
     /* au graphe de dépendance */
where isconstrainedeq   := ()isconstrained(NewEq)
where matcha      := (match)l =? g at omega
where bmatch      := ()$matcha == false
choose
try
if isconstrainedeq
             /* L'égalité NewEq est contrainte */
             /* Ajout d'arc C avec comme contrainte de réductibilité redc */
             /* au graphe de dépendance */
        where redc       := ()EqToRedConst(C,NewEq)
        where Graph     := ()add_Cedges(counter,Cancestor3,C,redc,Graph0)
            /* On réalise une paire critique basique et pas une simplification */
        if bmatch or isEcycle([counter1,counter2,S,T],Graph) or
        isEcycle([counter1,counter,S,T],Graph)
        where  BCPNewEqG  := ()[NewEq,Graph]
try
if not isconstrainedeq
          /* On réalise une paire critique basique et pas une simplification */
        if bmatch or isEcycle([counter1,counter2,S,T],Graph0)
        or isEcycle([counter1,counter,S,T],Graph0)
        where  BCPNewEqG  := ()[NewEq,Graph0]
end
end
end
```

Simplification E-cycle

```
[ESimpLeft] Ecycle_Simplification(g=d[c1]<counter1><Cancestor1>,
l=r[c2]<counter2><Cancestor2>, Graph_init, counter)
=>
 ESNewEqG
where omega    := (chooseOccurence)nvocc(g)
where sigma1   := ()constraint_to_subst(c1)
where sigma2   := ()constraint_to_subst(c2)
     /* Calcul du filtre minimum (voir la section~2.3.4.3) */
where matcha   := (matchs)sigma2(l) =? sigma1(g at omega)
where mu       := ()constraint_to_subst(matcha)
where min_mu   := ()min_match(g,c1,sigma1,l,c2,sigma2,omega,mu)
where mus2l    := ()mu(sigma2(l))
where mus2r    := ()mu(sigma2(r))
     /* Condition d'ordre */
if mus2l >lpo mus2r
     /* Calcul de Cancestor pour l'égalité conclusion */
where Cancestor3   := ()simple_append(Cancestor1,Cancestor2)
where min_mur := ()min_mu(r)
where unif     := (unifys)c1 & c2 & matcha
where NewEq  := ()g[min_mur] at omega= d[unif]<counter><Cancestor3>
where isposempty  := ()eq_list[int](omega,nil)
choose
try
if isposempty
              /* omega est la position vide */
        where sigma1d       := ()sigma1(d)
        if not is_renaming(mu) or lpo(sigma1d,mus2r) or eq_term(sigma1d,mus2r)
        where isconstrainedeq:= ()isconstrained(NewEq)
        choose
        try
        if isconstrainedeq
                     /* L'égalité NewEq est contrainte. On ajoute des */
                     /* arcs C */
                where redc   := ()EqToRedConst(C, NewEq)
                where GraphC := ()add_Cedges(counter,Cancestor3,C,redc,Graph_init)
                if not isEcycle([counter1,counter2,S,T],GraphC)
                if not isEcycle([counter1,counter,S,T],GraphC)
                     /* On ajoute les arcs S car aucun cycle n'est créé */
                where Gs     := ()add_Sedge([counter1,counter2,S,T],GraphC)
                where Graph  := ()add_Sedge([counter1,counter,S,T],Gs)
                where  ESNewEqG  := ()[NewEq, Graph]
        try
        if not isconstrainedeq
        if not isEcycle([counter1,counter2,S,T],Graph_init)
        if not isEcycle([counter1,counter,S,T],Graph_init)
                where Gs     := ()add_Sedge([counter1,counter2,S,T],Graph_init)
                where Graph  := ()add_Sedge([counter1,counter,S,T],Gs)
                where  ESNewEqG  := ()[NewEq, Graph]
        end
```

```
try
if not isposempty
            /* omega n'est pas la position vide */
        where isconstrainedeq:= ()isconstrained(NewEq)
        choose
        try
        if isconstrainedeq
                    /* L'égalité NewEq est contrainte. On ajoute des */
                    /* arcs C */
               where redc   := ()EqToRedConst(C, NewEq)
               where GraphC := ()add_Cedges(counter,Cancestor3,C,redc,Graph_init)
               if not isEcycle([counter1,counter2,S,T],GraphC)
               if not isEcycle([counter1,counter,S,T],GraphC)
                    /* On ajoute les arcs S car aucun cycle n'est créé */
               where Gs     := ()add_Sedge([counter1,counter2,S,T],GraphC)
               where Graph  := ()add_Sedge([counter1,counter,S,T],Gs)
               where  ESNewEqG := ()[NewEq, Graph]
        try
        if not isconstrainedeq
        if not isEcycle([counter1,counter2,S,T],Graph_init)
        if not isEcycle([counter1,counter,S,T],Graph_init)
               where Gs     := ()add_Sedge([counter1,counter2,S,T],Graph_init)
               where Graph  := ()add_Sedge([counter1,counter,S,T],Gs)
               where  ESNewEqG := ()[NewEq, Graph]
        end
end
end
```



Blocage E-cycle

```
[EBlockingLeft] Ecycle_Simplification(g=d[c1]<counter1><Cancestor1>,
l=r[c2]<counter2><Cancestor2>, Graph_init, counter)
=>
 EBNewEqG
where propgg    := (propagate)propagate(g=d[c1]<counter1><Cancestor1>)
where prop      := ()lhs(equality(propgg))
     /* omega est une position variable de g */
     /* On a la même règle pour d */
where omega     := (chooseOccurence)nvocc(prop)
where sigma1    := ()constraint_to_subst(c1)
if isvarpos(g,omega,sigma1)
where sigma2    := ()constraint_to_subst(c2)
     /* Calcul du filtre */
where matcha    := (matchs)sigma2(l) =? sigma1(g) at omega
where mu        := ()constraint_to_subst(matcha)
where mus2l     := ()mu(sigma2(l))
where mus2r     := ()mu(sigma2(r))
where unif      := (unifys)c1 & mu(c2)
     /* Condition d'ordre */
if mus2l >lpo mus2r
     /* Ajout des arcs S si aucun E-cycle n'est créé */
if not isEcycle([counter1,counter2,S,T],Graph_init)
where Graph     := ()add_Sedge([counter1,counter2,S,T],Graph_init)
where EBNewEqG := ()[eqempty,Graph]
end
```

4.3.3 Propriétés des graphes de dépendance

Dans cette section, nous donnons deux propriétés des graphes de dépendance obtenues par construction.

Le théorème 13 prouve qu'un E-cycle ne contient pas seulement des arcs C.

Le théorème 14 prouve que c'est seulement une règle de suppression et de ce fait, l'addition d'un arc S qui est susceptible de créer un E-cycle. Il prouve également qu'un E-cycle contient au moins un arc S.

Théorème 13 *Un E-cycle ne contient pas seulement des arcs C.*

Preuve : En effet, les arcs C vont des égalités contraintes aux égalités initiales non contraintes de l'ensemble des égalités E à compléter. □

Théorème 14 *Soit* $(E_0 = E, G_0 = G_{init}) \to (E_1, G_1) \to \ldots \to (E_{n-1}, G_{n-1}) \to (E_n, G_n) \ldots$ *une dérivation de transitions de graphes. Supposons que l'existence d'un E-cycle dans le graphe de dépendance est possible. Si* G_{n-1} *ne contient pas de E-cycle et* G_n *contient un E-cycle, alors* E_n *a été obtenu de* E_{n-1} *par une règle de simplification ou de suppression.*

Preuve : Pour prouver que seules les règles de simplification et de suppression sont susceptibles de créer un E-cycle, nous prouvons que ni une inférence de paire critique basique, ni une rétraction ne peuvent créer de E-cycle.

- Supposons que G_n a été obtenu de G_{n-1} dans la dérivation de transitions de graphes par une inférence de paire critique basique et que G_{n-1} ne contient pas de E-cycle.

Dans ce cas, un nouveau sommet v, des arcs C et un arc I sont ajoutés au graphe de dépendance G_{n-1} pour former G_n.
Les arcs C sont ajoutés du sommet v à tous les sommets $v_1, \ldots, v_n$ tels que les égalités $label(v_i)$, pour $i \in \{1, \ldots, n\}$, soient les égalités initiales à partir desquelles l'égalité $label(v)$ a été déduite. Un arc I est ajouté du sommet étiqueté par l'égalité « gauche » de l'inférence au sommet v. Le seul E-cycle de G_n qui peut être créé est alors celui considérant le sommet v, le nouvel arc I ajouté et un arc C ajouté, mais ceci n'est pas un E-cycle (voir théorème 12). Dans le cas, où aucun arc C n'est ajouté, il est évident qu'un E-cycle ne peut être créé.

- Supposons que G_n est obtenu de G_{n-1} dans une dérivation de transitions de graphes par une rétraction et que G_{n-1} ne contient pas de E-cycle.
  Une rétraction ne peut créer de E-cycle car elle consiste seulement à changer l'étiquette d'un sommet et à supprimer éventuellement des arcs C.

Ainsi, ce sont seulement les règles *Simplification E-cycle* et *Blocage E-cycle* qui sont susceptibles de créer un E-cycle. □

4.3.4 Un exemple

Pour illustrer *BCES*, nous développons maintenant l'exemple de Nieuwenhuis et Rubio (Nieuwenhuis et Rubio, 1992a), qui montre que la complétion basique avec simplification standard est incomplète. Nous adoptons le même plan de recherche. Nous obtenons un système convergent en utilisant *BCES*. En effet, *BCES* est complet (voir la section 4.4).

Exemple 34 *Soit* $E = \{a \approx b\ (1),\ f(g(x)) \approx g(x)\ (2),\ f(g(a)) \approx b\ (3)\}$.

Nous utilisons l'ordre lexicographique sur les chemins basé sur la précédence $f \succ_{prec} g \succ_{prec} a \succ_{prec} b$.

Nous montrons les différentes étapes de la construction du graphe de dépendance sur les figures 4.1 à 4.9. Le graphe de dépendance final est visible sur la figure 4.10.

1. $$\frac{f(g(x)) \approx g(x)\ (2) \qquad f(g(a)) \approx b\ (3)}{g(x) \approx b\,[\![x =^{?} a]\!]\ (4)}$$

Nous ajoutons des arcs C de l'égalité (4) aux égalités initiales (2) et (3). La contrainte de réductibilité associée à ces arcs est $Red(a)$.

Nous ajoutons un arc I de l'égalité (2) à l'égalité (4). La contrainte de réductibilité associée à cet arc est $\neg Red(a)$.

Nous remarquons que la règle de blocage E-cycle est applicable ici, mais nous ne l'appliquons pas car son application est optionnelle et que nous voulons suivre le plan de recherche donné dans (Nieuwenhuis et Rubio, 1992a). Nous sommes donc en présence d'une inférence de paire critique basique.

Cette première étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.1.

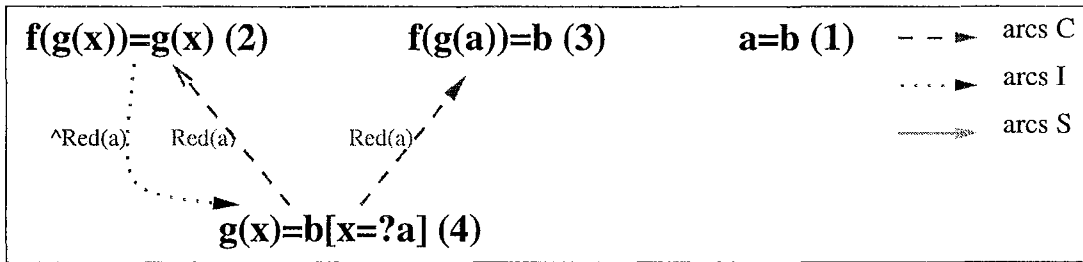


FIG. 4.1 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ *- Etape 1*

2. $$\frac{f(g(a)) \approx b\ (3) \qquad g(x) \approx b[\![x =^? a]\!]\ (4)}{f(b) \approx b\ (5)}$$

Nous n'ajoutons pas d'arc C car l'égalité (5) n'est pas une égalité contrainte. Cependant, l'ensemble des égalités initiales dont dépend l'égalité (5) est sauvé. L'égalité (5) dépend des égalités initiales (2) et (3). Ainsi, $Cancestor(v_5) = \{v_2, v_3\}$*, où* v_i *est le sommet étiqueté par l'égalité i pour* $i \in \{2,3,5\}$.

Nous ajoutons un arc I de l'égalité (3) à l'égalité (5). La contrainte de réductibilité associée à cet arc est $\top$.

L'égalité (3) peut être standard simplifiée par l'égalité (4). Cependant, il n'y a pas de simplification E-cycle. En effet, si nous ajoutons des arcs S, un E-cycle est créé. L'arc S de l'égalité (3) à l'égalité (4) (dont la contrainte de réductibilité est $\top$*) et l'arc C de l'égalité (4) à l'égalité (3) (dont la contrainte de réductibilité est* $Red(a)$*) décrivent un E-cycle. Nous sommes donc en présence d'une inférence de paire critique basique.*

Si nous supprimons l'égalité (3) comme dans une simplification standard, alors nous ne pouvons plus construire un système confluent (l'égalité $g(b) \approx b$ *n'a pas de preuve de réécriture), et le système d'inférence ne serait pas complet. La présence de ce E-cycle nous empêche de supprimer l'égalité (3). Ainsi, nous avons utilisé le graphe de dépendance pour détecter l'incomplétude. La condition de réductibilité-relative-à de la simplification basique détecte également le problème. Cependant, la simplification basique empêche certaines simplifications qui ne cause pas d'incomplétude et la simplification E-cycle les autorise.*

Cette deuxième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.2.

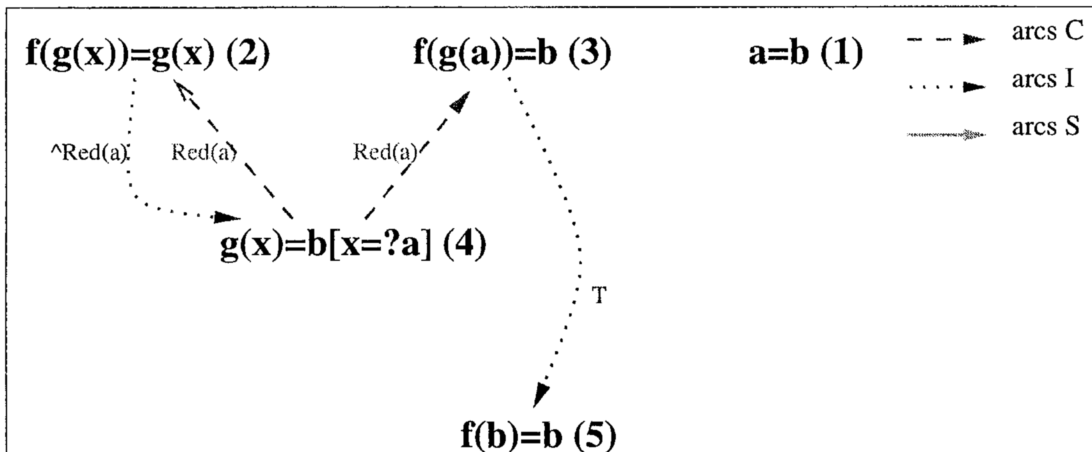


FIG. 4.2 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ *- Etape 2*

3. $$\frac{f(g(x)) \approx g(x)\ (2) \qquad g(x) \approx b[\![x =^? a]\!]\ (4)}{f(b) \approx g(x)[\![x =^? a]\!]\ (6)}$$

Nous sommes en présence d'une inférence de paire critique basique.
Nous ajoutons des arcs C de l'égalité (6) aux égalité initiales (2) et (3). La contrainte de réductibilité associée à ces arcs est Red(a).
Nous ajoutons un arc I de l'égalité (2) à l'égalité (6). La contrainte de réductibilité associée à cet arc est $\neg$Red(a).
Cette troisième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.3.

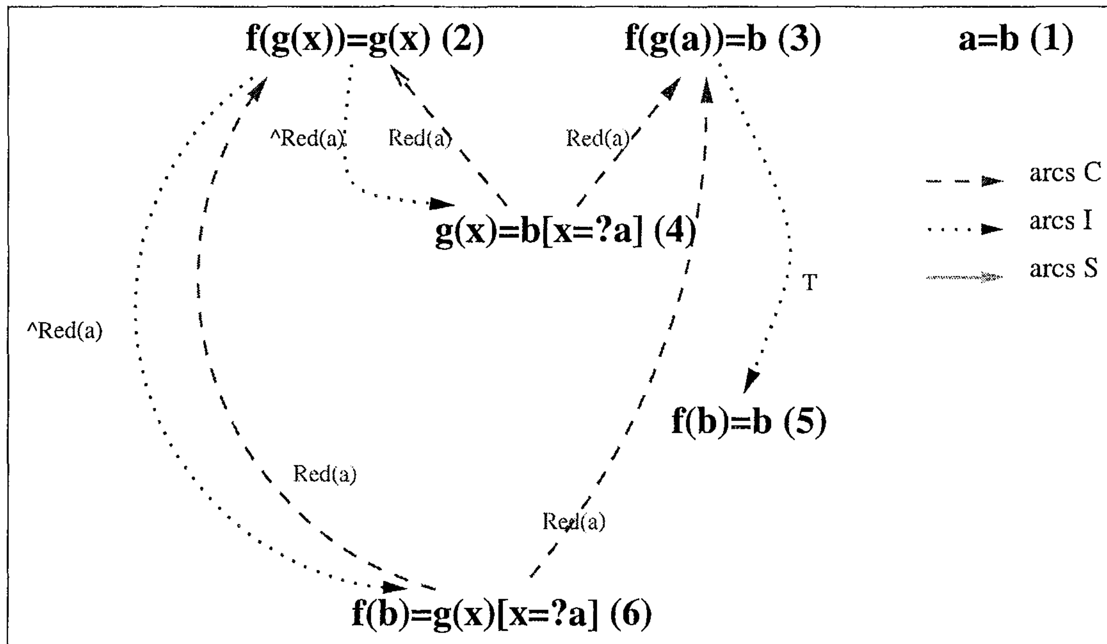


FIG. 4.3 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ - *Etape 3*

4. $$\frac{f(g(a)) \approx b\ (3) \qquad a \approx b\ (1)}{f(g(b)) \approx b\ (7)}$$

Nous sommes en présence d'une inférence de simplification E-cycle.

Nous n'ajoutons pas d'arc C car l'égalité (7) est une égalité non contrainte. Cependant, l'ensemble des égalités initiales dont dépend l'égalité (7) est mis à jour. L'égalité (7) dépend des égalités initiales (1) et (3).

Nous ajoutons des arcs S de l'égalité (3) à l'égalité (1) et de l'égalité (3) à l'égalité (7), car ils ne créent pas de E-cycle. L'égalité (3) est E-cycle simplifiée.

Cette quatrième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.4.

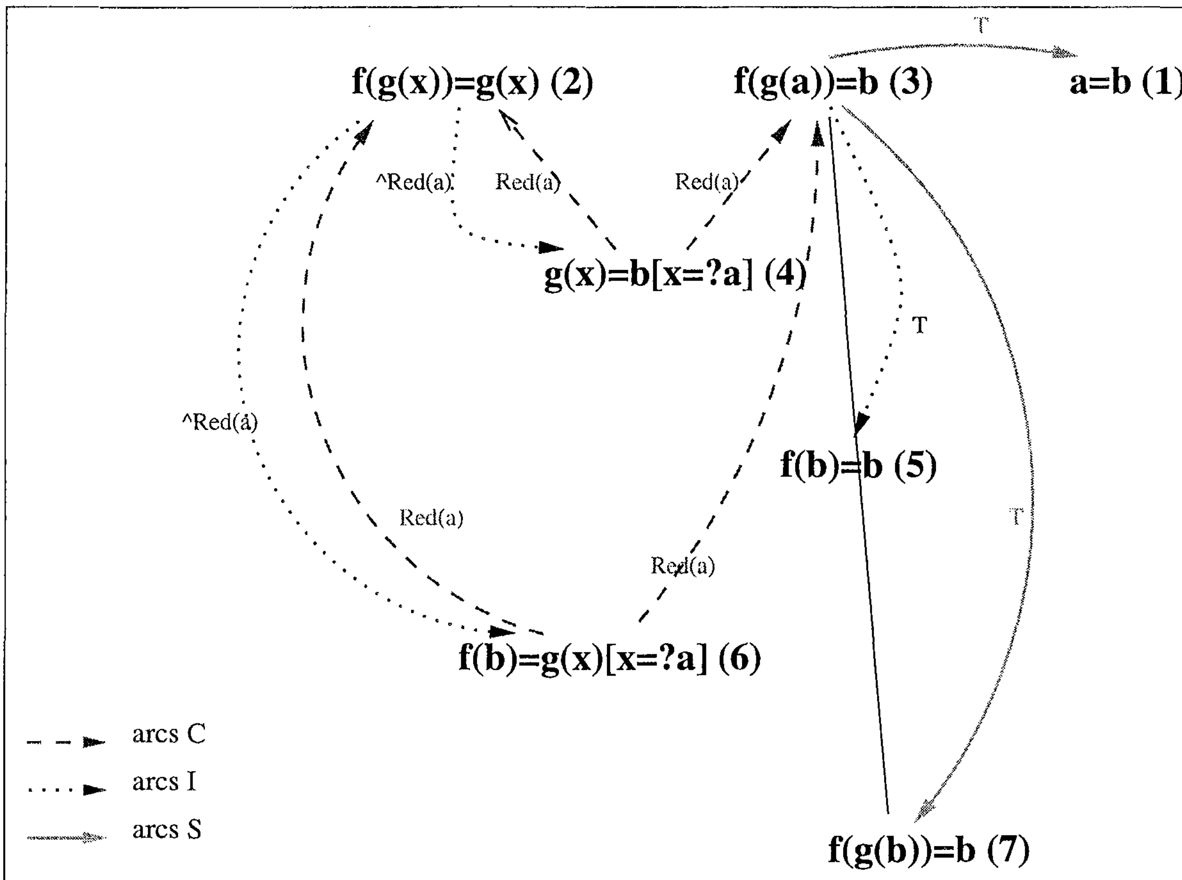


FIG. 4.4 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ *- Etape 4*

5.
$$\frac{f(g(x)) \approx g(x)\ (2) \qquad f(g(b)) \approx b\ (7)}{g(x) \approx b\, [\![x =^? b]\!]\ (8)}$$

Nous sommes en présence d'une inférence de paire critique basique.

Nous ajoutons des arcs C de l'égalité (8) aux égalités initiales (1), (2) et (3). La contrainte de réductibilité associée à ces arcs est Red(b).

Nous ajoutons un arc I de l'égalité (2) à l'égalité (8). La contrainte de réductibilité associée à cet arc est ¬Red(b).

Cette cinquième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.5.

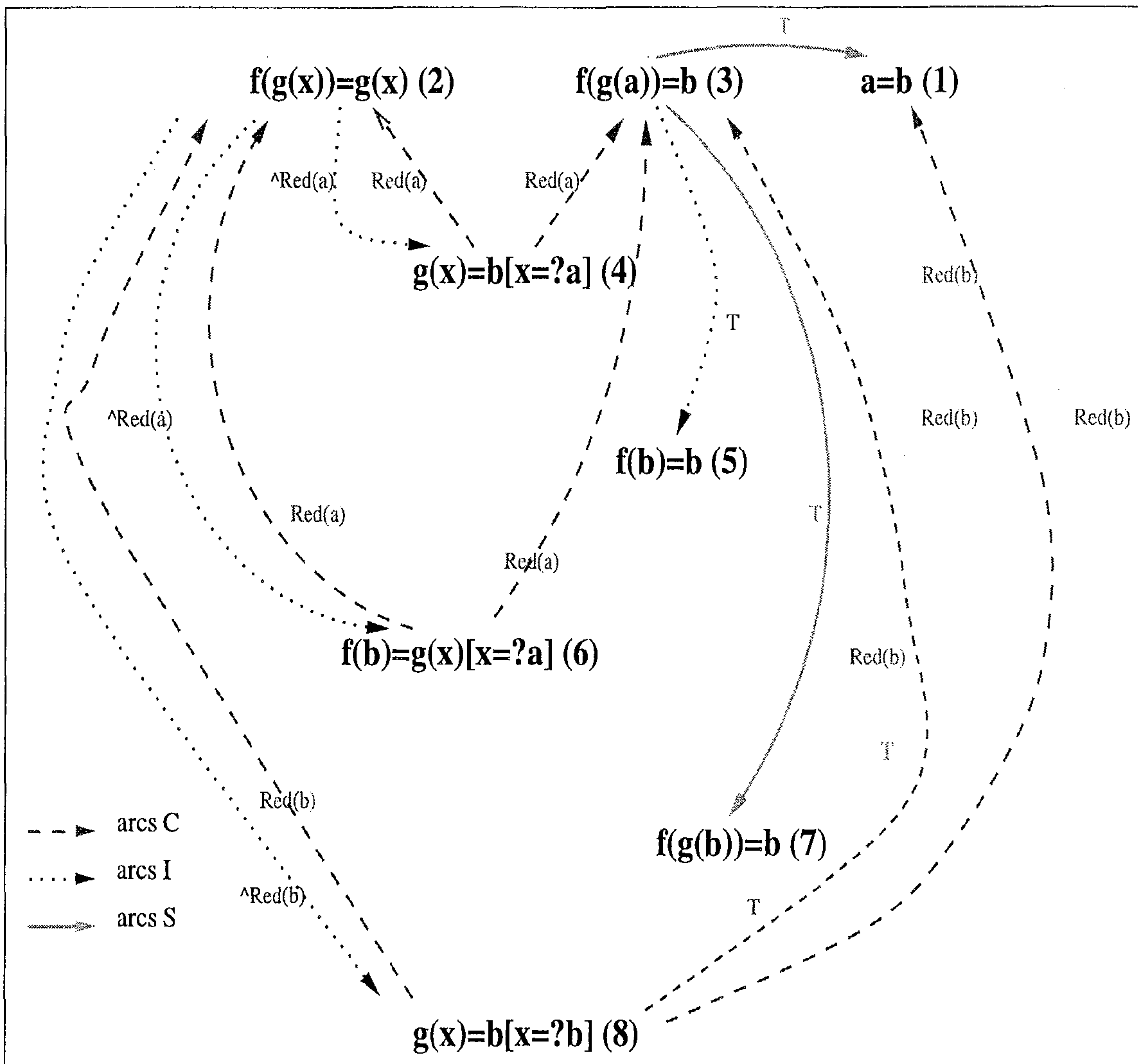


FIG. 4.5 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ *- Etape 5*

6. $$\frac{f(g(b)) \approx b \ (7) \qquad g(x) \approx b [\![x =^? b]\!] \ (8)}{f(b) \approx b \ (9)}$$

Nous sommes en présence d'une inférence de paire critique basique.

Nous n'ajoutons pas d'arc C car l'égalité (9) est une égalité non contrainte. Cependant, l'ensemble des égalités initiales dont dépend l'égalité (9) est mis à jour. L'égalité (9) dépend des égalités initiales (1), (2) et (3).

Nous ajoutons un arc I de l'égalité (7) à l'égalité (9). La contrainte de réductibilité associée à cet arc est $\top$.

L'égalité (7) peut être standard simplifiée par l'égalité (8). Cependant, l'égalité (7) ne peut pas être E-cycle simplifiée par l'égalité (8). En effet, si nous ajoutons des arcs S, un E-cycle est créé. L'arc S de l'égalité (7) à l'égalité (8) (dont la contrainte de réductibilité est $\top$*), l'arc C de l'égalité (8) à l'égalité (3) (dont la contrainte de réductibilité est Red(b)) et l'arc S de l'égalité (3) à l'égalité (7) (dont la contrainte de réductibilité est* $\top$*) décrivent un E-cycle.*

Cette sixième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.6.

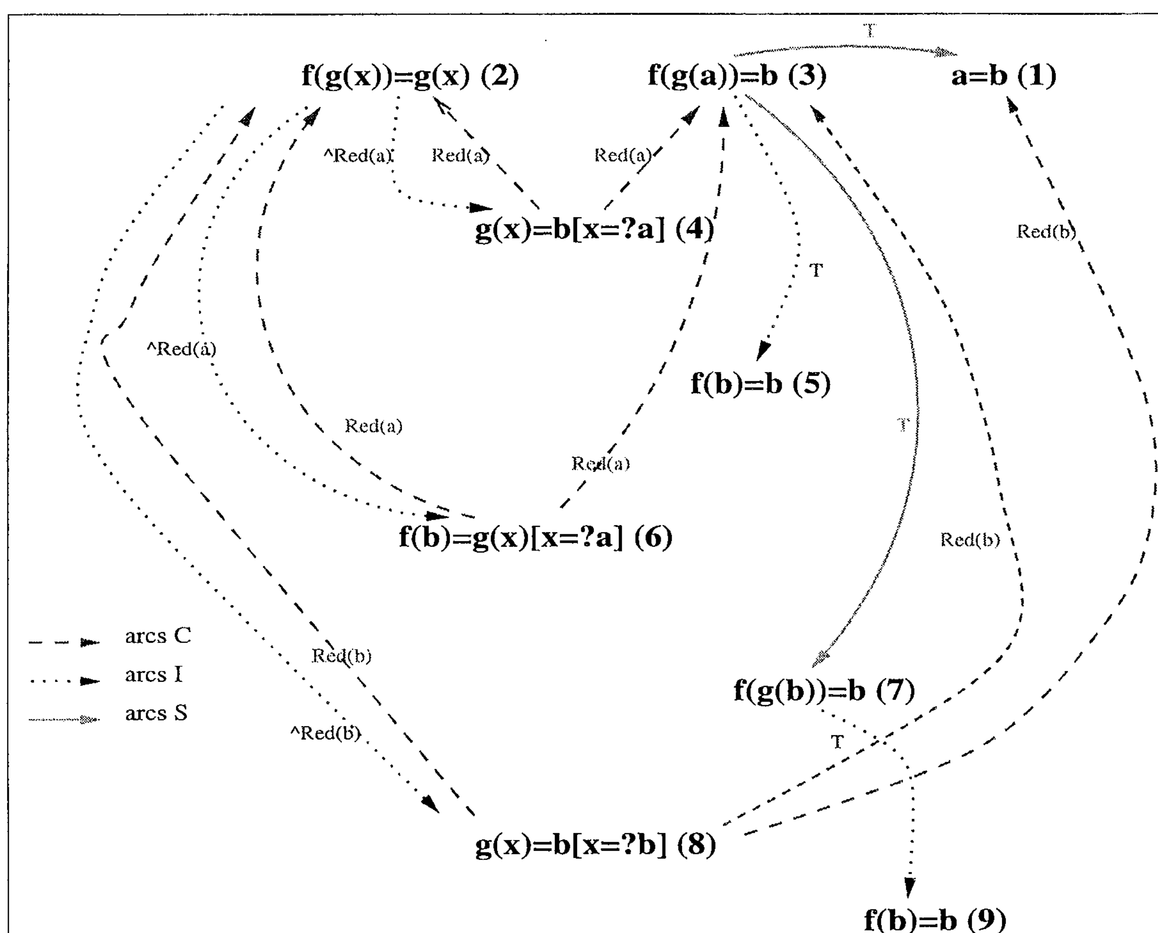


FIG. 4.6 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b \ (1), f(g(x)) \approx g(x) \ (2), f(g(a)) \approx b \ (3)\}$ *- Etape 6*

7. $$\frac{f(g(x)) \approx g(x)\ (2) \qquad g(x) \approx b\,[\![x =^? b]\!]\ (8)}{f(b) \approx g(x)\,[\![x =^? b]\!]\ (10)}$$

Nous sommes en présence d'une inférence de paire critique basique.

Nous ajoutons des arcs C de l'égalité (10) aux égalités initiales (1), (2) et (3). La contrainte de réductibilité associée à ces arcs est Red(b).

Nous ajoutons un arc I de l'égalité (2) à l'égalité (10). La contrainte de réductibilité associée à cet arc est $\neg Red(b)$.

Cette septième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.7.

