N° d'ordre : 6610

UNIVERSITÉ PARIS-SUD
UFR SCIENTIFIQUE D'ORSAY

THÈSE

Présentée par

XAVIER URBAIN

Pour obtenir le grade de

DOCTEUR EN SCIENCES DE L'UNIVERSITÉ PARIS XI

Spécialité : INFORMATIQUE

Sujet :

APPROCHE INCRÉMENTALE DES PREUVES AUTOMATIQUES DE TERMINAISON

Soutenue le 1er octobre 2001 devant la commission d'examen composée de

M. THOMAS ARTS	
M. BERNHARD GRAMLICH	rapporteur
M. JEAN-PIERRE JOUANNAUD	
M. CLAUDE MARCHÉ	
Mme MARIE-CHRISTINE ROUSSET	présidente
M. MICHAËL RUSINOWITCH	rapporteur

Remerciements

Je voudrais en premier lieu remercier mon directeur de thèse, Claude Marché. En plus de ses qualités scientifiques incontestées, j'aimerais exprimer ici combien j'ai apprécié sa disponibilité, sa générosité et un « certain état d'esprit » rare et précieux[1].

Mon directeur de thèse officiel, Jean-Pierre Jouannaud, m'a laissé une grande liberté dans mes travaux. J'aimerais le remercier de la confiance qu'il m'a ainsi accordée ; j'espère m'en être montré digne.

Bernhard Gramlich & Michaël Rusinowitch ont accepté d'être rapporteurs de ma thèse. Qu'ils se soient intéressés à mon travail est particulièrement important pour moi. Merci à eux.

Je suis très heureux d'avoir pu compter Thomas Arts parmi les membres du jury. J'aimerais le remercier des échanges intéressants que nous avons eus et pour les importants exemples qu'il m'a fournis.

Merci enfin à Marie-Christine Rousset de m'avoir fait l'honneur de présider le jury.

Je fus, pendant ces trois années, entouré par l'équipe DÉMONS du L.R.I. Étudier au sein de cette équipe est une véritable chance. Je n'oublierai pas l'ambiance qui y règne, les gâteaux et ses membres que j'apprécie. Merci.

Merci à mes formidables amis, je leur dois plus que du soutien. Les inestimables membres du CORN dans le désordre : Laurent « Bloᗺ » Vaucher (chouettes lettrines !), Philippe « Pachy » Renault, Frédéric Hantrais, Guillaume Hanrot (chouette fonte !), David Gross-Amblard et Gégé Amblard-Gross, Sylvain « S. » Pogodalla. Merci à Maman, encore à David, à Guillaume et à Jean-Christophe Filliâtre de (mais pas que de) leur participation à la relecture du document. Merci à Pierre, Julien, Evelyne, Benjamin, Stephan, Philippe, etc. Je ne peux pas tous vous citer ici.

Je tiens à remercier tout particulièrement ma mère qui a tout fait pour que j'en sois ici un jour.

Merci enfin à Hélène, de ce qui n'est pas écrit aussi.

[1]Son gâteau aux framboises n'est pas mal non plus.

Table des matières

I Introduction **9**

II Relations, récriture **15**
II.1 Ordres . . . 15
II.2 Relations sur les termes . . . 16
II.2.1 Signatures et termes . . . 16
II.2.2 Ordres sur les termes . . . 18
II.2.3 Récriture de termes . . . 21
II.2.4 Caractéristiques des systèmes de récriture . . . 22
II.2.5 Stratégies de réduction . . . 24

III Terminaison **27**
III.1 Problème de l'arrêt et indécidabilité . . . 27
III.1.1 Récriture et machines de Turing . . . 27
III.2 Preuve de terminaison . . . 28
III.3 Méthodes classiques . . . 29
III.3.1 Terminaison à l'aide d'ordres de simplification . . . 29
III.4 Critères des paires de dépendance . . . 30
III.4.1 Paires de dépendance . . . 30
III.4.2 Critères de terminaison . . . 32
III.4.3 Graphes de dépendance . . . 34
III.5 Des ordres pour la terminaison . . . 37
III.5.1 Méthodes syntaxiques . . . 37
III.5.2 Méthodes sémantiques . . . 38
III.6 Méthodes transformationnelles . . . 42
III.7 Transformation des contraintes d'ordre . . . 42

STRUCTURE HIÉRARCHIQUE DES SYSTÈMES DE RÉCRITURE

I Unions de systèmes **49**
I.1 Unions et modularité . . . 49
I.2 Unions et terminaison . . . 50
I.2.1 Unions disjointes . . . 51
I.2.2 Analyse structurelle des contre-exemples . . . 52

I.2.3 Constructeurs partagés et unions de systèmes composables . . . 53
I.2.4 Unions hiérarchiques . . . 54
I.2.5 Approche modulaire par paires de dépendance . . . 55
I.2.6 Propriétés de la terminaison $\mathcal{C}_{\mathcal{E}}$ . . . 59

II Modules de récriture **63**
II.1 Modules et extensions . . . 63
II.2 Décomposition en modules . . . 64

III Terminaison et modules **67**
III.1 Ordres π-extensibles . . . 67
III.1.1 Des ordres pour $\mathcal{C}_{\mathcal{E}}$ . . . 67
III.1.2 Construction des ordres π-extensibles . . . 67
III.2 Paires de dépendance de modules . . . 69

IV Preuves incrémentales **71**
IV.1 Un module pour un système . . . 71
IV.2 Deux modules indépendants pour un système . . . 77
IV.3 Exemple complet . . . 79
IV.4 Modules et graphes . . . 83

TERMINAISON DES SYSTÈMES ASSOCIATIFS ET COMMUTATIFS

I Introduction **89**

II Récriture avec symboles associatifs et commutatifs **91**
II.1 Récriture AC . . . 92
II.2 Terminaison de la récriture AC . . . 95
II.2.1 Compatibilité et interprétations polynomiales . . . 96
II.2.2 Compatibilité et ordres sur les chemins . . . 97
II.2.3 Un RPO compatible AC sans système de normalisation . . . 101

III Critères de terminaison et paires de dépendance **103**
III.1 Un premier critère à l'aide de paires de dépendance . . . 103
III.2 Critères avec symboles marqués . . . 109
III.2.1 Symboles marqués et termes marqués . . . 109
III.2.2 Conditions de dépendance . . . 111
III.2.3 Critères de terminaisons avec marques . . . 113
III.2.4 Calcul des paires marquées . . . 114

IV Approche modulaire de la terminaison AC **121**
IV.1 Modules AC . . . 121
IV.1.1 Paires de dépendance relatives AC . . . 121
IV.2 Preuves incrémentales pour la récriture AC . . . 122

Implantation

I Introduction **129**

II Un outil de preuves : C*i*ME2 **131**
II.1 Présentation générale 131
II.2 La terminaison dans C*i*ME2 132
II.2.1 Principes généraux 132
II.2.2 Comparaisons de polynômes 133
II.2.3 Recherche de preuve 134

III Modules **137**
III.1 Problématique 137
III.2 Extension du langage et implantation 138
III.2.1 Structures de données du langage 138
III.2.2 Choix d'implantation 139
III.3 Exemple d'exécution 140

IV Critères AC **151**
IV.1 Problématique 151
IV.2 Implantation 151

V Exemple **155**

VI Conclusion **183**

Chapitre I

Introduction

La place extraordinairement importante qu'a prise l'informatique aujourd'hui a entraîné une production intensive de programmes. On les trouve présents dans notre vie à tous les niveaux, dans les ordinateurs personnels bien sûr mais aussi des ascenseurs aux cartes de crédit en passant par les avions et les centrales nucléaires. Or parmi ces programmes qui nous entourent, beaucoup sont erronés et les plus répandus ne sont pas épargnés. Une telle situation est déjà inquiétante, il vaudrait mieux éviter qu'elle ne devienne catastrophique.

L'idéal serait d'avoir les moyens de garantir, pour chaque programme, qu'il est stable, qu'il respecte bien les spécifications dont il est issu, que celles-ci sont cohérentes et correctes, etc.

Une certaine discipline de programmation donne déjà quelques résultats. Une encapsulation rigoureuse et un souci de découpage d'un code important en différents *modules* permettent par exemple de séparer les problèmes d'implantation et de clarifier les relations entre les différentes structures opérationnelles. Ce n'est cependant pas suffisant.

Si on remarque que les programmes (et les ordinateurs) accomplissent des opérations symboliques, les propriétés désirées deviennent en fait des caractéristiques mathématiques de ces calculs. On aimerait donc en toute rigueur pouvoir se fier à leur comportement et leurs résultats comme à des preuves. L'application de méthodes formelles à la preuve de propriétés de programme est conséquemment en plein essor.

Parmi toutes ces propriétés, c'est la terminaison qui va nous intéresser.

Une propriété : la terminaison

La terminaison, c'est-à-dire la propriété caractéristique d'un programme dont toute exécution s'arrête au bout d'un temps fini, mérite son statut de propriété fondamentale dans la mesure où elle conditionne l'existence même du calcul défini par un programme : sans terminaison, pas de résultat, en tout cas pas dans un temps fini.

Elle intervient dans de nombreux domaines, on la trouve indispensable dans la définition de certains modèles de calculs, elle règle les oscillations de stabilisation des processus de contrôles, sa preuve donne aussi quelques indications sur la complexité d'algorithmes. Elle reste enfin un préliminaire incontournable à la preuve d'autres propriétés.

Sa célébrité en informatique vient aussi de ce qu'elle est, sous le nom de *problème de l'arrêt*, l'exemple paradigmatique du problème *indécidable*. Il n'existe en effet aucun algorithme capable de dire lorsqu'on l'applique sur un code de programme si toutes ses exécutions sont finies ou pas. Pour l'idée de la preuve en quelques lignes : si un tel algorithme existait, il est facile de voir qu'on pourrait écrire un programme prenant un code en argument, déterminant si ce code n'a que des exécutions finies et, dans ce cas, entrant dans une boucle infinie, ou alors, dans le cas contraire, stoppant son exécution. Or si on lui passait son propre code, ce programme s'arrêterait si et seulement s'il ne s'arrêtait pas, ce qui pour le moins est absurde.

L'application de méthodes formelles nécessite ensuite qu'on s'intéresse à un *formalisme* particulier.

Un formalisme : la récriture

Un programme manipulant des expressions symboliques, on pourrait penser décrire ces opérations à l'aide du raisonnement équationnel classique dont le principe est le remplacement au sein des expressions de termes par des termes égaux. Si sa puissance et sa lisibilité en font le mode de raisonnement le plus répandu, le raisonnement équationnel fait en revanche énormément appel à l'intuition : « où remplacer ? » et « par quoi remplacer ? » sont deux questions difficiles à résoudre. Il est donc peu adapté à la modélisation de programmes.

Permettre une décision automatique du chemin à suivre est l'un des buts de la *récriture*. Le principe est simple : orienter les équations en ne considérant les égalités valables que « dans un seul sens ». Le résultat, lui, est redoutablement puissant. On dispose à présent d'un modèle de calcul lisible (en fait aussi lisible que le cas équationnel) et à la sémantique claire (à comparer avec le λ-calcul), programmable et simple à programmer. Il en existe des implantations très efficaces comme le système C*i*ME au L.R.I., ÉLAN développé au L.O.R.I.A. ou encore le langage MAUDE du S.R.I.

À titre d'exemple comparons l'addition d'entiers de Church en λ-calcul :

$$\lambda x.\lambda y.\lambda p.\lambda q.((xp)((yp)q))$$

À l'addition d'entiers de Peàno définie par un système de récriture :

$$\begin{cases} x+0 & \rightarrow & x \\ x+s(y) & \rightarrow & s(x+y) \end{cases}$$

Où 0 est une constante et s l'opération « successeur ».

Par ailleurs, la récriture étant en relation étroite avec la logique équationnelle, elle est non seulement un outil de calcul mais aussi un outil de preuve dans cette logique. À titre d'exemple, sur la structure de groupe définie par :

$$\begin{array}{rcl} x.e & = & x, \\ x.x^{-1} & = & e, \\ (x.y).z & = & x.(y.z), \end{array}$$

La récriture permet de prouver automatiquement des propriétés comme $e.x = x$, $e^{-1} = e$, $(x^{-1})^{-1} = x$, etc.

Les systèmes de récriture jouant ainsi un rôle important dans l'implantation et l'analyse des spécifications de types abstraits algébriques (consistance, preuve de propriétés, exécutabilité), c'est dans ce cadre que nous étudierons la terminaison des programmes.

Nous consacrerons le prochain chapitre à la définition formelle des objets clefs, fondamentaux pour la suite de notre étude : les relations, les algèbres de termes et, en particulier, les relations d'ordre et de récriture sur les termes. Nous passerons alors en revue certaines de leurs caractéristiques avant de nous concentrer sur la notion qui nous intéresse : la terminaison (chapitre III).

Cette propriété est indécidable (section III.1) : les méthodes de preuves existantes sont donc incomplètes. La plupart des méthodes utilisées couramment ont toutefois en commun un fil conducteur et nous verrons alors qu'ordres et récriture sont intimement liés dans la preuve de terminaison.

Cette liaison est exploitée de différentes façons. Après une présentation des approches classiques destinées à montrer la terminaison, nous détaillerons la méthode dite des *paires de dépendance* [3, 4]. Elle autorise la définition dans la section III.4 de critères puissants, affaiblissant en la rendant moins exclusive la relation entre ordres et récriture.

Prouver la terminaison d'un système revient pour toutes ces approches à savoir lui associer un ordre convenable. Nous exposerons en conclusion de ces prolégomènes les principales techniques dont on dispose pour tenter d'exhiber cet ordre adéquat.

Nous aborderons alors la première partie de ce mémoire, consacrée à l'étude de la structure hiérarchique des systèmes de récriture.

Modularité

Si la modularité est devenue incontournable en programmation, c'est que chaque programme devrait en fait être développé de façon modulaire et incrémentale. Les efforts en ce sens sont perceptibles dans de nombreuses formes de langages, des *templates* de `C++` au système de *modules* d'`Ocaml` en passant par l'encapsulation des langages à objets. Ils sont aussi sensibles au cœur des langages de plus haut niveau comme ceux des assistants de preuves [14].

Une telle discipline porte ses fruits tant au niveau de la propreté du code que de la clarté des spécifications et permet d'adjoindre à la sémantique d'un algorithme une méthodologie de développement.

Les entiers, par exemple, sont en pratique définis une fois pour toute, il suffit alors de préciser pour chaque nouvelle fonction (sur les entiers) qu'elle travaille sur eux. Prenons l'addition sur les entiers en binaire : la structure des entiers est précisée (disons dans un module `Mes_entiers`), on introduit ensuite le calcul d'addition proprement dit. Si on veut introduire une fonction f plus complexe utilisant les entiers et l'addition, il suffit de rendre évidente la dépendance de f vis-à-vis de l'ensemble *entiers* + *addition*. En `C++` on utilise alors une directive de précompilation `#include`, dans le langage `Ocaml`, on peut préciser `open Mes_entiers`[1].

L'avantage d'un découpage hiérarchique est encore plus sensible dès qu'on s'intéresse aux preuves de propriétés. Une preuve de terminaison, en particulier, est toujours difficile à faire. Elle est d'autant plus délicate à effectuer (et à automatiser) que le programme est gros. Accumuler des informations sur des « briques » de base et les assembler pour obtenir de l'information sur un sous-ensemble plus important permet alors d'adopter pour la preuve une stratégie *Divide and Conquer*.

On imagine mal aujourd'hui avoir à redéfinir et recoder une structure d'entiers, par exemple, à chaque fois qu'on introduit une nouvelle opération les manipulant. Qui voudrait prouver à nouveau les propriétés d'une structure à chaque ajout d'une partie de code dont *on sait* qu'elle n'y interviendra pas ?

La vague de la révolution modulaire a déferlé sur toute l'informatique. Toute ? Non [35]. Si on peut affirmer que la programmation « monolithique » a vécu, l'approche incrémentale n'a pas bouleversé le regard porté sur les systèmes de récriture et encore moins, malgré de nombreux travaux, la *pratique* des preuves de terminaison.

Nous tenterons pour commencer (chapitre I) d'analyser l'échec relatif des méthodes existantes à manipuler de façon satisfaisante des unions de systèmes.

[1] La sémantique de cette opération est en fait plus complexe car elle dispense d'avoir à préciser à chacun de ses appels le module original d'une fonction.

L'extrême difficulté qu'il y a à garantir qu'une union possède le même comportement que ses constituants vis-à-vis de la terminaison a en effet trop souvent été contournée en posant des restrictions fortes sur les relations entre les composants, en ne traitant pas ce qui dans la terminaison même pose problème.

Cette propriété n'est en effet même pas modulaire pour les systèmes qui ne partagent pas de symboles.

Exemple 1.

Considérons cet ensemble de systèmes dû à Toyama [83].

$$R : \left\{ \; f(0,1,x) \;\; \rightarrow \;\; f(x,x,x) \right. \qquad\qquad \pi : \left\{ \begin{array}{lcl} G(x,y) & \rightarrow & x \\ G(x,y) & \rightarrow & y \end{array} \right.$$

Les deux systèmes R *et* π *terminent. Si on étudie maintenant leur union*

$$\left\{ \begin{array}{lcl} G(x,y) & \rightarrow & x \\ G(x,y) & \rightarrow & y \\ f(0,1,x) & \rightarrow & f(x,x,x), \end{array} \right.$$

Il existe une réduction infinie du terme $f(G(0,1),G(0,1),G(0,1))$ *:*

$$\begin{array}{rcl} f(G(0,1),G(0,1),G(0,1)) & \rightarrow & f(0,G(0,1),G(0,1)) \\ & \rightarrow & f(0,1,G(0,1)) \\ & \rightarrow & f(G(0,1),G(0,1),G(0,1)) \rightarrow \ldots \end{array}$$

Nous verrons dans la suite que l'existence d'une telle réduction dans cet exemple est due à la projection effectuée par le système π.

La plupart des approches ont tenté de lutter malgré tout contre la non-modularité de la terminaison et les résultats sur les unions plus complexes de systèmes ont été obtenus en contraignant lourdement les relations entre constituants, écartant de ce fait la plupart des extensions naturelles et communes en programmation.

Nous adopterons la démarche inverse et préférerons permettre des unions peu contraintes (qu'on appellera *hiérarchiques*), mais par contre limiter la terminaison à une notion plus contrôlable dans les compositions de systèmes sans être trop restrictive, la terminaison $\mathcal{C}_\mathcal{E}$. Cette notion de terminaison a en particulier la propriété de supporter l'ajout de règles de projection ; l'exemple 1 ne pose alors pas de problème : le système R ne termine pas $\mathcal{C}_\mathcal{E}$, on ne cherchera pas à prouver automatiquement sa terminaison. Un tel système possède une forme qu'on peut qualifier de « pathologique », nous estimons en effet qu'en pratique, se limiter à la terminaison $\mathcal{C}_\mathcal{E}$ n'est pas une restriction. Certaines des caractéristiques fondamentales de la terminaison $\mathcal{C}_\mathcal{E}$ seront exposées au paragraphe I.2.6.

Structure hiérarchique

Dans toute la littérature c'est l'union ensembliste des règles qui définit un système et celle-ci n'est pas adaptée à l'expression de la hiérarchie intrinsèque des systèmes :

- Elle entraîne une redondance de définition au niveau des sous-ensembles communs dans les cas d'unions non disjointes ;

– Elle ne conserve pas d'information de séquentialité de construction et se prête donc mal à une démarche incrémentale.

Elle contribue ainsi à éloigner la récriture des réalités de la programmation.

Afin de fournir un cadre destiné à l'étude de la structure des systèmes de récriture nous introduirons au chapitre II la notion de *modules de récriture*. Nous verrons que, grâce à ces modules, la structure des systèmes de récriture prend une forme hiérarchique naturelle. Les unions classiques sont aisément représentées par des extensions de modules, ceux-ci définissent ainsi un cadre général susceptible d'être utilisé dans la preuve de terminaison.

Nous construirons alors au chapitre III des *paires de dépendance relatives*, c'est-à-dire des paires de dépendance de modules qui permettront de définir dans le chapitre IV des critères de terminaison puissants et automatisables, en particulier les théorèmes 18 et 19. Ces derniers pourront être mis en œuvre à l'aide d'ordres π-extensibles, des ordres qui permettent une expression de la terminaison $\mathcal{C}_{\mathcal{E}}$. La section IV.3 détaillera ensuite un exemple complet de preuve de terminaison faisant appel à ces nouvelles techniques. Les optimisations par graphes de dépendance sont tout à fait adaptables au cadre de l'analyse par modules, nous conclurons cette partie sur cette amélioration des critères.

Forts de ces résultats généraux, nous passerons à l'étude de la récriture modulo la théorie équationnelle décrivant l'associativité et la commutativité.

Associativité et commutativité

Il est très courant d'avoir à manipuler des opération naturellement associatives et commutatives. Imaginons un instant qu'on choisisse comme règles d'addition des entiers de Peàno :

$$\begin{cases} 0+x & \to & x \\ x+s(y) & \to & s(x+y). \end{cases}$$

Le terme $s(0)+0$ n'est pas réductible par ces règles ! Il est clair que sur un tel exemple on désire avoir une addition commutative et l'équation $x+y=y+x$ devrait être ajoutée. Remarquons qu'elle doit effectivement être ajoutée car elle n'est pas une conséquence logique de $0+x=x$ et $s(x)+y=s(x+y)$ (il existe des modèles de ces équations où $+$ n'est pas commutatif).

Il est malheureusement impossible d'exprimer directement la commutativité comme règle de récriture sans perdre d'importantes propriétés : $x+y\to y+x$ ne termine pas. L'approche classique consiste à considérer comme équivalents des termes obtenus l'un à partir de l'autre par commutativité et à appliquer la récriture sur les classes d'équivalence. On parle de récriture *modulo* commutativité.

Qu'en est-il de l'associativité ? Une opération associative, comme le produit dans les groupes vu précédemment, se traite par une règle $(x.y).z \to x.(y.z)$ qui termine, mais pour une opération à la fois associative et commutative cette règle ne termine pas modulo commutativité :

$$(a+b)+c \to a+(b+c) \equiv (c+b)+a \to c+(b+a) \equiv (a+b)+c, \text{ etc.}$$

On peut appliquer les règles de cette façon indéfiniment, le système ne termine plus.

Les opérations associatives-commutatives sont donc usuellement traitées par la récriture modulo associativité et commutativité (AC).

Remarquons que l'ajout d'AC modifie la logique du problème mais peut aussi être un facteur d'accélération d'un calcul : par exemple, en supposant le symbole « $+$ » associatif et commutatif dans l'addition des entiers de Peàno, il suffit d'une étape de récriture[2] pour obtenir le résultat du calcul de

$$0 + \overbrace{sss...s}^{1000}(0)$$

Alors qu'il en faut 1001 avec le système donné page 10 et la récriture non AC.

Nous définirons donc en premier lieu les relations de récriture modulo AC ; notre attention sera tout particulièrement portée sur la récriture *étendue AC sur termes aplatis*. Nous verrons ensuite à quel point la preuve de terminaison est compliquée par l'introduction des équations AC. Elles requièrent en effet de la part de l'ordre impliqué dans la preuve une compatibilité vis-à-vis des classes d'équivalence. Nous présenterons certaines des solutions trouvées pour satisfaire ces nouvelles contraintes : les polynômes AC, pour l'approche sémantique, et, pour les ordres syntaxiques, des systèmes de normalisation ou encore l'ordre AC-RPO.

Deux nouvelles approches avec paires de dépendance seront alors proposées. Si la méthode originale de Arts & Giesl peut être affinée par l'introduction de nouveaux symboles (en fait des copies avec marques) pour distinguer les instances de paires, les *paires étendues AC* définies dans un premier temps (section III.1) ne comporteront pas de symboles marqués. Une introduction directe des marques aboutit en fait à un échec si on cherche à affiner ainsi les critères de terminaison.

Nous prendrons alors le recul nécessaire par rapport à la relation (forte) liant paires de dépendance et structure des termes afin de définir des conditions plus générales et abstraites qui feront d'un ensemble de paires marquées les vérifiant un véritable *ensemble de paires de dépendance marquées*, utilisable pour la preuve de terminaison. Ce sont les *conditions de dépendance* (définition 85) de la section III.2.

La définition des ensembles de paires devenue intentionnelle, il nous faudra développer des moyens d'exhiber des ensembles idoines. Nous donnerons en fait dans le paragraphe III.2.4 deux méthodes de calcul de tels ensembles. Nous aurons alors totalement étendu à la terminaison modulo AC la puissance des paires de dépendance, leur souplesse d'utilisation mais aussi leur adéquation à l'automatisation.

La généralité du cadre d'étude défini par les modules est telle qu'elle autorise son application à la récriture AC. Le chapitre IV conclura cette deuxième partie, consacrée à la récriture AC, en l'unissant à la première pour aboutir aux *paires de dépendance relatives* AC.

Un outil de preuve

Tous ces résultats, toutes ces méthodes ont été développés avec le souci constant d'obtenir des critères automatisables. Il était donc normal d'en faire profiter un outil de preuve de terminaison. C'est à la présentation de l'implantation réalisée que sera dédiée la troisième partie.

Nous nous livrerons au cours du chapitre II à une description du système C*i*ME2 avec une attention particulière aux traitement des problèmes de terminaison. Les troisième et quatrième chapitres seront alors consacrés à l'extension du système par, respectivement, l'approche modulaire et la récriture AC. Pour ces deux cas nous exposerons la problématique d'une telle extension puis justifierons les choix faits, tant du côté des spécifications que de l'implantation. Nous terminerons cette dernière partie par quelques exemples d'exécution.

[2]Lors d'une réduction automatique, à l'aide d'un outil de récriture comme C*i*ME2 ou ÉLAN, une étape de récriture AC est plus coûteuse en temps qu'une étape de récriture classique, cependant ce prix peut être largement compensé par un nombre bien plus petit de pas à effectuer.

Chapitre II

Relations, récriture

Nous allons définir les notions fondamentales qui nous serviront dans toute la suite. Nous préciserons dans un premier temps des considérations sur les *relations binaires* générales avant de nous concentrer sur les relations d'*ordre*, puis, après avoir défini les *termes* sur une signature (section II.2), sur la relation de *récriture*. Nous décrirons alors au cours du paragraphe II.2.4 certaines des caractéristiques recherchées sur de telles relations avec un intérêt tout particulier pour la *terminaison*.

Une relation binaire sur un ensemble E est un ensemble de couples d'éléments de E. On dit de deux éléments formant un couple qu'ils sont *en relation*.

Définition 1. — Une *relation binaire* $\mathcal{R}$ sur un ensemble E est une partie de $E \times E$. Les éléments (s, t) de cette relation sont en général notés $s\mathcal{R}t$.

On utilise la notation $\circ$ pour la composition des relations : pour deux relations $\mathcal{R}$ et $\mathcal{S}$ sur un ensemble E, $\mathcal{R} \circ \mathcal{S}$ désigne la relation $\{(s,t) \mid \text{ il existe } s' \in E \text{ tel que } s\mathcal{R}s' \text{ et } s'\mathcal{S}t\}$.

Enfin, pour deux relations $\mathcal{R}$ et $\mathcal{S}$ sur un ensemble E, on note $\mathcal{R} - \mathcal{S}$ la différence ensembliste $\mathcal{R} \setminus (\mathcal{R} \cap \mathcal{S})$.

Définition 2. — Une relation binaire $\mathcal{R}$ sur E est :

- *Symétrique* si pour tous s et t de E, $s\mathcal{R}t$ entraîne $t\mathcal{R}s$;
- *Antisymétrique* si pour tous s et t de E, ($s\mathcal{R}t$ et $t\mathcal{R}s$) entraîne $s = t$;
- *Réflexive* si pour tout élément s de E, $s\mathcal{R}s$;
- *Antiréflexive* si pour tout s de E, on n'a *pas* $s\mathcal{R}s$;
- *Transitive* si pour tous s, t et u de E, ($s\mathcal{R}t$ et $t\mathcal{R}u$) entraîne $s\mathcal{R}u$.

Nous nous intéresserons dans la suite à la finitude éventuelle des séquences d'éléments en relation et en particulier aux relations *noethériennes*.

Définition 3. — Une relation sur un ensemble E est *noethérienne* s'il n'existe pas de chaîne infinie d'éléments de E en relation.

II.1 Ordres

Définition 4. — Soit $\mathcal{R}$ une relation binaire sur un ensemble E. $\mathcal{R}$ définit

- Une *relation d'équivalence* si elle est symétrique, réflexive et transitive ;

- Un *préordre* si elle est réflexive et transitive ;
- Un *ordre* si elle est réflexive, transitive et antisymétrique ;
- Un *ordre strict* si elle est antiréflexive, transitive et antisymétrique.

Dans une relation donnée par un préordre $\geq$ on distingue la *partie stricte associée* $>$ comme la différence $\geq - \leq$ où $\leq = \{(s,t) \mid (t,s) \in \geq\}$; le préordre $\geq$ peut être désigné comme *partie large* de la relation.

Définition 5. — Un préordre est dit *bien fondé* si sa partie stricte associée définit une relation noethérienne.

Les ordres bien fondés vérifient cette propriété fondamentale :

Proposition 1.
Toute relation incluse dans un ordre bien fondé est bien fondée.

II.2 Relations sur les termes

II.2.1 Signatures et termes

Définition 6. — Une *signature* $\mathcal{F}$ est un ensemble de *symboles* muni d'une application $\text{AR}_{\mathcal{F}} : \mathcal{F} \to \mathbb{N}$. On appelle *arité* du symbole $f \in \mathcal{F}$ l'entier $\text{AR}_{\mathcal{F}}(f)$.

On distingue parmi les symboles de $\mathcal{F}$ ceux d'arité nulle qu'on désigne comme *constantes*.

On décrira une signature par l'ensemble de ses symboles accompagnés éventuellement de leur arité si elle est nécessaire. La signature $\mathcal{F}$ donnée par $\{f \; ; \; a\}$ et $\text{AR}_{\mathcal{F}} : f \mapsto 1, a \mapsto 0$ sera par exemple notée $\{f : 1 \; ; \; a : 0\}$ ou encore $\{f : 1 \; ; \; a : \text{constante}\}$.

Définition 7. — Soient $\mathcal{F}$ une signature et X un ensemble dénombrable de symboles disjoint de $\mathcal{F}$. L'algèbre des *termes* sur $\mathcal{F}$ et X, notée $T(\mathcal{F}, X)$ est le plus petit ensemble tel que :

1. $X \subseteq T(\mathcal{F}, X)$ et on dit de X qu'il est l'ensemble des *variables* ;
2. Si $t_1, \ldots, t_n \in T(\mathcal{F}, X)^n$ et si $f \in \mathcal{F}$ avec $\text{AR}(f) = n$ alors $f(t_1, \ldots, t_n) \in T(\mathcal{F}, X)$.

Un élément de $T(\mathcal{F}, X)$ est un *terme*. Si t est un terme, on note $\mathcal{V}ar(t)$ l'ensemble des variables qu'il contient. Un terme sans variable est dit *clos*.

Les termes peuvent être représentés de façon naturelle sous forme d'arbres. On désigne alors les symboles les constituant avec des *positions*.

Définition 8. — Une *position* dans un terme est un mot sur l'alphabet $\mathbb{N} - \{0\}$. Le mot vide (représentant une position à la racine) est noté Λ. La concaténation de p et q est notée $p.q$ ou pq. $\mathcal{P}os(t)$ désigne l'ensemble des positions du terme t.

Le *symbole de tête* est le nom de fonction (ou de variable) positionné à la racine. On désigne par $\Lambda(t)$ le symbole de tête du terme t, voir figure II.1.

On définit le *sous-terme* de s à la position p (noté $s|_p$) par :

- $s|_\Lambda = s$,
- $f(t_1, \ldots, t_n)|_{ip} = t_i|_p$.

Un sous-terme à la position p est dit *propre* si $p \neq \Lambda$.

Le *remplacement* de s par t à la position p (noté $s[t]_p$) est défini par :

- $s[t]_\Lambda = t$,
- $f(t_1, ..., t_n)[t]_{ip} = f(t_1, ..., t_i[t]_p, ..., t_n)$.

Le terme $t = f(f(a, x), g(h(x, b, y)))$ est représenté sous forme d'arbre à la figure II.1. On peut y lire en particulier que $\Lambda(t) = f$ et que le sous-terme de t à la position 21 est $h(x, b, y)$.

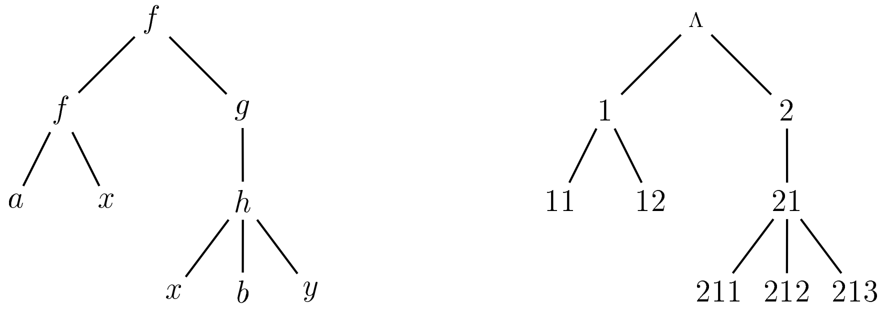


FIG. II.1. Terme et positions dans le terme représenté sous forme d'arbre.

Définition 9. — Un *contexte* C est un terme contenant une ou plusieurs occurrences d'une constante spéciale $\Box \notin \mathcal{F}$ appelée *trou*[1].

Dans le cas où $C[\dots]$ est un contexte avec n occurrences de $\Box$ (à des positions éventuellement précisées) et où $t_1, \dots, t_n$ sont n termes, $C[t_1, \dots, t_n]$ désigne le remplacement des occurrences de $\Box$ par les t_i aux positions précisées le cas échéant et, sinon, respectivement dans l'ordre lexicographique sur les positions.

Définition 10. — Étant donnée une algèbre de termes $T(\mathcal{F}, X)$, une *substitution* est une application σ de l'ensemble X des variables vers les termes. On appellera $\{x \in X \mid \sigma(x) \neq x\}$ le *support* de la substitution.

Nous décrirons éventuellement une substitution à l'aide de son support : $\{x \mapsto t_1 \ ; \ y \mapsto t_2\}$, par exemple, désignera σ de support $\{x \ ; \ y\}$ telle que $\sigma(x) = t_1$ et $\sigma(y) = t_2$.

L'application d'une substitution est usuellement écrite en notation postfixée et associative à gauche, $x\sigma$ désignant $\sigma(x)$.

Une substitution peut facilement être étendue comme endomorphisme de $T(\mathcal{F}, X)$ en posant simplement :

$$f(t_1, \dots, t_n)\sigma = f(t_1\sigma, \dots, t_n\sigma).$$

On dit alors d'un terme $t\sigma$ qu'il est une *instance* de t.

Une substitution est dite *close* si l'image de son support ne contient que des termes clos.

On représente parfois les termes sous la forme de triangles (dont le sommet supérieur correspond à la position du symbole de tête) et les instanciations comme des ajouts à la base du triangle.

Définition 11. — On dit qu'un terme s *filtre* un terme t s'il existe une substitution σ telle que $s\sigma = t$.

[1] C'est donc un élément de $T(\mathcal{F} \cup \{\Box\}, X)$ mais pas un terme de $T(\mathcal{F}, X)$.

On dit que deux termes s et t sont *unifiables* s'il existe une substitution σ (alors appelée *unificateur*) telle que $s\sigma = t\sigma$.

Le filtrage et l'unification sont décidables ; deux termes unifiables possèdent en outre un unificateur le plus général (mgu), unique à renommage des variables près.

Exemple 2.

Pour définir l'addition sur les entiers naturels écrits en base 2, on peut utiliser la signature $\mathcal{F}_{\text{bin}} = \{\# : 0_{\mathbb{N}} \,;\; 0 : 1_{\mathbb{N}} \,;\; 1 : 1_{\mathbb{N}} \,;\; + : 2_{\mathbb{N}}\}$ *où les symboles* 0 *et* 1 *sont postfixes et où* $+$ *est infixe.*

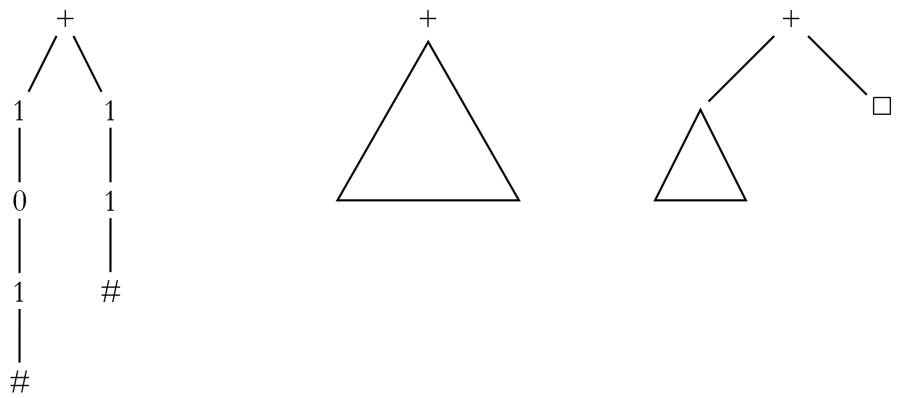
$\#$ *dénote en fait* $0_{\mathbb{N}}$, $(x)0$ *dénote* $2 \times x$, $(x)1$, $2 \times x + 1$ *et* $+$, *l'addition.*

L'entier 5 sera ainsi représenté par $(((\#)1)0)1$ *et l'addition de 5 et de 3 par le terme* $t = (((\#)1)0)1+(((\#)1)1)$. *En particulier* $\Lambda(t) = +$.

En posant $t_2 = t|_2 = (((\#)1)1)$, *on peut écrire* $t = C[t_2]_2$ *où* $C = (((\#)1)0)1+\square$.

Tout ceci est exprimé graphiquement par :

Arbre t *détaillé* — *Arbre* t *peu détaillé* — *Contexte* C *peu détaillé*



II.2.2 Ordres sur les termes

La notion d'« ordering pairs » a été introduite par Kusakari, Nakamura et Toyama en 1999. Elle permet une distinction claire des propriétés requises pour un ordre devant à terme servir aux preuves de terminaison. Plutôt que la traduction littérale « paires d'ordres » nous utiliserons dans la suite la désignation un peu abusive d'*ordres sur les termes*, plus proche de l'intuition de la relation proprement dite.

Définition 12. — Un *ordre sur les termes* (aussi connu comme *ordering pair* [54]) est une paire de relations sur les termes $(\succeq, \succ)$ telle que :

1. $\succeq$ est un préordre ;
2. $\succ$ est un ordre strict ;
3. $\succ \circ \succeq \subseteq \succ$ et $\succeq \circ \succ \subseteq \succ$.

On dit d'un ordre sur les termes $(\succeq, \succ)$ qu'il est :

- *Bien fondé* si $\succ$ est bien fondé ;
- *Stable* si $s \succeq t$ entraîne $s\sigma \succeq t\sigma$ et si $s \succ t$ entraîne $s\sigma \succ t\sigma$ pour toute substitution σ ;

– *(Faiblement) monotone* si $s \succeq t$ implique $C[s] \succeq C[t]$ pour tout contexte C ;
– *Strictement monotone* s'il est monotone et que $s \succ t$ implique $C[s] \succ C[t]$ pour tout contexte C.

Il est important de remarquer ici que l'intuition de la comparaison stricte de deux termes vis-à-vis d'un préordre $\geq$ ne correspond pas dans le cas général à l'ordre strict associé $>$. Cette partie stricte n'est en effet pas stable par instanciation des termes. L'exemple suivant dû à Enno Ohlebusch illustre cette particularité.

Exemple 3.
Considérons la signature $\mathcal{F} = \{f\ :\ unaire\ ;\ a\ :\ constante\}$ et le système $R(\mathcal{F})$ réduit à une seule règle $f(x) \to f(a)$. Nous pouvons définir le préordre $\geq$ sur $T(\mathcal{F}, X)$ par $\geq = \to_R^\star$. Pour cette relation $f(x) \geq f(a)$ et $f(a) \not\geq f(x)$, donc par définition de la partie stricte, $f(x) > f(a)$. Pourtant, en appliquant la substitution $\sigma = \{x \mapsto a\}$, nous obtenons $f(x)\sigma = f(a) \geq f(a) = f(x)\sigma$, perdant ainsi le caractère strict de l'orientation.

Afin d'obtenir une relation stable sous l'application de substitutions, on définit la notion de partie *stricte stable*.

Définition 13. — Soit $\geq$ un préordre sur $T(\mathcal{F}, X)$ de partie stricte $>$. La partie *stricte stable* $\succ$ de $\geq$ est définie par : $s \succ t$ si et seulement si $s\sigma > t\sigma$ pour toute substitution close σ sur n'importe quelle extension de $\mathcal{F}$.

Deux classes d'ordres sur les termes nous intéresseront tout particulièrement du fait de leurs propriétés caractéristiques.

Définition 14. — Un ordre sur les termes $(\succeq, \succ)$ est un ordre de

– *Réduction* s'il est bien fondé, monotone et stable ; on dira d'un tel ordre qu'il est de réduction *stricte* s'il est strictement monotone, de réduction *faible* s'il est (faiblement) monotone ;
– *Simplification* s'il est stable, strictement monotone et possède la propriété de sous-terme : $C[t] \succ t$.

Dans toute la suite, lorsque nous aurons à comparer des éléments d'une algèbre $T(\mathcal{F}, X)$, nous considérerons des *ordres sur les termes* (au sens de la définition 12) que nous désignerons par leur premier élément $\geq$ et dont le second élément, noté $>$, sera la partie stricte stable de $\geq$.

Extensions d'ordres sur les termes

De nouveaux ordres sur les termes peuvent être obtenus par composition lexicographique de deux ordres $(\geq^1, >^1)$ et $(\geq^2, >^2)$.

Définition 15. — On définit la *composition lexicographique* $((\geq^1, >^1), (\geq^2, >^2))_{\mathsf{lex}} = (\geq, >)$ de deux ordres sur les termes $(\geq^1, >^1)$ et $(\geq^2, >^2)$ de la manière suivante :

- $s > t$ si $s >^1 t$ ou si $s \geq^1 t$ et $s >^2 t$;
- $s \geq t$ si $s >^1 t$ ou si $s \geq^1 t$ et $s \geq^2 t$.

La paire $(\geq, >)$ ainsi définie est bien un ordre sur les termes.

Proposition 2.
Si $(\geq^1, >^1)$ est strictement monotone et si $(\geq^2, >^2)$ est faiblement monotone, alors la composition $((\geq^1, >^1),(\geq^2, >^2))_{\mathsf{lex}}$ est faiblement monotone.

Si $(\geq^1, >^1)$ et $(\geq^2, >^2)$ sont tous deux strictement monotones, alors $((\geq^1, >^1), (\geq^2, >^2))_{\mathsf{lex}}$ est strictement monotone.

PREUVE. Pour la première implication et pour le cas de deux termes s et t tels que $s \geq t$ où $(\geq, >) = ((\geq^1, >^1), (\geq^2, >^2))_{\mathsf{lex}}$. Soit C un contexte à un trou.

– Soit $s >^1 t$, par stricte monotonie de $(\geq^1, >^1)$: $C[s] >^1 C[t]$ et donc $C[s] \geq C[t]$;
– Soit $s \geq^1 t$ et $s >^2 t$, par monotonie de $(\geq^1, >^1)$ et $(\geq^2, >^2)$: $C[s] \geq^1 C[t]$, $C[s] \geq^2 C[t]$ et donc $C[s] \geq C[t]$;
– Soit $s \geq^1 t$ et $s \geq^2 t$, par monotonie de $(\geq^1, >^1)$ et $(\geq^2, >^2)$: $C[s] \geq^1 C[t]$, $C[s] \geq^2 C[t]$ et donc $C[s] \geq C[t]$.

□

Il est enfin possible d'étendre une relation d'ordre sur les termes à une relation d'ordre sur les multiensembles de termes.

Définition 16. — Soit $(\geq, >)$ un ordre sur les termes. On définit inductivement l'*extension lexicographique de* $(\geq, >)$ sur les suites de termes comme suit :

$$
\begin{array}{llll}
[e_1, \ldots, e_n] & >^{\mathsf{lex}} & [\,] & \text{si } n >_{\mathbb{N}} 0\,; \\
[e_1, \ldots, e_n] & >^{\mathsf{lex}} & [e'_1, \ldots, e'_m] & \text{si } e_1 > e'_1\,; \\
[e_1, e_2, \ldots, e_n] & >^{\mathsf{lex}} & [e'_1, e'_2, \ldots, e'_m] & \text{si } e_1 = e'_1 \text{ et } [e_2, \ldots, e_n] >^{\mathsf{lex}} [e'_2, \ldots, e'_m]\,; \\
[e_1, \ldots, e_n] & \geq^{\mathsf{lex}} & [\,] & \text{si } n \geq_{\mathbb{N}} 0\,; \\
[e_1, e_2, \ldots, e_n] & \geq^{\mathsf{lex}} & [e'_1, e'_2, \ldots, e'_m] & \text{si } e_1 > e'_1\,; \\
[e_1, e_2, \ldots, e_n] & \geq^{\mathsf{lex}} & [e'_1, e'_2, \ldots, e'_m] & \text{si } e_1 \geq e'_1 \text{ et } [e_2, \ldots, e_n] \geq^{\mathsf{lex}} [e'_2, \ldots, e'_m].
\end{array}
$$

Définition 17. — Un *multiensemble* est une application M_E d'un ensemble E vers $\mathbb{N}$ telle que $\{e \in E \mid M_E(e) \neq 0\}$ est fini. On appelle $M_E(e)$ la *multiplicité* de e dans M.

Les multiensembles sont souvent notés en extension entre accolades, chaque e apparaissant en $M_E(e)$ occurrences. En l'absence d'ambiguïté on pourra omettre la précision de l'ensemble E.

On dit qu'un élément e appartient à un multiensemble M (ce qu'on note alors $e \in M$) si $M(e) \geq 1$.

Soient M et N deux multiensembles ; M est *inclus* dans N (notation $M \subseteq N$) si pour tout e, $M(e) \leq N(e)$.

Enfin $M' = M \setminus N$ est défini par $M'(e) = Max(0, M(e) - N(e))$.

Définition 18. — Soient M et N deux multiensembles de termes. L'*extension multiensemble* d'un ordre sur les termes $(\geq, >)$ est un couple $(\geq_{\mathsf{mul}}, >_{\mathsf{mul}})$ défini inductivement par :

– $M \geq_{\mathsf{mul}} M$;
– Si $M \geq_{\mathsf{mul}} N$ et si $e \geq e'$ alors $M \cup \{e\} \geq_{\mathsf{mul}} N \cup \{e'\}$;
– Si $M \geq_{\mathsf{mul}} N$ et si $e > e_1, \ldots,\ e > e_k$ pour $k \geq 0$ alors $M \cup \{e\} >_{\mathsf{mul}} N \cup \{e_1\,; \ldots;\ e_k\}$;
– Si $M >_{\mathsf{mul}} N$ et si $e \geq e'$ alors $M \cup \{e\} >_{\mathsf{mul}} N \cup \{e'\}$.

II.2.3 Récriture de termes

Définition 19. — Une *relation de récriture* est une relation $\rightarrow$ sur les termes, monotone et stable par instanciation. On note $\rightarrow^+$ sa clôture transitive et $\rightarrow^\star$ sa clôture réflexive/transitive.

Puisqu'il est impossible de représenter en extension une relation de récriture éventuellement infinie, on se ramène à un représentant canonique de cette relation : le *système de récriture*, souvent abrégé en TRS (de l'anglais « Term Rewriting System »).

Définition 20. — Soit $T(\mathcal{F}, X)$ une algèbre de termes. Une *règle de récriture* sur $\mathcal{F}$ est un couple $l \rightarrow r$ de termes de $T(\mathcal{F}, X)$ tel que l n'est pas une variable et tel que $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Un *système de récriture* $R(\mathcal{F})$ est un ensemble R de règles sur $\mathcal{F}$. Si aucune confusion n'est possible quant à la signature de référence, on notera abusivement R pour $R(\mathcal{F})$. Le système R définit la relation de récriture $\rightarrow_R$ comme suit :

$s \rightarrow_R t$ s'il existe une règle $l \rightarrow r \in R$, une position p et une substitution σ telles que

$$s|_p = l\sigma \ \text{ et } \ t = s[r\sigma]_p.$$

On dit alors que $s|_p$ est un *radical* et que s *se réduit en* t *par la règle* $l \rightarrow r$ *à la position* p, ce qui est noté :

$$s \xrightarrow[l \rightarrow r]{p,\sigma} t.$$

Si de telles précisions ne sont pas nécessaires, on peut omettre la substitution, la position ou encore remplacer la désignation de la règle utilisée par celle du système :

$$s \xrightarrow[R]{p} t$$

Permettant ainsi la confusion volontaire avec la désignation de la relation.

Une règle ne s'applique ainsi qu'à un sous-terme filtré par son membre gauche.

Remarque 1. — La représentation d'une relation de récriture par un système ne permet pas, dans le cas général, de décider de l'appartenance d'un couple de termes donné à la relation[2].

Suivant la terminologie utilisée par Thomas Arts & Jürgen Giesl [3, 4] nous distinguerons dans la signature d'un système les symboles *définis* des symboles *constructeurs*.

Définition 21. — Soit R un système de récriture sur une signature $\mathcal{F}$. On définit la partition $\mathcal{F} = D \cup C$ telle que :

- $f \in D$ si et seulement s'il existe une règle $l \rightarrow r \in R$ telle que $\Lambda(l) = f$;
- $C = \mathcal{F} \setminus D$.

Les éléments de D sont les symboles *définis*, les éléments de C les symboles *constructeurs*.

Exemple 4.

Sur la signature de l'exemple 2 (en omettant le parenthésage par associativité à gauche de 1 *et* 0*) nous pouvons*

[2] C'est un problème d'accessibilité [15, 44]. Voir le paragraphe III.1.1 pour plus de détails.

proposer un système R_+ de calcul de l'addition, en introduisant une règle de simplification $\#0 \to \#$ puis celles de l'addition.

$$R_+ \left\{ \begin{array}{lll} \#0 & \to & \# \\ x{+}\# & \to & x \\ \#{+}x & \to & x \\ x0{+}y0 & \to & (x{+}y)0 \\ x0{+}y1 & \to & (x{+}y)1 \\ x1{+}y0 & \to & (x{+}y)1 \\ x1{+}y1 & \to & ((x{+}y){+}\#1)0 \end{array} \right. \begin{array}{r} (1) \\ (2) \\ (3) \\ (4) \\ (5) \\ (6) \\ (7) \end{array}$$

R_+ admet deux symboles définis 0 *et* $+$*, les constructeurs sont donc* $\#$ *et* 1.

Le terme $\#101{+}\#11$ *peut donner lieu à la réduction suivante (le symbole de tête du radical est cerclé de gris, nous précisons le numéro de la règle appliquée) :*

$$\begin{array}{llll} \#101 + \#11 \xrightarrow[(7)]{\Lambda} & ((\#10 + \#1){+}\#1)0 & \xrightarrow[(5)]{11} & ((\#1 + \#)1{+}\#1)0 \\ \xrightarrow[(2)]{111} & ((\#1)1 + \#1)0 & \xrightarrow[(7)]{1} & (((\#1 + \#){+}\#1)0)0 \\ \xrightarrow[(2)]{111} & ((\#1 + \#1)0)0 & \xrightarrow[(7)]{11} & ((((\# + \#){+}\#1)0)0)0 \\ \xrightarrow[(3)]{1111} & (((\# + \#1)0)0)0 & \xrightarrow[(3)]{111} & ((((\#)1)0)0)0. \end{array}$$

Le terme $\#1000$ *ne peut être réduit et représente bien* $8 = 5 + 3$.

II.2.4 Caractéristiques des systèmes de récriture

Une réduction consistant essentiellement en des remplacements d'expressions syntaxiques par d'autres, les caratéristiques syntaxiques des systèmes donnent tout naturellement certaines informations quant aux diverses propriétés — directement syntaxiques mais aussi sémantiques — qu'ils peuvent avoir.

Définition 22. — Un système est dit *à branchement fini* si tout radical n'est filtré que par un nombre fini de membres gauches de règles.

Définition 23. — On dit d'un terme qu'il est *linéaire* si chacune de ses variables n'apparaît qu'une seule fois.

Une règle est *linéaire à gauche* si son membre gauche est linéaire. Symétriquement une règle est dite *linéaire à droite* si son membre droit est linéaire. Enfin, une règle *linéaire* est une règle à la fois linéaire à gauche et à droite.

La non-préservation du nombre d'occurrences des variables peut-être due, entre autres, à une caractéristique *projective* ou *dupliquante* du système.

Définition 24. — Une règle dont une variable apparaît strictement plus souvent dans le membre droit que dans le membre gauche est une règle *dupliquante*.

Un système *dupliquant* est un système comportant des règles dupliquantes.

Définition 25. — Si un membre droit de règle est réduit à une variable, on dit de cette règle qu'elle est *projective*.

Si en outre le membre gauche peut être réduit en plusieurs variables distinctes (par différentes règles) on dit du système constitué de ces règles qu'il effectue une *projection non déterministe* (projection ND). Un tel système est projectif ND.

Exemple 5.

Considérons le système constitué de cinq règles :

$$\left\{\begin{array}{lcll} f(x,y,z) & \to & g(y,y) & (1) \\ f(x,y,y) & \to & g(x,y) & (2) \\ g(x,y) & \to & y & (3) \\ h(x) & \to & x & (4) \\ f(x,y,z) & \to & h(x). & (5) \end{array}\right.$$

La règle (1) *est linéaire* à gauche *sans être linéaire à droite ;* (2) *est quant à elle linéaire* à droite *mais pas à gauche ; Les règles* (3), (4) *et* (5) *sont* linéaires *car linéaires à gauche* et *à droite.*

On peut remarquer que (1) *est* dupliquante *et que* (3) *et* (4) *sont* projectives.

Le terme $f(x,y,z)$ *est réductible à la fois*

- *En* x *en appliquant* (5) *puis* (4) *;*
- *En* y *à l'aide d'une réduction par* (1) *suivie d'un pas de* (3).

Ce système a donc un comportement projectif ND.

Un système définit un calcul ; un résultat de ce calcul est un terme sur lequel ce calcul ne se poursuit pas.

Définition 26. — Un terme qui ne peut être réduit par un système R est *en forme normale* (pour R). On dira d'un terme qu'il est *normalisable (par R)* s'il admet une forme normale pour R c'est-à-dire s'il existe une réduction qui aboutit à une forme normale. Ce terme sera dit *fortement normalisable (par R)* si *toute* réduction mène à une forme normale.

Un système R est *normalisant* si tout terme est normalisable. Il est *fortement normalisant* (on dit aussi qu'il *termine*) lorsque la relation $\to_R$ est noethérienne, c'est-à-dire que tout terme est fortement normalisable.

Exemple 6.