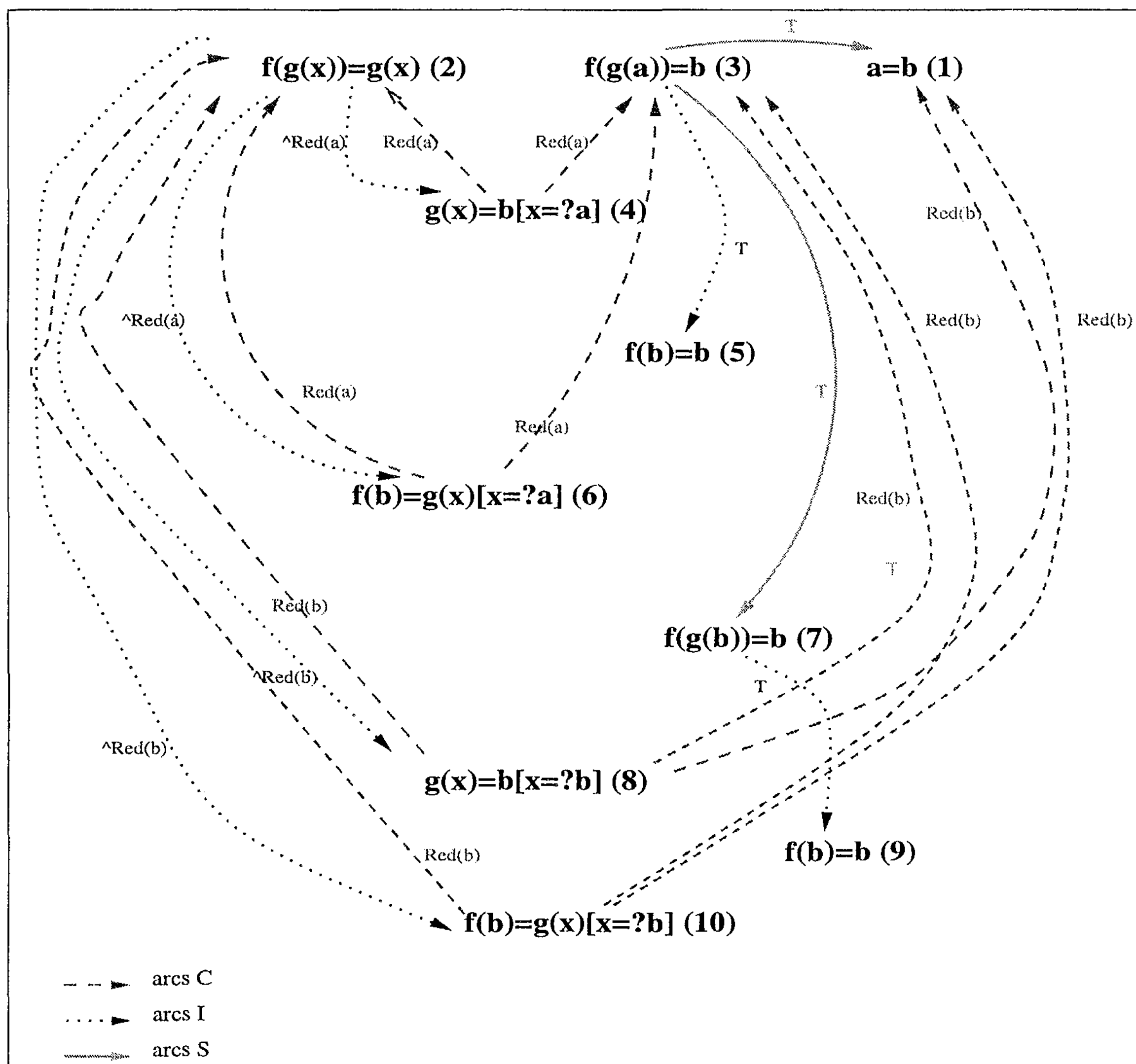


FIG. 4.7 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ *- Etape 7*

8. $$\frac{f(b) \approx b\ (9) \qquad f(b) \approx b\ (5)}{b \approx b}$$

Nous sommes en présence d'une inférence de simplification E-cycle.

L'égalité triviale $b \approx b$ n'est pas ajoutée au graphe de dépendance. Elle ne sera jamais impliquée dans un E-cycle.

L'égalité (9) peut être standard simplifiée par l'égalité (5). Nous ajoutons un arc S de l'égalité (9) à l'égalité (5). En effet, l'addition de cet arc S ne crée pas de E-cycle. L'égalité (9) est E-cycle simplifiée.

Cette huitième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.8.

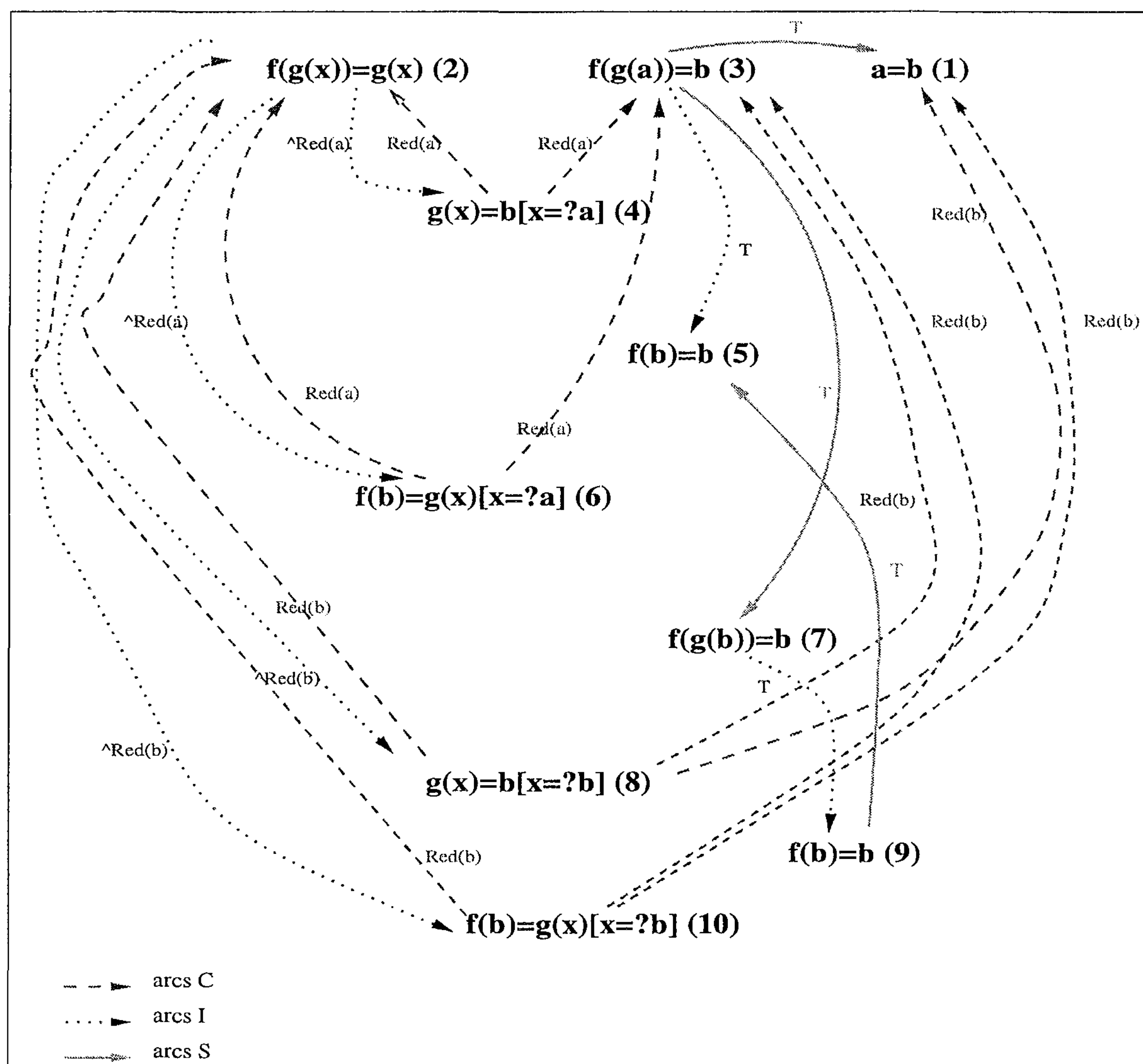


FIG. 4.8 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b\ (1), f(g(x)) \approx g(x)\ (2), f(g(a)) \approx b\ (3)\}$ *- Etape 8*

9. $$\frac{f(b) \approx g(x) \llbracket x =^? b \rrbracket \ (10) \qquad f(b) \approx b \ (5)}{g(x) \approx b \llbracket x =^? b \rrbracket \ (11)}$$

Nous sommes en présence d'une inférence de simplification E-cycle.
Nous ajoutons des arcs C de l'égalité (11) aux égalité initiales (1), (2) et (3). La contrainte de réductibilité associée à ces arc est Red(b).
L'égalité (10) peut être standard simplifiée par l'égalité (5). Nous ajoutons des arcs S de l'égalité (10) aux égalités (5) et (11). En effet, l'addition de ces arcs S ne crée pas de E-cycle. L'égalité (10) est E-cycle simplifiée.
Cette neuvième étape dans la construction du graphe de dépendance de la complétion de E est visible sur la figure 4.9.

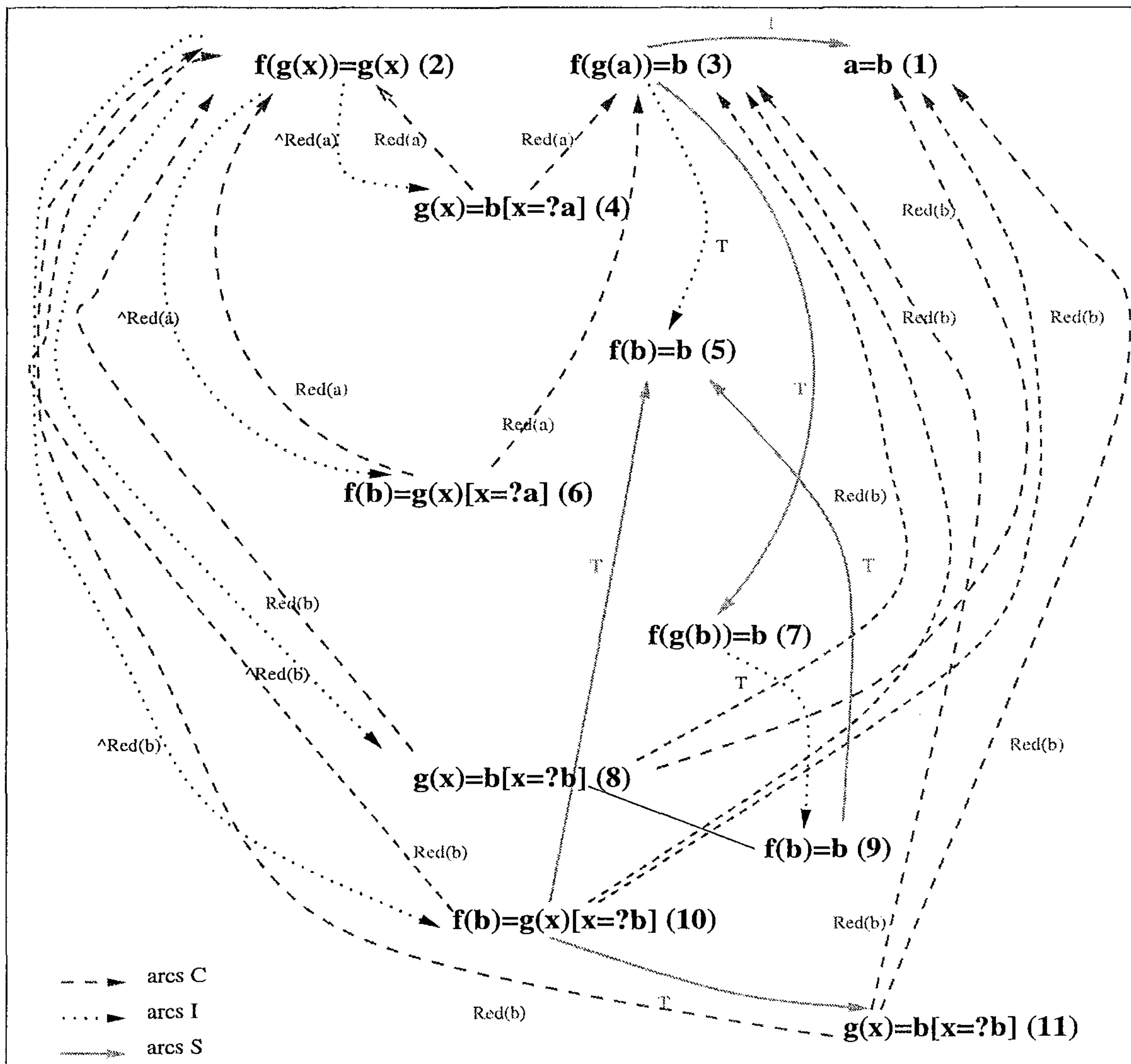


FIG. 4.9 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{a \approx b \ (1), f(g(x)) \approx g(x) \ (2), f(g(a)) \approx b \ (3)\}$ *- Etape 9*

10. $$\frac{g(x) \approx b \llbracket x =^? b \rrbracket \ (11) \qquad g(x) \approx b \llbracket x =^? b \rrbracket \ (8)}{b \approx b}$$

Nous sommes en présence d'une inférence de simplification E-cycle.

L'égalité triviale $b \approx b$ n'est pas ajoutée au graphe de dépendance.

L'égalité (11) peut être E-cycle simplifiée par l'égalité (8). En effet, l'addition d'un arc S de l'égalité (11) à l'égalité (8) ne crée pas de E-cycle.

Le graphe de dépendance final est visible sur la figure 4.10.

L'ensemble saturé est $E_\infty = \{a \approx b \ (1), f(g(x)) \approx g(x) \ (2), f(b) \approx b \ (5), f(g(b)) \approx b \ (7), g(x) \approx b \llbracket x =^? b \rrbracket \ (8)\}$ *(Nous supprimons de E_∞ les égalités avec des contraintes réductibles par rapport à E_∞, car elles ne créent pas de E-cycle).*

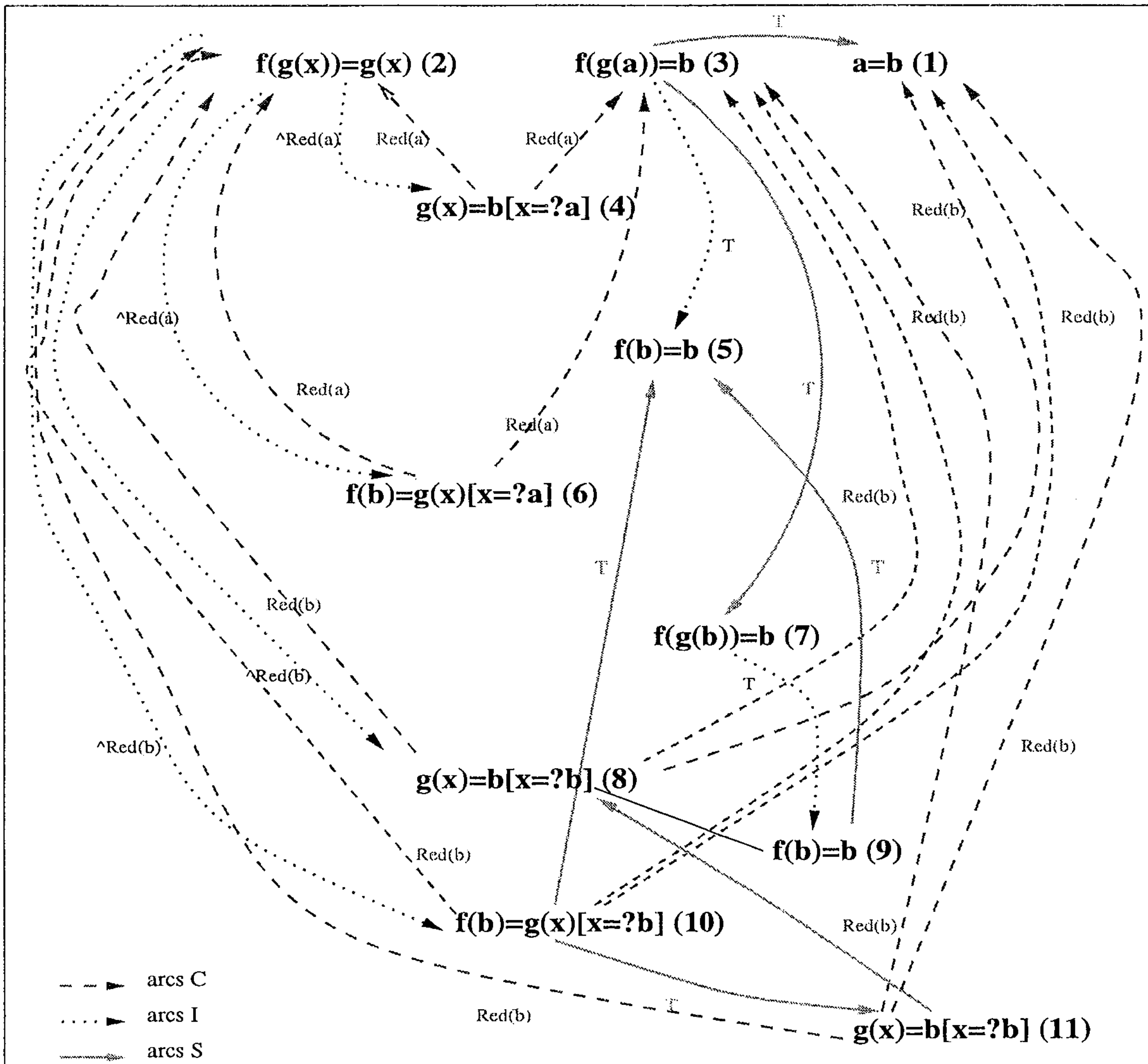


FIG. 4.10 – *Graphe de dépendance final de la complétion basique avec simplification E-cycle de* $E = \{a \approx b \ (1), f(g(x)) \approx g(x) \ (2), f(g(a)) \approx b \ (3)\}$

4.4 BCES est complet

Dans cette section, nous prouvons la complétude de *BCES*. Notre preuve de complétude est basée sur la technique de construction d'un modèle d'égalités de Herbrand de l'ensemble saturé des égalités (Bachmair et Ganzinger, 1994; Bachmair et al., 1995; Nieuwenhuis et Rubio, 1992a; Bofill et al., 1999). Les points importants de cette méthode de preuve de complétude ont été décrits dans la section 2.2. En particulier, cette méthode nécessite l'utilisation d'un ordre de réduction total sur les termes clos $\succ$. L'originalité de notre preuve réside ici dans le fait que $\succ$ est instancié par l'ordre $\succ_g$ construit directement à partir du graphe de dépendance clos GG_∞, instance du graphe de dépendance G_∞ construit avec le système d'inférence *BCES*.

4.4.1 Graphe de dépendance clos

Nous allons d'abord donner à l'aide d'un exemple les raisons qui nous ont poussés à définir la notion de graphe de dépendance clos.

Exemple 35 *Soit E l'ensemble d'égalités $\{h(f(a)) \approx a,\ f(x) \approx g(x)\}$ que nous voulons compléter en utilisant les règles d'inférence de* BCES. *Considérons l'inférence de paire critique basique suivante :*

$$\frac{h(f(a)) \approx a \qquad f(x) \approx g(x)}{h(g(x)) \approx b\, [\![x =^? a]\!]}$$

Si nous mettons à jour le graphe de dépendance suite à la réalisation de cette inférence, nous ajoutons un sommet d'étiquette $h(g(x)) \approx b\, [\![x =^? a]\!]$.

Nous ajoutons deux arcs C, un de l'égalité $h(g(x)) \approx b\, [\![x =^? a]\!]$ à l'égalité $h(f(a)) \approx a$ et l'autre de l'égalité $h(g(x)) \approx b\, [\![x =^? a]\!]$ à l'égalité $f(x) \approx g(x)$. La contrainte de réductibilité associée à ces arcs est $EqToRedConst(x =^? a) = Red(a)$.

Nous ajoutons également un arc I de l'égalité $h(f(a)) \approx a$ à l'égalité $h(g(x)) \approx b\, [\![x =^? a]\!]$, dont la contrainte de réductibilité est $\neg EqToRedConst(x =^? a) = \neg Red(a)$.

Cependant, l'arc C de l'égalité $h(g(x)) \approx b\, [\![x =^? a]\!]$ à l'égalité $f(x) \approx g(x)$ représente en fait un arc C de l'égalité $h(g(a)) \approx b$ instance close de $h(g(x)) \approx b\, [\![x =^? a]\!]$ à l'égalité $f(a) \approx g(a)$ instance close de $f(x) \approx g(x)$.

C'est pourquoi, nous considérons les instances closes des égalités, des inférences et des graphes de dépendance.

4.4.1.1 Description sommaire des graphes de dépendance clos

Dans les graphes de dépendance clos, les étiquettes des sommets sont des égalités closes. Les arcs sont ajoutés seulement si les contraintes de réductibilité qui leur sont associées sont satisfaisables dans un système de réécriture clos particulier défini plus loin. Dans un graphe de dépendance clos, nous ne parlons plus de E-cycle mais simplement de cycle.

Nous définissons GG_{init} pour un ensemble d'égalités (non clos) E à compléter. GG_{init} est le graphe de dépendance clos initial. $GG_{init} = (V_{init}, ED_{init})$, où V_{init} est l'ensemble des sommets tels que $e \in Gr(E)$ soit l'étiquette d'un sommet de V_{init} et $ED_{init} = \emptyset$. Comme dans le cas non clos, nous définissons l'ensemble *Cancestor*. $Cancestor(v) = \{v\}$ pour tout $v \in V_{init}$.

Les arcs sont ajoutés dans le graphe de dépendance clos de la manière suivante :

- Les arcs C sont ajoutés des instances closes des égalités contraintes aux instances closes des égalités initiales de E non contraintes.
- Un arc I est ajouté comme dans le cas clos d'une prémisse « gauche » à la conclusion d'une inférence.

- Nous ajoutons aussi un arc I de la prémisse « gauche » à la prémisse « droite » d'une inférence.
- Nous ajoutons également des arcs I des égalités « gauches » aux égalités « droites » et aux égalités conclusions des inférences closes qui représentent des inférences à des positions de variables pas dans la contrainte dans le cas non clos.[11]

 Exemple 36 – *Considérons un ensemble d'égalités E contenant les égalités $h(f(x)) \approx b$ et $a \approx b$, telles que $a \approx b$ soit irréductible par une égalité plus petite de E. L'égalité $h(f(a)) \approx b$ est une instance close de $h(f(x)) \approx b$. Il existe une inférence entre $h(f(a)) \approx b$ et $a \approx b$ qui nous permet de déduire l'égalité $h(f(b)) \approx b$. Cette inférence simule une inférence de paire critique à une position de variable pas dans le contrainte dans le cas non clos.*

 Des arcs I sont donc ajoutés du sommet étiqueté par $h(f(a)) \approx b$ aux sommets étiquetés par $a \approx b$ et $h(f(b)) \approx b$.

 – *Considérons un ensemble d'égalités E contenant les égalités $h(f(x)) \approx b\,[\![x =^? a]\!]$ et $a \approx b$, telles que $a \approx b$ soit irréductible par une égalité plus petite de E. L'égalité $h(f(a)) \approx b$ est une instance close de $h(f(x)) \approx b\,[\![x =^? a]\!]$. Il existe une inférence entre $h(f(a)) \approx b$ et $a \approx b$ qui nous permet de déduire $h(f(b)) \approx b$. Cette inférence simule une inférence de paire critique à une position de variable dans la contrainte dans le cas non clos.*

 Aucun arc I n'est ajouté dans ce cas.

4.4.1.2 Les transitions de graphes clos modulo un ensemble d'égalités

Les conséquences de l'application des règles d'inférence sur le graphe de dépendance clos sont formalisées par des *transitions de graphes clos modulo E'*, où E' est l'ensemble des égalités closes dans lequel la satisfaisabilité des contraintes de réductibilité est testée. Dans la preuve de complétude, cet ensemble sera instancié par $Gr(E_\infty)$, si E est l'ensemble des égalités à compléter.

Nous avons besoin de passer du cas clos au cas non clos. C'est pourquoi nous parlons de dérivations de transitions de graphes de dépendance clos associées à des dérivations de transitions de graphes de dépendance (non clos).

La définition 44 est une extension de la définition 41 dans le cas de graphes de dépendance clos.

Définition 44 *Soit $(E_0 = E, G_0 = G_{init}) \to (E_1, G_1) \to \ldots$ une dérivation de transitions de graphes partant de E. Alors, $(GE_0 = Gr(E), GG_0 = GG_{init}) \to (GE_1 = Gr(E_1), GG_1) \to \ldots$ est sa dérivation de transitions de graphes clos modulo associée, où pour i, GG_{i+1} est défini de la manière suivante :*

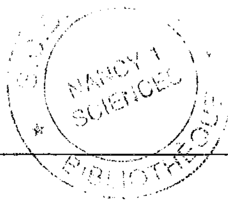
1. *Supposons que E_{i+1} est créé à partir de E_i par une inférence de paire critique basique.*
 Alors, GE_i est $\{e_0\,[\![c_0]\!], e_1\,[\![c_1]\!]\} \cup \Gamma$, et GE_{i+1} est $\{e_0\,[\![c_0]\!], e_1\,[\![c_1]\!], e_2\,[\![c_2]\!]\} \cup \Gamma$, où Γ est un ensemble d'égalités, et $e_2\,[\![c_2]\!]$ est la conclusion de l'inférence de paire critique basique entre $e_0\,[\![c_0]\!]$ comme égalité « gauche » et $e_1\,[\![c_1]\!]$ comme égalité « droite ».
 Soit $e_i\,[\![c_i]\!]$ égal à $s_i \approx t_i\,[\![c_i]\!]$ pour $i \in \{0,1,2\}$.
 Soit $\Sigma = \{\sigma \mid \sigma \in Sol_G^{e_0}(c_0) \cap Sol_G^{e_1}(c_1)\}$. L'ensemble des instances closes de l'inférence de paire critique basique est $\{\sigma(e_0), \sigma(e_1) \mid \sigma \in \Sigma\} \cup \Gamma' \to \{\sigma(e_0), \sigma(e_1), \sigma(e_2) \mid \sigma \in \Sigma\} \cup \Gamma'$, où Γ' contient les instance closes de $e_0\,[\![c_0]\!]$ et $e_1\,[\![c_1]\!]$, $\sigma'(e_0)$ et $\sigma''(e_1)$ telles que $\sigma' \notin \Sigma$

11. Dans le cas non clos, aucune inférence n'est réalisée à une position de variable. Cependant, nous parlons d'inférence dans le cas présent par abus de langage. Cette remarque est valide pour tout cette section.

et $\sigma'' \notin \Sigma$, et les éléments de $Gr(\Gamma)$. Pour σ donné, $\sigma(e_0)$ est l'étiquette du sommet u_σ, $\sigma(e_1)$ est l'étiquette du sommet v_σ, et $\sigma(e_2)$ est l'étiquette du sommet w_σ.

- $V_{i+1} = V_i \cup \{w_\sigma \mid \sigma \in \Sigma\}$
- $ED_{i+1} = ED_i \cup E_C \cup E_I \cup E_V$ *où :*
  - $E_C = \{(w_\sigma,v,C) \mid \sigma \in \Sigma,\ v \in Cancestor(u_\sigma) \cup Cancestor(v_\sigma)$, *et* $EqToRedConst(\sigma(c_2))$ *est satisfaisable dans* $E'\}$
  - $E_I = \{(u_\sigma,v_\sigma,I),(u_\sigma,w_\sigma,I) \mid \sigma \in \Sigma,\ \sigma(e_0) \succ_e \sigma(e_1),\ \sigma(e_0) \succ_e \sigma(e_2)$ *et* $\neg EqToRedConst(\sigma(c_2)) \wedge Red(\sigma(s_0),\sigma(e_0)) \wedge \neg Red(\sigma(s_1),\sigma(e_1))$ *est satisfaisable dans* $E'\}$
  - E_V *est défini ci-dessous en 4.*
- $Cancestor(w_\sigma)$ *est égal à* $w_\sigma \cup Cancestor(u_\sigma) \cup Cancestor(v_\sigma)$.

2. *Supposons que E_{i+1} est créé à partir de E_i par une règle de suppression.*
   Supposons que E_i est $\{e_0,e_1,\ldots,e_n\} \cup \Gamma$ et E_{i+1} est $\{e_1,e_2,\ldots,e_n\} \cup \Gamma$, où Γ est un ensemble d'égalités. e_0 est supprimé par une règle de suppression à cause de $e_1,\ldots,e_n$.
   Soit Σ l'ensemble des substitutions closes instances des filtres utilisés pour les suppressions. L'ensemble des instances closes de la règle de suppression est $\{\sigma(e_0),\sigma(e_1),\ldots,\sigma(e_n) \mid \sigma \in \Sigma\} \cup \Gamma' \to \{\sigma(e_1),\sigma(e_2),\ldots,\sigma(e_n) \mid \sigma \in \Sigma\} \cup \Gamma'$, où Γ' contient les autres instances des e_j, $j \in \{0,\ldots,n\}$, et les éléments de $Gr(\Gamma)$. Pour $j \in \{0,\ldots,n\}$, et pour σ donné, soit $\sigma(e_j)$ l'étiquette du sommet v_{j_σ}.
   - $V_{i+1} = V_i$
   - $ED_{i+1} = (ED_i - E_I) \cup E_S$ *où :*
     - $E_I = \{e_d \mid e_d$ *est un arc I de* ED_i *d'une égalité « gauche » à une égalité « droite » d'une inférence suivie par un arc S de* $E_S\}$
     - $E_S = \bigcup_{j\in\{1,\ldots,n\}} \{(v_{0\sigma},v_{j_\sigma},S) \mid \sigma(e_j) \in \Sigma\}$.
3. *Supposons que E_{i+1} est crée à partir de E_i par une règle de rétraction.*
   Soit E_i égal à $\{e_1\} \cup \Gamma$ et E_{i+1} égal à $\{e'_1\} \cup \Gamma$, où e'_1 est un rétracte de e_1 et Γ est un ensemble d'égalités.
   Soit Σ l'ensemble des substitution closes qui rendent l'égalité e_1 close. L'ensemble des instances closes de la règle de rétraction est $\{\sigma(e_1) \mid \sigma \in \Sigma\} \cup Gr(\Gamma) \to \{\sigma(e'_1) \mid \sigma \in \Sigma\} \cup Gr(\Gamma)$. Pour σ donné, soit $\sigma(e_1)$ l'étiquette de $v_{1\sigma}$. L'étiquette de $\sigma(e_1)$ devient $\sigma(e'_1)$.
   - $V_{i+1} = V_i$
   - $ED_{i+1} = ED_i \cup E_V - E_C$ *où :*
     - $E_C = \{e_d \mid e_d$ *est un arc C sortant de* $v_{1\sigma}\}$, *si* e'_1 *est une égalité contrainte, sinon* $E_C = \emptyset$.
     - E_V *est défini plus loin en 4.*
4. *E_V est l'union de tous les ensembles E_I pour toutes les inférences de paires critiques à des positions de variables pas dans la contrainte.*
   Considérons une inférence de paire critique à une position de variable pas dans la contrainte entre une instance de e_0 $[\![c_0]\!]$ comme égalité « gauche » et une instance de e_1 $[\![c_1]\!]$ comme égalité « droite ». La conclusion de l'inférence est une instance de e_2 $[\![c_2]\!]$. Soit e_i $[\![c_i]\!]$ égal à $s_i \approx t_i$ $[\![c_i]\!]$ pour $i \in \{0,1,2\}$.
   Soit $\sigma(e_0)$, $\sigma(e_1)$ et $\sigma(e_2)$ des instances closes de e_0 $[\![c_0]\!]$, e_1 $[\![c_1]\!]$ et e_2 $[\![c_2]\!]$ telles que $\sigma \in Sol_G^{e_0}(c_0) \cap Sol_G^{e_1}(c_1)$. Pour σ donné, $\sigma(e_0)$ est l'étiquette du sommet u_σ, $\sigma(e_1)$ est l'étiquette du sommet v_σ et $\sigma(e_2)$ est l'étiquette du sommet w_σ.



Alors :

- $E_I = \{(u_\sigma,w_\sigma,I),(u_\sigma,v_\sigma,I) \mid \sigma \in Sol_G^{e_0}(c_2) \cap Sol_G^{e_1}(c_2),\ \sigma(e_0) \succ_e \sigma(e_1),\ \sigma(e_0) \succ_e \sigma(e_2)$ *and* $\neg EqToRedConst(\sigma(c_2)) \wedge Red(\sigma(s_0),\sigma(e_0)) \wedge \neg Red(\sigma(s_1),\sigma(e_1))$ *est satisfaisable dans* $E'\}$. □

4.4.2 Preuve de complétude

Dans cette section, nous donnons d'abord un résultat de complétude concernant les dérivations de transitions de graphes clos et ensuite nous prouvons la complétude de *BCES*. Nous donnons une suite de théorèmes et de lemmes qui vont nous permettre de prouver nos deux théorèmes de complétude.

Le théorème suivant montre que le cycle d'un graphe de dépendance clos contient au moins un arc C et un arc S. La preuve est faite par contradiction.

Théorème 15 *Soit* $(EG_0 = Gr(E),GG_0 = GG_{init}) \to (EG_1,GG_1) \to \ldots \to (EG_n,GG_n)\ldots$ *une dérivation de transitions de graphes clos modulo* E'. *Si* GG_n *contient un cycle, alors ce cycle contient au moins (a) un arc C et (b) un arc S.*

Preuve : Soit $\mathcal{C}_g$ un cycle présent dans GG_n.

(a) $\mathcal{C}_g$ ne contient pas seulement des arcs I et des arcs S, car les arcs I et S vont toujours vers des égalités plus petites par rapport à $\prec_e$ dans le graphe de dépendance clos et $\prec_e$ est bien fondé. Ainsi, les arcs I et S ne peuvent former un cycle et en particulier $\mathcal{C}_g$. La contradiction prouve que $\mathcal{C}_g$ contient au moins un arc C.

(b) Si $\mathcal{C}_g$ ne contient pas d'arc S, alors par le théorème 13 et par (a), il contient des arcs C et des arcs I. Ainsi, $\mathcal{C}_g$ doit contenir un sous-chemin avec un arc I suivi d'un arc C. Or, le théorème 12 prouve que ce cas ne peut arriver[12]. La contradiction prouve que $\mathcal{C}_g$ ne contient pas seulement des arcs C et I. Ainsi, $\mathcal{C}_g$ contient au moins un arc S. □

Le lemme suivant prouve que si un arc I va d'un sommet v_1 à un sommet v_2 dans GG_∞, alors $label(v_1)$ est réductible dans $Gr(E_\infty)$.

Lemme 4 *Soit* $(EG_0 = Gr(E),GG_0) \to (EG_1,GG_1) \to \ldots \to (EG_n,GG_n)\ldots$ *une dérivation de transitions de graphes clos modulo* $Gr(E_\infty)$. *S'il existe un arc du sommet* v_1 *au sommet* v_2 *dans* GG_∞*, alors* $label(v_1)$ *est réductible dans* $Gr(E_\infty)$.

Preuve : Soit $label(v_1) = \sigma(s_1) \approx \sigma(t_1)$ tel que $\sigma(s_1) \approx \sigma(t_1)$ soit une instance close de l'égalité $s_1 \approx t_1 [\![c_1]\!]$.

S'il existe un arc I du sommet v_1 au sommet v_2 dans GG_∞, alors $Red(\sigma(s_1),\sigma(s_1) \approx \sigma(t_1))$ est satisfaisable dans $Gr(E_\infty)$. Donc, $\sigma(s_1)$ est réductible dans $Gr(E_\infty)$ et $\sigma(s_1) \approx \sigma(t_1)$ l'est aussi. □

Le lemme suivant prouve qu'un cycle dans un graphe de dépendance clos ne contient pas d'arc I d'une égalité « gauche » à une égalité « droite » dans une inférence.

Lemme 5 *Soit* $(EG_0 = Gr(E),GG_0 = GG_{init}) \to (EG_1,GG_1) \to \ldots \to (EG_n,GG_n)\ldots$ *une dérivation de transitions de graphes clos modulo* $Gr(E_\infty)$. *Si* GG_i *est un graphe de dépendance contenant un cycle* $\mathcal{C}_g$*, alors* $\mathcal{C}$ *ne contient pas d'arc I de l'égalité « gauche » à l'égalité « droite » d'une inférence.*

12. Les théorèmes 12, 13 et 14 ont été donnés pour le cas non clos. Mais, avec la même logique, ils s'appliquent également dans la cas clos.

Preuve : Supposons que $\mathcal{C}_g$ contient un arc I e_d d'une égalité « gauche » $u[s] \approx v$ à une égalité « droite » $s \approx t$.

- D'après le théorème 12, e_d ne peut pas être suivi par un arc C dans $\mathcal{C}_g$.
- e_d ne peut pas être suivi par un autre arc I dans $\mathcal{C}_g$.
  Comme e_d est dans GG_i, $\neg Red(s,s \approx t) \wedge Red(u[s],u[s] \approx v)$ est satisfaisable dans $Gr(E_\infty)$. S'il existe un arc I sortant du sommet d'étiquette l'égalité $s \approx t$, alors $Red(s,s \approx t)$ est satisfaisable dans $Gr(E_\infty)$ (voir le lemme 4).
  Ainsi, e_d ne peut pas être suivi par un autre arc I dans $\mathcal{C}_g$, à cause de l'insatisfaisabilité de la conjonction des contraintes de réductibilité.
- e_d ne peut pas être suivi par un arc S dans $\mathcal{C}_g$.
  En effet, les arcs I des égalités « gauches » aux égalité « droites » suivis d'arcs S sont enlevés du graphe de dépendance clos pendant la construction du graphe de dépendance clos (voir la définition 44).

e_d ne peut être suivi d'aucun arc dans $\mathcal{C}_g$. Ainsi, aucun cycle ne contient un arc I d'une égalité « gauche » à une égalité « droite » dans une inférence. □

Le lemme suivant prouve que s'il existe un cycle ou un chemin infini dans le cas clos, alors il existe un E-cycle dans le cas non clos. Dans la preuve de ce lemme, nous avons basiquement besoin de prouver que les arcs supplémentaires ajoutés dans le cas clos ne créent pas de cycles qui n'existent pas déjà dans la cas non clos. Les lemmes 5 et le théorème 15 sont utilisés.

Lemme 6 *Soit* $(E_0 = E,G_0 = G_{init}) \to (E_1,G_1) \to \ldots \to (E_n,G_n)\ldots$ *une dérivation de transitions de graphes et* $(EG_0 = Gr(E),GG_0 = GG_{init}) \to (EG_1,GG_1) \to \ldots \to (EG_n,GG_n)\ldots$ *sa dérivation de transitions de graphes clos modulo* $Gr(E_\infty)$ *associée, alors pour tout* i*, (a) si* GG_i *contient un cycle, alors* G_i *contient un E-cycle et (b) si* GG_i *contient un chemin infini, alors* G_i *contient un E-cycle.*

Preuve : (a) Soit $\mathcal{C}_g$ un cycle de GG_i. Soit e_{di} les arcs de $\mathcal{C}_g$ pour $i \in \{1,\ldots,n\}$. Nous pouvons remarquer en utilisant le lemme 5 que e_{di} n'est pas un arc I d'une égalité « gauche » à une égalité « droite » d'une inférence. Pour $i \in \{1,\ldots,n-1\}$, e_{di} est d'une égalité e_i à une égalité e_{i+1} et e_{dn} est de l'égalité e_n à l'égalité e_0.

La contrainte de réductibilité associée à chaque arc e_{di} est satisfaisable dans $Gr(E_\infty)$. Le cycle $\mathcal{C}_g$ correspond donc à un E-chemin $\mathcal{C}'$ dans le cas non clos.

Nous allons montrer que $\mathcal{C}'$ est un cycle puis un E-cycle.

Les seuls arcs du cas clos qui n'apparaissent pas au niveau non clos sont les arcs des égalités « gauches » aux égalités « droites » et les arcs des égalités « gauches » aux égalités conclusion d'une inférence qui simulent une inférence à une position de variable pas dans la contrainte dans le cas non clos.

Le premier type d'arcs n'existe pas dans C_g d'après le lemme 5. Le second type d'arcs représente des boucles dans le cas non clos, car les égalités « gauches » et les égalités conclusions sont égales. Ainsi, $\mathcal{C}'$ est un cycle.

e_0 est une instance close de e'_0. Comme $\mathcal{C}_g$ existe, la conjonction des contraintes de réductibilité des arcs de $\mathcal{C}'$ n'est pas contradictoire syntaxiquement. D'après le théorème 15, $\mathcal{C}_g$ contient au moins un arc C et un arc S donc $\mathcal{C}'$ aussi. Ainsi, $\mathcal{C}'$ est un E-cycle.

(b) Nous prouvons maintenant que si GG_i contient un chemin infini $\mathcal{I}_g$, alors ce chemin contient un nombre infini d'arcs C. Nous prouvons alors que G_i contient un E-cycle.

Supposons que nous avons un chemin infini contenant les arcs $e_{d_1}, e_{d_2}, e_{d_3}, \ldots$ dans le graphe de dépendance clos GG_i qui contient seulement un nombre fini d'arcs C distincts. Comme

il y a un nombre fini d'arcs C, alors il y a un dernier arc C dans la séquence. Soit e_{d_n} ce dernier arc C. Alors, $e_{d_{n+1}}, e_{d_{n+2}}, \ldots$ est un chemin infini contenant seulement des arcs I et S. Ceci est impossible, car les arcs I et S vont toujours vers des égalités plus petites par rapport à $\prec_e$, et $\prec_e$ est bien fondé. La contradiction prouve qu'il y a un nombre infini d'arcs C distincts dans un chemin infini du graphe de dépendance clos GG_i.
Un arc C présent dans GG_i est également présent dans le graphe de dépendance G_i à cause de la satisfaisabilité de la contrainte de réductibilité qui lui est associée. Cependant, il est impossible pour $\mathcal{I}_{ng}$, le E-chemin associé à $\mathcal{I}_g$ dans le cas non clos, de contenir un nombre infini distinct d'arcs C, car les arcs C vont des égalités contraintes aux égalités initiales et il y a seulement un nombre fini d'égalités initiales. En effet, ceci veut dire que $\mathcal{I}_{ng}$ va à travers certains noeuds étiquetés par les égalités initiales un nombre infini de fois et si l'on traverse un noeud plus d'une fois dans G_i, alors nous avons un cycle dans G_i.
Comme $\mathcal{I}_g$ existe, la conjonction des contraintes de réductibilité des arcs de $\mathcal{I}_{ng}$ n'est pas contradictoire syntaxiquement. Comme $\mathcal{I}_g$ contient des arcs C et des arcs S, le cycle de G_i est un E-cycle. □

Nous donnons maintenant des définitions utiles dans la première preuve de complétude que nous allons faire : la preuve de complétude concernant les dérivations de transitions de graphes clos.

Définition 45 *$Irred(Gr(E_\infty))$ est un ensemble d'égalités orientées closes défini par :*
$Irred(Gr(E_\infty)) = \{\sigma(s) \to \sigma(t) \in Gr(s \approx t\ [\![c]\!]) \mid s \approx t\ [\![c]\!] \in E_\infty$ et $\sigma(s)$ est irréductible par une égalité plus petite que $\sigma(s) \approx \sigma(t)$ dans $Irred(Gr(E_\infty)) - \{\sigma(s) \to \sigma(t)\}\}$. □

Remarque 3 *$Irred(Gr(E_\infty)) \subseteq Gr(E_\infty)$.*

Lemme 7 *$Irred(Gr(E_\infty))$ est convergent.*

Preuve : $Irred(Gr(E_\infty))$ est clos, réduit à gauche par construction. Il est donc confluent. De plus, il est terminant car les règles de réécriture peuvent être orientées en utilisant $\succ_t$. Donc, il est convergent. □

Le principal théorème de complétude que nous donnons ici prouve la complétude des dérivations de transitions de graphes clos. La preuve de ce théorème est basée sur la construction d'un modèle d'égalités de Herbrand pour $Gr(E_\infty)$ qui est un système de réécriture clos convergent en utilisant l'ordre $\succ_g$ défini ci-dessous.

Définition 46 *Soit $\succ_g$ la relation définie par $e \succ_g e'$ si et seulement si il existe un chemin du sommet v au sommet v' dans GG_∞, tels que $label(v) = e$ et $label(v') = e'$.* □

Le lemme 8 montre que $\succ_g$ est un ordre bien fondé.

Lemme 8 *$\succ_g$ est un ordre de réduction bien fondé.*

Preuve : Il est évident que $\succ_g$ est réflexive et transitive. De plus, $\succ_g$ est anti-symétrique. En effet, la relation $\succ_g$ est construite à partir du graphe GG_∞ qui ne contient pas de cycle si nous utilisons la complétion basique avec simplification E-cycle (voir le lemme 6). D'après ce même lemme, GG_∞ ne contient pas de chemin infini, $\succ_g$ est donc bien fondé. $\succ_g$ est un ordre de réduction, car il est défini sur des égalités closes. □

On peut remarquer que $\succ_g$ n'est pas total, mais il a l'avantage d'être défini exactement sur les égalités utilisées dans la preuve de complétude comme nous allons le voir.

Nous définissons la notion de redondance en utilisant cet ordre.

Définition 47 *Une égalité close e est* g-redondante *dans un ensemble d'égalités closes E s'il existe des égalités* $e_1,\ldots,e_n \in E$ *telles que* $e_1,\ldots,e_n \models e$ *et* $e_i \prec_g e$ *pour tout i.* □

La simplification E-cycle et le blocage E-cycle sont des exemples de règles de simplification et de suppression basées sur la notion de g-redondance, comme nous l'exprimons dans le lemme suivant.

Lemme 9 *Soit E un ensemble d'égalités non contraintes. Soit* $(E_0 = E, G_0 = G_{init}) \to (E_1,G_1) \to \ldots \to (E_n,G_n)\ldots$ *une dérivation de transitions de graphes et* $(EG_0 = Gr(E), GG_0 = GG_{init}) \to (EG_1,GG_1) \to \ldots \to (EG_n,GG_n)\ldots$ *sa dérivation de transitions de graphes modulo* $Gr(E_\infty)$ *associée, où* GG_∞ *ne contient ni cycle ni chemin infini. Soit e une égalité qui est E-cycle simplifiée ou E-cycle bloquée dans un* E_i*. Alors, toutes les instances closes* e' *de e sont g-redondantes dans* $Gr(E_\infty)$*.*

Théorème 16 *Soit E un ensemble d'égalités non contraintes. Soit* $(EG_0 = Gr(E), GG_0 = GG_{init}) \to (EG_1,GG_1) \to \ldots \to (EG_n,GG_n)\ldots$ *une dérivation de transitions de graphes clos modulo* $Gr(E_\infty)$*, où* GG_∞ *ne contient pas de cycle et de chemin infini. Alors, cette dérivation de transitions de graphes clos est complète dans le sens où* $Gr(E_\infty)$ *est convergent.*

Preuve : Nous allons prouver que $Gr(E_\infty)$ est convergent en prouvant qu'il est terminant (**Etape 1**) et confluent (**Etape 2**).

Etape 1 : $Gr(E_\infty)$ est terminant, car ses égalités peuvent être orientées en utilisant $\succ_t$, qui est total sur les termes clos.

Etape 2 : Nous voulons appliquer le théorème 3. Nous allons prouver que $Gr(E_\infty)$ est confluent en prouvant que $Irred(Gr(E_\infty)) \subseteq Gr(E_\infty)$ (**Etape 2.1**), que $Irred(Gr(E_\infty))$ est confluent (**Etape 2.2**) et que $Irred(Gr(E_\infty)) \equiv Gr(E_\infty)$ (**Etape 2.3**).

Etape 2.1 : $Irred(Gr(E_\infty)) \subseteq Gr(E_\infty)$, à cause de la remarque 3.

Etape 2.2 : Le lemme 7 prouve que $Irred(Gr(E_\infty))$ est confluent.

Etape 2.3 : Nous allons prouver que $Irred(Gr(E_\infty)) \equiv Gr(E_\infty)$ en prouvant que $Gr(E_\infty) \models Irred(Gr(E_\infty))$ (**Etape 2.3.1**) et que $Irred(Gr(E_\infty)) \models Gr(E_\infty)$ (**Etape 2.3.2**).

Etape 2.3.1 : $Gr(E_\infty) \models Irred(Gr(E_\infty))$ est trivial, car $Irred(Gr(E_\infty)) \subseteq Gr(E_\infty)$.

Etape 2.3.2 : Nous allons prouver que $Irred(Gr(E_\infty)) \models Gr(E_\infty)$ en prouvant que pour toutes les règles de réécriture closes $s \to t$, si $Irred(Gr(E_\infty)) \models s \approx t$, alors $Gr(E_\infty) \models s \approx t$. Ceci peut être fait en prouvant que pour toute règle de réécriture close $s \to t$, si $s \to t \in Gr(E_\infty)$, alors $Irred(Gr(E_\infty)) \models s \approx t$.

Soit $s \to t$ une règle de réécriture $\sigma(s') \to \sigma(t')$, instance close d'une égalité $s' \approx t' [\![c]\!]$ de E_∞. Nous allons prouver que $Irred(Gr(E_\infty)) \models s \approx t$ par récurrence.

L'hypothèse de récurrence est la suivante : Pour toute égalité close e' telle que $e' \prec_g s \approx t$, si $e' \in Gr(E_\infty)$, alors $Irred(Gr(E_\infty)) \models e'$.

Cas 1 : Supposons que $s \to t \in Irred(Gr(E_\infty))$.
Trivialement, $Irred(Gr(E_\infty)) \models s \approx t$.

Cas 2 : Supposons que $s \to t \notin Irred(Gr(E_\infty))$.

Cas 2.1 : Supposons que σ est irréductible dans $Gr(E_\infty)$ et que $\sigma(c)$ est irréductible dans $Gr(E_\infty)$.

Comme $s \to t \notin Irred(Gr(E_\infty))$, il existe une plus petite règle de réécriture $u \to v$ dans $Irred(Gr(E_\infty))$ qui réduit s. Alors, u est irréductible dans $Gr(E_\infty)$. En effet, sinon, il existe une plus petite règle de réécriture qui réduit s. Ainsi, s

est de la forme $s = s[u]_p$. Comme σ est irréductible dans $Gr(E_\infty)$, il existe une inférence de paire critique qui peut être transposée dans le cas non clos, car cette inférence n'est pas à une position de variable. Cette inférence est :

$$\frac{s[u] \approx t \qquad u \approx v}{s[v] \approx t}$$

Comme E_∞ est saturé, il existe un arc I de $s[u] \approx t$ à $u \approx v$. Cet arc ne peut être supprimé, car nous ne pouvons ajouter un arc S sortant de $u \approx v$, car $u \approx v$ est irréductible dans $Gr(E_\infty)$ (voir le lemme 4).

Il y a des arcs I du sommet étiqueté par $s[u] \approx t$ au sommet étiqueté par $u \approx v$ et également, au sommet étiqueté par $s[v] \approx t$. $Red(s[u],s[u] \approx t) \wedge \neg Red(u,u \approx v) \wedge \neg EqToRedConst(\sigma(c))$ est satisfaisable dans $Gr(E_\infty)$. Ainsi, en utilisant l'ordre $\succ_g$, nous obtenons que : $s[v] \approx t \prec_g s[u] \approx t$ et $u \approx v \prec_g s[u] \approx t$.

Cas 2.1.a : Supposons que $s[v] \to t \in Gr(E_\infty)$.

Comme $s[v] \to t \in Gr(E_\infty)$ et $s[v] \approx t \prec_g s[u] \approx t$ et par l'hypothèse de récurrence, nous déduisons que $Irred(Gr(E_\infty)) \models s[v] \approx t$. De plus, $Irred(Gr(E_\infty)) \models u \approx v$ (parce que $u \to v \in Irred(Gr(E_\infty))$). Ainsi, $Irred(Gr(E_\infty)) \models s[u] \approx t$.

Cas 2.1.b : Supposons que $s[v] \to t \notin Gr(E_\infty)$.

$s[v] \to r$ n'est pas dans $Gr(E_\infty)$, car il est g-redondant dans $Gr(E_i)$ pour un i et donc $Gr(E_\infty)$ (voir le lemme 9). Ainsi, $s[v] \approx t$ est impliqué par des égalités plus petites de $Gr(E_\infty)$. Il existe $e_1 \in Gr(E_\infty), \ldots, e_n \in Gr(E_\infty)$ telles que $\forall i, e_i \prec_g s[v] \approx t$ et $e_1, \ldots, e_n \models s[v] \approx t$.

Comme $e_i \in Gr(E_\infty)$ et $e_i \prec_g s[u] \approx t$, nous pouvons appliquer l'hypothèse de récurrence aux e_i. $e_i \in Gr(E_\infty)$ implique que $Irred(Gr(E_\infty)) \models e_i$. Or, $e_1, \ldots, e_n \models s[v] \approx t$, donc par transitivité, $Irred(Gr(E_\infty)) \models s[v] \approx t$.

Comme $Irred(Gr(E_\infty)) \models s[v] \approx t$ et $Irred(Gr(E_\infty)) \models u \approx v$ (car $u \to v \in Irred(Gr(E_\infty))$), nous déduisons que $Irred(Gr(E_\infty)) \models s[u] \approx t$.

Cas 2.2 : Supposons que σ est réductible dans $Gr(E_\infty)$ et $\sigma(c)$ est irréductible dans $Gr(E_\infty)$.

Comme σ est réductible dans $Gr(E_\infty)$ et $\sigma(c)$ est irréductible dans $Gr(E_\infty)$, il existe une inférence de paire critique dans le cas clos qui est une inférence de paire critique à une position de variable pas dans la contrainte dans le cas non clos. Cette inférence est :

$$\frac{s[u] \approx t \qquad u \approx v}{s[v] \approx t}$$

Il existe un arc I de $s[u] \approx t$ à $u \approx v$. Il existe des arcs I du sommet étiqueté par $s[u] \approx t$ au sommet étiqueté par $u \approx v$ et également au sommet étiqueté par $s[v] \approx t$. $Red(s[u],s[u] \approx t) \wedge \neg Red(u,u \approx v) \wedge \neg EqToRedConst(\sigma(c))$ est satisfaisable dans $Gr(E_\infty)$. Ainsi en utilisant $\succ_g$, $s[v] \approx t \prec_g s[u] \approx t$ et $u \approx v \prec_g s[u] \approx t$. Comme σ est réductible dans $Gr(E_\infty)$ et $\sigma(c)$ est irréductible dans $Gr(E_\infty)$, $s[u] \approx t$ et $s[v] \approx t$ sont des instances de la même égalité $s' \approx t' [\![c]\!]$ et sont dans $Gr(E_\infty)$.

Comme $s[v] \approx t \in Gr(E_\infty)$ et $s[v] \approx t \prec_g s[u] \approx t$ à cause des arcs I et en utilisant l'hypothèse de récurrence, nous pouvons déduire que $Irred(Gr(E_\infty)) \models s[v] \approx t$. De plus, $Irred(Gr(E_\infty)) \models u \approx v$ (car $s[u] \approx t \succ_g u \approx v$ et à cause de l'hypothèse de récurrence). Ainsi, $Irred(Gr(E_\infty)) \models s[u] \approx t$.

Cas 2.3 : Supposons que la contrainte $\sigma(c)$ est réductible dans $Gr(E_\infty)$.

Dans ce cas, il y a une inférence de paire critique dans le cas clos qui correspond à une inférence de paire critique dans la contrainte dans le cas non clos. L'égalité $s' \approx t' [\![c]\!]$ est E-cycle bloquée.
$s \approx t$ n'est pas dans $Gr(E_\infty)$, car il est g-redondant dans $Gr(E_i)$ pour un i et donc dans $Gr(E_\infty)$. Ainsi, $s \approx t$ est impliqué par des égalités plus petites de $Gr(E_\infty)$. Il existe $e_1 \in Gr(E_\infty), \ldots, e_n \in Gr(E_\infty)$, telles que $\forall i, e_i \prec_g s \approx t$. De plus, $e_1, \ldots, e_n \models s \approx t$.
Comme $e_i \in Gr(E_\infty)$ et $e_i \prec_g s \approx t$, nous pouvons appliquer l'hypothèse de récurrence aux e_i. Ainsi, $e_i \in Gr(E_\infty)$ implique que $Irred(Gr(E_\infty)) \models e_i$. Or, $e_1, \ldots, e_n \models s \approx t$ donc par transitivité, $Irred(Gr(E_\infty)) \models s \approx t$.

□

Le théorème suivant prouve la complétude de la complétion basique avec simplification E-cycle. La preuve est basée sur la correspondance entre la construction procédurale du graphe de dépendance non clos en utilisant *BCES* et notre construction abstraite du graphe de dépendance clos présenté dans cette section.

Théorème 17 *La complétion basique avec simplification E-cycle est complète.*

Preuve : Soit $(E_0 = E, G_0 = G_{init}) \rightarrow (E_1, G_1) \rightarrow \ldots (E_n, G_n) \rightarrow \ldots$ une dérivation de transitions de graphes obtenue en utilisant les règles d'inférence de *BCES*, qui décrit la complétion basique avec simplification E-cycle et $(EG_0 = Gr(E), GG_0 = GG_{init}) \rightarrow (EG_1, GG_1) \rightarrow \ldots (EG_n, GG_n) \rightarrow \ldots$ sa dérivation de transitions de graphes clos associée modulo $Gr(E_\infty)$.
Pour tout i, G_i est un graphe de dépendance graphe sans E-cycle, car l'ensemble des règles d'inférence de *BCES* évite la création de E-cycle. Ainsi, d'après le lemme 6, pour tout i, le graphe de dépendance clos GG_i ne contient pas de cycle et pas de chemin infini. Nous pouvons alors appliquer le théorème 16 pour conclure que la complétion basique avec simplification E-cycle est complète. □

4.5 Comparaison avec la simplification basique

Dans cette section, nous comparons les stratégies de simplification E-cycle et de simplification basique. Nous les comparons d'un point de vue général et par rapport au nombre de simplifications réalisées.

4.5.1 La simplification E-cycle n'est pas locale

Nous comparons ici la règle *Simplification Basique* par rapport à la règle *Simplification E-cycle*.

Une des principales caractéristiques de la règle *Simplification E-cycle* explicitée ci-dessous est que le test d'application de cette règle n'est pas *local* comme c'est le cas pour la règle *Simplification Basique* (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995).

- Dans le cas de l'application de la règle *Simplification Basique*, le test à réaliser ne prend en compte que les égalités de l'inférence. C'est dans ce sens que l'on dit que le test d'application de la règle *Simplification Basique* est *local* et que la simplification basique est *locale*. Il s'agit en effet de savoir s'il s'agit d'une simplification standard et si l'égalité « droite » est substitution-réduite relativement à l'égalité « gauche ».

- Dans le cas de l'application de la règle *Simplification E-cycle*, le test ne prend pas seulement en compte les égalités de l'inférence à réaliser. La simplification E-cycle n'est donc pas *locale.* En effet, le test est basé sur le graphe de dépendance et prend en compte toutes les égalités déduites jusqu'à ce point de la déduction.

4.5.2 La simplification basique est contenue dans la simplification E-cycle

Dans cette section, nous prouvons que la simplification basique est contenue dans la simplification E-cycle. Ainsi, nous montrons que, pour toute inférence de simplification, si une égalité peut être basique-simplifiée, alors aucun E-cycle n'est créé dans le graphe de dépendance et ainsi, l'égalité peut également être E-cycle simplifiée. La preuve de cette propriété est basée sur une suite de lemmes qui montrent que certains motifs d'arcs ne peuvent pas être ajoutés au graphe de dépendance si l'on considère la stratégie de simplification basique.

Nous montrons également que la simplification E-cycle n'est pas contenue dans la simplification basique en fournissant un exemple où une égalité peut être E-cycle-simplifiée mais pas basique-simplifiée.

Lemme 10 *Soit* $(E_0 = E, G_0 = G_{init}) \to (E_1, G_1) \to \cdots \to (E_n, G_n) \cdots$ *une dérivation de transitions de graphes telle que* $E_0 \to E_1 \to \cdots \to E_n$ *soit une dérivation de* BCBS. *Alors, pour tout i,* G_i *ne contient pas de E-chemin avec un arc I suivi par des arcs S et ensuite par un arc C.*

Preuve : Soit G_i un graphe de dépendance obtenu par une dérivation de transitions de graphes en utilisant les règles de *BCBS* et contenant un E-chemin $\mathcal{P}$ avec un arc I e_{inf} suivi par des arcs S et ensuite par un arc C e_{const}.

e_{inf} va du sommet v_1 au sommet v_2. Le E-chemin d'arcs S va du sommet v_2 au sommet v_n $(n > 2)$ et e_{const} va du sommet v_n au sommet v_{n+1}. Nous pouvons remarquer que si $n = 2$, nous obtenons le théorème 12.

Pour $i \geq 1$, soient e_i l'étiquette du sommet v_i et $Constraint(e_i) = c_i$.

Comme nous faisons de la simplification basique, l'égalité e_n est substitution-réduite relativement à l'égalité e_2. Ainsi, si $EqToRedConst(c_n)$ n'est pas contradictoire syntaxiquement, alors $EqToRedConst(c_2)$ ne l'est pas non plus. La contrainte de réductibilité associée à l'arc e_{inf} est $\neg EqToRedConst(c_2)$, la contrainte de réductibilité associée aux arcs S est $\top$ et la contrainte de réductibilité associée à l'arc e_{const} est $EqToRedConst(c_n)$. Ainsi, par l'inconsistance des contraintes, e_{inf} et e_{const} ne peuvent pas être contenus dans un E-chemin. Ainsi, il y a une contradiction.

Il est donc impossible d'avoir un E-chemin composé d'un arc I e_{inf} suivi d'arcs S et ensuite par un arc C e_{const} dans G_i. □

Lemme 11 *Soit* $(E_0 = E, G_0 = G_{init}) \to (E_1, G_1) \to \cdots \to (E_n, G_n) \cdots$ *une dérivation de transitions de graphes telle que* $E_0 \to E_1 \to \cdots \to E_n$ *soit une dérivation de* BCBS. *Alors, s'il existe i tel que* G_i *contienne un E-cycle, alors ce E-cycle ne contient pas d'arc I.*

Preuve : Soit G_i un graphe de dépendance obtenu par une dérivation de transitions de graphes en utilisant *BCBS* et contenant un E-cycle $\mathcal{C}$ avec un arc I e_{inf}.

$\mathcal{C}$ contient un E-chemin composé d'un arc I e_{inf} suivi par des arcs S et ensuite par un arc C e_{const}, car $\mathcal{C}$ contient au moins un arc C (voir le théorème 13), au moins un arc S (voir le théorème 14) et car un arc I ne peut pas être suivi par un arc C (voir le théorème 12). Cependant, ceci est impossible (voir le lemme 10).

Il y a donc une contradiction et nous pouvons conclure que G_i ne contient pas d'arc I. □

Théorème 18 *Soit* $(E_0 = E, G_0 = G_{init}) \to (E_1, G_1) \to \cdots \to (E_n, G_n) \cdots$ *une dérivation de transitions de graphes telle que* $E_0 \to E_1 \to \cdots \to E_n$ *soit une dérivation de* BCBS. *Alors, il n'y a pas de i tel que* G_i *contienne un E-cycle.*

Preuve : Soit G_i un graphe de dépendance obtenu par une dérivation de transitions de graphes utilisant les règles de *BCBS* et contenant un E-cycle $\mathcal{C}$.
Par définition, $\mathcal{C}$ contient au moins un arc C et un arc S (voir le théorème 13 et le théorème 14 respectivement). D'après le lemme 11, $\mathcal{C}$ ne contient pas d'arc I. Ainsi, $\mathcal{C}$ contient seulement des arcs C et S. Ainsi, $\mathcal{C}$ contient un chemin $\mathcal{P}$ composé d'un arc C suivi par un nombre d'arcs S supérieur ou égal à 0 suivi par un arc C.
Soient $v_0 \cdots, v_n$ les sommets du chemin $\mathcal{P}$, et soit e_i l'étiquette du sommet v_i. L'arc du sommet v_1 au sommet v_2 est un arc C. Ainsi, e_1 est une égalité contrainte et e_2 est une égalité non contrainte. Comme il y a un arc S du sommet v_{i-1} au sommet v_i pour $i \in \{3, \cdots, n-1\}$, nous savons que l'égalité e_i est substitution-réduite relativement à l'égalité e_{i-1} pour $i \in \{3, \cdots, n-1\}$, car nous faisons de la simplification basique. Comme e_2 est une égalité non contrainte, ceci implique que chaque égalité e_i est non contrainte pour $i \in \{3, \cdots, n-1\}$. En particulier, e_{n-1} est une égalité non contrainte. Comme il y a un arc C du sommet v_{n-1} au sommet v_n, l'égalité e_{n-1} est contrainte. La contradiction prouve que $\mathcal{C}$ n'existe pas. □

Nous déduisons du théorème 18 le théorème 19 dont la preuve est triviale.

Théorème 19 *Si* $E_0 \to E_1 \to \cdots \to E_n$ *est une dérivation de* BCBS, *alors c'est aussi une dérivation de* BCES.

Nous avons donc prouvé que toute dérivation de *BCBS* est une dérivation de *BCES*, donc que la complétion basique avec simplification basique est contenue dans la complétion basique avec simplification E-cycle.

Inversement, nous donnons un exemple qui montre que toute dérivation de *BCES* n'est pas une dérivation de *BCBS*, donc que la complétion basique avec simplification E-cycle n'est pas contenue dans la complétion basique avec simplification basique.

Exemple 37 *Soit* $E = \{g(x) \approx f(x)\ (1), g(a) \approx b\ (2), h(f(a)) \approx b\ (3)\}$. *Nous utilisons l'ordre lexicographique sur les chemins basé sur la précédence* $h \succ_{prec} g \succ_{prec} f \succ_{prec} a \succ_{prec} b$. *Suivons le plan de recherche suivant en utilisant* BCES.

1. *Considérons l'inférence de paire critique basique suivante.*
$$\frac{g(x) \approx f(x)\ (1) \qquad g(a) \approx b\ (2)}{f(x) \approx b [\![x =^? a]\!]\ (4)}$$
Nous ajoutons des arcs C de l'égalité (4) aux égalités (1) et (2). La contrainte de réductibilité associée à ces arcs est $Red(a)$.
Nous ajoutons un arc I de l'égalité (1) à l'égalité (4). La contrainte de réductibilité associée à cet arc est $\neg Red(a)$.
2. *Considérons l'inférence suivante.*
$$\frac{h(f(a)) \approx b\ (3) \qquad f(x) \approx b [\![x =^? a]\!]\ (4)}{h(b) \approx b\ (5)}$$
Cette inférence correspond à une inférence de simplification E-cycle. L'égalité $h(f(a)) \approx b$ *(3) est E-cycle simplifiée par l'égalité* $f(x) \approx b [\![x =^? a]\!]$ *(4). En effet, aucun E-cycle n'est créé, quand on ajoute des arcs S de l'égalité (3) aux égalités (4) et (5).*
Toutefois, cette inférence ne correspond pas à une inférence de simplification basique. L'égalité $h(f(a)) \approx b$ *ne peut pas être basique-simplifiée par l'égalité* $f(x) \approx b [\![x =^? a]\!]$, *car l'égalité* $f(x) \approx b [\![x =^? a]\!]$ *n'est pas substitution-réduite relativement à l'égalité* $h(f(a)) \approx b$.

Le graphe de dépendance pour les deux inférences réalisées ici est visible sur la figure 4.11. L'ensemble saturé pour BCES E_∞ *est* $\{g(x) \approx f(x)\ (1), g(a) \approx b\ (2), f(x) \approx b [\![x =^? a]\!]\ (4), h(b) \approx b\ (5)\}$

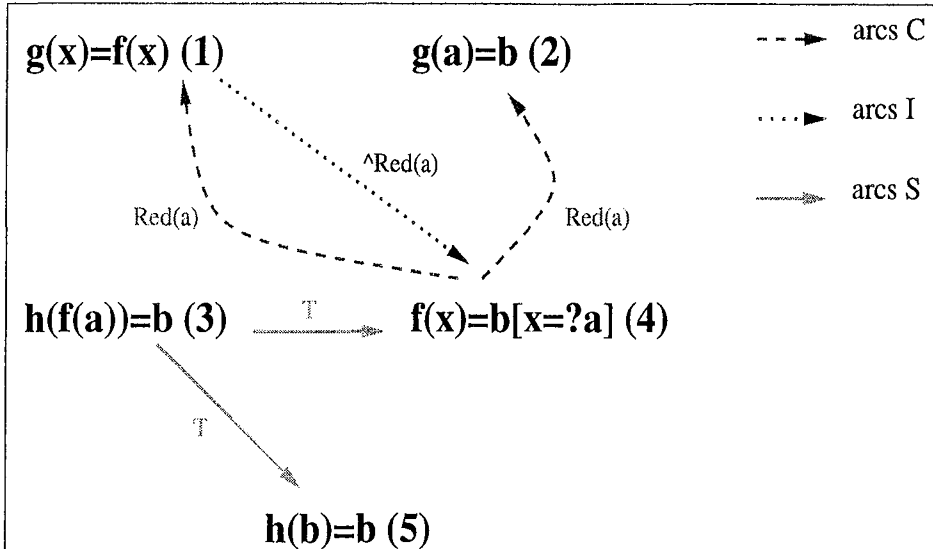


FIG. 4.11 – *Graphe de dépendance de la complétion basique avec simplification E-cycle de* $E = \{g(x) \approx f(x)\ (1), g(a) \approx b\ (2), h(f(a)) \approx b\ (3)\}$

La simplification basique est donc contenue strictement dans la simplification E-cycle. Ainsi, la simplification E-cycle permet donc de faire plus de simplifications que la simplification basique.