Sur la signature $\mathcal{F} = \{f : 1; a : 0\}$, *le système*

$$R : \left\{\begin{array}{l} f(x) \to a \\ f(x) \to f(x) \end{array}\right.$$

Donne à tout terme une forme normale : a. *Il ne termine cependant pas puisque* $f(x)$ *peut être réduit indéfiniment.*

Le système restreint à la seule règle $f(x) \to a$ *est lui, en revanche, fortement normalisant.*

L'application non-déterministe des règles ne laisse rien supposer sur l'unicité du résultat éventuel ni sur celle du calcul menant à un résultat. La propriété de *confluence* d'une relation peut alors ôter toute importance aux choix faits lors du calcul en garantissant que chacun d'eux peut être compensé par la suite.

Définition 27. — Soient $l_1 \to r_1$ et $l_2 \to r_2$ deux règles d'un système R telles que l_1 est unifiable avec un sous-terme (non réduit à une variable) de l_2 (mais pas à la racine si $l_1 \to r_1 = l_2 \to r_2$). La paire de termes $(C[r_1]\sigma, r_2\sigma)$ où σ est un unificateur le plus général tel que $l_2 = C[u]$ et $u\sigma = l_1\sigma$ est appelée une *paire critique* de R.

On dit d'un système qui admet des paires critiques qu'il est *overlapping*.

Les paires critiques sont en fait définies modulo renommage des variables. Autrement dit, avant de superposer des règles, on renomme leurs variables pour en obtenir des ensembles disjoints.

Définition 28. — Si pour un système R, $t_1 \leftarrow s \to t_2$ implique l'existence d'un s' tel que $t_1 \to^\star s'$ et $t_2 \to^\star s'$, on dit que R est *localement confluent*. Il est *confluent en un pas* si $t_1 \to s'$ et $t_2 \to s'$. Un système est enfin *confluent* si sa clôture réflexive/transitive est localement confluente.

La confluence en un pas est décidable ; elle implique la confluence locale pour les systèmes fortement normalisants.

La confluence est une propriété indécidable en général ; elle est toutefois entraînée par la confluence locale dans le cas des systèmes fortement normalisants[3].

Si un système P est fortement normalisant et tel que tout terme t a une unique forme normale pour P, nous désignerons celle-ci par $t{\downarrow_P}$.

Définition 29. — Un système *orthogonal* est un système linéaire à gauche sans paire critique.

Un système est *faiblement orthogonal* s'il n'admet que des paires critiques triviales, c'est-à-dire que pour toute paire critique (s, t), s est identique à t.

L'intérêt porté aux systèmes orthogonaux est dû au fait remarquable qu'ils sont tous confluents [76] et qu'on peut connaître certaines de leurs propriétés en étudiant seulement la relation qu'ils induisent en suivant une *stratégie* d'application particulière (cf. théorème 1, page 25).

II.2.5 Stratégies de réduction

On peut restreindre la relation de récriture en affaiblissant le non-déterminisme, c'est-à-dire en favorisant voire en imposant la ou les règles à utiliser parmi toutes celles qui sont applicables à un terme. On définit alors une stratégie de réduction.

Dans le cas de la récriture normalisée, par exemple, la réduction par un ensemble de règles R d'un terme n'est autorisée que si celui-ci est en forme normale pour un (autre) ensemble de règles P. Nous nous intéresserons plus particulièrement à une stratégie proche de l'évaluation de langages fonctionnels : la stratégie innermost.

Définition 30. — La stratégie de réduction telle que $s \xrightarrow{p} t$ seulement si tout sous-terme propre de $s|_p$ est en forme normale est appelée stratégie *innermost*.

Cette stratégie pose la normalisation des arguments (l'anglais « innermost » peut en fait être traduit littéralement par « le plus interne ») comme préalable obligatoire à l'évaluation d'une fonction et correspond donc en ce sens à l'*évaluation stricte* de langages comme OCAML [67] ou SML [64] en opposition à l'*évaluation paresseuse* à la HASKELL [41].

[3] C'est le fameux lemme de Newman.

En restreignant le choix des radicaux utilisables, le choix de cette stratégie peut accorder aux systèmes concernés des propriétés qu'ils n'auraient pas dans le cas général. Pour certains systèmes, il est possible d'obtenir grâce à elle des résulats qu'on peut étendre à ces systèmes considérés sans stratégie.

On a en particulier pour les systèmes orthogonaux le résultat suivant d'O'Donnell [68] :

Théorème 1. (O'Donnell 77)

Un système orthogonal est fortement normalisant si et seulement s'il est fortement normalisant pour la stratégie de réduction innermost.

Exemple 7.

Considérons le système :

$$R \;:\; \begin{cases} f(g(x)) & \to \quad f(g(x)) \\ g(x) & \to \quad a. \end{cases}$$

Puisqu'il admet en particulier une paire critique $(f(g(x)), a)$*,* R *n'est pas orthogonal.*

S'il autorise dans le cas standard la réduction infinie $f(g(x)) \to f(g(x)) \to \dots$*, un tel système termine toutefois sur tout terme pour la stratégie innermost. Celle-ci va en effet faire disparaître tous les symboles* g *nécessaires au radical* $f(g(x))$ *et à l'intérieur de celui-ci.*

Chapitre III

Terminaison

III.1 Problème de l'arrêt et indécidabilité

La terminaison, garantie de l'existence d'un résultat du calcul sur tout terme et donc en particulier de la totalité des fonctions définies par des systèmes de récriture, préalable incontournable à la recherche d'autre propriétés, n'admet hélas pas d'algorithme de décision. Le problème de l'arrêt des machines de Turing s'y réduit en effet.

III.1.1 Récriture et machines de Turing

Nous allons voir comment coder n'importe quelle machine de Turing sous forme d'un système de règles. Dershowitz a proposé à ce sujet un codage en deux règles seulement [21], nombre encore réduit par Dauchet : il est en fait possible de se limiter à une seule règle linéaire à gauche [15].

Nous donnons ici une traduction simple, inspirée des travaux de Huet & Lankford [44] en quelques règles de récriture de mots[1], un choix naturel puisque les machines de Turing travaillent sur des rubans.

Considérons une machine de Turing $(Q, q_0, A = \{0, 1\}, b, \delta)$ où Q est l'ensemble des états, q_0 l'état initial, 0 et 1 les deux lettres de l'alphabet de travail A, b le caractère de blanc et δ la fonction de transition de $Q \times A$ vers $Q \times A \times \{\blacktriangleleft, \blacktriangleright\}$ qui à un état et à un caractère lu associe un nouvel état, un caractère écrit et un déplacement vers la droite $\blacktriangleright$ ou la gauche $\blacktriangleleft$ sur le ruban borné à gauche.

Afin d'exprimer les configurations successives d'une machine de Turing dans le formalisme de la récriture sur les mots, nous considérons les mots sur la signature $\mathcal{F} = Q \cup A \cup \{b\}$.

Dans un mot $q_i\alpha$, q_i représente l'état actuel de la machine et α le caractère sur lequel pointe la tête. Les transitions peuvent donc être codées de la manière suivante :

$$\begin{array}{rll} q_i\alpha \to \beta q_j & & \text{si} \quad \delta(q_i, \alpha) = (q_j, \beta, \blacktriangleright), \\ \left.\begin{array}{r} 0q_i\alpha \to q_j 0\beta \\ 1q_i\alpha \to q_j 1\beta \end{array}\right\} & & \text{si} \quad \delta(q_i, \alpha) = (q_j, \beta, \blacktriangleleft). \end{array}$$

Il reste à ajouter les cas dégénérés, c'est-à-dire faisant intervenir le caractère blanc, pour obtenir la simulation désirée.

On obtient ainsi une correspondance entre les configurations successives de la machine et les mots obtenus par réduction à l'aide du système.

[1]La récriture de mots se réduit en fait à une classe particulière de la récriture de termes restreinte aux termes monadiques.

Problème de l'arrêt. S'il existait un moyen de décider de la terminaison d'un système de récriture, on aurait un algorithme de décision de l'arrêt d'une machine de Turing simplement en composant la traduction des machines vers les systèmes et la procédure de décision sur iceux. Puisqu'on lui ramène ainsi le problème de l'arrêt des machines de Turing, le problème de la terminaison des systèmes de récriture est indécidable.

Accessibilité. Le problème de l'accessibilité consiste en la question suivante : étant donnés deux termes s et t, existe-t-il une séquence de réductions qui, à partir de s, aboutit à t ? Il se résume en fait au problème de l'arrêt : il suffit de réduire toute configuration d'une machine arrêtée à un mot bien précis, un nouvel état caractéristique q_s par exemple, en ajoutant cet état à Q et les règles de simplification :

$$\begin{array}{ll} q_s 0 \to q_s, & q_s 1 \to q_s, \\ 0 q_s \to q_s, & 1 q_s \to q_s, \end{array}$$

Ainsi que, pour tout $q \in Q$ et $\alpha \in \{0,1\}$ les règles de réduction :

$$\begin{array}{rl} q\alpha \to q_s\alpha & \text{si } \delta(q,\alpha) \text{ n'est pas défini,} \\ bq\alpha \to bq_s\alpha & \text{si } \delta(q,\alpha) = (q', \beta, \blacktriangleleft). \end{array}$$

L'accessibilité d'une configuration quelconque à bq_sb est alors une expression de l'arrêt.

III.2 Preuve de terminaison

La terminaison se révélant être une propriété indécidable, on s'intéresse au développement de méthodes correctes mais forcément incomplètes permettant de prouver la normalisation forte dans la plupart des cas de systèmes rencontrés dans la pratique.

D'une manière générale on montre la terminaison d'un programme en prouvant qu'au cours d'une exécution, un variant de celui-ci décroît strictement vis-à-vis d'un ordre bien fondé. De par la nature de l'ordre, ce variant ne peut décroître indéfiniment, l'exécution ne s'éternise donc pas : le programme s'arrête sur un résultat.

Exemple 8.

Pour le programme décrit par la seule règle : $f(f(x)) \to x$, il suffit de prendre pour variant le nombre de symboles f du terme à réduire à chaque pas.

Dans un formalisme de programme quelconque, la recherche de ce variant peut reposer très lourdement sur l'expérience et l'intuition de qui cherche à montrer la terminaison. En ce qui concerne les systèmes de récriture, la preuve peut toutefois être conceptuellement simplifiée par le fait que le système R décrit lui-même une *relation* entre les termes. Si cette relation $\to_R$ peut être plongée dans une relation d'ordre bien fondée, la propriété 1 entraîne alors que $\to_R$ est bien fondée.

On peut remarquer en particulier qu'un système R, s'il termine, décrit une relation $\to_R$ bien fondée par définition : c'est l'*ordre de récriture*. Cet ordre est à rapprocher du variant associé au programme qui peut, puisque le programme termine par hypothèse, être la différence entre le nombre de pas maximal possible sur l'exécution considérée et le nombre de pas déjà effectués.

Dans le cadre de cette approche plus générale, la recherche ne porte donc plus sur une mesure de complexité qui décroît au fur et à mesure des applications de R mais directement sur un ordre bien fondé contenant $\to_R$.

Exemple 9.

En reprenant dans ce contexte l'exemple 8, on peut montrer que le système $\{f(f(x)) \to x\}$ *termine dans la mesure où, en considérant le nombre de* f *dans le terme avec l'ordre naturel sur les entiers, pour toute instanciation* σ*,* $f(f(x))\sigma$ *est supérieur à* $x\sigma$*.*

Savoir si un couple de termes fait partie de $\to_R^\star$ demeure toutefois indécidable (cf. remarque 1). On peut cependant restreindre l'ensemble des ordres candidats à ceux qui, étant bien fondés, contiennent déjà les règles de R et sont en outre stables par instanciation et monotones : les ordres de réduction (déf. 14).

Théorème 2.

Un système R termine si et seulement s'il existe un ordre de réduction $\succ$ tel que $\to_R \subseteq \succ$.

PREUVE. Immédiat puisqu'en particulier pour un système R fortement normalisant, l'ordre de récriture est un ordre de réduction. □

Un ordre apte à prouver la terminaison d'un système de récriture doit donc se plier à certaines *contraintes*. La recherche d'une preuve de terminaison revient alors dans la plupart des cas à déterminer ces contraintes d'ordre et à les résoudre.

La génération de ces contraintes s'est historiquement d'abord limitée à l'orientation stricte des règles du système étudié. On cherchait alors à trouver directement un ordre convenable pour le théorème 2 à l'aide de ce que nous désignerons dans la suite comme les « Méthodes classiques ».

Une analyse plus fine de la structure des termes non fortement normalisables par un système R a donné le jour en 1997 aux critères de terminaison à l'aide de *paires de dépendance*. Les contraintes requises, pour plus nombreuses qu'elles puissent être, deviennent moins draconiennes : il suffit d'orienter *largement* les règles de R, la décroissance stricte n'étant nécessaire que sur lesdites paires de dépendance.

III.3 Méthodes classiques

III.3.1 Terminaison à l'aide d'ordres de simplification

L'importance des travaux consacrés aux ordres de simplification, qu'ils concernent la définition de nouveaux ordres ou l'étude de leurs propriétés, vient en particulier du fait que parmi les ordres susceptibles d'être utilisés dans les preuves de terminaison ils étaient jusqu'à peu les seuls qu'on pouvait déterminer automatiquement.

Théorème 3. (Dershowitz 82) [20]

Sur une signature finie, tout ordre de simplification est bien fondé.

En particulier s'il existe un ordre de simplification $>$ tel que pour toute règle $l \to r$ d'un système R on ait $l > r$ alors R termine.

Définition 31. — L'ordre dit *de plongement* $\trianglelefteq$ est défini par :
$t = f(t_1, \dots, t_n) \trianglelefteq g(t'_1, t'_1 \dots, t'_m) = t'$ si

1. Il existe i, $1 \leq i \leq m$ tel que $t \trianglelefteq t'_i$, ou bien
2. $f = g$ et pour tout j, $1 \leq j \leq n$, $t_j \trianglelefteq t'_j$.

Remarque 2. — Le plongement, bien fondé par le théorème de Kruskal, est un ordre de simplification et tout ordre de simplification le contient.

Deux termes sont en fait en relation par l'ordre de plongement s'il est possible d'obtenir l'un à partir de l'autre simplement à l'aide de projections. On peut donc considérer les systèmes qui préservent la terminaison quand on autorise le passage au sous-terme.

Définition 32. — On dit d'un système R qu'il *termine simplement* si

$$R \bigcup_{f \mid \mathrm{AR}(f) \geq 1} \{f(x_1, \ldots, x_n) \to x_i \mid 1 \leq i \leq n\} \text{ est fortement normalisant.}$$

Cette définition est bien équivalente à une preuve de terminaison par le théorème 2 et à l'aide d'un ordre de simplification :

Lemme 1. (Kurihara & Ohuchi) [51]
Un système sur une signature finie termine simplement si et seulement s'il existe un ordre de simplification $>$ tel que pour toute règle $l \to r \in R$ on ait $l > r$.

III.4 Critères des paires de dépendance

La démarche qui consiste à destiner les ordres que nous venons de voir à l'application directe du théorème 2 est trop restrictive. Arts & Giesl proposent en 1997 une analyse plus fine de la structure des termes non fortement normalisables qui permet de dégager de nouvelles contraintes d'ordres, plus nombreuses mais plus souples [2].

Ils constatent en effet qu'à partir de tout terme donnant lieu à une réduction infinie on peut obtenir une dérivation d'une forme bien particulière. Prouver qu'il n'existe pas de dérivation de la sorte, quel que soit le terme considéré, suffit alors — en montrant par là que tous les termes ne donnent lieu qu'à des réductions finies — à mettre en évidence la terminaison du système.

Remarquons que grâce à la plus grande souplesse de ces contraintes, on peut davantage en espérer une résolution automatique.

III.4.1 Paires de dépendance

Principe de base. Un argument de minimalité suffit pour montrer que si un terme t donne lieu à une réduction infinie, alors il admet un sous-terme non fortement normalisable $f(u_1, \ldots, u_n)$ tel que tous les u_i sont pourtant fortement normalisables.

On obtient ainsi l'existence d'une réduction particulière comme celle de la figure III.1.

Pour garantir la terminaison il reste à prouver qu'une telle dérivation ne peut se produire. Cette dérivation reposant essentiellement sur l'existence de sous-termes pouvant déclencher une réduction, c'est cette propriété que nous allons chercher à contenir dans de nouvelles contraintes.

Définition 33. — Considérons un système de récriture $R(\mathcal{F})$. Une paire $\langle l, s\rangle$ telle qu'il existe une règle $l \to r \in R$ où s est un sous-terme de r dont le symbole de tête $\Lambda(s)$ est défini est une *paire de dépendance de* $l \to r$.

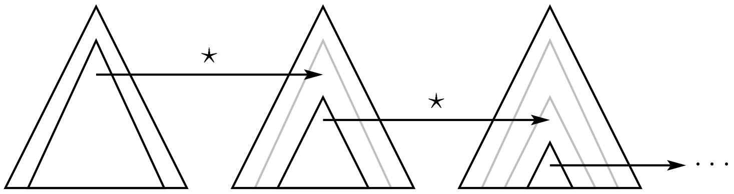


FIG. III.1. Structure de la réduction particulière.

L'union des paires de dépendance de toutes les règles de R forme l'ensemble des *paires de dépendance de* R, noté $\mathrm{DP}(R)$.

Il est en fait possible d'affiner cette définition en distinguant les symboles de tête des membres de paires. Cette différenciation est justifiée en particulier car la position marquée délimite le terme minimal non fortement normalisant dans la réduction particulière, figure III.1. Nous utilisons ici et sauf mention contraire ces paires « marquées » et préciserons le cas échéant leurs avantages et les difficultés techniques pouvant en résulter[2].

Définition 34. — Soit $R(\mathcal{F})$ un système de récriture. Pour chaque symbole $f \in \mathcal{F}$ défini dans R on considère sa *copie marquée* $\widehat{f}$ telle que $\widehat{f} \notin \mathcal{F}$.

On note :

$$\widehat{\mathcal{F}}_R = \mathcal{F} \cup \bigcup_{\substack{f \in \mathcal{F} \\ f \text{ défini dans } R}} \{\widehat{f}\},$$

La mention du système de récriture de référence étant omise en l'absence d'ambiguïté.

Si un terme s de $T(\mathcal{F}, X)$ a en tête un symbole défini, $\widehat{s}$ représente le terme de $T(\widehat{\mathcal{F}}, X)$ obtenu en remplaçant $\Lambda(s)$ par sa copie marquée. Nous appellerons *marquage* cette opération.

Définition 35. — Pour un système de récriture $R(\mathcal{F})$, une paire $\langle \widehat{l}, \widehat{s} \rangle$ telle qu'il existe une règle $l \to r \in R$ où s est un sous-terme de r dont le symbole de tête $\Lambda(s)$ est défini est une *paire de dépendance marquée de* $l \to r$.

Puisque nous ne considérons dans la suite quasiment que des paires marquées, nous les désignerons abusivement sous le nom de « paires de dépendance » respectant ainsi la nomenclature de Arts & Giesl.

Définition 36. — Une *chaîne de dépendance* d'un système R est une séquence $\ldots, \langle s_j, t_j \rangle, \ldots$ munie d'une substitution σ telle que pour deux paires $\langle s_i, t_i \rangle, \langle s_{i+1}, t_{i+1} \rangle$ consécutives quelconques :

$$t_i \sigma \xrightarrow[R]{\neq \Lambda}{}^{\star} s_{i+1} \sigma.$$

[2]Une attention toute particulière sera portée aux marques lors de notre étude de la terminaison des systèmes AC, cf. section III.2 de la deuxième partie.

Les paires de dépendance sont définies modulo renommage des variables. Nous considérerons toujours que, dans une chaîne de dépendance, les variables sont distinctes d'une paire à l'autre.

Les chaînes de dépendance permettent en particulier de décrire la réduction de forme particulière qui nous intéresse. En effet, dans une telle dérivation les pas de DP relient le terme minimal non fortement normalisant à un sous-terme de son réduit en tête, lui-même minimal et non fortement normalisant, comme illustré par la figure III.2.

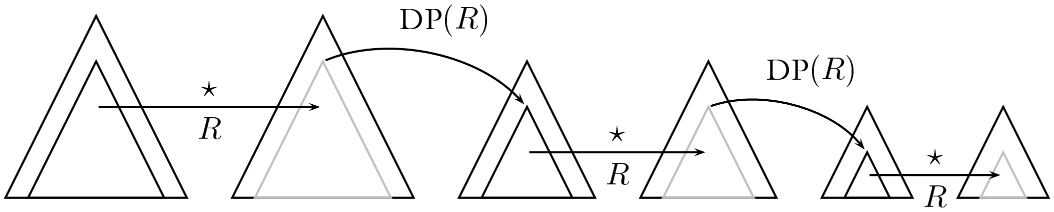


FIG. III.2. Réduction avec paires de dépendance.

Les chaînes de dépendance que nous rencontrerons dans la suite seront dites « minimales » dans le sens où elles correspondront toujours à la réduction particulière du principe de base (figure III.1).

Définition 37. — Une chaîne de dépendance est *minimale* si les sous-termes propres de toute instance d'un membre gauche de paire sont fortement normalisables.

Puisqu'elle est capable de décrire la réduction minimale particulière dont le caractère fini ou infini est celui de la relation $\rightarrow_R$, nous allons désormais nous intéresser à la nouvelle relation

$$\xrightarrow[R]{} \cup \xrightarrow[\mathrm{DP}(R)]{\Lambda} .$$

III.4.2 Critères de terminaison

Ainsi passés des dérivations aux chaînes, il nous reste à trouver un ordre bien fondé dans lequel plonger notre chaîne de dépendance minimale. Les contraintes sur les pas de récriture (et donc les règles) sont considérablement allégées : si règles et paires doivent encore décroître, il suffit désormais que seule la décroissance des pas de DP soit stricte, la terminaison étant alors assurée par le caractère fini des réductions entre instances de paires.

Les paires de dépendance permettent de définir un *critère* de terminaison, c'est-à-dire une condition à la fois nécessaire et suffisante pour garantir qu'un système est fortement normalisant.

Théorème 4. Complétude.
Si un système R est fortement normalisant, alors il n'existe aucune chaîne de dépendance infinie.

PREUVE. On peut en effet directement traduire en réduction infinie tout chaîne infinie :

$$\dots \langle \widehat{s}_i, \widehat{t}_i \rangle,\ \langle \widehat{s}_{i+1}, \widehat{t}_{i+1} \rangle, \dots \text{ munie de } \sigma$$

$$\dots C[s_i\sigma]_p \xrightarrow[R]{p} C[t_i\sigma]_p \xrightarrow[R]{}^{\star} C'[s_{i+1}\sigma]_q \xrightarrow[R]{q} C'[t_{i+1}\sigma]_q \xrightarrow[R]{}^{\star} \dots$$

□

Théorème 5. Correction.
Soit R un système de récriture. S'il n'existe aucune chaîne de dépendance de R infinie, alors R est fortement normalisant.

PREUVE. Nous commençons par donner formellement la clef du principe de base sous la forme d'un lemme. Afin d'alléger les notations nous utiliserons les caractères gras : $f(\mathbf{u})$ représentera ainsi de façon condensée le terme $f(u_1, \ldots, u_n)$ et une propriété énoncée sur $\mathbf{u}$ s'appliquera en fait à tous les u_i, $1 \leq i \leq n$.

Lemme 2.
Si un terme s est non fortement normalisable, alors on peut écrire s comme $C[f(\mathbf{u})]$ où $f(\mathbf{u})$ est non fortement normalisable mais $\mathbf{u}$ est fortement normalisable.

PREUVE. Par minimalité. □

Preuve du théorème de correction

Supposons qu'il existe une suite infinie de réductions à partir d'un terme t, nous allons construire à partir de celle-ci une chaîne de dépendance infinie.

D'après le lemme 2, t contient un sous-terme non fortement normalisable $f_1(\mathbf{u}_1)$ tel que $\mathbf{u}_1$ est fortement normalisable. On ne peut donc réduire indéfiniment $\mathbf{u}_1$. Il existe ainsi $\mathbf{v}_1$ tel que

$$f_1(\mathbf{u}_1) \stackrel{\neq\Lambda}{\longrightarrow}^{\star} f_1(\mathbf{v}_1)$$

Et $f_1(\mathbf{v}_1)$ est réductible à la racine.

Il y a donc une substitution σ telle que $f_1(\mathbf{v}_1) = f_1(\mathbf{w}_1)\sigma$ avec une règle $f_1(\mathbf{w}_1) \rightarrow r \in R$. Nous avons $f_1(\mathbf{w}_1)\sigma \rightarrow r\sigma$ avec, par le lemme, $r\sigma$ contenant un sous-terme $f_2(\mathbf{u}_2)$ non fortement normalisable.

Or pour toute variable x apparaissant dans $(\mathbf{w_1})$, $x\sigma$ est fortement normalisable, ce qui signifie que f_2 ne peut apparaître par σ et donc $r = C[f_2(\bar{\mathbf{u}}_2)]$ tel que $f_2(\bar{\mathbf{u}}_2)\sigma = f_2(\mathbf{u}_2)$ non fortement normalisable avec $\mathbf{u}_2$ fortement normalisable.

Nous pouvons ainsi écrire la paire $\langle \widehat{f_1}(\mathbf{w}_1), \widehat{f_2}(\bar{\mathbf{u}}_2)\rangle$ et la munir de la substitution σ.

Définissons maintenant la construction inductive des paires de dépendance.

Supposons que nous disposions d'une séquence : $\ldots \langle \widehat{f_{n-1}}(\mathbf{w}_{n-1}), \widehat{f_n}(\bar{\mathbf{u}}_n)\rangle, \sigma$. Nous avons ainsi la réduction $f_{n-1}(\mathbf{w}_{n-1})\sigma \rightarrow C[f_n(\bar{\mathbf{u}}_n)]\sigma$ avec $f_n(\bar{\mathbf{u}}_n)\sigma = f_n(\mathbf{u}_n)$ où $f_n(\mathbf{u}_n)$ est non fortement normalisable mais avec $\bar{\mathbf{u}}_\mathbf{n}$ fortement normalisable. Donc $f_n(\mathbf{u}_n) \rightarrow^{\neq\Lambda\,\star} f_n(\mathbf{v}_n)$ et il faut réduire à Λ.

Nous savons qu'il existe τ telle que $f_n(\mathbf{w}_n)\tau = f_n(\mathbf{v}_n)$ et $f_n(\mathbf{w}_n) \rightarrow g \in R$. Mais $f_n(\mathbf{w}_n)\tau$ se récrit en $g\tau$ qui est source d'une réduction infinie ; en appliquant le lemme 2, $g\tau$ s'écrit $C'[h(\mathbf{u})]$. Or pour tout x variable de $(\mathbf{w_n})$, $x\tau$ est fortement normalisable donc $g\tau = C[h(\bar{\mathbf{u}})]\tau$.

En posant $h = f_{n+1}$ et $\bar{\mathbf{u}} = \bar{\mathbf{u}}_{n+1}$ nous obtenons ainsi une nouvelle paire de dépendance.

Il reste à vérifier que

$$\ldots \langle \widehat{f_{n-1}}(\mathbf{w}_{n-1}), \widehat{f_n}(\bar{\mathbf{u}}_n)\rangle, \langle \widehat{f_n}(\mathbf{w}_n), \widehat{f_{n+1}}(\bar{\mathbf{u}}_{n+1})\rangle$$

Munie de $\sigma \circ \tau$ est bien une chaîne de dépendance, ce qui est vrai car pour récrire, on peut toujours renommer les variables (les variables de deux paires différentes sont distinctes). □

Ce théorème de correction admet un corollaire définissant de nouvelles contraintes d'ordre.

Corollaire 5–1.

Soit R un système de récriture, s'il existe un ordre de réduction faible (au sens de la définition 14) sur les termes $(\succeq, \succ)$ tel que :

1. $l \succeq r$ pour toute règle $l \to r \in R$ et
2. $s \succ t$ pour toute paire $\langle s, t\rangle \in \mathrm{DP}(R)$,

Alors R est fortement normalisant.

PREUVE. On a en effet $\to_R \subseteq \succeq$. S'il existait une chaîne de dépendance infinie, on aurait par définition une substitution σ telle que

$$t_i\sigma \xrightarrow[R]{\neq\Lambda}{}^{\star}\, s_{i+1}\sigma$$

Pour tout i et donc une suite infinie

$$s_i\sigma \succ t_i\sigma \succeq s_{i+1}\sigma \succ t_{i+1}\sigma \succeq \dots$$

Ce qui serait en contradiction avec l'hypothèse : $\succ$ bien fondé. □

III.4.3 Graphes de dépendance

La condition de stricte décroissance sur *toutes* les paires de dépendance est parfois encore trop forte. En effet, certaines paires ne peuvent apparaître qu'un nombre fini de fois dans une chaîne infinie. Les détecter, c'est se donner les moyens d'affaiblir les contraintes sur l'ordre et ainsi permettre un traitement plus efficace.

Exemple 10.

Considérons le système réduit à la seule règle $f(f(x)) \to f(s(f(x)))$.
Ses deux paires de dépendance sont :

$$\langle \widehat{f}(f(x)), \widehat{f}(x)\rangle, \tag{III.1}$$
$$\langle \widehat{f}(f(y)), \widehat{f}(s(f(y)))\rangle. \tag{III.2}$$

On ne trouvera toutefois jamais plusieurs occurrences de la seconde paire dans une chaîne puisque qu'il n'existe aucune substitution σ telle que $\widehat{f}(s(f(y))\sigma \to^\star \widehat{f}(f(x))\sigma$. Il n'est donc pas nécessaire d'en réclamer une orientation.

Afin de détecter, pour prouver la terminaison d'un système R, les paires cruciales de $\mathrm{DP}(R)$, on construit un graphe dont les nœuds sont des paires et tel que les paires qui peuvent apparaître consécutivement dans une chaîne soient reliées. Toute chaîne infinie correspond donc à un chemin infini dans le graphe qui est fini si le système lui-même l'est et qui par conséquent doit dans ce cas contenir des cycles.

Dans la suite de cette section, les systèmes considérés sont tous finis.

Définition 38. — Soit R un système de récriture. On appelle *graphe de dépendance de* R le graphe $\mathcal{G}$ dont les nœuds sont les paires de dépendance de R et tel que $(\langle s, t\rangle,\ \langle s', t'\rangle) \in \mathcal{G}$ si et seulement s'il existe une substitution σ vérifiant

$$t\sigma \xrightarrow{\neq\Lambda}{}^{\star}\, s'\sigma.$$

Prouver la terminaison du système, revient en fait à vérifier la décroissance des paires qui interviennent *dans les cycles* pourvu qu'au moins l'une d'elles décroisse *strictement*.

Exemple 11. (Arts & Giesl 97)

Considérons un système de division des entiers de Peàno :

$$\begin{cases} x - 0 & \to & x \\ s(x) - s(y) & \to & x - y \\ 0 \div s(y) & \to & 0 \\ s(x) \div s(y) & \to & s((x-y) \div s(y)). \end{cases}$$

Il n'existe pas d'ordre de simplification pour ce système[3]. *On extrait trois paires de dépendance :*

$$\langle s(x)\widehat{-}s(y), x\widehat{-}y\rangle, \tag{III.1}$$
$$\langle s(x)\widehat{\div}s(y), (x-y)\widehat{\div}s(y)\rangle, \tag{III.2}$$
$$\langle s(x)\widehat{\div}s(y), x\widehat{-}y\rangle, \tag{III.3}$$

Qu'on peut agencer en :

$$\begin{tikzcd} (1) \arrow[loop left] & (3) \arrow[l] & (2) \arrow[l] \arrow[loop right] \end{tikzcd}$$

Puisque (3) *n'est pas sur un cycle du graphe, son orientation n'est pas à considérer.*

Le raffinement des graphes préserve la correction des critères par paires de dépendance. Il suffit en fait de limiter l'analyse aux cycles élémentaires du graphe considéré.

Théorème 6. Critère de correction avec graphes.

Soient R un système de récriture fini et $\mathcal{G}$ son graphe de dépendance. S'il existe un préordre $\succeq$ de réduction faible tel que :

1. $l \succeq r$ pour toute règle $l \to r \in R$,
2. $s \succeq t$ pour tout cycle élémentaire $\mathcal{C}$ de $\mathcal{G}$ et pour toute paire $\langle s, t\rangle \in \mathcal{C}$,
3. Pour tout cycle élémentaire $\mathcal{C}$ de $\mathcal{G}$ il existe une paire $\langle s, t\rangle \in \mathcal{C}$ telle que $s \succ t$,

Alors R est fortement normalisant.

Preuve. Un graphe de dépendance est fini car les paires sont en nombre fini. Le chemin associé à une chaîne infinie passe donc un nombre infini de fois par un cycle élémentaire. Les cycles élémentaires sont de longueur finie donc à chaque fois une décroissance stricte est effectuée, ce qui contredit le caractère bien fondé de l'ordre. On conclut par le théorème 5. □

Il n'est pas possible de déterminer automatiquement le graphe de dépendance : l'existence de σ telle que $s\sigma \to^\star t\sigma$ pour s et t donnés est indécidable en général. On introduit donc une notion de

[3]En effet : en instanciant y par $s(x) - y$ on plonge le membre gauche de la dernière règle dans le membre droit, le plongement n'est donc pas contenu dans un ordre qui oriente correctement cette règle.

graphe de dépendance approché dont on sait qu'il contient le graphe de dépendance. Le théorème de correction avec graphes reste bien sûr valable sur ce graphe approché.

Approximation du graphe

Plusieurs méthodes d'approximation des graphes de dépendance ont été proposées. L'approximation originale de Arts & Giesl repose sur l'unifiabilité des membres gauches avec une transformation syntaxique des membres droits et admet une amélioration par *narrowing* [4]. Toyama et Kusakari [53, 55] utilisent des concepts de réduction ω et Ω. Middeldorp [61] quant à lui met en jeu des automates d'arbre. Nous décrivons ici la méthode de Arts & Giesl souvent suffisamment puissante et peu coûteuse.

On veut déterminer un ensemble d'arcs qui contient le graphe de dépendance. Les seuls symboles de racine stables par réduction sont les constructeurs : le critère de sélection retenu quant aux paires à considérer est la possible unification des membres dont les sous-termes ayant des symboles définis à la racine ont été remplacés par des variables distinctes *quelles que soient les occurrences*.

Définition 39. — Soit $R(\mathcal{F})$ un système de récriture. Pour tout terme t,

- $\mathrm{CAP}(t)$ désigne le terme obtenu en remplaçant par des variables les sous-termes de t dont le symbole à la racine est défini dans R ;
- $\mathrm{REN}(t)$ est le terme obtenu en remplaçant les variables de t par de nouvelles variables distinctes (pour toutes occurrences).

Le terme t est *connectable* à un terme t' si $\mathrm{REN}(\mathrm{CAP}(t))$ et t' sont unifiables.

Par unification syntaxique sur les termes modifiés par REN et CAP on détermine des arcs entre les paires de dépendance du système. Le graphe obtenu contient bien le graphe de dépendance.

Proposition 3.
Soit $R(\mathcal{F})$ un système de récriture. S'il existe une substitution σ telle que

$$t\sigma \xrightarrow[R]{\neq\Lambda}{}^{\star}\ t'\sigma,$$

Alors t est connectable à t'.

PREUVE. Par induction sur la structure de t, supposons que $t\sigma$ se récrive en un terme t'' par le système R.

- Si t est une variable ou si son symbole de tête est défini alors $\mathrm{REN}(\mathrm{CAP}(t))$ est une variable et donc unifiable à t''.
- Si t a pour symbole de tête un constructeur c, on peut alors poser $t = c(\dots, t_i, \dots)$. Ainsi $\mathrm{REN}(\mathrm{CAP}(t))$ s'écrit $c(\dots, \mathrm{REN}(\mathrm{CAP}(t_i)), \dots)$. Comme un terme dont la racine est un constructeur ne peut se réduire qu'en un (autre) terme dont la racine est le même constructeur, t'' s'écrit $c(\dots, t''_i, \dots)$ avec les $t_i\sigma$ se récrivant en t''_i. Par hypothèse et par distinction des variables après application de REN, $\mathrm{REN}(\mathrm{CAP}(t))$ est unifiable à t''.

On a le résultat pour $t\sigma \to^\star t''$, donc en particulier pour $t\sigma \to^\star t'\sigma$, il vient ainsi que $\mathrm{REN}(\mathrm{CAP}(t))$ est unifiable à $t'\sigma$ et finalement, comme il ne contient que de nouvelles variables, à t'. □

Remarque 3. — Il s'agit bien d'unification et pas de filtrage : prenons $s = f(x, s(y))$ et $t = f(s(u), v)$ où f est le seul symbole défini, s et t sont unifiables mais s ne filtre pas t. Pourtant avec $R : \{f(x, y) \to$

$s(x)\}$ et $\sigma = \{x \mapsto f(u,u),\ v \mapsto s(y)\}$ on peut avoir la réduction

$$s\sigma = f(f(u,u), s(y)) \xrightarrow[R]{\neq\Lambda}{}^{\star} f(s(u), s(y)) = t\sigma.$$

Enfin, puisque plusieurs règles peuvent s'appliquer à un même terme, il est important de renommer chacune des occurrences des variables pour ne pas obtenir de conclusions erronées.

Exemple 12.
Considérons le système de Toyama [83] :

$$\begin{cases} G(0,1) & \to & 0 \\ G(0,1) & \to & 1 \\ f(0,1,x) & \to & f(x,x,x). \end{cases}$$

Ce système ne termine pas :

$$\begin{aligned} & f(G(0,1), G(0,1), G(0,1)) \to f(0, G(0,1), G(0,1)) \\ & \quad \to f(0,1,G(0,1)) \to f(G(0,1), G(0,1), G(0,1)) \to \ldots \end{aligned}$$

On n'extrait de cet ensemble de règles qu'une seule paire de dépendance : $\langle \widehat{f}(0,1,x), \widehat{f}(x,x,x)\rangle$ *et si on ne renommait pas toutes les occurrences, on ne trouverait pas d'arête et donc pas de cycle sur le graphe. On pourrait alors conclure à tort que le système termine.*

III.5 Des ordres pour la terminaison

Les ordres utilisés dans la pratique pour prouver la terminaison des systèmes de récriture peuvent être distingués en deux catégories : les ordres *syntaxiques*, où ne rentre en considération que la forme des termes à comparer, et les ordres *sémantiques*, pour lesquels on définit une interprétation des termes dans un domaine $\mathcal{D}$ muni d'un ordre $\succ_{\mathcal{D}}$.

III.5.1 Méthodes syntaxiques

Le principe fondamental des méthodes syntaxiques est la comparaison des termes ne prenant en considération que leur structure, éventuellement en se référant à un ordre sur l'alphabet.

On définit habituellement dans ce cadre des *ordres sur les chemins*, construits à partir d'une *précédence*, c'est-à-dire à partir d'un ordre arbitraire sur les symboles d'une signature.

Introduits indépendamment par Nachum Dershowitz [19, 24] et David Plaisted [73], ces ordres reposent sur l'idée qu'un terme s est plus petit qu'un terme t s'il est construit à partir de sous-termes inférieurs (pour l'ordre) aux sous-termes de t dans une structure de symboles de fonctions inférieurs (cette fois pour la précédence) à ceux de t.

Si la comparaison des sous-termes est une extension multiensemble dans le cas des articles précédents, elle peut aussi être menée lexicographiquement comme dans le *LPO* de Kamin & Levy [47]. Ces deux approches sont toutefois subsumées par une définition plus générale proposée en 1982 par Dershowitz [20] et que nous donnons ici.

Définition 40. — Soit $\mathcal{F}$ une signature. On appelle *précédence* un préordre sur $\mathcal{F}$.

Une *fonction de statut admissible* pour une précédence $\succeq$ est une application $\mathcal{S}$ de $\mathcal{F}$ vers $\{\mathsf{mul}\ ;\ \mathsf{lex}\}$ telle que $f \simeq g$ entraîne $\mathcal{S}(f) = \mathcal{S}(g)$ et si en outre $\mathcal{S}(f) = \mathcal{S}(g) = \mathsf{lex}$, alors f et g ont même arité.

Définition 41. — Soient $\mathcal{F}$ une signature et X un ensemble dénombrable de variables, soient $\succeq$ une précédence et $\mathcal{S}$ une fonction de statut admissible pour $\succeq$.

L'*ordre récursif sur les chemins* (RPO) est la relation $\succeq_{\mathrm{RPO}}$ sur $T(\mathcal{F}, X)$ définie par $s \succeq_{\mathrm{RPO}} t$ si et seulement si :

- $s = x \in X$ et $t = x$ ou
- $s = f(s_1, \ldots, s_n)$ avec $f \in \mathcal{F}$ et
  - $s_i \succeq_{\mathrm{RPO}} t$ pour un i, $1 \leq i \leq n$ ou
  - $t = g(t_1, \ldots, t_m)$ avec $g \in \mathcal{F}$ et
    - $f \succ g$ et pour tout j, $1 \leq j \leq m$, $s \succ_{\mathrm{RPO}} t_j$ ou
    - $f \simeq g$ et
      - $\mathcal{S}(f) = \mathsf{mul}$ et $\{s_1, \ldots, s_n\}(\succeq_{\mathrm{RPO}})_{\mathrm{mul}}\{t_1, \ldots, t_m\}$ ou
      - $\mathcal{S}(f) = \mathsf{lex}$ donc $n = m$ et $(s_1, \ldots, s_n)(\succeq_{\mathrm{RPO}})_{\mathrm{lex}}(t_1, \ldots, t_m)$ avec pour tout j, $1 \leq j \leq m$, $s \succ_{\mathrm{RPO}} t_j$ et $s \succ_{\mathrm{RPO}} t$ si $s \succeq_{\mathrm{RPO}} t$ et $t \not\succeq_{\mathrm{RPO}} s$.

RPO peut être utilisé pour montrer la terminaison de systèmes de récriture.

Proposition 4. (Dershowitz 82) [20, 21]
RPO est un ordre de simplification sur $T(\mathcal{F}, X)$.

Exemple 13.
La terminaison du système décrivant la fonction d'Ackermann

$$\begin{cases} A(0, x) & \to \quad s(x) \\ A(s(x), 0) & \to \quad A(x, s(0)) \\ A(s(n), s(m)) & \to \quad A(n, A(s(n), m)), \end{cases}$$

Est prouvée par le RPO avec la précédence $A > s$ *où* $\mathcal{S}(A) = \mathsf{lex}$. *Pour la dernière règle, par exemple,* $s(n)$ *est plus grand que* n *par sous-terme et il reste à vérifier que* $A(s(n), s(m))$ *est plus grand que* $A(s(n), m)$, *ce qui est vrai car* $s(m)$ *est supérieur à* m *par la propriété de sous-terme de l'ordre.*

III.5.2 Méthodes sémantiques

Les méthodes sémantiques reposent sur une interprétation des termes dans un domaine muni d'un ordre bien fondé.

Définition 42. — Soient $T(\mathcal{F}, X)$ une algèbre de termes et $\mathcal{D}$ un domaine muni d'un ordre $\geq_{\mathcal{D}}$. On définit une *interprétation homomorphique* φ par l'association à chaque $f \in \mathcal{F}$ d'arité n d'une fonction $[\![f]\!]_\varphi : \mathcal{D}^n \to \mathcal{D}$ étendue à tous les termes par :

$$\begin{aligned} \varphi(x) &= x, \\ \varphi(f(s_1, \ldots, s_n)) &= [\![f]\!]_\varphi(\varphi(s_1), \ldots, \varphi(s_n)). \end{aligned}$$

On peut alors comparer les termes interprétés par l'*ordre d'évaluation* sur les fonctions sur $\mathcal{D}$.

Définition 43. — Soit $\mathcal{D}$ un domaine muni d'un ordre $\geq_{\mathcal{D}}$. L'*ordre d'évaluation* $\succeq_{\mathcal{D}}$ est défini par :

$$f \succeq_{\mathcal{D}} g \text{ si et seulement si pour tout } (x_1, \ldots, x_n) \in \mathcal{D}^n,\ f(x_1, \ldots, x_n) \geq_{\mathcal{D}} g(x_1, \ldots, x_n)\ ;$$
$$f \succ_{\mathcal{D}} g \text{ si et seulement si pour tout } (x_1, \ldots, x_n) \in \mathcal{D}^n,\ f(x_1, \ldots, x_n) >_{\mathcal{D}} g(x_1, \ldots, x_n).$$

Remarque 4. — L'ordre $\succ_{\mathcal{D}}$ de la définition 43 correspond bien à la partie stricte-stable de $\succeq_{\mathcal{D}}$ mais *n'est pas* sa partie stricte associée.

On compare les termes en se ramenant à leur interprétation :

Définition 44. — L'ordre $(\geq_\varphi, >_\varphi)$ induit par l'interprétation homomorphique φ vers les fonctions sur $\mathcal{D}$ est défini par :

$$\begin{array}{lll} s \geq_\varphi t & \text{si} & \varphi(s) \succeq_{\mathcal{D}} \varphi(t)\ ; \\ s >_\varphi t & \text{si} & \varphi(s) \succ_{\mathcal{D}} \varphi(t). \end{array}$$

Il définit bien une relation convenable.

Lemme 3.
L'ordre $(\geq_\varphi, >_\varphi)$ est stable par instanciation.

PREUVE. Soient s et t deux termes tels que $s >_\varphi t$ et σ une substitution. Par définition de $>_\varphi$, pour tous $x_1, \ldots, x_n$, $\varphi(s)(x_1, \ldots, x_n) \succ_{\mathcal{D}} \varphi(t)(x_1, \ldots, x_n)$. En particulier :

$$\varphi(s\sigma)(x_1, \ldots, x_n) = \varphi(s)(x_1\sigma, \ldots, x_n\sigma) \succ_{\mathcal{D}} \varphi(t)(x_1\sigma, \ldots, x_n\sigma) = \varphi(t\sigma)(x_1, \ldots, x_n),$$

C'est-à-dire $s\sigma >_\varphi t\sigma$.

Le même raisonnement s'applique à $\geq_\varphi$. □

Lemme 4.
Si pour chaque symbole f la fonction $[\![f]\!]_\varphi$ est croissante (resp. strictement) selon chacun de ses arguments, alors $\geq_\varphi$ (resp. $>_\varphi$) est monotone.

PREUVE. En effet si $\varphi(u) \succeq_{\mathcal{D}} \varphi(v)$ et si $[\![f]\!]_\varphi$ est croissante en chacun de ses arguments on a

$$\begin{array}{lll} \varphi(f(s_1, \ldots, u, \ldots, s_n)) = & & \\ [\![f]\!]_\varphi(\varphi(s_1), \ldots, \varphi(u), \ldots, \varphi(s_n)) & \succeq_{\mathcal{D}} & [\![f]\!]_\varphi(\varphi(s_1), \ldots, \varphi(v), \ldots, \varphi(s_n)) \\ & & = \varphi(f(s_1, \ldots, v, \ldots, s_n)) \end{array}$$

Et donc $f(s_1, \ldots, u, \ldots, s_n)) \geq_\varphi f(s_1, \ldots, v, \ldots, s_n))$. □

Lemme 5.
Si $\succeq_{\mathcal{D}}$ est bien fondé alors $\geq_\varphi$ est bien fondé.

PREUVE. Immédiat. □

Trouver une interprétation sémantique faisant (par définition) appel à l'intuition et à l'expérience, une automatisation des preuves est inconcevable avec un choix infini de domaines. Afin de décharger autant que faire se peut cette recherche sur une machine, on se restreint souvent à une portion minorée de l'ensemble des entiers, c'est-à-dire à $\mathcal{D}_\mu = \{n \in \mathbb{Z} \text{ tels que } n \geq \mu\}$ où $\geq$ est l'ordre naturel sur $\mathbb{Z}$. On dit de telles interprétations qu'elles sont *arithmétiques*.

Interprétations polynomiales

Introduite par Lankford [56] en 1979, l'interprétation des termes par des fonctions polynomiales sur les entiers est toujours l'une des méthodes les plus utilisées.

Définition 45. — Une *interprétation polynomiale* de termes de $T(\mathcal{F}, X)$ est une interprétation arithmétique sur un domaine $\mathcal{D}_\mu$ telle que pour tout $f \in \mathcal{F}$, $[\![f]\!]$ est une fonction polynomiale.

Il est possible, dans le cas des interprétations polynomiales, de se ramener au domaine $\mathcal{D}_0$ en effectuant la translation $f_0(x_1, \ldots, x_n) = f_\mu(x_1 + \mu, \ldots, x_n + \mu) - \mu$ et de définir ainsi un ordre $(\geq_{\varphi_0}, >_{\varphi_0})$ à partir de $(\geq_{\varphi_\mu}, >_{\varphi_\mu})$.

Lemme 6.
Soit $(\geq_{\varphi_0}, >_{\varphi_0})$ obtenu par translation à partir de $(\geq_{\varphi_\mu}, >_{\varphi_\mu})$.

$$\begin{array}{lll} s \geq_{\varphi_0} t & \text{si et seulement si} & s \geq_{\varphi_\mu} t, \\ s >_{\varphi_0} t & \text{si et seulement si} & s >_{\varphi_\mu} t. \end{array}$$

PREUVE. En effet, pour un terme t : $\varphi_0(t) = \varphi_\mu(t) - \mu$. □

Remarque 5. — Il existe un moyen simple de s'assurer de la monotonie d'un ordre sémantique à base d'interprétations polynomiales : si tous les coefficients de la fonction polynomiale sont positifs, celle-ci est croissante.

Exemple 14.
Prouvons la terminaison du système :

$$\left\{ \begin{array}{lll} x + 0 & \rightarrow & x \\ x + s(y) & \rightarrow & s(x + y). \end{array} \right.$$

L'interprétation polynomiale ($\mu = 1$) *:*

$$\begin{array}{l} [\![0]\!] = 1, \\ [\![s]\!](x) = x + 1, \\ [\![+]\!](x, y) = x + 2y, \end{array}$$

Induit une application φ *telle que :*

$$\begin{array}{ll} \varphi(x) = x, & \varphi(x + 0) = x + 2, \\ \varphi(x + s(y)) = x + 2y + 2, & \varphi(s(x + y)) = x + 2y + 1. \end{array}$$

Les règles sont alors correctement orientées en comparant les images grâce à l'ordre d'évaluation : pour toute valeur entière de x *supérieure ou égale à* 1*, la fonction qui à* x *associe* $x + 2$ *prend une valeur supérieure à celle prise par la fonction qui à* x *associe* x *et de même pour les fonctions qui associent à* x *respectivement* $x + 2y + 2$ *et* $x + 2y + 1$*. On dispose bien d'un ordre permettant de montrer la terminaison du système.*

Remarque 6. — Une translation de l'interprétation utilisée dans l'exemple 14 permet de se ramener du domaine $\mathcal{D}_1 = \{x \in \mathbb{Z} \mid x \geq 1\}$ à $\mathcal{D}_0 = \mathbb{N}$ avec l'interprétation polynomiale :

$$\begin{array}{l} [\![0]\!] = 0, \\ [\![s]\!](x) = x + 1, \\ [\![+]\!](x, y) = x + 2y + 2. \end{array}$$

Exemple 15.

Pour prouver la terminaison du système de différenciation décrit par les règles suivantes [22] :

$$\begin{array}{rcl}
Dt & \to & 1, \\
D(\mathit{constante}) & \to & 0, \\
D(x+y) & \to & Dx + Dy, \\
D(x \times y) & \to & (y \times Dx) + (x \times Dy), \\
D(x-y) & \to & Dx - Dy, \\
D(neg(x)) & \to & neg(Dx), \\
D(x/y) & \to & (Dx/y) - (x \times Dy/y^2), \\
D(Log(x)) & \to & Dx/x, \\
D(x^y) & \to & (y \times x^{y-1} \times Dx) + (x^y \times (Log(x)) \times Dy),
\end{array}$$

Il suffit de prendre l'interprétation polynomiale :

$$\begin{array}{lll}
[\![+]\!](x,y) = x+y, & [\![-]\!](x,y) = x+y, & [\![x^y]\!](x,y) = x+y, \\
[\![neg]\!](x) = x+1, & [\![\mathit{constante}]\!] = 2, & [\![\times]\!](x,y) = x+y, \\
[\![/]\!](x,y) = x+y, & [\![D]\!](x) = x^2, & [\![Log]\!](x) = x+1, \\
[\![t]\!] = 2. & &
\end{array}$$

La mise en œuvre des interprétations polynomiales entraîne la nécessité de comparer deux fonctions polynomiales sur $\mathbb{N}$ à coefficients entiers et, ce faisant, se heurte au dixième problème de Hilbert [42]. Ce problème fut montré indécidable par Matijasevic (à partir de travaux de Robinson) [16, 59]. Comme souvent en pareil cas, on s'est efforcé de trouver des méthodes correctes, incomplètes mais satisfaisantes pour le plus grand nombre de cas possible afin de comparer des fonctions polynomiales ou, ce qui revient au même, de vérifier leur positivité sur toutes valeurs naturelles de leurs arguments. Nous reviendrons plus en détails sur ces méthodes dans le courant de la troisième partie, paragraphe II.2.2.

Exponentielles

Il existe également des méthodes sémantiques autorisant l'interprétation des termes comme des fonctions exponentielles.

Exemple 16. (Dershowitz 95) [22]

On souhaite utiliser une approche sémantique pour prouver la terminaison du système de mise en forme normale disjonctive :

$$\left\{\begin{array}{lcl}
\neg\neg x & \to & x \\
\neg(x \vee y) & \to & (\neg x) \wedge (\neg y) \\
\neg(x \wedge y) & \to & (\neg x) \vee (\neg y) \\
x \wedge (y \vee z) & \to & (x \wedge y) \vee (x \wedge z) \\
(y \vee z) \wedge x & \to & (x \wedge y) \vee (x \wedge z).
\end{array}\right.$$

Il est possible d'utiliser l'interprétation exponentielle suivante :

$$[\![\vee]\!](x,y) = x + y + 1, \qquad [\![\wedge]\!](x,y) = x \times y,$$
$$[\![\neg]\!](x) = 2^x, \qquad [\![\mathit{constante}]\!] = 2.$$

III.6 Méthodes transformationnelles

Il s'agit de transformations du système R, dont on doit prouver la terminaison, en un système R', plus facile à étudier, telles que si R' est fortement normalisant alors R est fortement normalisant.

Une des plus célèbres instances de méthodes transformationnelles est sans doute l'*étiquetage sémantique* proposé par Hans Zantema [85] où à chaque symbole de fonction on appose une étiquette dépendant de la sémantique de ses arguments, les règles du système original R étant étendues en conséquence. On obtient ainsi un nouveau système R' avec la propriété que R termine si et seulement si R' termine. Le problème est alors d'obtenir un R' dont la terminaison est plus aisée à prouver que celle de R. Le choix de l'étiquetage reste à la charge de l'intuition de qui veut prouver la terminaison ; cette méthode reste donc très difficile à automatiser.

On peut également mentionner la *dummy elimination* de Ferreira et Zantema [27] et son extension par Kusakari, Nakamura et Toyama (*Argument Filtering Transformation*) [54] où les arguments non pertinents sont « retirés » du système. Ces méthodes sont toutefois subsumées par une approche couplée paires de dépendance/transformation de contraintes d'ordre [33].