4.6 Expérience pratique en ELAN : l'implantation ECC

La complétion basique avec simplification E-cycle est implantée dans le système *ECC* (E-cycle complétion) écrit en ELAN (Borovanský, Cirstea, Dubois, Kirchner, Kirchner, Moreau, Ringeissen et Vittek, 1998).

ECC peut tout aussi bien réaliser :
- de la complétion standard,
- de la complétion basique avec simplification standard,
- de la complétion basique avec simplification basique sans rétraction,
- de la complétion basique avec simplification basique avec rétraction, et
- de la complétion basique avec simplification E-cycle.

Nous donnons dans le tableau qui suit page 191 la liste des problèmes utilisés pour tester *ECC*. Ce tableau rassemble également nos résultats expérimentaux. Les exemples choisis sont des exemples communément utilisés pour tester les systèmes basés sur la complétion. Nous n'avons pas testé *ECC* sur des exemples de taille plus importante. Notre limitation provient du fait que l'interpréteur ELAN, utilisé pour tester *ECC*, possède des problèmes de mémoire. Ces problèmes de mémoire sont liés au stockage du graphe de dépendance pendant le processus de complétion et au test de satisfaisabilité des contraintes de réductibilité qui est un problème NP-complet (Hermann, 1998). Il existe maintenant un compilateur ELAN très efficace, mais il était en développement lors de l'implantation de *ECC*. Nous envisageons, dans l'avenir, de tester *ECC* avec le compilateur, les résultats seront de ce fait plus probants.

Les tests actuels réalisés avec l'interpréteur ELAN nous permettent déjà de prouver pratiquement le pouvoir de la simplification E-cycle, comme nous le montrons.

Nous obtenons des résultats pour la complétion basique avec simplification E-cycle sur les exemples **curry**, **groupl**, **knz86**, **rubio** et **zeh89**.

Nous comparons la complétion basique avec simplification E-cycle à la complétion basique avec d'autres stratégies de simplification sur ces exemples dans *ECC* et nous nous intéressons principalement au nombre d'égalités déduites. De ce fait, nous comptons le nombre d'égalités conclusion des inférences qui ne sont pas des égalités triviales.

Les résultats significatifs obtenus sont décrits ci-dessous.

- Nous obtenons exactement le même nombre d'égalités déduites pour la complétion basique avec simplification E-cycle et pour la complétion basique avec simplification basique sans rétraction sur les exemples testés à part pour **zeh89**.

  Ce dernier exemple termine pour la complétion basique avec simplification E-cycle. Le nombre d'égalités déduites est de 97. Mais, il ne termine pas pour la complétion basique avec simplification basique sans rétraction. Dans la complétion basique avec simplification E-cycle, nous n'appliquons pas la règle *Rétraction E-cycle*. En effet, comme nous l'avons montré dans la section 4.3.1.4, la rétraction n'a d'intérêt en complétion basique avec simplification E-cycle que si les égalités concernées deviennent non contraintes. De plus, la rétraction enlève l'avantage principal à la complétion basique qui est l'utilisation des contraintes. Nous pouvons donc conclure que notre stratégie est plus puissante.

- Pour la complétion basique avec simplification E-cycle et pour la complétion basique avec simplification basique avec rétraction, nous obtenons exactement le même nombre d'égalités déduites dans les deux cas sur les exemples cités.

  Nous pouvons donc conclure que notre méthode est plus puissante car elle n'utilise pas la rétraction et permet d'obtenir exactement le même nombre d'égalités déduites que la complétion basique avec simplification basique avec rétraction.

L'adresse http://www.loria.fr/~scharff/ECC regroupe des détails sur le système *ECC* et sur les résultats expérimentaux obtenus.

Problèmes	Axiomes	BCSS	BCBS (sans rétraction)	BCBS (avec rétraction	BCES
comm	$h(h(a,b),b) \approx d$ $f(o,x) \approx x$ $f(x,o) \approx x$ $f(x,f(x,x)) \approx e$ $f(i(x),f(x,y)) \approx y$ $h(x,y) \approx f(x,f(y,f(i(x),i(y))))$ $h > i > f > e > o > d > a > b$	66 (Preuve pratique de l'incomplétude de *BCSS*)	Ne termine pas.	118	Pas de résultat [a]
curry	$imp(x,a) \approx a$ $imp(a,x) \approx x$ $m(a,x) \approx x$ $m(x,a) \approx x$ $m(m(x,y),z) \approx m(x,m(y,z))$ $imp(x,imp(y,z)) \approx imp(m(x,y),z)$ $imp(x,m(y,z)) \approx m(imp(x,y),imp(x,z))$ $imp > m > a$	70	70	70	70
exa85	$f(x,o) \approx x$ $f(x,i(x)) \approx o$ $h(f(x,y)) \approx f(h(x),h(y))$ $f(f(x,y),z) \approx f(x,f(y,z))$ $h > i > f > o$	361	Ne termine pas.	466	Pas de résultat.
group1 - exa79	$f(o,x) \approx x$ $f(x,o) \approx x$ $f(i(x),f(x,y)) \approx y$ $i > f > o$	36	40	36	40
group2 - exa80	$f(o,x) \approx x$ $f(i(x),x) \approx o$ $f(f(x,y),z) \approx f(x,f(y,z))$ $i > f > o$	109	Ne termine pas.	123	Pas de résultat.

[a] *Pas de résultat* signifie que l'interpréteur n'avait pas assez de mémoire pour résoudre le problème.

Problèmes	Axiomes	BCSS	BCBS (sans rétraction)	BCBS (avec rétraction	BCES
group3	$f(o,x) \approx x$ $f(x,o) \approx x$ $f(i(x),x) \approx o$ $f(x,i(x)) \approx o$ $f(f(x,y),z) \approx f(x,f(y,z))$ $i > f > o$	93	Ne termine pas.	93	Pas de résultat.
knz86	$h(o) \approx s(o)$ $f(o,y) \approx y$ $g(o,y) \approx y$ $s(s(x)) \approx x$ $f(s(x),y) \approx s(f(x,y))$ $g(s(x),y) \approx f(g(x,y),o)$ $f(f(g(x,y),o),o) \approx g(x,y)$ $g > f > o > h > s$ Problème proposé dans (Kapur, Narendran et Zhang, 1986).	27	27	27	27
p2 - kb_bench	$f(e1,x) \approx x$ $f(e2,x) \approx x$ $f(x,i1(x)) \approx e1$ $f(x,i2(x)) \approx e2$ $f(f(x,y),z) \approx f(x,f(y,z))$ $i2 > i1 > f > e2 > e1$	497	Pas de résultat.	525	Pas de résultat.
p63 - x2_quant	$f(b,c) \approx f(d,c)$ $f(o,x) \approx x$ $f(i(x),x) \approx o$ $f(f(x,y),z) \approx f(x,f(y,z))$ $i > f > o > b > c > d$ Les trois derniers axiomes permettent de prouver que$f(b,c) \approx f(d,c)$, c'est-à-dire que$f(x,y) \approx f(z,y) \Rightarrow x \approx y$. Cet exemple est proposé dans (Pelletier, 1986).	173	Ne termine pas.	263	Pas de résultat.

Problèmes	Axiomes	BCSS	BCBS (sans rétraction)	BCBS (avec rétraction	BCES
p64	$f(o,x) \approx x$ $f(i(x),x) \approx o$ $f(f(x,y),z) \approx f(x,f(y,z))$ $f(b,c) \approx o$ $f(c,b) \approx d$ $i > f > o > b > c > d$ Les trois premiers axiomes permettent de prouver que $f(y,x) \approx o \Rightarrow f(x,y) \approx o$, c'est-à-dire ici, que $o \approx d$. Cet exemple est proposé dans (Pelletier, 1986).	339	Ne termine pas.	419	Pas de résultat.
revfalse	$rev(null) \approx null$ $app(null,x) \approx null$ $app(cons(x,y),z) \approx cons(x,app(y,z))$ $rev(cons(x,y)) \approx app(rev(y),cons(x,null))$ $rev > app > cons > null$	5	5	5	5
rubio	$a \approx b$ $f(g(a)) \approx b$ $f(g(x)) \approx g(x)$ $f > g > a > b$ Problème proposé dans (Nieuwenhuis et Rubio, 1992a).	8	8	8	8
zeh89	$exp(o) \approx e$ $p(x,o) \approx x$ $m(x,e) \approx x$ $p(p(x,y),z) \approx p(x,p(y,z))$ $m(m(x,y),z) \approx m(x,m(y,z))$ $exp(p(x,y)) \approx m(exp(x),exp(y))$ $exp > m > p > e > o$	97	Ne termine pas.	97	97

4.7 Stratégies de simplification incomplètes

Dans cette section, nous définissons trois stratégies de simplification S_1, S_2 et S_3. Nous donnons des exemples montrant l'incomplétude de ces stratégies de simplification en combinaison avec la complétion basique. Nous développons ces même exemples en utilisant le système d'inférence *BCES*. La complétion basique avec simplification E-cycle étant complète, nous obtenons à chaque fois un système d'égalités convergent.

Notre intention est de trouver la « ligne » qui sépare les stratégies de simplification complètes des stratégies de simplification incomplètes. Notre intuition est que la stratégie de simplification E-cycle est très proche de la « ligne » qui sépare les stratégies de simplification complètes

des stratégies de simplification incomplètes. Nous n'avons toutefois actuellement aucune preuve concernant cette remarque.

4.7.1 Stratégies S_1, S_2 et S_3

S_1 La stratégie S_1 est appelée « *simplification arrière* » (*Backward Simplification*) dans la littérature.

Toute simplification d'égalité e_1 par une égalité e_2 est une « *simplification arrière* », si e_1 peut être simplifiée par une simplification standard par e_2 et si e_2 est générée après e_1.

S_2 La stratégie S_2 est appelée « *simplification avant* » (*Forward Simplification*) dans la littérature.

Toute simplification d'égalité e_1 par une égalité e_2 est une « *simplification avant* », si e_1 peut être simplifiée par une simplification standard par e_2 et si e_2 existait quand e_1 a été générée.

S_3 La stratégie S_3 pourrait s'appeler la *stratégie de simplification initiale.*

Toute simplification d'égalité e_1 par une égalité e_2 est une « *simplification initiale* », si e_1 peut être simplifiée par une simplification standard par e_2 et si toutes les égalités de l'ensemble initial qui ont permis de générer le simplificateur e_2 existent dans l'espace de recherche.

4.7.2 Incomplétude de la stratégie S_1

L'exemple de Nieuwenhuis et Rubio (Nieuwenhuis et Rubio, 1992a) développé dans la section 4.3.4 montre que la complétion basique avec la stratégie S_1 est incomplète. En effet, dans l'exemple 34, la seconde inférence est une « simplification arrière ». L'égalité (3) est « arrière-simplifiée » par l'égalité (4). La suppression de l'égalité (3) de l'espace de recherche ne permet pas d'obtenir un système d'égalités saturé convergent, car on ne peut pas montrer que l'égalité $g(b) \approx b$ est vraie par une preuve par réécriture.

4.7.3 Incomplétude de la stratégie S_2

Nous donnons ici un exemple qui montre que la complétion basique avec « simplification avant » est incomplète. Nous répondons ainsi au problème ouvert numéro un de R. Nieuwenhuis (Nieuwenhuis, 1999). Nous développons également cet exemple avec la complétion basique avec simplification E-cycle.

Exemple 38 *Soit* $E = \{h(f(f(a))) \approx b \ (1),\ f(x) \approx x \ (2)\}$.

Nous utilisons l'ordre lexicographique sur les chemins basé sur la précédence suivante : $h \succ_{prec} f \succ_{prec} a \succ_{prec} b$.

Supposons que le plan de recherche suivant est utilisé pour faire de la complétion basique.

1. $$\frac{h(f(f(a))) \approx b \ (1) \qquad f(x) \approx x \ (2)}{h(x) \approx b[\![x =^? f(a)]\!] \ (3)}$$

Cette inférence correspond à une inférence de paire critique basique.

Nous ajoutons des arcs C de l'égalité (3) aux égalités initiales (1) et (2). La contrainte de réductibilité associée à ces arcs est $Red(f(a))$.

Nous ajoutons un arc I de l'égalité (1) à l'égalité (3). La contrainte de réductibilité associée à cet arc est $\neg Red(f(a))$.

Il y a une simplification standard de l'égalité $h(f(f(a))) \approx b$ (1) *par l'égalité* $f(x) \approx x$ (2), *mais il n'y a pas de simplification E-cycle. En effet, un E-cycle contenant les sommets contenant les égalités (1) et (3) est détecté.*

2. $$\frac{h(f(f(a))) \approx b \ (1) \qquad f(x) \approx x \ (2)}{h(f(x)) \approx b \, [\![x =^? a]\!] \ (4)}$$
Cette inférence correspond à une inférence de paire critique basique.
Nous ajoutons des arcs C de l'égalité (4) aux égalités initiales (1) et (2). La contrainte de réductibilité associée à ces arcs est $Red(a)$.
Nous ajoutons un arc I de l'égalité (1) à l'égalité (4). La contrainte de réductibilité associée à cet arc est $\neg Red(a)$.
Il y a une simplification standard de l'égalité $h(f(f(a))) \approx b$ (1) *par l'égalité* $f(x) \approx x$ (2), *mais il n'y a pas de simplification E-cycle. En effet, un E-cycle contenant les sommets contenant les égalités (1) et (3) est détecté.*

3. $$\frac{h(f(x)) \approx b \, [\![x =^? a]\!] \ (4) \qquad h(x) \approx b \, [\![x =^? f(a)]\!] \ (3)}{b \approx b}$$
Nous pouvons « simplifier avant » l'égalité (4) par l'égalité (3). Cette simplification détruit la complétude. En effet, si nous supprimons l'égalité (4) de l'espace de recherche, il n'y a pas d'autre inférence à réaliser et nous obtenons le système d'égalités saturé suivant : $E'_\infty = \{h(f(f(a))) \approx b \ (1),\ f(x) \approx x \ (2),\ h(x) \approx b \, [\![x =^? f(a)]\!] \ (3)\}$. *La complétion avec « simplification avant » n'est pas complète car le système* E'_∞ *n'est pas convergent. Nous ne pouvons pas prouver que* $h(a) \approx b$ *est vraie par une preuve par réécriture, alors que cette égalité est vraie dans* E.
Cependant, le développement de cet exemple avec la complétion basique avec simplification E-cycle fournit un ensemble d'égalités saturé convergent.
L'égalité triviale $b \approx b$ *n'est pas ajoutée au graphe de dépendance. Elle ne sera jamais impliquée dans un E-cycle.*
Il y a simplification standard de l'égalité (3) par l'égalité (4), mais il n'y a pas de simplification E-cycle. En effet, un E-cycle contenant les sommets contenant les égalités (1), (3) et (4) est détecté. Cette inférence correspond donc à une inférence de paire critique basique.

4. $$\frac{h(f(x)) \approx b \, [\![x =^? a]\!] \ (4) \qquad f(x) \approx x \ (2)}{h(x) \approx b \, [\![x =^? a]\!] (5)}$$
Cette inférence correspond donc à une inférence de simplification E-cycle.
Nous ajoutons des arcs C de l'égalité (5) aux égalité initiales (1) et (2). La contrainte de réductibilité associée à ces arcs est $Red(a)$.
Il y a une simplification standard et une simplification E-cycle de l'égalité (4) par l'égalité (2). Nous ajoutons des arcs S de l'égalité (4) aux égalités (2) et (5). Aucun E-cycle n'est créé.

Le graphe de dépendance pour les inférences précédentes de complétion basique avec simplification E-cycle est visible sur la figure 4.12. L'ensemble d'égalités saturé obtenu est $E_\infty = \{h(f(a)) \approx b \ (1),\ f(x) \approx x \ (2),\ h(x) \approx b \, [\![x =^? f(a)]\!] \ (3),\ h(x) \approx b \, [\![x =^? a]\!] \ (5)\}$.

4.7.4 Incomplétude de la stratégie S_3

Nous pensions que la stratégie de simplification S_3 était complète. En effet, si les égalités initiales dont dépend le simplificateur sont préservées, il est toujours possible de générer une version réduite de l'égalité simplificatrice.

L'exemple 38 nous montre toutefois que la stratégie initiale est incomplète pour la complétion basique. Dans la troisième inférence, la simplification de l'égalité (4) par l'égalité (3) est une

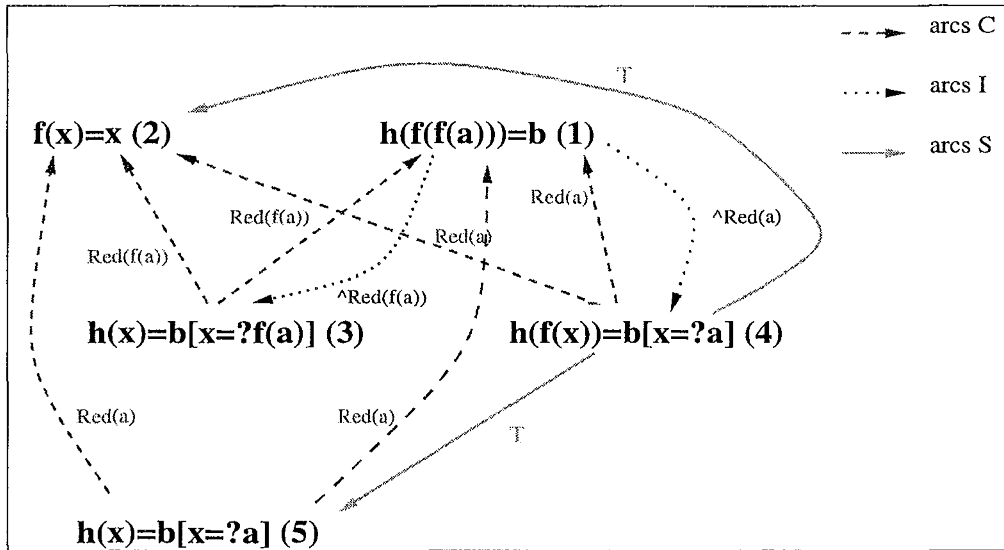


FIG. 4.12 – *Graphe de dépendance pour la complétion basique avec simplification E-cycle de* $E = \{h(f(f(a))) \approx b\ (1),\ f(x) \approx x\ (2)\}$

simplification initiale. Or, comme pour la « simplification avant », la simplification de l'égalité (4) implique l'incomplétude de la stratégie.

4.8 Conclusion

Nous avons présenté dans ce chapitre une nouvelle stratégie de simplification complète en combinaison avec la complétion basique. Le système d'inférence *BCES* a été défini. Nous avons atteint le but que nous nous étions fixé, à savoir proposer une stratégie de simplification complète en combinaison avec la complétion basique faisant plus de simplifications que la simplification basique. La simplification basique était jusqu'à ce jour la seule stratégie de simplification complète en combinaison avec la complétion basique.

La simplification E-cycle est liée à la construction d'un graphe de dépendance pendant le processus de complétion. Le graphe de dépendance détermine les ancêtres d'une égalité. Il mémorise les dépendances entre les égalités. Il permet de contrôler la complétude de la complétion de telle sorte que si une simplification est réalisée, la complétude est préservée. L'idée derrière la simplification E-cycle est qu'une égalité ne peut pas simplifier un de ses ancêtres obtenu dans la processus de complétion. Toute simplification d'un ancêtre est schématisée par un E-cycle dans le graphe de dépendance, un cycle particulier du graphe de dépendance, sachant que la création de E-cycle dans le graphe de dépendance est interdite.

La simplification E-cycle est une stratégie élégante, facile à comprendre et dont l'intérêt et le pouvoir ont été prouvés théoriquement et pratiquement dans l'implantation *ECC*.

Nous pensons que notre approche est plus facile à comprendre que la simplification basique, car elle est basée sur l'utilisation d'un graphe. La définition de la condition de réductibilité d'une égalité relativement à une autre égalité (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995) utilisée pour la simplification basique est en effet une notion difficile à comprendre. De plus, le nom de condition de réductibilité-relativement-à de la littérature ne capture pas les éléments essentiels de cette notion. Nous avons donc renommé dans le chapitre 2 cette notion

en substitution-réductibilité-relativement-à-modulo (un filtre) pour capturer les deux éléments importants de cette notion : la vision de la contrainte sous forme de substitution et son entière dépendance des substitutions solutions des contraintes des prémisses, et l'existence du filtre à travers l'élément modulo.

Nous avons prouvé la complétude de la simplification E-cycle en utilisant la technique de preuve basée sur la construction d'un modèle d'égalités de Herbrand pour l'ensemble des égalités saturé obtenu lors de la complétion basique, initialisée par L. Bachmair et H. Ganzinger (Bachmair et Ganzinger, 1994). Nous avons utilisé une plate-forme abstraite utilisant des graphes de dépendance clos pour aboutir à cette preuve. Nous pensons que cette plate-forme pourrait être utile pour analyser la complétude de différentes stratégies de simplification en combinaison avec la complétion basique et pour déterminer la « ligne » qui sépare les stratégies de simplification complètes des stratégies de simplification incomplètes. Dans cet objectif, nous avons montré sur des exemples l'incomplétude des stratégies de simplification S_1, S_2 et S_3 définies dans ce chapitre. Nous avons ainsi répondu par la négative au problème ouvert numéro un de R. Nieuwenhuis (Nieuwenhuis, 1999), qui conjecturait que la stratégie S_2 (*forward*) était complète. Les mêmes exemples développés avec la complétion basique avec simplification E-cycle nous ont fourni un ensemble convergent, car notre méthode est complète.

Nous pensons que la simplification E-cycle est près de la « ligne » qui sépare les stratégies de simplification complètes des stratégies de simplification incomplètes. Mais, nous n'avons pas étudié plus en détail cette proposition.

Nous avons comparé notre méthode avec la simplification basique et prouvé que la simplification basique est contenue strictement dans la simplification E-cycle. Ainsi, la simplification E-cycle permet donc de faire plus de simplifications que la simplification basique. L'intérêt et le pouvoir de la simplification E-cycle dans le cas de la complétion basique sont donc établis.

Nous avons implanté une procédure de complétion basique avec simplification E-cycle dans le système *ECC* écrit en ELAN et donné quelques résultats expérimentaux. *ECC* permet également de faire de la complétion standard, de la complétion basique avec simplification standard, de la complétion basique avec simplification basique avec et sans rétraction. Nous avons donc comparer les différentes stratégies de simplification implantées dans *ECC*. Les résultats nous ont permis de prouver le pouvoir pratique de la simplification E-cycle. En particulier, nous avons trouvé un exemple qui termine pour la complétion basique avec simplification E-cycle mais qui ne termine pas pour la complétion basique avec simplification basique. Il s'agit de l'exemple *zeh89*. Nous avons également insisté sur le fait que la rétraction n'a pas lieu d'être pour la simplification E-cycle mais que la rétraction est souvent implantée dans les systèmes de recherche de preuve et de déduction de la littérature pour augmenter le nombre de simplifications à réaliser. Concernant les points faibles de *ECC*, il faut souligner que *ECC* n'est qu'un prototype et n'a pas été testé sur des exemples de tailles importantes. Il a été testé principalement avec l'interpréteur ELAN, alors qu'il existe maintenant un compilateur très efficace. Pour valider la simplification E-cycle, un élément important serait d'utiliser cette stratégie dans des systèmes existants dont l'efficacité a déjà été démontrée.

Notre intention était d'étendre cette méthode de simplification à la complétion basique modulo une théorie équationnelle. Or, la simplification E-cycle nécessite de calculer les substitutions solutions des contraintes des prémisses d'une inférence. C'est également le cas de la simplification basique. Dans la complétion basique avec simplification E-cycle, les contraintes de réductibilité étiquetant les arcs sont calculées à partir des substitutions solutions des contraintes des prémisses. Ces solutions sont de plus nécessaires pour calculer le filtre existant entre l'égalité simplifiante et un sous terme de l'égalité à simplifier. La simplification ne se fait que s'il existe un filtre et si la simplification ne crée pas de E-cycle. Calculer les substitutions solutions des contraintes

d'unification dans la théorie vide est linéaire par rapport à la taille des termes (Paterson et Wegman, 1978; Huet, 1976). Cependant, dans le cas de théories équationnelle, le problème est plus complexe. Dans le cas de la théorie associative-commutative, le nombre d'unificateurs plus généraux d'un problème d'unification associative-commutative contenant un seul symbole associatif-commutatif est doublement exponentiel (Kapur et Narendran, 1992; Domenjoud, 1992). L'extension de notre méthode au cas équationnel apparaît donc de ce point de vue limitée. Nous avons donc amélioré la simplification basique dans le cas de la théorie vide mais l'extension de notre méthode paraissant difficile, nous avons choisi d'explorer une autre direction pour proposer des solutions aux problèmes liés à la simplification en complétion basique modulo une théorie équationnelle et particulièrement une méthode concrète et pratique.

Chapitre 5

Complétion basique modulo avec simplification

Sommaire

5.1 **Introduction** 199
5.2 **Contraintes de contraintes** 201
5.3 **Motivations** 202
5.4 **Le système d'inférence $BCICS_m$** 203
5.4.1 La règle d'inférence *Paire Critique Basique 2* 203
5.4.2 La règle d'inférence *Simplification* 204
5.4.3 La règle d'inférence *Paire Critique Basique dans la contrainte* 206
5.4.4 La règle d'inférence *Simplification dans la Contrainte* 211
5.4.5 Terminaison 213
5.5 **$BCICS_m$ est correct et complet** 213
5.5.1 Correction 213
5.5.2 Complétude 216
5.6 **Conclusion** 217

5.1 Introduction

Nous avons montré dans le chapitre 2 le cheminement qui a mené à développer la complétion avec contraintes modulo et en particulier la complétion basique modulo. La complétion contrainte modulo allie les avantages de la complétion modulo et de la complétion contrainte.

Nous résumons ici les principales étapes importantes.

Le raisonnement modulo a été motivé dans un premier temps par le fait que des axiomes tels que l'axiome de commutativité ne peuvent pas être orientés en règle de réécriture. Une autre raison est que le traitement spécifique de certains axiomes peut éviter la non-terminaison de la complétion. Travailler modulo permet également de séparer la déduction (les règles d'inférence de la complétion) qui est en général un processus indécidable de la partie calcul (unification, filtrage) qui est décidable dans certaines théories (Dowek et al., 1998). L'inconvénient de la complétion modulo est qu'elle déduit une égalité par unificateur plus général du problème d'unification considéré. Dans le cas AC par exemple, le nombre des unificateurs plus généraux d'un problème d'unification est doublement exponentiel par rapport à la taille des termes du problème (Domenjoud, 1992), le nombre d'égalités déduites sera donc également doublement exponentiel. Le cas de la complétion A est pire dans le sens où l'unification modulo A est infinitaire.

La complétion contrainte sauve le problème d'unification dans une contrainte au lieu d'appliquer le plus général unificateur. Elle contrôle l'expansion de l'espace de recherche dans le sens où elle interdit les inférences dans les contraintes. La complétion contrainte est très puissante dans le cas modulo, car elles ne calculent pas les unificateurs les plus généraux. Seule la satisfaisabilité des contraintes est testée. Pour chaque inférence, une seule égalité conclusion est déduite. La complétion basique est un cas particulier de la complétion contrainte où les contraintes sont vues comme des substitutions.

Les stratégies de contraction sont cruciales pour toutes les formes de complétion, en particulier pour les complétions contrainte et basique. Dans la littérature, les stratégies de contraction ont été étudiées principalement pour la complétion basique. Il existe seulement deux stratégies de simplification complètes en combinaison avec la complétion basique, la simplification basique (Nieuwenhuis et Rubio, 1992a; Bachmair et al., 1995) et la simplification E-cycle (Lynch et Scharff, 1998; Lynch et Scharff, 1999b) que nous avons décrite dans le chapitre 4. L'inconvénient de ces deux stratégies de simplification est qu'elles ont besoin de calculer les substitutions solutions des contraintes des égalités prémisses, ce qui implique la perte du bénéfice majeur de la complétion basique. Toutefois, l'avantage de la simplification E-cycle par rapport à la simplification basique est qu'elle permet plus de simplifications.

La simplification basique requiert l'existence d'un filtre et teste la condition de substitution-réductibilité-relativement-à sur les substitutions solutions des contraintes des prémisses de l'inférence considérée et en prenant en compte le filtre calculé.

Dans le cas de la stratégie de simplification E-cycle, un graphe de dépendance est construit pendant le processus de déduction. Les sommets de ce graphe sont des égalités. Lorsqu'une inférence est réalisée, un nouveau sommet et des arcs sont ajoutés. Les arcs schématisent les dépendances entre les égalités. Ils sont étiquetés par des contraintes de réductibilité calculées à partir des substitutions solutions des contraintes des égalités prémisses des inférences qui les ont créées. Un E-cycle est un cycle dans le graphe de dépendance sans conflit dans les contraintes de réductibilité étiquetant ses arcs. Une simplification E-cycle est réalisée s'il existe un filtre et si cette simplification ne crée pas de E-cycle dans le graphe de dépendance.

Dans la théorie vide, résoudre un problème d'unification ou de filtrage est linéaire par rapport à la taille des termes (Paterson et Wegman, 1978). Mais l'unification et le filtrage dans des théories équationnelles sont en général plus complexes. L'extension des stratégies de simplification basique et E-cycle à la complétion basique modulo ne peut donc pas être efficace en pratique.

La complétion contrainte et basique modulo AC en présence de la règle de contraction de simplification basique a été définie et prouvée complète dans (Vigneron, 1994a; Vigneron, 1994b). La stratégie de simplification basique est l'unique stratégie de contraction existante complète en combinaison avec les complétions contrainte et basique modulo AC. Les tests concernant l'application de la règle de contraction consistent dans un premier temps à calculer les unificateurs les plus généraux des problèmes d'unification AC des prémisses, sachant que le nombre d'unificateurs plus généraux des problèmes d'unification AC est doublement exponentiel par rapport à la taille des termes du problème (Kapur et Narendran, 1992; Domenjoud, 1992). Des problèmes de filtrage AC sont alors à considérer en tenant compte des substitutions solutions du problème d'unification AC précédemment résolu. Le test de substitution-réductibilité-relativement-à des prémisses de l'inférence de contraction est fonction des substitutions solutions du problème d'unification AC et des problèmes de filtrage AC. Les tests concernant l'application de la règle de contraction impliquent que la stratégie de simplification basique ne peut être efficace en pratique. DATAC (Vigneron, 1997; Vigneron, 1998) implante la complétion contrainte et basique modulo AC. Dans l'implantation de la règle de simplification basique, de nombreuses approximations ont été réalisées. Le test de substitution-réductibilité-relativement-à n'a pas été implanté, ce qui

cause une perte de la complétude du système d'inférence.

L'idée ici est de développer une nouvelle méthode de complétion basique modulo avec simplification qui est complète, qui autorise plus de simplifications et ne demande pas de résoudre les contraintes à l'inverse des travaux de (Vigneron, 1994a; Vigneron, 1994b). Nous voulons de plus qu'il n'y ait pas un écart insurmontable entre la théorie et la pratique. Nous présentons le système d'inférence $BCICS_m$ (*Complétion Basique modulo E avec inférences dans les contraintes et Simplification*) composé des règles d'inférence suivantes : *Paire Critique Basique 2*, *Simplification*, *Paire Critique Basique dans la contrainte* et *Simplification dans la contrainte.* Comme leurs noms l'indiquent, les règles *Paire Critique Basique 2* et *Paire Critique dans la Contrainte* calculent des paires critiques et les règles *Simplification* et *Simplification dans la Contrainte* sont des règles de contraction. Les règles *Paire Critique dans la Contrainte* et *Simplification dans la Contrainte* réalisent des inférences dans les contraintes. Notre approche prend donc le contre-pied de toutes les approches actuelles de la recherche de preuve et de la déduction, car les inférences sont permises dans les formules et dans les contraintes. Les règles d'inférence dans les contraintes sont utiles pour garantir la complétude du système d'inférence. Elles sont définies à l'aide de fonctions décrites par des règles de réécriture nommées sur lesquelles on applique une stratégie. Nous présentons les règles et les stratégies en ELAN. Les règles d'inférence dans la contrainte entraînent des inférences supplémentaires à réaliser. Toutefois, le système d'inférence $BCICS_m$ préserve l'avantage majeur de la complétion basique qui est de ne pas résoudre les contraintes mais seulement de tester leur satisfaisabilité. Nous avons introduit des contraintes de contraintes, les contraintes d'ordre contraintes et les contraintes de filtrage contraintes pour définir les règles d'inférence et pour ne pas avoir à résoudre les contraintes. L'idée d'utiliser des contraintes sur des contraintes semble naturelle, les contraintes étant des formules. Notre approche apparaît donc comme novatrice et originale dans le sens où elle permet de hiérarchiser les inférences. Elle permet également de manipuler des contraintes modulo des théories infinitaires comme l'associativité, ce qui n'était pas possible jusqu'à présent. Un exemple intéressant dans ce sens est donné dans la section 5.3. Nous pouvons souligner un autre avantage du système $BCICS_m$ dans le contrôle de l'expansion de l'espace de recherche. Une application de la règle *Paire Critique Basique dans la Contrainte* correspond à une application de la règle *Paire Critique* et à éventuellement une ou plusieurs applications de la règle *Simplification Standard* de la complétion standard modulo. Complémentairement, une application de la règle *Simplification dans la contrainte* correspond à une ou plusieurs applications de la règle *Simplification Standard* de la complétion standard modulo.

Les résultats de ce chapitre ont été présentés dans (Lynch et Scharff, 1999a).

5.2 Contraintes de contraintes

Nous présentons ici à part des contraintes sur des contraintes. Nous définissons des *contraintes d'ordre contraintes* et des *contraintes de filtrage contraintes* dans les définitions 48 et 49. Leur utilité est établie dans le chapitre 5.

Définition 48 $(\bigwedge_i c_i)$ $[\![c]\!]$ *est une* contrainte d'ordre *contrainte si c_i est une contrainte d'ordre et c est une contrainte équationnelle.* □

Une contrainte d'ordre contrainte $(\bigwedge_i c_i)$ $[\![c]\!]$ est *valide* si, pour toute substitution close $\sigma \in Sol(c)$, $(\bigwedge_i \sigma(c_i))$ est valide. Nous disons que $(\bigwedge_i c_i)$ $[\![c]\!]$ est *satisfaisable* si, il existe une substitution close $\sigma \in Sol(c)$ telle que $(\bigwedge_i \sigma(c_i))$ soit valide.

Définition 49 $s_2 \llbracket c_2 \rrbracket \lhd^?_T s_1 \llbracket c_1 \rrbracket$ *est une* contrainte de filtrage contrainte *si* s_1 *et* s_2 *sont des termes et* c_1 *et* c_2 *sont des contraintes équationnelles.* □

Nous écrivons $s_2 \llbracket c_2 \rrbracket \lhd_T s_1 \llbracket c_1 \rrbracket$, s'il existe une substitution $\sigma_2 \in Sol(c_2)$, telle que pour toute substitution $\sigma_1 \in Sol(c_1)$, il existe un filtre ρ tel que $\rho(\sigma_2(s_2)) \approx_T \sigma_1(s_1)$.

5.3 Motivations

Dans cette section, nous nous appuyons sur un exemple pour motiver la nécessité d'un nouveau système d'inférence pour réaliser la complétion basique modulo.

Soient la précédence $g \succ_p h \succ_p b \succ_p f \succ_p a$ et l'ensemble d'égalités $E = \{g(f(a,x),f(x,a)) \approx h(x),\ g(z,z) \approx h(z)\}$. f est un symbole associatif.

Nous considérons d'abord le système d'inférence $BCBS_m$ présenté dans le chapitre 2. Il existe une inférence de **paire critique basique modulo** A entre les deux égalités de l'ensemble E qui déduit l'égalité $h(z) \approx h(x) \llbracket z =^?_A f(a,x) \wedge z =^?_A f(x,a) \rrbracket$. Le système obtenu est saturé.

$$\frac{g(z,z) \approx h(z) \qquad g(f(a,x),f(x,a)) \approx h(x)}{h(z) \approx h(x) \llbracket z =^?_A f(a,x) \wedge z =^?_A f(x,a) \rrbracket}$$

En définissant une séquence de termes $t_0, \ldots, t_i, \ldots$ telle que $t_0 = a$ et $t_{i+1} = f(a,t_i)$ pour tout $i \geq 0$, l'ensemble complet des unificateurs les plus généraux de la contrainte $z =^?_A f(a,x) \wedge z =^?_A f(x,a)$ est $\bigcup_i \{x \mapsto t_i,\ z \mapsto t_{i+1} \mid i \geq 0\}$. Cet ensemble est infini. En effet, la contrainte $z =^?_A f(a,x) \wedge z =^?_A f(x,a)$ est équivalente à la contrainte $f(a,x) =^?_A f(x,a)$.

Si nous considérons une inférence de **paire critique modulo** A entre les égalités $g(z,z) \approx h(z)$ et $g(f(a,x),f(x,a)) \approx h(x)$, un nombre infini d'égalités est déduit et la complétion ne termine pas. L'usage des contraintes permet de ne générer qu'une seule égalité et de faire terminer la complétion. Nous avons ici illustré l'intérêt de la complétion basique modulo A par rapport à la complétion modulo A.

Nous allons maintenant montrer le problème de la complétion basique modulo que nous voulons résoudre dans ce chapitre.

Pour ce faire, nous ajoutons à E l'égalité $h(f(a,a)) \approx b$ donc $E = \{g(f(a,x),f(x,a)) \approx h(x),\ g(z,z) \approx h(z),\ h(f(a,a)) \approx b\}$.

En complétion basique modulo A, nous pouvons réaliser une inférence de **paire critique basique modulo** A entre $h(f(a,a)) \approx b$ comme égalité « gauche » et $h(z) \approx h(x) \llbracket z =^?_A f(a,x) \wedge z =^?_A f(x,a) \rrbracket$ comme égalité « droite ».

$$\frac{h(f(a,a)) \approx b \qquad h(z) \approx h(x) \llbracket z =^?_A f(a,x) \wedge z =^?_A f(x,a) \rrbracket}{h(x) \approx b \llbracket z =^?_A f(a,a) \wedge z =^?_A f(a,x) \wedge z =^?_A f(x,a) \rrbracket}$$

Si nous simplifions la contrainte de l'égalité conclusion de l'inférence précédente, nous obtenons l'égalité $h(x) \approx b \llbracket x =^?_A a \rrbracket$.

Cette dernière inférence est une inférence de paire critique basique, il ne peut s'agir d'une inférence de simplification basique, car $h(z) \approx h(x) \llbracket z =^?_A f(a,x) \wedge z =^?_A f(x,a) \rrbracket$ n'est pas substitution réduite relativement à $h(f(a,a)) \approx b$ modulo $\rho := id$.

Dans notre approche, cette dernière inférence est une inférence de simplification. La règle de simplification *Simplification* que nous proposons ne repose pas sur la condition de substitution-réductibilité-relativement-à mais seulement sur la validité de la contrainte de filtrage contrainte $h(z) \llbracket z =^?_A f(a,x) \wedge z =^?_A f(x,a) \rrbracket \lhd_A h(f(a,a))$ qui est vraie dans ce cas. Pour assurer la complétude, des règles d'inférence dans les contraintes ont été mises au point.

Cet exemple peut être généralisé. Reprenons le cas de la suite t_i définie précédemment. Supposons que nous ayons l'égalité $h(t_n) \approx b$ dans E au lieu de l'égalité $h(f(a,a)) \approx b$ donc $E = \{g(f(a,x),f(x,a)) \approx h(x),\ g(z,z) \approx h(z),\ h(t_n) \approx b\}$. Dans le cas de complétion basique modulo A, une série de n inférences de paires critiques basiques avec comme égalité « droite » l'égalité $h(z) \approx h(x) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!]$ est requise, considérant n instances différentes de $h(z) \approx h(x) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!]$. Ainsi, n nouvelles égalités sont créées.

Les inférences de paires critiques basiques sont les suivantes.

$$\frac{h(t_n) \approx b \qquad h(z) \approx h(x) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!]}{h(x) \approx b [\![x =_A^? t_{n-1}]\!]}$$

$$\frac{h(x) \approx b [\![x =_A^? t_i]\!] \qquad h(z) \approx h(x) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!]}{h(x) \approx b [\![x =_A^? t_{i-1}]\!]} \text{ pour } i \in \{1, \ldots ,n-1\}.$$

Dans notre approche, il s'agit de n inférences de simplification. Les égalités $h(t_n) \approx b$ et $h(x) \approx b [\![x =_A^? t_i]\!]$ pour $i \in \{1, \ldots ,n-1\}$ sont simplifiées car nous avons $h(z) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!] \lhd_A h(t_n)$ et $h(z) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!] \lhd_A h(x) [\![x =_A^? t_{i-1}]\!]$ pour $i \in \{1, \ldots ,n-1\}$ et il nous reste seulement l'égalité $h(x) \approx b [\![x =_A^? a]\!]$.

La complétion basique modulo A avec simplification basique génère donc n égalités sur cet exemple tandis que notre méthode permet d'en déduire une seule. L'exemple pourrait être encore généralisé pour produire un nombre exponentiel d'égalités dans le cas de la complétion basique modulo A avec simplification basique et dans notre cas, nous n'obtiendrions toujours qu'une seule égalité.

Pour régler l'inconvénient de la simplification basique qui est de permettre peu de simplifications, il est toutefois possible de rétracter l'égalité $h(z) \approx h(x) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!]$. Dans notre cas, nous avons besoin de n instances particulières de l'égalité $h(z) \approx h(x) [\![z =_A^? f(a,x) \wedge z =_A^? f(x,a)]\!]$ et résoudre les contraintes supprime l'avantage majeur de la complétion basique.

Pour résumer, la complétion basique modulo A permet de faire terminer l'exemple donné. Cet exemple ne termine pas pour la complétion modulo A. La complétion basique modulo A avec simplification basique fait toutefois exploser l'espace de recherche car peu de simplifications sont autorisées. La rétraction n'est pas une solution satisfaisante. Notre approche propose un nouveau traitement des contraintes pour améliorer les points faibles de la complétion basique modulo avec simplification basique.

5.4 Le système d'inférence $BCICS_m$

Nous présentons dans cette section le système d'inférence de la complétion basique modulo un ensemble d'axiomes AX, $BCICS_m$, composé des règles d'inférence *Paire Critique Basique 2*, *Simplification*, *Paire Critique Basique dans la Contrainte* et *Simplification dans la Contrainte*. Les principales caractéristiques de ce système d'inférence est qu'il utilise des contraintes sans les résoudre mais seulement en testant leur satisfaisabilité et qu'il autorise des inférences dans les contraintes. De plus, il possède une stratégie de contraction concrète. Par concrète, nous entendons que le fossé entre la théorie et la pratique ne soit pas infranchissable.

5.4.1 La règle d'inférence *Paire Critique Basique 2*

La règle *Paire Critique Basique 2* est la suivante.

Paire Critique Basique 2

$$\frac{u[s']_\omega \approx v\,[\![c_1]\!] \qquad s \approx t\,[\![c_2]\!]}{u[t]_\omega \approx v\,[\![s =^?_{AX} s' \ \wedge\ c_1 \ \wedge\ c_2]\!]} \quad \text{si :}$$

- $s \approx t\,[\![c_2]\!]$ est une égalité existante ou l'extension d'une égalité existante,
- $u[s']_\omega \approx v\,[\![c_1]\!]$ est (1) une égalité existante ou (2) l'extension d'une égalité existante et $u = s'$,
- s' n'est pas une variable,
- soit $\psi := s =^?_{AX} s' \ \wedge\ c_1 \ \wedge\ c_2$,
- ψ est satisfaisable, et
- $((u[s']_\omega \succ^?_t v) \ \wedge\ (s \succ^?_t t) \ \wedge\ (u \approx v \succ^?_e s \approx t))\,[\![\psi]\!]$ est satisfaisable.

La transition correspondant à cette règle d'inférence est la suivante. Γ est un ensemble d'égalités.

$$\{u[s']_\omega \approx v[\![c_1]\!],\, s \approx t[\![c_2]\!]\} \cup \Gamma \Rightarrow \{u[s']_\omega \approx v[\![c_1]\!],\, s \approx t[\![c_2]\!],\, u[t]_\omega \approx v[\![s =^?_{AX} s' \ \wedge\ c_1 \ \wedge\ c_2]\!]\} \cup \Gamma$$

Dans la complétion modulo et dans la complétion basique modulo, il est nécessaire d'étendre les égalités pour réaliser les inférences nécessaires à la complétude du système d'inférence. Dans le système d'inférence $BCICS_m$, les extensions sont également indispensables. L'application de la règle *Paire Critique Basique 2* et la nécessité des extensions sont illustrées dans l'exemple 39. Dans ce chapitre, les exemples traités concernent la théorie vide, la théorie A, la théorie C ou la théorie AC.

Exemple 39 – *Considérons les deux égalités* $f(a,b) \approx c$ *et* $f(b,d) \approx e$ *et supposons que* f *est un symbole de fonction associatif.*

Nous pouvons appliquer la règle Paire Critique Basique 2 *entre* $f(a,b) \approx c$ *et* $f(b,d) \approx e$ *si nous utilisons l'extension* $f(f(a,b),x') \approx f(c,x')$ *de* $f(a,b) \approx c$ *et l'extension* $f(y',f(b,d)) \approx f(y',e)$ *de* $f(b,d) \approx e$. *Nous déduisons l'égalité* $f(c,x') \approx f(y',e)[\![f(f(a,b),x') =^?_A f(y',f(b,d))]\!]$. *Sans les extensions, nous ne pourrions prouver que* $f(c,d) \approx f(a,e)$ *est vraie par une preuve par réécriture.*

– *Supposons que* f *est un symbole associatif et considérons l'application de la règle* Paire Critique Basique 2 *entre* $x \approx d[\![x =^?_A f(a,b)]\!]$ *et* $f(b,c) \approx e$. *L'égalité* $x \approx d[\![x =^?_A f(a,b)]\!]$ *doit être étendue en* $f(x,z') \approx f(d,z')\,[\![x =^?_A f(a,b)]\!]$ *et l'égalité* $f(b,c) \approx e$ *doit être étendue en* $f(x',f(b,c)) \approx f(x',e)$. *L'inférence* Paire Critique Basique 2 *à réaliser a donc lieu entre* $f(x,z') \approx f(d,z')[\![x =^?_A f(a,b)]\!]$ *et* $f(x',f(b,c)) \approx f(x',e)$. *Nous déduisons l'égalité* $f(x',e) \approx f(d,z')\,[\![x =^?_A f(a,b) \wedge f(x,z') =^?_A f(x',f(b,c))]\!]$.

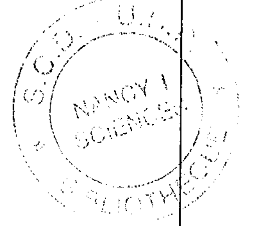
5.4.2 La règle d'inférence *Simplification*

La règle *Simplification* est la suivante.

Simplification

$$\frac{u[s']_\omega \approx v\,[\![c_1]\!] \qquad s \approx t\,[\![c_2]\!]}{u[t]_\omega \approx v\,[\![s =^?_{AX} s' \;\wedge\; c_1 \;\wedge\; c_2]\!]} \quad \text{si :}$$

- $s \approx t\,[\![c_2]\!]$ est une égalité existante ou une extension d'une égalité existante,
- $u \approx v\,[\![c_1]\!]$ est une égalité existante,
- s' n'est pas une variable,
- $s\,[\![c_2]\!] \lhd_{AX} s'\,[\![c_1]\!]$, et
- soit $\psi := s =^?_{AX} s' \;\wedge\; c_1 \;\wedge\; c_2$,
- $((s \succ^?_t t) \;\wedge\; (u \approx v \succ^?_e s \approx t))\,[\![\psi]\!]$ est valide.



L'égalité $u[s']_\omega \approx v\,[\![c_1]\!]$ est supprimée de l'espace de recherche.

La transition correspondant à cette règle d'inférence est la suivante. Γ est un ensemble d'égalités.

$$\{u[s']_\omega \approx v\,[\![c_1]\!],\, s \approx t\,[\![c_2]\!]\} \cup \Gamma \Rightarrow \{s \approx t\,[\![c_2]\!],\, u[t]_\omega \approx v\,[\![s =^?_{AX} s' \;\wedge\; c_1 \;\wedge\; c_2]\!]\} \cup \Gamma$$

L'originalité de cette règle d'inférence est qu'aucun filtre n'est appliqué dans la conclusion comme c'est le cas pour les règles *Simplification Standard 1*, *Simplification Standard 2*, *Simplification Basique* et *Simplification E-cycle*. Seule la validité d'un problème de filtrage AX contraint est testée et le problème de filtrage AX contraint est sauvé comme un problème d'unification AX dans la contrainte de l'égalité conclusion.

L'exemple 40 illustre l'utilisation de la règle d'inférence *Simplification*.

Exemple 40 – *Supposons que $AX = \emptyset$. Considérons les égalités $f(g(a)) \approx b$ et $g(x) \approx h(x)\,[\![x =^? a]\!]$.*

$f(g(a)) \approx b$ peut être simplifiée par $g(x) \approx h(x)\,[\![x =^? a]\!]$ en utilisant la règle Simplification, *car les conditions d'application de cette règle sont vérifiées. En particulier, la contrainte de filtrage contrainte $g(x)\,[\![x =^? a]\!] \lhd^? g(a)$ est valide à cause du filtre $\rho = id$. Le problème de filtrage est sauvé dans la contrainte de l'égalité conclusion comme une contrainte d'unification, ici $g(x) =^? g(a)$. L'inférence de simplification est la suivante.*

$$\frac{f(g(a)) \approx b \qquad g(x) \approx h(x)\,[\![x =^? a]\!]}{f(h(x)) \approx b\,[\![g(x) =^? g(a) \;\wedge\; x =^? a]\!]}$$

Nous pouvons remarquer que cette inférence n'est pas une inférence de simplification basique, la condition de substitution-réductibilité-relativement-à n'étant pas satisfaite.

– *Considérons que AX est la théorie commutative. Soit k un symbole commutatif. Soient les égalités $f(g(a)) \approx b$ et $g(x) \approx h(y,z)\,[\![x =^?_C a \wedge k(y,z) =^?_C k(c,d)]\!]$.*

L'égalité $f(g(a)) \approx b$ peut être simplifiée par $g(x) \approx h(y,z)\,[\![x =^?_C a \wedge k(y,z) =^?_C k(c,d)]\!]$ en utilisant la règle Simplification, *car les conditions d'application de cette règle sont vérifiées. En particulier, la contrainte de filtrage contrainte $g(x)\,[\![x =^?_C a \wedge k(y,z) =^?_C k(c,d)]\!] \lhd^?_C g(a)$ est valide à cause du filtre $\rho = id$. Le problème de filtrage est sauvé dans la contrainte de l'égalité conclusion comme une contrainte d'unification ici, $g(x) =^?_C g(a)$. L'inférence de simplification est la suivante.*

$$\frac{f(g(a)) \approx b \qquad g(x) \approx h(y,z)\,[\![x =^?_C a \wedge k(y,z) =^?_C k(c,d)]\!]}{f(h(y,z)) \approx b\,[\![g(a) =^?_C g(x) \;\wedge\; x =^?_C a \;\wedge\; k(y,z) =^?_C k(c,d)]\!]}$$

Nous pouvons remarquer que cette inférence n'est pas une inférence de simplification basique, la condition de substitution-réductibilité-relativement-à n'étant pas satisfaite.

5.4.3 La règle d'inférence *Paire Critique Basique dans la contrainte*

Les règles d'inférence dans les contraintes ont été définies pour assurer la complétude du système d'inférence $BCICS_m$. Une inférence dans la contrainte déduit une égalité conclusion avec le même squelette que l'égalité « gauche » de l'inférence. La contrainte de l'égalité « gauche » est transformée pour former la contrainte de l'égalité conclusion.

Nous décrivons la règle d'inférence *Paire Critique Basique dans la contrainte* en utilisant la fonction $TransfoCP$ définie par des règles de réécriture nommées et la stratégie d'application de ces règles nommées *OneDedStrat*. Pour ce faire, nous utilisons la syntaxe ELAN (Borovanský, Cirstea, Dubois, Kirchner, Kirchner, Moreau, Ringeissen et Vittek, 1998). Dans cette section, nous présentons d'abord la fonction $TransfoCP$, puis la stratégie *OneDedStrat*, et enfin la règle d'inférence *Paire Critique Basique dans la contrainte.*

La fonction $TransfoCP$ est définie par les règles de réécriture nommées *Transfo* et *EndTransfoCP*. $TransfoCP$ a comme arguments trois égalités et une contrainte d'unification AX et retourne comme résultat une égalité. Le deuxième et le quatrième argument de la fonction $TransfoCP$ ne sont jamais modifiés. Ils sont utilisés pour sauver les prémisses originaux de l'inférence et sont utilisés à certaines étapes de réécriture. Le troisième argument sauve tous les unificateurs impliqués dans le traitement de l'inférence.

L'égalité conclusion de l'application de la règle d'inférence *Paire Critique Basique dans la contrainte* sur une égalité e_1 comme égalité « gauche » et e_2 comme égalité « droite » est calculée par le terme $TransfoCP(e_1,e_2,\top,e_1)$ sur lequel nous appliquons la stratégie *OneDedStrat*. Lorsque nous appliquons la stratégie *AllDedStrat* sur le terme $TransfoCP(e_1,e_2,\top,e_1)$, toutes les égalités conclusion de l'inférence *Paire Critique Basique dans la contrainte* entre e_1 et e_2 sont déduites. Nous l'écrivons $(AllDedStrat)TransfoCP(e_1,e_2,\top,e_1)$. La stratégie *OneDedStrat*, quant à elle, appliquée sur le terme $TransfoCP(e_1,e_2,\top,e_1)$ renvoie une seule égalité conclusion de l'inférence *Paire Critique Basique dans la contrainte* entre e_1 et e_2. L'application de la stratégie *OneDedStrat* sur le terme $TransfoCP(e_1,e_2,\top,e_1)$ est notée $(OneDedStrat)TransfoCP(e_1,e_2,\top,e_1)$.

La règle de réécriture *Transfo* est la suivante.

[Transfo]

$TransfoCP(e\llbracket u =^?_{AX} v[s'] \wedge c\rrbracket \ , \ s \approx t\llbracket c_2\rrbracket \ , \ \psi, \ e\llbracket c'\rrbracket)$

$\rightarrow$

$TransfoCP(e\llbracket u =^?_{AX} v[t''] \wedge s' =^?_{AX} s'' \wedge c\rrbracket \ , \ s \approx t\llbracket c_2\rrbracket \ , \ s' =^?_{AX} s'' \wedge \psi \ , \ e\llbracket c'\rrbracket)$

si :

- $s'' \approx t''$ est $s \approx t$ ou une extension de $s \approx t$,
- s' n'est pas une variable,
- $s' =^?_{AX} s''$ est satisfaisable, et
- $u =^?_{AX} v[s'] \wedge c$ (a) n'est pas satisfaisable ou (b) la règle *Transfo* est appliquée pour la première fois.

- Dans la règle $Transfo$, la condition d'extension de $s \approx t$ est que $top(s')$ est un symbole de la théorie AX. Nous n'étendons pas dans la contrainte. Ainsi, seule l'égalité « droite » d'une inférence *Paire Critique Basique dans la Contrainte* peut être étendue.
- La condition (a) implique que la règle de réécriture *Transfo* est seulement appliquée lorsque la contrainte de l'égalité en premier argument de la fonction $TransfoCP$ est insatisfaisable. Initialement, le premier argument de la fonction $TransfoCP$ est l'égalité « gauche » de l'inférence à réaliser. La contrainte de cette égalité est satisfaisable, c'est pourquoi la condition (b) est présente.

La règle de réécriture *EndTransfoCP* est la suivante :

[EndTransfoCP]
$TransfoCP(u \approx v\,[\![c_1]\!]\ ,\ s \approx t\,[\![c_2]\!],\ \psi,\ u \approx v\,[\![c]\!])$
$\rightarrow$
$u \approx v\,[\![c_1 \wedge c_2]\!]$
si :
- $c_1 \wedge c_2$ est satisfaisable, et
- $u \approx v\,[\![c]\!] \not\Delta_{AX}\ u \approx v\,[\![c_1 \wedge c_2]\!]$.

La dernière condition d'application de la règle de réécriture *EndTransfoCP* permet d'éviter les inférences à des positions de variables.

Par exemple, cette condition nous permet d'éviter une inférence de *Paire Critique Basique dans la contrainte* entre l'égalité « gauche » $f(x) \approx x\,[\![y =^? a \wedge x =^? a]\!]$ et l'égalité « droite » $a \approx b$ si nous considérons $f(x) \approx x\,[\![y =^? b \wedge x =^? a]\!]$ comme l'égalité conclusion. En effet, y n'est pas une variable de $f(x) \approx x$. $f(x) \approx x\,[\![y =^? a \wedge x =^? a]\!]$ et $f(x) \approx x\,[\![y =^? b \wedge x =^? a]\!]$ représentent la même égalité. Une inférence dans la contrainte de ce type est inutile.

Nous définissons maintenant les stratégies *AllDedStrat* et *OneDedStrat*.

La stratégie *AllDedStrat* appliquée sur un terme $TransfoCP(e_1, e_2, \top, e_1)$ nous permet de calculer toutes les égalités conclusion de l'application d'une inférence *Paire Critique Basique dans la contrainte* entre l'égalité « gauche » e_1 et l'égalité « droite » e_2. Elle contrôle l'application des règles de réécriture *Transfo* et *EndTransfoCP*. La règle *Transfo* est appliquée une première fois (si possible) et ensuite jusqu'à ce que la contrainte de l'égalité en premier argument de la fonction *Transfo* soit satisfaisable. La règle *EndTransfoCp* est alors appliquée. Si l'application de la stratégie *AllDedStrat* échoue, alors la règle *Paire Critique Basique dans la Contrainte* ne peut pas s'appliquer sur les deux égalités données en entrée. La stratégie *OneDedStrat* fournit un résultat parmi ceux calculés par *AllDedStrat*. Elle échoue si la stratégie *AllDedStrat* échoue.

L'écriture ELAN de ces deux stratégies est la suivante.

[AllDedStrat]
```
repeat*(don't know(Transfo));
EndTransfoCP
```

[OneDedStrat]
```
dc one(
   repeat*(don't know(Transfo));
   EndTransfoCP)
```

Considérons quelques exemples.

Exemple 41 *Soit $AX = \emptyset$.*

Nous considérons les deux égalités $h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!]$ et $g(a) \approx b$.

(AllDedStrat)$TransfoCP(h(x,y) \approx x[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!], g(a) \approx b, \top, h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$ permet de déduire les trois égalités calculées ci-dessous. Pour chaque application de la règle $Transfo$, le terme choisi pour s' est souligné.

1. $TransfoCP(h(x,y) \approx x[\![f(x,y,x) =^? f(\underline{g(z)},g(z'),g(z))]\!], g(a) \approx b, \top, h(x,y) \approx x[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

$\rightarrow^{Transfo}$

$TransfoCP(h(x,y) \approx x\,[\![f(x,y,x) =^? f(b,\underline{g(z')},g(z)) \wedge g(z) =^? g(a)]\!],\ g(a) \approx b,\ g(z) =^? g(a),\ \ h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

La contrainte $f(x,y,x) =^? f(b,g(z'),g(z)) \wedge g(z) =^? g(a)$ *est insatisfaisable. Nous appliquons donc la règle de réécriture* Transfo.

$\rightarrow^{Transfo}$

$TransfoCP(h(x,y) \approx x\,[\![f(x,y,x) =^? f(b,b,\underline{g(z)}) \wedge g(z) =^? g(a) \wedge g(z') =^? g(a)]\!],\ g(a) \approx b,\ g(z) =^? g(a) \wedge g(z') =^? g(a),\ h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

$\rightarrow^{Transfo}$

$TransfoCP(h(x,y) \approx x\,[\![f(x,y,x) =^? f(b,b,b) \wedge g(z) =^? g(a) \wedge g(z') =^? g(a)]\!],\ g(a) \approx b,\ g(z) =^? g(a) \wedge g(z') =^? g(a),\ h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

La contrainte $f(x,y,x) =^? f(b,b,b) \wedge g(z) =^? g(a) \wedge g(z') =^? g(a)$ *est satisfaisable. Nous appliquons donc la règle de réécriture* EndTransfoCP.

$\rightarrow^{EndTransfoCP}$

$h(x,y) \approx x\,[\![f(x,y,x) =^? f(b,b,b) \wedge g(z) =^? g(a) \wedge g(z') =^? g(a)]\!]$

En simplifiant la contrainte, l'égalité obtenue est $h(x,y) \approx x\,[\![x =^? b \wedge y =^? b]\!]$ *(1).*

2. $TransfoCP(h(x,y) \approx x[\![f(x,y,x) =^? f(\underline{g(z)},g(z'),g(z))]\!],\ g(a) \approx b,\ \top,\ h(x,y) \approx x[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

   $\rightarrow^{Transfo}$

   $TransfoCP(h(x,y) \approx x\,[\![f(x,y,x) =^? f(b,\underline{g(z')},g(z)) \wedge g(z) =^? g(a)]\!],\ g(a) \approx b,\ g(z) =^? g(a),\ h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

   $\rightarrow^{Transfo}$

   $TransfoCP(h(x,y) \approx x\,[\![f(x,y,x) =^? f(b,g(z'),b) \wedge g(z) =^? g(a)]\!],\ g(a) \approx b,\ g(z) =^? g(a),\ h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

   $\rightarrow^{EndTransfoCP}$

   $h(x,y) \approx x\,[\![f(x,y,x) =^? f(b,g(z'),b) \wedge g(z) =^? g(a)]\!]$

   En simplifiant la contrainte, l'égalité obtenue est $h(x,y) \approx x\,[\![x =^? b \wedge y =^? g(z')]\!]$ *(2).*

3. $TransfoCP(h(x,y) \approx x[\![f(x,y,x) =^? f(g(z),\underline{g(z')},g(z))]\!],\ g(a) \approx b,\ \top,\ h(x,y) \approx x[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

   $\rightarrow^{Transfo}$

   $TransfoCP(h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),b,g(z)) \wedge g(z') =^? g(a)]\!],\ g(a) \approx b,\ g(z') =^? g(a),\ h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!])$

   $\rightarrow^{EndTransfoCP}$

   $h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),b,g(z)) \wedge g(z') =^? g(a)]\!]$

   En simplifiant la contrainte, l'égalité obtenue est $h(x,y) \approx x\,[\![x =^? g(z) \wedge y =^? b]\!]$ *(3).*

Dans cet exemple, seules les égalités (2) et (3) sont utiles. En effet, nous pourrions introduire un critère de minimalité pour comparer les différentes égalités déduites et ainsi supprimer certaines égalités. Ce critère serait basé sur une sélection des termes s' qui doivent être utilisés dans l'application de la règle $Transfo$. L'égalité (1) n'est pas utile, car dans la contrainte de l'égalité (1), nous réécrivons tous les termes qui sont réécrits dans l'égalité (3) ou l'égalité (2) plus d'autres.

Exemple 42 *Considérons maintenant le cas où $AX = A$.*

1. *Considérons les deux égalités $g(x) \approx x\,[\![f(x,z) =^?_A f(a,f(y,b))]\!]$ et $f(a,c) \approx d$. $(AllDedStrat)TransfoCP(g(x) \approx x\,[\![f(x,z) =^?_A f(a,f(y,b))]\!],\ f(a,c) \approx d,\ \top,\ g(x) \approx x\,[\![f(x,z) =^?_A f(a,f(y,b))]\!])$ déduit une seule égalité conclusion calculée dans la suite. Dans le calcul, l'égalité $f(a,c) \approx d$ est étendue en l'égalité $f(f(a,c),x') \approx f(d,x')$. Pour chaque application de la règle $Transfo$, le terme choisi pour s' est souligné.*

   $TransfoCP(g(x) \approx x\,[\![f(x,z) =^?_A \underline{f(a,f(y,b))}]\!],\ f(a,c) \approx d,\ \top,\ g(x) \approx x\,[\![f(x,z) =^?_A f(a,f(y,b))]\!])$
   $\rightarrow_{Transfo}$
   $TransfoCP(g(x) \approx x\,[\![f(x,z) =^?_A f(d,x') \wedge f(f(a,c),x') =^?_A f(a,f(y,b))]\!],\ f(a,c) \approx d,\ f(f(a,c),x') =^?_A f(a,f(y,b)),\ g(x) \approx x\,[\![f(x,z) =^?_A f(a,f(y,b))]\!])$
   $\rightarrow_{EndTransfoCP}$
   $g(x) \approx x\,[\![f(x,z) =^?_A f(d,x') \wedge f(f(a,c),x') =^?_A f(a,f(y,b))]\!]$

   La contrainte de l'égalité de départ $f(x,z) =^?_A f(a,f(y,b))$ a trois solutions plus générales $\sigma_1 := \{x \mapsto a, z \mapsto f(y,b)]$, $\sigma_2 := \{x \mapsto f(a,y), z \mapsto b\}$ et $\sigma_3 := \{x \mapsto f(a,x'), y \mapsto f(x',y'), z \mapsto f(y',b)\}$.
   La contrainte de l'égalité finale $f(x,z) =^?_A f(d,x') \wedge f(f(a,c),x') =^?_A f(a,f(y,b))$ a deux solutions plus générales $\sigma'_2 := \{x \mapsto d, y \mapsto c, z \mapsto b, x' \mapsto d\}$ et $\sigma'_3 := \{x \mapsto d, y \mapsto f(c,y'), z \mapsto f(y',b), x' \mapsto f(y',b)\}$.
   Nous pouvons remarquer que la transformation de la contrainte de départ $f(x,z) =^?_A f(a,f(y,b))$ correspond à modifier les solutions σ_2 et σ_3 de cette contrainte pour former respectivement σ'_2 et σ'_3.