III.7 Transformation des contraintes d'ordre

Plutôt que prouver la terminaison d'un nouveau système de récriture, il est possible de transformer directement les contraintes d'ordre issues du système original. Ces approches permettent, par exemple par élimination des arguments inopportuns, de construire un ordre de réduction faible à partir d'un ordre de réduction donné. Elles sont en particulier tout à fait adaptées aux critères par paires de dépendance où les ordres requis ne sont pas strictement monotones.

Définition 46. — On appelle *schéma de programme récursif* (RPS) un système de récriture tel que :

- Chaque symbole défini n'est à la racine que d'une seule règle ;
- Toute règle est de la forme $f(x_1, \ldots, x_n) \to r$ où les x_i sont des variables deux-à-deux disjointes et r est un terme arbitraire.

La terminaison des RPS est un problème décidable : il suffit qu'il n'y ait pas de règles récursives ou mutuellement récursives.

Définition 47. — Un *système filtre d'arguments* (abrégé en AFS, de l'anglais Argument Filtering System) est un RPS tel que pour chaque règle $f(x_1, \ldots, x_n) \to r$, le membre droit r est ou bien

- Une variable (un des x_i) ou bien
- Un terme $N_f(y_1, \ldots, y_m)$ où N_f est un nouveau symbole n'apparaissant que dans cette règle et où l'ensemble des y_i est inclus dans celui des x_i.

Tout AFS est fortement normalisant et confluent (car orthogonal).

Théorème 7. (Arts 97) [3]
Soient P un RPS et $\geq$ un préordre de réduction faible. La relation $\succeq$ définie par $s \succeq t$ si et seulement si $s{\downarrow_P} \geq t{\downarrow_P}$ est un préordre de réduction faible.

On a ainsi gagné un moyen de construire des préordres de réduction faibles.

Exemple 17. (Ferreira & Zantema) [28]
Considérons le système :

$$\left\{ \begin{array}{lcl} f(g(x)) & \to & g(f(f(x))) \\ f(h(x)) & \to & h(g(x)). \end{array} \right.$$

On en extrait deux paires de dépendance :

$$\begin{array}{l} \langle \widehat{f}(g(x)), \widehat{f}(x) \rangle, \\ \langle \widehat{f}(g(x)), \widehat{f}(f(x)) \rangle. \end{array}$$

En appliquant le RPS : $\{f(x) \to x\ ;\ h(x) \to 0\}$, *les contraintes deviennent :*

$$\begin{array}{rcl} g(x) & \geq & g(x), \\ 0 & \geq & 0, \\ \widehat{f}(g(x)) & > & \widehat{f}(x). \end{array}$$

On peut alors aisément conclure grâce à RPO avec tous les symboles égaux dans la précédence.

Première partie

Structure hiérarchique des systèmes de récriture

Introduction

Afin de proposer un cadre plus adapté à l'étude des comportements modulaires des systèmes de récriture et pour profiter davantage des informations sur la structure hiérarchique des systèmes, nous allons dans un premier temps et le chapitre I étudier les unions ensemblistes de systèmes et certains des travaux sur la terminaison qu'elles ont suscités, notamment ceux de Gramlich, Krishna Rao, Dershowitz et Ohlebusch. Nous introduirons au cours du chapitre II la notion de *modules de récriture* (définition 58) et verrons comment exprimer grâce à elle différentes sortes d'extensions. Nous proposerons alors (chapitre III) une classe d'ordres au bon comportement et construirons sur les modules des *paires de dépendance relatives* (définition 60). Ces ordres et ces paires nous permettront d'exposer au chapitre IV de nouvelles méthodes de preuve de terminaison concernant les unions hiérarchiques, méthodes autorisant un déroulement incrémental de la preuve et donc une application de techniques automatiques à de gros systèmes.

Chapitre I

Unions de systèmes

Les systèmes, comme les signatures, sont des ensembles : on peut étudier leur composition en se penchant sur les propriétés caractéristiques de leurs unions respectives.

I.1 Unions et modularité

L'union de deux systèmes de récriture $R_1(\mathcal{F}_1)$ et $R_2(\mathcal{F}_2)$ est différemment considérée suivant l'état de l'intersection $\mathcal{F}_1 \cap \mathcal{F}_2$ par rapport à l'union des règles.

Définition 48. — Deux systèmes $R_1(\mathcal{F}_1)$ et $R_2(\mathcal{F}_2)$ sont :

- *Disjoints* si $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$;
- *À constructeurs partagés* si les signatures $\mathcal{F}_1$ et $\mathcal{F}_2$ ne partagent que des symboles constructeurs ($\mathcal{F}_1 \cap \mathcal{F}_2 = E$ et pour tout $f \in E$, f n'est jamais en tête d'un membre gauche de règle) ;
- *Composables* si R_1 restreint aux règles dont le symbole de tête est dans $\mathcal{F}_2$ est égal à R_2 restreint aux règles dont le symbole de tête est dans $\mathcal{F}_1$.

Une union $R_1 \cup R_2$ sera par suite elle-même qualifiée d'union :

- *Disjointe*, si $\mathcal{F}_1$ et $\mathcal{F}_2$ sont disjointes ;
- *À constructeurs partagés*, si R_1 et R_2 ne partagent que des constructeurs ;
- *De (systèmes) composables*, si R_1 et R_2 sont composables.

On appelle *union hiérarchique* une union $R_1(\mathcal{F}_1) \cup R_2(\mathcal{F}_2)$ telle qu'en notant D_1 et D_2 les ensembles des symboles définis de R_1 et R_2, et C_1 et C_2 leurs constructeurs respectifs :

- $(D_2 \setminus D_1) \cap \mathcal{F}_1 = \emptyset$ et
- $\{l \to r \mid \Lambda(l) \in D_1 \cap D_2\} = R_1 \cap R_2$.

Une union hiérarchique est donc une union où des symboles d'un système peuvent apparaître comme constructeurs dans l'autre et où les symboles définis dans les deux systèmes sont en tête des mêmes règles pour chacun d'eux.

Exemple 18.

En prenant :

- $R_1 : \{f(x) \to x\}$ *et* $R_2 : \{g(x) \to x\}$, $R_1 \cup R_2$ *est une union disjointe ;*
- $R_1 : \{f(h(x)) \to x\}$ *et* $R_2 : \{g(x) \to h(x)\}$, $R_1 \cup R_2$ *est une union à constructeurs partagés (où le seul constructeur est* h*) ;*

– $R_1 : \{f(h(x)) \to x,\ h(a) \to b\}$ *et* $R_2 : \{g(x) \to h(x),\ h(a) \to b\}$, $R_1 \cup R_2$ *constitue une union de systèmes composables ;*

– $R_1 : \{f(g(x)) \to x\}$ *et* $R_2 : \{g(x) \to x\}$, $R_1 \cup R_2$ *est enfin une union hiérarchique,* g *étant défini dans* R_2 *mais servant comme constructeur dans* R_1.

On a clairement les inclusions suivantes :

– Les unions à constructeurs partagés contiennent les unions disjointes ;

– Les unions de systèmes composables subsument les unions à constructeurs partagés ;

– Les unions hiérarchiques englobent les unions de systèmes composables.

Ces relations sont récapitulées à la figure I.1.

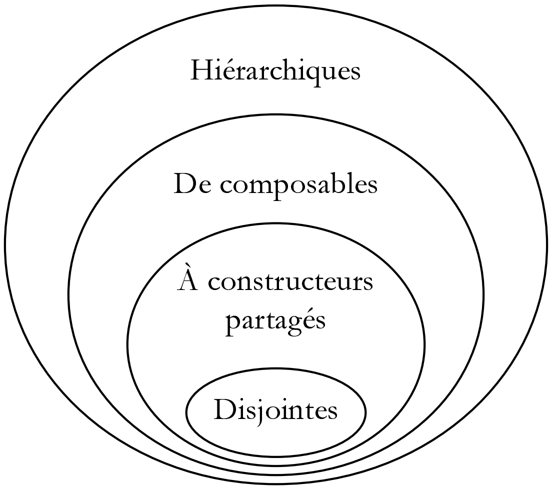


FIG. I.1. Relation entre les différents types d'union de systèmes.

Définition 49. — Une propriété P est *modulaire pour un certain type d'union (disjointe, à constructeurs partagés, de systèmes composables, etc.)* si l'union de ce type de deux systèmes vérifiant P possède également la propriété P.

Nous nous intéressons au comportement de ces différentes unions vis-à-vis de propriétés que peuvent avoir les systèmes, la terminaison en particulier.

I.2 Unions et terminaison

L'importance tant théorique que pratique des propriétés modulaires a généré une importante quantité de travaux, notamment sur la terminaison de systèmes de récriture. Plutôt qu'un inventaire de tous les résultats obtenus à ce jour nous proposons une description des approches utilisées, d'abord dans le cas des unions disjointes — avec une mise en évidence de l'influence des projections — et leurs extensions aux constructeurs partagés et concernant les unions hiérarchiques, ensuite dans le cadre d'une preuve à l'aide des critères par paires de dépendance. Nous présentons enfin certaines caractéristiques d'une notion de terminaison contrôlant l'ajout de règles projectives.

I.2.1 Unions disjointes

La terminaison n'est pas une propriété qui se transmet facilement aux unions de systèmes. Même une union de systèmes fortement normalisants disjoints est susceptible de ne pas terminer. Ce comportement est illustré par un fameux exemple de Toyama [83].

Exemple 19.

Reprenons l'exemple de Toyama constitué des deux systèmes R et π suivants :

$$R : \left\{ \begin{array}{lcl} f(0,1,x) & \to & f(x,x,x) \end{array} \right. \qquad\qquad \pi : \left\{ \begin{array}{lcl} G(x,y) & \to & x \\ G(x,y) & \to & y. \end{array} \right.$$

Le système projectif π termine puisque la taille des termes diminue strictement lors d'une réduction ; R est également fortement normalisant[1]*. Pourtant l'union de ces deux systèmes disjoints permet une réduction infinie :*

$$\begin{array}{rcl} f(G(0,1),G(0,1),G(0,1)) & \to & f(0,G(0,1),G(0,1)) \\ & \to & f(0,1,G(0,1)) \\ & \to & f(G(0,1),G(0,1),G(0,1)) \to \ldots \end{array}$$

Supposer que la non préservation de la propriété de terminaison est due au fait que $R \cup \pi$ n'est pas confluent[2] peut être réfuté à l'aide d'un exemple du même acabit, dû à Drosten [25] puis simplifié par Middeldorp [60].

Exemple 20.

Considérons les deux systèmes R_1 et R_2 suivants :

$$R_1 : \left\{ \begin{array}{ccl} f(0,1,x) & \to & f(x,x,x) \\ f(x,y,z) & \to & 2 \\ 0 & \to & 2 \\ 1 & \to & 2 \end{array} \right. \qquad\qquad R_2 : \left\{ \begin{array}{lcl} h(x,x,y) & \to & y \\ h(x,y,y) & \to & x. \end{array} \right.$$

R_1 et R_2 sont clairement confluents et fortement normalisants[3]*, il est pourtant possible de réduire indéfiniment $f(h(0,1,1),h(0,1,1),h(0,1,1))$ par $R_1 \cup R_2$:*

$$\begin{array}{rcl} f(h(0,1,1),h(0,1,1),h(0,1,1)) & \to & f(0,h(0,1,1),h(0,1,1)) \\ & \to & f(0,h(2,1,1),h(0,1,1)) \\ & \to & f(0,h(2,2,1),h(0,1,1)) \\ & \to & f(0,1,h(0,1,1)) \\ & \to & f(h(0,1,1),h(0,1,1),h(0,1,1)) \to \ldots \end{array}$$

[1] La preuve est effectuée à l'aide d'un argument de minimalité et en constatant qu'une réduction par R d'un terme t produit un terme avec au moins autant de symboles.

[2] C'était en particulier une conjecture de Hsiang.

[3] La terminaison de R_1 est montrée de façon semblable à la preuve de normalisation forte du système R de l'exemple 19.

I.2.2 Analyse structurelle des contre-exemples

Si l'union de deux systèmes disjoints fortement normalisants ne termine pas forcément, c'est qu'il existe un moyen de créer des radicaux pour l'un des systèmes tout en récrivant par l'autre.

En analysant la structure de contre-exemples à la modularité de la terminaison pour les unions disjointes et en particulier l'importance de l'évolution de l'entrelacement des signatures au cours d'une réduction, Michaël Rusinowitch [79] propose, comme responsable de cette création, la projection.

En effet, si les deux systèmes $R_1(\mathcal{F}_1)$ et $R_2(\mathcal{F}_2)$ sont disjoints, le seul moyen pour, disons, R_1 de produire un (sous) terme réductible par R_2 à partir d'un terme t (irréductible pour R_2) est de *combiner* (et non pas créer) des symboles de $\mathcal{F}_2$, c'est-à-dire de supprimer les symboles de $\mathcal{F}_1$ qui masquent le radical dans t (voir figure I.2). Dans le cadre de la récriture du premier ordre, seule une projection peut faire cela.

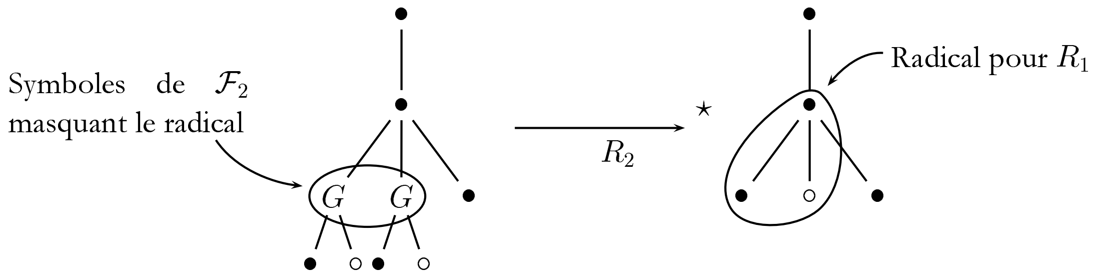


FIG. I.2. Mise en évidence de radical par projection.

Une manière d'assurer un comportement modulaire de la terminaison serait alors de contrôler les effets de la projection.

Bernhard Gramlich considère alors les contre-exemples minimaux pour l'entrelacement des signatures, c'est-à-dire les termes non fortement normalisables et dont le nombre maximal d'alternances de signatures lors d'un parcours en profondeur est le plus petit possible. Il introduit la notion de systèmes *préservant la terminaison sous projection non déterministe.* Ce sont des systèmes qui terminent et restent fortement normalisants lorsqu'on leur ajoute un nouveau symbole binaire G et les règles du système π :

$$\pi : \left\{ \begin{array}{lcl} G(x,y) & \to & x \\ G(x,y) & \to & y. \end{array} \right.$$

La propriété de terminaison préservée sous projection non déterministe est aussi appelée *terminaison* $\mathcal{C}_{\mathcal{E}}$, une terminologie introduite par Enno Ohlebusch [69], $\mathcal{C}_{\mathcal{E}}$ désignant une terminaison « $\mathcal{C}$ollapse $\mathcal{E}$xtended », c'est-à-dire étendue à l'union du système et de π.

Gramlich tire de cette notion un résultat fondamental [36, 37] :

Théorème 8. (Gramlich 91)
Soient $R_1(\mathcal{F}_1)$ et $R_2(\mathcal{F}_2)$ deux systèmes de récriture disjoints fortement normalisants et tels que leur union disjointe $R_1 \cup R_2$ ne termine pas.

Il existe alors un j, $j \in \{1,2\}$, tel que R_j ne préserve pas la terminaison sous projection non déterministe et tel que l'autre système R_{3-j} est projectif.

Remarque 7. — La terminaison $\mathcal{C}_\mathcal{E}$ est une propriété beaucoup moins restrictive que la terminaison simple, nous le verrons au cours du paragraphe I.2.6. En particulier si l'ajout des projections pour *tous* les symboles permet l'élimination de constructeurs qui pouvaient « masquer un radical », cette opération n'est pas possible à l'aide des règles de π.

La plupart des conditions générales[4] requises pour obtenir un comportement modulaire de la terminaison sont issues de cette approche.

Une dernière tentative mérite d'être mentionnée, celle des *ordres d'extension* d'Albert Rubio [77]. Dans ce cadre, on s'occupe directement des compositions des relations d'ordre dans lesquelles sont plongés les systèmes. Il en ressort éventuellement un ordre dit d'extension possédant les propriétés qui le rendent utilisable pour la preuve de terminaison de l'union des systèmes. Les conditions sur les termes et sur les ordres sont cependant assez fortes, en particulier des restrictions syntaxiques (linéarité par exemple) sont nécessaires pour que l'ordre d'extension contiennent effectivement tous les ordres à partir desquels il est formé lorsqu'on cherche à comparer des termes non clos.

I.2.3 Constructeurs partagés et unions de systèmes composables

On aborde avec le partage de constructeurs des cas bien plus courants dans la pratique que la séparation complète des symboles. L'utilisation des mêmes constructeurs par exemple est naturelle pour des fonctions complètement différentes : on peut faire de l'arithmétique ou construire des listes avec les mêmes entiers.

On distingue encore deux approches dans le cadre de la récriture standard : Middeldorp et Toyama posent des restrictions syntaxiques et se limitent ainsi à des systèmes dont tous les sous-termes propres des membres gauches ne comportent que des constructeurs [63] ; Kurihara, Ohuchi et Gramlich se concentrent sur la terminaison simple [37, 50].

Théorème 9. (Middeldorp & Toyama) [63]
La terminaison est une propriété modulaire des unions à constructeurs partagés de systèmes tels que pour toute règle $l \rightarrow r$, tout sous-terme propre de l ne contient que des constructeurs (et des variables).

L'inclusion d'une relation de récriture dans une ordre de simplification a été montrée modulaire par Kurihara et Ohuchi pour les unions à constructeurs partagés [50].

On a en particulier pour les systèmes terminant simplement :

Théorème 10. (Gramlich 94) [37]
La terminaison simple est une propriété modulaire des unions à constructeurs partagés de systèmes à branchement fini.

On peut enfin utiliser une stratégie particulière d'application des règles.

Théorème 11. (Gramlich 95) [38]
La terminaison suivant la stratégie innermost est modulaire pour les unions de systèmes à constructeurs partagés.

---

[4] Il existe bien sûr des critères importants sans rapport avec la terminaison $\mathcal{C}_\mathcal{E}$. Ceux-ci concernent en fait des systèmes sans paire critique, ou encore sans règle dupliquante et cet aspect un peu *ad hoc* ne nous autorise pas à les inscrire dans le cadre d'une approche globale de la terminaison.

Remarque 8. — En conjuguant les théorèmes 1 et 11 un système orthogonal peut donc facilement bénéficier d'un traitement modulaire.

Il est remarquable dans la littérature qu'à ce jour les propriétés (relatives à la terminaison) connues comme modulaires pour les systèmes à constructeurs partagés le soient également pour les systèmes composables. Les extensions à ce type de systèmes de la propriété d'inclusion dans un ordre de simplification ci-dessus ainsi que du théorème 11 sont dues à Enno Ohlebusch [70] ; la modularité de la terminaison simple pour les unions de systèmes composables à branchement fini a été obtenue par Bernhard Gramlich [39].

I.2.4 Unions hiérarchiques

Le cas des unions hiérarchiques est plus délicat à traiter si on persévère dans l'utilisation des techniques dédiées aux unions précédentes. En particulier, la relative stabilité du niveau d'entrelacement des signatures disparaît avec l'utilisation dans, disons, R_2, de symboles définis dans R_1.

On peut alors penser restreindre l'étude à certaines classes d'unions hiérarchiques présentant des propriétés de stabilité exploitables. C'est la démarche de Krishna Rao lorsqu'il introduit les extensions *propres* et *propres restreintes* [74, 75] bâties sur une relation de dépendance entre les symboles définis.

Définition 50. — Soit $\succeq_d$ un plus petit préordre sur un ensemble de symboles définis d'un système R tel que $f \succeq_d g$ s'il existe une règle $l \to r \in R$ telle que $\Lambda(l) = f$ et g apparaît dans r. On dit alors que f *dépend de* g.

Soient $R_1(\mathcal{F}_1)$, de symboles définis D_1 et de constructeurs C_1, et $R_2(\mathcal{F}_2)$, de symboles définis D_2 et de constructeurs C_2, deux systèmes de récriture.

$R_2(\mathcal{F}_2)$ est une *extension propre* de $R_1(\mathcal{F}_1)$ si

1. $D_1 \cap D_2 = C_1 \cap D_2 = \emptyset$;
2. Pour chaque règle $l \to r \in R_1$, tout sous-terme r' de r tel que :
   – $\Lambda(r') \in \{f \in D_2 \mid f \succeq_d g \text{ pour un } g \in D_1\}$
   – $\Lambda(r') \succeq_d \Lambda(l)$,

   Ne contient aucun symbole dépendant d'un élément de D_1 ailleurs qu'à la racine.

$R_2(\mathcal{F}_2)$ est une *extension propre restreinte* de $R_1(\mathcal{F}_1)$ si c'est une extension propre telle qu'aucun membre gauche des règles de R_2 ne contient, ailleurs qu'à Λ, de symbole de $D_1 \cup \{f \in D_2 \mid f \succeq_d g \text{ pour un } g \in D_1\}$.

Les extensions propres sont des cas particuliers d'extensions hiérarchiques. Leur caractéristique est que les symboles dépendant de D_1 n'apparaissent jamais sous un symbole de fonction en récursion mutuelle avec $\Lambda(l)$.

Exemple 21.

Considérons ces deux systèmes d'arithmétique sur les entiers de Peàno :

$$R_+ \;:\; \left\{ \begin{array}{lcl} x+0 & \to & x \\ x+s(y) & \to & s(x+y) \end{array} \right. \qquad R_\times \;:\; \left\{ \begin{array}{lcl} x\times 0 & \to & 0 \\ x\times s(y) & \to & x+(x\times y), \end{array} \right.$$

$R_\times$ constitue une extension propre de R_+.

Définition 51. — Soient $R_1(\mathcal{F}_1)$, de symboles définis $D_1 \cup D$ et de constructeurs C_1, et $R_2(\mathcal{F}_2)$, de symboles définis $D_2 \cup D$ et de constructeurs C_2, deux systèmes de récriture ($D_1 \cap D = \emptyset = D_2 \cap D$).

$R_2(\mathcal{F}_2)$ est une *extension propre généralisée* de $R_1(\mathcal{F}_1)$ si

1. $D_1 \cap D_2 = C_1 \cap D_2 = \emptyset$,
2. Pour chaque règle $l \to r \in R_1$ on a que pour tout sous-terme r' de r,
   - Si $\Lambda(r') \in \{f \in D_2 \mid f \succeq_d g \text{ pour un } g \in D_1\} \setminus D$,
   - Et si $\Lambda(r') \succeq_d \Lambda(l)$,

   Alors r' ne contient aucun symbole dépendant d'un élément de D_1 ailleurs qu'à la racine.

Les extensions propres généralisées sont également des cas particuliers d'extensions hiérarchiques.

Krishna Rao peut alors généraliser à cette classe particulière d'extensions hiérarchiques la modularité de la terminaison simple [74].

Théorème 12. (Krishna Rao 94)

La terminaison simple est une propriété modulaire des extensions propres généralisées.

Dershowitz, quant à lui, se limite aux *extensions sur constructeurs* et essaye deux approches : 1) Contrôler l'évolution de l'entrelacement des signatures et 2) Se ramener à une réduction par un seul des systèmes impliqués [23].

Définition 52. — Une extension hiérarchique d'un système $R_1(\mathcal{F}_1)$ par un système $R_2(\mathcal{F}_2)$ est une extension *sur constructeurs* si les membres gauches des règles de R_2 ne contiennent pas de symboles définis dans R_1.

Le contrôle de l'entrelacement est obtenu par une sévère contrainte de *platitude*.

Définition 53. — Un terme est dit *plat* pour une signature $\mathcal{F}$ si dans tous les chemins partant de Λ on ne trouve au plus qu'une seule occurrence d'un symbole de $\mathcal{F}$.

On qualifiera par extension de *système plat* un système dont les membres de toutes les règles sont plats.

De ses deux approches Dershowitz obtient des critères très pointus (voire trop vis-à-vis des systèmes rencontrés dans la pratique) mais avec des contraintes restrictives (linéarité, platitude) et même parfois sémantiques (confluence requise dans certains cas).

Citons enfin les unions *alien decreasing* de Fernandez & Jouannaud [26]. Cette restriction, qui peut être appliquée à des cas d'extensions encore plus généraux que le cas hiérarchique est toutefois inadaptée à une automatisation des preuves.

I.2.5 Approche modulaire par paires de dépendance

Le raffinement par graphe des paires de dépendance limite les contraintes aux cycles dudit graphe. On dispose alors d'un premier moyen de découpage de la preuve de terminaison : les cycles disjoints donnent des informations de terminaison orthogonales et peuvent être exploités, au moins dans un premier temps, séparément. Cette considération amène Arts & Giesl à proposer de nouveaux résultats [5] issus d'une étude approfondie des graphes de dépendance.

Remarque 9. — Dans la mesure où ils utilisent une recherche de cycles dans les graphes de dépendance, ces résultats ne peuvent s'appliquer qu'à des systèmes *finis*.

Précisons tout d'abord ce qu'est une partie fortement connexe.

Définition 54. — Soit $\mathcal{G}$ un graphe. On appelle *partie fortement connexe* de $\mathcal{G}$ tout ensemble $\mathcal{C}$ de sommets tel que pour tous a et b éléments de $\mathcal{C}$ il existe un chemin dans $\mathcal{G}$ de a vers b.

Un telle partie sera *strictement* fortement connexe si ces chemins sont tous non vides.

Remarque 10. — Se concentrer sur les parties strictement fortement connexes permet de s'affranchir du traitement des singletons $\{s\}$ tels qu'il n'y a pas d'arc de s vers lui-même dans $\mathcal{G}$.

Si l'article original de Arts & Giesl [5] utilise des cycles du graphe, nous préférons ici pour des raisons de clarté la notion de partie strictement fortement connexe. Dans un graphe fini on ne trouve en effet qu'un nombre fini de parties connexes, ce qui n'est pas le cas des cycles ; il est de plus incorrect ici (puisqu'on va en fait utiliser des ordres différents selon les cycles) de se limiter aux cycles élémentaires.

Théorème 13. (Arts & Giesl 98)
Un système R est fortement normalisant si pour chaque partie strictement fortement connexe $\mathcal{C}$ du graphe de dépendance de R, il n'existe aucune chaîne de dépendance infinie constituée de paires de $\mathcal{C}$.

PREUVE. Par l'absurde. Si R possède une chaîne infinie, considérons l'ensemble $\mathcal{C}$ des paires qui y apparaissent une infinité de fois. $\mathcal{C}$ est une partie strictement fortement connexe car pour toutes paires P_1 et P_2 de $\mathcal{C}$, P_2 apparaît après P_1 et réciproquement. Ainsi, à partir d'un certain rang, la chaîne est une chaîne de paires de $\mathcal{C}$. □

Ce théorème permet de prouver la terminaison d'un système de façon modulaire. Il suffit en effet de prouver l'absence de chaîne infinie pour chaque partie strictement fortement connexe indépendamment des autres.

Théorème 14. (Arts & Giesl 98)
Soit R un système de récriture. Si pour chaque partie strictement fortement connexe $\mathcal{C}$ de son graphe de dépendance il existe un préordre bien fondé, stable et faiblement monotone $\geq_{\mathcal{C}}$ tel que :

1. $l \geq_{\mathcal{C}} r$ pour toute règle $l \to r \in R$,
2. $s \geq_{\mathcal{C}} t$ pour toute paire $\langle s, t\rangle \in \mathcal{C}$,
3. $s >_{\mathcal{C}} t$ pour au moins une paire $\langle s, t\rangle \in \mathcal{C}$,

Alors R est fortement normalisant.

Remarquons toutefois que la première condition impose des contraintes sur *toutes* les règles du système.

En combinant une approche par RPS et cette analyse du graphe, Giesl & Ohlebusch définissent les notions de terminaison DP-*simple* et DP-*quasi simple* [29]. Puisqu'elles utilisent les paires de dépendance, ces notions tendent à faciliter l'automatisation des preuves en affaiblissant les contraintes.

Définition 55. — Un système fini R termine DP-*simplement* si pour chaque partie strictement fortement connexe $\mathcal{C}$ du graphe de dépendance de R il existe un AFS P et un ordre de simplification $>_{\mathcal{C}}$ tel que :

1. $l{\downarrow_P} \geq_{\mathcal{C}} r{\downarrow_P}$ pour toute règle $l \to r \in R$,
2. $s{\downarrow_P} \geq_{\mathcal{C}} t{\downarrow_P}$ pour toute paire $\langle s, t\rangle \in \mathcal{C}$,
3. $s{\downarrow_P} >_{\mathcal{C}} t{\downarrow_P}$ pour au moins une paire $\langle s, t\rangle \in \mathcal{C}$.

Définition 56. — Un système fini R termine DP-*quasi simplement* si pour chaque partie strictement fortement connexe $\mathcal{C}$ du graphe de dépendance de R il existe un AFS P et un préordre de simplification $\geq_{\mathcal{C}}$ tel que :

1. $s \geq_{\mathcal{C}} t$ entraîne $\mathcal{V}ar(s) \supseteq \mathcal{V}ar(t)$ pour tous termes s et t,
2. $l{\downarrow_P} \geq_{\mathcal{C}} r{\downarrow_P}$ pour toute règle $l \to r \in R$,
3. $s{\downarrow_P} \geq_{\mathcal{C}} t{\downarrow_P}$ pour toute paire $\langle s, t\rangle \in \mathcal{C}$,
4. $s{\downarrow_P} >_{\mathcal{C}} t{\downarrow_P}$ pour au moins une paire $\langle s, t\rangle \in \mathcal{C}$.

Remarque 11. — Les ordres pertinents appartiennent maintenant à une classe particulière (et donc réduite) d'ordres faiblement monotones. Le principe de terminaison DP-quasi simple ne constitue donc pas un cadre d'étude *général* des comportements modulaires de la terminaison.

La terminaison DP-quasi simple montre un bon comportement dans le cas des unions disjointes de systèmes respectant certaines restrictions vis-à-vis de l'AFS.

Théorème 15. (Giesl & Ohlebusch 98)
La terminaison DP-quasi simple est une propriété modulaire pour les unions disjointes de systèmes finis tels que pour l'AFS P utilisé :

– Pour toute règle $l \to r$, $l{\downarrow_P} \notin X$ ou alors $l{\downarrow_P} = r{\downarrow_P}$,
– Pour toute paire $\langle s, t\rangle$ sur une partie strictement fortement connexe, $s{\downarrow_P} \notin X$ ou alors $s{\downarrow_P} = t{\downarrow_P}$.

L'exemple suivant de Giesl & Ohlebusch [29] met par contre en évidence son comportement non modulaire pour les unions à constructeurs partagés.

Exemple 22.

$$R_1 \ : \ \begin{cases} f(b(x)) & \to & x \\ f(c(x)) & \to & f(x) \end{cases} \qquad R_2 \ : \ \begin{cases} g(d(x)) & \to & g(x) \\ g(c(x)) & \to & c(g(b(c(x)))). \end{cases}$$

R_1 et R_2 terminent DP-*quasi simplement dans la mesure où R_1 termine simplement et où il suffit de l'AFS $\{b(x) \to b'\}$ et d'un RPO pour montrer la terminaison de R_2. Il n'existe pourtant pas de préordre de simplification $\geq$ tel que :*

$$\begin{array}{rcll} \widehat{f}(c(x)) & > & \widehat{f}(x) & \text{(issu de la seule paire sur un cycle),} \quad \text{(I.1)} \\ f(c(x)) & \geq & f(x), & \text{(I.2)} \\ f(b(x)) & \geq & x, & \text{(I.3)} \\ g(d(x)) & \geq & g(x), & \text{(I.4)} \\ g(c(x)) & \geq & c(g(b(c(x)))). & \text{(I.5)} \end{array}$$

En effet, supposons qu'il existe un préordre de simplification respectant toutes ces contraintes et regardons le terme $\widehat{f}(c(g(c(x))))$. On a par la première contrainte $\widehat{f}(c(g(c(x)))) > \widehat{f}(g(c(x)))$ mais alors par (I.5) puis sous-terme on déduit que $\widehat{f}(g(c(x))) \geq \widehat{f}(c(g(b(c(x))))) \geq \widehat{f}(c(g(c(x))))$ et donc $\widehat{f}(c(g(c(x)))) > \widehat{f}(c(g(c(x))))$

ce qui est impossible. Il faut donc qu'il y ait une application d'un AFS[5] *(disons P) avant la comparaison par le préordre de simplification. Remarquons que le contre-exemple précédent est toujours valable si les règles de cet AFS ne concernent que f ou* d.

L'inégalité stricte (I.1) interdit à c *d'être un symbole défini (utilement) dans P. On ne peut pas non plus éliminer le seul argument de* b *sans perdre la condition d'inclusions d'ensembles de variables de la définition 56 pour la contrainte (I.3). L'argument de g doit aussi rester car (I.5) deviendrait $g' \geq c(g')$ et l'inégalité stricte (I.1) serait compromise. Avec la projection $b(x) \to x$, on aurait par (I.5) $g(c(x)) \geq c(g(c(x)))$ et par suite $\widehat{f}(c(g(c(x)))) > \widehat{f}(c(g(c(x))))$; avec $g(x) \to x$, (I.5) donnerait $c(x) \geq c(b(c(x)))$ et donc $\widehat{f}(c(c(x))) > \widehat{f}(c(x)) \geq \widehat{f}(c(b(c(x))) \geq \widehat{f}(c(c(x)))$. Il n'existe pas non plus d'AFS convenable, ce système ne termine donc pas* DP*-quasi simplement.*

Le cas innermost.

Il est possible de spécialiser ces résultats à la terminaison innermost afin d'en affaiblir les hypothèses — la stratégie innermost étant un terrain très favorable à l'approche par paires de dépendance — et de les utiliser dans les cas où la terminaison innermost des systèmes entraîne leur normalisation forte.

Afin de limiter les contraintes à un sous-ensemble seulement des règles, Arts & Giesl définissent la notion de *règle utilisable*.

Définition 57. — Soit R un système de récriture. Posons $R_\Lambda(f) = \{l \to r \in R \mid \Lambda(l) = f\}$.
L'ensemble $\mathcal{U}_R(t)$ des *règles utilisables* pour un terme t est défini par :

$$\begin{array}{l} \mathcal{U}_R(x) = \emptyset \text{ pour } x \in X, \\ \mathcal{U}_R(f(t_1, \ldots, t_n)) = R_\Lambda(f) \cup \bigcup_{l \to r \in R_\Lambda(f)} \mathcal{U}_{R \setminus R_\Lambda(f)}(r) \cup \bigcup_{j=1}^n \mathcal{U}_{R \setminus R_\Lambda(f)}(t_j). \end{array}$$

On définit en outre pour tout ensemble $\mathcal{C}$ de paires de dépendance de R :

$$\mathcal{U}_R(\mathcal{C}) = \bigcup_{\langle s,t \rangle \in \mathcal{C}} \mathcal{U}_R(t).$$

La contrainte de décroissance large des règles est maintenant limitée aux seules règles utilisables de chaque partie strictement fortement connexe.

Théorème 16. (Arts & Giesl 98)
Soit R un système de récriture. Si pour chaque partie strictement fortement connexe $\mathcal{C}$ de son graphe de dépendance il existe un préordre bien fondé, stable et faiblement monotone $\geq_\mathcal{C}$ tel que :

1. $l \geq_\mathcal{C} r$ pour toute règle $l \to r \in \mathcal{U}_R(\mathcal{C})$,
2. $s \geq_\mathcal{C} t$ pour toute paire $\langle s, t \rangle \in \mathcal{C}$,
3. $s >_\mathcal{C} t$ pour au moins une paire $\langle s, t \rangle \in \mathcal{C}$,

Alors R termine pour la stratégie innermost.

[5] Et cet AFS ne doit pas se résumer à un renommage des symboles.

I.2.6 Propriétés de la terminaison $\mathcal{C}_{\mathcal{E}}$

La terminaison $\mathcal{C}_{\mathcal{E}}$ possède de bonnes propriétés quant aux unions de systèmes. Elle est en particulier modulaire pour les unions de systèmes :

– Disjoints (Gramlich, Ohlebusch) [37,69],

– À branchement fini et constructeurs partagés (Gramlich) [37],

– À branchement fini et composables (Kurihara & Ohuchi) [52].

La restriction aux systèmes à branchement fini est nécessaire dans le cas de systèmes partageant des constructeurs comme l'illustre l'exemple suivant dû à Ohlebusch [69].

Exemple 23.
Considérons les deux systèmes :

$$R_1 \quad : \quad \left\{ \begin{array}{lcl} f_j(c_j,x) & \to & f_{j+1}(x,x) \\ f_j(x,y) & \to & x \\ f_j(x,y) & \to & y \end{array} \right| j \in \mathbb{N} \quad \text{et} \quad R_2 \quad : \quad \left\{ \begin{array}{lcl} G(x,y) & \to & x \\ G(x,y) & \to & y \\ a & \to & c_j \end{array} \right| j \in \mathbb{N}.$$

Ces deux systèmes sont $\mathcal{C}_{\mathcal{E}}$ fortement normalisants[6] *mais leur union permet la réduction infinie :*

$$f_1(c_1,a) \underset{R_1}{\longrightarrow} f_2(a,a) \underset{R_2}{\longrightarrow} f_2(c_2,a) \underset{R_1}{\longrightarrow} f_3(a,a) \to \dots$$

Dans la mesure où le cas des unions de composables englobe celui des constructeurs partagés, cette restriction est également impérative pour les systèmes composables ainsi que dans toute extension plus générale.

Nous nous restreindrons donc à l'étude des systèmes à branchement fini.

Proposition 5. (Gramlich 94) [37]
Un système R fortement normalisant qui ne duplique pas ses variables termine $\mathcal{C}_{\mathcal{E}}$.

PREUVE. Soit $\#_G(t)$ le nombre d'occurrences du symbole G dans le terme t et $\geq_\#$ un ordre tel que $s \geq_\# t$ si $\#_G(s) \geq_{\mathbb{N}} \#_G(t)$ où $\geq_{\mathbb{N}}$ est l'ordre naturel sur $\mathbb{N}$.
Toute réduction par $R \cup \pi$ fait décroître strictement $(\geq_\#, \to_R^+)_{\mathsf{lex}}$. □

Proposition 6. (Gramlich 94) [37]
Un système R projectif ND et fortement normalisant termine $\mathcal{C}_{\mathcal{E}}$.

PREUVE. Par contradiction. Supposons qu'il existe une réduction infinie par $R \cup \pi$. Par hypothèse sur R nous savons qu'il existe un terme α contenant au moins deux variables qui peut se réduire par R en l'une et aussi en l'autre. En remplaçant chaque occurrence de $G(x,y)$ instanciée par une substitution σ par $\alpha\sigma$ on obtient une réduction infinie ne faisant intervenir que des pas de R (par simulation des pas de π) ce qui est en contradiction avec la normalisation forte de R. □

Proposition 7. (Gramlich 94) [37]
Un système non dupliquant qui termine est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

[6]Nous verrons pourquoi à la proposition 8.

Proposition 8. (Gramlich 94) [37]
Un système R qui termine simplement est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

De cette dernière proposition, également vérifiée dans le cas des systèmes à branchement infini, on peut déduire que puisque les deux systèmes de l'exemple 23 terminent simplement, ils sont en fait $\mathcal{C}_{\mathcal{E}}$ fortement normalisants.

La terminaison DP-quasi simple est en fait déjà plus forte que la terminaison $\mathcal{C}_{\mathcal{E}}$.

Proposition 9. (Ohlebusch 01)
Un système R fini termine DP-quasi simplement si et seulement si $R \cup \pi$ termine DP-quasi simplement.

On a donc l'implication de la terminaison $\mathcal{C}_{\mathcal{E}}$ par la terminaison DP-quasi simple. La réciproque n'est pas vraie.

Exemple 24.
Ohlebusch montre que le système R fortement normalisant suivant ne termine pas DP-*quasi simplement.*

$$R \ : \ \begin{cases} h(g(x)) & \to \ x \\ g(c) & \to \ c \\ f(d,e,x) & \to \ f(x,g(e),e) \\ f(d,e,x) & \to \ f(g(d),x,d). \end{cases}$$

Puisqu'il est non dupliquant, il termine pourtant $\mathcal{C}_{\mathcal{E}}$ par la proposition 7.

Plus générale que les terminaisons simple et DP-quasi simple, la terminaison $\mathcal{C}_{\mathcal{E}}$ n'est pas trop restrictive en pratique. On peut remarquer à ce sujet que tous les systèmes proposés par Arts & Giesl comme illustration de la puissance des paires de dépendance sont $\mathcal{C}_{\mathcal{E}}$ fortement normalisants. C'est pourquoi, plutôt que nous limiter — comme au cours des travaux antérieurs — à l'étude de classes particulières d'extensions, nous choisissons d'étudier cette version plus forte de la notion de terminaison qui, sans vraiment nous pénaliser, autorise de nouvelles approches des preuves.

La figure I.3 résume les relations d'implication des différentes notions de terminaison.

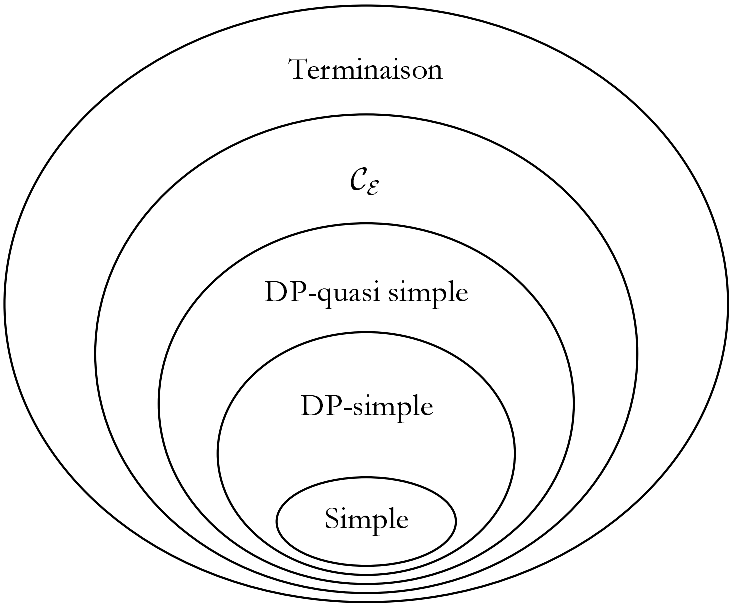


FIG. I.3. Relation entre les différentes notions de terminaison.

Chapitre II

Modules de récriture

Nous allons maintenant définir les *modules de récriture* et voir de quelle manière ils permettent une mise en évidence de la structure hiérarchique des systèmes de récriture.

II.1 Modules et extensions

D'un point de vue opérationnel, les modules sont des ensembles de nouveaux symboles accompagnés des règles qui les définissent.

Définition 58. — Soit R_1 un système de récriture sur une signature $\mathcal{F}_1$. Un *module étendant* R_1 est un couple $[\mathcal{F}_2 \mid R_2]$ tel que :

1. $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$ (signatures disjointes) ;
2. R_2 est un système de récriture sur $\mathcal{F}_1 \cup \mathcal{F}_2$;
3. Pour toute règle $l \to r$ de R_2, $\Lambda(l) \in \mathcal{F}_2$.

Il est facile de voir que $R_1 \cup R_2$ est bien un système de récriture sur $\mathcal{F}_1 \cup \mathcal{F}_2$. On dit alors que le système $R_1 \cup R_2$ sur $\mathcal{F}_1 \cup \mathcal{F}_2$ est une *extension hiérarchique du système $R_1(\mathcal{F}_1)$ par le module* $[\mathcal{F}_2 \mid R_2]$, extension notée :

$$R_1(\mathcal{F}_1) \longleftarrow [\mathcal{F}_2 \mid R_2].$$

Remarque 12. — En considérant que la flèche associe vers la base de la hiérarchie, nous écrirons parfois simplement $[\mathcal{F}_0 \mid R_0] \longleftarrow [\mathcal{F}_1 \mid R_1]$ pour désigner (abusivement) l'extension de la hiérarchie que $[\mathcal{F}_0 \mid R_0]$ domine par le module $[\mathcal{F}_1 \mid R_1]$.

Les habituelles notions d'unions peuvent se retrouver exprimées de façon directe en tant qu'extensions de modules. À cet effet on posera qu'un module $[\mathcal{F}_1 \mid R_1]$ *étend* $[\mathcal{F}_0 \mid R_0]$ *indépendamment de* $[\mathcal{F}_2 \mid R_2]$ si :

- $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$,
- $[\mathcal{F}_0 \mid R_0] \longleftarrow [\mathcal{F}_1 \mid R_1]$ et
- $[\mathcal{F}_0 \mid R_0] \longleftarrow [\mathcal{F}_2 \mid R_2]$.

Une telle extension n'est autre qu'une *union de systèmes composables* [52, 62, 63, 70]. Nous parlerons ainsi de l'*union disjointe* $R_1 \cup R_2$ si $[\mathcal{F}_1 \mid R_1]$ et $[\mathcal{F}_2 \mid R_2]$ étendent $[\emptyset \mid \emptyset]$ indépendamment. Enfin, l'union sera *à constructeurs partagés* si $[\mathcal{F}_1 \mid R_1]$ et $[\mathcal{F}_2 \mid R_2]$ étendent indépendamment $[\mathcal{F}_0 \mid \emptyset]$.

Les modules permettant de visualiser la structure hiérarchique des systèmes, il peut être plus opportun de les présenter sous forme graphique. Une extension $R_1(\mathcal{F}_1) \longleftarrow [\mathcal{F}_2 \mid R_2]$ sera alors décrite par :

$$\begin{tikzcd}
\begin{array}{|c|}\hline \mathcal{F}_1 \\ \hline R_1 \\ \hline \end{array} \\
\begin{array}{|c|}\hline \mathcal{F}_2 \\ \hline R_2 \\ \hline \end{array} \arrow[u]
\end{tikzcd}$$

Quant à une extension de $R_0(\mathcal{F}_0)$ par $[\mathcal{F}_1 \mid R_1]$ indépendamment de $[\mathcal{F}_2 \mid R_2]$, elle sera représentée :

$$\begin{tikzcd}
& \begin{array}{|c|}\hline \mathcal{F}_0 \\ \hline R_0 \\ \hline \end{array} & \\
\begin{array}{|c|}\hline \mathcal{F}_1 \\ \hline R_1 \\ \hline \end{array} \arrow[ur] & & \begin{array}{|c|}\hline \mathcal{F}_2 \\ \hline R_2 \\ \hline \end{array} \arrow[ul]
\end{tikzcd}$$

Remarque 13. — La symétrie de la représentation ne nous fait pas perdre d'information au niveau hiérarchique puisque par définition les modules sont indépendants.

Exemple 25.

Considérons un système calculant l'addition et la multiplication dans l'arithmétique de Peàno.

$$\begin{array}{rcl}
\mathcal{F} & = & \{s, 0, +, \times\} \\
R & : & \left\{ \begin{array}{lcl} x + 0 & \to & x \\ x + s(y) & \to & s(x+y) \\ x \times 0 & \to & 0 \\ x \times s(y) & \to & x + (x \times y). \end{array} \right.
\end{array}$$

Une façon naturelle (en tout cas pour un programmeur) de comprendre ce système est de considérer l'addition bâtie sur les constructeurs s et 0 et la multiplication définie à partir *de l'addition et des constructeurs. Cette vision du système, replacée dans le formalisme des modules, consiste à introduire le symbole $+$ et les règles qui le définissent* puis $\times$ *et les règles de multiplication. On peut ainsi obtenir le découpage de la figure II.1.*

Différentes expressions de la hiérarchie sont cependant possibles. Remarquons en particulier que tout système $R(\mathcal{F})$ peut être considéré comme une extension du système vide sur les constructeurs par le module constitué des symboles définis avec *toutes* les règles de R.

II.2 Décomposition en modules

Un système de récriture peut toujours être soumis à l'étude comme une certaine hiérarchie de modules qui sera utilisée telle quelle. Il existe toutefois une représentation canonique : les systèmes

$$
\begin{array}{lcl}
\mathcal{F}_0 & = & \{s, 0\} \\
R_0 & : & \emptyset \\
\\
\mathcal{F}_1 & = & \{+\} \\
R_1 & : & \left\{ \begin{array}{lcl} x + 0 & \to & x \\ x + s(y) & \to & s(x + y) \end{array} \right. \\
\\
\mathcal{F}_2 & = & \{\times\} \\
R_2 & : & \left\{ \begin{array}{lcl} x \times 0 & \to & 0 \\ x \times s(y) & \to & x + (x \times y) \end{array} \right.
\end{array}
$$

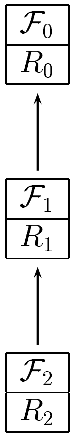


FIG. II.1. Découpage en modules d'un système d'addition & multiplication.

peuvent en effet être automatiquement présentés en tant qu'extensions de modules *minimaux*, c'est-à-dire qui ne peuvent être vus eux-mêmes comme extensions de systèmes non vides.

Ce découpage d'un système $R(\mathcal{F})$, basé sur une représentation en graphe de relations de « dépendance » entre les symboles[1], est effectué en deux étapes.

1. On construit un graphe $\mathcal{G}$ dont les sommets sont les symboles de $\mathcal{F}$ et tel qu'il existe un arc d'un sommet x à un sommet y si et seulement s'il y a une règle $l \to r \in R$ telle que
   - $x = \Lambda(l)$ et
   - y apparaît dans l ou dans r ;
2. On assemble en « paquets » les symboles des parties fortement connexes de $\mathcal{G}$ (voir déf. 54), c'est-à-dire tous les symboles f et g tels que

$$f \underset{\mathcal{G}}{\to}^{\star} g \text{ et } g \underset{\mathcal{G}}{\to}^{\star} f.$$

Il reste à construire les modules à partir de ces paquets de symboles, ce qui est facilement réalisé en joignant pour chacun d'entre eux les règles dans lesquelles il est en tête du membre gauche. La hiérarchie des modules est alors lisible sur le graphe $\mathcal{G}$.

Remarquons qu'il n'y a pas de cycle dans la hiérarchie obtenue puisque les symboles définis de manière mutuellement récursive apparaissent dans les mêmes paquets et donc, à terme, dans les mêmes modules. Cette décomposition est clairement unique.

Remarque 14. — Il est toujours possible de rassembler à des fins de clarté les symboles non définis pouvant être joints à partir des mêmes paquets.

La décomposition présentée figure II.1 est en particulier minimale en tenant compte de la remarque 14.

[1]Il s'agit formellement d'une dépendance syntaxique entre les symboles de fonctions (définis) et les symboles présents dans la définition des premiers.

Chapitre III

Terminaison et modules

Nous disposons à présent d'un cadre d'étude des systèmes permettant de tirer profit de leur structure hiérarchique. Afin de nous concentrer sur la terminaison tout respectant notre politique affichée : restriction sur la notion de terminaison plutôt que sur les extensions, il nous faut disposer d'ordres permettant de cerner au mieux la terminaison $\mathcal{C}_\mathcal{E}$. Nous introduisons donc la notion d'ordres *π-extensibles*.

Il reste à développer une approche de preuve de terminaison. Les *paires de dépendance relatives* que nous proposons section III.2 vont en fait concentrer les avantages respectifs de la représentation hiérarchique en modules et de $\mathcal{C}_\mathcal{E}$ grâce à l'utilisation des ordres π-extensibles. Nous serons alors prêts à aborder les preuves de façon incrémentale.

Rappelons que pour toute signature $\mathcal{F}$ on a *toujours* dans la suite $G \notin \mathcal{F}$.

III.1 Ordres π-extensibles

III.1.1 Des ordres pour $\mathcal{C}_\mathcal{E}$

Définition 59. — Nous dirons d'un ordre sur les termes $(\succeq, \succ)$ sur $T(\mathcal{F}, X)$ qu'il est *π-extensible* s'il existe un ordre de réduction $(\succeq', \succ')$ sur $T(\mathcal{F} \cup \{G : 2\}, X)$ tel que :

1. La restriction de $(\succeq', \succ')$ à $T(\mathcal{F}, X)$ est exactement $(\succeq, \succ)$;
2. $G(s,t) \succeq' s$ et $G(s,t) \succeq' t$ pour tous s et t de $T(\mathcal{F} \cup \{G : 2\}, X)$.

Un $(\succeq', \succ')$ idoine sera désigné comme *ordre associé à* $(\succeq, \succ)$.

Un ordre π-extensible est *fortement π-extensible* si lui-même et un ordre associé sont tous deux strictement monotones. Dans le cas contraire nous parlerons d'ordre *faiblement π-extensible*.

Ainsi définis, les ordres π-extensibles remplissent bien la tâche pour laquelle ils sont conçus.

Théorème 17.
Si $(\succeq, \succ)$ est un ordre π-extensible strictement monotone et si, pour toute règle $l \to r$ d'un système de récriture R, $l \succ r$, alors R est $\mathcal{C}_\mathcal{E}$ fortement normalisant.

III.1.2 Construction des ordres π-extensibles

Les ordres π-extensibles existent ! En effet tout ordre de simplification est π-extensible, en particulier RPO et les ordres induits par interprétations polynomiales conviennent. Nous proposons dans la suite quelques combinaisons d'ordres π-extensibles permettant d'en obtenir de nouveaux.

Proposition 10.
Si $(\succeq_1, \succ_1)$ est un ordre fortement π-extensible et si $(\succeq_2, \succ_2)$ est π-extensible, alors leur composition lexicographique $((\succeq_1, \succ_1), (\succeq_2, \succ_2))_{lex}$ est π-extensible.

En outre, si $(\succeq_2, \succ_2)$ est lui-même fortement π-extensible, alors $((\succeq_1, \succ_1), (\succeq_2, \succ_2))_{lex}$ est également fortement π-extensible.

PREUVE. Il nous faut trouver un ordre $(\succeq', \succ')$ qui satisfait les deux conditions de la définition 59. Supposons que $(\succeq'_1, \succ'_1)$, strictement monotone, et $(\succeq'_2, \succ'_2)$ sont des ordres associés respectivement à $(\succeq_1, \succ_1)$ et $(\succeq_2, \succ_2)$. On peut alors vérifier que $(\succeq', \succ') = ((\succeq'_1, \succ'_1), (\succeq'_2, \succ'_2))_{lex}$ convient, la monotonie stricte de $(\succeq'_1, \succ'_1)$ entraînant la (faible) monotonie de la composition.

Il faut vérifier trois choses :

1. L'orientation correcte de $G(s,t)$ et s pour tous s et t ;
2. L'orientation correcte de $G(s,t)$ et t pour tous s et t ;
3. L'égalité de la restriction à $T(\mathcal{F}, X)$ de $(\succeq', \succ')$ et de $((\succeq_1, \succ_1), (\succeq_2, \succ_2))_{lex}$.

- $G(s,t) \succeq' s$. En effet, de trois choses l'une, ou bien $G(s,t) \succ'_1 s$, ou bien $G(s,t) \succeq'_1 s$ et $G(s,t) \succ'_2 s$, ou bien $G(s,t) \succeq'_1 s$ et $G(s,t) \succeq'_2 s$. Dans tous ces cas, par définition de la composition lexicographique, on a $G(s,t) \succeq' s$.
- $G(s,t) \succeq' t$. Similaire au cas précédent.
- $(\succeq', \succ')|_{T(\mathcal{F},X)} \subseteq ((\succeq_1, \succ_1), (\succeq_2, \succ_2))_{lex}$. Détaillons le cas $s \succeq' t$ pour s et t termes de $T(\mathcal{F}, X)$: soit $s \succ'_1 t$ et par hypothèse $s \succ_1 t$; soit $s \succeq'_1 t$ et il reste deux cas 1) $s \succ'_2 t$ mais alors par hypothèse $s \succ_2 t$ et 2) $s \succeq'_2 t$ mais alors par hypothèse $s \succeq_2 t$, c.q.f.d.
- $(\succeq', \succ')|_{T(\mathcal{F},X)} \supseteq ((\succeq_1, \succ_1), (\succeq_2, \succ_2))_{lex}$. Immédiat.

Un raisonnement similaire permet de traiter le cas $\succ$. □

À l'instar des compositions lexicographiques, les RPS permettent aussi de construire des ordres π-extensibles.

Proposition 11.
Si $(\succeq_1, \succ_1)$ est un ordre π-extensible et si P est un RPS sur $\mathcal{F}$ tel que $G \notin \mathcal{F}$, alors $(\succeq, \succ)$ défini par $s \succeq t$ (resp. $\succ$) si et seulement si $s{\downarrow_P} \succeq_1 t{\downarrow_P}$ (resp. $\succ_1$) est π-extensible.

PREUVE. Il nous faut de nouveau trouver un ordre $(\succeq', \succ')$ satisfaisant les deux conditions de la définition 59. Supposons que $(\succeq'_1, \succ'_1)$ est un ordre associé à $(\succeq_1, \succ_1)$, alors $(\succeq', \succ')$ défini par $s \succeq' t$ (resp. $\succ'$) si et seulement si $s{\downarrow_P} \succeq'_1 t{\downarrow_P}$ (resp. $\succ'_1$) convient.

Vérifions dans un premier temps l'orientation correcte des règles de π. Puisque $G(s{\downarrow_P}, t{\downarrow_P}) \succeq'_1 s{\downarrow_P}$ par π-extensibilité de $\succeq_1$ nous avons bien par définition $G(s,t) \succeq' s$. Le même raisonnement tient pour la comparaison à t.

Il reste à montrer que $(\succeq', \succ')|_{T(\mathcal{F},X)} = (\succeq, \succ)$.

- $(\succeq', \succ')|_{T(\mathcal{F},X)} \subseteq (\succeq, \succ)$. Si $s \succeq' t$ nous avons $s{\downarrow_P} \succeq'_1 t{\downarrow_P}$. Or $G \notin \mathcal{F}$ donc s et t sont des termes de $T(\mathcal{F}, X)$. Par π-extensibilité de $\succeq_1$ nous déduisons alors $s{\downarrow_P} \succeq_1 t{\downarrow_P}$ c'est-à-dire $s \succeq t$.
- $(\succeq', \succ')|_{T(\mathcal{F},X)} \supseteq (\succeq, \succ)$. Si $s \succeq t$ alors puisque $G \notin \mathcal{F}$, s et t sont des termes de $T(\mathcal{F}, X)$. Ainsi $s{\downarrow_P} \succeq_1 t{\downarrow_P}$ d'où $s{\downarrow_P} \succeq'_1 t{\downarrow_P}$ ce qui par définition signifie $s \succeq' t$.

Un raisonnement similaire permet de traiter le cas $\succ$. □

III.2 Paires de dépendance de modules

Maintenant que sont posées les bases de notre approche modulaire, nous voilà prêts à l'utiliser pour prouver la terminaison des systèmes.

La plupart des méthodes utilisent implicitement une notion de rang de termes, c'est-à-dire un indice sur la façon dont les différentes signatures en jeu sont entrelacées au sein du terme. Dans notre formalisme, ces signatures figurent déjà chacune dans différents modules de la hiérarchie. On aimerait donc retrouver l'information d'entrelacement dès l'examen de la structure hiérarchique du système considéré.

Nous allons définir dans cette section les *paires de dépendance relatives*, concernant les modules, avec deux motivations : tout d'abord inclure l'information d'entrelacement dans une sélection de sous-termes pertinents, ensuite, au cours d'une conduite incrémentale de la preuve, écarter de celle-ci les symboles qui n'interviennent pas au niveau considéré.

Définition 60. — Considérons l'extension hiérarchique $R_1(\mathcal{F}_1) \longleftarrow [\mathcal{F}_2 \mid R_2]$. Pour toute règle $l \to r \in R_2$, une paire $\langle l, r' \rangle$ où r' est un sous-terme de r tel que $\Lambda(r') \in \mathcal{F}_2$ est une *paire de dépendance du module* $[\mathcal{F}_2 \mid R_2]$.

Nous noterons $\mathrm{DP}(M)$ l'ensemble de toutes les paires de dépendance du module M

Exemple 26.

L'ensemble des paires de dépendance du module définissant la multiplication de l'exemple 25 est réduit à une seule paire :

$$\mathrm{DP}([\mathcal{F}_\times \mid R_\times]) : \{ \ \langle x \times s(y), x \times y \rangle \ \}.$$

On peut remarquer que, contrairement à ce qui serait fait dans l'approche non modulaire d'Arts & Giesl, $\langle x \times s(y), (x \times y) + x \rangle$ *n'est* pas *considérée comme une paire de dépendance.*

La définition de paires de dépendance avec symboles marqués est tout à fait similaire.

Définition 61. — Considérons l'extension hiérarchique $R_1(\mathcal{F}_1) \longleftarrow [\mathcal{F}_2 \mid R_2]$. Pour toute règle $l \to r \in R_2$, une paire $\langle \widehat{l}, r' \rangle$ où r' est un sous-terme de r tel que $\Lambda(r') = \widehat{f}$ avec $f \in \mathcal{F}_2$ est une *paire de dépendance (marquée) du module* $[\mathcal{F}_2 \mid R_2]$.

Sauf mention du contraire, nous ne considérerons dans la suite que des paires avec symboles marqués. En l'absence d'ambiguïté, nous utiliserons en particulier la notation $\mathrm{DP}(M)$ pour désigner l'ensemble de toutes les paires de dépendance marquées du module M.

L'exemple 26 devient :

Exemple 27.

L'ensemble des paires de dépendance marquées du module définissant la multiplication de l'exemple 25 est réduit à une seule paire :

$$\mathrm{DP}([\mathcal{F}_\times \mid R_\times]) : \{ \ \langle x \widehat{\times} s(y), x \widehat{\times} y \rangle \ \}.$$

Remarque 15. — Dans le cas d'un système R vu comme une extension des constructeurs par le module des règles et symboles définis, les paires de dépendance relatives sont exactement les paires de dépendance au sens d'Arts & Giesl.

Définition 62. — Une *chaîne de dépendance d'un module* $[\mathcal{F} \mid R]$ *sur un système* R' (avec $R \subseteq R'$) est une séquence de paires $\ldots \langle s_j, t_j \rangle \ldots$ ($\langle s_j, t_j \rangle \in \mathrm{DP}([\mathcal{F} \mid R])$) munie d'une substitution σ telle que pour tout couple $\langle s_i, t_i \rangle$, $\langle s_{i+1}, t_{i+1} \rangle$ de paires successives, on a :

$$t_i\sigma \xrightarrow[R']{\neq\Lambda}{}^{\star} s_{i+1}\sigma.$$

Nous considérerons dans toute la suite des chaînes de dépendance *minimales*, au sens où tout sous-terme propre d'une σ-instance des membres gauches est fortement normalisable par R'.

On peut tout à fait exprimer la $\mathcal{C}_\mathcal{E}$ terminaison en termes de chaînes de dépendance de modules.

Proposition 12.
Un système de récriture $R(\mathcal{F})$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant si et seulement s'il n'existe aucune chaîne infinie de $[\mathcal{F} \mid R]$ sur $R \cup \pi$.

PREUVE. Puisque G n'appartient pas à $\mathcal{F}$ et que les membres droits de π sont des variables, $\mathrm{DP}(R \cup \pi) = \mathrm{DP}(R)$ et on peut alors prouver la terminaison $\mathcal{C}_\mathcal{E}$ de R en utilisant la remarque 15. □

On peut tirer de cette remarque un moyen de tester la terminaison.

Corollaire.
Si $(\succeq, \succ)$ est un ordre (faiblement) π-extensible tel que :

1. $l \succeq r$ pour toute règle $l \to r \in R$,
2. $s \succ t$ pour toute paire $\langle s, t \rangle \in \mathrm{DP}(R)$,

Alors R est $\mathcal{C}_\mathcal{E}$ fortement normalisable.

Chapitre IV

Preuves incrémentales

Disposant désormais d'un schéma hiérarchique, les modules, et d'un moyen syntaxique de preuve de terminaison dans ce formalisme, les paires de dépendance relatives, nous pouvons nous consacrer à la conduite incrémentale des preuves de terminaison des systèmes de récriture généraux.

Nous étudions d'abord les extensions d'un système par un seul module, puis le cas où un module étend un système indépendamment d'un autre. Le cas général n'est en effet qu'une combinaison de ces deux schémas.

IV.1 Un module pour un système

L'extension d'un système par un seul module est, si l'on peut dire, le cas de base ; une extension hiérarchique entre typiquement dans ce cadre.

Théorème 18.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_\mathcal{E}$ fortement normalisant,
2. S'il n'existe pas de chaîne infinie de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2$,

Alors $R_1 \cup R_2$ est fortement normalisant.

PREUVE. Par contradiction. Nous allons supposer qu'il existe une chaîne de dépendance infinie de $R_1 \cup R_2$ et montrer que nous pouvons alors conclure :

- Soit à l'existence d'une chaîne infinie de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2$, contredisant la deuxième hypothèse ;
- Soit à la non-terminaison $\mathcal{C}_\mathcal{E}$ de R_1 qui contredit cette fois la première prémisse du théorème 18.

Supposons que $R_1 \cup R_2$ ne termine pas. Il existe alors une chaîne infinie de $[\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2]$ sur $R_1 \cup R_2$ (les symboles marqués n'apparaissent qu'en tête). Parmi les paires de $\mathrm{DP}([\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2])$ on trouve :

① Les paires de $[\mathcal{F}_1 \mid R_1]$;

② Les paires de $[\mathcal{F}_2 \mid R_2]$;

③ Enfin les paires « hybrides » c'est-à-dire les paires $\langle \widehat{l}, \widehat{r'} \rangle$ telles que $l \to r \in R_2$ et r' est un sous-terme de r dont le symbole de tête $\Lambda(r')$ appartient à $\mathcal{F}_1$.

En ce qui concerne les deux premiers cas nous disposons d'informations dues en particulier aux hypothèses. Afin d'éviter le dernier cas, nous avons besoin d'un lemme.

Lemme 7.
Soient R_1 un TRS sur $\mathcal{F}_1$ et $[\mathcal{F}_2 \mid R_2]$ un module tel que $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$.

Alors pour toute paire $\langle u_1, v_1\rangle \in \mathrm{DP}(R_1)$ et toute paire $\langle u_2, v_2\rangle$ de $[\mathcal{F}_2 \mid R_2]$ il n'existe pas de substitution σ telle que :

$$v_1\sigma \xrightarrow{\neq\Lambda}{}^{\star} u_2\sigma.$$

PREUVE. Puisque $\Lambda(u_2\sigma) = \Lambda(u_2) \in \widehat{\mathcal{F}_2}$ et $\Lambda(v_1\sigma) = \Lambda(v_1) \in \widehat{\mathcal{F}_1}$, on a forcément $\Lambda(u_2) \neq \Lambda(v_1)$. □

Ce lemme nous apprend que les paires ② et ③ ne peuvent suivre que des ② et que les paires ① ne peuvent succéder qu'à celles des genres ① et ③. Nous nous trouvons donc face à trois possibilités : la chaîne de dépendance ne contient

1. Que des paires ②, c'est-à-dire du module $[\mathcal{F}_2 \mid R_2]$ ou
2. Que des paires ①, c'est-à-dire du module $[\mathcal{F}_1 \mid R_1]$ ou encore
3. Qu'un nombre fini (éventuellement nul) de paires ② suivies d'*une seule* paire ③ *puis* d'une infinité de paires ①.

- Premier cas : l'existence d'une chaîne infinie de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2$ contredit directement la deuxième hypothèse du théorème 18.
- Cas 2 & 3 : dans ces deux cas il y a une chaîne infinie de $[\mathcal{F}_1 \mid R_1]$ sur $R_1 \cup R_2$. Nous allons montrer dans la suite que cette chaîne peut être transformée en une chaîne de $[\mathcal{F}_1 \mid R_1]$ sur $R_1 \cup \pi$. Ce faisant, nous obtiendrons une chaîne infinie de $[\mathcal{F}_1 \mid R_1]$ sur $R_1 \cup \pi$, c'est-à-dire une chaîne infinie de $[\mathcal{F}_1 \cup \{G\} \mid R_1 \cup \pi]$ sur $R_1 \cup \pi$ qui contredit la première hypothèse du théorème, à savoir que R_1 est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

  Nous nous servons dans cette preuve, ainsi que dans celle du théorème 19 (section IV.2), d'un résultat plus général présenté comme un lemme technique clef, le lemme 8. Sa démonstration fournit un moyen d'obtenir la transformation désirée.

  Lemme 8.
  Soient S_1 et S_2 deux systèmes de récriture sur une signature $\mathcal{F}_1$. Soit S_3 un TRS sur $\mathcal{F}_1 \cup \mathcal{F}_2$ tel que :

  – $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$;
  – Pour toute règle $l \to r \in S_3$, $\Lambda(l) \in \mathcal{F}_2$.

  Alors d'une chaîne infinie minimale de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup S_3$, il est possible d'obtenir une chaîne infinie de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup \pi$ comportant la même séquence de paires mais munie d'une nouvelle substitution ainsi que de nouveaux pas de récriture.

Ce qui reste de la preuve du théorème 18 consiste essentiellement à appliquer le lemme 8 en prenant $R_1 = S_1 = S_2$ et $R_2 = S_3$. Nous pouvons ainsi construire pour toute chaîne de dépendance infinie de $[\mathcal{F}_1 \mid R_1]$ sur $R_1 \cup R_2$ une chaîne correspondante de $[\mathcal{F}_1 \mid R_1]$ sur $R_1 \cup \pi$, c'est-à-dire une chaîne de dépendance infinie de $R_1 \cup \pi$. Dans la mesure où nous avons supposé R_1 $\mathcal{C}_\mathcal{E}$ fortement normalisant, ce résultat contredit les hypothèses. Le théorème 18 est prouvé. $\square$

Preuve du lemme 8

Nous sommes amenés pour cette preuve à interpréter les termes de façon similaire à ce qui fut proposé par B. Gramlich en 1991 et 1994 [36, 37] (cf. remarque 17, page 76).

L'ensemble des termes infinis sur une signature $\mathcal{F}$ et un ensemble de variable X est désigné par $T_\infty(\mathcal{F}, X)$.

Définition 63. — Posons $S = S_1 \cup S_2 \cup S_3$; soit $>$ un ordre arbitraire mais *total* sur $T_\infty(\widehat{\mathcal{F}_1} \cup \{G : 2\} \cup \{\perp : 0\}, X)$.

L'interprétation $I(x) : T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X) \to T_\infty(\widehat{\mathcal{F}_1} \cup \{G : 2\} \cup \{\perp : 0\}, X)$ est définie par :

$$\begin{array}{rcl} I(x) & = & x \text{ si } x \in X, \\ I(f(t_1 \dots t_n)) & = & \left\{ \begin{array}{l} f(I(t_1) \dots I(t_n)) \text{ si } f \in \widehat{\mathcal{F}_1}, \\ \mathrm{Comb}(\mathrm{Red}(f(t_1 \dots t_n))) \text{ si } f \in \mathcal{F}_2, \end{array} \right. \end{array}$$

Où

$$\begin{array}{rcl} \mathrm{Red}(t) & = & \{I(t') / \, t \xrightarrow[S_1 \cup S_2 \cup S_3]{} t'\}, \\ \mathrm{Comb}(\emptyset) & = & \perp, \\ \mathrm{Comb}(\{a\} \cup E) & = & G(a, \mathrm{Comb}(E)) \text{ où pour tout } e \in E, a < e). \end{array}$$

$\mathrm{Red}(t)$ représente l'ensemble E des interprétations des réduits en un pas de t. Afin de lever toute ambiguïté sur la construction de l'arbre $\mathrm{Comb}(E)$ à partir de l'ensemble E qui, par définition, n'est pas ordonné, il est nécessaire d'imposer un ordre $>$ total sur E déterminant une stratégie de construction.

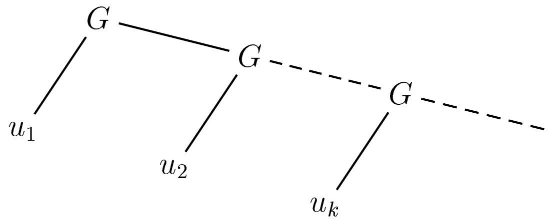


FIG. IV.1. Structure des termes interprétés.

Remarque 16. — L'arbre de combinaison des termes prend l'apparence d'un « peigne à gauche ». Comme illustré par la figure IV.1, l'interprétation d'un terme $t = f(t_1, \dots, t_n)$ avec $f \in \mathcal{F}_2$ est en fait une séquence d'interprétations (les « dents » du peigne) de ses réduits en un pas, les u_i étant les éléments de $\mathrm{Red}(t)$.

Puisque les dents sont elles-mêmes des interprétations, il est possible d'atteindre chacune d'entre elles à l'aide d'une réduction $\xrightarrow[\pi_2]{}^\star \xrightarrow[\pi_1]{}$ idoine.

Il nous faut encore quelques lemmes pour nous assurer du bon comportement de notre interprétation.

Pour une substitution σ nous désignerons par $I(\sigma)$ la substitution σ' telle que $x\sigma' = I(x\sigma)$ pour toute variable x.

Lemme 9.
Pour tout $t \in T(\widehat{\mathcal{F}_1}, X)$ et toute substitution σ,

$$I(t\sigma) = tI(\sigma).$$

PREUVE. Par induction sur la structure de t. □

Lemme 10.
Pour tous $t_1, \ldots, t_n$ de $T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X)$ et pour tout contexte C sur $\mathcal{F}_1$ à n trous,

$$I(C[t_1, \ldots, t_n]) = C[I(t_1), \ldots, I(t_n)].$$

PREUVE. Par induction sur la structure de C. □

Lemme 11.
Pour tout terme t fortement normalisable par $S_1 \cup S_2 \cup S_3$, $I(t)$ est un terme fini.

PREUVE. Immédiat puisque nous nous sommes restreints aux systèmes à branchement fini. □

Lemme 12.
Pour tous s et t de $T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X)$ et toute règle $l \to r \in S_1 \cup S_2$,

$$\text{Si } s \xrightarrow[l\to r]{p} t \text{ alors } I(s) \xrightarrow[S_1\cup S_2\cup\pi]{}^{+} I(t).$$

En outre, si $p \neq \Lambda$ et $\Lambda(s) \in \widehat{\mathcal{F}_1}$ alors $I(s) \xrightarrow[S_1\cup S_2\cup\pi]{\neq\Lambda}^{+} I(t)$.

PREUVE. On distingue deux cas suivant l'appartenance des symboles susceptibles d'apparaître sur le chemin reliant Λ à p.

1. S'il y a seulement des symboles de $\mathcal{F}_1$ alors $s = C[s_1, \ldots, l\sigma, \ldots, s_n]$, $s|_p = l\sigma$ et C est un contexte à n trous sur $\mathcal{F}_1$. On a :

$$\begin{array}{rcl} I(s) & = & I(C[s_1, \ldots, l\sigma, \ldots, s_n]) \\ & = & C[I(s_1), \ldots, I(l\sigma), \ldots, I(s_n)] \text{ (lemme 10)} \\ & = & C[I(s_1), \ldots, lI(\sigma), \ldots, I(s_n)] \text{ (lemme 9)} \\ & \xrightarrow[S_1\cup S_2]{p} & C[I(s_1), \ldots, rI(\sigma), \ldots, I(s_n)] \text{ (hypothèse)} \\ & = & C[I(s_1), \ldots, I(r\sigma), \ldots, I(s_n)] \text{ (lemme 9)} \\ & = & I(C[s_1, \ldots, r\sigma, \ldots, s_n]) \\ & = & I(t). \end{array}$$

2. Si des symboles de $\mathcal{F}_2$ interviennent, alors il existe un plus petit $p' < p$ (pour l'ordre préfixe) tel que $\Lambda(s|_{p'})$ appartienne à $\mathcal{F}_2$. Nous pouvons derechef supposer (sans perte de généralité) que $s = C[s_1, \ldots, s', \ldots, s_n]$ où C est un contexte à n trous (ou bien vide) sur $\mathcal{F}_1$, $p = p'q$ et $s|_{p'} = s'$ avec $s' = C'[l\sigma] \xrightarrow[S_1 \cup S_2]{} C'[r\sigma] = t'$. On a :

$$\begin{array}{rcl} I(s) & = & I(C[s_1, \ldots, s', \ldots, s_n]) \\ & = & C[I(s_1), \ldots, I(s'), \ldots, I(s_n)] \text{ (lemme 10).} \end{array}$$

D'après la définition 63, $I(s') = \mathrm{Comb}(\mathrm{Red}(s'))$. Or

$$s'|_q = l\sigma \xrightarrow[S_1 \cup S_2]{} r\sigma.$$

Par définition de Red, on déduit alors que $I(r\sigma) \in \mathrm{Red}(l\sigma)$. $I(t')$ est donc un sous-terme de $I(s')$ et

$$I(s') \underset{\pi}{\rightarrow}^{+} I(t').$$

Finalement,

$$\begin{array}{rcl} C[I(s_1), \ldots, I(s'), \ldots, I(s_n)] & \underset{\pi}{\rightarrow}^{+} & C[I(s_1), \ldots I(t'), \ldots, I(s_n)] \\ & = & I(C[s_1, \ldots, t', \ldots, s_n]) \text{ (lemme 10)} \\ & = & I(t). \end{array}$$

□

Lemme 13.

Pour tous s et t de $T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X)$, Si $s \xrightarrow[S_3]{p} t$ alors $I(s) \underset{\pi}{\rightarrow}^{+} I(t)$.

De plus, si $\Lambda(s) \in \widehat{\mathcal{F}_1}$ alors $I(s) \xrightarrow[\pi]{\neq\Lambda}{}^{+} I(t)$.

PREUVE. Tout à fait similaire à la preuve du lemme 12, cas 2. □

Nous pouvons à présent passer à la preuve du lemme 8.

PREUVE. Soit $\langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots$ une chaîne de dépendance de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup S_3$ munie d'une substitution σ. Soit σ' la substitution telle que pour tout x, $x\sigma' = I(x\sigma)$.

Dans la mesure où la chaîne considérée est minimale, la substitution σ est fortement normalisable, le lemme 11 garantit alors que σ' ne substitue que des termes finis.

Nous allons montrer que $\langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots$ munie de σ' est une chaîne de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup \pi$.

Pour ce faire, il nous faut prouver que pour tout i,

$$v_i\sigma' \xrightarrow[S_1 \cup S_2 \cup \pi]{\neq\Lambda}{}^{\star} u_{i+1}\sigma'.$$

Nous savons que

$$v_i\sigma \xrightarrow[S_1 \cup S_2 \cup S_3]{\neq\Lambda}{}^{\star} u_{i+1}\sigma.$$

Considérons un pas $s \xrightarrow[S_1 \cup S_2 \cup S_3]{p} t$ de cette dérivation. Puisque

$$\Lambda(s) = \Lambda(t) = \Lambda(v_i) = \Lambda(u_{i+1}) \in \widehat{\mathcal{F}_1},$$

Alors du lemme 13 ou, le cas échéant, du lemme 12 nous déduisons

$$I(s) \xrightarrow[S_1 \cup S_2 \cup \pi]{\neq \Lambda}{}^{\star} I(t).$$

Nous pouvons reconstituer la séquence en juxtaposant les pas afin d'obtenir

$$I(v_i \sigma) \xrightarrow[S_1 \cup S_2 \cup \pi]{\neq \Lambda}{}^{\star} I(u_{i+1} \sigma).$$

Dans la mesure où $I(v_i\sigma) = v_i\sigma'$ et $I(u_{i+1}\sigma) = u_{i+1}\sigma'$ le lemme 9 nous permet de conclure. □

Corollaires

En considérant l'extension du système R_1 par le module $[\mathcal{F}_2 \cup \{G : 2\} \mid R_2 \cup \pi]$ et en appliquant le théorème 18 on obtient comme corollaire une condition suffisante pour garantir la composabilité de la terminaison $\mathcal{C}_{\mathcal{E}}$.

Corollaire 18–1.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant,
2. S'il n'existe pas de chaîne infinie de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2 \cup \pi$,

Alors $R_1 \cup R_2$ est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

Enfin ces résultats sont rendus effectifs grâce à l'utilisation d'ordres idoines.

Corollaire 18–2.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant ;
2. S'il existe un ordre de réduction faible (resp. faiblement π-extensible) $(\succeq, \succ)$ tel que :
   - $R_1 \cup R_2 \subseteq \succeq$ et
   - $\mathrm{DP}([\mathcal{F}_2 \mid R_2]) \subseteq \succ$,

Alors $R_1 \cup R_2$ est fortement normalisant (resp. $\mathcal{C}_{\mathcal{E}}$ fortement normalisant).

PREUVE. Par contradiction. Supposons qu'il existe une chaîne infinie, puisque la mesure de complexité décroît largement pour chaque pas de récriture et strictement à chaque paire de dépendance, il existe une séquence d'éléments strictement décroissante pour $(\succeq, \succ)$ ce qui est en contradiction avec l'hypothèse précisant que cet ordre est bien fondé. □

Remarque 17. — Comparaison avec l'interprétation de B. Gramlich.
Notre interprétation des termes est assez proche de l'abstraction Φ utilisée par B. Gramlich en 1991 et 1994 [36, 37]. Elle s'en distingue cependant principalement par la définition de l'ensemble $\mathrm{Red}(t)$. Nous considérons en effet *tous* les réduits *en un pas* de t alors que la fonction $SUCC^{\mathcal{F}_1}(t)$ apparaissant

dans Φ sélectionne *parmi les réduits par* $\rightarrow_S^\star$ (on considère alors la clôture transitive et non la réduction en un pas) ceux dont *le symbole à la racine est dans* $\mathcal{F}_1$.

Les conséquences de ces différences portent sur la taille des termes interprétés (plus importante dans notre cas) mais ont aussi une influence sur la preuve : je conjecture en effet qu'il est possible de montrer (plus difficilement) grâce à Φ des lemmes proches des lemmes 12 et 13, à ceci près que les interprétations des deux termes s et t peuvent, en raison de la prise en compte de réductions en plus d'un pas, être identiques ; on aurait alors en conclusion

$$\Phi(s) \xrightarrow[S_1 \cup S_2 \cup \pi]{}^{\star} \Phi(t)$$

Plutôt que

$$\Phi(s) \xrightarrow[S_1 \cup S_2 \cup \pi]{}^{+} \Phi(t).$$

Remarquons toutefois que même après ce changement ces lemmes permettent toujours de prouver les théorèmes 18 et 19.

IV.2 Deux modules indépendants pour un système

Avec une extension d'un même système par deux modules, nous arrivons au cœur de la nature modulaire de la preuve. Tout d'abord parce qu'extensions hiérarchiques et unions de composables permettent ensemble de décrire le cas hiérarchique général, mais aussi parce qu'il faut éviter de compliquer les preuves avec des contraintes venues de règles inopportunes.

Théorème 19.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$, soit $[\mathcal{F}_3 \mid R_3]$ un module étendant R_1 indépendamment de R_2.

1. Si $R_1 \cup R_2$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant,
2. S'il n'existe pas de chaîne infinie de $[\mathcal{F}_3 \mid R_3]$ sur $R_1 \cup R_3 \cup \pi$,

Alors $R_1 \cup R_2 \cup R_3$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant.

Preuve. Supposons qu'il existe une chaîne de dépendance infinie de $R_1 \cup R_2 \cup R_3 \cup \pi$. Nous allons montrer que dans ce cas, nous pouvons conclure soit à la non-terminaison $\mathcal{C}_\mathcal{E}$ de $R_1 \cup R_2$ qui contredit la première hypothèse, soit à l'existence d'une chaîne infinie du module $[\mathcal{F}_3 \mid R_3]$ sur $R_1 \cup R_3 \cup \pi$ contredisant cette fois la deuxième hypothèse.

Par définition des extensions hiérarchiques et à l'aide du lemme 7, nous savons que les chaînes de $R_1 \cup R_2 \cup R_3 \cup \pi$ sont

- Soit des chaînes de $[\mathcal{F}_3 \mid R_3]$;
- Soit des chaînes de $[\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2]$ sur $R_1 \cup R_2 \cup R_3 \cup \pi = R$;
- Soit des chaînes constituées d'un nombre *fini* de paires de $[\mathcal{F}_3 \mid R_3]$, puis d'*une seule* paire hybride $\langle \widehat{s}, \widehat{t} \rangle$ telle que $\Lambda(s) \in \mathcal{F}_3$ et $\Lambda(t) \in \mathcal{F}_1 \cup \mathcal{F}_2$, enfin d'une chaîne de $[\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2]$ sur $R_1 \cup R_2 \cup R_3 \cup \pi = R$.

Il suffit donc de montrer la finitude des chaînes de $[\mathcal{F}_3 \mid R_3]$ sur R et des chaînes de $[\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2]$ sur R.

- Il n'existe pas de chaîne infinie de $[\mathcal{F}_3 \mid R_3]$ sur R. En effet, dans le cas contraire, en appliquant le lemme 8 avec
  - $S_1 = R_1$,
  - $S_2 = R_3 \cup \pi$,
  - $S_3 = R_2$,
  - $\mathcal{F}_1 = \mathcal{F}_1 \cup \mathcal{F}_3$ et $\mathcal{F}_2 = \mathcal{F}_2$,

  Nous pourrions construire à partir d'une chaîne infinie de $[\mathcal{F}_3 \mid R_3]$ sur R une chaîne infinie de $[\mathcal{F}_3 \mid R_3]$ sur $R_1 \cup R_3 \cup \pi$, or ces chaînes sont toutes finies par hypothèse.
- Il n'existe pas de chaîne infinie de $[\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2]$ sur R. En effet, dans le cas contraire, en appliquant le lemme 8 avec
  - $S_1 = \emptyset$,
  - $S_2 = R_1 \cup R_2$ et
  - $S_3 = R_3 \cup \pi$,

  Nous pourrions construire à partir d'une chaîne infinie de $[\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2]$ sur R une chaîne infinie de $[\mathcal{F}_1 \cup \mathcal{F}_2 \mid R_1 \cup R_2]$ sur $R_1 \cup R_2 \cup \pi$, or ces chaînes ne peuvent être que finies puisque, par hypothèse, $R_1 \cup R_2$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant.

$R_1 \cup R_2 \cup R_3$ est donc $\mathcal{C}_\mathcal{E}$ fortement normalisant. □

Remarque 18. — Parmi les hypothèses du théorème 19, **aucune ne lie R_2 et R_3**. C'est là un point crucial : ne pas avoir de conditions sur R_2 et R_3 garantit en effet une conduite de preuve réellement incrémentale.

Remarquons en outre que l'hypothèse de terminaison $\mathcal{C}_\mathcal{E}$ de $R_1 \cup R_2$ ne peut être allégée : nous pourrions alors retrouver le contre-exemple de Toyama en choisissant

- $R_1 = \emptyset$,
- $R_2 = \{f(0, 1, x) \rightarrow f(x, x, x)\}$ et
- $R_3 = \pi$.

Corollaires

La proposition 12 permet d'obtenir en corollaire du théorème 19 un précédent résultat de Kurihara & Ohuchi [52] : la terminaison $\mathcal{C}_\mathcal{E}$ est une propriété modulaire pour les unions de systèmes composables.

Nous disposons enfin d'un moyen simple d'utiliser le théorème 19 à l'aide des ordres π-extensibles.

Corollaire 19–1.

Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$, soit $[\mathcal{F}_3 \mid R_3]$ un module étendant R_1 indépendamment de R_2.

1. Si $R_1 \cup R_2$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant ;
2. S'il existe un ordre faiblement π-extensible $(\succeq, \succ)$ tel que :
   - $R_1 \cup R_3 \subseteq \succeq$,
   - $\mathrm{DP}([\mathcal{F}_3 \mid R_3]) \subseteq \succ$,

Alors $R_1 \cup R_2 \cup R_3$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant.

IV.3 Exemple complet

Nous proposons maintenant un exemple complet de preuve incrémentale en utilisant nos résultats. Afin de ne pas surcharger les interprétations, nous utilisons exceptionnellement ici des paires de dépendance *non marquées*.

Considérons un système R sur une signature $\mathcal{F}_{\mathbb{N}} = \{\# : \textit{constante}\,;0,1 : \textit{suffixes unaires}\}$ décrivant les entiers codés en binaire.

$$\begin{array}{ll} \mathcal{F}_{\mathbb{N}} & \{\# : \textit{constante}\,; 1, 0 : \textit{unaires}\} \\ R_{\mathbb{N}} & \{\#0 \to \#. \end{array}$$

Nous pouvons vouloir effectuer des opérations arithmétiques sur ces entiers. Définissons par exemple l'addition grâce au module $[\mathcal{F}_+ \mid R_+]$:

$$\begin{array}{ll} \mathcal{F}_+ & \{+ : \textit{infixe binaire}\} \\ R_+ & \left\{ \begin{array}{lll} x + \# & \to & x \\ \# + x & \to & x \\ x0 + y0 & \to & (x+y)0 \\ x0 + y1 & \to & (x+y)1 \\ x1 + y0 & \to & (x+y)1 \\ x1 + y1 & \to & ((x+y) + \#1)0. \end{array} \right. \end{array}$$

La terminaison de $R_{\mathbb{N}} \cup R_+$ est prouvée à l'aide de paires de dépendance et d'une interprétation polynomiale.

$$\mathrm{DP}([\mathcal{F}_+ \mid R_+]) : \left\{ \begin{array}{l} \langle x0 + y0, x + y\rangle\ ; \\ \langle x0 + y1, x + y\rangle\ ; \\ \langle x1 + y0, x + y\rangle\ ; \\ \langle x1 + y1, x + y\rangle\ ; \\ \langle x1 + y1, (x + y) + \#1)\rangle \end{array} \right\} \qquad \begin{array}{l} [\![\#]\!] = 0, \\ [\![0]\!](x) = x + 1, \\ [\![1]\!](x) = x + 1, \\ [\![+]\!](x, y) = x + y. \end{array}$$

Pour l'ordre induit par cette interprétation, les paires de $\mathrm{DP}([\mathcal{F}_+ \mid R_+])$ décroissent strictement et les règles de $R_{\mathbb{N}} \cup R_+ \cup \pi$ décroissent largement. Le corollaire 18–2, nous permet alors de conclure sur la terminaison $\mathcal{C}_{\mathcal{E}}$ de $R_{\mathbb{N}} \cup R_+$.

Nous pouvons aussi désirer soustraire.

$$\begin{array}{ll} \mathcal{F}_- & \{- : \textit{infixe binaire}\} \\ R_- & \left\{ \begin{array}{lll} x - \# & \to & x \\ \# - x & \to & \# \\ x0 - y0 & \to & (x-y)0 \\ x1 - y1 & \to & (x-y)0 \\ x1 - y0 & \to & (x-y)1 \\ x0 - y1 & \to & ((x-y) - \#1)1. \end{array} \right. \end{array}$$

Les paires de dépendance relatives ainsi qu'une interprétation polynomiale permettent ici aussi de montrer que $R_{\mathbb{N}} \cup R_-$ termine $\mathcal{C}_{\mathcal{E}}$. En effet, pour

$$\begin{array}{l} [\![\#]\!] = 0, \\ [\![0]\!](x) = x + 1, \\ [\![1]\!](x) = x + 1, \\ [\![-]\!](x, y) = x, \end{array}$$

Les paires de dépendance de $[\mathcal{F}_- \mid R_-]$ décroissent strictement tandis que les règles de $R_{\mathbb{N}} \cup R_-$ décroissent largement. Nous pouvons déduire du corollaire 19–1 que $R_{\mathbb{N}} \cup R_- \cup R_+$ est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

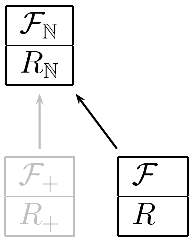


FIG. IV.2. Indépendance de $[\mathcal{F}_+ \mid R_+]$ et $[\mathcal{F}_- \mid R_-]$.
La preuve de terminaison de l'union utilise la terminaison $\mathcal{C}_{\mathcal{E}}$ de $[\mathcal{F}_+ \mid R_+]$ sans que ce module intervienne dans les contraintes d'ordre.

Pour maintenant comparer ces entiers entre eux, il nous faut des opérateurs booléens. Ajoutons donc un module $[\mathcal{F}_{\mathsf{Bool}} \mid R_{\mathsf{Bool}}]$.

$$
\begin{array}{ll}
\mathcal{F}_{\mathsf{Bool}} & \{\mathit{true}, \mathit{false} : \mathit{constantes}\,; \neg : \mathit{unaire}\,; \wedge : \mathit{infixe\ binaire}\,; \mathit{if} : \mathit{ternaire}\} \\
R_{\mathsf{Bool}} & \left\{ \begin{array}{lcl}
\neg(\mathit{true}) & \to & \mathit{false} \\
\neg(\mathit{false}) & \to & \mathit{true} \\
x \wedge \mathit{true} & \to & x \\
x \wedge \mathit{false} & \to & \mathit{false} \\
\mathit{if}(\mathit{true}, x, y) & \to & x \\
\mathit{if}(\mathit{false}, x, y) & \to & y.
\end{array} \right.
\end{array}
$$

Ce système n'a pas de paires de dépendance et donc termine $\mathcal{C}_{\mathcal{E}}$.

Nous pouvons alors définir une comparaison dans le module $[\mathcal{F}_{ge} \mid R_{ge}]$ qui étend $R_{\mathbb{N}}$ et R_{Bool}.

$$
\begin{array}{ll}
\mathcal{F}_{ge} & \{ge : \mathit{binaire}\} \\
R_{ge} & \left\{ \begin{array}{lcl}
ge(x0, y0) & \to & ge(x, y) \\
ge(x0, y1) & \to & \neg ge(y, x) \\
ge(x1, y0) & \to & ge(x, y) \\
ge(x1, y1) & \to & ge(x, y) \\
ge(x, \#) & \to & \mathit{true} \\
ge(\#, x0) & \to & ge(\#, x) \\
ge(\#, x1) & \to & \mathit{false}.
\end{array} \right.
\end{array}
$$

La terminaison de $R_{\mathbb{N}} \cup R_{\mathsf{Bool}} \cup R_{ge}$ est directement montrée en utilisant RPO avec pour précédence : $\{ge > \neg > (\mathit{true}, \mathit{false})\}$. Puisque RPO est un ordre de simplification, il est π-extensible. L'union est donc $\mathcal{C}_{\mathcal{E}}$ fortement normalisante par le théorème 17. Nous pouvons alors appliquer le théorème 19 et ainsi obtenir que $R_{\mathbb{N}} \cup R_{\mathsf{Bool}} \cup R_{ge} \cup R_+ \cup R_-$ est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

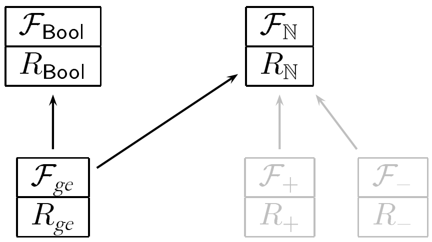


FIG. IV.3. Indépendance des opérations arithmétiques et des comparaisons.

Ajoutons une nouvelle fonction sur les entiers : le logarithme de base 2, arrondi par défaut. Il est techniquement plus facile de définir tout d'abord un *Log'* tel que $Log'(x) = Log(x) + 1$ avec pour convention $Log'(0) = 0$.

$$
\begin{array}{ll}
\mathcal{F}_{Log'} & \{Log' : \mathit{unaire}\} \\
R_{Log'} & \left\{ \begin{array}{lcl} Log'(\#) & \to & \# \\ Log'(x1) & \to & Log'(x) + \#1 \\ Log'(x0) & \to & \mathit{if}(ge(x, \#1), Log'(x) + \#1, \#). \end{array} \right.
\end{array}
$$

On utilise de nouveau les paires de dépendance et une interprétation polynomiale.

$$
\mathrm{DP}([\mathcal{F}_{Log'} \mid R_{Log'}]) : \left\{ \begin{array}{l} \langle Log'(x1), Log'(x) \rangle \ ; \\ \langle Log'(x0), Log'(x) \rangle \end{array} \right\}
$$

En considérant l'ordre induit par l'interprétation déjà utilisée pour $R_{\mathbb{N}} \cup R_{+}$ à laquelle on ajoute

$$
\begin{array}{lll}
[\![\mathit{false}]\!] = 0, & [\![\mathit{true}]\!] = 0, & [\![\neg]\!](x) = 0, \\
[\![ge]\!](x) = 0, & [\![\mathit{if}]\!](x, y, z) = y + z, & [\![\wedge]\!](x, y) = x, \\
[\![Log']\!](x) = x, & &
\end{array}
$$

Les paires de dépendance relatives décroissent strictement alors que les règles de $R_{\mathbb{N}} \cup R_{+} \cup R_{\mathsf{Bool}} \cup R_{ge} \cup R_{Log'}$ décroissent largement. Nous pouvons alors prouver, grâce au corollaire 19–1, la terminaison $\mathcal{C}_{\mathcal{E}}$ de $R_{\mathbb{N}} \cup R_{+} \cup R_{\mathsf{Bool}} \cup R_{ge} \cup R_{Log'} \cup R_{-}$.

Il reste à corriger le calcul du logarithme :

$$
\begin{array}{ll}
\mathcal{F}_{Log} & \{Log : \mathit{unaire}\} \\
R_{Log} & \{ \ Log(x) \ \to \ Log'(x) - \#1.
\end{array}
$$

Puisque $[\mathcal{F}_{Log} \mid R_{Log}]$ n'a pas de paire de dépendance, en appliquant le théorème 19 nous obtenons la terminaison $\mathcal{C}_{\mathcal{E}}$ de $R_{\mathbb{N}} \cup R_{+} \cup R_{-} \cup R_{\mathsf{Bool}} \cup R_{ge} \cup R_{Log'} \cup R_{Log}$.

Si nous souhaitons maintenant travailler sur des arbres binaires d'entiers, il nous suffit de définir un module $[\mathcal{F}_{\mathsf{Tree}} \mid R_{\mathsf{Tree}}]$ adéquat étendant $R_{\mathbb{N}}$:

$$
\begin{array}{ll}
\mathcal{F}_{\mathsf{Tree}} & \{\mathcal{L}, \mathit{Val} : \mathit{unaires} \ ; \ \mathcal{N} : \mathit{ternaire}\} \\
R_{\mathsf{Tree}} & \left\{ \begin{array}{lcl} \mathit{Val}(\mathcal{L}(x)) & \to & x \\ \mathit{Val}(\mathcal{N}(x, l, r) & \to & x. \end{array} \right.
\end{array}
$$

Ce système n'a pas de paires de dépendance.

Tester si un arbre est un arbre binaire de recherche (*BS* pour *Binary Search*) peut être réalisé à l'aide d'un module $[\mathcal{F}_{BS} \mid R_{BS}]$ étendant R_{ge} et R_{Tree} tel que

$$
\begin{array}{ll}
\mathcal{F}_{BS} & \{BS, Min, Max : \mathit{unaires}\} \\
R_{BS} & \left\{ \begin{array}{lcl}
Min(\mathcal{L}(x)) & \to & x \\
Min(\mathcal{N}(x,l,r)) & \to & Min(l) \\
Max(\mathcal{L}(x)) & \to & x \\
Max(\mathcal{N}(x,l,r)) & \to & Max(r) \\
BS(\mathcal{L}(x)) & \to & \mathit{true} \\
BS(\mathcal{N}(x,l,r)) & \to & (ge(x, Max(l)) \wedge ge(Min(r), x)) \wedge \\
 & & (BS(l) \wedge BS(r)).
\end{array} \right.
\end{array}
$$

Les six paires de dépendance de ce module

$$
\mathrm{DP}([\mathcal{F}_{BS} \mid R_{BS}]) : \left\{ \begin{array}{l}
\langle Min(\mathcal{N}(x,l,r)), Min(l) \rangle \ ; \\
\langle Max(\mathcal{N}(x,l,r)), Max(r) \rangle \ ; \\
\langle BS(\mathcal{N}(x,l,r)), Max(l) \rangle \ ; \\
\langle BS(\mathcal{N}(x,l,r)), Min(r) \rangle \ ; \\
\langle BS(\mathcal{N}(x,l,r)), BS(l) \rangle \ ; \\
\langle BS(\mathcal{N}(x,l,r)), BS(r) \rangle.
\end{array} \right\}
$$

Décroissent strictement tandis que décroissent largement les règles de $R_{\mathbb{N}} \cup R_{\mathsf{Bool}} \cup R_{ge} \cup R_{\mathsf{Tree}} \cup R_{BS}$ pour l'ordre induit par l'interprétation polynomiale :

$$
\begin{array}{lll}
[\![\#]\!] = 0, & [\![0]\!](x) = x+1, & [\![1]\!](x) = x+1, \\
[\![\mathit{false}]\!] = 0, & [\![\mathit{true}]\!] = 0, & [\![\neg]\!](x) = 0, \\
[\![ge]\!](x) = 0, & [\![\mathit{if}]\!](x,y,z) = y+z, & [\![\wedge]\!](x,y) = x, \\
[\![\mathcal{L}]\!](x) = x+1, & [\![\mathcal{N}]\!](x,l,r) = x+l+r+1, & [\![Min]\!](x) = x, \\
[\![Max]\!](x) = x, & [\![BS]\!](x) = x. &
\end{array}
$$

$R_{\mathbb{N}} \cup R_{\mathsf{Bool}} \cup R_{ge} \cup R_{\mathsf{Tree}} \cup R_{BS}$ termine donc $\mathcal{C}_{\mathcal{E}}$ par le corollaire 18–1.

Enfin, pour décider si un arbre est bien équilibré (*WB* pour *Well Balanced*), c'est-à-dire si la différence de taille entre le sous-arbre droit et le sous-arbre gauche est au plus de 1, il nous faut aussi savoir calculer la taille d'un arbre.

$$
\begin{array}{ll}
\mathcal{F}_{WB} & \{WB, Size : \mathit{unaires}\} \\
R_{WB} & \left\{ \begin{array}{lcl}
Size(\mathcal{L}(x)) & \to & \#1 \\
Size(\mathcal{N}(x,l,r)) & \to & (Size(l) + Size(r)) + \#1 \\
WB(\mathcal{L}(x)) & \to & \mathit{true} \\
WB(\mathcal{N}(x,l,r)) & \to & \begin{array}{ll}
\mathit{if} & (ge(Size(l), Size(r)), \\
 & ge(\#1, Size(l) - Size(r)), \\
 & ge(\#1, Size(r) - Size(l))) \\
\wedge & (WB(l) \wedge WB(r)).
\end{array}
\end{array} \right.
\end{array}
$$

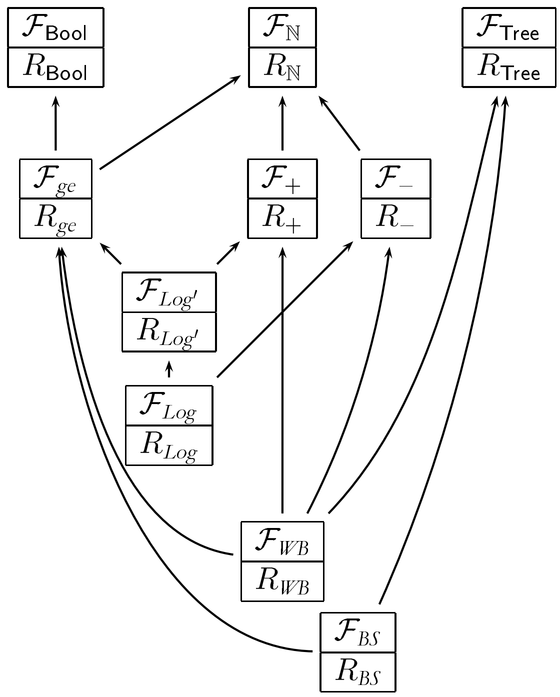


FIG. IV.4. Hiérarchie des modules.

L'ensemble de paires de dépendance

$$\mathrm{DP}([\mathcal{F}_{BS} \mid R_{BS}]) : \left\{ \begin{array}{l} \langle \mathit{Size}(\mathcal{N}(x,l,r)), \mathit{Size}(l)\rangle \ ; \\ \langle \mathit{Size}(\mathcal{N}(x,l,r)), \mathit{Size}(r)\rangle \ ; \\ \langle \mathit{WB}(\mathcal{N}(x,l,r)), \mathit{Size}(l)\rangle \ ; \\ \langle \mathit{WB}(\mathcal{N}(x,l,r)), \mathit{Size}(r)\rangle \ ; \\ \langle \mathit{WB}(\mathcal{N}(x,l,r)), \mathit{WB}(l)\rangle \ ; \\ \langle \mathit{WB}(\mathcal{N}(x,l,r)), \mathit{WB}(r)\rangle \end{array} \right\}$$

Et l'interprétation polynomiale

$$\begin{array}{lll} [\![\#]\!] = 0, & [\![0]\!](x) = x+1, & [\![1]\!](x) = x+1 \\ [\![+]\!](x,y) = x+y, & [\![-]\!](x,y) = x, & [\![\mathit{false}]\!] = 0, \\ [\![\mathit{true}]\!] = 0, & [\![\neg]\!](x) = 0, & [\![\mathit{ge}]\!](x) = 0, \\ [\![\mathit{if}]\!](x,y,z) = y+z, & [\![\wedge]\!](x,y) = x, & [\![\mathcal{L}]\!](x) = x+1, \\ [\![\mathcal{N}]\!](x,l,r) = x+l+r+1, & [\![\mathit{Size}]\!](x) = x, & [\![\mathit{WB}]\!](x) = x, \end{array}$$

Permettent alors de prouver (par application du corollaire 19–1) la terminaison $\mathcal{C}_{\mathcal{E}}$ de l'union de toutes les règles.

IV.4 Modules et graphes

On peut sensiblement affiner les précédents critères de terminaison en utilisant une notion de graphes de dépendance. En effet, c'est l'existence d'une chaîne de dépendance infinie qui forme la clef de voûte des corollaires 18–1, 18–2 et 19–1.

Si l'ajout des règles de π à un système ne change rien à l'ensemble de ses paires de dépendance, il modifie en revanche son graphe en ajoutant éventuellement des arêtes. Pour le système de Toyama (exemple 19) par exemple, le graphe de $R : \{f(0,1,x) \to f(x,x,x)\}$ ne comporte pas d'arête, ce qui permet de conclure immédiatement sur la terminaison. En ajoutant π, la substitution $\sigma : \{x \mapsto G(0,1)\}$ autorise la réduction :

$$\widehat{f}(x,x,x)\sigma \xrightarrow[R\cup\pi]{\neq\Lambda}{}^{\star} \widehat{f}(0,1,x)\sigma.$$

Le graphe a désormais un cycle.

Définition 64. — Le *graphe de dépendance d'un module* $[\mathcal{F} \mid R]$ *sur un système* R' est un graphe dont les nœuds sont les paires de dépendance de $[\mathcal{F} \mid R]$ et dont un nœud $\langle s_1, t_1\rangle$ est relié par un arc à un nœud $\langle s_2, t_2\rangle$ si et seulement s'il existe une substitution σ telle que $t_1\sigma \xrightarrow[R']{}{}^{\star} s_2\sigma$.

Les graphes de dépendance fournissent un moyen efficace pour s'assurer que toutes les chaînes sont finies.

Théorème 20.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_\mathcal{E}$ fortement normalisant,
2. S'il n'existe pas de chemin infini dans le graphe de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2 \cup \pi$,

Alors $R_1 \cup R_2$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant.

Les corollaires des théorèmes 18 et 19 ainsi améliorés deviennent :

Corollaire 18–3.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_\mathcal{E}$ fortement normalisant ;
2. S'il existe un ordre de réduction faible $(\succeq, \succ)$ tel que :
   - $R_1 \cup R_2 \subseteq \succeq$,
   - DP($[\mathcal{F}_2 \mid R_2]$) $\subseteq \succeq$,
   - Dans toute partie strictement fortement connexe du graphe de dépendance de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2$ au moins une paire de dépendance décroît pour $\succ$,

Alors $R_1 \cup R_2$ est fortement normalisant.

Corollaire 18–4.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_\mathcal{E}$ fortement normalisant ;
2. S'il existe un ordre de réduction faiblement π-extensible $(\succeq, \succ)$ tel que :
   - $R_1 \cup R_2 \subseteq \succeq$,
   - DP($[\mathcal{F}_2 \mid R_2]$) $\subseteq \succeq$,
   - Dans toute partie strictement fortement connexe du graphe de dépendance de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2 \cup \pi$ au moins une paire de dépendance décroît pour $\succ$,

Alors $R_1 \cup R_2$ est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

Corollaire 19–2.

Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$, soit $[\mathcal{F}_3 \mid R_3]$ un module étendant R_1 indépendamment de R_2.

1. Si $R_1 \cup R_2$ est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant ;
2. S'il existe un ordre faiblement π-extensible $(\succeq, \succ)$ tel que :
   - $R_1 \cup R_3 \subseteq \succeq$,
   - $\mathrm{DP}([\mathcal{F}_3 \mid R_3]) \subseteq \succeq$,
   - Dans toute partie strictement fortement connexe du graphe de dépendance de $[\mathcal{F}_3 \mid R_3]$ sur $R_1 \cup R_3 \cup \pi$ au moins une paire de dépendance décroît pour $\succ$,

Alors $R_1 \cup R_2 \cup R_3$ est $\mathcal{C}_{\mathcal{E}}$ fortement normalisant.

Deuxième partie

Terminaison des systèmes associatifs et commutatifs

Chapitre I

Introduction

Pour définir un calcul, ou encore dans le cadre de la démonstration automatique, il est très courant d'avoir à manipuler des symboles associatifs et commutatifs (AC). Exprimer directement la commutativité et l'associativité comme des règles d'un système sans perdre la propriété de terminaison est toutefois impossible. On travaille donc avec une relation de récriture sur les classes d'équivalence modulo AC.

L'introduction de ce nouveau type de relation complique singulièrement les preuves de terminaison. Les relations d'ordre utilisées pour montrer la terminaison du système modulo AC doivent en effet être *compatibles* c'est-à-dire respecter dans les comparaisons les appartenances aux classes d'équivalence. Cette propriété se traduit par de fortes contraintes sur les ordres employés. L'importance de la récriture AC a donc suscité des travaux nombreux concernant, en particulier, la définition d'ordres compatibles sémantiques [8] mais aussi et surtout syntaxiques [7,18,78]. Ces ordres sont cependant des ordres de simplification.