2. *Considérons les deux égalités $g(x,y) \approx x\,[\![f(a,x) =^?_A f(x,a)]\!]$ et $f(a,a) \approx b$.*
   La contrainte $f(x,a) =^?_A f(a,x)$ a un nombre infini de solutions plus générales. $\sigma_1 := \{x \mapsto a\}$, $\sigma_2 := \{x \mapsto f(a,a)\}$, $\sigma_3 := \{x \mapsto f(a,f(a,a))\} \cdots$ sont solutions.
   $(AllDedStrat)TransfoCP(g(x,y) \approx x\,[\![f(a,x) =^?_A f(x,a)]\!], f(a,a) \approx b)$ contient un ensemble infini d'égalités. Considérons la dérivation d'une égalité issue de $(AllDedStrat)TransfoCP(g(x,y) \approx x\,[\![f(a,x) =^?_A f(x,a)]\!], f(a,a) \approx b)$.
   Dans la suite, nous remarquons que l'égalité $f(a,a) \approx b$ est étendue en l'égalité $f(z',f(a,a)) \approx f(z',b)$. Pour chaque application de la règle $Transfo$, le terme choisi pour s' est souligné.

   $TransfoCP(g(x,y) \approx x\,[\![f(a,x) =^?_A \underline{f(x,a)}]\!],\ f(a,a) \approx b,\ \top,\ g(x,y) \approx x\,[\![f(a,x) =^?_A f(x,a)]\!])$
   $\rightarrow_{Transfo}$
   $TransfoCP(g(x,y) \approx x[\![f(a,x) =^?_A \underline{f(z',b)} \wedge f(x,a) =^?_A f(z',f(a,a))]\!],\ f(a,a) \approx b,\ f(x,a) =^?_A f(z',f(a,a)),\ g(x,y) \approx x\,[\![f(a,x) =^?_A f(x,a)]\!])$

$\rightarrow^{Transfo}$

$TransfoCP(g(x,y) \approx x[\![f(a,x) =^?_A f(z',b) \wedge f(x,a) =^?_A f(b,y') \wedge f(f(a,a),y') =^?_A f(z',f(a,a))]\!], f(a,a) \approx b,\ f(x,a) =^?_A f(z',f(a,a)) \wedge f(z',f(a,a)) =^?_A f(f(a,a),y'),\ g(x,y) \approx x[\![f(a,x) =^?_A f(x,a)]\!])$

$\rightarrow^{EndTransfoCP}$

$g(x,y) \approx x[\![f(a,x) =^?_A f(z',b) \wedge f(x,a) =^?_A f(b,y') \wedge f(f(a,a),y') =^?_A f(z',f(a,a))]\!]$

La contrainte $f(a,x) =^?_A f(z',b) \wedge f(x,a) =^?_A f(b,y') \wedge f(f(a,a),y') =^?_A f(z',f(a,a))$ est satisfaisable. $\sigma' := \{x \mapsto b\}$ est une solution. Cette solution correspond à la transformation de la solution $\sigma := \{x \mapsto f(a,a)\}$ de la contrainte $f(a,x) =^?_A f(x,a)$.

Nous définissons maintenant la règle d'inférence *Paire Critique Basique dans la Contrainte* en utilisant la fonction $TransfoCP$ et la stratégie *OneDedStrat.*

Paire Critique Basique dans la Contrainte

$$\frac{u \approx v\,[\![c_1]\!] \qquad s \approx t\,[\![c_2]\!]}{(OneDedStrat)TransfoCP(u \approx v\,[\![c_1]\!], s \approx t\,[\![c_2]\!], \top, u \approx v\,[\![c_1]\!])} \quad \text{si :}$$

- $((u \succ_t v) \ \wedge\ (s \succ_t t) \ \wedge\ (u \approx v \succ_e s \approx t))\,[\![c_1 \wedge c_2]\!]$ est satisfaisable.

Cette règle d'inférence s'écrit sous la forme de la transition suivante. Γ est un ensemble d'égalités.

$$\{u \approx v\,[\![c_1]\!],\ s \approx t\,[\![c_2]\!]\} \cup \Gamma$$

$$\Rightarrow$$

$$\{u \approx v\,[\![c_1]\!],\ s \approx t\,[\![c_2]\!],\ (OneDedStrat)TransfoCP(u \approx v\,[\![c_1]\!], s \approx t\,[\![c_2]\!], \top, u \approx v\,[\![c_1]\!])\} \cup \Gamma$$

Nous fournissons maintenant les exemples d'application de la règle *Paire Critique Basique dans la Contrainte* correspondant aux calculs effectués dans les exemples 41 et 42.

Exemple 43 *1. Dans l'exemple 41, nous avons effectué les trois inférences de* Paire Critique Basique dans la Contrainte *suivantes.*

(a)
$$\frac{h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!] \qquad g(a) \approx b}{h(x,y) \approx x\,[\![x =^? b \wedge y =^? b]\!]\ (1)}$$

(b)
$$\frac{h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!] \qquad g(a) \approx b}{h(x,y) \approx x\,[\![x =^? b \wedge y =^? g(z')]\!]\ (2)}$$

(c)
$$\frac{h(x,y) \approx x\,[\![f(x,y,x) =^? f(g(z),g(z'),g(z))]\!] \qquad g(a) \approx b}{h(x,y) \approx x\,[\![x =^? g(z) \wedge y =^? b]\!]\ (3)}$$

2. Dans l'exemple 42, nous avons effectué les deux inférences de Paire Critique Basique dans la Contrainte *suivantes.*

(a)
$$\frac{g(x) \approx x\,[\![f(x,z) =^?_A f(a,f(y,b))]\!] \qquad f(a,c) \approx d}{g(x) \approx x\,[\![f(x,z) =^?_A f(d,x') \wedge f(f(a,c),x') =^?_A f(a,f(y,b))]\!]}$$

(b)
$$\frac{g(x,y) \approx x\,[\![f(a,x) =^?_A f(x,a)]\!] \qquad f(a,a) \approx b}{g(x,y) \approx x\,[\![f(a,x) =^?_A f(z',b) \wedge f(x,a) =^?_A f(b,y') \wedge f(f(a,a),y') =^?_A f(z',f(a,a))]\!]}$$

La caractéristique de $BCICS_m$ évoquée ci-dessous présente l'un des avantages du système d'inférence $BCICS_m$. $BCICS_m$ contrôle l'expansion de l'espace de recherche. Une application de la règle d'inférence *Paire Critique Basique dans la contrainte* correspond dans le cas de la complétion standard modulo à l'application d'une fois la règle de *Paire Critique* plus éventuellement à l'application de la règle de *Simplification Standard* une ou plusieurs fois. Cet avantage est dû à l'utilisation des contraintes. L'exemple 44 illustre cet avantage.

Exemple 44 *Considérons l'inférence de* Paire Critique Basique dans la Contrainte *suivante.*

$$\frac{h(x,y) \approx x \llbracket f(x,y,x) =^? f(g(z),g(z'),g(z)) \rrbracket \qquad g(a) \approx b}{h(x,y) \approx x \llbracket f(x,y,x) =^? f(b,g(z'),b) \wedge g(z) =^? g(a) \rrbracket}$$

Si nous appliquons la contrainte, cet inférence correspond à une inférence de Paire Critique *et une inférence de* Simplification Standard *de la complétion standard. Ces inférences sont données ci-dessous.*

$$\frac{h(g(z),g(z')) \approx g(z) \qquad g(a) \approx b}{h(b,g(x')) \approx g(a)} \ (\text{Paire Critique})$$

$$\frac{h(b,g(z')) \approx g(a) \qquad g(a) \approx b}{h(b,g(z')) \approx b} \ (\text{Simplification Standard})$$

5.4.4 La règle d'inférence *Simplification dans la Contrainte*

De la même manière que dans la section précédente, nous définissons une fonction *TransfoSimp* pour calculer la conclusion d'une inférence de *Simplification dans la Contrainte.* La fonction *TransfoSimp* est définie en utilisant la règle de réécritures *Transfo* définie précédemment et la règle de réécriture *EndTransfoSimp* définie ci-dessous. Elle a comme arguments trois égalités et une contrainte d'unification AX et retourne comme résultat une égalité. L'égalité conclusion de l'application de la règle d'inférence *Simplification dans la contrainte* sur une égalité e_1 comme égalité « gauche » et e_2 comme égalité « droite » est calculée par le terme $TransfoSimp(e_1,e_2,\top,e_1)$ sur lequel nous appliquons la stratégie *OneSimpStrat.* Nous l'écrivons $(OneSimpStrat)TransfoSimp(e_1,e_2,\top,e_1)$. La stratégie duale de *AllDedStrat*, *AllSimpStrat*, n'a aucun sens dans le cas d'une simplification.

La règle de réécriture *EndTransfoSimp* est la suivante.

[EndTransfoSimp]

$TransfoSimp(u \approx v \llbracket c_1 \rrbracket\ ,\ s \approx t \llbracket c_2 \rrbracket,\ \psi,\ u \approx v \llbracket c \rrbracket)$
$\rightarrow$
$u \approx v \llbracket c_1 \wedge c_2 \rrbracket$
si :
– $s' \llbracket c_2 \rrbracket \lhd_{AX} s'' \llbracket c \rrbracket$ pour tout $s' =^?_{AX} s''$ dans ψ,
– $c_1 \wedge c_2$ est satisfaisable,
– pour tout $s' =^?_{AX} s''$ dans ψ, (a) $head(s')$ n'est pas un symbole de la théorie AX, ou (b) s' est en dessous d'un symbole la théorie AX, et
– $u \approx v \llbracket c \rrbracket \not\lhd_{AX} u \approx v \llbracket c_1 \wedge c_2 \rrbracket$.

L'écriture ELAN de la stratégie *OneSimpStrat* est donnée ci-dessous.

[OneSimpStrat]

```
dc one(
```

```
repeat*(don't know(Transfo));
EndTransfoSimp)
```

La règle d'inférence *Simplification dans la contrainte* est définie comme suit en utilisant la fonction $TransfoSimp$ et la stratégie *OneSimpStrat*.

Simplification dans la Contrainte

$$\frac{u \approx v\,[\![c_1]\!] \qquad s \approx t\,[\![c_2]\!]}{(OneSimpStrat)TransfoSimp(u \approx v\,[\![c_1]\!], \top, s \approx t\,[\![c_2]\!], u \approx v\,[\![c_1]\!])} \quad \text{si :}$$

– $((s \succ_t t) \ \wedge \ (u \approx v \succ_e s \approx t))\,[\![c_1 \wedge c_2]\!]$ est satisfaisable.

L'égalité $u \approx v\,[\![c_1]\!]$ est supprimée de l'espace de recherche. Cette règle d'inférence s'écrit sous la forme de la transition ci-dessous. Γ est un ensemble d'égalités.

$$\{u \approx v\,[\![c_1]\!],\ s \approx t\,[\![c_2]\!]\} \cup \Gamma$$

$$\Rightarrow$$

$$\{s \approx t\,[\![c_2]\!],\ (OneSimpStrat)TransfoSimp(u \approx v\,[\![c_1]\!], \top, s \approx t\,[\![c_2]\!], u \approx v\,[\![c_1]\!])\} \cup \Gamma$$

La règle d'inférence *Simplification dans la Contrainte* est utilisée dans l'exemple suivant.

Exemple 45

$$\frac{g(f(x)) \approx x\,[\![f(\underline{g(y)}) =^? f(x) \wedge g(\underline{g(a)}) =^? g(x) \wedge y =^? a]\!] \qquad g(a) \approx b}{g(f(x)) \approx x\,[\![f(b) =^? f(x) \wedge g(b) =^? g(x) \wedge y =^? a \wedge g(y) =^? g(a) \wedge g(a) =^? g(a)]\!]}$$

Cette inférence a été obtenue en appliquant la stratégie OneSimpStrat sur le terme $TransfoSimp(g(f(x)) \approx x\,[\![f(g(y)) =^? f(x) \wedge g(g(a)) =^? g(x) \wedge y =^? a]\!], \top, g(a) \approx b, g(f(x)) \approx x\,[\![f(g(y)) =^? f(x) \wedge g(g(a)) =^? g(x) \wedge y =^? a]\!])$. *Pour chaque application de la règle* $Transfo$, *le terme choisi pour* s' *est souligné.*

$TransfoSimp(g(f(x)) \approx x\,[\![f(g(y)) =^? f(x) \wedge g(\underline{g(a)}) =^? g(x) \wedge y =^? a]\!], \top, g(a) \approx b, g(f(x)) \approx x\,[\![f(g(y)) =^? f(x) \wedge g(g(a)) =^? g(x) \wedge y =^? a]\!])$

$\rightarrow^{Transfo}$

$TransfoSimp(g(f(x)) \approx x\,[\![f(\underline{g(y)}) =^? f(x) \wedge g(b) =^? g(x) \wedge y =^? a \wedge g(a) =^? g(a)]\!], g(a) \approx b, g(a) =^? g(a), g(f(x)) \approx x\,[\![f(g(y)) =^? f(x) \wedge g(g(a)) =^? g(x) \wedge y =^? a]\!]$

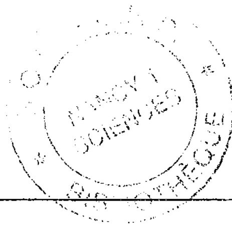
$\rightarrow^{Transfo}$

$TransfoSimp(g(f(x)) \approx x\,[\![f(b) =^? f(x) \wedge g(b) =^? g(x) \wedge y =^? a \wedge g(a) =^? g(a) \wedge g(y) =^? g(a)]\!], g(a) \approx b, g(a) =^? g(a) \wedge g(y) =^? g(a), g(f(x)) \approx x[\![f(g(y)) =^? f(x) \wedge g(g(a)) =^? g(x) \wedge y =^? a]\!]$

$\rightarrow^{EndTransfoSimp}$

$g(f(x)) \approx x\,[\![f(b) =^? f(x) \wedge g(b) =^? g(x) \wedge y =^? a \wedge g(a) =^? g(a) \wedge g(y) =^? g(a)]\!]$

La règle EndTransfoSimp *a pu être appliquée, car les conditions d'application de cette règle sont satisfaites. En particulier, nous avons* $g(a) \lhd g(a)$ *et* $g(y) \lhd g(a)$ *et* $g(f(x)) \approx x\,[\![f(g(y)) =^? f(x) \wedge g(g(a)) =^? g(x) \wedge y =^? a]\!] \not\lhd g(f(x)) \approx x\,[\![f(b) =^? f(x) \wedge g(b) =^? g(x) \wedge y =^? a \wedge g(a) =^? g(a) \wedge g(y) =^? g(a)]\!]$.



5.4.5 Terminaison

Conjecture 1 – *L'application des règles de réécriture* Transfo *et* EndTransfoCP *suivant la stratégie* OneDedStrat *termine.*
- *L'application des règles de réécriture* Transfo *et* EndTransfoSimp *suivant la stratégie* OneSimpStrat *termine.*

5.5 $BCICS_m$ est correct et complet

5.5.1 Correction

Dans cette section, nous prouvons la correction de $BCICS_m$. Normalement, la correction est prouvée en montrant que toutes les instances closes de la conclusion d'une inférence sont impliquées par les instances closes des prémisses de l'inférence. Les règles d'inférence *Paire Critique Basique 2* et *Simplification* ne posent pas de problème par rapport à la correction. Cependant, dans le cas d'une inférence dans la contrainte, les instances closes de la conclusion d'une inférence ne sont pas toujours impliquées par les instances closes des prémisses de l'inférence. L'exemple 46 illustre ce fait. Nous devons donc considérer pour la correction seulement les égalités qui sont *contraintes correctement.* Nous définissons cette notion dans la suite.

Exemple 46 *Considérons une inférence de* Paire Critique dans la contrainte *entre l'égalité « droite »* $g(x) = x\,[\![f(a) =^? f(x)]\!]$ *et l'égalité « gauche »* $f(a) \approx f(b)$. *Nous déduisons* $g(x) \approx x\,[\![f(b) =^? f(x)]\!]$. *Cette inférence dans la contrainte ne correspond pas à une inférence si nous appliquons les contraintes. En effet,* $g(a) \approx b$ *et* $f(a) \approx f(b)$ *n'impliquent pas* $g(b) \approx b$.

Dans la suite, nous avons besoin de voir les contraintes équationnelles comme des ensembles d'égalités.

Définition 50 *Soit* c *une contrainte équationnelle de la forme* $u_1 =^?_E v_1 \wedge \ldots \wedge u_n =^?_E v_n$.
$\overline{c}$ *est l'ensemble d'égalités* $\{u_1 \approx v_1, \ldots, u_n \approx v_n\}$. □

Nous définissons la notion d'*égalité contrainte correctement* par rapport à des ensembles d'égalités E et E_0, où E est une théorie équationnelle et E_0 est un ensemble d'égalités. Nous allons montrer que les égalités qui sont contraintes correctement sont impliquées par les égalités de l'ensemble initial à compléter E_0 modulo E.

Définition 51 *Soit* E *une théorie équationnelle et* E_0 *un ensemble d'égalités. Soit* $s \approx t\,[\![c]\!]$ *une égalité contrainte. Nous disons que* $s \approx t\,[\![c]\!]$ *est* contrainte correctement par rapport à E et E_0, *si* $E \cup E_0 \cup \overline{\sigma(c)} \models \sigma(s) \approx \sigma(t)$, *où* σ *est une substitution telle que pour toutes les variables* $x \in Var(c) \cup Var(s \approx t)$, $\sigma(x) = c_x$, *où* c_x *est une constante*[13]. *Un ensemble d'égalités est* contraint correctement par rapport à E et E_0, *si chaque égalité le composant est contrainte correctement par rapport à* E *et* E_0. □

Exemple 47 – *Dans l'exemple 46,* $g(x) \approx x[\![f(b) =^? f(x)]\!]$ *n'est pas contrainte correctement par rapport à* $g(x) = x\,[\![f(a) =^? f(x)]\!]$ *et* $f(a) \approx f(b)$, *car* $g(x) = x\,[\![f(a) =^? f(x)]\!], f(a) \approx f(b), f(c_x) \approx f(b) \not\models g(c_x) \approx c_x$, *où* c_x *est une constante.*

13. σ est utilisée pour exprimer le fait que c et $s \approx t$ ont des variables communes, mais c et $s \approx t$ ne partagent pas de variables avec les éléments des ensembles E et E_0.

- *Supposons que E soit la théorie vide. Soit $E_0 = \{f(a) \approx d$ (1), $f(x) \approx g(x)$ (2), $a \approx b$ (3)$\}$. L'égalité $g(x) \approx d \llbracket f(x) =^? f(a) \rrbracket$ est contrainte correctement par rapport à E et E_0, car $f(a) \approx d, f(x) \approx g(x), a \approx b, f(c_x) \approx f(a) \models g(c_x) \approx d$, où c_x est une constante.*

Nous montrons dans le lemme 12 que la règle de réécriture *Transfo* préserve la propriété « d'être correctement contrainte ».

Lemme 12 *Considérons l'application de la règle de réécriture* Transfo *suivante.*

$Transfo(u \approx v \llbracket c_1'[s_1'] \rrbracket, s \approx t \llbracket c_2 \rrbracket, c, u \approx v \llbracket c_1 \rrbracket) \rightarrow^{Transfo} Transfo(u \approx v \llbracket c_1'' \rrbracket, s \approx t \llbracket c_2 \rrbracket, c', u \approx v \llbracket c_1 \rrbracket)$

Alors, si $u \approx v \llbracket c_1' \wedge c_2 \rrbracket$ et $s \approx t \llbracket c_2 \rrbracket$ sont contraintes correctement par rapport à E et E_0, alors $u \approx v \llbracket c_1'' \wedge c_2 \rrbracket$ l'est également.

Preuve : c_1'' est de la forme $c_1'[t'] \wedge s_1' =^? s'$ et c' est de la forme $s_1' =^? s \wedge c$.
$u \approx v \llbracket c_1'[s_1'] \wedge c_2 \rrbracket$ est contrainte correctement par rapport à E et E_0, donc $E \cup E_0 \cup \overline{\sigma(c_1'[s_1']) \cup \sigma(c_2)} \models \sigma(u) \approx \sigma(v)$, où σ est une substitution telle que pour toutes les variables $x \in Var(c_1') \cup Var(u \approx v)$, $\sigma(x) = c_x$
$s \approx t \llbracket c_2 \rrbracket$ est contrainte correctement par rapport à E et E_0 donc $E \cup E_0 \cup \overline{\sigma'(c_2)} \models \sigma'(s) \approx \sigma'(t)$, où σ' est une substitution telle que pour toutes les variables $x \in Var(c_2) \cup Var(s \approx t)$, $\sigma'(x) = d_x$.
Soit $\sigma'' = \sigma'\sigma$. $u \approx v \llbracket c_1' \rrbracket$ et $s \approx t \llbracket c_2 \rrbracket$ ne partagent pas de variables, donc $E \cup E_0 \cup \overline{\sigma''(c_1'[s_1'])} \cup \overline{\sigma''(c_2)} \models \sigma''(u) \approx \sigma''(v)$ et $E \cup E_0 \cup \overline{\sigma''(c_2)} \models \sigma''(s) \approx \sigma''(t)$ et ainsi $E \cup E_0 \cup \overline{\sigma''(c_1[t'])} \cup \overline{\sigma''(c_2)} \cup \sigma''(s_1') \approx \sigma''(s') \models \sigma''(u) \approx \sigma''(v)$. D'où, $u \approx v \llbracket c_1'' \wedge c_2 \rrbracket$ est contrainte correctement par rapport à E et E_0. □

Le lemme précédent nous permet de conclure dans le lemme 13 qu'une inférence de *Paire Critique dans la contrainte* déduit une égalité contrainte correctement.

Le lemme suivant concerne le calcul de paires critiques. Cependant, le résultat peut être étendu pour la simplification.

Lemme 13 *Soient e_1 et e_2 deux égalités contraintes correctement.*

S'il y a une égalité telle que $e_3 \in (AllDedStrat)Transfo(e_1, e_2, \top, e_1)$, alors e_3 est contrainte correctement.

Preuve : Supposons qu'il y ait une inférence de *Paire Critique dans la contrainte* entre $e_1 = u \approx v \llbracket c_1 \rrbracket$ comme égalité « gauche » et $e_2 = s \approx t \llbracket c_2 \rrbracket$ comme égalité « droite », qui déduit $e_3 = u \approx v \llbracket c_1' \rrbracket$.
Nous avons prouvé dans le lemme 12 que la règle de réécriture *Transfo* préserve la propriété « d'être contrainte correctement ». La règle de réécriture *EndTransfoCP* préserve également cette propriété. En effet, si l'on considère une application de la règle de réécriture *EndTransfoCP* sur $Transfo(e, \ldots)$ où e est contrainte correctement, nous obtenons l'égalité e elle-même. Par récurrence, cela prouve que la règle d'inférence *Paire Critique dans la contrainte* préserve la propriété « d'être contrainte correctement », donc e_3 est contrainte correctement. □

Considérons l'exemple suivant.

Exemple 48 *Soit E la théorie équationnelle vide. Soient $E_0 = \{f(a) \approx b$ (1), $f(x) \approx g(x)$ (2), $a \approx c$ (3)$\}$. Considérons la précédence $f \succ_p g \succ_p a \succ_p b \succ_p c$.*

La simplification de $f(a) \approx b$ *par* $f(x) \approx g(x)$ *déduit l'égalité* $g(x) \approx b \llbracket f(x) =^? f(a) \rrbracket$. *L'égalité* $g(x) \approx b \llbracket f(x) =^? f(a) \rrbracket$ *est contrainte correctement par rapport à* E *et* E_0*, car* $f(a) \approx b, f(x) \approx g(x), a \approx c, f(c_x) \approx a \models g(c_x) \approx b$*, où* c_x *est une constante.*

Il y a alors une simplification dans la contrainte de $g(x) \approx b \llbracket x =^? a \rrbracket$ *par* $a \approx c$ *qui déduit l'égalité* $g(x) \approx b \llbracket x =^? c \rrbracket$.

En effet, la contrainte $x =^? a$ *de* $g(x) \approx b \llbracket x =^? a \rrbracket$ *est transformée en* $x =^? c$ *en utilisant la règle de réécriture* Transfo *puis la règle de réécriture* EndTransfoSimp. *Il est facile de voir que* $E \cup E_0 \cup d_x \approx a \models g(d_x) \approx b$ *et* $E \cup E_0 \models a \approx c$ *implique que* $E \cup E_0 \cup d_x \approx c \models g(d_x) \approx c$*, où* d_x *est une constante. Ainsi, l'égalité* $g(x) \approx b \llbracket x =^? c \rrbracket$ *est contrainte correctement par rapport à* E *et* E_0.

Le lemme suivant est prouvé à partir de la définition 51 et du lemme 13.

Lemme 14 *Soit* E *une théorie équationnelle et* E_0 *un ensemble d'égalités. Soit* E_i *un ensemble d'égalités contraintes correctement par rapport à* E *et* E_0*. Supposons que* $E_0 \cup E \models E_i$*. Soit* e *la conclusion d'une inférence entre éléments de* E_i*. Alors,* e *est contrainte correctement par rapport à* E *et* E_0.

Preuve : Nous distinguons l'inférence qui a permis d'obtenir e.

Cas 1 e est obtenue par une inférence *Paire Critique Basique 2.*
e est $u[t]_\omega \approx v \llbracket c_1 \wedge c_2 \wedge s =^?_E s' \rrbracket$. Elle est obtenue à partir des prémisses, e_1, $u[s']_\omega \approx v \llbracket c_1 \rrbracket$ et e_2, $s \approx t \llbracket c_2 \rrbracket$.
Comme $u[s']_\omega \approx v \llbracket c_1 \rrbracket$ est contrainte correctement par rapport à E et E_0, alors $u[s']_\omega \approx v \llbracket c_1 \wedge c_2 \rrbracket$ l'est également.
Comme $u[s']_\omega \approx v \llbracket c_1 \wedge c_2 \rrbracket$ et $s \approx t \llbracket c_2 \rrbracket$ sont contraintes correctement par rapport à E et E_0, $E \cup E_0 \cup \overline{\sigma(c_1)} \cup \overline{\sigma(c_2)} \models \sigma(u[s']_\omega) \approx \sigma(v)$ et $\overline{\sigma'(c_2)} \models \sigma'(s) \approx \sigma'(t)$.
Soit $\sigma'' = \sigma'\sigma$. Comme $u \approx v \llbracket c_1 \rrbracket$ et $s \approx t \llbracket c_2 \rrbracket$ ne partagent pas de variables, nous pouvons déduire que $E \cup E_0 \cup \overline{\sigma''(c_1)} \cup \overline{\sigma''(c_2)} \models \sigma''(u[s']_\omega) \approx \sigma''(v)$ et que $E \cup E_0 \cup \overline{\sigma''(c_2)} \models \sigma''(s) \approx \sigma''(t)$. Ainsi, $E \cup E_0 \cup \overline{\sigma''(c_1)} \cup \overline{\sigma''(c_2)} \cup \{\sigma''(s) \approx \sigma''(s')\} \models \sigma''(u[t]_\omega) \approx \sigma''(v)$. $u[t]_\omega \approx v \llbracket c_1 \wedge c_2 \wedge s =^?_E s' \rrbracket$, c'est-à-dire e, est donc contrainte correctement par rapport à E et E_0.

Cas 2 e est obtenue par l'application de la règle *Simplification.*
La preuve est similaire à la preuve du *Cas 1.*

Cas 3 e est obtenue par l'application de la règle *Paire Critique dans la contrainte.*
Soient e_1, $u \approx v \llbracket c_1 \rrbracket$ et e_2, $s \approx t \llbracket c_2 \rrbracket$.
Comme $u \approx v \llbracket c_1 \rrbracket$ et $s \approx t \llbracket c_2 \rrbracket$ sont contraintes correctement par rapport à E et E_0, e est contrainte correctement par rapport à E_0 et E d'après le lemme 13.

Cas 4 e est obtenue par l'application de la règle *Simplification dans la contrainte.*
La preuve de ce cas est similaire à la preuve du *Cas 3.*

□

Le lemme 15 est une conséquence du lemme 14.

Lemme 15 *Soit* E *une théorie équationnelle, et* E_0 *un ensemble d'égalités. Si* $s \approx t \llbracket c \rrbracket$ *est correctement contrainte par rapport à* E *et* E_0*, alors* $E \cup E_0 \models s \approx t \llbracket c \rrbracket$.

Preuve : $s \approx t \llbracket c \rrbracket$ est correctement contrainte par rapport à E et E_0, donc $E \cup E_0 \cup \overline{\sigma(c)} \models \sigma(u) \approx \sigma(v)$ où σ est une substitution telle que pour toutes les variables $x \in Var(c) \cup Var(u \approx v)$, $\sigma(x) = d_x$. D'où, $E \cup E_0 \models u \approx v \llbracket c \rrbracket$ par la définition de la sémantique d'une contrainte. □

Le théorème suivant est notre théorème de correction. Il montre que toute inférence de $BCICS_m$ est correcte. La preuve de ce théorème est une conséquence des lemmes précédents.

Théorème 20 *Soit E une théorie équationnelle et E_0 un ensemble d'égalités contraint correctement par rapport à E et E_0. Soit E_0, $E_1, \ldots$ une dérivation de complétion. Soit $e \in E_i$ pour un certain i. Alors, $E_0 \cup E \models e$.*

Preuve : La preuve est d'abord faite par récurrence. E_0 est un ensemble d'égalités contraintes correctement. Donc, par le lemme 14, pour tout i, E_i est contraint correctement. D'après le lemme 15, nous pouvons conclure que $E \cup E_0 \models e$ pour $e \in E_i$. □

La seule condition sur l'ensemble initial d'égalités à compléter est que les égalités de cette ensemble doivent être contraintes correctement par rapport à E et E_0. Cette condition est vérifiée si l'on part d'un ensemble non contraint, mais certains ensembles contraints sont également autorisés (ceux qui sont contraints correctement). Cette condition est intéressante, car certains systèmes d'inférence de la complétion basique requiert que l'ensemble initial des égalités ne soit pas contraint (Nieuwenhuis et Rubio, 1992b).

5.5.2 Complétude

Cette section concerne la complétude de $BCICS_m$.

Pour prouver la complétude, il est nécessaire de faire le lien entre le cas clos et le cas non clos (*lifting*). Si E est l'ensemble d'égalités à compléter, il faut montrer que toute égalité persistante de $Gr(E_\infty)$ est une instance close d'une égalité persistante de E_∞ et que toute inférence entre éléments de $Gr(E_\infty)$ peut être obtenue par l'instanciation d'une inférence entre éléments de E_∞.

Nous considérons deux cas. Nous traitons d'abord les inférences dans le cas clos qui ne sont pas des inférences à des positions de variables dans le cas non clos dans le lemme 16, puis nous considérons les inférences dans le cas clos qui sont des inférences dans la contrainte dans le cas non clos dans le lemme 2.

Lemme 16 *Soit $u \approx v\,[\![c_1]\!]$ une égalité ou l'extension d'une égalité. Soit $s \approx t\,[\![c_2]\!]$ une égalité ou l'extension d'une égalité. Soit σ_1 une solution close de c_1, et soit σ_2 une solution close de c_2. Supposons que $\sigma_1(u) = \sigma_1(u[s']_p)$, où $E \models \sigma_1(s') \approx \sigma_2(s)$, et p n'est pas une position de variables de u. Alors, il existe une inférence de Paire Critique Basique dont la conclusion a $\sigma_1(u)[\sigma_2(t)]_p \approx \sigma_1(v)$ comme instance.*

Preuve : Il existe une inférence de paire critique dans le cas clos qui peut être transposée à une inférence dans le cas non clos à une position qui n'est pas une position de variable. L'inférence close est :

$$\frac{\sigma_1(u)[\sigma_1(s')]_p \approx \sigma_1(v) \qquad \sigma_2(s) \approx \sigma_2(t)}{\sigma_1(u)[\sigma_2(t)] \approx \sigma_1(v)}$$

Comme p n'est pas une position de variable de u, alors il existe une inférence dans le cas non clos entre $u[s']_p \approx v\,[\![c_1]\!]$ et $s \approx t\,[\![c_2]\!]$ qui déduit la conclusion $u[t]_p \approx v\,[\![c_1 \wedge c_2 \wedge s =^?_E s']\!]$. $\sigma_1(u)[\sigma_2(t)] \approx \sigma_1(v)$ est une instance close de cette conclusion. □

Pour les inférences dans le cas clos qui sont des inférences dans la contrainte dans le cas non clos, le lemme est plus difficile à formuler.

Dans un premier temps, nous considèrons deux instances closes d'égalités impliquées dans une inférence dans le cas clos. Cette inférence est une inférence dans la contrainte dans le cas non clos. Nous montrons comment construire cette inférence dans le cas non clos. Il faut tenir

compte du fait qu'une inférence de $BCICS_m$ peut correspondre à plusieurs inférences dans le cas clos.