On mesure alors l'ampleur du problème pour l'automatisation des preuves : les ordres disponibles sont des ordres de simplification et leurs conditions d'applications sont très contraignantes (voir par exemple la condition APO [déf. 74] de Bachmair et Plaisted). De nombreuses tentatives de preuve à l'aide de ceux-ci sont donc vouées à l'échec, soit parce que le système concerné ne termine pas simplement, soit parce que la signature ou le système ne vérifient pas les contraintes de définition requises.

Une utilisation d'un critère à l'aide de paires de dépendance permettrait de réduire ces deux obstacles à une automatisation efficace des preuves de terminaison de systèmes AC, en élargissant la recherche d'ordre au-delà de la classe des ordres de simplification, d'une part, en affaiblissant les contraintes de définition d'autre part.

Nous allons tenter de définir une extension des critères par paires de dépendance à la terminaison de la récriture modulo AC tout d'abord en nous concentrant sur la récriture dite *étendue* sur des représentants canoniques des classes d'équivalence : les termes plats. Nous passerons ensuite (paragraphes II.2.1 et II.2.2) en revue les différentes contraintes sur les ordres (sémantiques puis syntaxiques) induites par la condition de compatibilité et certaines des solutions proposées jusqu'à présent.

Nous présenterons alors au cours du chapitre III de nouvelles méthodes étendant la technique des paires de dépendance. Les critères développés seront dans un premier temps définis sur des paires sans marques. Les propriétés d'associativité et de commutativité des symboles compliquent en effet singulièrement la manipulation des symboles distingués. Afin d'affaiblir ces critères grâce à l'utilisation de marques nous aurons besoin dans la section III.2 d'une abstraction de la définition d'un ensemble de paires de dépendance en introduisant des *conditions de dépendance* (définition 85). Enfin, puisque ces conditions définissent les ensembles de paires adéquats de façon intentionnelle et afin d'obtenir une automatisation aussi complète que possible nous donnerons, paragraphe III.2.4, deux moyens de contruire, à partir d'un système, des ensembles de paires de dépendance convenables.

Il existe une approche, par Giesl & Kapur, de paires de dépendance modulo une théorie équationnelle arbitraire [30]. L'emploi d'une théorie arbitraire peut introduire de nouvelles relations entre les modules de récriture (ce qui n'est pas le cas de AC) et bouleverser ainsi la notion même de hiérarchie, nous nous limitons donc au cas associatif et commutatif.

Nous présenterons enfin au chapitre IV une méthode de preuve incrémentale de terminaison modulo AC regroupant les avantages des modules de récriture et des conditions de dépendance.

Chapitre II

Récriture avec symboles associatifs et commutatifs

Une signature AC est une signature $\mathcal{F}$ divisée en trois sous-ensembles $\mathcal{F}_{\mathrm{AC}}$, $\mathcal{F}_C$ et $\mathcal{F}_F$ qui contiennent respectivement les symboles associatifs-commutatifs, les symboles commutatifs et les symboles qui ne sont ni commutatifs ni associatifs qu'on désignera dans la suite comme symboles *libres*[1].

Exemple 28.
Une signature AC *décrivant une structure de calcul sur des entiers de Peàno peut être donnée par :*

$$\begin{array}{lcl} \mathcal{F}_F & : & \{0, \mathit{true}, \mathit{false} : \textit{constantes}\ ;\ s : \textit{unaire}\ ;\ \geq : \textit{binaire}\} \\ \mathcal{F}_{\mathrm{AC}} & : & \{+, \times : \textit{binaires}\} \\ \mathcal{F}_C & : & \{= : \textit{binaire}\}. \end{array}$$

Définition 65. — Si $\mathcal{F}$ est une signature AC, on considère la congruence sur $T(\mathcal{F}, X)$ générée par les axiomes de commutativité : pour tout $f \in \mathcal{F}_C \cup \mathcal{F}_{\mathrm{AC}}$

$$f(x, y) = f(y, x)$$

Et les axiomes d'associativité : pour tout $f \in \mathcal{F}_{\mathrm{AC}}$

$$f(f(x, y), z) = f(x, f(y, z)).$$

Cette congruence est appelée *égalité modulo* AC et notée $=_{\mathrm{AC}}$.

Les symboles AC présentent la propriété de pouvoir s'écrire avec un nombre quelconque d'arguments. En effet le symbole $+$, par exemple, pour l'addition des entiers est binaire au premier abord. On a néanmoins $a + (b + c) = (a + b) + c = a + b + c$ qu'on peut noter $+(a, b, c)$. L'arité 2 de $+$ est alors remise en question. On définit donc les notions de termes *variadiques* et d'*aplatissement*.

Définition 66. — L'ensemble des termes *variadiques* sur une signature $\mathcal{F}$, noté $T_{\mathrm{Var}}(\mathcal{F}, X)$ est défini de la façon suivante :

– Si $x \in X$ alors $x \in T_{\mathrm{Var}}(\mathcal{F}, X)$;

[1] L'indice F vient de l'anglais « free » traduisant *libre*.

- Si $f \notin \mathcal{F}_{\mathrm{AC}}$ avec $\mathrm{AR}(f) = n$ et si $t_1, \ldots, t_n \in T_{\mathrm{Var}}(\mathcal{F}, X)$ alors $f(t_1, \ldots, t_n) \in T_{\mathrm{Var}}(\mathcal{F}, X)$;
- Si enfin $f \in \mathcal{F}_{\mathrm{AC}}$ et si $t_1, \ldots, t_n \in T_{\mathrm{Var}}(\mathcal{F}, X)$ (avec $n \geq 2$) alors $f(t_1, \ldots, t_n) \in T_{\mathrm{Var}}(\mathcal{F}, X)$.

La congruence $=_{\mathrm{AC}}$ est alors adaptée à $T_{\mathrm{Var}}(\mathcal{F}, X)$.

Définition 67. — Soit $\mathcal{F}$ une signature AC. On considère la congruence sur $T_{\mathrm{Var}}(\mathcal{F}, X)$ générée par les axiomes de commutativité : pour tout $f \in \mathcal{F}_C \cup \mathcal{F}_{\mathrm{AC}}$

$$f(x, y) = f(y, x),$$

Les axiomes de permutation : pour tout $f \in \mathcal{F}_{\mathrm{AC}}$ avec $n \geq 2$ et où p est une permutation de $[1, \ldots, n]$

$$f(x_1, \ldots, x_n) = f(x_{p(1)}, \ldots, x_{p(n)})$$

Et les axiomes d'associativité : pour tout $f \in \mathcal{F}_{\mathrm{AC}}$ avec $n \geq 2$, $1 \geq i \geq n$ et $k \geq 2$

$$f(x_1, \ldots, x_{i-1}, f(y_1, \ldots, y_k), x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, y_1, \ldots, y_k, x_{i+1}, \ldots, x_n).$$

Cette congruence est appelée *égalité modulo* AC, on la note $=_{\mathrm{AC}}$.

Remarque 19. — En restreignant l'application de cette définition aux termes à arité fixe (c'est-à-dire non variadiques) on retrouve bien la congruence de la définition 65.

Définition 68. — Soit $\mathcal{F}$ une signature. *L'aplatissement* est une application, notée *flat*, de $T_{\mathrm{Var}}(\mathcal{F}, X)$ dans $T_{\mathrm{Var}}(\mathcal{F}, X)$ telle que :

- Pour $f \in (\mathcal{F} \setminus \mathcal{F}_{\mathrm{AC}})$ d'arité n, $\mathit{flat}(f(t_1, \ldots, t_n)) = f(\mathit{flat}(t_1), \ldots, \mathit{flat}(t_n))$;
- Pour $g \in \mathcal{F}_{\mathrm{AC}}$, $\mathit{flat}(g(s_1, \ldots, s_n)) = g(t_{1,1}, \ldots, t_{1,k_1}, \ldots, t_{n,1}, \ldots, t_{n,k_n})$ où :
  - Si $\Lambda(s_i) \neq g$ alors $k_i = 1,\ t_{i,1} = s_i$,
  - Si $\Lambda(s_i) = g$ alors $g(t_{i,1}, \ldots, t_{i,k_i}) = \mathit{flat}(s_i)$.

Un terme est dit *aplati* si $s = \mathit{flat}(s)$; on notera par la suite $\overline{s}$ la forme aplatie d'un terme s.

L'associativité-commutativité correspond en fait à la *congruence de permutation* sur les termes aplatis. On dispose donc ainsi d'un moyen simple d'exhiber un représentant de classe.

II.1 Récriture AC

Puisqu'il est impossible d'exprimer directement la commutativité et l'associativité par des règles de récriture sans perdre la terminaison, la récriture modulo AC est définie comme une relation de récriture sur les classes d'AC-équivalence.

On considère qu'un système de récriture associatif-commutatif est composé de deux parties : d'une part l'ensemble R des règles de récriture et, d'autre part, l'ensemble AC des équations d'associativité et de commutativité.

Définition 69. — Soit R un système de règles sur $\mathcal{F}$. On dit qu'un terme $s \in T(\mathcal{F}, X)$ se *récrit modulo* AC en t s'il existe un contexte C, une position p, une règle $l \to r \in R$ et une substitution σ tels que :

$$s =_{\mathrm{AC}} C[l\sigma]_p \quad \text{et} \quad C[r\sigma]_p =_{\mathrm{AC}} t.$$

Ce pas de récriture est noté :

$$s \xrightarrow[R/\mathrm{AC}]{} t.$$

Dans les cas où aucune confusion n'est possible, nous omettrons éventuellement le système utilisé.

La taille des classes d'équivalence peut être très importante, ce qui rend leur calcul coûteux sinon irréalisable. On contourne ce problème grâce à une relation plus faible : la récriture AC étendue. Cette relation, due à Peterson et Stickel [72] interdit les étapes équationnelles « au-dessus » (c'est-à-dire à une position préfixe) des pas de récriture.

Définition 70. — On appelle *relation de récriture* AC *étendue* pour un système de règles R la relation $\rightarrow_{\mathrm{AC}/R}$ telle que pour deux termes s et t on ait :

$$s \xrightarrow[\mathrm{AC}/R]{} t \ \text{ si et seulement si } \ s|_p =_{\mathrm{AC}} l\sigma \ \text{ et } \ t = s[r\sigma]_p$$

Pour une position p, une règle $l \rightarrow r \in R$ et une substitution σ. La mention du système ou du type de récriture étant éventuellement omise en l'absence d'ambiguïté.

Un terme n'est maintenant susceptible d'être réduit par une système R que s'il a un sous-terme AC équivalent à une instance de membre gauche d'une règle de R.

Telle quelle, cette nouvelle relation n'est pas capable de simuler une réduction par la récriture modulo AC. En effet si la récriture étendue, incluse dans la récriture par classes, conserve certaines de ses propriétés (les notions de formes normales ou de confluence sont définies de manière analogue) elle n'hérite pas de la *cohérence* [46] en général.

Définition 71. — Soient S un ensemble d'équations, $\rightarrow_{R^S}$ une relation arbitraire comprise entre $\rightarrow_R$ et $\rightarrow_{R/S}$. On dit que $\rightarrow_{R^S}$ est *cohérente modulo* $\rightarrow_S$ si :

$$s \xleftrightarrow[S]{\star} t \xrightarrow[R]{} u \ \text{ implique l'existence de } \ v, v' \ \text{ tels que } s \xrightarrow[R^S]{\star} v \xleftrightarrow[S]{\star} v' \xleftarrow[R^S]{\star} u.$$

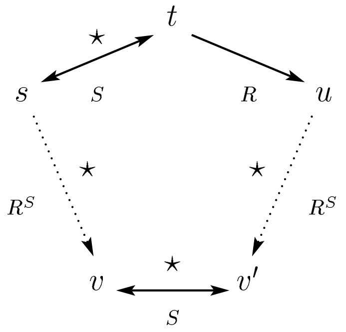


FIG. II.1. Diagramme de cohérence modulo une relation.

Exemple 29.

Si on considère le système composé de la seule règle $a + b \rightarrow c$ *et de l'ensemble d'équations décrivant* AC *(*$+$ *étant* *un symbole* AC*) alors* $(a + c) + b \rightarrow_{/\mathrm{AC}} c + c$*, mais* $(a + c) + b$ *ne se récrit pas par* $\rightarrow_{\backslash \mathrm{AC}}$ *et on ne peut pas refermer le diagramme.*

AC possède cependant quelques propriétés qui permettent une gestion de ces difficultés.

Nous avons vu dans l'exemple précédent que la récriture étendue ne permettait pas toujours de réduire suffisamment un terme. Pour remédier à ce problème et rétablir la cohérence, on définit des règles complémentaires qui permettent d'appliquer artificiellement des étapes équationnelles un niveau au-dessus de la position de réduction. Dans le cas de cet exemple (qu'on généralise aisément) nous devrions ajouter la règle dite *étendue* suivante :

$$((a + x) + b) \rightarrow (c + x)$$

Où x est une variable fraîche.

Il reste à déterminer un représentant de chaque classe d'équivalence ; nous choisirons de travailler sur des termes en forme aplatie.

En combinant aplatissement et récriture étendue on définit une nouvelle notion de récriture.

Définition 72. — Soit R un système de récriture. Un terme s se récrit en un terme t par *récriture* AC *sur termes aplatis* [57] s'il existe une position p telle que l'une des deux propositions suivantes est vérifiée :

1. $s|_p =_{\mathrm{AC}} \overline{l\sigma}$ et $t = \overline{s[r\sigma]_p}$;
2. $l = f(l_1, \dots, l_n)$ avec $f \in \mathcal{F}_{\mathrm{AC}}$, $s|_p =_{\mathrm{AC}} \overline{f(l_1, \dots, l_n, x)\sigma}$ et $t = \overline{s[f(r, x)\sigma]_p}$, pour une règle $l \rightarrow r \in R$, σ une substitution et x une variable sans occurrence dans l (cas de *filtrage étendu*).

Dans la suite et dans tous les cas où la confusion avec la récriture par classe sera impossible nous noterons un tel pas $s \rightarrow_{R/\mathrm{AC}} t$.

Les problèmes de cohérence disparaissent car les règles étendues sont déjà dans la définition et tout pas $s \rightarrow t$ pour la récriture modulo entraîne l'existence d'une réduction étendue sur termes aplatis entre $\overline{s}$ et $\overline{t}$. En reprenant ainsi l'exemple 29 avec cette nouvelle notion, $((a + c) + b)$ se récrit directement en $(c + c)$.

Il n'est pas toujours nécessaire d'avoir recours au filtrage étendu pour des règles d'une certaine forme. En fait une règle $l \rightarrow r$ a besoin du filtrage étendu si et seulement si les deux conditions suivantes sont vérifiées :

1. Le symbole de tête f de l est associatif-commutatif ;
2. Il n'y a pas de variable x dans l telle que :
   (a) Il n'y ait qu'une seule occurrence de x dans l et uniquement à profondeur 1 ;
   (b) Soit $r = x$, soit $r = f(x, r')$ sans que x n'apparaisse dans r'.

Proposition 13.

Soit un pas de récriture $s \xrightarrow[l \rightarrow r/\mathrm{AC}]{p,\sigma} t$ utilisant le filtrage étendu et tel que la règle $l \rightarrow r$ ne remplisse pas les conditions.
Alors il existe une substitution σ' telle que $s \xrightarrow[l \rightarrow r/\mathrm{AC}]{p,\sigma'} t$ ne nécessite pas de filtrage étendu.

Preuve. Si le filtrage étendu est utilisé pour le pas de réduction AC sur termes aplatis

$$s \xrightarrow[l \rightarrow r/\mathrm{AC}]{p,\sigma} t,$$

C'est que le symbole $\Lambda(l)$, disons $+$, est AC et que de plus $s|_p = (l+z)\sigma$ et $t = s[(r+z)\sigma]_p$ où z est une variable fraîche.

Pourtant, puisque $l \to r$ ne remplit pas les conditions précitées, nous savons qu'il existe une variable x telle que $l = l' + x$ et soit $r = x$ soit $r = r' + x$ où x n'apparaît ni dans l' ni dans r'. Considérons alors la substitution σ' définie par $x\sigma' = x\sigma + z\sigma$ et $y\sigma' = y\sigma$ pour une variable y distincte de x et z. Nous pouvons en déduire :

$$s|_p = (l+z)\sigma = (l'+x+z)\sigma = (l'+x)\sigma' = l\sigma'.$$

Et soit

$$t = s[(r+z)\sigma]_p = s[(x+z)\sigma]_p = s[x\sigma']_p = s[r\sigma']_p \ \text{ si } r = x,$$

Soit

$$t = s[(r+z)\sigma]_p = s[(r'+x+z)\sigma]_p = s[(r'+x)\sigma']_p = s[r\sigma']_p \ \text{ si } r = r'+x.$$

Cette réduction n'a donc pas besoin de filtrage étendu. □

Exemple 30.

Les règles $x + 0 \to x$ ou encore $x + (-x) + y \to y$ n'ont pas besoin de filtrage étendu et pourtant le symbole de tête $+$ de leurs membres gauches est associatif-commutatif.

En revanche, toutes les règles suivantes doivent être étendues :

- $x + (-x) \to 0$ *car x apparaît deux fois dans le membre gauche ,*
- $R = \{x \times 0 \to 0\}$ *puisque x est sans occurrence dans le membre droit : par exemple, $a \times b \times 0 \to_R a \times 0$ utilise le filtrage étendu ;*
- $R = \{x + s(y) \to s(x+y)\}$, *dans ce cas x n'apparaît pas à profondeur 1 dans le membre droit : le pas $0 + s(0) + 0 \to_R s(0+0) + 0$ a besoin de filtrage étendu.*

Dans toute la suite et sauf mention du contraire nous n'utiliserons que la récriture AC sur termes aplatis. Cette relation (pour un système R) sera donc simplement désignée par $\to_R$.

II.2 Terminaison de la récriture AC

À l'instar du calcul, il suffit de vérifier la terminaison de la récriture étendue sur termes aplatis pour garantir le résultat concernant la récriture par classes.

Théorème 20.

Un système termine modulo AC si et seulement s'il termine pour la récriture AC sur termes aplatis.

PREUVE. Supposons qu'il existe une réduction infinie pour la récriture modulo AC issue d'un terme s et débutant par

$$s \xrightarrow[l\to r/\mathrm{AC}]{p} t.$$

Cela signifie $s =_{\text{AC}} s'$ où $s'|_p = l\sigma$ pour une substitution σ. Il est alors facile de voir que $\overline{s}$ contient une instance (éventuellement étendue) de l et qu'il est donc susceptible d'être réduit par récriture AC sur termes aplatis en un terme plat égal modulo AC à t. On peut donc, en répétant cette construction, bâtir une réduction infinie pour la récriture AC sur termes aplatis.

La réciproque est triviale. □

Il reste toutefois à prouver la terminaison des systèmes pour la récriture sur les termes aplatis.

Un ordre susceptible de prouver la terminaison d'une relation modulo AC doit respecter l'équivalence des termes pour la théorie équationnelle. Cette prise en compte du statut associatif-commutatif de certains symboles restreint l'ensemble des ordres utilisables auparavant à celui des ordres *compatibles* AC, c'est-à-dire vérifiant :

$$\text{Si}\quad \left.\begin{array}{ccc} s & \succ & t \\ s & =_{\text{AC}} & s' \\ t & =_{\text{AC}} & t' \end{array}\right\} \quad \text{alors} \quad s' \succ t' \quad \text{et si} \quad \left.\begin{array}{ccc} s & \succeq & t \\ s & =_{\text{AC}} & s' \\ t & =_{\text{AC}} & t' \end{array}\right\} \quad \text{alors} \quad s' \succeq t'.$$

II.2.1 Compatibilité et interprétations polynomiales

Forcer la compatibilité pour les interprétations polynomiales est réalisé en limitant le choix des polynômes aux polynômes associatifs-commutatifs.

Définition 73. — Le polynôme P est un *polynôme associatif-commutatif* si :

1. $P(P(X,Y),Z) = P(X,P(Y,Z))$;
2. $P(X,Y) = P(Y,Z)$.

On obtient un ordre compatible AC si toutes les interprétations des symboles AC sont des fonctions polynomiales AC.

S'il est difficile de trouver une interprétation convenable pour un système AC, il est en revanche aisé de vérifier qu'un polynôme est associatif-commutatif, comme le montre le lemme suivant :

Lemme 14. (Ben Cherifa et Lescanne)

Le polynôme $P(X,Y)$ est AC si et seulement si c'est un polynôme symétrique du premier degré en chacune des indéterminées : $aXY + b(X+Y) + c$ tel que $b^2 = b + ac$.

La démonstration de ce lemme [8] est aisée : la commutativité impose un polynôme symétrique, l'associativité intervient dans le degré en chacune des indéterminées et dans les relations entre coefficients.

Exemple 31.

Reprenons l'exemple de l'arithmétique de Peàno mais où cette fois $+$ et $\times$ sont, comme on peut l'attendre, associatifs et commutatifs.

$$\begin{array}{lcl} \mathcal{F}_F & : & \{0 : \textit{constante}\,;\ s : \textit{unaire}\} \\ \mathcal{F}_{\text{AC}} & : & \{+, \times : \textit{binaires}\} \end{array}$$

$$R : \left\{\begin{array}{lcl} x + 0 & \to & x \\ x + s(y) & \to & s(x+y) \\ x \times 0 & \to & 0 \\ x \times s(y) & \to & x + (x \times y). \end{array}\right.$$

Considérons alors les interprétations :

$$
\begin{array}{ll}
[\![\times]\!]_1(x,y) = xy + 2(x+y) + 2, & [\![\times]\!]_2(x,y) = \textit{inutile}, \\
[\![+]\!]_1(x,y) = x + y + 1, & [\![+]\!]_2(x,y) = 2xy + 2(x+y) + 1, \\
[\![s]\!]_1(x) = x + 1, & [\![s]\!]_2(x) = x + 1, \\
[\![0]\!]_1 = 1, & [\![0]\!]_2 = 1.
\end{array}
$$

La composition lexicographique de ces deux interprétations polynomiales permet de conclure sur la terminaison du système.

Exemple 32.

La terminaison de l'exemple d'arithmétique sur des entiers en binaire (exemple 4, page 21) avec $+$ *et* $\times$ AC *peut, elle aussi, être prouvée au moyen de compositions d'interprétations polynomiales. Pour cela on considère :*

$$
\begin{array}{ll}
[\![\times]\!]_1(x,y) = xy - (x+y) + 2, & [\![\times]\!]_2(x,y) = xy + 2(x+y) + 2, \\
[\![+]\!]_1(x,y) = x + y - 2, & [\![+]\!]_2(x,y) = x + y, \\
[\![0]\!]_1(x) = x + 1, & [\![0]\!]_2(x) = x + 1, \\
[\![1]\!]_1(x) = x + 2, & [\![1]\!]_2(x) = x, \\
[\![\#]\!]_1 = 2, & [\![\#]\!]_2 = 2.
\end{array}
$$

II.2.2 Compatibilité et ordres sur les chemins

Le RPO n'est pas compatible AC. En fait il n'est plus monotone dès que des pas associatifs-commutatifs peuvent être impliqués.

Exemple 33. (Delor & Puel) [18]

Sur la signature $\{f\ ;\ g\}$*, où* f *est* AC *et avec la précédence* $f > g$ *nous obtenons l'inégalité* $f(x,y) >_{\mathrm{RPO}} g(x,y)$*. Plongeons ces deux termes dans le contexte* $f(\Box, z)$ *:* $f(x,y,z)$ *et* $f(g(x,y),z)$ *sont orientés dans le « mauvais » sens.*

On doit donc définir un nouvel ordre susceptible d'orienter les symboles AC.

Proposé par Bachmair et Plaisted [7] puis amélioré par Bachmair et Dershowitz [6], l'« Associative Path Ordering » ou APO permet, avec certaines conditions sur la précédence et grâce à un système de normalisation, de traiter le cas des symboles AC tout en conservant d'intéressantes propriétés (sous-terme, etc.). Delor et Puel [18] en ont tiré des ordres plus puissants en affaiblissant les contraintes : les ordres de la famille de MAPO décrits plus loin.

Définition 74. — On dit qu'une précédence $>$ vérifie *la condition* APO *sur les chemins* (*Associative Path Condition*) si pour tout symbole $f \in \mathcal{F}_{\mathrm{AC}}$ l'une des deux conditions suivantes est vérifiée :

- f est minimal dans $\mathcal{F}$;
- Il existe g dans $\mathcal{F}_{\mathrm{AC}}$ tel que f est minimal dans $\mathcal{F} \setminus \{g\}$.

On définit alors un système de récriture pour cette précédence sur $\mathcal{F}$, D, avec l'ensemble des règles

$$f(g(x,y),z) \to g(f(x,z), f(y,z))$$

Où f et g sont AC et $f > g$. Ce système est un système de distribution des symboles AC non minimaux sur les autres.

La propriété fondamentale est que D termine et est confluent [7]. Une raison d'être de la condition sur la précédence est qu'elle impose la confluence du système. Celui-ci peut alors être utilisé à des fins de normalisation.

La procédure de normalisation est une transformation des termes vers les termes qui à t associe sa D-forme normale aplatie, soit $\overline{t\downarrow_D}$.

Définition 75. — L'ordre APO est défini par :

$$s >_{\text{APO}} t \text{ si et seulement si } \overline{s\downarrow_D} >_{\text{RPO}} \overline{t\downarrow_D}.$$

Les contraintes sur la précédence permettent à cet ordre d'avoir certaines propriétés, la principale étant :

Théorème 21. (Bachmair & Plaisted) [7]
APO est monotone et vérifie sous-terme.

Ordre modifié sur les chemins

Le principal problème d'APO est d'être très restrictif en ce qui concerne la précédence : on cherche alors l'affaiblissement de la condition.

Deux des tentatives proposées par Delor et Puel [18] sont à base de modifications du système de normalisation en rapport avec deux types de précédence rencontrés en pratique. Ces modifications sont nécessaires car dès qu'on assouplit l'« associative path condition » la confluence, voire la terminaison, du système de normalisation d'APO n'est plus assurée (cf. exemple 34, page 98).

La première proposition, EAPO (« Extended Associative Path Ordering »), utilise un système non-convergent mais profite de l'existence d'une stratégie qui en fait un système de normalisation. La seconde, MAPO (« Modified Associative Path Ordering »), traite un autre cas de précédence avec directement un système de normalisation.

Extended Associative Path Ordering. Dans le cas traité ici on dispose de n symboles l_k tels que $l_k < l_{k+1}$ pour la précédence avec $k < n$. L'extension d'APO va être en fait définie à partir d'une précédence $<$ sur une signature $\mathcal{F}$ telle que pour tout symbole f de $\mathcal{F}_{\text{AC}}$ il existe une chaîne l de symboles AC l_i pour $1 \leq i \leq n_l$ et un index i_0 tels que $f = l_{i_0}$, $l_i < l_{i+1}$ et l_{i+1} est minimal dans $\mathcal{F} \setminus \{l_1, \ldots, l_i\}$ pour tout i, $1 \leq i \leq n_l$.

L'idée consiste à obtenir une extension d'APO de la forme :

$$s <_{\text{EAPO}} t \text{ si et seulement si } \overline{s\downarrow_D} <_{\text{RPO}} \overline{t\downarrow_D}.$$

Il paraît naturel pour la normalisation de choisir à nouveau comme système les règles de distribution des symboles selon la précédence mais on perd alors la confluence.

Exemple 34.
Pour trois symboles $f < g < k$, le système :

$$R : \left\{ \begin{array}{lcl} h(f(x,y),z) & \rightarrow & f(h(x,z),h(y,z)) \\ h(g(x,y),z) & \rightarrow & g(h(x,z),h(y,z)) \\ g(f(x,y),z) & \rightarrow & f(g(x,z),g(y,z)) \end{array} \right.$$

N'est pas confluent.

On peut en outre montrer [18] qu'aucune interprétation polynomiale ne prouve sa terminaison. Il existe néanmoins une stratégie qui en fait un système de normalisation : les deux premières règles terminent, en effectuant d'abord les distributions sur les symboles les plus petits pour la précédence on atteint une forme normale unique. Il reste à normaliser par la dernière règle, confluente et qui termine.

Plus de trois symboles AC interviennent souvent dans la pratique : il convient de définir formellement le système général D qui nous intéresse. Pour une chaîne l de n_l symboles comparables, $2 \leq k \leq n_l$ on a :

– $1 \leq j < k \leq n_l$, la règle de distribution

$$R_{k,j}(l) = l_k(l_j(x, y), z) \to l_j(l_k(x, z), l_k(y, z)),$$

– Pour chaque k

$$D_k(l) = \{R_{k,j}(l) \mid 1 \leq j < k\},$$

– Pour chaque l

$$D(l) = \bigcup_{2 \leq k \leq n_l} D_k(l),$$

– Enfin pour toutes les chaînes l on a le système général

$$D = \bigcup D(l).$$

Chacun des ensembles $D(l)$ ne porte que sur les symboles de l. On peut normaliser séparément et commutativement par les systèmes en rapport avec des chaînes différentes (et non projectifs). Dans la suite on ne considère donc que le cas d'une seule chaîne de longueur n. Les démonstrations des propriétés énoncées n'ont pas été publiées mais sont néanmoins disponibles [17].

– Les D_k terminent mais ne sont pas confluents. La stratégie suivante mène à une forme normale unique : normalisation par $R_{k,1}$ puis par $R_{k,2}$ jusqu'à $R_{k,k-1}$ et on recommence jusqu'à l'irréductibilité pour D_k. On peut donc parler de normalisation par D_k.
– La stratégie suivante fait de D un système de normalisation :

$$\downarrow_D \stackrel{\text{def}}{=} \downarrow_{D_n} \downarrow_{D_{n-1}} \cdots \downarrow_{D_2} .$$

Puisqu'on bénéficie d'un système de normalisation, on peut définir l'« Extended Path Ordering» comme désiré, à savoir :

$$s <_{\text{EAPO}} t \text{ si et seulement si } \overline{s \downarrow_D} <_{\text{RPO}} \overline{t \downarrow_D}.$$

Théorème 22. (Delor et Puel)
EAPO est un ordre de simplification sur les termes clos.

Il suffit de vérifier la stabilité d'EAPO sur un certain ensemble de substitutions pour avoir la stabilité par substitution [18].

Modified Associative Path Ordering. La précédence sur la signature est cette fois du type : $g < u_1 < \ldots < u_n < f$, où g et f sont deux symboles AC comparables et où les u_i sont des symboles unaires. L'idée reste la même : on cherche une relation $<_{\mathrm{MAPO}}$ et un système de normalisation D tels que :

$$s <_{\mathrm{MAPO}} t \text{ si et seulement si } \overline{s \downarrow_D} <_{\mathrm{RPO}} \overline{t \downarrow_D}.$$

La condition définie est un peu plus générale que la condition APO sur les chemins pour assurer la terminaison de D.

Définition 76. — On dit qu'une précédence vérifie *la condition MAPO sur les chemins* si, pour tout symbole f associatif-commutatif, l'une des deux propositions suivantes est vérifiée :

- f est minimal dans $\mathcal{F}$;
- Il existe un symbole AC g et une chaîne de symboles unaires $U = \{u_i \text{ pour } 1 \leq i \leq n\}$ tels que f est minimal dans $\mathcal{F} \setminus (U \bigcup \{g\})$ et $g < u_1 < \ldots < u_n < f$.

On peut légèrement étendre cette condition en ajoutant une constante inférieure au symbole minimal f.

Le système D comporte les règles de distribution des symboles AC non minimaux sur les autres et des règles nécessaires pour obtenir la confluence du système. Pour tous symboles AC f et g et tous symboles unaires u_i, u_j et u_k tels que $g < u_i < f$ et $g < u_j < u_k < f$ on définit D comme :

$$\left\{ \begin{array}{lcl} f(g(x,y),z) & \rightarrow & g(f(x,z),f(y,z)) \\ f(x,g(y,z)) & \rightarrow & g(f(x,y),f(x,z)) \\ f(u_i(x),y) & \rightarrow & u_i(f(x,y)) \\ f(x,u_i(y)) & \rightarrow & u_i(f(x,y)) \\ u_i(g(x,y)) & \rightarrow & g(u_i(x),u_i(y)) \\ u_k(u_j(x)) & \rightarrow & u_j(u_k(x)). \end{array} \right.$$

Sous les hypothèses de la condition MAPO on peut montrer [18] que le système D/AC termine et est confluent. On peut donc le prendre comme système de normalisation.

On peut définir le « Modified Path Ordering» de manière similaire à l'EAPO c'est-à-dire :

$$s <_{\mathrm{MAPO}} t \text{ si et seulement si } \overline{s \downarrow_D} <_{\mathrm{RPO}} \overline{t \downarrow_D}.$$

Théorème 23. (Delor et Puel) [17]

MAPO est un ordre de simplification.

Exemple 35.

La preuve de terminaison du système des anneaux commutatifs :

$$\left\{ \begin{array}{lcl} x + 0 & \rightarrow & x \\ x + (-x) & \rightarrow & 0 \\ -0 & \rightarrow & 0 \\ -(-x) & \rightarrow & x \\ -(x+y) & \rightarrow & (-x)+(-y) \\ x \times 1 & \rightarrow & x \\ x \times (y+z) & \rightarrow & (x \times y) + (x \times z) \\ x \times 0 & \rightarrow & 0 \\ x \times (-y) & \rightarrow & -(x \times y) \end{array} \right.$$

Se fait grâce à MAPO *avec la précédence* $\times > - > + > 0$ *qui vérifie bien la condition* MAPO.

Si MAPO peut être utilisé dans les cas de précédence avec un seul symbole AC et des symboles unaires qui lui sont inférieurs pour la précédence, D n'est plus confluent dès que des symboles unaires sont inférieurs (toujours pour la précédence) à deux symboles AC comparables.

Exemple 36.

Considérons le cas $f > g > u$ *où* f *et* g *sont* AC *et* u *unaire.*

$$\begin{tikzcd}
 & f(g(x,y),h(z)) \arrow[dl] \arrow[dr] & \\
h(f(g(x,y),z)) \arrow[d] & & g(f(x,h(z)),f(y,h(z))) \arrow[d, "\star"] \\
h(g(f(x,z),f(y,z))) & & h(h(g(f(x,z),f(y,z))))
\end{tikzcd}$$

Et on ne peut pas refermer le diagramme car les termes sont irréductibles.

Dans ce cas on définit (et pour ce cas seulement où un seul symbole unaire est inférieur aux symboles AC) un nouvel ensemble de règles de normalisation D' comme :

$$\left\{\begin{array}{lcl}
g(u(x),y) & \rightarrow & u(g(x,y)) \\
f(u(x),y) & \rightarrow & u(f(x,y)) \\
f(g(x,y),z) & \rightarrow & g(f(x,z),f(y,z)) \\
u(u(x)) & \rightarrow & u(x).
\end{array}\right.$$

Ce système est bien de normalisation. De plus si u est minimal on a stabilité de l'ordre par sous-terme et par contexte.

II.2.3 Un RPO compatible AC sans système de normalisation

Pour se passer des systèmes de normalisations qui, en plus de contraindre les précédences, peuvent perturber l'orientation correcte des règles, Albert Rubio définit [78] un ordre qui utilise directement le principe du RPO sans interprétation préliminaire.

Nous donnons ici la version de cet ordre pouvant utiliser des précédences partielles sur les symboles de la signature. L'importance de cette possibilité devient claire dès qu'on veut ajouter des symboles à la signature.

Afin de permettre de tels ajouts il faut redéfinir la notion d'extension multiensemble d'un ordre compatible AC en particulier en la rendant dépendante de certains symboles.

Définition 77. — Soient $>$ un ordre compatible AC sur $T(\mathcal{F}, X)$ et $\succ$ une précédence partielle (sur $\mathcal{F}$). Soit f un symbole de $\mathcal{F}_{\text{AC}}$.

L'*extension multiensemble* de $>$ *par rapport à* f, notée $>^f_{\mathsf{mul}}$, est définie comme la plus petite relation transitive contenant :

$$M \cup \{s\} >^f_{\mathsf{mul}} N \cup \{t_1\,;\ldots;\,t_n\} \text{ si } \left\{\begin{array}{l}
M \text{ et } N \text{ sont équivalents modulo AC,} \\
\text{Pour tout } i,\ 1 \leq i \leq n, \\
\quad \bullet\ s > t_i \text{ et} \\
\quad \bullet\ \text{Si } \Lambda(s) \not\succ f \text{ alors } \Lambda(s) \succeq \Lambda(t_i).
\end{array}\right.$$

L'application sans interprétation d'un ordre à base de RPO implique de savoir traiter les ensembles de sous-termes de s dont les symboles de tête sont éventuellement comparables pour la précédence à $\Lambda(s)$.

Définition 78. — Soient $s = f(s_1, \ldots, s_n)$ un terme avec $f \in \mathcal{F}_{\text{AC}}$ et $\succ$ une précédence sur $\mathcal{F}$. On définit les ensembles :

$$\mathit{BigHead}(s) = \{s_i,\ 1 \leq i \leq n \mid \Lambda(s_i) \succ f\}\ ;$$

$$\mathit{NoSmall}(s) = \{s_i,\ 1 \leq i \leq n \mid f \not\succ \Lambda(s_i)\}.$$

Afin de permettre le plongement dans un terme s de termes à travers des symboles qui ne sont pas plus grands pour la précédence $\succ$ que $\Lambda(s)$ on définit l'ensemble $\mathit{EmbNoBig}(s)$.

Définition 79. — Soit s un terme de la forme $f(s_1, \ldots, s_n)$ avec $f \in \mathcal{F}_{\text{AC}}$, $\succ$ une précédence sur $\mathcal{F}$. On désigne par $\overline{t}^f$ le terme t après aplatissement du symbole f à la racine uniquement.

$$\begin{aligned}&\mathit{EmbNoBig}(s) = \\ &\quad \{f(s_1, \ldots, s_{i-1}, \overline{v_j}^f, s_{i+1}, \ldots, s_n) \mid s_i = h(v_1, \ldots, v_k) \text{ avec } h \not\succ f \text{ et pour } 1 \leq j \leq k\}.\end{aligned}$$

Enfin pour comparer des termes contenant des variables on introduit une interprétation $\#$ des termes vers des expressions diophantiennes par $\#(f(t_1, \ldots, t_n)) = \#'(t_1) + \ldots + \#'(t_n)$ où $\#'(x) = x$ pour $x \in X$ et $\#'(t) = 1$ sinon.

On peut alors définir l'ordre AC-RPO.

Définition 80. — Soient s et t deux termes de $T(\mathcal{F}, X)$, $\succ$ une précédence sur $\mathcal{F}$ et $>_{\text{ÉVAL.}}$ l'ordre d'évaluation sur les fonctions sur $\mathbb{N}$. L'ordre AC-RPO est défini par $s = f(s_1, \ldots, s_n) >_{\text{AC-RPO}} g(t_1, \ldots, t_m) = t$ si et seulement si l'une des conditions suivantes est vérifiée :

1. $s_i \geq_{\text{AC-RPO}} t$ pour un i, $1 \leq i \leq n$;
2. $f \succ g$ et $s >_{\text{AC-RPO}} t_j$ pour tout j, $1 \leq j \leq m$;
3. $f = g \notin \mathcal{F}_{\text{AC}}$, $[s_1, \ldots, s_n] >^{\mathsf{lex}}_{\text{AC-RPO}} [t_1, \ldots, t_n]$ et $s >_{\text{AC-RPO}} t_j$ pour tout j, $1 \leq j \leq m$;
4. $f = g \in \mathcal{F}_{\text{AC}}$ et il existe un $s' \in \mathit{EmbNoBig}(s)$ tel que $s' \geq_{\text{AC-RPO}} t$;
5. $f = g \in \mathcal{F}_{\text{AC}}$, $s >_{\text{AC-RPO}} t'$ pour tout $t' \in \mathit{EmbNoBig}(t)$, $\mathit{NoSmall}(s)(\geq_{\text{AC-RPO}})^f_{\mathsf{mul}}\mathit{NoSmall}(t)$ et ou bien :
   (a) $\mathit{BigHead}(s)(>_{\text{AC-RPO}})_{\mathsf{mul}}\mathit{BigHead}(t)$ ou
   (b) $\#(s) >_{\text{ÉVAL.}} \#(t)$ ou
   (c) $\#(s) \geq_{\text{ÉVAL.}} \#(t)$ et $\{s_1\ ; \ldots ;\ s_n\}(>_{\text{AC-RPO}})_{\mathsf{mul}}\{t_1\ ; \ldots ;\ t_m\}$.

Théorème 24. (Rubio 99) [78]
AC-RPO est un ordre de simplification compatible avec AC.

Chapitre III

Critères de terminaison et paires de dépendance

Nous débuterons ce chapitre par la première[1] approche par paires de dépendance de la terminaison AC. Cette méthode [58] n'utilise que des paires sans marques.

L'introduction de marques est fortement compliquée par le caractère associatif et commutatif de certains symboles. Le marquage empêche en effet la commutation (lors des pas AC) des symboles sans marque et de leurs copies marquées.

Une définition intentionnelle des ensembles de paires de dépendance marquées nous permettra ensuite de définir une nouvelle approche par paires de dépendance marquées des preuves de terminaison modulo AC.

Le gain de puissance dû aux paires de dépendance est toutefois obtenu au prix d'une augmentation très sensible de leur nombre. Le passage aux critères avec marques ajoute encore une dimension combinatoire à ce coût.

III.1 Un premier critère à l'aide de paires de dépendance

Nous nous attachons dans un premier temps à définir un critère de terminaison AC à l'aide de paires de dépendance non marquées.

L'idée principale reste la même : montrer que si un système ne termine pas, il existe des réductions infinies faisant intervenir des paires de dépendance. Il nous faut donc définir ces paires dans le cadre de la récriture associative-commutative.

Définition 81. — À l'ensemble des paires de dépendance d'une règle $f(t_1, \ldots, t_n) \to r$ on ajoute, si f est un symbole associatif-commutatif, les paires de $f(t_1, \ldots, t_n, x) \to \overline{f(r, x)}$ où x est une nouvelle variable n'apparaissant pas dans la règle. L'union de ces paires forme alors l'ensemble des paires de dépendance *étendues AC*.

Afin d'obtenir un représentant canonique de la classe d'équivalence AC de $f(r, x)$ l'aplatissement est requis si $\Lambda(r) = f$.

Remarque 20. — Si la règle ne demande pas de filtrage étendu (proposition 13) alors il n'est pas nécessaire de considérer ses paires étendues AC.

[1] Une autre approche a été proposée indépendamment à la même époque par Kusakari et Toyama [55].

Exemple 37.

Reprenons l'exemple 25 de calcul dans l'arithmétique de Peàno. À partir du système constitué des quatre règles :

$$\begin{array}{llll@{\qquad}llll}
x+0 & \to & x & (1) & x \times 0 & \to & 0 & (3)\\
x+s(y) & \to & s(x+y) & (2) & x \times s(y) & \to & (x \times y)+x & (4)
\end{array}$$

On peut extraire de la règle (2) *les paires de dépendance étendues AC :*

$$\begin{array}{l}
\langle x+s(y), x+y\rangle,\\
\langle x+s(y)+z, x+y\rangle,\\
\langle x+s(y)+z, s(x+y)+z\rangle;
\end{array}$$

De la règle (3) *:*

$$\langle x \times 0 \times z, 0 \times z\rangle;$$

Enfin de la quatrième règle :

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

Afin d'obtenir un critère de terminaison il nous faut tout d'abord garantir (comme dans le cas de la récriture standard) l'existence d'une réduction particulière.

Lemme 15.

Soient $R(\mathcal{F})$ un système AC non fortement normalisant et t un terme non fortement normalisable par R, alors t contient un sous-terme $u = f(u_1, \ldots, u_n)$ tel que :

1. u est non fortement normalisable ;
2. Tous les u_i, $1 \leq i \leq n$, sont fortement normalisables ;
3. f est un symbole défini ;
4. Si $f \in \mathcal{F}_{\mathrm{AC}}$, alors il existe un k, $2 \leq k \leq n$, tel que $f(u_1, \ldots, u_k)$ est non fortement normalisable cependant que $f(u_1, \ldots, u_{k-1})$ est fortement normalisable.

PREUVE. À l'aide d'un argument de minimalité, on peut trouver un sous-terme u satisfaisant (1) et (2). Toute réduction infinie partant de u doit comporter un pas de réduction en tête du terme, f est donc défini. Si f est AC, on trouve également k par minimalité. □

Dans toute la suite nous utiliserons la notation $f(\mathbf{u})$ pour le terme $f(u_1, \ldots, u_n)$ et $f(\mathbf{u}', \mathbf{u}'')$ pour $f(u_1, \ldots, u_k, u_{k+1}, \ldots, u_n)$. On peut remarquer que $\mathbf{u}''$ est vide lorsque $k = n$.

Nous dirons qu'un vecteur $\mathbf{u}$ est fortement normalisable si chacune de ses composantes l'est.

Lemme 16.

Soient R un TRS non fortement normalisant et t un terme non fortement normalisable par R.

Alors il existe des positions p et p', des contextes C et C', une paire de dépendance $\langle s_1, s_2\rangle$ et une substitution σ tels que :

$$t \xrightarrow[R]{>p}{}^{\star} C[s_1\sigma]_p \xrightarrow[R]{p} C[C'[s_2\sigma]_{p'}]_p$$

Où l'instance $s_2\sigma$ est non fortement normalisable.

PREUVE. Si t est non fortement normalisable alors par le lemme 15, il peut être écrit $C[f(\mathbf{u})]_p$ où $f(\mathbf{u})$ n'est pas fortement normalisable alors que $\mathbf{u}$ l'est.

1. Si f n'est pas un symbole associatif-commutatif.
   On peut déduire du fait que $\mathbf{u}$ est fortement normalisable qu'il doit y avoir un pas de récriture en tête du terme (c'est-à-dire à la position Λ) dans toute réduction infinie de $f(\mathbf{u})$. On sait donc qu'il existe une dérivation infinie à partir de t commençant comme

   $$C[f(\mathbf{u})]_p \xrightarrow[R]{>p}{}^{\star} C[f(\mathbf{v})]_p \xrightarrow[l\to r]{p} C[t']_p$$

   Où $l \to r$ est une règle de R et où t' est non fortement normalisable.
   Ceci signifie qu'il existe une substitution σ telle que :

   $$f(\mathbf{v}) \equiv \overline{l\sigma} \text{ et } t' \equiv \overline{r\sigma}.$$

   Puisque t' est lui-même non fortement normalisable, le lemme 15 nous donne que $t' = C'[f'(\mathbf{u}')]_{p'}$ où $f'(\mathbf{u}')$ n'est pas fortement normalisable.
   En outre, pour toute variable y de l, $y\sigma$ est un sous-terme de $\mathbf{v}$ et par suite est fortement normalisable. Le symbole f' à la position p' dans t' ne provient donc pas de la substitution σ : le terme r peut s'écrire $C'[f'(\mathbf{w})]_{p'}$ où $\mathbf{w}\sigma = \mathbf{u}'$ et la paire $\langle l, f'(\mathbf{w})\rangle$ est bien une paire de dépendance de $l \to r$.
2. Si f est associatif-commutatif.
   Du lemme 15, nous savons qu'il existe un k tel que $f(\mathbf{u}) = f(u_1, \ldots, u_k, u_{k+1}, \ldots, u_n) = f(\mathbf{u}', \mathbf{u}'')$ où $f(\mathbf{u}')$ n'est pas fortement normalisable alors que $f(u_1, \ldots, u_{k-1})$ l'est. Comme dans le cas précédent, $\mathbf{u}'$ fortement normalisable entraîne que dans toute réduction infinie de $f(\mathbf{u}')$, on peut trouver un pas de récriture à Λ.
   Il existe donc une réduction infinie à partir de t commençant comme

   $$C[f(\mathbf{u}', \mathbf{u}'')]_p \xrightarrow[R]{>p}{}^{\star} C[f(\mathbf{v}, \mathbf{u}'')]_p \xrightarrow[l\to r]{p} C[t']_p$$

   Où $l \to r \in R$ et t' n'est pas fortement normalisable.
   Il y a ainsi une substitution σ telle que soit :

   $$f(\mathbf{v}, \mathbf{u}'') \equiv \overline{l\sigma}$$

   Soit (en utilisant le filtrage étendu) :

   $$f(\mathbf{v}, \mathbf{u}'') \equiv \overline{f(l, x)\sigma}$$

   Avec une nouvelle variable x.
   Nous allons distinguer deux cas suivant l'utilisation ou non du filtrage étendu par le pas considéré. Remarquons toutefois que dans la mesure où nous étudions une réduction infinie de $f(\mathbf{u}')$, si $\mathbf{u}''$ n'est pas vide alors le filtrage est nécessairement étendu.

(a) Le filtrage n'est pas étendu.
Comme nous venons de le faire remarquer, $\mathbf{v}''$ est vide, nous avons donc

$$f(\mathbf{v}) \equiv \overline{l\sigma} \xrightarrow{\Lambda} t'.$$

Puisque t' est non fortement normalisable, nous pouvons grâce au lemme 15 écrire

$$t' = C'[f'(\mathbf{w}')]_{p'}$$

Où $f'(\mathbf{w}')$ n'est pas fortement normalisable. Comme dans le premier cas, la position p' ne peut être à l'intérieur de la substitution σ, ainsi $\mathbf{w}' = \mathbf{w}\sigma$ et $\langle l, f'(\mathbf{w})\rangle$ est une paire de dépendance de R.

(b) Le filtrage est étendu.
Nous avons

$$f(\mathbf{v}, \mathbf{u}'') \equiv \overline{f(l, x)\sigma}$$

Où l'étape de récriture est en fait une récriture de $f(\mathbf{v})$: $\mathbf{u}''$ fait donc partie de $x\sigma$. $f(\mathbf{v})$ peut ainsi s'écrire $f(\mathbf{v}', \mathbf{v}'')$ ($\mathbf{v}''$ éventuellement vide) où $f(\mathbf{v}') = l\sigma$ et $f(\mathbf{v}'', \mathbf{u}'') = x\sigma$ ($\mathbf{u}''$ ou $\mathbf{v}''$ est non vide).
Notre réduction infinie de t commence donc comme :

$$C[f(\mathbf{u}', \mathbf{u}'')]_p \xrightarrow[R]{>p}{}^{\star} C[f(\mathbf{v}', \mathbf{v}'', \mathbf{u}'')]_p \xrightarrow[l\to r]{p} C[f(\mathbf{r}\sigma, \mathbf{v}'', \mathbf{u}'')]_p$$

Où $\mathbf{r} = r$ si $\Lambda(r)$ est différent de f et $r = f(\mathbf{r})$ sinon.
Puisque $f(\mathbf{r}\sigma, \mathbf{v}'')$ est non fortement normalisable, nous savons grâce au lemme 15 que :

$$f(\mathbf{r}\sigma, \mathbf{v}'') = C'[f'(\mathbf{w}')]_{p'}$$

Où $f'(\mathbf{w}')$ est non fortement normalisable. Ainsi p' n'appartient pas à la partie instanciée par σ et $\mathbf{w}' = \mathbf{w}\sigma$.

– Si $f' \neq f$ ou si C'' n'est pas le contexte vide, le symbole f' ne peut provenir de σ car $\mathbf{v}''$ est fortement normalisable. f' apparaît donc dans r et $\langle f(l, x), f'(\mathbf{w})\rangle$ est une paire de dépendance de $l \to r$.
– Si $f' = f$ et si C'' est vide, $\langle f(l, x), f(r, x)\rangle$ est immédiatement une paire de dépendance de $l \to r$.

□

Théorème 25.
La relation $\xrightarrow[R]{}$ induite par un système de récriture associatif-commutatif R termine si et seulement s'il n'existe pas de chaînes de dépendance de R infinies.

PREUVE. Soient R un TRS AC non fortement normalisant et t un terme non fortement normalisable par R. On peut enchaîner les applications du lemme 16 à t afin d'obtenir une réduction infinie

$$\cdots \xrightarrow[l_1\to r_1]{p_1} C_2[f_2(\mathbf{u_2})]_{p_2} \xrightarrow[R]{>p_2}{}^{\star} C_2[f_2(\mathbf{v_2})]_{p_2} \xrightarrow[l_2\to r_2]{p_2} C_3[f_3(\mathbf{u_3})]_{p_3} \cdots$$

N'oublions pas qu'au cours de cette réduction, à chaque étape entre deux paires, on a :

$$f_i(\mathbf{u_i}) \xrightarrow[R]{>\Lambda}{}^{\star} f_i(\mathbf{v_i}).$$

Réciproquement, s'il existe une chaîne de dépendance infinie $\ldots \langle s_i, t_i \rangle \langle s_{i+1}, t_{i+1} \rangle \ldots$, on obtient directement de la définition des chaînes (définition 36) une réduction infinie par R avec

$$\ldots t_i\sigma \xrightarrow[R]{>\Lambda}{}^{\star} s_{i+1}\sigma$$

Suivi d'un pas à l'intérieur de $t_{i+1}\sigma$, sous-terme de $s_{i+1}\sigma$. □

Corollaire et application

Le précédent théorème peut être appliqué à la preuve de terminaison grâce au corollaire 25–1.

Corollaire 25–1.
S'il existe un ordre de réduction faible AC-compatible $(\succeq, \succ)$ tel que

– Pour chaque règle $l \rightarrow r \in R$, $l \succeq r$ et

– Pour chaque paire de dépendance $\langle s, t \rangle$ de R, $s \succ t$,

Alors R est fortement normalisant.

PREUVE. Par contradiction : posons $(\succeq, \succ)$ un tel ordre et supposons que R ne termine pas. Nous disposons alors d'une chaîne infinie $\ldots \langle s_i, t_i \rangle \langle s_{i+1}, t_{i+1} \rangle \ldots$ munie d'une substitution σ. De la stabilité de $(\succeq, \succ)$ et puisque $s_i \succ t_i$ nous savons que

$$s_i\sigma \succ t_i\sigma \ ;$$

De $t_i\sigma \xrightarrow[R]{>\Lambda}{}^{\star} s_{i+1}\sigma$, $l \succeq r$ pour toute règle $l \rightarrow r$ et puisque $(\succeq, \succ)$ est faiblement monotone nous avons en outre

$$t_i\sigma \succeq s_{i+1}\sigma.$$

Nous pouvons donc conclure sur l'existence d'une séquence décroissante infinie

$$\ldots s_i\sigma \succ t_i\sigma \succeq s_{i+1}\sigma \succ t_{i+1}\sigma \ldots$$

Une telle séquence contredit la nature bien fondée de $(\succeq, \succ)$. □

Exemple 38.
Considérons le système suivant, où $\cup$ et $\cap$ sont AC *et* eq *commutatif, qui calcule l'intersection multiensembliste. Les multiensembles d'entiers de Peàno sont représentés par la constante $\emptyset$, le symbole $\{\_\}$ qui construit un singleton et*

l'union.

$$\mathcal{F} \quad \left\{ \begin{array}{l} \emptyset, \mathit{true}, \mathit{false}, 0 : \textit{constantes} \,;\, \{\_\}, s : \textit{unaire} \,; \\ \cup, \cap, \mathit{eq} : \textit{binaires}, \mathit{if} : \textit{ternaire} \end{array} \right\}$$

$$R \quad \left\{ \begin{array}{lcl} \mathit{if}(\mathit{true}, x, y) & \to & x \\ \mathit{if}(\mathit{false}, x, y) & \to & y \\ \mathit{eq}(0,0) & \to & \mathit{true} \\ \mathit{eq}(0, s(x)) & \to & \mathit{false} \\ \mathit{eq}(s(x), s(y)) & \to & \mathit{eq}(x,y) \\ \emptyset \cup x & \to & x \\ x \cap \emptyset & \to & \emptyset \\ x \cap (y \cup z) & \to & (x \cap y) \cup (x \cap z) \\ \{x\} \cap \{y\} & \to & \mathit{if}(\mathit{eq}(x,y), \{x\}, \emptyset). \end{array} \right.$$

Pour prouver sa terminaison en appliquant le lemme 25–1 il nous faut trouver un ordre convenable, faisant décroître largement les règles de R *et strictement les paires étendues* AC *de* R*, à savoir :*

$$\begin{array}{l} \langle \mathit{eq}(s(x), s(y)), \mathit{eq}(x,y) \rangle, \\ \langle \{x\} \cap \{y\}, \mathit{if}(\mathit{eq}(x,y), \{x\}, \emptyset) \rangle, \\ \langle \{x\} \cap \{y\}, \mathit{eq}(x,y) \rangle, \\ \langle x \cap (y \cup z), (x \cap y) \cup (x \cap z) \rangle, \\ \langle x \cap (y \cup z), x \cap y \rangle, \\ \langle x \cap (y \cup z), x \cap z \rangle, \\ \langle y \cap x \cap \emptyset, y \cap \emptyset \rangle, \\ \langle z \cap \{x\} \cap \{y\}, \mathit{if}(\mathit{eq}(x,y), \{x\}, \emptyset) \rangle, \\ \langle z \cap \{x\} \cap \{y\}, \mathit{eq}(x,y) \rangle, \\ \langle z \cap \{x\} \cap \{y\}, z \cap \mathit{if}(\mathit{eq}(x,y), \{x\}, \emptyset) \rangle, \\ \langle t \cap x \cap (y \cup z), (x \cap y) \cup (x \cap z) \rangle, \\ \langle t \cap x \cap (y \cup z), x \cap y \rangle, \\ \langle t \cap x \cap (y \cup z), x \cap z \rangle, \\ \langle t \cap x \cap (y \cup z), t \cap ((x \cap y) \cup (x \cap z)) \rangle. \end{array}$$

L'ordre induit par l'interprétation polynomiale :

$$\begin{array}{l} [\![\mathit{true}]\!] = 0, \\ [\![\mathit{false}]\!] = 0, \\ [\![\mathit{if}]\!](x_0, x_1, x_2) = x_2 + x_1, \\ [\![0]\!] = 0, \\ [\![s]\!](x) = x + 1, \\ [\![\mathit{eq}]\!](x_0, x_1) = x_0 x_1, \\ [\![\emptyset]\!] = 0, \\ [\![\{\_\}]\!](x) = x, \\ [\![\cup]\!](x_0, x_1) = x_1 + x_0 + 2, \\ [\![\cap]\!](x_0, x_1) = 2x_0x_1 + 2x_1 + 2x_0 + 1. \end{array}$$

Suffit à prouver la terminaison AC *du système.*

III.2 Critères avec symboles marqués

Nous allons maintenant définir des critères de terminaison tirant profit d'un marquage de certains symboles et généralisant ainsi les critères avec symboles marqués utilisés pour la récriture standard.

L'introduction des marques dans les preuves de terminaison est rendue techniquement délicate dès qu'on aborde le cas de la récriture AC. Les pas d'associativité et de commutativité ne permettent plus de se limiter à la traduction d'une paire non marquée vers une paire marquée. Nous verrons en particulier qu'à une paire sans marques peuvent correspondre plusieurs paires marquées.

L'extension aux critères avec marques se fera en deux temps. Nous définirons tout d'abord des conditions abstraites dites *conditions de dépendance* consistant en un ensemble de contraintes devant être satisfaites par un ensemble de paires marquées préservant la correction du critère ; nous donnerons ensuite deux méthodes permettant de construire effectivement des ensembles de paires adéquats.

III.2.1 Symboles marqués et termes marqués

Supposons donné comme précédemment un système de récriture R dont nous voulons prouver la terminaison à l'aide de critères avec marques. À cette fin nous définissons tout d'abord l'ensemble des symboles marqués ainsi que l'opération de marquage en tête d'un terme.

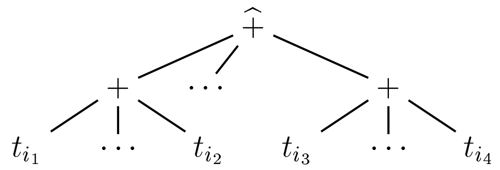
Si le critère standard des paires de dépendance ne considère que des termes dont le symbole de tête est le seul marqué, nous allons quant à nous être confrontés à des symboles AC et donc amenés à considérer des termes *partiellement marqués*. Ces termes sont ceux dont le symbole de tête $\widehat{f}$ est marqué mais qui en outre peuvent comporter des occurrences non marquées f sous $\widehat{f}$, y compris si celui-ci est AC.

Définition 82. — Soit t un terme dont le symbole de tête est f. Un terme t' est dit *partiellement marqué* de t (ou issu d'un marquage partiel de t) si :

- $\Lambda(t') = \widehat{f}$;
- Le seul symbole marqué de t' est $\widehat{f}$;
- Si f n'est pas AC, la seule occurrence de $\widehat{f}$ est à Λ ;
- Une occurrence de $\widehat{f}$ n'apparaît jamais dans t' en dessous d'un symbole non marqué ;
- Le terme obtenu en remplaçant dans t' chaque symbole marqué $\widehat{f}$ par sa copie non marquée f est AC équivalent à t.

Nous désignerons par $Par(t)$ l'ensemble des termes partiellement marqués de t.

Illustrons cette définition de façon plus graphique : en posant que $+$ est associatif-commutatif nous pouvons décrire $Par(t_1 + \cdots + t_n)$ comme l'ensemble des termes de la forme



Pour toute permutation des t_i. Remarquons en particulier que $\widehat{t} \in Par(t)$ (voir définition 34, page 31).

Exemple 39.

Reprenons le cas du système de multiplication d'entiers de Peàno. Les seuls symboles définis de $\mathcal{F}$ *étant* $+$ *et* $\times$, $\widehat{\mathcal{F}} = \{0\ ;\ s\ ;\ +\ ;\ \times\ ;\ \widehat{+}\ ;\ \widehat{\times}\}$. *Nous avons donc par exemple :*

$$\begin{array}{rcl} \widehat{0+s(0)} & = & 0\widehat{+}s(0), \\ \mathit{Par}(0+s(0)) & = & \{0\widehat{+}s(0)\}, \\ 0+s(\widehat{0)+s}(s(0)) & = & 0\widehat{+}s(0)\widehat{+}s(s(0)), \\ \mathit{Par}(0+s(0)+s(s(0))) & = & \{0\widehat{+}s(0)\widehat{+}s(s(0))\ ; \\ & & \ (0+s(0))\widehat{+}s(s(0))\ ; \\ & & \ 0\widehat{+}(s(0)+s(s(0)))\ ; \\ & & \ s(0)\widehat{+}(0+s(s(0)))\}. \end{array}$$

Définition 83. — Soit R un TRS, $\widehat{R}$ désigne le système d'« effacement des marques » :

$$\{\widehat{f}(\mathbf{u}) \to f(\mathbf{u}) \ \mid f \text{ symbole défini dans } R\}.$$

Nous noterons $\xrightarrow{m.e.}$ la relation de récriture modulo AC induite par $\widehat{R}$ restreinte aux paires de termes partiellement marqués d'un même t telle que des pas de $\xrightarrow{m.e.}$ ne peuvent en aucun cas effacer la marque située à Λ.

$\mathit{Par}_{\downarrow}(t)$ représente l'ensemble des $u \in \mathit{Par}(t)$ en $\xrightarrow{m.e.}$ forme normale.

Exemple 40.

Si $+$ *est un symbole* AC *défini, il est possible de récrire* $0\widehat{+}s(0)\widehat{+}s(s(0))$ *par* $\xrightarrow{m.e.}$ *en chacun des trois termes* $(0+s(0))\widehat{+}s(s(0))$, $0\widehat{+}(s(0)+s(s(0)))$, *ou* $s(0)\widehat{+}(0+s(s(0)))$, *lesquels sont tous en forme normale.*

Proposition 14.

Pour tout terme t et tout $t' \in \mathit{Par}(t)$,

$$\widehat{t} \xrightarrow{m.e.}{}^{\star} t'.$$

PREUVE. Trivial. □

Remarquons qu'en particulier, $\mathit{Par}_{\downarrow}(t)$ est l'ensemble des formes normales de $\widehat{t}$ pour $\xrightarrow{m.e.}$.

Définition 84. — Si $\prec$ est l'ordre préfixe, la *tête* d'un terme t est définie comme :

$$\mathit{Head}(t) = \left\{ p \in \mathcal{P}os(t) \text{ tels que } \ \forall q \prec p, \Lambda(t_q) = \Lambda(t) \text{ ou } \Lambda(t_q) = \widehat{\Lambda(t)} \ \right\}.$$

Réciproquement, nous définirons le *corps* de t comme $\mathcal{P}os(t) \setminus \mathit{Head}(t)$.

Les pas de récriture seront internes ou externes suivant leur position, c'est-à-dire s'ils sont respectivement dans le corps ou non. Nous noterons $s \xrightarrow{i} t$ les pas internes et $s \xrightarrow{o} t$ les externes.

Exemple 41.

Considérons le système R ne comportant qu'une seule règle $g(x) \to x$ sur la signature $\mathcal{F}$: $\{a$: constante ; g : unaire ; $+$: binaire$\}$. Pour le terme $s = a + g(a + a)$:

$$Head(s) = \{\Lambda\}.$$

Puisque $s \to t = a + (a + a)$, on a :

$$Head(t) = \{\Lambda\ ;\ 2\}.$$

III.2.2 Conditions de dépendance

Nous pouvons à présent définir les conditions de dépendance.

Définition 85. — Soit $\langle u, v\rangle$ une paire (non marquée) de R. On appellera un ensemble de paires $\{\ldots ;\ \langle u_i, v_i\rangle\ ; \ldots\}$ *ensemble de paires de dépendance marquées* pour $\langle u, v\rangle$ si chacune des *conditions de dépendance* suivantes est vérifiée :

1. Pour tout i, il existe une substitution σ_i telle que $\widehat{v\sigma_i} \xrightarrow{m.e.}{}^{\star} v_i$;
2. Pour toute substitution σ, pour tout $t \in Par(u\sigma)$, il existe un i et une substitution σ' tels que

$$\begin{aligned} &\sigma =_{\mathrm{AC}} \sigma_i\sigma', \\ &t \xrightarrow{m.e.}{}^{\star} u_i\sigma'. \end{aligned}$$

Un *ensemble de paires de dépendance marquées* pour R est une union des ensembles de paires de dépendance marquées pour toutes les paires non marquées de R.

Supposons donné un ensemble de paires de dépendance marquées de R. Les *chaînes de dépendance marquées* AC sont définies comme suit.

Définition 86. — Une *chaîne marquée* de paires de dépendance est une séquence $\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \ldots$ de paires marquées de R telle que, pour une substitution σ et pour tout i,

$$t_i\sigma(\xrightarrow[R]{i})^{\star} \cdot \xrightarrow{m.e.}{}^{\star} s_{i+1}\sigma.$$

Théorème 26.

Soient R un TRS et $\widehat{DP}$ un ensemble de paires marquées. Supposons que $\widehat{DP}$ vérifie les conditions de dépendance. S'il existe une chaîne infinie non marquée, alors il existe une chaîne infinie marquée.

PREUVE. Nous allons construire la chaîne marquée à partir de la chaîne non marquée. Pour ce faire, nous montrons deux lemmes intermédiaires.

Lemme 17.

Si $t(\xrightarrow{i})^{\star}s$ alors pour tout $t' \in Par(t)$ il existe un $s' \in Par(s)$ tel que $t'(\xrightarrow{i})^{\star}s'$.

PREUVE. Hypothèse : $t \xrightarrow{i\ n} s \implies \forall t' \exists s'$ tel que $t' \xrightarrow{i\ \star} s'$.
Par induction sur n :

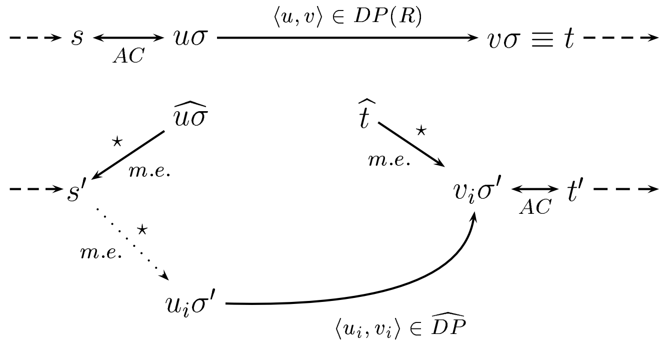


FIG. III.1. Chaîne sans marques et chaîne marquée correspondante.

– Cas de base : $t =_{\mathrm{AC}} s$ nous pouvons alors choisir comme s' n'importe quel membre de la classe d'équivalence AC de t'.

– Cas général : $t \xrightarrow{i\ n+1} s$ peut être décomposé en $t \xrightarrow{i\ n} u \xrightarrow{i} s$. Puisque les pas sont internes, on peut supposer sans perte de généralité que $Head(s) = \wedge$.
Posons donc $u = f(u_1, \ldots, u_k)$ et $s = f(s_1, \ldots, s_k)$; il existe $i \in \{1, \ldots, k\}$ tel que $u_i \to s_i$.
Ainsi pour tout u', il existe un $s' \in Par(s)$ tel que $u' \xrightarrow{i} s'$.

Par hypothèse $\forall t' \in Par(t)$, $\exists u' \in Par(u)$ tel que $t' \xrightarrow{i\ \star} u'$, nous obtenons ainsi $t' \xrightarrow{i\ \star} u' \xrightarrow{i} s'$ c'est-à-dire $t' \xrightarrow{i\ \star} s'$.

□

Lemme 18.
Si $s \xrightarrow[\langle u,v\rangle]{DP} t$ alors pour tout $s' \in Par(s)$, il existe une paire $\langle u_i, v_i\rangle \in \widehat{DP}$ et deux termes $s'' \in Par(s)$ et $t' \in Par(t)$ tels que

$$s' \xrightarrow{m.e.}{}^{\star} s'' \xrightarrow[\langle u_i,v_i\rangle]{\widehat{DP}} t'.$$

En outre, si $s = u\sigma$ alors $t' = \widehat{v\sigma}$.

PREUVE. Nous allons exhiber s' et t' convenables pour s et t donnés, la construction étant illustrée par la figure III.1.

Si $s \xrightarrow[\langle u,v\rangle]{DP} t$ alors il existe une substitution σ telle que $s =_{AC} u\sigma$ et $t \equiv v\sigma$. Considérons $\widehat{(u\sigma)}$:

– $\widehat{(u\sigma)} \xrightarrow{m.e.}{}^{\star} \widehat{u}\sigma$, car $Head(u) \subseteq Head(u\sigma)$;

– $\widehat{(u\sigma)} \xrightarrow{m.e.}{}^{\star} s'$, puisque $s =_{AC} u\sigma \implies \widehat{(u\sigma)} =_{AC} \widehat{s} \xrightarrow[\widehat{R}]{\star} s'$.

Donc $s' \in Par(u\sigma)$ et par la condition 2 de la définition 85 nous obtenons un i et une substitution σ' tels que

$$\sigma =_{AC} \sigma_i\sigma' \text{ et } s \xrightarrow{m.e.}{}^{\star} u_i\sigma'.$$

Ainsi, en choisissant $s'' = u_i\sigma'$, nous avons

$$s' \xrightarrow{m.e.}{}^{\star} s'' \xrightarrow[\langle u_i, v_i \rangle]{\widehat{DP}} v_i\sigma';$$

En posant $t' = v_i\sigma'$, il nous reste à prouver que $t' \in \mathit{Par}(t)$. La condition 1 de la définition 85 nous dit alors que $\widehat{v\sigma_i} \xrightarrow{m.e.}{}^{\star} v_i$, c'est-à-dire, puisque $t = v\sigma$ et $\sigma =_{AC} \sigma_i\sigma'$, que $\widehat{t} \xrightarrow{m.e.}{}^{\star} v_i\sigma' = t'$. □

Preuve du théorème 26

Soit $\langle s_1, t_1 \rangle, \langle s_2, t_2 \rangle, \ldots$ une chaîne de dépendance non marquée, infinie et munie d'une substitution σ telle que pour tout i,

$$t_i\sigma \xrightarrow{\neq\Lambda} s_{i+1}\sigma.$$

Nous allons construire inductivement une chaîne infinie marquée à partir de cette chaîne en « traduisant » chacun de ses pas.

À partir d'un quelconque $s'_1 \in \mathit{Par}(s_1\sigma)$, nous pouvons appliquer le lemme 18 afin d'obtenir une paire marquée $\langle s_{1_i}, t_{1_i} \rangle$ telle que, pour un $t'_1 \in \mathit{Par}(t_1\sigma)$,

$$s'_1 \xrightarrow{m.e.}{}^{\star} s''_1 \xrightarrow[\langle s_{1_i}, t_{1_i} \rangle]{\widehat{DP}} t'_1.$$

Il nous suffit alors d'appliquer le lemme 17 à t'_1 pour avoir un s'_2 tel que $t'_1 \xrightarrow{i}{}^{\star} s'_2$ avec $s'_2 \in \mathit{Par}(s_2\sigma)$.

En procédant de la sorte pour chaque pas on obtient par induction la chaîne désirée. □

III.2.3 Critères de terminaisons avec marques

Les corollaires suivants nous munissent de critères de terminaison avec marques. Dans la mesure où certains pas effacent les marques, nous demanderons que ceux-ci décroissent largement pour l'ordre utilisé dans la preuve de terminaison.

Corollaire 26–1.

S'il existe un ordre de réduction faible AC-compatible $\succeq$ tel que $l \succeq r$ pour chaque règle de R, $s \succ t$ pour toute paire de dépendance $\langle s, t \rangle$ de R et, pour chaque symbole défini AC f, $\widehat{f}(x, y) \succeq f(x, y)$ alors R est AC fortement normalisant.

Preuve. La démonstration de ce corollaire est tout à fait similaire à celle du corollaire 25–1. La dernière condition nous assure de la décroissance (large) des pas d'effacement. □

L'optimisation par graphes de dépendance s'adapte directement au critère avec marques.

Corollaire 26–2.

Soit R un système de récriture fini. Soient $\widehat{DP}(R)$ un ensemble de paires de dépendance marquées et $\mathcal{G}$ le graphe de dépendance (approché) associé. S'il existe un ordre de réduction faible AC compatible $\succeq$ tel que :

– $l \succeq r$ pour toute règle $l \to r \in R$;

– $\widehat{f}(x, y) \succeq f(x, y)$ pour tout symbole AC défini f ;
– $u \succeq v$ pour chaque paire $\langle u, v\rangle \in \widehat{DP}(R)$ de toute partie fortement connexe de $\mathcal{G}$;
– $u \succ v$ pour au moins une paire $\langle u, v\rangle$ de chaque partie fortement connexe de $\mathcal{G}$;

Alors R est AC fortement normalisant.

III.2.4 Calcul des paires marquées

Ce qu'il nous faut après le théorème 26 est un moyen de calculer un ensemble $\widehat{DP}$ idoine pour un système quelconque.

On peut remarquer à ce propos que poser des marques sur la tête des membres aplatis des paires de dépendance non marquées *ne suffit pas* comme le montre l'exemple 42.

Exemple 42.
Soit R le système suivant :

$$\begin{cases} f(x) & \to & x + a \\ c + a & \to & f(b + c). \end{cases}$$

Ce système n'est pas fortement normalisant car $f(b + c)$ peut être réduit indéfiniment :

$$f(b + c) \to b + c + a \to b + f(b + c) \to \cdots$$

Il a six paires de dépendance

$$\begin{array}{ll} \langle f(x), x + a\rangle & \langle c + a + x, f(b + c) + x\rangle \\ \langle c + a, f(b + c)\rangle & \langle c + a + x, f(b + c)\rangle \\ \langle c + a, b + c\rangle & \langle c + a + x, b + c\rangle \end{array}$$

Et la chaîne infinie correspondant à la réduction de $f(b + c)$ est :

$$\langle f(b + c), b + c + a\rangle, \langle c + a + b, f(b + c)\rangle, \langle f(b + c), b + c + a\rangle, \ldots$$

En se limitant à marquer la tête des membres gauches, on obtient un ensemble de six paires

$$\begin{array}{ll} \langle \widehat{f}(x), x \widehat{+} a\rangle & \langle c \widehat{+} a \widehat{+} x, f(b + c) \widehat{+} x\rangle \\ \langle c \widehat{+} a, \widehat{f}(b + c)\rangle & \langle c \widehat{+} a \widehat{+} x, \widehat{f}(b + c)\rangle \\ \langle c \widehat{+} a, b \widehat{+} c\rangle & \langle c \widehat{+} a \widehat{+} x, b \widehat{+} c\rangle \end{array}$$

Pour lequel la chaîne correspondante est :

$$\langle \widehat{f}(b + c), (b + c) \widehat{+} a\rangle$$

Forcément finie puisqu'aucun membre gauche n'est filtré par $(b + c) \widehat{+} a$.

Définition 87. — Soit $t = u_1 + \ldots + u_n$ un terme aplati et non marqué où $+$ est AC. Nous noterons

$$\begin{array}{rcl} I_t & = & \{i \in 1, \ldots, n \mid u_i \text{ est une variable}\}, \\ X_t & = & \{u_i \mid i \in I_t\} \text{ et} \\ m_t(x) & = & \#\{i \in I_t \mid u_i = x\}. \end{array}$$

$m_t(x)$ est appelé la *multiplicité* de x dans t.

Exemple 43.
Pour le terme $t = x + x + f(y) + z + f(z)$ *:* $I_t = \{1, 2, 4\}$, $X_t = \{x, z\}$, $m_t(x) = 2$ *et* $m_t(z) = 1$.

Le point important d'une chaîne de dépendance non marquée est le fait qu'un terme t soit une instance *modulo AC* d'un membre gauche d'une paire de dépendance $\langle u, v\rangle$, c'est-à-dire $t =_{AC} u\sigma$, les pas AC pouvant intervenir à n'importe quelle position.

De façon similaire, pour pouvoir traduire une chaîne non marquée en une chaîne marquée, il nous faut pouvoir récrire par $\xrightarrow{m.e.}$ un terme partiellement marqué (disons $t' \in Par(t)$) en une instance modulo AC d'un membre gauche de paire. L'introduction des marques a cependant fortement contraint les opérations AC puisque la distribution d'un symbole marqué, par exemple $\widehat{+}$, sur une copie non marquée $+$ est impossible.

Nous allons donc au devant de complications si, comme dans l'exemple qui suit, ce qui devrait être reconnu comme une instance d'une variable apparaissant dans une paire marquée se retrouve partagé de part et d'autre d'un symbole marqué (partage qui, notons-le, serait aisément résolu par AC dans le cadre des paires sans marques).

Exemple 44.
Considérons la règle $\{x + x + g(y) \to h(x, y)\}$ *d'un système* R *où* g *est un symbole défini et* $+$ *est AC et regardons le terme* $t = (a_1 + a_1 + a_2 + a_4)\widehat{+}(a_2 + a_3 + a_3 + a_4 + g(y))$. *Si* t *n'est clairement pas une instance d'une version marquée de* $x + x + g(y)$, *il donne néanmoins lieu à une dérivation infinie et donc à une chaîne infinie. Nous devons donc en tenir compte.*

Pour régler cette difficulté technique, nous allons introduire une fonction *Split*. Cette fonction pour un terme t associe à une variable tous ses partages possibles en tenant compte de sa multiplicité dans t. La correction de cette fonction sera démontrée en même temps que la proposition 15.

Définition 88. — La fonction *Split* est définie des variables vers l'ensemble des ensembles de substitutions comme suit :

$$Split_t(x) = \bigcup_{i=1}^{m_t(x)} \{\sigma \mid \sigma(x) = x_0 + \ldots + x_i\}$$

Où $+$ est le symbole à Λ de t et où les x_j sont des variables fraîches.

Nous noterons Σ_t l'ensemble de toutes les compositions possibles de toutes les σ de $Split_t(x)$ pour toutes les variables x de X_t, c'est-à-dire :

$$\Sigma_t = \bigcup_{\substack{\sigma_1 \in Split_t(x_1) \\ \vdots \\ \sigma_k \in Split_t(x_k) \\ \text{où } \{x_1, \ldots, x_k\} = X_t}} \sigma_1 \circ \cdots \circ \sigma_k$$

Exemple 45.
Pour le terme $t = x + x + f(y) + z + f(z)$, *on obtient*

$$Split_t(x) = \{x \mapsto x_0, x \mapsto x_0 + x_1, x \mapsto x_0 + x_1 + x_2\}$$

Et

$$\Sigma_t = \left\{\begin{array}{l} x \mapsto x_0 \\ z \mapsto z_0 \end{array}\right. ; \left\{\begin{array}{l} x \mapsto x_0 + x_1 \\ z \mapsto z_0 \end{array}\right. ; \left\{\begin{array}{l} x \mapsto x_0 + x_1 + x_2 \\ z \mapsto z_0 \end{array}\right. ;$$
$$\left\{\begin{array}{l} x \mapsto x_0 \\ z \mapsto z_0 + z_1 \end{array}\right. ; \left\{\begin{array}{l} x \mapsto x_0 + x_1 \\ z \mapsto z_0 + z_1 \end{array}\right. ; \left\{\begin{array}{l} x \mapsto x_0 + x_1 + x_2 \\ z \mapsto z_0 + z_1 \end{array}\right.$$

Construire un ensemble de paires de dépendance marquées consiste maintenant essentiellement :

- Tout d'abord, en un calcul de Σ, ce qui est facile puisque les domaines des substitutions sont disjoints.
- Ensuite, en un marquage en tête de tous les membres de la classe d'équivalence AC des instanciations par chaque $\sigma \in \Sigma$ des membres gauches des paires non marquées ; les membres gauches des paires marquées sont alors obtenus en choisissant un élément dans chaque classe.
- Enfin, en une construction des membres droits correspondants. Nous exposerons deux variantes : dans la première nous marquerons toute la tête des instanciations des membres droits des paires non marquées alors que la seconde où le symbole à Λ est le seul marqué verra distingués tous les partages de sous-termes de part et d'autre de la position marquée.

Première méthode

Définition 89. — Soit R un système de récriture. La fonction *Pairs* est définie comme :

$$\mathit{Pairs}(R) = \bigcup_{\substack{\langle u,v\rangle \in DP(R),\\ \sigma \in \Sigma_u}} \{\langle u', \widehat{(v\sigma)}\rangle \text{ telle que } u' \in \mathit{Par}_{\downarrow}(u\sigma)\}.$$

Proposition 15.
Pairs(R) est un ensemble de paires de dépendance de R.

PREUVE. Il nous faut vérifier que pour chaque paire de dépendance de R non marquée $\langle u, v\rangle$, l'ensemble

$$\bigcup_{\sigma\in\Sigma_u} \{\langle u', \widehat{(v\sigma)}\rangle \text{ telle que } u' \in \mathit{Par}_{\downarrow}(u\sigma)\}$$

Remplit les deux conditions de dépendance.

Pour une paire marquée $\langle u_i, v_i\rangle$ de cet ensemble obtenue à partir d'une substitution $\sigma \in \Sigma_u$ donnée, la première condition

$$\widehat{v\sigma_i} \xrightarrow{m.e.}{}^{\star} v_i$$

Est trivialement vérifiée avec $\sigma_i = \sigma$ puisqu'en fait $v_i = \widehat{(v\sigma)}$.

Quant à la seconde condition, en supposant donnés une substitution σ et un terme $t \in \mathit{Par}(u\sigma)$, nous devons trouver deux substitutions σ_i et σ' vérifiant

$$\begin{aligned} \sigma &=_{AC} \sigma_i\sigma' \\ t &\xrightarrow{m.e.}{}^{\star} u_i\sigma'. \end{aligned}$$

Puisque par la proposition 14 on peut toujours réduire t avec $\xrightarrow{m.e.}$ en un terme de $\mathit{Par}_{\downarrow}(u\sigma)$, il est possible de considérer directement et sans perte de généralité que $t \in \mathit{Par}_{\downarrow}(u\sigma)$. Il reste alors à prouver que

$$\begin{aligned} \sigma &=_{AC} \sigma_i\sigma' \\ t &=_{AC} u_i\sigma'. \end{aligned}$$

Afin de favoriser une meilleure lisibilité de la preuve nous allons poser que le symbole AC en tête de t est $+$, c'est-à-dire que $t = t_1 \widehat{+} t_2$ où t_1 et t_2 ne comportent aucune marque.

Une substitution σ_i convenable va être construite par composition de substitutions $\sigma_i^x \in \mathit{Split}_u(x)$ bien choisies, pour chaque variable x de u, afin que σ_i soit dans Σ_u :

1. Si x n'apparaît pas à profondeur 1 dans u il suffit de choisir $\sigma_i(x) = x$;
2. Si x est présent à profondeur 1 dans u avec la multiplicité $m_u(x) = m$ alors
   (a) Si le symbole à la racine de $x\sigma$ n'est pas $+$, nous n'avons qu'à prendre $\sigma_i^x : x \mapsto x_0$ qui est toujours dans $\mathit{Split}_u(x)$.
   (b) Si le symbole à Λ de $x\sigma$ est $+$, disons $x\sigma = s_1 + \ldots + s_n$, dans la mesure où la multiplicité de x dans u est m, nous pouvons écrire :

   $$u = \underbrace{x + \cdots + x}_{m \text{ fois}} + u',$$

   D'où

   $$u\sigma = \underbrace{s_1 + \cdots + s_1}_{m \text{ fois}} + \cdots + \underbrace{s_n + \cdots + s_n}_{m \text{ fois}} + u'\sigma.$$

   Ainsi

   $$\widehat{u\sigma} = \underbrace{s_1 \widehat{+} \cdots \widehat{+} s_1}_{m \text{ fois}} \widehat{+} \cdots \widehat{+} \underbrace{s_n \widehat{+} \cdots \widehat{+} s_n}_{m \text{ fois}} \widehat{+} u''$$

   Où $u'' = \widehat{u'\sigma}$ si à sa racine on trouve $+$ et $u'' = u'\sigma$ sinon. Nous obtenons donc

   $$\widehat{u\sigma} \xrightarrow{m.e.}{}^{\star} t_1 \widehat{+} t_2$$

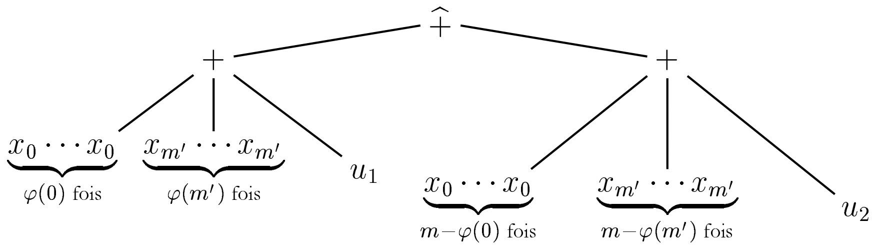
   Et tout le travail consiste à regarder combien d'occurrences des s_j restent après l'effacement des marques au sein de la partie gauche t_1 et de la partie droite t_2. Nous définissons à cet effet pour chaque k, $0 \leq k \leq m$, S_k^x comme l'ensemble des u_j apparaissant exactement k fois dans t_1 (et donc $m - k$ fois dans t_2). Certains de ces ensembles peuvent être vides et on obtient finalement $m' + 1$ ensembles non vides avec $0 \leq m' \leq m$. Soient $m' + 1$ nouvelles variables $x_0, \ldots, x_{m'}$ et une application $\varphi : [0, m'] \to [0, m]$ associant à chaque x_j l'un des ensembles non vides $S_{\varphi(j)}^x$. Il reste alors à choisir $\sigma_i^x : x \mapsto x_0 + \cdots + x_{m'}$.

Prenons maintenant pour σ' la substitution qui, pour chaque variable x, associe à une nouvelle variable x_j la somme des éléments de $S^x_{\varphi(j)}$: $x_j\sigma' = w_1 + \cdots + w_k$ si $S^x_{\varphi(j)} = \{w_1, \cdots, w_k\}$ (ou encore simplement $x_j\sigma' = w_1$ si cet ensemble ne contenait qu'un seul élément).

Il reste finalement à choisir le $u_i \in \mathit{Par}_{\downarrow}(u\sigma_i)$ adéquat tel que $t_1\widehat{+}t_2 =_{AC} u_i\sigma'$ en sélectionnant les u_i ayant, pour chaque x, exactement $\varphi(j)$ occurrences de la variable fraîche x_j à gauche du symbole de tête $\widehat{+}$ et $m' - \varphi(j)$ à droite. C'est-à-dire que pour chaque variable x on peut écrire

$$\begin{aligned} u_i \;=\; & (\underbrace{x_0 + \cdots + x_0}_{\varphi(0)\text{ fois}} + \cdots + \underbrace{x_{m'} + \cdots + x_{m'}}_{\varphi(m')\text{ fois}} + u_1) \\ & \widehat{+}\, (\underbrace{x_0 + \cdots + x_0}_{m-\varphi(0)\text{ fois}} + \cdots + \underbrace{x_{m'} + \cdots + x_{m'}}_{m-\varphi(m')\text{ fois}} + u_2) \end{aligned}$$

Ou encore, de façon plus graphique,



Et comme maintenant, dans un $u_i\sigma'$, un terme donné w d'un $S^x_{\varphi(j)}$ apparaît exactement $\varphi(j)$ fois (autant que dans t_1) on a $u_i\sigma' = t_1\widehat{+}t_2 = t$. □

Exemple 46. (Suite de l'exemple 44)

Les S^x_i sont

$$\begin{aligned} S^x_0 &= \{a_3\}, \\ S^x_1 &= \{a_2, a_4\}, \\ S^x_2 &= \{a_1\}. \end{aligned}$$

Puisque $m_t(x) = 2$, x peut être découpé en x_0, x_1 et x_2. On a alors $u_i = (x_1+x_2+x_2)\widehat{+}(x_0+x_0+x_1+g(y))$ et $t = u_i\sigma'$ avec

$$\sigma' : \left\{ \begin{array}{lcl} x_0 & \mapsto & a_3\ ; \\ x_1 & \mapsto & a_2 + a_4\ ; \\ x_2 & \mapsto & a_1 \end{array} \right\}.$$

Deuxième méthode

Nous présentons à présent une version modifiée de la fonction *Pairs* proposant un ensemble de paires plus faciles à orienter (ce qui est toujours important dès qu'on cherche un ordre adéquat) mais au prix d'un surcoût combinatoire sensible. La principale différence est en fait que les membres gauches sont à présent en forme normale pour $\xrightarrow{m.e.}$.

La facilité d'orientation est obtenue grâce à la conjonction de la précédence des symboles marqués sur leurs copies non marquées (corollaire 26–1) et de l'unicité du symbole avec marque dans les termes : le membre droit ainsi modifié devient « plus petit ».

Le revers de la médaille de cette décroissance est le nombre important des paires modifiées : le calcul de l'ensemble de ces paires requiert le parcours de tous les partages de part et d'autre du symbole marqué des membres droits.

Définition 90. — Soit R un système de récriture, la fonction *ModifiedPairs* est définie comme :

$$\mathit{ModifiedPairs}(R) = \bigcup_{\substack{\langle u, v\rangle \in DP(R),\\ \sigma \in \Sigma_u}} \left\{ \langle u', v'\rangle \text{ telle que } \begin{array}{l} u' \in \mathit{Par}_{\downarrow}(\widehat{u\sigma}), \\ v' \in \mathit{Par}_{\downarrow}(\widehat{v\sigma}) \end{array} \right\}.$$

Proposition 16.
ModifiedPairs(R) est un ensemble de paires de dépendance de R.

PREUVE. Immédiate puisque les membres droits des paires modifiées sont obtenus en effaçant les marques des membres droits des paires issues de la première méthode. □

Chapitre IV

Approche modulaire de la terminaison AC

Nous allons à présent combiner la puissance des paires de dépendance étendues AC et l'analyse modulaire pour former les paires de dépendance relatives AC, illustrant ainsi la généralité de l'approche par modules.

Les étapes AC interagissant peu avec la structure modulaire des systèmes, les preuves et définitions sont proches de celles du cas standard. Nous détaillerons donc principalement les différences introduites par les équations AC.

Afin de conserver les propriétés cruciales, notamment sur la terminaison $\mathcal{C}_{\mathcal{E}}$ (dont la définition ne change pas modulo AC) nous restreignons de nouveau (cf. paragraphe I.2.6) notre étude aux systèmes à branchement fini.

IV.1 Modules AC

Nous étendons la notion de modules à la récriture étendue AC sur termes aplatis. Les signatures des modules AC peuvent contenir des symboles associatifs et commutatifs ; la relation définie par les règles est étendue.

Définition 91. — Soit R_1 un système de récriture AC sur une signature $\mathcal{F}_1$. Un *module étendant* R_1 est un couple $[\mathcal{F}_2 \mid R_2]$ tel que :

1. $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$ (signatures disjointes) ;
2. R_2 est un système de récriture AC sur $\mathcal{F}_1 \cup \mathcal{F}_2$;
3. Pour toute règle $l \to r$ de R_2, $\Lambda(l) \in \mathcal{F}_2$.

Remarque 21. — Les permutations induites par le caractère AC de certains symboles ne concernent en fait qu'un seul symbole à la fois. Les étapes AC ne peuvent en aucun cas permuter des signatures au sein d'un terme, elles n'introduisent aucune dépendance entre les modules.

IV.1.1 Paires de dépendance relatives AC

Les paires de dépendance non marquées sont en fait les paires étendues relatives à la signature du module concerné.

Définition 92. — Considérons l'extension hiérarchique $R_1(\mathcal{F}_1) \longleftarrow [\mathcal{F}_2 \mid R_2]$.

L'*ensemble des paires de dépendance relatives* AC de $[\mathcal{F}_2 \mid R_2]$ est le sous-ensemble des paires étendues AC de $R_1 \cup R_2$ dont les symboles définis sont limités à ceux de $\mathcal{F}_2$.

La similarité de ces définitions avec le cas standard autorise la confusion des dénominations : la présence de symboles associatifs et commutatifs suffit en effet à lever l'ambiguïté.

IV.2 Preuves incrémentales pour la récriture AC

La remarque 21 va nous permettre d'énoncer et de démontrer des critères tout à fait comparables à ceux du chapitre IV de la première partie.

En particulier puisque les étapes AC n'autorisent pas la permutation de symboles de signatures différentes, les symboles marqués ne peuvent en aucun cas intervenir dans la partie instanciée des paires.

Nous pouvons ainsi proposer une version AC du lemme 8.

Lemme 19.

Soient S_1 et S_2 deux systèmes de récriture AC sur une signature $\mathcal{F}_1$. Soit S_3 un TRS AC sur $\mathcal{F}_1 \cup \mathcal{F}_2$ tel que :

- $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$;
- Pour toute règle $l \to r \in S_3$, $\Lambda(l) \in \mathcal{F}_2$.

Alors d'une chaîne infinie minimale de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup S_3$, il est possible d'obtenir une chaîne infinie de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup \pi$ comportant la même séquence de paires mais munie d'une nouvelle substitution ainsi que de nouveaux pas de récriture.

PREUVE. Nous utilisons de nouveau une interprétation, cette fois concernant les termes AC aplatis.

Définition 93. — Posons $S = S_1 \cup S_2 \cup S_3$; soit $>$ un ordre arbitraire mais *total* sur $T_\infty(\widehat{\mathcal{F}_1} \cup \{G : 2\} \cup \{\perp : 0\}, X)$.

L'interprétation $I^{\mathrm{AC}}(x) : T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X) \to T_\infty(\widehat{\mathcal{F}_1} \cup \{G : 2\} \cup \{\perp : 0\}, X)$ est définie par :

$$\begin{array}{rcl} I^{\mathrm{AC}}(x) & = & x \text{ si } x \in X, \\ I^{\mathrm{AC}}(f(t_1 \ldots t_n)) & = & \left\{ \begin{array}{l} f(I^{\mathrm{AC}}(t_1) \ldots I^{\mathrm{AC}}(t_n)) \text{ si } f \in \widehat{\mathcal{F}_1}, \\ \mathrm{Comb}(\mathrm{Red}(f(t_1 \ldots t_n))) \text{ si } f \in \mathcal{F}_2, \end{array} \right. \end{array}$$

Où

$$\begin{array}{rcl} \mathrm{Red}(t) & = & \{I^{\mathrm{AC}}(t') / \, t \xrightarrow[S_1 \cup S_2 \cup S_3/\mathrm{AC}]{} t'\}, \\ \mathrm{Comb}(\emptyset) & = & \perp, \\ \mathrm{Comb}(\{a\} \cup E) & = & G(a, \mathrm{Comb}(E)) \text{ où pour tout } e \in E, a < e). \end{array}$$

Remarque 22. — L'ensemble $\mathrm{Red}(t)$ des t' tels que t se réduit par un pas de $S_1 \cup S_2 \cup S_3$ modulo AC en t' est bien fini ; nous prendrons pour chaque classe AC un seul représentant.

L'interprétation d'un terme t commençant par un symbole de $\mathcal{F}_2$ est un peigne constitué des interprétations de représentants canoniques (car aplatis) des classes d'équivalence modulo AC des réduits de t en un pas par la récriture étendue AC sur termes aplatis.

Il est donc possible d'atteindre chacune de ces interprétations à l'aide d'une réduction $\underset{\pi_2}{\longrightarrow}^{\star} \underset{\pi_1}{\longrightarrow}$ adéquate.

Les preuves des lemmes suivants sont semblables à celles de leurs versions standard (cf. chapitre IV, première partie).

Lemme 20.

Pour tout $t \in T(\widehat{\mathcal{F}_1}, X)$ et toute substitution σ,

$$I^{\mathrm{AC}}(t\sigma) = tI^{\mathrm{AC}}(\sigma).$$

Lemme 21.

Pour tous $t_1, \ldots, t_n$ de $T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X)$ et pour tout contexte C sur $\mathcal{F}_1$ à n trous,

$$I^{\mathrm{AC}}(C[t_1, \ldots, t_n]) = C[I^{\mathrm{AC}}(t_1), \ldots, I^{\mathrm{AC}}(t_n)].$$

Lemme 22.

Pour tout terme t fortement normalisable par $S_1 \cup S_2 \cup S_3$, $I^{\mathrm{AC}}(t)$ est un terme fini.

Il reste à vérifier que l'interprétation permet de simuler les pas du système « abstrait ».

Lemme 23.

Pour tous s et t de $T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X)$ et toute règle $l \to r \in S_1 \cup S_2$,

$$\text{Si } s \xrightarrow[l \to r/\mathrm{AC}]{p} t \text{ alors } I^{\mathrm{AC}}(s) \xrightarrow[S_1 \cup S_2 \cup \pi/\mathrm{AC}]{}^{+} I^{\mathrm{AC}}(t).$$

En outre, si $p \neq \Lambda$ et $\Lambda(s) \in \widehat{\mathcal{F}_1}$ alors $I^{\mathrm{AC}}(s) \xrightarrow[S_1 \cup S_2 \cup \pi/\mathrm{AC}]{\neq \Lambda}^{+} I^{\mathrm{AC}}(t)$.

Preuve. Nous savons par la remarque 21 que les étapes AC ne feront jamais permuter de symbole de $\mathcal{F}_1$ avec un symbole de $\mathcal{F}_2$. La preuve du lemme 12 peut donc être reprise ici.

– S'il n'y a que des symboles de $\mathcal{F}_1$ sur le chemin reliant Λ à p on a par application des lemmes 21 puis 20 et des hypothèses : $I^{\mathrm{AC}}(s) \xrightarrow[S_1 \cup S_2/\mathrm{AC}]{} I^{\mathrm{AC}}(t)$;

– S'il existe un plus petit $p' < p$ (pour l'ordre préfixe) tel que $\Lambda(s|_{p'}) \in \mathcal{F}_2$, en supposant que $s = C[s_1, \ldots, s', \ldots, s_n]$ où C est un contexte sur $\mathcal{F}_1$, $p = p'q$ et $s|_{p'} = s'$, on a par le lemme 21 $I^{\mathrm{AC}}(s) = C[I^{\mathrm{AC}}(s_1), \ldots, I^{\mathrm{AC}}(s'), \ldots, I^{\mathrm{AC}}(s_n)]$.

  On déduit alors de la définition de l'interprétation que

$$C[I^{\mathrm{AC}}(s_1), \ldots, I^{\mathrm{AC}}(s'), \ldots, I^{\mathrm{AC}}(s_n)] \xrightarrow[\pi/\mathrm{AC}]{}^{+} I^{\mathrm{AC}}(t).$$

  Le cas échéant un aplatissement du terme finalement obtenu doit être effectué.

□

De la même façon (deuxième cas) on peut montrer la correction de la simulation des pas de S_3.

Lemme 24.
Pour tous s et t de $T(\widehat{\mathcal{F}_1} \cup \mathcal{F}_2, X)$, si $s \xrightarrow[S_3/AC]{p} t$ alors $I(s) \xrightarrow[\pi/AC]{}^{+} I(t)$.
De plus, si $\Lambda(s) \in \widehat{\mathcal{F}_1}$ alors $I(s) \xrightarrow[\pi/AC]{\neq\Lambda}^{+} I(t)$.

Preuve du lemme 19

Soit $\langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots$ une chaîne de dépendance de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup S_3$ munie d'une substitution σ. Soit σ' la substitution telle que pour tout x, $x\sigma' = I(x\sigma)$.

Dans la mesure où la chaîne considérée est minimale, la substitution σ est fortement normalisable, le lemme 22 garantit alors que σ' ne substitue que des termes finis.

Nous allons montrer que $\langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots$ munie de σ' est une chaîne de $[\mathcal{F}_1 \mid S_2]$ sur $S_1 \cup S_2 \cup \pi$.

Pour ce faire, il nous faut prouver que pour tout i,

$$v_i\sigma' \xrightarrow[S_1 \cup S_2 \cup \pi/AC]{\neq\Lambda}^{\star} u_{i+1}\sigma'.$$

Nous savons que

$$v_i\sigma \xrightarrow[S_1 \cup S_2 \cup S_3/AC]{\neq\Lambda}^{\star} \xrightarrow{m.e.}^{\star} u_{i+1}\sigma.$$

Considérons un pas $s \xrightarrow[S_1 \cup S_2 \cup S_3/AC]{p} t$ de cette dérivation. Puisque

$$\Lambda(s) = \Lambda(t) = \Lambda(v_i) = \Lambda(u_{i+1}) \in \widehat{\mathcal{F}_1},$$

Alors du lemme 24 ou, le cas échéant, du lemme 23 nous déduisons

$$I^{AC}(s) \xrightarrow[S_1 \cup S_2 \cup \pi/AC]{\neq\Lambda}^{\star} I^{AC}(t).$$

Nous pouvons reconstituer la séquence en juxtaposant les pas afin d'obtenir

$$I^{AC}(v_i\sigma) \xrightarrow[S_1 \cup S_2 \cup \pi]{\neq\Lambda}^{\star} I^{AC}(u_{i+1}\sigma).$$

Dans la mesure où $I^{AC}(v_i\sigma) = v_i\sigma'$ et $I^{AC}(u_{i+1}\sigma) = u_{i+1}\sigma'$ le lemme 20 nous permet de conclure.

□

À l'aide du lemme 19 nous pouvons démontrer les résultats suivants concernant respectivement les cas d'extensions par un module puis plusieurs modules indépendants. La remarque 21 permet en effet de se ramener aux trois cas de chaînes évoqués au cours des preuves des théorèmes 18 et 19.

Théorème 27.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$ où $\mathcal{F}_1 \cup \mathcal{F}_2$ est une signature AC.

1. Si R_1 est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC,
2. S'il n'existe pas de chaîne infinie de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2/\mathrm{AC}$,

Alors $R_1 \cup R_2$ est fortement normalisant modulo AC.

Théorème 28.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$, soit $[\mathcal{F}_3 \mid R_3]$ un module étendant R_1 indépendamment de R_2.

1. Si $R_1 \cup R_2$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC,
2. S'il n'existe pas de chaîne infinie de $[\mathcal{F}_3 \mid R_2]$ sur $R_1 \cup R_3 \cup \pi/\mathrm{AC}$,

Alors $R_1 \cup R_2 \cup R_3$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC.

En particulier nous pouvons à présent étendre le résultat de Kurihara & Ohuchi [52] à la récriture modulo associativité et commutativité.

Corollaire 28–1.
La terminaison $\mathcal{C}_\mathcal{E}$ AC est modulaire pour les unions de systèmes AC composables.

Corollaires

Ces théorèmes entraînent des critères de terminaison susceptibles d'être mis en application grâce à l'utilisation d'ordres π-extensibles pourvu que ceux-ci soient compatibles AC.

Corollaire 27–1.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC,
2. S'il n'existe pas de chaîne infinie de $[\mathcal{F}_2 \mid R_2]$ sur $R_1 \cup R_2 \cup \pi/\mathrm{AC}$,

Alors $R_1 \cup R_2$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC.

Corollaire 27–2.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$.

1. Si R_1 est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC ;
2. S'il existe un ordre de réduction faible (resp. faiblement π-extensible) compatible AC $(\succeq, \succ)$ tel que :
   - $R_1 \cup R_2 \subseteq \succeq$ et
   - $\mathrm{DP}([\mathcal{F}_2 \mid R_2]) \subseteq \succ$, pour un ensemble de paires relatives AC,

Alors $R_1 \cup R_2$ est fortement normalisant (resp. $\mathcal{C}_\mathcal{E}$ fortement normalisant) modulo AC.

Corollaire 28–2.
Soit $[\mathcal{F}_1 \mid R_1] \longleftarrow [\mathcal{F}_2 \mid R_2]$ une extension hiérarchique de $R_1(\mathcal{F}_1)$, soit $[\mathcal{F}_3 \mid R_3]$ un module étendant R_1 indépendamment de R_2.

1. Si $R_1 \cup R_2$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC ;
2. S'il existe un ordre faiblement π-extensible compatible AC $(\succeq, \succ)$ tel que :
   - $R_1 \cup R_3 \subseteq \succeq$,
   - $\mathrm{DP}([\mathcal{F}_3 \mid R_3]) \subseteq \succ$, pour un ensemble de paires relatives AC marquées,

Alors $R_1 \cup R_2 \cup R_3$ est $\mathcal{C}_\mathcal{E}$ fortement normalisant modulo AC.

Troisième partie

Implantation

Chapitre I

Introduction

Il est illusoire, puisqu'elle est présente dans de nombreux domaines, d'espérer de chacun des utilisateurs de la récriture qu'il soit un spécialiste de la preuve de terminaison des systèmes. La normalisation forte est pourtant une propriété fondamentale, que ce soit comme garantie de totalité d'une fonction ou encore comme caractéristique préliminaire et sa mise en évidence, tâche toujours ardue, doit donc pouvoir être effectuée par toute personne ayant à spécifier, programmer, prouver, au sens large : manipuler des systèmes de récriture.

On peut trouver de nombreuses implantations de méthodes de preuve de terminaison. REVE [10], ORME [11], COMTES [45], POLO [32], REVEAL [1], REDUX [9], RRL [48], TERMINATIONLAB [66], et C*i*ME2 [13] accompagné de son résolveur de contraintes efficace figurent parmi les principales. Cependant, si une certaine forme d'incrémentalité existait au sein des preuves basées uniquement sur des précédences, aucun n'autorisait jusqu'à présent de preuves réellement modulaires.

C'est avec le double souci : d'une part, de rendre la preuve de terminaison accessible aux non-spécialistes et, d'autre part, de permettre à une machine d'effectuer ce travail sur des systèmes importants que nous considérons le développement d'un outil automatique d'aide à la preuve de terminaison. L'évolution du système C*i*ME vers le système C*i*ME2 s'est déroulée dans cet état d'esprit : fournir un outil permettant de prouver la terminaison du plus grand nombre de systèmes *issus de la pratique* plutôt que certains cas pathologiques.

Nous allons dans un premier temps décrire le système C*i*ME2 et plus particulièrement sa boîte à outils de terminaison. Nous présenterons alors l'implantation au sein de ce programme des critères développés au cours de cette thèse, c'est-à-dire permettant une approche incrémentale de la preuve et ainsi une application à la preuve sur de gros systèmes, et autorisant un traitement performant des systèmes dont certains symboles sont associatifs et commutatifs.

Chapitre II

Un outil de preuves : C*i*ME2

À l'origine destiné à la complétion modulo une théorie équationnelle[1], C*i*ME, développé par l'équipe DÉMONS du L.R.I. à Orsay [12, 13], a maintenant évolué en un outil généraliste de traitement de la récriture. Son implantation est réalisée en OCAML, un langage de la famille ML développé et distribué par l'I.N.R.I.A. [67].

II.1 Présentation générale

La seconde version du système C*i*ME, C*i*ME2, fournit un langage fortement typé et un environnement de programmation équipé d'un puissant résolveur de contraintes diophantiennes sur des domaines finis et spécialisé dans la manipulation de systèmes de récriture de termes, mais aussi de mots. Dans « manipulation », nous incluons le calcul et la normalisation à l'aide de règles, mais aussi la complétion (Knuth-Bendix) des systèmes, ainsi que des outils de preuve de terminaison.

Sans vouloir reprendre la description complète qu'en ont fait les auteurs [13], nous présentons brièvement les caractéristiques de C*i*ME2 utiles à notre propos. L'utilisateur définit dans un fichier ou directement dans un interpréteur des fonctions, des signatures, des termes, des systèmes ou encore des ordres. Il est possible d'abstraire des valeurs (construction `let`), de définir des fonctions d'ordre supérieur et de manipuler des applications partielles (syntaxe curryfiée).

Exemple 47.

Il est possible d'abstraire des valeurs :

```
CiME> let x = 4*5;
x : int = 20
```

On peut également définir des fonctions récursivement :

```
CiME> let fun fact n = if n <= 1 then 1 else n * (fact (n-1));
fact : int -> int = <fun>
CiME> fact 7;
- : int = 5040
CiME> fact 100;
- : int =
93326215443944152681699238856266700490715968264381621468592963895
21759999322991560894146397615651828625369792082722375825118521
```

[1] C*i*ME est en fait l'acronyme de *C*omplét*i*on *M*odulo *E*.

```
9168640000000000000000000000
```

Le solveur intégré traite les contraintes d'égalité ou d'inégalité entre fonctions polynomiales à un nombre arbitraire de variables. Les fonctions prédéfinies permettent, sur l'entrée d'une borne B de tenter une résolution d'un ensemble de contraintes pour des valeurs entières des variables comprises entre 0 et B.

Exemple 48.