Conjecture 2 *Soient $u \approx v\,[\![c_1]\!]$ et $s \approx t\,[\![c_2]\!]$ deux égalités. Soit $s' \approx t'\,[\![c_2']\!]$ l'égalité $s \approx t\,[\![c_2]\!]$ ou une de ses extensions. Soit σ_1 une solution close de c_1, et σ_2 une solution clos de c_2'. Supposons que $\sigma_1(u) = \sigma_1(u)[s'']_p$, où $s'' =_E \sigma_2(s')$, p est une position de variable de u mais $u\,[\![c_1]\!] \not\succ_E \sigma_1(u)[t']_p$. Alors, il existe des positions $p_1, \cdots, p_n$ dans $\sigma_1(u \approx v)$, et des égalités closes $s_1 \approx t_1, \cdots, s_n \approx t_n$, et des positions $q_1, \cdots, q_m$ dans c_1, et des égalités $s_1' \approx t_1'\,[\![c_2']\!], \cdots, s_m' \approx t_m'\,[\![c_2']\!]$ telles que :*

1. *$s_i \approx t_i$ soit une instance close de $s \approx t\,[\![c_2]\!]$ ou une instance close d'une extension de $s \approx t\,[\![c_2]\!]$ pour $i \in \{1, \ldots, n\}$.*
2. *$\sigma_1(u \approx v)_{|p_i} =_E s_i$ pour tout i.*
3. *$s_i' \approx t_i'\,[\![c_2']\!]$ soit $s \approx t\,[\![c_2]\!]$ ou une extension de $s \approx t\,[\![c_2]\!]$.*
4. *$\sigma_1(u \approx v)[t_1]_{p_1} \cdots [t_n]_{p_n}$ soit une instance d'un élément de $(AllDedStrat)TransfoCP(u \approx v\,[\![c_1]\!], \top, s \approx t\,[\![c_2]\!], u \approx v\,[\![c_1]\!])$*
5. *$\sigma_1(u \approx v)[t_1]_{p_1} \cdots [t_n]_{p_n}$ soit une instance d'une élément de $(AllSimpStrat)TransfoSimp(u \approx v\,[\![c_1]\!], \top, s \approx t\,[\![c_2]\!], u \approx v\,[\![c_1]\!])$ et $u \approx v\,[\![c_1]\!]$ soit redondante en présence de toutes les instances de tous les éléments de $(AllSimpStrat)TransfoSimp(u \approx v\,[\![c_1]\!], \top, s \approx t\,[\![c_2]\!], u \approx v\,[\![c_1]\!])$ et de $s \approx t\,[\![c_2]\!]$.*

Nous établissons maintenant la complétude de $BCICS_m$.

Conjecture 3 *Soit E une théorie équationnelle pour laquelle la complétion modulo E est complète. Soit E_∞ un ensemble d'égalités saturé par les règles d'inférence de $BCICS_m$. L'ensemble des instances closes de E_∞ est saturé par les règles de la complétion modulo E, SC_m, et est donc convergent.*

5.6 Conclusion

Nous sommes partis des constatations que l'utilisation des contraintes et de stratégies de contraction pose des problèmes de compatibilité par rapport à la complétude, et que les stratégies de simplification complètes actuelles pour la complétion basique (Simplification Basique et Simplification E-cycle) ne sont pas applicables efficacement à la complétion basique modulo, car elles nécessitent de résoudre les contraintes, pour proposer un nouveau système d'inférence pour la complétion basique modulo, le système d'inférence $BCICS_m$. Le système d'inférence $BCICS_m$ s'appuie sur des idées nouvelles et novatrices. Devant l'incompatibilité entre les contraintes et les stratégies de contraction par rapport à la complétude et le manque d'idées pour résoudre ce problème, la voie que nous avons choisie d'explorer est celle d'autoriser les inférences dans les contraintes. Notre approche prend donc le contre-pied de toutes les approches actuelles de la recherche de preuve et de la déduction avec contraintes, car les inférences sont permises dans les formules et dans les contraintes. Autoriser les inférence dans les contraintes et utiliser des contraintes de contraintes impliquent qu'il n'est plus nécessaire de résoudre les contraintes pour réaliser de la simplification, seule la validité de problèmes de filtrage contraints est testée. Ainsi, nous nous sommes placés à un haut niveau en définissant et utilisant des contraintes de contraintes. Contraindre des contraintes nous a paru naturel car les contraintes sont des formules.

Nous avons prouvé la correction de $BCICS_m$ en introduisant la notion d'égalités contraintes correctement par rapport à l'ensemble d'égalité initial à compléter E_0 et la théorie équationnelle

considérée E. Intuitivement, une égalité est contrainte correctement par rapport à E_0 modulo E si elle est impliquée par E_0 et E. Un autre point intéressant concernant $BCICS_m$ est que la seule condition sur l'ensemble initial d'égalités à compléter E_0 est que les égalités de cette ensemble doivent être contraintes correctement par rapport à E et E_0. Cette condition est vérifiée si l'on part d'un ensemble non contraint, mais certains ensembles contraints sont également autorisés (ceux qui sont contraints correctement). Cette condition est intéressante, car certains systèmes d'inférence de la complétion basique requiert que l'ensemble initial des égalités ne soit pas contraint (Nieuwenhuis et Rubio, 1992b).

Nous avons comparé BC_m et $BCICS_m$ dans le cas de la théorie associative, qui est infinitaire. Nous avons ainsi établi que $BCICS_m$ permet de contrôler l'explosion de l'espace de recherche de manière plus efficace que BC_m. L'exemple traité concerne la complétion de l'ensemble $E_0 = \{g(f(a,x),f(x,a)) \approx h(x),\ g(z,z) \approx h(z),\ h(t_n) \approx b\}$ où t_i est la suite définie par $t_0 = a$ et $t_{i+1} = f(a,t_i)$ pour tout $i \geq 0$ dans la théorie associative.

$BCICS_m$ est un système d'inférence complet. Nous avons laissé la preuve en conjecture pour lors, la preuve est faite mais elle n'a pas été retranscrite dans ce manuscrit pour l'instant.

Nous envisageons d'implanter une procédure de complétion basique modulo basée sur $BCICS_m$ dans le cas de la théorie équationnelle associative en ELAN. Notre choix se porte d'une part sur la théorie associative, car notre approche permet de manipuler les théories infinitaires, ce qui est tout à fait nouveau et qu'il existe un algorithme d'unification modulo l'associativité non complet proposé par G. Plotkin (Plotkin, 1972) facile à implanter en ELAN et d'autre part sur ELAN, car l'implantation des règles d'inférence notamment pour les règles d'inférence dans les contraintes a déjà été pensée en ELAN.

Conclusion

Dans ce document, nous avons présenté nos trois contributions dans le domaine de la preuve automatique de théorèmes et de la déduction. Notre recherche a été menée au niveau théorique, avec le développement d'un algorithme distribué pour la complétion close des graphe SOUR, le développement d'une stratégie de simplification, la *Simplification E-cycle*, complète en combinaison avec la *Complétion Basique* et la mise au point d'un système d'inférence complet pour la *Complétion Basique modulo*, autorisant la simplification et ne nécessitant pas de résoudre les contraintes, et au niveau pratique avec le développement de deux systèmes, *CWD* qui implante notre algorithme distribué de complétion close des graphes SOUR et *ECC* qui implante des procédures de *Complétion Basique* avec diverses stratégies de simplification.

Parallélisme

Nous avons présenté un algorithme distribué pour la complétion close des graphes SOUR. Dans notre approche, chaque noeud du graphe SOUR initial est un processus représentant un terme. Notre approche est donc de grain fin. Les arcs sont des canaux de communication. La complétion est réalisée par coopération entre les processus par passage de message (*message passing*).

Nous disposons grâce à notre approche et de façon indépendante de différents algorithmes distribués : un algorithme qui réalise la complétion des graphes SOUR, un algorithme basé sur l'utilisation des graphes SOUR permettant de déterminer si deux termes sont unifiables (égaux), un algorithme basé sur l'utilisation des graphes SOUR et LPO permettant de déterminer si un terme est plus grand, plus petit ou égal à un autre terme et un algorithme pour détecter la terminaison d'un programme concurrent où les processus ne travaillent que par réception de messages et de manière asynchrone. Pour développer notre algorithme distribué de calcul de l'orientation, nous avons proposé une implantation graphique de l'ordre LPO qui est d'un grand intérêt théorique dans le sens où dans leurs travaux sur la terminaison des systèmes de réécriture, T. Genet et I. Gnaedig ont utilisé une implantation similaire (Genet et Gnaedig, 1997). On pourra également noté son intérêt pédagogique. Exprimer les définitions sous forme de graphe facilite la compréhension et d'ailleurs dans ce travail de thèse, les graphes sont un élément fondamental.

Notre modèle de calcul a diverses bonnes propriétés. Les processus fonctionnent de manière complètement locale, ils utilisent l'information contenue dans leur état, il n'y a pas mémoire globale ou de contrôle global. Nous pensons que dans le futur c'est dans ce sens qu'il faut aller, car ce genre de modèle rend la parallélisation des stratégies de contraction plus abordable et les stratégies de contraction sont cruciales pour le contrôle de l'expansion de l'espace de recherche. De plus, il n'y a pas de travail redondant ce qui est une caractéristique très importante de notre modèle. Une application parallèle où du travail redondant est réalisé ne peut pas donner de résultats probants. Un troisième point est que nous utilisons une stratégie de contraction dont

l'implantation est très facile à réaliser. En effet, une simplification implique la suppression d'un arc et seuls deux processus ont à être au courant de la simplification et ils n'ont qu'à modifier leur état.

Nous pensons que cette approche est fondamentalement nouvelle et prometteuse, car il n'avait jamais été imaginé de combiner le grain fin, des opérations complètement locales, et une stratégie de contraction compatible avec le grain fin de parallélisme et la localité des opérations.

Le fil conducteur dans cette thèse est l'élément « stratégie de simplification ». Dans le cas d'applications parallèles et concurrentes, il convient de se poser des questions quant à la compatibilité des stratégies de contraction et du parallélisme et de la concurrence. Nous aimerions étudier quels critères peuvent amener la compatibilité entre les stratégies de contraction et le parallélisme et la concurrence. Nous avons donné quelques éléments de réponse avec la mise au point de notre modèle de calcul basé sur l'utilisation des graphes SOUR clos mais nous pensons que c'est un point à éclaircir en général pour pouvoir développer des prouveurs de théorèmes parallèles efficaces.

Simplification et contraintes

Nous avons atteint le but que nous nous étions fixé au début de cette thèse, à savoir proposer une stratégie de simplification complète en combinaison avec la complétion basique faisant plus de simplifications que la simplification basique. La simplification basique était jusqu'à ce jour la seule stratégie de simplification complète en combinaison avec la complétion basique.

La simplification E-cycle est liée à la construction d'un graphe de dépendance pendant le processus de complétion. Le graphe de dépendance schématisent les dépendances entre les égalités.

L'utilisation simultanée de contraintes et de stratégies de contraction pose des problèmes de compatibilité vis à vis de la complétude. Le graphe de dépendance capture les dépendances entre les égalités et nous permet de raisonner pour savoir quelles simplifications détruisent la complétude et quelles simplifications ne la détruisent pas.

Nous avons prouvé la complétude de la complétion basique avec simplification E-cycle en utilisant la technique de preuve basée sur la construction d'un modèle d'égalités de Herbrand pour l'ensemble des égalités saturé obtenu lors de la complétion basique avec un ordre construit à partir du graphe de dépendance. Nous avons utilisé une plate-forme abstraite utilisant des graphes de dépendance clos pour aboutir à cette preuve. Nous pensons que cette plate-forme pourrait être utile pour analyser la complétude de différentes stratégies de simplification en combinaison avec la complétion basique et pour déterminer la « ligne » qui sépare les stratégies de simplification complètes des stratégies de simplification incomplètes et que que la simplification E-cycle est près de la « ligne ». Nous n'avons pas étudié plus en détail cette proposition mais cette voie d'exploration est l'une de nos perspectives. Indépendamment de cette idée, nous avons montré l'incomplétude de diverses stratégies de simplification. Nous avons répondu par la négative au problème ouvert numéro un de R. Nieuwenhuis (Nieuwenhuis, 1999), qui conjecturait que la stratégie S_2 (*forward*) était complète.

Nous pensons que la simplification E-cycle outre son pouvoir théorique a l'intérêt d'être facilement compréhensible car elle est basée sur une structure de graphe, ce qui fait d'elle également une stratégie facilement intégrable dans les prouveurs existants. Il n'y a qu'à ajouter la structure de graphe, implanter les opérations pour la mise à jour du graphe et intégrer cette mise à jour aux règles d'inférence. La complétion basique avec simplification E-cycle a été validée par l'implantation *ECC*, mais nous voudrions passer à l'étape supérieure en l'intégrant à un système

existant dont l'efficacité a déjà été démontrée. Mais pour que l'éventail des systèmes possibles soit assez large, il faudrait étendre l'utilisation de la simplification E-cycle à la paramodulation basique.

Simplification, contraintes et modulo

Devant l'incompatibilité entre les contraintes et les stratégies de contraction par rapport à la complétude qui implique de résoudre les contraintes, et le manque d'idées pour résoudre ce problème dans le cas de la complétion modulo, la voie que nous avons choisie d'explorer est celle d'autoriser les inférences dans les contraintes. Notre approche prend donc le contre-pied de toutes les approches actuelles de la recherche de preuve et de la déduction, car les inférences sont permises dans les formules et dans les contraintes. Nous avons proposé un système d'inférence $BCICS_m$ correct et complet, qui ne nécessite pas de résoudre les contraintes et qui ne nécessite pas de partir d'un ensemble non contraint comme c'est requis par de nombreux systèmes d'inférence (Nieuwenhuis et Rubio, 1992b).

Le travail sur la complétion modulo et les résultats obtenus méritent d'être validés par une implantation. Nous envisageons donc d'implanter une procédure de complétion basique modulo basée sur $BCICS_m$ dans le cas de la théorie équationnelle associative en ELAN. Notre choix se porte sur la théorie associative pour deux raisons. Nous avons comparé BC_m et $BCICS_m$ dans le cas de la théorie associative. Nous avons ainsi établi que $BCICS_m$ permet de contrôler l'explosion de l'espace de recherche de manière plus efficace que BC_m. La manipulation de théories infinitaires est donc un apport important de notre approche. De plus, il existe un algorithme d'unification modulo l'associativité non complet proposé par G. Plotkin (Plotkin, 1972), qui n'est pas basé sur la résolution d'équations diophantiennes, facile à implanter en ELAN. Notre choix du langage de programmation se porte sur ELAN car l'implantation des règles d'inférence notamment pour les règles d'inférence dans les contraintes a déjà été pensée en ELAN.

Bibliographie

Abdulrab, H. et Pécuchet, J.-P. (1988). Solving systems of linear diophantine equations and associative unification, *Technical report*, LIR, Faculté des Sciences, BP 118, 76134 Mont-Saint-Aignan cedex, France.

Abdulrab, H. et Pécuchet, J.-P. (1990). Solving word equations, *Journal of Symbolic Computation* 8(5) : 499–522.

Alouini, E. (1997). *Étude et mise en oeuvre de la réécriture conditionnelle concurrente sur des machines parallèles à mémoire distribuée*, PhD thesis, Université Henri poincaré-Nancy I. *file ://ftp.loria.fr/pub/loria/protheo/THESIS_1997/alouini.ps.gz

Anantharaman, S. et Bousdira, W. (Fall-1992). Reveal : A users guide, *Rapport de Recherche*, Laboratoire d'Informatique Fondamentale d'Orléans.

Anantharaman, S. et Hsiang, J. (1990). An automated proof of the Moufang identities in alternative rings, *Journal of Automated Reasoning* **6** : 79–109.

Anantharaman, S., Hsiang, J. et Mzali, J. (1989). Sbreve2 : A term rewriting laboratory with (AC-)unfailing completion, *in* N. Dershowitz (ed.), *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, Vol. 355 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 533–537.

Anantharaman, S. et Mzali, J. (1989). Unfailing completion modulo a set of equations, *Research Report 470*, LRI-Orsay (Fr.).

Astrachan, O. et Loveland, D. (1991). Meteors : high performance theorem provers using model elimination, *in* R. Boyer (ed.), *Automated Reasoning - Essays in Honor of Woody Bledsoe*, pp. 31–60.

Avenhaus, J., Denzinger, J. et Fuchs, M. (1995). DISCOUNT : A system for distributed equational deduction, *in* J. Hsiang (ed.), *Rewriting Techniques and Applications, 6th International Conference , RTA-95*, LNCS 914, Springer-Verlag, Kaiserslautern, Germany, pp. 397–402.

Baader, F. et Siekmann, J. (1993). Unification theory, *in* D. M. Gabbay, C. J. Hogger et J. A. Robinson (eds), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press, Oxford, UK.

Bachmair, L. (1987). *Proof methods for equational theories*, PhD thesis, University of Illinois, Urbana-Champaign, (Ill., USA). Revised version, August 1988.

Bachmair, L. et Dershowitz, N. (1994). Equational inference, canonical proofs, and proof orderings, *Journal of Association for Computing Machinery* 41(2) : 236–276.

Bachmair, L., Dershowitz, N. et Plaisted, D. (1989). Completion without failure, *in* H. Aït-Kaci et M. Nivat (eds), *Resolution of Equations in Algebraic Structures, Volume 2 : Rewriting Techniques*, Academic Press inc., pp. 1–30.

Bachmair, L. et Ganzinger, H. (1993). Associative-Commutative Superposition, *Technical report MPI-I-93-267*, Max Planck Institut für Informatik, Saarbrücken.

Bachmair, L. et Ganzinger, H. (1994). Rewrite-based equational theorem proving with selection and simplification, *Journal of Logic and Computation* 4(3) : 1–31.

Bachmair, L. et Ganzinger, H. (1998). Equational reasoning, *in* W. Bibel et P. Schmitt (eds), *Automated Deduction - A Basis for Applications*, Vol. 1, Kluwer Academic Publishers.

Bachmair, L., Ganzinger, H., Lynch, C. et Snyder, W. (1992). Basic paramodulation and superposition, *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs (N.Y., USA)*, pp. 462–476.

Bachmair, L., Ganzinger, H., Lynch, C. et Snyder, W. (1995). Basic paramodulation, *Information and Computation* **121**(2) : 172–192.

Bachmair, L. et Plaisted, D. (1985). Termination orderings for associative-commutative rewriting systems, *Journal of Symbolic Computation* **1** : 329–349.

Baumagarter, P. et Bruening, S. (1994). Protein : a prover with theory extension interface, *in* A. Bundy (ed.), *Proceedings of the CADE'92*, Vol. 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 769–773.

Beguelin, A., Dongarra, J. et Geist, G. A. (1994). Parallel virtual machine user's guide and reference manual, *Technical report*, Oak Ridge National Laboratory.

Benanav, D., Kapur, D. et Narendran, P. (1987). Complexity of matching problems, *Journal of Symbolic Computation* **3**(1 & 2) : 203–216.

Birkhoff, G. (1935). On the structure of abstract algebras, *Proceedings Cambridge Phil. Soc.* **31** : 433–454.

Bofill, M., Godoy, G., Nieuwenhuis, R. et Rubio, A. (1999). Paramodulation with non-monotonic orderings, *Proceedings of the LICS'99*, Lecture Notes in Computer Science, Springer-Verlag.

Bonacina, M.-P. (1992). *Distributed Automated Deduction*, PhD thesis, State University of New York at Stony Brook.

Bonacina, M.-P. (1996). On the reconstruction of proofs in distributed theorem proving : a modified Clause-Diffusion method, *Journal of Symbolic Computation* **21**(4–6) : 507–522.

Bonacina, M.-P. (1997a). The Clause-Diffusion theorem prover Peers-mcd, *in* W. W. McCune (ed.), *Proceedings of the Fourteenth International Conference on Automated Deduction (CADE-14)*, Vol. 1249 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 53–56.

Bonacina, M.-P. (1997b). Experiments with subdivision of search in distributed theorem proving, *in* M. Hitz et E. Kaltofen (eds), *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO97)*, ACM Press, pp. 88–100.

Bonacina, M.-P. (1999a). A taxonomy of theorem proving strategies, *in* W. M. Veloso Manuela et F. Michael (eds), *"Artificial Intelligence Today"*, Vol. 1500 of *Lecture Notes in Computer Science*, Springer-Verlag. To appear.

Bonacina, M.-P. (1999b). Ten years of parallel theorem proving : a perspective, *Proceedings of FLoC'99 Workshop on Strategies in automated deduction, Strategies'99.* Invited talk.

Bonacina, M.-P. et Hsiang, J. (1994). Parallelization of deduction strategies : an analytical study, *Journal of Automated Reasoning* **13** : 1–33.

Bonacina, M.-P. et Hsiang, J. (1995a). The clause-diffusion methodology for distributed deduction, *Fundamenta Informaticae* **24** : 177–207.

Bonacina, M.-P. et Hsiang, J. (1995b). Distributed deduction by clause-diffusion : Distributed contraction and the aquarius prover, *Journal of Symbolic Computation* **19** : 245–267.

Bonacina, M.-P. et McCune, W. (1994). Distributed theorem proving by Peers, *in* A. Bundy (ed.), *Proceedings of the Twelfth International Conference on Automated Deduction (CADE-12)*, Vol. 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 841–845.

Borovanský, P. (1995). Implementation of higher-order unification based on calculus of explicit substitutions, *in* M. Bartošek, J. Staudek et J. Wiedermann (eds), *Proceedings of the SOFSEM'95 : Theory and Practice of Informatics*, Vol. 1012 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 363–368.

Borovanský, P. (1998). *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France. also TR CRIN 98-T-326.
*http ://www.loria.fr/ borovan/these.ps

Borovanský, P., Cirstea, H., Dubois, H., Kirchner, C., Kirchner, H., Moreau, P.-E., Ringeissen, C. et Vittek, M. (1998). ELAN *V 3.3 User Manual*, third edn, LORIA, Nancy (France).

Borovanský, P., Kirchner, C. et Kirchner, H. (1996). Controlling rewriting by rewriting, *in* J. Meseguer (ed.), *Proceedings of the first international workshop on rewriting logic*, Vol. 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California).

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E. et Ringeissen, C. (1998). An overview of ELAN, *in* C. Kirchner et H. Kirchner (eds), *Proceedings of the second International Workshop on Rewriting Logic and Applications*, Vol. 15, `http ://www.elsevier.nl/locate/entcs/volume16.html`, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France).

Bose, S., Clarke, E. M., Long, D. E. et Michaylov, S. (1992). PARTHENON : A parallel theorem prover for non-Horn clauses, *Journal of Automated Reasoning* **8**(2) : 153–181.

Brand, D. (1975). Proving theorems with the modification method, *SIAM Journal of Computing* **4** : 412–430.

Bündgen, R., Göbel, M. et Küchlin, W. (1994). A fine-grained parallel completion procedure, *Proceedings of the ACM-ISSAC'94 International Symposium on Symbolic and Algebraic Computation, Oxford, England*, ACM Press, pp. 269–277.

Bündgen, R., Göbel, M. et Küchlin, W. (1995). Parallel redux → paredux, *in* J. Hsiang (ed.), *Proceedings of the RTA'95*, Vol. 914 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 408–413.

Buntine, W. et Bürckert, H.-J. (1989). On solving equations and disequations, *Technical Report SR-89-03*, Universität Kaiserslautern.

Bürckert, H.-J. (1989). Matching — A special case of unification?, *Journal of Symbolic Computation* **8**(5) : 523–536.

Bürckert, H.-J. (1990). A resolution principle for clauses with constraints, *Research report RR-90-02*, DFKI GmBH, Kaiserslautern.

Caferra, R. et Zabel, N. (1990). A method for simultaneous search for refutations and models using equational problems. Grenoble, France.

Castro, C. (1998a). COLETTE, Prototyping CSP Solvers Using a Rule-Based Language, *Proceedings of The Fourth International Conference on Artificial Intelligence and Symbolic Computation, Theory, Implementations and Applications, AISC'98*, Vol. 1476 of *Lecture Notes in Artificial Intelligence*, Plattsburgh, NY, USA.
*file ://ftp.loria.fr/pub/loria/protheo/COMMUNICATIONS_1998/Castro-AISC98.ps.gz

Castro, C. (1998b). *Une approche déductive de la résolution de problèmes de satisfaction de contraintes*, PhD thesis, Université Henri Poincaré – Nancy 1.

Chang, C. L. et Lee, R. C. (1973). *Symbolic Logic and Mechanical Theorem Proving*, Academic Press inc., New York (USA).

Cirstea, H. et Kirchner, C. (1998). Using rewriting and strategies for describing the B predicate prover, *Proceedings of the Workshop on Strategies in Automated Deduction, CADE-15*, Lindau, Germany.

Clavel, M., Duràn, F., Eker, S., Lincoln, P. et Meseguer, J. (1998). An Introduction to Maude (Beta Version), *Technical report*, SRI International, Computer Science Laboratory, Menlo Park, (CA, USA).
*ftp ://ftp.csl.sri.com/pub/rewriting/beta/maude-beta-doc.ps

Comon, H. (1990). Solving inequations in term algebras, *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pp. 62–69.

Comon, H. (1992). Completion of rewrite systems with membership constraints, *in* W. Kuich (ed.), *Proceedings of ICALP 92*, Vol. 623 of *Lecture Notes in Computer Science*, Springer-Verlag.

Conry, E., MacIntosh, D. et Meyer, R. (1990). Dares : A distributed automated reasonong system, *Proceedings of the 11th Conference of the American Association for Artificial Intelligence*, pp. 78–85.

Dauchet, M. (1989). Simulation of Turing machines by a left-linear rewrite rule, *in* N. Dershowitz (ed.), *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, Vol. 355 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 109–120.

Dauchet, M. (1992). Simulation of Turing machines by a regular rule, *Theoretical Computer Science* **103** : 409–420.

Denzinger, J. (1993). *Teamwork : A method to design distributed knowledge based theorem provers*, PhD thesis, University of Kaiserlautern.

Denzinger, J. (1994). Goal oriented equational theorem proving using teamwork, *Proceedings of KI'94*, Vol. 861, Lecture Notes in Artificial Intelligence, pp. 343–354.

Denzinger, J. (1995). Knowledge-based distributed search using teamwork, *Proceedings of IC-MAS'95*, AAAI-Press, pp. 81–88.

Denzinger, J. et Dahn, I. (1998). Cooperating theorem provers, *in* W. Bibel et P. Schmitt (eds), *Automated Deduction - A Basis for Applications*, Vol. 2, Kluwer Academic Publishers.

Dershowitz, N. (1982). Orderings for term-rewriting systems, *Theoretical Computer Science* **17** : 279–301.

Dijkstra, E., Feijen, W. et Van Gasteren, A. (1983). Derivation of a termination detection algorithm for distributed computation, *Information Processing Letters* **16** : 217–219.

Domenjoud, E. (1992). A technical note on AC-unification. the number of minimal unifiers of the equation $\alpha x_1 + \cdots + \alpha x_p \doteq_{AC} \beta y_1 + \cdots + \beta y_q$, *Journal of Automated Reasoning* **8** : 39–44. Also as research report CRIN 89-R-2.

Dowek, G., Hardin, T. et Kirchner, C. (1998). Theorem proving modulo, *Rapport de Recherche 3400*, Institut National de Recherche en Informatique et en Automatique. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

Dwork, C., Kanellakis, P. et Mitchell, J. C. (1984). On the sequential nature of unification, *Journal of Logic Programming* 1(1) : 35–50.

Eker, S. (1993). Some ideas on elementary AC matching, note technique.

Fages, F. (1983). *Formes Canoniques dans les Algèbres Booléennes et Application à la Démonstration Automatique en Logique du Premier Ordre*, Thèse de Doctorat de Troisième Cycle, Université de Paris 7 (France).

Fuchs, D. et Denzinger, J. (1998). Knowledge-based cooperation between theorem provers by

techs, *in* W. Bibel et P. Schmitt (eds), *Automated Deduction - A Basis for Applications*, Vol. 2, Kluwer Academic Publishers.

Gallier, J., Narendran, P., Plaisted, D., Raatz, S. et Snyder, W. (1993). Finding canonical rewriting systems equivalent to a finite set of ground equations in polynomial time, *Journal of Association for Computing Machinery* **40**(1) : 1–16.

Genet, T. (1998). Decidable approximations of sets of descendants and sets of normal forms, *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, Vol. 1379 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 151–165.
*file ://ftp.loria.fr/pub/loria/protheo/COMMUNICATIONS_1998/Genet-RTA98.ps.gz

Genet, T. et Gnaedig, I. (1997). Termination proofs using gpo ordering constraints, *in* M. Dauchet (ed.), *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, Vol. 1214 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 249–260.
*file ://ftp.loria.fr/pub/loria/protheo/COMMUNICATIONS_1997/GenetGnaedig-CAAP97.ps.gz

Gropp, W., Lusk, E. et Skjellum, A. (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface.*, MIT Press, Cambridge, Masschussets.

Hermann, M. (1989). Crossed term rewriting systems, *Research report 89-R-003*, Centre de Recherche en Informatique de Nancy.

Hermann, M. (1998). Constrained reachability is np-complete. http ://www.loria.fr/~hermann/publications.html#notes.

Herold, A. (1985). Combination of unification algorithm, *Memo SEKI 85-VIII-KL*, Universität Kaiserslautern.

Hintermeier, C. (1995). *Déduction avec sortes ordonnées et égalités*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1.

Hsiang, J. et Rusinowitch, M. (1986). A new method for establishing refutational completeness in theorem proving, *in* J. Siekmann (ed.), *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, Vol. 230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 141–152.

Hsiang, J. et Rusinowitch, M. (1987). On word problems in equational theories, *in* T. Ottman (ed.), *Proceedings of the 14th ICALP*, Vol. 267 of *LNCS*, Springer Verlag, pp. 54–71.

Hsiang, J. et Rusinowitch, M. (1991). Proving refutational completeness of theorem proving strategies : The transfinite semantic tree method, *Journal of the ACM* **38**(3) : 559–587.

Huet, G. (1976). *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*, Thèse de Doctorat d'Etat, Université de Paris 7 (France).

Huet, G. (1980). Confluent reductions : Abstract properties and applications to term rewriting systems, *Journal of the ACM* **27**(4) : 797–821. Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.

Huet, G. et Lankford, D. S. (1978). On the uniform halting problem for term rewriting systems, *Technical Report 283*, Laboria, France.

Hullot, J.-M. (1979). Associative-commutative pattern matching, *Proceedings 9th International Joint Conference on Artificial Intelligence.*

Hullot, J.-M. (1980). Canonical forms and unification, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, pp. 318–334.

Jaffar, J. (1990). Minimal and complete word unification, *Journal of the ACM* .

Jouannaud, J.-P. et Kirchner, C. (1991). Solving equations in abstract algebras : a rule-based survey of unification, *in* J.-L. Lassez et G. Plotkin (eds), *Computational Logic. Essays in honor of Alan Robinson*, The MIT press, Cambridge (MA, USA), chapter 8, pp. 257–321.

Jouannaud, J.-P. et Kirchner, H. (1986). Completion of a set of rules modulo a set of equations, *SIAM Journal of Computing* **15**(4) : 1155–1194. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

Jouannaud, J.-P. et Muñoz, M. (1984). Termination of a set of rules modulo a set of equations, *Proceedings 7th International Conference on Automated Deduction, Napa Valley (Calif., USA)*, Vol. 170 of *Lecture Notes in Computer Science*, Springer-Verlag.

Jouannaud, J.-P. et Okada, M. (1991). Satisfiability of systems of ordinal notations with the subterm property is decidable, *in* J. Leach Albert, B. Monien et M. Rodríguez Artalejo (eds), *Proceedings 18th ICALP Conference, Madrid (Spain)*, Vol. 510 of *Lecture Notes in Computer Science*, Springer-Verlag.

Kamin, S. et Lévy, J.-J. (1982). Attempts for generalizing the recursive path ordering, *Inria, Rocquencourt* .

Kapur, D. et Narendran, P. (1985). A finite Thue system with decidable word problem and without equivalent finite canonical system, *Theoretical Computer Science* **35** : 337–344.

Kapur, D. et Narendran, P. (1986). NP-completeness of the set unification and matching problems, *in* J. Siekmann (ed.), *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, Vol. 230 of *Lecture Notes in Computer Science*, Springer-Verlag.

Kapur, D. et Narendran, P. (1992). Double-exponential complexity of computing a complete set of AC-unifiers, *Proceedings of LICS'92*, Santa-Cruz (California, USA).

Kapur, D., Narendran, P. et Zhang, H. (1986). Proof by induction using test sets, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, Vol. 230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 99–117.

Kapur, D., Sivakumar, G. et Zhang, H. (1986). RRL : A rewrite rule laboratory, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, Vol. 230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 692–693.

Kapur, D. et Zhang, H. (1991). A case study of the completion procedure : ring commutativity problems, *in* J.-L. Lassez et G. Plotkin (eds), *Computational Logic. Essays in honor of Alan Robinson*, The MIT press, Cambridge (MA, USA), chapter 10, pp. 360–394.

Kirchner, C. (1986). Computing unification algorithms, *Proceedings 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pp. 206–216.

Kirchner, C., Kirchner, H. et Rusinowitch, M. (1990). Deduction with symbolic constraints, *Revue d'Intelligence Artificielle* 4(3) : 9–52. Special issue on Automatic Deduction.

Kirchner, C., Lynch, C. et Scharff, C. (1996a). A fine-grained concurrent completion procedure, *in* H. Ganzinger (ed.), *Proceedings of RTA'96*, Vol. 1103 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 3–17.

Kirchner, C., Lynch, C. et Scharff, C. (1996b). A fine-grained concurrent completion procedure, *Technical Report 2990*, INRIA.

Kirchner, C. et Ringeissen, C. (1998). Rule-Based Constraint Programming, *Fundamenta Informaticae* **34**(3) : 225–262.

Kirchner, C. et Viry, P. (1992). Implementing parallel rewriting, *in* B. Fronhöfer et G. Wrightson (eds), *Parallelization in Inference Systems*, Vol. 590 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 123–138.

Kirchner, H. (1985). *Preuves par complétion dans les variétés d'algèbres*, Thèse de Doctorat d'Etat, Université Henri Poincaré – Nancy 1.

Kirchner, H. et Moreau, P.-E. (1995). Prototyping completion with constraints using computational systems, *in* J. Hsiang (ed.), *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, Vol. 914 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 438–443.

Knuth, D. E. et Bendix, P. B. (1970). Simple word problems in universal algebras, *in* J. Leech (ed.), *Computational Problems in Abstract Algebra*, Pergamon Press, Oxford, pp. 263–297.