La première phase consiste à définir les contraintes :

```
CiME> let constr = dioph_constraint
  "x^2 + y^2 = z^2; x > 0; y > 0; z > 0";
constr : dioph_constraint = { -1*z^2 + 1*y^2 + 1*x^2 = 0;
                              1*x + -1 >= 0;
                              1*y + -1 >= 0;
                              1*z + -1 >= 0;
                              }
(4 inequalitie(s) over 3 variable(s).)
```

Il suffit ensuite de demander au système de les résoudre en précisant une borne (ici 100).

```
CiME> dioph_solve 100 constr;
Solution :
x = 4
y = 3
z = 5
- : unit = ()
```

Une boîte à outils de terminaison comportant différents critères, dont ceux par paires de dépendance, autorise enfin la recherche automatique de preuves de terminaison, essentiellement par recherche d'une interprétation polynomiale convenable. Cette dernière fonctionnalité de C*i*ME2 est en particulier utilisée par le système TALP [71] pour l'analyse de la terminaison de programmes logiques.

II.2 La terminaison dans C*i*ME2

II.2.1 Principes généraux

Un système de récriture est défini dans C*i*ME2 à l'aide de la signature et de l'ensemble de variables sur lesquels il est construit et de l'ensemble de règles qui le composent.

Exemple 49.

On commence par définir une signature :

```
CiME> let F_peano = signature "
0 : constant;
s : unary;
+,* : infix binary;
";
F_peano : signature = <signature>
```

Puis un ensemble de variables :

```
CiME> let X = vars "x y z";
X : variable_set = <variable set>
```

On peut alors construire un système de récriture.

```
CiME> let R_peano = TRS F_peano X "
x+0 -> x;
x+s(y) -> s(x+y);
x*0 -> 0;
x*s(y) -> (x*y)+x;
";
R_peano : (F_peano,X) TRS = { x + 0 -> x,
                              x + s(y) -> s(x + y),
                              x * 0 -> 0,
                              x * s(y) -> (x * y) + x } (4 rules)
```

Une preuve de terminaison peut alors être recherchée sur ce système en évaluant l'expression.

`termination` *nom* ;

C*i*ME2 détermine alors des contraintes d'ordre en fonction des critères que l'utilisateur souhaite voir utilisés et cherche un ordre les satisfaisant. Les critères disponibles sont :

1. `standard`, la décroissance stricte de toutes les règles est alors exigée ;
2. `dp`, le critère des paires de dépendance est employé ;
3. `marks`, les paires de dépendance utilisent les symboles marqués ;
4. `nomarks`, les paires de dépendance sont sans marques ;
5. `graph`, les contraintes sont affinées par une analyse du graphe de dépendance ;
6. `nograph`, le graphe de dépendance n'est pas pris en compte.

Les ordres recherchés sont pour l'instant à base d'interprétations polynomiales. Celles-ci étant dans la plupart des cas d'autant plus simples que le critère choisi est puissant et la recherche aboutissant à une énumération de valeurs possibles pour les coefficients, il peut s'avérer utile de limiter la recherche à certaines catégories de fonctions polynomiales. C*i*ME2 offre trois sortes d'interprétations reprenant la terminologie de Steinbach :

1. `linear`, tous les monômes sont de degré au plus 1 ;
2. `simple`, le degré de chaque indéterminée est au plus 1 dans chaque monôme ;
3. `simple-mixed`, comme dans le cas `simple`, le degré de chaque indéterminée est au plus 1 dans chaque monôme, sauf dans les cas d'interprétations de symboles unaires qui prennent la forme de fonctions polynomiales du second degré.

II.2.2 Comparaisons de polynômes

La comparaison de deux fonctions polynomiales sur les entiers est un problème indécidable [59]. Il existe toutefois plusieurs méthodes incomplètes [8,31,43,81] pour vérifier qu'une fonction polynomiale P_μ est à valeurs positives sur un domaine $[\mu, +\infty[$.

La méthode proposée par Ben Cherifa et Lescanne [8] consiste à choisir $\mu = 2$ et transformer la fonction P en une fonction polynomiale P' telle que $P \geq P'$ est imposé par la transformation ; P' (et par transitivité P) est positive si tous ses coefficients sont positifs.

Giesl [31] analyse quant à lui les dérivées partielles de P : si $P(\mu, \ldots, \mu) \geq_{\mathbb{N}} 0$ et si soit tous ses coefficients sont nuls, soit toutes ses dérivées partielles $\frac{\partial P(x_1, \ldots, x_n)}{\partial x_i}$, $1 \leq i \leq n$ sont, récursivement, positives alors P est positive.

L'approche de Hong et Jakus [43] est d'effectuer une translation de D_μ vers D_0 (cf. paragraphe III.5.2 des préliminaires), on vérifie alors que tous les coefficients de P_0 sont positifs.

Hong et Jakus ont en fait montré que leur approche et celle de Giesl permettent de prouver la positivité des mêmes fonctions, la complexité de ces tests étant, en outre, équivalente.

Le test de positivité utilisé dans C*i*ME2 est comparable à celui de Hong et Jakus. Toutefois, plutôt qu'effectuer une translation de D_μ vers D_0 à chaque comparaison, le système calcule directement et une fois pour toutes les interprétations de chaque symbole dans D_0.

II.2.3 Recherche de preuve

Les deux principales étapes de la recherche de preuve sont, premièrement, la génération de contraintes d'ordres en fonction des critères choisis et ensuite la tentative de résolution de ces contraintes.

L'approche décidée par l'utilisateur, par paires de dépendance ou encore simplement standard, induit un certain nombre de contraintes que l'ordre doit respecter. Dans le cas d'une recherche d'interprétation polynomiale, le système traduit ces contraintes d'ordre en inégalités sur des polynômes de la forme désirée (cf. paragraphe II.2.1). Les conditions de positivité sont alors calculées de façon purement algébrique. Il en résulte des contraintes polynomiales sur les coefficients. Celles-ci sont donc pseudo-linéarisées[2] à l'aide de techniques de chaînes d'additions avant que C*i*ME2 ne tente de les résoudre sur un domaine fini (dont les bornes ont été auparavant précisées).

Exemple 50.

Pour le système d'arithmétique de Peàno qu'on vient de définir, si on souhaite par exemple une preuve à l'aide du critère standard (sans paires de dépendance) avec une interprétation simple (au sens de Steinbach) et en fixant la borne des coefficients à 6 (valeur par défaut), le système répondra en fournissant une interprétation convenable :

```
CiME> termination R_peano;
Entering the termination expert. Verbose level = 0
[0] = 1;
[s](X0) = 1*X0 + 1;
[+](X0,X1) = 2*X1 + 1*X0;
[*](X0,X1) = 2*X0*X1 + 2*X1 + 1*X0;
- : unit = ()
```

[2]Le degré maximal est alors 2.

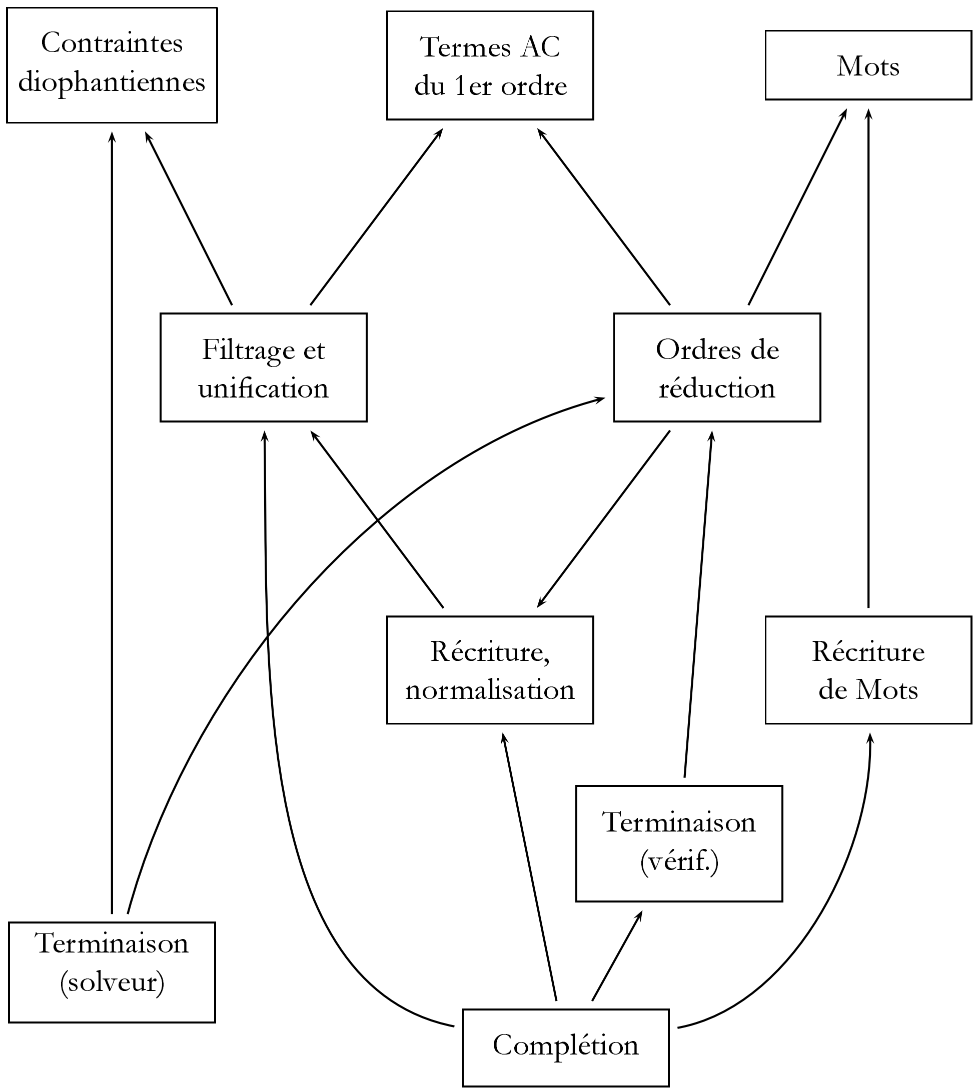


FIG. II.1. Architecture modulaire de C*i*ME2.

Chapitre III

Modules

L'adjonction de capacités de preuve incrémentale à un outil permet d'envisager un passage à l'échelle sur des systèmes importants résultant par exemple d'autres applications, ou encore faisant partie d'un vaste projet.

III.1 Problématique

L'extension d'un langage de programmation spécialisé, comme C*i*ME2, à la manipulation de systèmes définis hiérarchiquement ne va pas sans poser plusieurs problèmes, tant au niveau de la spécification qu'en ce qui concerne l'implantation proprement dite.

En premier lieu il faut être capable de fournir à l'utilisateur une structure de donnée adaptée au nouveau côté incrémental des définitions et des preuves, c'est-à-dire qui reste évolutive en conservant toutes les informations nécessaires. C'est le cas des systèmes hiérarchiques.

Remarque 23. — Les signatures sur lesquelles nous allons travailler vont être également amenées à évoluer. En effet : une extension d'un système par un module donne un nouveau système sur une signature augmentée. Le type abstrait des signatures a donc à prendre en compte cette situation sans que cette modification soit spécifiée par l'utilisateur. C'est donc un problème d'implantation du système et non de définition de l'extension du langage.

L'utilisateur, enfin, doit pouvoir mener sa preuve de terminaison de trois façons différentes :

1. Au fur et à mesure de l'ajout de règles, le système doit alors pouvoir se reposer sur ce qui a déjà été prouvé $\mathcal{C}_\mathcal{E}$ fortement normalisant ;

2. Sur un système entier, mais en respectant une hiérarchie décidée et fournie par ses soins ;

3. Enfin sur un système entier mais en suivant la hiérarchie issue du découpage en modules minimaux du système.

De nouvelles directives doivent alors lui permettre de faire ses choix à ce niveau.

III.2 Extension du langage et implantation

III.2.1 Structures de données du langage

Hiérarchies

Définissons les modules simplement : ils contiennent de nouveaux symboles, de nouvelles règles et caractérisent la hiérarchie de la structure dont ils font partie par l'ensemble des modules qu'ils étendent.

Nous choisissons de regrouper toute cette information dans la définition d'une hiérarchie de modules (formant un système) et introduisons à cet effet le type `HTRS`. Cette hiérarchie définissant un système on doit pouvoir préciser pour chaque module :

– L'ensemble de variables sur lequel il est formé ;

– L'ensemble des nouveaux symboles qui le caractérisent ;

– L'ensemble des nouvelles règles introduites ;

– Enfin l'ensemble des hiérarchies dont il dépend.

Une instance du type abstrait des hiérarchies de modules doit donc pouvoir être définie par une opération de construction de la forme :

`let` *nom* = `HTRS` {*dépendances*} *nouveaux_symboles* *variables* *règles* ;

Fonctions et directives de modularité

Il reste à lancer sur de telles hiérarchies la recherche de preuve selon la stratégie désirée.

Lancer une preuve respectant une hiérarchie donnée devrait être immédiat puisque le `HTRS` soumis décrit déjà une hiérarchie. On peut ainsi mener une preuve incrémentalement par

1. Définition d'une extension du `HTRS` puis,
2. Preuve de la hiérarchie décrite par icelui.

Les résultats connus des preuves précédentes — dans le temps et la hiérarchie — permettant alors de limiter la nouvelle recherche au module fourni et à son insertion dans le schéma hiérarchique ce qui est bien le déroulement caractéristique d'une preuve incrémentale.

La décomposition d'un `HTRS` en une hiérarchie de modules *minimaux* ne change rien d'autre quant à la stratégie de preuve que la hiérarchie sur laquelle elle est menée. C'est une preuve incrémentale sur le système proposé mais cette fois selon une hiérarchie de module minimaux. Ce choix par l'utilisateur n'est donc pas justifié au niveau de la fonction de recherche de preuve, il est davantage une précision globale de la hiérarchie à suivre lors des preuves.

Les deux premiers cas de stratégie de preuve peuvent ainsi être résumés en une seule fonction. Il suffit alors de lancer la recherche par une commande du type :

`m_termination` *nom* ;

Qui lance la recherche de preuve de terminaison du système *nom* selon la hiérarchie qu'il propose. Le choix de la décomposition minimale est alors précisé comme un critère :

`termcrit "minimal"`;

Dès lors que cette expression a été interprétée, `m_termination` *nom* ; provoque une recherche selon la hiérarchie non plus décrite par *nom* mais celle issue de sa décomposition en modules minimaux.

III.2.2 Choix d'implantation

Signatures et hiérarchies

Une hiérarchie est en fait déterminée dès qu'on sait dire de quelles hiérarchies dépend un module donné. Les opérations de construction, de ce type récursif (désigné ci-après `htrs`) sont donc réduites à l'opération d'extension qui, sur la donnée de hiérarchies $H_1, \ldots, H_n$ (vides le cas échéant) pour les dépendances et d'une signature, d'un ensemble de variables et d'un ensemble de nouvelles règles pour le module extenseur M, retourne la hiérarchie formée de l'extension hiérarchique des H_i par M.

Déroulement de la preuve

La détermination des contraintes de terminaison ne pose pas de problème particulier quant à sa réalisation. Deux fonctions retournent une conjonction de contraintes sur l'entrée d'un `htrs`, respectivement pour les critères sans et avec symboles marqués.

Il reste à mener la preuve en suivant la hiérarchie proposée. On va ainsi chercher pour chaque module une preuve de terminaison du sous-système qu'il définit par extension. Si une preuve est trouvée alors la hiérarchie correspondante est marquée comme terminant $\mathcal{C}_{\mathcal{E}}$. Si enfin le module concerné étend une hiérarchie qui n'a pas encore été montrée fortement normalisante, on commence par chercher une preuve de terminaison de celle-ci.

La terminaison est prouvée quand tous les modules définissent des systèmes hiérarchiques fortement normalisants.

Décomposition en modules minimaux

Comme nous l'avons précisé lors de la section II.2 de la première partie, il est toujours possible de présenter un système de récriture comme une hiérarchie de modules minimaux, où *minimaux* signifie qu'ils ne peuvent eux-mêmes être décomposés. Si ces modules ne peuvent être décomposés, c'est en particulier que chaque symbole introduit a besoin de toutes les règles et de tous les autres symboles définissant le module.

Pour un système $R(\mathcal{F})$, on définit un graphe $\mathcal{G}$ (cf. section II.2, première partie) ayant pour nœuds les symboles et comportant un arc de f vers g s'il existe une règle $l \to r \in R$ telle que $f = \Lambda(l)$ et il existe une occurrence de g dans l ou r. Si on décrit les dépendances entre les symboles à l'aide d'un tel graphe $\mathcal{G}$, les signatures des modules minimaux correspondent alors aux composantes fortement connexes de $\mathcal{G}$ (aux parties fortement connexes de taille maximale).

La construction automatique d'un tel graphe sur les symboles est aisée. Il reste à déterminer efficacement[1] les composantes fortement connexes donnant les signatures de modules.

Si on adjoint alors à chaque symbole de ces composantes toutes les règles dont il est le symbole de tête du membre gauche, on définit de fait les modules. Leur hiérarchie est alors déterminée en fonction de celle des symboles. Il suffit enfin de composer les fonctions :

– De construction du graphe à partir d'une signature et d'un `htrs` (sur cette signature) puis,

– De construction de `htrs` à partir de modules.

On obtient alors la liste des systèmes hiérarchiques constituant le système de départ.

[1]Nous utilisons pour cette tâche l'algorithme de Tarjan [82].

III.3 Exemple d'exécution

Une preuve incrémentale

Reprenons l'exemple complet de la section IV.3 en ajoutant une règle d'associativité pour l'addition. Ce système de calcul du logarithme est maintenant overlapping, les critères utilisant la terminaison innermost ne s'appliquent donc pas.

Définissons la hiérarchie. Tout d'abord les variables puis la signature des entiers avec la règle de simplification.

```
CiME> let my_vars = vars "x y z";
CiME> let sig_nat = signature
  "# : constant ; 0,1 : postfix unary ;";

CiME> let module_nat = HTRS {} sig_nat my_vars "(#)0 -> #;";
```

La preuve de terminaison $\mathcal{C}_\mathcal{E}$ de ce système sans paires et donc au graphe de dépendance vide est aisée. Sur la commande :

```
CiME> m_termination module_nat;
```

Le système renvoie :

```
Entering the termination expert for modules. Verbose level = 0
checking each of the 0 strongly connected components :
Termination proof found.
- : unit = ()
```

Nous pouvons maintenant introduire l'arithmétique.

```
CiME> let sig_plus = signature "+ : infix binary;";

CiME> let module_plus =
  HTRS {module_nat} sig_plus my_vars
    " x + # -> x ;
      # + x -> x;
      (x)0 + (y)0 -> (x + y)0;
      (x)0 + (y)1 -> (x + y)1;
      (x)1 + (y)0 -> (x + y)1;
      (x)1 + (y)1 -> ((x + y) + (#)1)0;
      x + (y + z) -> (x + y) + z;";
```

La hiérarchie ainsi définie termine $\mathcal{C}_\mathcal{E}$.

```
CiME> m_termination module_plus;
Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ (#)0 -> # } (1 rules)
is already proven.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
```

```
[#] = 0;
[+](X0,X1) = X1 + X0 + 1;
[0](X0) = X0 + 1;
[1](X0) = X0 + 2;
['+‘](X0,X1) = 2*X1 + X0;

Termination proof found.
- : unit = ()
```

Le système s'est effectivement servi de la terminaison $\mathcal{C}_\mathcal{E}$ de `module_nat`.

```
CiME> let sig_minus = signature "- : infix binary;";

CiME> let module_minus = HTRS {module_nat} sig_minus my_vars
    " x - # -> x;
      # - x -> #;
      (x)0 - (y)0 -> (x - y)0;
      (x)0 - (y)1 -> ((x - y) - (#)1)1;
      (x)1 - (y)0 -> (x - y)1;
      (x)1 - (y)1 -> (x - y)0;";
```

Lors de la preuve de terminaison de la hiérarchie induite par `module_minus` les règles d'addition n'interviennent pas.

```
CiME> m_termination module_minus;
Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ (#)0 -> # } (1 rules)
is already proven.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[-](X0,X1) = X0;
[0](X0) = X0 + 1;
[1](X0) = X0 + 1;
['-‘](X0,X1) = X1*X0;

Termination proof found.
- : unit = ()
```

Introduisons les booléens.

```
CiME> let sig_bool =  signature
                 "true, false : constant; not : unary; if : 3;";

CiME> let module_bool = HTRS {} sig_bool my_vars
  " not(false) -> true ;
    not(true) -> false ;
```

```
    if(true,x,y) -> x ;
    if(false,x,y) -> y;";
```

Ici aussi le graphe est vide.

```
CiME> m_termination module_bool;
Entering the termination expert for modules. Verbose level = 0
checking each of the 0 strongly connected components :
Termination proof found.
- : unit = ()
```

Le résultat de cette preuve sera utilisé pour montrer la terminaison $\mathcal{C}_\mathcal{E}$ des règles de comparaison.

```
CiME> let sig_ge =  signature "ge: binary;";

CiME> let module_ge =
  HTRS {module_bool;module_nat} sig_ge my_vars
    " ge((x)0,(y)0) -> ge(x,y);
      ge((x)0,(y)1) -> not(ge(y,x));
      ge((x)1,(y)0) -> ge(x,y);
      ge((x)1,(y)1) -> ge(x,y);
      ge(x,#)   -> true;
      ge(#,(x)1)  -> false;
      ge(#,(x)0)  -> ge(#,x);";
```

Remarquons que le système ne prend pas en compte les règles d'arithmétique dans cette preuve.

```
CiME> m_termination module_ge;
Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ not(false) -> true,
  not(true) -> false,
  if(true,V_0,V_1) -> V_0,
  if(false,V_0,V_1) -> V_1 } (4 rules)
is already proven.

CE-termination of
{ (#)0 -> # } (1 rules)
is already proven.

checking each of the 2 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[true] = 0;
[ge](X0,X1) = 0;
[0](X0) = X0 + 1;
[false] = 0;
[1](X0) = 0;
[not](X0) = 0;
```

```
[if](X0,X1,X2) = X2 + X1;
['ge`](X0,X1) = X1;

checking component 2 (disjunction of 1 constraints)
[#] = 0;
[true] = 0;
[ge](X0,X1) = 0;
[0](X0) = X0 + 1;
[false] = 0;
[1](X0) = X0 + 1;
[not](X0) = 0;
[if](X0,X1,X2) = X2 + X1;
['ge`](X0,X1) = X1*X0;

Termination proof found.
- : unit = ()
```

Il y a bien une interprétation satisfaisante pour chaque partie strictement fortement connexe du graphe de dépendance.

```
CiME> let sig_log' =  signature "log' : unary;";

CiME> let module_log' = HTRS {module_ge; module_plus} sig_log' my_vars
    " log'(#) -> #;
      log'((x)0) -> if(ge(x,(#)1),log'(x) + (#)1,#);
      log'((x)1) -> log'(x) + (#)1;";

CiME> let sig_log =  signature "log : unary;";

CiME> let module_log =
  HTRS {module_log';module_minus} sig_log my_vars
    " log(x) -> log'(x) - (#)1;";
```

Pour Log' nous obtenons :

```
CiME> m_termination module_log';
Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ (#)0 -> #,
  not(false) -> true,
  not(true) -> false,
  if(true,V_0,V_1) -> V_0,
  if(false,V_0,V_1) -> V_1,
  ge((V_0)0,(V_1)0) -> ge(V_0,V_1),
  ge((V_0)0,(V_1)1) -> not(ge(V_1,V_0)),
  ge((V_0)1,(V_1)0) -> ge(V_0,V_1),
  ge((V_0)1,(V_1)1) -> ge(V_0,V_1),
```

```
  ge(V_0,#) -> true,
  ge(#,(V_0)1) -> false,
  ge(#,(V_0)0) -> ge(#,V_0) } (12 rules)
is already proven.

CE-termination of
{ (#)0 -> #,
  V_0 + # -> V_0,
  # + V_0 -> V_0,
  (V_0)0 + (V_1)0 -> (V_0 + V_1)0,
  (V_0)0 + (V_1)1 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)0 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)1 -> ((V_0 + V_1) + (#)1)0,
  V_0 + (V_1 + V_2) -> (V_0 + V_1) + V_2 } (8 rules)
is already proven.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[+](X0,X1) = X1 + X0;
[true] = 0;
[ge](X0,X1) = X1;
[log'](X0) = X0;
[0](X0) = X0 + 1;
[false] = 1;
[1](X0) = X0 + 1;
[not](X0) = 1;
[if](X0,X1,X2) = X2*X0 + X1;
['log'`](X0) = X0;

Termination proof found.
- : unit = ()
```

Quant au module du logarithme proprement dit, la preuve est immédiate puisque le graphe est vide.

```
CiME> m_termination module_log;
Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ V_0 + # -> V_0,
  # + V_0 -> V_0,
  (V_0)0 + (V_1)0 -> (V_0 + V_1)0,
  (V_0)0 + (V_1)1 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)0 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)1 -> ((V_0 + V_1) + (#)1)0,
  V_0 + (V_1 + V_2) -> (V_0 + V_1) + V_2,
  (#)0 -> #,
```

```
  not(false) -> true,
  not(true) -> false,
  if(true,V_0,V_1) -> V_0,
  if(false,V_0,V_1) -> V_1,
  ge((V_0)0,(V_1)0) -> ge(V_0,V_1),
  ge((V_0)0,(V_1)1) -> not(ge(V_1,V_0)),
  ge((V_0)1,(V_1)0) -> ge(V_0,V_1),
  ge((V_0)1,(V_1)1) -> ge(V_0,V_1),
  ge(V_0,#) -> true,
  ge(#,(V_0)1) -> false,
  ge(#,(V_0)0) -> ge(#,V_0),
  log'(#) -> #,
  log'((V_0)0) -> if(ge(V_0,(#)1),log'(V_0) + (#)1,#),
  log'((V_0)1) -> log'(V_0) + (#)1 } (22 rules)
is already proven.

CE-termination of
{ (#)0 -> #,
  V_0 - # -> V_0,
  # - V_0 -> #,
  (V_0)0 - (V_1)0 -> (V_0 - V_1)0,
  (V_0)0 - (V_1)1 -> ((V_0 - V_1) - (#)1)1,
  (V_0)1 - (V_1)0 -> (V_0 - V_1)1,
  (V_0)1 - (V_1)1 -> (V_0 - V_1)0 } (7 rules)
is already proven.
checking each of the 0 strongly connected components :
Termination proof found.
- : unit = ()
```

Une preuve modulaire

Si maintenant nous souhaitons une preuve modulaire d'un « gros » système calculant le logarithme, nous définissons comme auparavant un ensemble de variables pertinent,

```
CiME> let my_vars = vars "x y z";
```

La signature complète du système est :

```
CiME> let sig_whole = signature
" log : unary;
  log' : unary;
  ge: binary;
  true, false : constant; not : unary; if : 3;
  - : infix binary;
  + : infix binary;
  # : constant ; 0,1 : postfix unary ;";
```

Il reste à définir le système.

```
CiME> let whole = HTRS {} sig_whole my_vars "
 (#)0 -> #;
 x + # -> x ;
 # + x -> x;
 (x)0 + (y)0 -> (x + y)0;
 (x )0 + (y)1 -> (x + y)1;
 (x)1 + (y)0 -> (x + y)1;
 (x)1 + (y)1 -> ((x + y) + (#)1)0;
 x + (y + z) -> (x + y) + z;
 x - # -> x;
 # - x -> #;
 (x)0 - (y)0 -> (x - y)0;
 (x)0 - (y)1 -> ((x - y) - (#)1)1;
 (x)1 - (y)0 -> (x - y)1;
 (x)1 - (y)1 -> (x - y)0;
 not(false) -> true ;
 not(true) -> false ;
 if(true,x,y) -> x ;
 if(false,x,y) -> y;
 ge((x)0,(y)0) -> ge(x,y);
 ge((x)0,(y)1) -> not(ge(y,x));
 ge((x)1,(y)0) -> ge(x,y);
 ge((x)1,(y)1) -> ge(x,y);
 ge(x,#) -> true;
 ge(#,(x)1) -> false;
 ge(#,(x)0) -> ge(#,x);
 log'(#) -> #;
 log'((x)0) -> if(ge(x,(#)1),log'(x) + (#)1,#);
 log'((x)1) -> log'(x) + (#)1;
 log(x) -> log'(x) - (#)1;";
```

Si nous choisissons une décomposition en modules minimaux et une preuve utilisant les graphes de dépendance :

```
CiME> termcrit "minimal";
CiME> termcrit "graph";
```

La terminaison est rapidement prouvée (les symboles entre doubles cotes sont les symboles marqués).

```
CiME> m_termination whole;
Entering the termination expert for modules. Verbose level = 0
Checking module:
{}
```

(Ce module est un module ne comportant que des constructeurs et aucune règle donc aucune paire.)

```
checking each of the 0 strongly connected components :
Termination proof found.
```

```
Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ (#)0 -> # } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.
```

(La seule règle de ce module n'avait pas de paire de dépendance, le graphe de dépendance ne comportait donc pas de partie strictement fortement connexe.)

```
Checking module:
{ V_0 + (V_1 + V_2) -> (V_0 + V_1) + V_2,
  (V_0)1 + (V_1)1 -> ((V_0 + V_1) + (#)1)0,
  (V_0)1 + (V_1)0 -> (V_0 + V_1)1,
  (V_0)0 + (V_1)1 -> (V_0 + V_1)1,
  (V_0)0 + (V_1)0 -> (V_0 + V_1)0,
  # + V_0 -> V_0,
  V_0 + # -> V_0 } (7 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[0](X0) = X0 + 1;
[1](X0) = X0 + 2;
[+](X0,X1) = X1 + X0 + 1;
['+`](X0,X1) = 2*X1 + X0;

Termination proof found.

Checking module:
{ V_0 - # -> V_0,
  # - V_0 -> #,
  (V_0)0 - (V_1)0 -> (V_0 - V_1)0,
  (V_0)0 - (V_1)1 -> ((V_0 - V_1) - (#)1)1,
  (V_0)1 - (V_1)0 -> (V_0 - V_1)1,
  (V_0)1 - (V_1)1 -> (V_0 - V_1)0 } (6 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[0](X0) = X0 + 1;
```

```
[1](X0) = X0 + 1;
[-](X0,X1) = X0;
['-`](X0,X1) = X1*X0;

Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ if(false,V_0,V_1) -> V_1,
  if(true,V_0,V_1) -> V_0 } (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ not(false) -> true,
  not(true) -> false } (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ ge(#,(V_0)0) -> ge(#,V_0),
  ge(#,(V_0)1) -> false,
  ge(V_0,#) -> true,
  ge((V_0)1,(V_1)1) -> ge(V_0,V_1),
  ge((V_0)1,(V_1)0) -> ge(V_0,V_1),
  ge((V_0)0,(V_1)1) -> not(ge(V_1,V_0)),
  ge((V_0)0,(V_1)0) -> ge(V_0,V_1) } (7 rules)

checking each of the 2 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[true] = 0;
```

```
[false] = 0;
[not](X0) = 0;
[#] = 0;
[0](X0) = X0 + 1;
[1](X0) = 0;
[ge](X0,X1) = 0;
['ge`](X0,X1) = X1;

checking component 2 (disjunction of 1 constraints)
[true] = 0;
[false] = 0;
[not](X0) = 0;
[#] = 0;
[0](X0) = X0 + 1;
[1](X0) = X0 + 1;
[ge](X0,X1) = 0;
['ge`](X0,X1) = X1*X0;

Termination proof found.

Checking module:
{ log'(#) -> #,
  log'((V_0)0) -> if(ge(V_0,(#)1),log'(V_0) + (#)1,#),
  log'((V_0)1) -> log'(V_0) + (#)1 } (3 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[true] = 0;
[false] = 1;
[not](X0) = 1;
[#] = 0;
[0](X0) = X0 + 1;
[1](X0) = X0 + 1;
[ge](X0,X1) = X1;
[if](X0,X1,X2) = X2*X0 + X1;
[+](X0,X1) = X1 + X0;
[log'](X0) = X0;
['log'`](X0) = X0;

Termination proof found.

Checking module:
{ log(V_0) -> log'(V_0) - (#)1 } (1 rules)
```

```
checking each of the 0 strongly connected components :
Termination proof found.

- : unit = ()
CiME>
```

Pour chaque hiérarchie une interprétation polynomiale est proposée qui ne dépend que des règles opportunes.

Le tableau III.1 résume les temps passés par le système à trouver une preuve de terminaison de ce système à l'aide d'inteprétations polynomiales simples[2].

Recherche	Sans modules	Avec modules
Sans graphe	156600s	35s
Avec graphes	500s	9s

TAB. III.1 Comparaison des temps nécessaires à la preuve de terminaison.

[2] *Simples* est à prendre ici au sens de Steinbach (cf. paragraphe II.2.1). Les temps sont en secondes sur un ordinateur biprocesseur (P. III 933MHz) disposant de 2Go de mémoire vive et exploité par un système UNIX (LINUX Debian).

Chapitre IV

Critères AC

C*i*ME2 doit permettre de travailler sur des termes contenant des symboles associatifs et commutatifs aussi bien dans la définition et l'utilisation des systèmes de récriture que dans leur preuve de terminaison.

IV.1 Problématique

Le langage est en fait déjà capable de manipuler des symboles et des termes AC en représentation aplatie. Il reste donc à appliquer de façon adéquate les critères de terminaison étendus aux systèmes sur une signature AC. Il faut :

1. Choisir un critère étendu si la signature comporte des symboles AC ;
2. Calculer, si on ne désire pas de marques, les paires étendues non marquées ;
3. Calculer le cas échéant un ensemble convenable de paires étendues AC, c'est-à-dire vérifiant les conditions de dépendance (déf. 85) ;
4. Restreindre la recherche des interprétations à celles qui sont compatibles avec AC.

IV.2 Implantation

Le premier point se réduit en fait à un test aisé, le deuxième ne requiert qu'un test et calcul simple pour chaque règle : si la règle doit être étendue, on lui associe son extension. Il suffit alors de générer les paires de dépendance.

Plus délicat est le calcul d'un ensemble de paires vérifiant les conditions de dépendance. Il nous faut évaluer les fonctions *Split* et *Pairs* (définitions 88 et 89), c'est-à-dire dans un premier temps produire toutes les substitutions intéressantes (par *Split*) et dans un deuxième temps pouvoir construire un ensemble des formes normales partiellement marquées d'un terme donné (permettant alors de définir un ensemble de paires idoine). Ces opérations sont réalisées à l'aide de manipulations combinatoires et de calcul de bipartition d'ensembles.

Le déroulement de la preuve est alors le suivant : si la signature ne comporte pas de symbole AC on génère un ensemble de contraintes dépendant du critère retenu, si un symbole associatif-commutatif est détecté, on produit un ensemble de contraintes modulé le cas échéant par l'application de critères par paires (paires étendues AC, etc.). Le système tente alors de trouver un ordre convenable en respectant le caractère éventuellement AC des interprétations pour les symboles qui le requièrent.

Exemple 51.

Prenons le cas de l'arithmétique de Peàno. À notre connaissance, prouver la terminaison de ce système à l'aide d'interprétations polynomiales simples demandait jusqu'à présent une composition lexicographique de deux interprétations, rendant ainsi la preuve très ardue sinon hors de portée des outils de preuve de terminaison.

En affaiblissant et réduisant les contraintes grâce à une approche modulaire et en utilisant la puissance des paires de dépendance relatives AC, C*i*ME2 *montre la terminaison* $\mathcal{C}_{\mathcal{E}}$ *modulo* AC *de ce système en une fraction de seconde.*

```
let X = vars "x y z";

let F1 = signature "
         0 : constant;
         s : unary;
         + : AC";

let R1 = HTRS {} F1 X "
         x+0 -> x;
         x+s(y) -> s(x+y);
         ";
```

La terminaison de ce système est aisément prouvée, même sans paires marquées :

```
CiME> termcrit ``dp'';
Termination now uses dependency pair criterion
- : unit = ()

CiME> termcrit ``nomarks'';
Dependency pair criterion now do not use marks
- : unit = ()

CiME> m_termination R1;
Entering the termination expert for modules. Verbose level = 0

[0] = 0;
[s](X0) = X0 + 1;
[+](X0,X1) = X1*X0 + 2*X1 + 2*X0 + 2;
- : unit = ()
```

Nous pouvons alors passer à la preuve de terminaison de l'extension complète.

```
let F2 = signature "* : AC";

let R2 = HTRS {R1} F2 X "
         x*0 -> 0;
         x*s(y) -> (x*y) + x;
         ";
```

Choisissons une preuve respectant cette hiérarchie, des paires de dépendance relatives AC *marquées et plaçons la borne supérieure des coefficients de polynômes à 2.*

```
CiME> termcrit "dp";
Termination now uses dependency pair criterion
- : unit = ()

CiME> termcrit "marks";
Dependency pair criterion now uses marks
- : unit = ()

CiME> termpolybound 2;
Search bound of polynomial coefficients is now 2
- : unit = ()

CiME> m_termination R2;
Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ V_0 + 0 -> V_0,
  V_0 + s(V_1) -> s(V_0 + V_1) } (2 rules)
is already proven.

[0] = 0;
[*](X0,X1) = 2*X1*X0 + 2*X1 + 2*X0 + 1;
[s](X0) = X0 + 1;
[+](X0,X1) = X1 + X0;
['*`](X0,X1) = X1*X0 + X1 + X0;
- : unit = ()
```

Nous obtenons ainsi la première preuve automatique de terminaison de ce système à l'aide d'interprétations polynomiales.

Chapitre V

Exemple

Nous proposons ici un exemple issu d'une spécification (fournie par Thomas Arts) de processus communicants développée à l'aide du langage μ-CRL [40].

Ce système est composé de 377 règles présentées modulairement comme suit :

```
   let my_vars = vars "B B1 B2 CS1 CS2 S1 S2 E1 E2 E3 N1 N2 H1 H2 T1 T2 Term1 F0
F1 F2 NewF Record Name Field Fields Pending Pendings Lock Locks Resource Resources
Client List Head Tail E Pid Client1 Client2 MCRLFree0 MCRLFree1";

   let sig_bool = signature
 " T,F:constant;
 or,and,imp: binary;
 not: unary;
 eq: binary;
 if: 3;";

   let trs_bool = HTRS {} sig_bool my_vars
  " or(T,T) -> T;    imp(F,B) -> T;
  or(F,T) -> T;      not(T) -> F;
  or(T,F) -> T;      not(F) -> T;
  or(F,F) -> F;      if(T,B1,B2) -> B1;
  and(T,B) -> B;     if(F,B1,B2) -> B2;
  and(B,T) -> B;     eq(T,T) -> T;
  and(F,B) -> F;     eq(F,F) -> T;
  and(B,F) -> F;     eq(T,F) -> F;
  imp(T,B) -> B;     eq(F,T) -> F ";

let sig_nat = signature " 0: constant ; s: unary ; eqn,le,less:binary";

let trs_nat = HTRS {} sig_nat my_vars "";


let sig_term = signature "
     pid: 1;
     a,excl,excllock,false,lock,locker,mcrlrecord,ok,pending,
     release,request,resource,tag,true,undefined: constant;
```

```
int: 1;
nil: constant;
cons: 2;
tuple: 2;
tuplenil: 1;
eqt:2;
element:2;
record_new: 1;
record_extract:3;
record_update: 4;
record_updates: 3;
case0:3;
locker2_promote_pending:2;
locker2_map_add_pending:3;
locker2_remove_pending:2;
locker2_add_pending:3;
locker2_map_promote_pending: 2;
case1:4 ;
case2: 3;
locker2_claim_lock: 3;
locker2_release_lock: 2;
locker2_map_claim_lock: 3;
case3: 4;
locker2_obtainable: 2;
case4: 3;
locker2_obtainables: 2;
case5: 4;
locker2_check_available: 2;
case6: 4;
locker2_check_availables: 2;
locker2_adduniq: 2;
case7: 4;
equal: 2;
andt: 2;
ort: 2;
hd: 1;
tl: 1;
append: 2;
subtract: 2;
delete: 2;
case8: 4;
gen_tag: 1;
gen_modtageq: 2;
member: 2;
case9: 4;
";
```

```
 let trs_term = HTRS {trs_bool;trs_nat} sig_term my_vars "
eqt(nil,undefined) -> F;
eqt(nil,pid(N2)) -> F;
eqt(nil,int(N2)) -> F;
eqt(nil,cons(H2,T2)) -> F;
eqt(nil,tuple(H2,T2)) -> F;
eqt(nil,tuplenil(H2)) -> F;
eqt(a,nil) -> F;
eqt(a,a) -> T;
eqt(a,excl) -> F;
eqt(a,false) -> F;
eqt(a,lock) -> F;
eqt(a,locker) -> F;
eqt(a,mcrlrecord) -> F;
eqt(a,ok) -> F;
eqt(a,pending) -> F;
eqt(a,release) -> F;
eqt(a,request) -> F;
eqt(a,resource) -> F;
eqt(a,tag) -> F;
eqt(a,true) -> F;
eqt(a,undefined) -> F;
eqt(a,pid(N2)) -> F;
eqt(a,int(N2)) -> F;
eqt(a,cons(H2,T2)) -> F;
eqt(a,tuple(H2,T2)) -> F;
eqt(a,tuplenil(H2)) -> F;
eqt(excl,nil) -> F;
eqt(excl,a) -> F;
eqt(excl,excl) -> T;
eqt(excl,false) -> F;
eqt(excl,lock) -> F;
eqt(excl,locker) -> F;
eqt(excl,mcrlrecord) -> F;
eqt(excl,ok) -> F;
eqt(excl,pending) -> F;
eqt(excl,release) -> F;
eqt(excl,request) -> F;
eqt(excl,resource) -> F;
eqt(excl,tag) -> F;
eqt(excl,true) -> F;
eqt(excl,undefined) -> F;
eqt(excl,pid(N2)) -> F;
eqt(excl, eqt(false,int(N2))) -> F;
eqt(false,cons(H2,T2)) -> F;
```

```
eqt(false,tuple(H2,T2)) -> F;
eqt(false,tuplenil(H2)) -> F;
eqt(lock,nil) -> F;
eqt(lock,a) -> F;
eqt(lock,excl) -> F;
eqt(lock,false) -> F;
eqt(lock,lock) -> T;
eqt(lock,locker) -> F;
eqt(lock,mcrlrecord) -> F;
eqt(lock,ok) -> F;
eqt(lock,pending) -> F;
eqt(lock,release) -> F;
eqt(lock,request) -> F;
eqt(lock,resource) -> F;
eqt(lock,tag) -> F;
eqt(lock,true) -> F;
eqt(lock,undefined) -> F;
eqt(lock,pid(N2)) -> F;
eqt(lock,int(N2)) -> F;
eqt(lock,cons(H2,T2)) -> F;
eqt(lock,tuple(H2,T2)) -> F;
eqt(lock,tuplenil(H2)) -> F;
eqt(locker,nil) -> F;
eqt(locker,a) -> F;
eqt(locker,excl) -> F;
eqt(locker,false) -> F;
eqt(locker,lock) -> F;
eqt(locker,locker) -> T;
eqt(locker,mcrlrecord) -> F;
eqt(locker,ok) -> F;
eqt(locker,pending) -> F;
eqt(locker,release) -> F;
eqt(locker,request) -> F;
eqt(locker,resource) -> F;
eqt(locker,tag) -> F;
eqt(locker,true) -> F;
eqt(locker,undefined) -> F;
eqt(locker,pid(N2)) -> F;
eqt(locker,int(N2)) -> F;
eqt(locker,cons(H2,T2)) -> F;
eqt(locker,tuple(H2,T2)) -> F;
eqt(locker,tuplenil(H2)) -> F;
eqt(mcrlrecord,nil) -> F;
eqt(mcrlrecord,a) -> F;
eqt(mcrlrecord,excl) -> F;
eqt(mcrlrecord,false) -> F;
eqt(mcrlrecord,lock) -> F;
eqt(mcrlrecord,locker) -> F;
eqt(mcrlrecord,mcrlrecord) -> T;
eqt(mcrlrecord,ok) -> F;
eqt(mcrlrecord,pending) -> F;
eqt(mcrlrecord,release) -> F;
eqt(mcrlrecord,request) -> F;
eqt(mcrlrecord,resource) -> F;
eqt(ok,resource) -> F;
eqt(ok,tag) -> F;
eqt(ok,true) -> F;
eqt(ok,undefined) -> F;
eqt(ok,pid(N2)) -> F;
eqt(ok,int(N2)) -> F;
eqt(ok,cons(H2,T2)) -> F;
eqt(ok,tuple(H2,T2)) -> F;
eqt(ok,tuplenil(H2)) -> F;
eqt(pending,nil) -> F;
eqt(pending,a) -> F;
eqt(pending,excl) -> F;
eqt(pending,false) -> F;
eqt(pending,lock) -> F;
eqt(pending,locker) -> F;
eqt(pending,mcrlrecord) -> F;
eqt(pending,ok) -> F;
eqt(pending,pending) -> T;
eqt(pending,release) -> F;
eqt(pending,request) -> F;
eqt(pending,resource) -> F;
eqt(pending,tag) -> F;
eqt(pending,true) -> F;
eqt(pending,undefined) -> F;
eqt(pending,pid(N2)) -> F;
eqt(pending,int(N2)) -> F;
eqt(pending,cons(H2,T2)) -> F;
eqt(pending,tuple(H2,T2)) -> F;
eqt(pending,tuplenil(H2)) -> F;
eqt(release,nil) -> F;
eqt(release,a) -> F;
eqt(release,excl) -> F;
eqt(release,false) -> F;
eqt(release,lock) -> F;
eqt(release,locker) -> F;
eqt(release,mcrlrecord) -> F;
eqt(release,ok) -> F;
eqt(request,mcrlrecord) -> F;
```

```
eqt(request,ok) -> F;
eqt(request,pending) -> F;
eqt(request,release) -> F;
eqt(request,request) -> T;
eqt(request,resource) -> F;
eqt(request,tag) -> F;
eqt(request,true) -> F;
eqt(request,undefined) -> F;
eqt(request,pid(N2)) -> F;
eqt(request,int(N2)) -> F;
eqt(request,cons(H2,T2)) -> F;
eqt(request,tuple(H2,T2)) -> F;
eqt(request,tuplenil(H2)) -> F;
eqt(resource,nil) -> F;
eqt(resource,a) -> F;
eqt(resource,excl) -> F;
eqt(resource,false) -> F;
eqt(resource,lock) -> F;
eqt(resource,locker) -> F;
eqt(resource,mcrlrecord) -> F;
eqt(resource,ok) -> F;
eqt(resource,pending) -> F;
eqt(resource,release) -> F;
eqt(resource,request) -> F;
eqt(resource,resource) -> T;
eqt(resource,tag) -> F;
eqt(resource,true) -> F;
eqt(resource,undefined) -> F;
eqt(resource,pid(N2)) -> F;
eqt(resource,int(N2)) -> F;
eqt(resource,cons(H2,T2)) -> F;
eqt(resource,tuple(H2,T2)) -> F;
eqt(resource,tuplenil(H2)) -> F;
eqt(tag,nil) -> F;
eqt(tag,a) -> F;
eqt(tag,excl) -> F;
eqt(tag,false) -> F;
eqt(tag,lock) -> F;
eqt(tag,locker) -> F;
eqt(tag,mcrlrecord) -> F;
eqt(tag,ok) -> F;
eqt(tag,pending) -> F;
eqt(tag,release) -> F;
eqt(tag,request) -> F;
eqt(tag,resource) -> F;
eqt(tag,tag) -> T;
eqt(tag,true) -> F;
eqt(tag,undefined) -> F;
eqt(tag,pid(N2)) -> F;
eqt(tag,int(N2)) -> F;
eqt(tag,cons(H2,T2)) -> F;
eqt(tag,tuple(H2,T2)) -> F;
eqt(tag,tuplenil(H2)) -> F;
eqt(true,nil) -> F;
eqt(true,a) -> F;
eqt(true,excl) -> F;
eqt(true,false) -> F;
eqt(true,lock) -> F;
eqt(true,locker) -> F;
eqt(true,mcrlrecord) -> F;
eqt(true,ok) -> F;
eqt(true,pending) -> F;
eqt(true,release) -> F;
eqt(true,request) -> F;
eqt(true,resource) -> F;
eqt(true,tag) -> F;
eqt(true,true) -> T;
eqt(true,undefined) -> F;
eqt(true,pid(N2)) -> F;
eqt(true,int(N2)) -> F;
eqt(true,cons(H2,T2)) -> F;
eqt(true,tuple(H2,T2)) -> F;
eqt(true,tuplenil(H2)) -> F;
eqt(undefined,nil) -> F;
eqt(undefined,a) -> F;
eqt(undefined,tuplenil(H2)) -> F;
eqt(pid(N1),nil) -> F;
eqt(pid(N1),a) -> F;
eqt(pid(N1),excl) -> F;
eqt(pid(N1),false) -> F;
eqt(pid(N1),lock) -> F;
eqt(pid(N1),locker) -> F;
eqt(pid(N1),mcrlrecord) -> F;
eqt(pid(N1),ok) -> F;
eqt(pid(N1),pending) -> F;
eqt(pid(N1),release) -> F;
eqt(pid(N1),request) -> F;
eqt(pid(N1),resource) -> F;
eqt(pid(N1),tag) -> F;
eqt(pid(N1),true) -> F;
eqt(pid(N1),undefined) -> F;
eqt(pid(N1),pid(N2)) -> eqt(N1,N2);
eqt(pid(N1),int(N2)) -> F;
eqt(pid(N1),cons(H2,T2)) -> F;
eqt(pid(N1),tuple(H2,T2)) -> F;
eqt(pid(N1),tuplenil(H2)) -> F;
eqt(int(N1),nil) -> F;
eqt(int(N1),a) -> F;
eqt(int(N1),excl) -> F;
eqt(int(N1),false) -> F;
```

```
  eqt(int(N1),lock) -> F;
  eqt(int(N1),locker) -> F;
  eqt(int(N1),mcrlrecord) -> F;
  eqt(int(N1),ok) -> F;
  eqt(int(N1),pending) -> F;
  eqt(int(N1),release) -> F;
  eqt(int(N1),request) -> F;
  eqt(int(N1),resource) -> F;
  eqt(int(N1),tag) -> F;
  eqt(int(N1),true) -> F;
  eqt(int(N1),undefined) -> F;
  eqt(cons(H1,T1),resource) -> F;
  eqt(cons(H1,T1),tag) -> F;
  eqt(cons(H1,T1),true) -> F;
  eqt(cons(H1,T1),undefined) -> F;
  eqt(cons(H1,T1),pid(N2)) -> F;
  eqt(cons(H1,T1),int(N2)) -> F;
  eqt(cons(H1,T1),tuple(H2,T2)) -> F;
  eqt(cons(H1,T1),tuplenil(H2)) -> F;
  eqt(tuple(H1,T1),nil) -> F;
  eqt(tuple(H1,T1),a) -> F;
  eqt(tuple(H1,T1),excl) -> F;
  eqt(tuple(H1,T1),false) -> F;
  eqt(tuple(H1,T1),lock) -> F;
  eqt(tuple(H1,T1),locker) -> F;
  eqt(tuple(H1,T1),mcrlrecord) -> F;
  eqt(tuple(H1,T1),ok) -> F;
  eqt(tuple(H1,T1),pending) -> F;
  eqt(tuple(H1,T1),release) -> F;
  eqt(tuple(H1,T1),request) -> F;
  eqt(tuple(H1,T1),resource) -> F;
  eqt(tuple(H1,T1),tag) -> F;
  eqt(tuple(H1,T1),true) -> F;
  eqt(tuple(H1,T1),undefined) -> F;
  eqt(tuple(H1,T1),pid(N2)) -> F;
  eqt(tuple(H1,T1),int(N2)) -> F;
  eqt(tuple(H1,T1),cons(H2,T2)) -> F;
  eqt(tuple(H1,T1),tuplenil(H2)) -> F;
  eqt(tuplenil(H1),nil) -> F;
  eqt(tuplenil(H1),a) -> F;
  eqt(tuplenil(H1),excl) -> F;
  eqt(tuplenil(H1),false) -> F;
  eqt(tuplenil(H1),lock) -> F;
  eqt(tuplenil(H1),locker) -> F;
  eqt(tuplenil(H1),mcrlrecord) -> F;
  eqt(tuplenil(H1),ok) -> F;
  eqt(tuplenil(H1),pending) -> F;
  eqt(tuplenil(H1),release) -> F;
  eqt(tuplenil(H1),request) -> F;
  eqt(tuplenil(H1),resource) -> F;
  eqt(tuplenil(H1),tag) -> F;
  eqt(tuplenil(H1),true) -> F;
  eqt(tuplenil(H1),undefined) -> F;
  eqt(tuplenil(H1),pid(N2)) -> F;
  eqt(tuplenil(H1),int(N2)) -> F;
  eqt(tuplenil(H1),cons(H2,T2)) -> F;
  eqt(tuplenil(H1),tuple(H2,T2)) -> F;
element(int(s(0)),tuplenil(T1)) -> T1;
element(int(s(0)),tuple(T1,T2)) -> T1;
eqt(tuplenil(H1),tuplenil(H2)) -> eqt(H1,H2);
element(int(s(s(N1))),tuple(T1,T2)) -> element(int(s(N1)),T2);
eqt(cons(H1,T1),cons(H2,T2)) -> and(eqt(H1,H2),eqt(T1,T2));
eqt(tuple(H1,T1),tuple(H2,T2)) -> and(eqt(H1,H2),eqt(T1,T2));

record_new(lock) ->
    tuple(mcrlrecord,
    tuple(lock, tuple(undefined, tuple(nil, tuplenil(nil)))));

record_extract(tuple(mcrlrecord,
  tuple(lock,
        tuple(F0, tuple(F1, tuplenil(F2))))),lock,resource) ->
tuple(mcrlrecord, tuple(lock, tuple(F0, tuple(NewF, tuplenil(F2)))));

record_update(tuple(mcrlrecord,
  tuple(lock,
        tuple(F0, tuple(F1, tuplenil(F2))))),lock,pending,NewF) ->
```

```
    tuple(mcrlrecord, tuple(lock, tuple(F0, tuple(F1, tuplenil(NewF)))));

record_updates(Record,Name,nil) -> Record;

record_updates(Record,Name,
              cons(tuple(Field, tuplenil(NewF)),Fields)) ->
    record_updates(record_update(Record,Name,Field,NewF),Name,Fields);

locker2_map_promote_pending(nil,Pending) ->
    nil;

locker2_map_promote_pending(cons(Lock,Locks),Pending) ->
    cons(
    locker2_promote_pending(Lock,Pending),
    locker2_map_promote_pending(Locks,Pending));

locker2_map_claim_lock(nil,Resources,Client) ->
    nil;

locker2_map_claim_lock(cons(Lock,Locks),Resources,Client) ->
    cons(locker2_claim_lock(Lock,Resources,Client),
    locker2_map_claim_lock(Locks,Resources,Client));

locker2_map_add_pending(nil,Resources,Client) ->
    nil;


locker2_promote_pending(Lock,Client) ->
    case0(Client,Lock,record_extract(Lock,lock,pending));

case0(Client,Lock,cons(Client,Pendings)) ->
    record_updates(Lock,lock,cons(tuple(excl, tuplenil(Client)),
      cons(tuple(pending, tuplenil(Pendings)),nil)));

case0(Client,Lock,MCRLFree0) ->
    Lock;

locker2_remove_pending(Lock,Client) ->
    record_updates(Lock,lock,cons(tuple(pending,
      tuplenil(subtract(record_extract(Lock,lock,pending),
                        cons(Client,nil)))),nil));

locker2_add_pending(Lock,Resources,Client) ->
    case1(Client,Resources,Lock,member(
      record_extract(Lock,lock,resource),Resources));

case1(Client,Resources,Lock,true) ->
```

```
    record_updates(Lock,lock,cons(tuple(pending,
      tuplenil(append(record_extract(Lock,lock,pending),
                      cons(Client,nil)))),nil));

case1(Client,Resources,Lock,false) ->
    Lock;

locker2_release_lock(Lock,Client) ->
    case2(Client,Lock,gen_modtageq(Client,record_extract(Lock,lock,excl)));

case2(Client,Lock,true) ->
    record_updates(Lock,lock,cons(tuple(excllock,excl),nil));

case4(Client,Lock,MCRLFree1) ->
    false;

locker2_obtainables(nil,Client) ->
    true;

locker2_obtainables(cons(Lock,Locks),Client) ->
    case5(Client,Locks,Lock,member(Client,record_extract(Lock,lock,pending)));

case5(Client,Locks,Lock,true) ->
    andt(locker2_obtainable(Lock,Client),locker2_obtainables(Locks,Client));

case5(Client,Locks,Lock,false) ->
    locker2_obtainables(Locks,Client);

locker2_check_available(Resource,nil) ->
    false;
locker2_check_available(Resource,cons(Lock,Locks)) ->
    case6(Locks,Lock,Resource,
          equal(Resource,record_extract(Lock,lock,resource)));

case6(Locks,Lock,Resource,true) ->
    andt(equal(record_extract(Lock,lock,excl),nil),
         equal(record_extract(Lock,lock,pending),nil));

case6(Locks,Lock,Resource,false) ->
    locker2_check_available(Resource,Locks);

locker2_check_availables(nil,Locks) ->
    true;
locker2_check_availables(cons(Resource,Resources),Locks) ->
    andt(locker2_check_available(Resource,Locks),
         locker2_check_availables(Resources,Locks));
```

```
locker2_adduniq(nil,List) ->
    List;

append(cons(Head,Tail),List) ->
    cons(Head,append(Tail,List));

subtract(List,nil) ->
    List;

subtract(List,cons(Head,Tail)) ->
    subtract(delete(Head,List),Tail);

delete(E,nil) ->
    nil;

delete(E,cons(Head,Tail)) ->
    case8(Tail,Head,E,equal(E,Head));

case8(Tail,Head,E,true) ->
    Tail;

case8(Tail,Head,E,false) ->
    cons(Head,delete(E,Tail));

gen_tag(Pid) ->
    tuple(Pid, tuplenil(tag));

gen_modtageq(Client1,Client2) ->
    equal(Client1,Client2);

member(E,nil) ->
    false;

member(E,cons(Head,Tail)) ->
    case9(Tail,Head,E,equal(E,Head));

case9(Tail,Head,E,true) ->
    true;

case9(Tail,Head,E,false) ->
    member(E,Tail);";
```

```
let sig_stack = signature "
   push1s: 2;
   pops: 1;
   tops: 1;
   istops: 2;
   eqs:2;
   empty:constant;
   stack:2;
   pushs:2;
   ";

let trs_stack = HTRS {trs_bool; trs_term} sig_stack my_vars "
   eqs(empty,empty) -> T;
   eqs(empty,stack(E2,S2)) -> F;
   eqs(stack(E1,S1),empty) -> F;
   eqs(stack(E1,S1),stack(E2,S2)) -> and(eqt(E1,E2),eqs(S1,S2));
   pushs(E1,S1) -> stack(E1,S1);
   pops(stack(E1,S1)) -> S1;
   tops(stack(E1,S1)) -> E1;
   istops(E1,empty) -> F;
   istops(E1,stack(E2,S1)) -> eqt(E1,E2);";


let sig_call = signature "
   nocalls: constant ;
   calls: 3 ;
   eqc: binary;
   push: 3;
   push1: 6;
   pop: 2;
   pop1: 5;
   stacked: 3;
   stacked1: 5";

let call_trs = HTRS {trs_bool;trs_stack;trs_term} sig_call my_vars "
   eqc(nocalls,nocalls) -> T;
   eqc(nocalls,calls(E2,S2,CS2)) -> F;
   eqc(calls(E1,S1,CS1),nocalls) -> F;
   eqc(calls(E1,S1,CS1),calls(E2,S2,CS2)) ->
     and(eqt(E1,E2), and(eqs(S1,S2), eqc(CS1,CS2)));
   push(E1,E2,nocalls) -> calls(E1,stack(E2,empty),nocalls);
   push(E1,E2,calls(E3,S1,CS1)) -> push1(E1,E2,E3,S1,CS1,eqt(E1,E3));
   push1(E1,E2,E3,S1,CS1,T) -> calls(E3,pushs(E2,S1),CS1);
";
```

Prouvons la terminaison de ce système grâce à C*i*ME2. Les modules simplifiant considérablement les interprétations, nous pouvons commencer par chercher des interprétations linéaires sur la décomposition en modules minimaux :

```
polyinterpkind ‘‘linear’’;
termcrit ‘‘minimal’’;
```

Le système fournit alors une preuve en un peu moins de trois secondes[1].