Kolata, G. (1996). With major math proof, brute computers show flash of reasoning power, New York Times. page C1.

Kozen, D. (1977). *Complexity of Finitely Presented Algebras*, PhD thesis, Cornell University.

Lankford, D. S. (1977). Some approaches to equality for computational logic : A survey and assessment, *Memo ATP-36*, Automatic Theorem Proving Project, University of Texas, Austin (Texas, USA).

Lassez, J.-L., Maher, M. J. et Marriot, K. (1986). Unification revisited, *Technical report*, IBM Thomas J. Watson Research Center.

Lescanne, P. (1986). Divergence of the Knuth-Bendix completion procedure and termination orderings, *Bulletin of European Association for Theoretical Computer Science* **30** : 80–83.

Letz, R., Mayr, K. et Goller, C. (1992). Setheo : a high performance theorem prover, *Journal of Automated Reasoning* **8**(2) : 183–212.

Letz, R. et Schumann, J. (1990). Partheo : A high-performance parallel theorem prover, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, Vol. 446 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 40–56.

Lindenstrauss, N. (1989). A parallel implementation of rewriting and narrowing, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, Vol. 35 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 569–573.

Livesey, M. et Siekmann, J. (1975). Termination and decidability results for string unification, *Technical report memo CSM-12*, University of Essex.

Livesey, M. et Siekmann, J. (1976). Unification of bags and sets, *Technical report*, Institut für Informatik I, Universität Karlsruhe.

Lusk, E. et McCune, W. (1992). Experiments with roo : a parallel automated deduction system, *in* B. Fronhöfer et G. Wrightson (eds), *Parallelization in Inference Systems*, Vol. 590 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 139–162.

Lynch, C. (1995). Paramodulation without duplication, *in* D. Kozen (ed.), *Proceedings 10th IEEE Symposium on Logic in Computer Science, San Diego (Ca., USA)*, IEEE, San Diego, pp. 167–177.

Lynch, C. (1997). Goal directed completion using sour graphs, *in* H. Comon (ed.), *Proceedings of the RTA'97*, Vol. 1232 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 8–22.

Lynch, C. et Scharff, C. (1998). Basic completion with e-cycle simplification, *Artificial Intelligence And Symbolic Computation*, Vol. 1476 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Plattsburgh (USA), pp. 209–221.

Lynch, C. et Scharff, C. (1999a). Basic completion modulo with simplification.

Lynch, C. et Scharff, C. (1999b). Basic completion with e-cycle simplification, *To appear in Fundamenta Informaticae* .

Lynch, C. et Snyder, W. (1995). Redundancy criteria for constrained completion, *Theoretical Computer Science*, Vol. 142, pp. 141–177.

Lynch, C. et Strogova, P. (1996). Patch graphs : an efficient data structure for completion of finitely presented groups, *in* J. Pfalzgraf (ed.), *Proc. Third International Conference on Artificial Intelligence and Symbolic Mathematical Computation*, Lecture Notes in Computer Science, Springer-Verlag.

Lynch, C. et Strogova, P. (1998). Sour graphs for efficient completion, *Discrete Mathematics and Theoritical Computer Science* **2** : 1–25.

Makanin, G. S. (1977). The problem of solvability of equations in a free semigroup, *Math. USSR Sbornik* **32**(2) : 129–198.

Martelli, A. et Montanari, U. (1982). An efficient unification algorithm, *ACM Transactions on Programming Languages and Systems* 4(2) : 258–282.

McCune, W. (1994). Otter 3.0 : Reference manual and guide, *Technical Report 6*, Argonne National Laboratory.

McCune, W. (1997a). 33 basic test problems : A practical evaluation of some paramodulation strategies, *in* R. Veroff (ed.), *Automated Reasoning and its Applications : Essays in Honor of Larry Wos*, MIT Press, pp. 71–114.

McCune, W. (1997b). Solution of the robbins problem, Journal of Automated Reasoning **19**(3) : 263–276.

McCune, W. (1997c). Well-behaved search and the robbins problem, Invited lecture at the Rewriting Techniques and Applications Conference, Sitges (Spain). http ://www.mcs.anl.gov/home/mccune/ar/robbins/.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* **96**(1) : 73–155.

Moreau, P.-E. (1999). *Compilation de règles de réécriture et de stratégies non déterministes*, PhD thesis, Université Henri Poincaré – Nancy 1.

Moreau, P.-E. et Kirchner, H. (1997). Compilation Techniques for Associative-Commutative Normalisation, *in* A. Sellink (ed.), *Second International Workshop on the Theory and Practice of Algebraic Specifications*, Springer-Verlag, Amsterdam. 12 pages.

Moreau, P.-E. et Kirchner, H. (1998). A compiler for rewrite programs in associative-commutative theories, *"Principles of Declarative Programming"*, number 1490 in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 230–249. Report LORIA 98-R-226.

Naher, S. (1993). Leda manual, *Technical Report MPI-I-93-109*, mpi.

Narendran, P. et Rusinowitch, M. (1996). Any ground associative-commutative theory has a finite canonical system, *Journal of Automated Reasoning* 17(1) : 131–143.

Nieuwenhuis, R. (1993). Simple lpo constraint solving methods, *Information Processing Letters* **47**(2).

Nieuwenhuis, R. (1999). Rewrite-based deduction and symbolic constraints, *in* H. Ganzinger (ed.), *Proceedings of the CADE'99*, Vol. 1632 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 302–312.

Nieuwenhuis, R. et Rivero, J. (1999). Solved forms for path ordering constraints, *in* P. Narendran et M. Rusinowitch (eds), *Proceedings of the RTA'99*, Lecture Notes in Computer Science, Springer-Verlag.

Nieuwenhuis, R. et Rubio, A. (1992a). Basic superposition is complete, *in* B. Krieg-Brückner (ed.), *Proceedings of ESOP'92*, Vol. 582 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 371–389.

Nieuwenhuis, R. et Rubio, A. (1992b). Theorem proving with ordering constrained clauses, *in* D. Kapur (ed.), *Proceedings 11th International Conference on Automated Deduction,*

Saratoga Springs (N.Y., USA), Vol. 607 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 477–491.

Nieuwenhuis, R. et Rubio, A. (1994). AC-superposition with constraints : no AC-unifiers needed, *in* A. Bundy (ed.), *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, Vol. 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 545–559.

Nieuwenhuis, R. et Rubio, A. (1995a). Theorem proving with ordering and equality constrained clauses, *Journal of Symbolic Computation* **19**(4) : 1–21.

Nieuwenhuis, R. et Rubio, A. (1995b). A total ac-compatible ordering based on rpo, *Theoretical Computer Science* **142**(2) : 209–227.

Nieuwenhuis, R. et Rubio, A. (1997). Paramodulation with built-in ac-theories and symbolic constraints, *Journal of Symbolic Computation* **23**(1) : 1–21.

Nivela, P. et Nieuwenhuis, R. (1993). Saturation of first-order (constrained) clauses with the *saturate* system, *in* C. Kirchner (ed.), *Proceedings 5th Conference on Rewriting Techniques and Applications, Montreal (Canada)*, Vol. 690 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 436–440.

Paterson, M. S. et Wegman, M. N. (1978). Linear unification, *Journal of Computer and System Sciences* **16** : 158–167.

Pécuchet, J.-P. (1981). *Equations avec constantes et algorithme de Makanin*, Thèse de doctorat, Université de Rouen (France).

Pelletier, F. (1986). Seventy five problems for testing automatic theorem provers, *Journal of Automated Reasoning* pp. 191–216.

Peterson, G. (1983). A technique for establishing completeness results in theorem proving with equality, *SIAM Journal of Computing* **12**(1) : 82–100.

Peterson, G. (1994). Constrained term-rewriting induction with applications, *Methods of Logic in Computer Science* 1(4) : 379–412.

Peterson, G. et Stickel, M. E. (1981). Complete sets of reductions for some equational theories, *Journal of the ACM* **28** : 233–264.

Plaisted, D. et Sattler-Klein, A. (1996). Proof lengths for equational completion, *Information and Computation* **125**(2) : 154–170.

Plotkin, G. (1972). Building-in equational theories, *Machine Intelligence* **7** : 73–90.

Ringeissen, C. (1997). Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language, *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, Vol. 1232 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 323–326.
*file ://ftp.loria.fr/pub/loria/protheo/COMMUNICATIONS_1997/Ringeissen-RTA97.ps.gz

Robinson, G. A. et Wos, L. (1969). Paramodulation and first-order theorem proving, *in* B. Meltzer et D. Mitchie (eds), *Machine Intelligence 4*, Edinburgh University Press, pp. 135–150.

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

Robinson, J. A. (1971). Computational logic : The unification computation, *Machine Intelligence* **6** : 63–72.

Rubio, A. (1994). *Automated Deduction with Constrained Clauses*, PhD thesis, Universitat Politècnica de Catalunya, Barcelona, Spain.

Rusinowitch, M. (1987). *Démonstration automatique par des techniques de réécriture*, Thèse de Doctorat d'Etat, Université Henri Poincaré – Nancy 1. Also published by InterEditions, Collection Science Informatique, directed by G. Huet, 1989.

Rusinowitch, M. et Vigneron, L. (1991). Automated deduction with associative commutative operators, *in* P. Jorrand et J. Kelemen (eds), *Fundamental of Artificial Intelligence Research*, Vol. 535 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 185–199.

Rusinowitch, M. et Vigneron, L. (1995). Automated Deduction with Associative-Commutative Operators, *Applicable Algebra in Engineering, Communication and Computation* **6**(1) : 23–56.

Siekmann, J. (1975). String-unification, *Internal report memo CSM-7*, University of Essex.

Siekmann, J. (1979). Unification of commutative terms, *Proceedings of the Conference on Symbolic and Algebraic Manipulation*, Vol. 72 of *Lecture Notes in Computer Science*, Springer-Verlag, Marseille (France), pp. 531–545. Also internal report SEKI, 1976.

Smolka, G. (1989). *Logic Programming over Polymorphically Order-Sorted Types*, PhD thesis, FB Informatik, Universität Kaiserslautern, Germany.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D. et Dongarra, J. (1996). *MPI: The Complete Reference*, The MIT Press.

Snyder, W. (1989). Efficient ground completion : An $O(n\ log\ n)$ algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations E, *in* N. Dershowitz (ed.), *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, Vol. 355 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 419–433.

Stickel, M. E. (1976). *Unification Algorithms for Artificial Intelligence Languages*, PhD thesis, Carnegie-Mellon University.

Stickel, M. E. (1981). A unification algorithm for associative-commutative functions, *Journal of the ACM* **28** : 423–434.

Sunderam, V. (1990). Pvm : A framework for parallel distributed computing, *Concurrency :Practice and Experience*, Vol. 2(4), pp. 315–339.

Suttner, C. et Schumann, J. (1993). Parallel automated theorem proving, *Parallel Processing for Artificial Intelligence* .

Tanenbaum, A. (1992). *Modern Operating systems*, Englewood Cliffs, NJ, Prentice Hall.

Tanenbaum, A. (1995). *Distributed Operating systems*, Englewood Cliffs, NJ, Prentice Hall.

Vigneron, L. (1994a). Associative-Commutative Deduction with Constraints, *in* A. Bundy (ed.), *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, Vol. 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 530–544.

Vigneron, L. (1994b). *Déduction automatique avec contraintes symboliques dans les théories équationnelles*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1.

Vigneron, L. (1997). daTac (system demonstration), International Conference TABLEAUX'97 - Analytic Tableaux and Related Methods.

Vigneron, L. (1998). Automated Deduction Techniques for Studying Rough Algebras, *Fundamenta Informaticae* **33**(1) : 85–103.
*file ://ftp.loria.fr/pub/loria/protheo/ARTICLES_1998/loria-98-R-140.ps.gz

Vittek, M. (1994). ELAN *: Un cadre logique pour le prototypage de langages de programmation avec contraintes*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1.

Weidenbach, C., Gaede, B. et Rock, G. (1996). Spass & flotter, version 0.42, *in* M. McRobbie et J. Slaney (eds), *Proceedings of the CADE'96*, Vol. 1104 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 141–145.

Wos, L., Carson, D. et Robinson, G. A. (1965). Efficiency and completeness of the set of support strategy in theorem proving, *Journal of the ACM* **12**: 536–541.

Yelick, K. A. et Garland, S. J. (1992). A parallel completion procedure for term rewriting systems, *in* D. Kapur (ed.), *11th International Conference on Automated Deduction*, LNAI 607, Springer-Verlag, Saratoga Springs, New York, USA, pp. 109–123.

Index thématique

$(\mathcal{T}(\mathcal{F},\mathcal{X})/E,\mathcal{F})$, 12
$ACL[\mathcal{F},\mathcal{P}]$, 17
E_∞, 35
GG_{init}, 178
G_∞, 162
Gr, 22
$Irred(Gr(E_\infty))$, 183
$Mod(E)$, 12
$SL[\mathcal{F},\mathcal{P}]$, 18
Sol_G, 19
Sol_I, 17
Sub, 10
T_∞, 162
$Th(E)$, 11
$\Box$, 14
$\mathcal{F}$, 10
$\mathcal{P}$, 10
$\mathcal{V}(t)$, 10
$\mathcal{X}$, 10
$\approx_E$, 12
$\perp$, 17
$\mathcal{T}(\mathcal{F})$, 10
$\mathcal{T}(\mathcal{F},\mathcal{X})$, 10
$\leftrightarrow_E$, 11
$\rightarrow_R$, 13
$\sim$, 11
$\overset{*}{\leftrightarrow}_E$, 11
$\succ_C$, 16
$\succ_L$, 16
$\succ_g$, 183
$\succ_p$, 15
$\succ_t$, 16
$\top$, 17
$\lhd$, 11, 18
$\lhd_T$, 18, 202
$\vdash$, 11

Abdulrab, H., *20*, *223*
algèbre, 11
  généré par les termes, 12
Alouini, E., *74*, *223*
Anantharaman, S., *2*, *34*, *49*, *51*, *223*
arc
  d'inférence (arc I), 157
  d'orientation, 81, 112
  d'unification, 81
  de contraintes (arcs C), 157
  de réécriture, 81
  de simplification (arc S), 157
  de sous-terme, 81
arité, 10
Astrachan, O., *34*, *77*, *223*
atome, 14
Avenhaus, J., *76*, *77*, *223*
axiome, 11
  de congruence, 69
  de réflexivité fonctionnelle, 69

Bündgen, R., *77*, *80*, *225*
Baader, F., *19*, *223*
Bachmair, L., *1–3*, *16*, *35–37*, *41*, *43*, *49*, *51*, *54*, *59–61*, *63*, *65*, *70–72*, *152*, *178*, *186*, *196*, *197*, *200*, *223*, *224*
Backward Simplification, 194
Baumagarter, P., *34*, *224*
BCBS, 63
  complètude de, 62
  correction de, 62
BCES, 152, 158
  complétude de, 178, 186
  correction de, 160
$BCICS_m$, 203
  complétude de, 216
  correction de, 213
BC_m, 65
BCSB, 58
  complètude de, 58
  correction de, 58

BCSS, 54
complètude de, 57
correction de, 57
BCSSSB, 59
Beguelin, A., *73*, *81*, *147*, *224*
Benanav, D., *21*, *224*
Bendix, P. B., *38*, *39*, *229*
Birkhoff, G., *11*, *12*, *224*
Bofill, M., *37*, *178*, *224*
Bonacina, M.-P., *3*, *5*, *32*, *33*, *73*, *74*, *77*, *80*, *224*
Borovanský, P., *5*, *22*, *23*, *32*, *189*, *206*, *225*
Bose, S., *77*, *225*
Bousdira, W., *34*, *223*
BP, 71
complétude de, 71
correction de, 71
Brand, D., *1*, *70*, *225*
Bruening, S., *34*, *224*
Buntine, W., *69*, *225*
but, 32
Bürckert, H.-J., *21*, *69*, *225*

Caferra, R., *69*, *225*
Carson, D., *67*, *233*
Castro, C., *23*, *155*, *225*
Chang, C. L., *67*, *225*
Cirstea, H., *5*, *23*, *189*, *206*, *225*, *226*
Clarke, E. M., *77*, *225*
clause, 14
maximale, 17
négative, 14
positive, 14
vide, 14
Clavel, M., *21*, *226*
clôture de congruence, 42
Comon, H., *21*, *52*, *226*
complëtion, 37
basique, 38, 51
des graphes SOUR, 81
modulo, 38, 64
close, 42
contrainte, 38, 51
modulo, 38, 64
de Knuth-Bendix, 38, 39
des graphes SOUR clos, 87
concurrente, 90
ordonnée, 38, 43
orientée but, 37
sans échec, 37, 38, 43
modulo, 38, 49
standard, 38, 43
complétude
complétion basique des graphes SOUR, 86
de $BCICS_m$, 216
de SC_m, 51
de BCBS, 62
de BCES, 178, 186
de BCSB, 58
de BCSS, 57
de BP, 71
de RF, 67
de UC, 49
réfutationnelle, 32
configuration, 81
RUR, 84
RUR (cas clos), 88
RUR-rhs, 88, 89
SUR, 84
SUR (cas clos), 88
congruence
sur les termes, 11
Conry, E., *77*, *226*
conséquence
logique, 14, 36
syntaxique, 11
contradiction
sytaxique
dans une contrainte de réductibilité, 154
contrainte
$\perp$, 17
$\top$, 17
CSMGU, 19
d'ordre, 19, 21
contrainte, 201
de contrainte, 201
de filtrage, 19, 21
contrainte, 201
de réductibilité, 153
contradiction syntaxique dans une, 154
satisfaisable, 154
ensemble complet des solutions,CSU, 19
équationnelle, 18
résolution, 17, 19

satisfaisable, 17
solution, 17
valide, 17
correction
complétion basique des graphes SOUR, 86
d'une dérivation de transitions de graphes, 162
de SC_m, 51
de BCBS, 62
de BCES, 160
de $BCICS_m$, 213
de BCSB, 58
de BCSS, 57
de BP, 71
de RF, 67
de UC, 49
CWD, 147

Dahn, I., *3*, *76*, *77*, *226*
Dauchet, M., *15*, *226*
demi-configuration, 81, 84
cas clos, 88
Denzinger, J., *3*, *76*, *77*, *223*, *226*, *227*
dérivation, 33
de transitions de graphes, 162
correcte, 162
équitable, 162
Dershowitz, N., *15*, *37*, *41*, *43*, *49*, *223*, *226*
SICO_ORDER_ARITY, 91
Dijkstra, E., *138*, *226*
Domenjoud, E., *2*, *4*, *21*, *39*, *198–200*, *226*
Dongarra, J., *73*, *81*, *147*, *224*, *232*
Dowek, G., *2*, *38*, *199*, *226*
Dubois, H., *5*, *23*, *189*, *206*, *225*
Duràn, F., *21*, *226*
Dwork, C., *104*, *226*

E-chemin, 157
E-cycle, 158
ECC, 153, 189
égalité, 11
conclusion, 43
consistante, 12
contrainte, 22
correctement, 213
droite, 43
extension, 49
modulo, 65
g-redondante, 184
gauche, 43
inconsistante, 12
instance d'une, 22
membre droit d'une, 11
membre gauche d'une, 11
persistante, 35
redondante, 34, 35
rétractée, 22
satisfaisable, 12
SR-redondante, 60
valide (vraie) dans une algèbre, 12, 36
Eker, S., *21*, *226*
ELAN, 22
ensemble
complet des solutions d'une contrainte, 19
saturé, 35
équation, 11, 18
équité
de dérivation de transitions de graphes, 162
état
d'un processus, 92
étiquette
de temps, 133, 134

Fages, F., *20*, *226*
Feijen, W., *138*, *226*
filtrage, 21
contrainte de, 201
problème de, 11
fonction
EqToRedConst, 157
IsContradictoire, 155
label, 157
TransfoCP, 206
TransfoSimp, 211
formule
contrainte, 22
instance d'une, 22
persistante, 35
redondante, 34, 35
rétractée, 22
satisfaisable, 36
valide (vraie) dans une algèbre, 36
Forward Simplification, 194

Fuchs, D., *3*, *76*, *77*, *227*
Fuchs, M., *76*, *77*, *223*

g-redondance, 184
Göbel, M., *77*, *80*, *225*
Gaede, B., *3*, *34*, *232*
Gallier, J., *42*, *147*, *227*
Ganzinger, H., *1–3*, *16*, *35*, *36*, *51*, *54*, *59–61*, *63*, *65*, *70–72*, *152*, *178*, *186*, *196*, *197*, *200*, *223*, *224*
Garland, S. J., *77*, *233*
Geist, G. A., *73*, *81*, *147*, *224*
Genet, T., *22*, *23*, *112*, *219*, *227*
Gnaedig, I., *22*, *23*, *112*, *219*, *227*
Godoy, G., *37*, *178*, *224*
Goller, C., *34*, *229*
graphe
  de dépendance, 157
    clos, 178
  SOUR, 81
    avec étiquette de temps, 136
    initial, 81
    sémantique, 83
    syntaxe, 82
Gropp, W., *73*, *227*

Hardin, T., *2*, *38*, *199*, *226*
Hermann, M., *42*, *158*, *189*, *227*
Herold, A., *20*, *227*
Hintermeier, C., *52*, *227*
Hsiang, J., *2*, *3*, *37*, *49*, *51*, *70*, *73*, *74*, *77*, *80*, *223*, *224*, *227*
Huet, G., *15*, *19*, *20*, *49*, *61*, *198*, *227*
Hullot, J.-M., *19*, *21*, *227*
Huss-Lederman, S., *73*, *232*

implantation
  CWD, 147
  ECC, 153, 189
inéquation, 18
interprétation, 11
  d'égalités de Herbrand, 36

Jaffar, J., *20*, *227*
Jouannaud, J.-P., *16*, *19*, *20*, *22*, *49*, *228*

Küchlin, W., *77*, *80*, *225*
Kamin, S., *15*, *228*
Kanellakis, P., *104*, *226*
Kapur, D., *4*, *21*, *34*, *43*, *51*, *198*, *200*, *224*, *228*
Kirchner, C., *1–3*, *5*, *17*, *19*, *20*, *22*, *23*, *31*, *38*, *51*, *74*, *77*, *81*, *159*, *189*, *199*, *206*, *225*, *226*, *228*
Kirchner, H., *1*, *5*, *17*, *22*, *23*, *49*, *51*, *159*, *189*, *206*, *225*, *228–230*
Knuth, D. E., *38*, *39*, *229*
Kolata, G., *152*, *229*
Kozen, D., *14*, *42*, *229*

langage
  de contraintes, 17
    décidable, 18
Lankford, D. S., *15*, *227*, *229*
Lassez, J.-L., *229*
Lee, R. C., *67*, *225*
Lescanne, P., *42*, *229*
Letz, R., *34*, *77*, *229*
lifting, 36
limite, 35
Lincoln, P., *21*, *226*
Lindenstrauss, N., *74*, *229*
littéral, 14
Livesey, M., *20*, *21*, *229*
localité
  d'une règle d'inférence, 186
logique équationnelle, 11
Long, D. E., *77*, *225*
Loveland, D., *34*, *77*, *223*
lpo, 15
  implantation graphique de, 112
Lusk, E., *73*, *77*, *227*, *229*
Lynch, C., *1–4*, *16*, *23*, *31*, *37*, *38*, *42*, *54*, *59–61*, *63*, *65*, *70*, *71*, *74*, *77*, *80*, *81*, *86*, *87*, *152*, *153*, *178*, *186*, *196*, *200*, *201*, *224*, *228–230*
Lévy, J.-J., *15*, *228*

MacIntosh, D., *77*, *226*
Maher, M. J., *229*
Makanin, G. S., *20*, *230*
Marriot, K., *229*
Martelli, A., *20*, *230*
Mayr, K., *34*, *229*
McCune, W., *2*, *34*, *71*, *77*, *152*, *224*, *229*, *230*
Meseguer, J., *21*, *22*, *226*, *230*

message
ANSWERO1, 115
ANSWERO2, 116
ANSWERU, 105
CALCULO, 115
CALCULO1, 116
CONFIG, 98
d'initialisation, 90
de complétion, 90
de terminaison, 90
INFOU, 105
INITPROCESS, 91
INITR, 91
INITS, 91
INITU, 91
INO, 115
NOTIFY, 138
OUTO, 115
REQUESTO1, 114
REQUESTO2, 114
REQUESTU, 105
SEMICONF, 98
SYMBOL, 91
UPDATER, 102
UPDATES, 102
UPDATESEMICONF, 102
Meyer, R., *77*, *226*
Michaylov, S., *77*, *225*
Mitchell, J. C., *104*, *226*
modèle, 11
d'égalité de Herbrand, 36
Montanari, U., *20*, *230*
Moreau, P.-E., *5*, *23*, *189*, *206*, *225*, *229*, *230*
morphisme, 12
Muñoz, M., *16*, *228*
Mzali, J., *49*, *51*, *223*

Naher, S., *147*, *230*
Narendran, P., *4*, *16*, *21*, *42*, *43*, *147*, *198*, *200*, *224*, *227*, *228*, *230*
Nieuwenhuis, R., *1–3*, *6*, *16*, *21*, *34*, *37*, *52*, *54*, *57*, *59*, *63*, *64*, *71*, *72*, *152*, *167*, *168*, *178*, *186*, *194*, *196*, *197*, *200*, *216*, *218*, *220*, *221*, *224*, *230*, *231*
Nivela, P., *21*, *34*, *231*

Okada, M., *22*, *228*
ordre, 14
bien-fondé, 15
de réduction, 14
de simplification, 15
extension lexicographique, 15
extension multi-ensemble, 15
monotone (stable par contexte), 15
partiel, 14
propriété de sous-terme, 15
stable par instanciation, 15
strict, 14
sur les clauses, 16
T-compatible, 16
total, 15
Otto, S., *73*, *232*

paire
critique, 39
bloquée, 51
parallélisme
de compétition, 76
de partitionnement, 76
ET, 73
grain de, 74
OU, 73
paramodulation, 69
basique, 71
modulo, 72
sans duplication, 42
passage du cas clos au cas non clos (lifting), 36
Paterson, M. S., *20*, *198*, *200*, *231*
Pelletier, F., *231*
Peterson, G., *1*, *2*, *49*, *70*, *153*, *231*
pic, 41
Plaisted, D., *16*, *42*, *43*, *49*, *147*, *223*, *224*, *227*, *231*
plan
de recherche de preuve, 32
Plotkin, G., *19*, *20*, *218*, *221*, *231*
position
dans un terme, 10
précédence, 15
preuve, 32
en vallée, 37
par réécriture, 37
pic dans une, 41
transformation de, 41
problème

d'unification, 18
du mot, 14
processus
état, 92
S-fils, 82
S-père, 82
prouveur de théorème, 32, 33
Pécuchet, J.-P., *20*, *223*, *231*

Raatz, S., *42*, *147*, *227*
raisonnement
arrière (backward), 33
avant (forward), 33
redondance, 35
reduction des contre-exemples, 37
règle
d'inférence, 32
Blocage Basique, 63
Blocage E-cycle, 160
Blocage Standard, 57, 58
de simplification, 34
de suppression, 34
locale, 186
Paire Critique, 43, 47
Paire Critique Basique, 52, 53
Paire Critique Basique 2, 203
Paire Critique Basique dans la Contrainte, 210
Paire Critique Basique modulo, 65
Paire Critique modulo, 50
Rétraction E-cycle, 160
Rétraction, 64
Simplification, 204
Simplification Basique, 61, 62
Simplification dans la contrainte, 212
Simplification E-cycle, 159
Simplification Standard, 55, 56
Simplification Standard 1, 44, 48
de réécriture, 13
ContradictionForEC1, 155
ContradictionForEC2, 155
EndTransfoCP, 207
EndTransfoSimp, 211
IsSolution, 155
nommée, 25
PostElementaryConjunct, 155
Transfo, 206
résolution, 67

RF, 67
complétude de, 67
correction de, 67
Ringeissen, C., *5*, *23*, *189*, *206*, *225*, *228*, *231*
Rivero, J., *16*, *230*
Robinson, G. A., *1*, *67*, *69*, *231*, *233*
Robinson, J. A., *20*, *35*, *231*
Rock, G., *3*, *34*, *232*
rpo, 15
Rubio, A., *1–3*, *16*, *21*, *37*, *50*, *52*, *54*, *57*, *59*, *63*, *64*, *71*, *72*, *152*, *167*, *168*, *178*, *186*, *194*, *196*, *200*, *216*, *218*, *221*, *224*, *230*, *231*
Rusinowitch, M., *1*, *16*, *17*, *37*, *51*, *70*, *72*, *159*, *227*, *228*, *230*, *232*

satisfaisabilité
d'une contrainte de réductibilité, 154
Sattler-Klein, A., *42*, *231*
saturation, 35
Scharff, C., *3*, *4*, *23*, *31*, *74*, *77*, *81*, *153*, *200*, *201*, *228*, *229*
Schumann, J., *3*, *73*, *75*, *77*, *80*, *229*, *232*
SC_m, 50
complètude de, 51
correction de, 51
sémantique
des graphes SOUR, 83
S-fils, 82
Siekmann, J., *19–21*, *223*, *229*, *232*
signature, 10, 23
Sivakumar, G., *51*, *228*
Skjellum, A., *73*, *227*
Smolka, G., *17*, *232*
Snir, M., *73*, *232*
Snyder, W., *1–3*, *16*, *42*, *54*, *59–61*, *63*, *65*, *70*, *71*, *147*, *152*, *153*, *178*, *186*, *196*, *200*, *224*, *227*, *229*, *232*
sorte, 24
ELAN
constraint, 45
eqconstrained, 44
redconstraint, 153
S-père, 82
SR-redondance, 60
Stickel, M. E., *2*, *21*, *49*, *231*, *232*
stratégie
de déduction, 35

de recherche de preuve, 32
    d'hyper-résolution, 68
    d'ordre, 33
    de réduction de sous-but, 34
    équitable, 33, 36
    hyper-paramodulation, 71
    linéaire, 68
    positive, 68
    SOS, 67
de simplification
    S_1, 194
    S_2, 194
    S_3 (stratégie de simplification initiale), 194
    basique, 59, 65
    E-cycle, 152, 158
    standard, 44, 54
ELAN, 25
    ;, 26
    AllDedStrat, 207
    ContradictionForECandDC, 156
    dc, 27
    dc one, 27
    dk, 26
    dk one, 27
    fail, 26
    first, 27
    first one, 27
    id, 26
    iterate +, 27
    iterate*, 27
    normalisation, 27
    OneDedStrat, 207
    OneSimpStrat, 211
    repeat*, 27
    repeat+, 27
Strogova, P., *38*, *42*, *80*, *81*, *86*, *87*, *230*
substitution, 10
    composition, 10
    domaine, 10
    identité, 10
    image, 10
    nouvelle, 10
    plus générale, 19
    réductibilité, 59
    renommage, 10
Sunderam, V., *73*, *81*, *96*, *147*, *232*
Suttner, C., *3*, *73*, *75*, *80*, *232*
symbole
    de tête, 10
syntaxe
    des graphes SOUR, 82
système
    d'inférence, 32
        BCBS, 63
        BCES, 152, 158
        BC_m, 65
        BCSB, 58
        BCSS, 54
        BCSSSB, 59
        BP, 71
        complet, 32, 36
        correct, 32
        PRF, 69
        réfutationnellement complet, 36
        RF, 67
        SC_m, 50
        UC, 43
    de calcul, 22
    de réécriture, 13
        Church-Rosser, 13
        confluent, 13
        convergent, 13
        équivalent, 13
        forme normale, 13
        localement confluent, 13
        terminaison, 15
        terminant, 13

Tanenbaum, A., *72*, *232*
terme, 10
    clos, 10
    congruence, 11
    instance, 11
    joingnables (termes), 13
    position, 10
    sous-terme, 10
    taille, 10
terminaison
    message de, 90
    programme concurrent, 138
théorie
    built-in, 38
    équationnelle, 11
        A, 11
        AC, 11

C, 11
intégrée, 38
modulo, 38
transformation
de preuve, 41
RUR, 86
clos, 89
RUR-rhs
clos, 89
SUR, 85
clos, 88
transition, 33
C, 95, 100
de graphes, 161
dérivation de, 162
O, 118
P, 102
T, 139
U, 107

UC, 43
complètude de, 49
correction de, 49
unificateur, 19
unification, 19
associative, 20
associative-commutative, 21
commutative, 20
finitaire, 19
infinitaire, 19
syntaxique, 20
U-basée, 19
unitaire, 19

valuation, 12
Van Gasteren, A., *138*, *226*
Vigneron, L., *1–3*, *64–66*, *72*, *200*, *201*, *232*
Viry, P., *74*, *228*
Vittek, M., *5*, *21–23*, *189*, *206*, *225*, *232*

Walker, D., *73*, *232*
Wegman, M. N., *20*, *198*, *200*, *231*
Weidenbach, C., *3*, *34*, *232*
Wos, L., *1*, *67*, *69*, *231*, *233*

Yelick, K. A., *77*, *233*

Zabel, N., *69*, *225*
Zhang, H., *34*, *51*, *228*

université HENRI POINCARÉ NANCY 1

FACULTÉ DES SCIENCES

S.C.D. - U.H.P. NANCY 1 SCIENCES

Mademoiselle **SCHARFF Christelle**

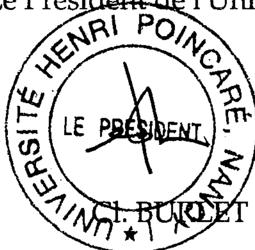
DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 4 octobre 1999 n° 266

Le Président de l'Université



Cl. BURLET

3C-F9