Pour des raisons de clarté de nombreuses preuves de terminaison des modules sans règles ont été omises. Cette omission est signalée par `[...]`.

```
Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

[...]

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.


Checking module:
{ and(T,V_0) -> V_0,
  and(V_0,T) -> V_0,
  and(F,V_0) -> F,
  and(V_0,F) -> F } (4 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ eqt(tuplenil(V_12),tuplenil(V_13)) -> eqt(V_12,V_13),
  eqt(tuplenil(V_12),tuple(V_13,V_15)) -> F,
  eqt(tuplenil(V_12),cons(V_13,V_15)) -> F,
  eqt(tuplenil(V_12),int(V_11)) -> F,
  eqt(tuplenil(V_12),pid(V_11)) -> F,
  eqt(tuplenil(V_12),undefined) -> F,
  eqt(tuplenil(V_12),true) -> F,
  eqt(tuplenil(V_12),tag) -> F,
  eqt(tuplenil(V_12),resource) -> F,
  eqt(tuplenil(V_12),request) -> F,
  eqt(tuplenil(V_12),release) -> F,
  eqt(tuplenil(V_12),pending) -> F,
  eqt(tuplenil(V_12),ok) -> F,
  eqt(tuplenil(V_12),mcrlrecord) -> F,
```

[1] L’ordinateur est un biprocesseur (P. III 933MHz) disposant de 2Go de mémoire vive et exploité par un système UNIX (LINUX Debian).

```
eqt(tuplenil(V_12),locker) -> F,
eqt(tuplenil(V_12),lock) -> F,
eqt(tuplenil(V_12),false) -> F,
eqt(tuplenil(V_12),excl) -> F,
eqt(tuplenil(V_12),a) -> F,
eqt(tuplenil(V_12),nil) -> F,
eqt(tuple(V_12,V_14),tuplenil(V_13)) -> F,
eqt(tuple(V_12,V_14),tuple(V_13,V_15)) ->
and(eqt(V_12,V_13),eqt(V_14,V_15)),
eqt(tuple(V_12,V_14),cons(V_13,V_15)) -> F,
eqt(tuple(V_12,V_14),int(V_11)) -> F,
eqt(tuple(V_12,V_14),pid(V_11)) -> F,
eqt(tuple(V_12,V_14),undefined) -> F,
eqt(tuple(V_12,V_14),true) -> F,
eqt(tuple(V_12,V_14),tag) -> F,
eqt(tuple(V_12,V_14),resource) -> F,
eqt(tuple(V_12,V_14),request) -> F,
eqt(tuple(V_12,V_14),release) -> F,
eqt(tuple(V_12,V_14),pending) -> F,
eqt(tuple(V_12,V_14),ok) -> F,
eqt(tuple(V_12,V_14),mcrlrecord) -> F,
eqt(tuple(V_12,V_14),locker) -> F,
eqt(tuple(V_12,V_14),lock) -> F,
eqt(tuple(V_12,V_14),false) -> F,
eqt(tuple(V_12,V_14),excl) -> F,
eqt(tuple(V_12,V_14),a) -> F,
eqt(tuple(V_12,V_14),nil) -> F,
eqt(cons(V_12,V_14),tuplenil(V_13)) -> F,
eqt(cons(V_12,V_14),tuple(V_13,V_15)) -> F,
eqt(cons(V_12,V_14),cons(V_13,V_15)) ->
  and(eqt(V_12,V_13),eqt(V_14,V_15)),
eqt(cons(V_12,V_14),int(V_11)) -> F,
eqt(cons(V_12,V_14),pid(V_11)) -> F,
eqt(cons(V_12,V_14),undefined) -> F,
eqt(cons(V_12,V_14),true) -> F,
eqt(cons(V_12,V_14),tag) -> F,
eqt(cons(V_12,V_14),resource) -> F,
eqt(int(V_10),undefined) -> F,
eqt(int(V_10),true) -> F,
eqt(int(V_10),tag) -> F,
eqt(int(V_10),resource) -> F,
eqt(int(V_10),request) -> F,
eqt(int(V_10),release) -> F,
eqt(int(V_10),pending) -> F,
eqt(int(V_10),ok) -> F,
eqt(int(V_10),mcrlrecord) -> F,
eqt(int(V_10),locker) -> F,
```

```
eqt(int(V_10),lock) -> F,
eqt(int(V_10),false) -> F,
eqt(int(V_10),excl) -> F,
eqt(int(V_10),a) -> F,
eqt(int(V_10),nil) -> F,
eqt(pid(V_10),tuplenil(V_13)) -> F,
eqt(pid(V_10),tuple(V_13,V_15)) -> F,
eqt(pid(V_10),cons(V_13,V_15)) -> F,
eqt(pid(V_10),int(V_11)) -> F,
eqt(pid(V_10),pid(V_11)) -> eqt(V_10,V_11),
eqt(pid(V_10),undefined) -> F,
eqt(pid(V_10),true) -> F,
eqt(pid(V_10),tag) -> F,
eqt(pid(V_10),resource) -> F,
eqt(pid(V_10),request) -> F,
eqt(pid(V_10),release) -> F,
eqt(pid(V_10),pending) -> F,
eqt(pid(V_10),ok) -> F,
eqt(pid(V_10),mcrlrecord) -> F,
eqt(pid(V_10),locker) -> F,
eqt(pid(V_10),lock) -> F,
eqt(pid(V_10),false) -> F,
eqt(pid(V_10),excl) -> F,
eqt(pid(V_10),a) -> F,
eqt(pid(V_10),nil) -> F,
eqt(undefined,tuplenil(V_13)) -> F,
eqt(undefined,a) -> F,
eqt(undefined,nil) -> F,
eqt(true,tuplenil(V_13)) -> F,
eqt(true,tuple(V_13,V_15)) -> F,
eqt(true,cons(V_13,V_15)) -> F,
eqt(true,int(V_11)) -> F,
eqt(true,pid(V_11)) -> F,
eqt(true,undefined) -> F,
eqt(true,true) -> T,
eqt(true,tag) -> F,
eqt(true,resource) -> F,
eqt(true,request) -> F,
eqt(true,release) -> F,
eqt(true,pending) -> F,
eqt(true,ok) -> F,
eqt(true,mcrlrecord) -> F,
eqt(true,locker) -> F,
eqt(true,lock) -> F,
eqt(true,false) -> F,
eqt(true,excl) -> F,
eqt(true,a) -> F,
```

```
eqt(true,nil) -> F,
eqt(tag,tuplenil(V_13)) -> F,
eqt(tag,tuple(V_13,V_15)) -> F,
eqt(tag,cons(V_13,V_15)) -> F,
eqt(tag,int(V_11)) -> F,
eqt(tag,pid(V_11)) -> F,
eqt(tag,undefined) -> F,
eqt(tag,true) -> F,
eqt(tag,tag) -> T,
eqt(tag,resource) -> F,
eqt(tag,request) -> F,
eqt(tag,release) -> F,
eqt(tag,pending) -> F,
eqt(tag,ok) -> F,
eqt(tag,mcrlrecord) -> F,
eqt(tag,locker) -> F,
eqt(tag,lock) -> F,
eqt(tag,false) -> F,
eqt(tag,excl) -> F,
eqt(tag,a) -> F,
eqt(tag,nil) -> F,
eqt(resource,tuplenil(V_13)) -> F,
eqt(resource,tuple(V_13,V_15)) -> F,
eqt(resource,cons(V_13,V_15)) -> F,
eqt(resource,int(V_11)) -> F,
eqt(resource,pid(V_11)) -> F,
eqt(resource,undefined) -> F,
eqt(resource,true) -> F,
eqt(resource,tag) -> F,
eqt(resource,resource) -> T,
eqt(resource,request) -> F,
eqt(resource,release) -> F,
eqt(resource,pending) -> F,
eqt(resource,ok) -> F,
eqt(resource,mcrlrecord) -> F,
eqt(resource,locker) -> F,
eqt(resource,lock) -> F,
eqt(resource,false) -> F,
eqt(resource,excl) -> F,
eqt(resource,a) -> F,
eqt(resource,nil) -> F,
eqt(request,tuplenil(V_13)) -> F,
eqt(request,tuple(V_13,V_15)) -> F,
eqt(request,cons(V_13,V_15)) -> F,
eqt(request,int(V_11)) -> F,
eqt(request,pid(V_11)) -> F,
eqt(request,undefined) -> F,
```

```
eqt(request,true) -> F,
eqt(request,tag) -> F,
eqt(request,resource) -> F,
eqt(request,request) -> T,
eqt(request,release) -> F,
eqt(request,pending) -> F,
eqt(request,ok) -> F,
eqt(request,mcrlrecord) -> F,
eqt(release,ok) -> F,
eqt(release,mcrlrecord) -> F,
eqt(release,locker) -> F,
eqt(release,lock) -> F,
eqt(release,false) -> F,
eqt(release,excl) -> F,
eqt(release,a) -> F,
eqt(release,nil) -> F,
eqt(pending,tuplenil(V_13)) -> F,
eqt(pending,tuple(V_13,V_15)) -> F,
eqt(pending,cons(V_13,V_15)) -> F,
eqt(pending,int(V_11)) -> F,
eqt(pending,pid(V_11)) -> F,
eqt(pending,undefined) -> F,
eqt(pending,true) -> F,
eqt(pending,tag) -> F,
eqt(pending,resource) -> F,
eqt(pending,request) -> F,
eqt(pending,release) -> F,
eqt(pending,pending) -> T,
eqt(pending,ok) -> F,
eqt(pending,mcrlrecord) -> F,
eqt(pending,locker) -> F,
eqt(pending,lock) -> F,
eqt(pending,false) -> F,
eqt(pending,excl) -> F,
eqt(pending,a) -> F,
eqt(pending,nil) -> F,
eqt(ok,tuplenil(V_13)) -> F,
eqt(ok,tuple(V_13,V_15)) -> F,
eqt(ok,cons(V_13,V_15)) -> F,
eqt(ok,int(V_11)) -> F,
eqt(ok,pid(V_11)) -> F,
eqt(ok,undefined) -> F,
eqt(ok,true) -> F,
eqt(ok,tag) -> F,
eqt(ok,resource) -> F,
eqt(mcrlrecord,resource) -> F,
eqt(mcrlrecord,request) -> F,
```

```
eqt(mcrlrecord,release) -> F,
eqt(mcrlrecord,pending) -> F,
eqt(mcrlrecord,ok) -> F,
eqt(mcrlrecord,mcrlrecord) -> T,
eqt(mcrlrecord,locker) -> F,
eqt(mcrlrecord,lock) -> F,
eqt(mcrlrecord,false) -> F,
eqt(mcrlrecord,excl) -> F,
eqt(mcrlrecord,a) -> F,
eqt(mcrlrecord,nil) -> F,
eqt(locker,tuplenil(V_13)) -> F,
eqt(locker,tuple(V_13,V_15)) -> F,
eqt(locker,cons(V_13,V_15)) -> F,
eqt(locker,int(V_11)) -> F,
eqt(locker,pid(V_11)) -> F,
eqt(locker,undefined) -> F,
eqt(locker,true) -> F,
eqt(locker,tag) -> F,
eqt(locker,resource) -> F,
eqt(locker,request) -> F,
eqt(locker,release) -> F,
eqt(locker,pending) -> F,
eqt(locker,ok) -> F,
eqt(locker,mcrlrecord) -> F,
eqt(locker,locker) -> T,
eqt(locker,lock) -> F,
eqt(locker,false) -> F,
eqt(locker,excl) -> F,
eqt(locker,a) -> F,
eqt(locker,nil) -> F,
eqt(lock,tuplenil(V_13)) -> F,
eqt(lock,tuple(V_13,V_15)) -> F,
eqt(lock,cons(V_13,V_15)) -> F,
eqt(lock,int(V_11)) -> F,
eqt(lock,pid(V_11)) -> F,
eqt(lock,undefined) -> F,
eqt(lock,true) -> F,
eqt(lock,tag) -> F,
eqt(lock,resource) -> F,
eqt(lock,request) -> F,
eqt(lock,release) -> F,
eqt(lock,pending) -> F,
eqt(lock,ok) -> F,
eqt(lock,mcrlrecord) -> F,
eqt(lock,locker) -> F,
eqt(lock,lock) -> T,
eqt(lock,false) -> F,
```

```
eqt(lock,excl) -> F,
eqt(lock,a) -> F,
eqt(lock,nil) -> F,
eqt(false,tuplenil(V_13)) -> F,
eqt(false,tuple(V_13,V_15)) -> F,
eqt(false,cons(V_13,V_15)) -> F,
eqt(excl,eqt(false,int(V_11))) -> F,
eqt(excl,pid(V_11)) -> F,
eqt(excl,undefined) -> F,
eqt(excl,true) -> F,
eqt(excl,tag) -> F,
eqt(excl,resource) -> F,
eqt(excl,request) -> F,
eqt(excl,release) -> F,
eqt(excl,pending) -> F,
eqt(excl,ok) -> F,
eqt(excl,mcrlrecord) -> F,
eqt(excl,locker) -> F,
eqt(excl,lock) -> F,
eqt(excl,false) -> F,
eqt(excl,excl) -> T,
eqt(excl,a) -> F,
eqt(excl,nil) -> F,
eqt(a,tuplenil(V_13)) -> F,
eqt(a,tuple(V_13,V_15)) -> F,
eqt(a,cons(V_13,V_15)) -> F,
eqt(a,int(V_11)) -> F,
eqt(a,pid(V_11)) -> F,
eqt(a,undefined) -> F,
eqt(a,true) -> F,
eqt(a,tag) -> F,
eqt(a,resource) -> F,
eqt(a,request) -> F,
eqt(a,release) -> F,
eqt(a,pending) -> F,
eqt(a,ok) -> F,
eqt(a,mcrlrecord) -> F,
eqt(a,locker) -> F,
eqt(a,lock) -> F,
eqt(a,false) -> F,
eqt(a,excl) -> F,
eqt(a,a) -> T,
eqt(a,nil) -> F,
eqt(nil,tuplenil(V_13)) -> F,
eqt(nil,tuple(V_13,V_15)) -> F,
eqt(nil,cons(V_13,V_15)) -> F,
eqt(nil,int(V_11)) -> F,
```

```
  eqt(nil,pid(V_11)) -> F,
  eqt(nil,undefined) -> F } (296 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[T] = 0;
[F] = 0;
[int](X0) = 0;
[tuple](X0,X1) = X1 + X0 + 1;
[tuplenil](X0) = X0 + 1;
[lock] = 0;
[mcrlrecord] = 0;
[undefined] = 0;
[nil] = 0;
[pending] = 0;
[cons](X0,X1) = X1 + X0 + 1;
[resource] = 0;
[false] = 0;
[true] = 0;
[excl] = 0;
[tag] = 0;
[and](X0,X1) = X1 + X0;
[pid](X0) = X0 + 1;
[a] = 0;
[locker] = 0;
[ok] = 0;
[release] = 0;
[request] = 0;
[eqt](X0,X1) = 0;
['eqt`](X0,X1) = X0;

Termination proof found.

Checking module:
{ push(V_7,V_8,nocalls) -> calls(V_7,stack(V_8,empty),nocalls),
  push(V_7,V_8,calls(V_9,V_5,V_3)) -> push1(V_7,V_8,V_9,V_5,V_3,
    eqt(V_7,V_9)) }
  (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ eqs(stack(V_7,V_5),stack(V_8,V_6)) -> and(eqt(V_7,V_8),eqs(V_5,V_6)),
  eqs(stack(V_7,V_5),empty) -> F,
  eqs(empty,stack(V_8,V_6)) -> F,
  eqs(empty,empty) -> T } (4 rules)
```

```
checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[T] = 0;
[F] = 0;
[int](X0) = 0;
[tuple](X0,X1) = 0;
[tuplenil](X0) = 0;
[lock] = 0;
[mcrlrecord] = 0;
[undefined] = 0;
[nil] = 0;
[pending] = 0;
[cons](X0,X1) = 0;
[resource] = 0;
[false] = 0;
[true] = 0;
[excl] = 0;
[tag] = 0;
[stack](X0,X1) = X1 + 1;
[and](X0,X1) = X1 + X0;
[pid](X0) = 0;
[a] = 0;
[locker] = 0;
[ok] = 0;
[release] = 0;
[request] = 0;
[eqt](X0,X1) = 0;
[empty] = 0;
[eqs](X0,X1) = 0;
['eqs`](X0,X1) = X0;

Termination proof found.

Checking module:
{ eqc(nocalls,nocalls) -> T,
  eqc(nocalls,calls(V_8,V_6,V_4)) -> F,
  eqc(calls(V_7,V_5,V_3),nocalls) -> F,
  eqc(calls(V_7,V_5,V_3),calls(V_8,V_6,V_4)) ->
  and(eqt(V_7,V_8),and(eqs(V_5,V_6),eqc(V_3,V_4))) } (4 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[T] = 0;
[F] = 0;
[int](X0) = 0;
[tuple](X0,X1) = 0;
```

```
[tuplenil](X0) = 0;
[lock] = 0;
[mcrlrecord] = 0;
[undefined] = 0;
[nil] = 0;
[pending] = 0;
[cons](X0,X1) = 0;
[resource] = 0;
[false] = 0;
[true] = 0;
[excl] = 0;
[tag] = 0;
[stack](X0,X1) = 0;
[and](X0,X1) = X1 + X0;
[pid](X0) = 0;
[a] = 0;
[locker] = 0;
[ok] = 0;
[release] = 0;
[request] = 0;
[eqt](X0,X1) = 0;
[empty] = 0;
[eqs](X0,X1) = 0;
[nocalls] = 0;
[calls](X0,X1,X2) = X2 + 1;
[eqc](X0,X1) = 0;
['eqc`](X0,X1) = X0;

Termination proof found.

Checking module:
{ istops(V_7,stack(V_8,V_5)) -> eqt(V_7,V_8),
  istops(V_7,empty) -> F } (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ tops(stack(V_7,V_5)) -> V_7 } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ pops(stack(V_7,V_5)) -> V_5 } (1 rules)

checking each of the 0 strongly connected components :
```

```
Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ case9(V_34,V_33,V_35,false) -> member(V_35,V_34),
  case9(V_34,V_33,V_35,true) -> true,
  member(V_35,nil) -> false,
  member(V_35,cons(V_33,V_34)) -> case9(V_34,V_33,V_35,equal(V_35,V_33)) }
  (4 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ gen_modtageq(V_37,V_38) -> equal(V_37,V_38) } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ gen_tag(V_36) -> tuple(V_36,tuplenil(tag)) } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ case8(V_34,V_33,V_35,false) -> cons(V_33,delete(V_35,V_34)),
  case8(V_34,V_33,V_35,true) -> V_34,
  delete(V_35,nil) -> nil,
  delete(V_35,cons(V_33,V_34)) -> case8(V_34,V_33,V_35,equal(V_35,V_33)) }
  (4 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ subtract(V_32,cons(V_33,V_34)) -> subtract(delete(V_33,V_32),V_34),
  subtract(V_32,nil) -> V_32 } (2 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[nil] = 0;
```

```
[cons](X0,X1) = X1 + 1;
[false] = 0;
[true] = 0;
[equal](X0,X1) = 0;
[delete](X0,X1) = X1;
[subtract](X0,X1) = X0;
[case8](X0,X1,X2,X3) = X0 + 1;
['subtract`](X0,X1) = X1;

Termination proof found.

Checking module:
{ append(cons(V_33,V_34),V_32) -> cons(V_33,append(V_34,V_32)) } (1 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[cons](X0,X1) = X1 + 1;
[append](X0,X1) = X0;
['append`](X0,X1) = X0;

Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_adduniq(nil,V_32) -> V_32 } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ record_extract(tuple(mcrlrecord,tuple(lock,tuple(V_17,
  tuple(V_18,tuplenil(V_19))))),lock,resource)
  -> tuple(mcrlrecord,tuple(lock,tuple(V_17,tuple(V_20,tuplenil(V_19))))) }
  (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_check_available(V_29,nil) -> false,
  locker2_check_available(V_29,cons(V_27,V_28)) ->
  case6(V_28,V_27,V_29,equal(V_29,record_extract(V_27,lock,resource))),
```

```
  case6(V_28,V_27,V_29,false) -> locker2_check_available(V_29,V_28),
  case6(V_28,V_27,V_29,true) ->
  andt(equal(record_extract(V_27,lock,excl),nil),
       equal(record_extract(V_27,lock,pending),nil)) }
  (4 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_check_availables(nil,V_28) -> true,
  locker2_check_availables(cons(V_29,V_30),V_28) ->
  andt(locker2_check_available(V_29,V_28),locker2_check_availables(V_30,V_28)) }
  (2 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[tuple](X0,X1) = 0;
[tuplenil](X0) = 0;
[lock] = 0;
[mcrlrecord] = 0;
[nil] = 0;
[pending] = 0;
[cons](X0,X1) = X1 + 1;
[resource] = 0;
[record_extract](X0,X1,X2) = 0;
[false] = 0;
[true] = 0;
[equal](X0,X1) = 0;
[excl] = 0;
[andt](X0,X1) = 0;
[case6](X0,X1,X2,X3) = 0;
[locker2_check_availables](X0,X1) = 0;
[locker2_check_available](X0,X1) = 0;
['locker2_check_availables`](X0,X1) = X0;

Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ case5(V_31,V_28,V_27,true) ->
  andt(locker2_obtainable(V_27,V_31),locker2_obtainables(V_28,V_31)),
```

```
  case5(V_31,V_28,V_27,false) -> locker2_obtainables(V_28,V_31),
  locker2_obtainables(cons(V_27,V_28),V_31) ->
  case5(V_31,V_28,V_27,member(V_31,record_extract(V_27,lock,pending))),
  locker2_obtainables(nil,V_31) -> true } (4 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 2 constraints)
[tuple](X0,X1) = 0;
[tuplenil](X0) = 0;
[lock] = 0;
[mcrlrecord] = 0;
[nil] = 0;
[pending] = 0;
[cons](X0,X1) = X1 + 1;
[resource] = 0;
[record_extract](X0,X1,X2) = 0;
[false] = 0;
[true] = 0;
[equal](X0,X1) = 0;
[member](X0,X1) = 0;
[locker2_obtainable](X0,X1) = 0;
[andt](X0,X1) = 0;
[locker2_obtainables](X0,X1) = 0;
[case9](X0,X1,X2,X3) = 0;
[case5](X0,X1,X2,X3) = 0;
['locker2_obtainables`](X0,X1) = X0;
['case5`](X0,X1,X2,X3) = X1;

Termination proof found.

Checking module:
{ case4(V_31,V_27,V_40) -> false } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_map_claim_lock(nil,V_30,V_31) -> nil,
  locker2_map_claim_lock(cons(V_27,V_28),V_30,V_31) ->
  cons(locker2_claim_lock(V_27,V_30,V_31),
  locker2_map_claim_lock(V_28,V_30,V_31)) }
```

```
  (2 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[nil] = 0;
[cons](X0,X1) = X1 + 1;
[locker2_claim_lock](X0,X1,X2) = 0;
[locker2_map_claim_lock](X0,X1,X2) = X0;
['locker2_map_claim_lock`](X0,X1,X2) = X0;

Termination proof found.

Checking module:
{ record_update(tuple(mcrlrecord,tuple(lock,tuple(V_17,tuple(V_18,
tuplenil(V_19))))),lock,pending,V_20)
  -> tuple(mcrlrecord,tuple(lock,tuple(V_17,tuple(V_18,tuplenil(V_20))))) }
  (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ record_updates(V_21,V_22,nil) -> V_21,
  record_updates(V_21,V_22,cons(tuple(V_23,tuplenil(V_20)),V_24)) ->
  record_updates(record_update(V_21,V_22,V_23,V_20),V_22,V_24) } (2 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[tuple](X0,X1) = 0;
[tuplenil](X0) = 0;
[lock] = 0;
[mcrlrecord] = 0;
[nil] = 0;
[pending] = 0;
[cons](X0,X1) = X1 + 1;
[record_update](X0,X1,X2,X3) = 0;
[record_updates](X0,X1,X2) = X0;
['record_updates`](X0,X1,X2) = X2;

Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.
```

```
Checking module:
{ case2(V_31,V_27,true) ->
  record_updates(V_27,lock,cons(tuple(excllock,excl),nil)) } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_release_lock(V_27,V_31) ->
  case2(V_31,V_27,gen_modtageq(V_31,record_extract(V_27,lock,excl))) }
  (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ case1(V_31,V_30,V_27,true) ->
  record_updates(V_27,lock,cons(tuple(pending,tuplenil(append(
  record_extract(V_27, lock, pending), cons(V_31,nil)))),nil)),
  case1(V_31,V_30,V_27,false) -> V_27 } (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ case0(V_31,V_27,V_39) -> V_27,
  case0(V_31,V_27,cons(V_31,V_26)) ->
  record_updates(V_27,lock,cons(tuple(excl,tuplenil(V_31)),
                          cons(tuple(pending, tuplenil(V_26)),nil))) }
  (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_promote_pending(V_27,V_31) ->
  case0(V_31,V_27,record_extract(V_27,lock,pending)) } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_map_promote_pending(cons(V_27,V_28),V_25) ->
  cons(locker2_promote_pending(V_27,V_25),
       locker2_map_promote_pending(V_28,V_25)),
  locker2_map_promote_pending(nil,V_25) -> nil } (2 rules)
```

```
checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[tuple](X0,X1) = 0;
[tuplenil](X0) = 0;
[lock] = 0;
[mcrlrecord] = 0;
[nil] = 0;
[pending] = 0;
[cons](X0,X1) = X1 + 1;
[resource] = 0;
[record_extract](X0,X1,X2) = 0;
[record_update](X0,X1,X2,X3) = 0;
[record_updates](X0,X1,X2) = X0;
[excl] = 0;
[case0](X0,X1,X2) = X1;
[locker2_promote_pending](X0,X1) = X0;
[locker2_map_promote_pending](X0,X1) = X0;
['locker2_map_promote_pending`](X0,X1) = X0;

Termination proof found.

Checking module:
{ locker2_add_pending(V_27,V_30,V_31) ->
  case1(V_31,V_30,V_27,member(record_extract(V_27,lock,resource),V_30)) }
  (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_remove_pending(V_27,V_31) ->
  record_updates(V_27,lock, cons(tuple(pending, tuplenil(subtract(
  record_extract(V_27, lock, pending), cons(V_31,nil)))),nil)) }
  (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ locker2_map_add_pending(nil,V_30,V_31) -> nil } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ record_new(lock) ->
  tuple(mcrlrecord,
```

```
        tuple(lock,tuple(undefined,tuple(nil,tuplenil(nil))))) }
  (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ element(int(s(0)),tuplenil(V_14)) -> V_14,
  element(int(s(0)),tuple(V_14,V_15)) -> V_14,
  element(int(s(s(V_10))),tuple(V_14,V_15)) ->
  element(int(s(V_10)),V_15) }
  (3 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[0] = 0;
[s](X0) = 0;
[int](X0) = 0;
[tuple](X0,X1) = X1 + X0 + 1;
[tuplenil](X0) = X0;
[element](X0,X1) = X1;
['element`](X0,X1) = X1;

Termination proof found.

Checking module:
{ if(F,V_1,V_2) -> V_2,
  if(T,V_1,V_2) -> V_1 } (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ eq(T,T) -> T,
  eq(F,F) -> T,
```

```
  eq(T,F) -> F,
  eq(F,T) -> F } (4 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ not(F) -> T,
  not(T) -> F } (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ imp(T,V_0) -> V_0,
  imp(F,V_0) -> T } (2 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ or(F,F) -> F,
  or(T,F) -> T,
  or(F,T) -> T,
  or(T,T) -> T } (4 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Execution time: 2.870000 sec
- : unit = ()
CiME>
```

Remarque 24. — Avec l'approche globale, aucune solution n'a été trouvée en ne cherchant que des interpétations linéaires. Le gain des modules est donc significatif quant à la nature des interprétations.

Nous présentons dans le tableau V.1 la comparaison en matières de temps et de place mémoire avec une tentative de preuve par approche globale utilisant paires et graphes de dépendance.

Recherche	Temps	Mémoire
Globale	> 10 jours	> 92,4Mo
Modulaire	2,8s	2,6Mo

TAB. V.1 Comparaison en temps et place nécessaires à la preuve de terminaison.

Chapitre VI

Conclusion

Nous avons défini, en introduisant la notion de *modules de récriture* (déf. 58), un cadre général particulièrement adapté à l'étude de la structure intrinsèquement hiérarchique des systèmes de récriture.

Les applications à la preuve de terminaison sont nombreuses. L'étude modulaire permet en effet une approche incrémentale de la preuve. Nous obtenons en outre grâce aux *paires de dépendance relatives* construites à partir des modules des critères puissants tant dans le cas standard (théorèmes 18 et 19 et corollaires) que dans le cas de la récriture modulo associativité et commutativité (théorèmes 27 et 28).

L'approche par modules combine en fait plusieurs avantages. Tout d'abord notre politique consistant à affaiblir la notion de terminaison jusqu'à la terminaison $\mathcal{C}_\mathcal{E}$, pour mieux alléger les contraintes sur les constituants des hiérarchies, et la généralité du concept de modules nous permettent d'exprimer directement des résultats antérieurs (Kurihara & Ohuchi [52]) comme de subsumer des méthodes dont les prémisses sont plus contraignantes au niveau de la nature des extensions ou encore sur une stratégie particulière à appliquer. Pour ne citer que deux exemples : les extensions propres sont en fait des extensions de modules très contraintes ; la restriction de la méthode des paires de dépendance relatives au cas de la récriture innermost permet quant à elle de retrouver des résultats de Arts & Giesl [5] qui en deviennent donc un cas particulier.

Les contraintes sur les ordres obtenues à partir d'une analyse modulaire sont moins nombreuses à chaque étape que dans le cas d'une analyse globale puisque seules les règles pertinentes entrent en jeu. Elles sont également moins sévères, pour la même raison mais aussi grâce à l'utilisation de paires relatives où, là encore, la généralité du cadre d'étude permet l'application d'optimisations, en particulier par graphes de dépendance. À la fois plus faibles et moins nombreuses ces contraintes de terminaison sont donc plus faciles à résoudre. Cette aisance est sensible au niveau de la simplicité des interprétations (ou des précédences) recherchées. La preuve de terminaison du système d'arithmétique de Peàno associatif et commutatif à l'aide d'interprétations polynomiales (exemple 51) en est une illustration. Enfin, le découpage en modules ainsi que les critères par paires relatives sont implantables et favorisent donc une recherche automatique des preuves, la simplicité des ordres convenables pour ces méthodes ajoutant alors l'efficacité à l'automatisation. Ces résultats ont été implantés au sein du système C*i*ME2. Des preuves de terminaison sont susceptibles d'être trouvées de façon totalement automatique en utilisant l'approche modulaire : la réduction du temps passé par le système à résoudre les contraintes est spectaculaire.

C*i*ME2 permet ainsi de traiter des systèmes volumineux, issus par exemple d'outils de spécifications de processus communicants comme μ-CRL et dont la terminaison peut servir à garantir des propriétés de progression. Un système de ce type comporte aisément un millier de règles, son éventuelle terminaison peut dorénavant être prouvée automatiquement en quelques secondes.

Bibliographie

[1] Siva Anantharaman. REVEAL: a users' guide. Rapport de recherche, Laboratoire d'Informatique Fondamentale d'Orléans, 1993.

[2] T. Arts et J. Giesl. Automatically proving termination where simplification orderings fail. In Michel Bidoit et Max Dauchet, éditeurs, *Theory and Practice of Software Development*, volume 1214 des *Lecture Notes in Computer Science*, Lille, France, avril 1997. Springer-Verlag.

[3] Thomas Arts. *Automatically proving termination and innermost normalisation of term rewriting systems.* Thèse de doctorat, Universiteit Utrecht, 1997.

[4] Thomas Arts et Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[5] Thomas Arts et Jürgen Giesl. Modularity of termination using dependency pairs. In Nipkow [65], pages 226–240.

[6] Leo Bachmair et Nachum Dershowitz. Commutation, transformation, and termination. In Siekmann [80], pages 5–20.

[7] Leo Bachmair et David A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1(4):329–349, décembre 1985.

[8] Ahlem Ben Cherifa et Pierre Lescanne. An actual implementation of a procedure that mechanically proves termination of rewriting systems based on inequalities between polynomial interpretations. In Siekmann [80], pages 42–51.

[9] R. Bündgen. Reduce the redex $\rightarrow$ ReDuX. In Kirchner [49], pages 446–450.

[10] Ahlem Ben Cherifa, Isabelle Gnaedig et Pierre Lescanne. Les outils de preuve de terminaison dans REVE. CRIN Nancy, janvier 1986.

[11] Ahlem Ben Cherifa et Pierre Lescanne. A method for proving termination of rewriting systems basd on elementary computations on polynomials. Greco de Programmation, 1986. 86-R-044.

[12] Evelyne Contejean et Claude Marché. CiME: Completion Modulo E. In Harald Ganzinger, éditeur, *7th International Conference on Rewriting Techniques and Applications*, volume 1103 des *Lecture Notes in Computer Science*, pages 416–419, New Brunswick, NJ, États-unis, juillet 1996. Springer-Verlag. Description du système disponible à `http://cime.lri.fr/`.

[13] Evelyne Contejean, Claude Marché, Benjamin Monate et Xavier Urbain. Cime version 2, 2000. Version préliminaire disponible à `http://cime.lri.fr/`.

[14] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, École Normale Supérieure de Lyon, 1998.

[15] Max Dauchet. Simulation of Turing machines by a regular rewrite rule. *Theoretical Computer Science*, 103(2):409–420, septembre 1992.

[16] M. Davis, Y. Matijasevic et J. Robinson. Diophantine equations: Positive aspects of a negative solution, 1976.

[17] Catherine Delor. Extension of the associative path ordering to a chain of associative commutative symbols, 1992. Non publié.

[18] Catherine Delor et Laurence Puel. Extension of the associative path ordering to a chain of associative-commutative symbols. In Kirchner [49], pages 389–404.

[19] Nachum Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, novembre 1979.

[20] Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, mars 1982.

[21] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, février 1987.

[22] Nachum Dershowitz. 33 examples of termination. In Hubert Comon et Jean-Pierre Jouannaud, éditeurs, *Term Rewriting*, volume 909 des *Lecture Notes in Computer Science*, pages 16–26. French Spring School of Theoretical Computer Science, Springer-Verlag, 1995.

[23] Nachum Dershowitz. Hierarchical termination. In N. Dershowitz et N. Lindenstrauss, éditeurs, *Proceedings of the Fourth International Workshop on Conditional and Typed Rewriting Systems (Jerusalem, Israel, July 1994)*, volume 968, pages 89–105, Berlin, 1995. Springer-Verlag.

[24] Nachum Dershowitz et Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, août 1979.

[25] K. Drosten. *Termerssetzungssysteme*. Thèse de doctorat, Université de Passau, Allemagne, 1989. En allemand.

[26] Maribel Fernandez et Jean-Pierre Jouannaud. Modular termination of term rewriting systems revisited. In Egidio Astesiano, Gianna Reggio et Andrzej Tarlecki, éditeurs, *Recent Trends in Data Type Specification*, volume 906 des *Lecture Notes in Computer Science*, pages 255–272, S. Margherita, Italie, 1994. Springer-Verlag.

[27] Maria C. F. Ferreira et Hans Zantema. Dummy elimination: Making termination easier. In *Fundamentals of Computation Theory*, pages 243–252, 1995.

[28] M.C.F. Ferreira et H. Zantema. Total termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 7(2):133–162, 1996.

[29] J. Giesl et E. Ohlebusch. Pushing the frontiers of combining rewrite systems farther outwards. In *Proceedings of the Second International Workshop on Frontiers of Combining Systems (FroCoS '98)*, volume 7 des *Studies in Logic and Computation*, pages 141–160, Amsterdam, Pays-Bas, 2000. Research Studies Press, John Wiley & Sons.

[30] Jürgen Giesl et Deepak Kapur. Dependency pairs for equational rewriting. In Aart Middeldorp, éditeur, *12th International Conference on Rewriting Techniques and Applications*, volume 2051 des *Lecture Notes in Computer Science*, pages 93–108, Utrecht, Pays-Bas, mai 2001. Springer-Verlag.

[31] Jürgen Giesl. Generating polynomial orderings for termination proofs. In Jieh Hsiang, éditeur, *6th International Conference on Rewriting Techniques and Applications*, volume 914 des *Lecture Notes in Computer Science*, pages 426–431, Kaiserslautern, Allemagne, avril 1995. Springer-Verlag.

[32] Jurgen Giesl. Polo - a system for termination proofs using polynomial orderings. Rapport technique IBN95-24, Technische Hochschule Darmstadt, Alexanderstr. 10, 64283 Darmstadt, Allemagne, 1995.

[33] Jurgen Giesl et Aart Middeldorp. Eliminating dummy elimination. In *Proceedings of the 17th Conference on Automated Deduction*, volume 1831 des *Lecture Notes in Artificial Intelligence*, pages 309–323, Pittsburgh, États-unis, 2000. Springer-Verlag.

[34] Rajeev Goré, Alexander Leitsch et Tobias Nipkow, éditeurs. *First International Joint Conference on Automated Reasoning*, volume 2083 des *Lecture Notes in Artificial Intelligence*, Sienne, Italie, juin 2001. Springer-Verlag.

[35] René Goscinny et Albert Uderzo. *Astérix le gaulois*, volume 1 des *Aventures d'Astérix le gaulois*. Dargaud, 1961.

[36] B. Gramlich. A structural analysis of modular termination of term rewriting systems. Rapport de recherche SR-91-15, Universität Kaiserslautern, 1991.

[37] Bernhard Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 5:131–158, 1994.

[38] Bernhard Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticæ*, 24:3–23, 1995.

[39] Bernhard Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems.* Thèse de doctorat, Universität Kaiserslautern, 1996.

[40] J.F. Groote et A. Ponse. Syntax and semantics of μ-CRL. In C. Verhoef et S.F.M. van Vlijmen, éditeurs, *Algebra of Communicating Processes*, pages 26–62. Springer-Verlag, 1995.

[41] Haskell, a purely functional language. `http://www.haskell.org/`.

[42] David Hilbert. Über die Theorie der Algebraischen formen. *Math. Ann.*, 36:473–534, 1890.

[43] H. Hong et D. Jakus. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21:23–38, 1998.

[44] Gérard Huet et Dallas S. Lankford. On the uniform halting problem for term rewriting systems. Rapport de recherche 283, INRIA, mars 1978.

[45] K.E. Madlener J. Avenhaus et J. Steinbach. Comtes - an experimental environment for the completion of term rewriting systems. In Nachum Dershowitz, éditeur, *Proc. 3rd Rewriting Techniques and Applications*, volume 355 des *Lecture Notes in Computer Science*, pages 542–546. RTA-89, Chapel Hill, avril 1989. Springer-Verlag.

[46] Jean-Pierre Jouannaud et Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4), novembre 1986.

[47] S. Kamin et Jean-Jacques Lévy. Two generalizations of the recursive path ordering. Disponible comme rapport de recherche du department of computer science, University of Illinois, Urbana-Champaign, 1980.

[48] Deepak Kapur et H. Zhang. An overview of the rewrite rule laboratory (RRL). In Nachum Dershowitz, éditeur, *Proc. 3rd Rewriting Techniques and Applications*, volume 355 des *Lecture Notes in Computer Science*, pages 559–563. RTA-89, Chapel Hill, avril 1989. Springer-Verlag.

[49] Claude Kirchner, éditeur. *5th International Conference on Rewriting Techniques and Applications*, volume 690 des *Lecture Notes in Computer Science*, Montreal, Canada, juin 1993. Springer-Verlag.

[50] Mahahito Kurihara et Azuma Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. *Theoretical Computer Science*, 103:273–282, 1992.

[51] Masahito Kurihara et Azuma Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. Rapport technique SF-36, Hokkaido University, Sapporo, Japon, 1990.

[52] Masahito Kurihara et Azuma Ohuchi. Decomposable termination of composable term rewriting systems. *IEICE*, E78–D(4):314–320, avril 1995.

[53] Keiichirou Kusakari. *Termination, AC-Termination and Dependency Pairs of Term Rewriting Systems*. Thèse de doctorat, JAIST, 2000.

[54] Keiichirou Kusakari, Masaki Nakamura et Yoshihito Toyama. Argument filtering transformation. In Gopalan Nadathur, éditeur, *Principles and Practice of Declarative Programming, International Conference PPDP'99*, volume 1702 des *Lecture Notes in Computer Science*, pages 47–61, Paris, 1999. Springer-Verlag.

[55] Keiichirou Kusakari et Yoshihito Toyama. On proving AC-termination by AC-dependency pairs. *IEICE Transactions on Information and Systems*, E84-D(5):604–612, 2001.

[56] Dallas S. Lankford. On proving term rewriting systems are Noetherian. Rapport technique MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979.

[57] Claude Marché. *Réécriture modulo une théorie présentée par un système convergent et décidabilité des problèmes du mot dans certaines classes de théories équationnelles*. Thèse de doctorat, Université Paris-sud, Orsay, France, octobre 1993.

[58] Claude Marché et Xavier Urbain. Termination of associative-commutative rewriting by dependency pairs. In Nipkow [65], pages 241–255.

[59] J. Matijasevic. Enumerable sets are diophantic, 1970.

[60] Aart Middeldorp. *Modular Properties of Term Rewriting Systems*. Thèse de doctorat, Vrije Universiteit, Amsterdam, Pays-Bas, 1990.

[61] Aart Middeldorp. Approximating dependency graphs using tree automata techniques. In Goré et al. [34], pages 593–610.

[62] Aart Middeldorp et Yoshihito Toyama. Completeness of combinations of constructor systems. Rapport technique R9058, CWI, octobre 1990.

[63] Aart Middeldorp et Yoshihito Toyama. Completeness of combinations of constructor systems. *Journal of Symbolic Computation*, 15(3):331–348, 1993.

[64] Robin Milner, Mads Tofte et Robert W. Harper. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, États-unis, 1997.

[65] Tobias Nipkow, éditeur. *9th International Conference on Rewriting Techniques and Applications*, volume 1379 des *Lecture Notes in Computer Science*, Tsukuba, Japon, avril 1998. Springer-Verlag.

[66] Pilar Nivela, Robert Nieuwenhuis et Albert Rubio. Terminationlab, 2000. Disponible à `http://www.lsi.upc.es/~albert/`.

[67] The objective caml language. `http://caml.inria.fr/`.

[68] Michael J. O'Donnell. Computing in systems described by equations. In *Lecture Notes in Computer Science*, volume 58. Springer, Berlin, Allemagne, 1977.

[69] Enno Ohlebusch. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, 136:333–360, 1994.

[70] Enno Ohlebusch. Modular properties of composable term rewriting systems. *Journal of Symbolic Computation*, 20:1–41, 1995.

[71] Enno Ohlebusch, Claus Claves et Claude Marché. TALP: A tool for the termination analysis of logic programs. In Leo Bachmair, éditeur, *11th International Conference on Rewriting Techniques and Applications*, volume 1833 des *Lecture Notes in Computer Science*, pages 270–273, Norwich, Royaume Uni, juillet 2000. Springer-Verlag. Disponible à `http://bibiserv.techfak.uni-bielefeld.de/talp/`.

[72] Gerald E. Peterson et Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, avril 1981.

[73] David A. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Rapport de recherche R-78-943, University of Illinois, Urbana, IL, États-unis, 1978.

[74] M. R. K. Krishna Rao. Simple termination of hierarchical combinations of term rewriting systems. In *Theoretical Aspects of Computer Software*, volume 789 des *Lecture Notes in Computer Science*, pages 203–223. 1994. Springer-Verlag.

[75] M.R.K. Krishna Rao. Modular proofs for completeness of hierarchical term rewriting systems. *Theoretical Computer Science*, 151:487–512, 1995.

[76] Barry K. Rosen. Tree-manipulating systems et church-rosser theorems. *J. of the Association for Computing Machinery*, 20(1):160–187, janvier 1973.

[77] Albert Rubio. Extension orderings. In *Proceedings of the 22nd International Colloquium on Automata, Languages, and Programming*, volume 944 des *Lecture Notes in Computer Science*, pages 511–522, Szeged, Hongrie, 1995. Springer-Verlag.

[78] Albert Rubio. A fully syntactic ac-rpo. In Paliath Narendran et Michaël Rusinowitch, éditeurs, *Rewriting Techniques and Applications*, volume 1631 des *Lecture Notes in Computer Science*, pages 133–147, Trente, Italie, Juillet 1999. Springer-Verlag.

[79] Michaël Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26:65–70, 1987.

[80] Jörg H. Siekmann, éditeur. *8th International Conference on Automated Deduction*, volume 230 des *Lecture Notes in Computer Science*, Oxford, Angleterre, 1986. Springer-Verlag.

[81] Joachim Steinbach. Proving polynomials positive. In R. Shyamasundar, éditeur, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 des *Lecture Notes in Computer Science*, pages 191–202, New Delhi, Inde, décembre 1992. Springer-Verlag.

[82] R.E. Tarjan. Depth-rst search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[83] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25:141–143, avril 1987.

[84] Xavier Urbain. Automated incremental termination proofs for hierarchically defined terms rewriting systems. In Goré et al. [34], pages 485–498.

[85] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticæ*, 24:89–105, 1995.

Index

$=_{\mathrm{AC}}$, 91, 92
G, 52
$I(\sigma)$, 74
$I(x)$, 73
$I^{\mathrm{AC}}(x)$, 122
I_t, 114
$T_\infty(\mathcal{F}, X)$, 73
X_t, 114
$\mathit{Head}(t)$, 110
$\mathrm{Comb}(E)$, 73, 122
$\Lambda(t)$, 16
$\mathit{Par}(t)$, 109
$\mathit{Par}_\downarrow(t)$, 110
$\mathrm{Red}(t)$, 73, 122
$\downarrow_P$, 24
$\longleftarrow$, 63
$\xrightarrow{m.e.}$, 110
π, 52
$m_t(x)$, 114
$s \xrightarrow{i} t$, 110
$s \xrightarrow{o} t$, 110
APO, 98
CAP, 36
REN, 36
RPO, 38
$\mathcal{V}ar$, 16
$\mathcal{C}_\mathcal{E}$, 52
$[\mathcal{F} \mid R]$, 63
$\mathcal{P}os$, 16

Accessibilité, 28
AFS, 42
Allitération, 25
Anacoluthe, 32
Antiréflexive, 15
Antisymétrique, 15
Aplatissement, 92
Approximation, 36–37
Arité, 16
Arrêt, 28

Branchement, 22

Chaîne de dépendance
  AC marquée, 111
  d'un module, 70
  minimale, 70
Chaîne de dépendance, 31, 32
Composition
  lexicographique, 19
Condition
  APO, 97
  MAPO, 100
Conditions de dépendance, 111
Confluence, 24
Contexte, 17
Corps, 110

Équivalence, 15
Extension, 63
  lexicographique, 20
  multiensemble, 20
  propre, 54
  sur constructeurs, 55
Extensions
  propres généralisées, 55
Externe, 110

Filtrage, 17
  étendu, 94
Forme normale, 23

Graphe de dépendance
    de modules, 84
Graphe de dépendance, 34–37
    approché, 36

Hiérarchique, 63

Indépendamment, 63
Indécidabilité, 28
Instance, 17
Interne, 110
Interprétation
    arithmétique, 40–42
    homomorphique, 38
    polynomiale, 40–41

Lexicographique, 68
Linéarité, 22
    à droite, 22
    à gauche, 22

Marquage, 31
    partiel, 109
Marque, 31
    Effacement de $\sim$, 110
ModifiedPairs, 119
Modulaire, 50
Module
    AC, 121
Modules, 63–65, 121
    Décomposition en $\sim$, 64–65
    minimaux, 65
Multiensemble, 20
Multiplicité, 20, 114
Multiplicité, 20, 114

Normalisation, 23
    forte, 23

Ordre, 15
    associé, 67
    bien fondé, 16
    Construction d'$\sim$, 67–68
    d'évaluation, 39
    d'extension, 53
    de récriture, 28
    de réduction, 19
    de simplification, 19
    π-extensible, 67
    sémantique, 38
    strict, 16
    sur les termes, 18
    sur un chemin, 37
    syntaxique, 37–38
Orthogonal, 24

Paire critique, 24
Paires de dépendance, 30–37
    AC marquées, 111
    de module, 69
    étendues AC, 103
    marquées, 31
    relatives, 69–70
    relatives AC, 121
Pairs, 116
Partie
    fortement connexe, 56
    large, 16
    stricte, 16
    stricte stable, 19
Peterson & Stickel, 93
π-extensible, 67
    Construction de $\sim$, 67–68
Plongement, 29
Polynôme AC, 96
Position, 16
Préordre, 15
Précédence, 38
Projection, 23

Racine, 16
Radical, 21
Récriture, 21
    AC étendue, 93
    AC sur aplatis, 94
    modulo AC, 92

Réflexive, 15
Règle, 21
  linéaire, 22
  utilisable, 58
Relation
  antiréflexive, 15
  antisymétrique, 15
  binaire, 15
  d'équivalence, 15
  d'ordre, 15
  de récriture, 21
  noethérienne, 15
  réflexive, 15
  symétrique, 15
  transitive, 15
RPO, 38
RPS, 42, 68

Signature, 16
Split, 115
Statut, 38
Substitution, 17
  Support de ~, 17
Symétrique, 15
Symbole, 16
  AC, 91
  constructeur, 21
  défini, 21
  libre, 91
  marqué, 31
Système
  confluent, 24
  de récriture, 21
  dupliquant, 22
  filtre, 42
  orthogonal, 24
  overlapping, 24
  plat, 55
  projectif, 23
Systèmes
  composables, 49, 63
  à constructeurs partagés, 49
  disjoints, 49

Tête, 110
Terme, 16
  clos, 16
  connectable, 36
  linéaire, 22
  normalisable, 23
  partiellement marqué, 109
  plat, 55
  Sous-~, 16
  variadique, 91
Terminaison, 23
  DP-quasi simple, 57
  DP-simple, 56
Toyama, 37, 51
Transitive, 15
Turing, 27

Unification, 17
Union
  de composables, 49, 63
  à constructeurs partagés, 49, 63
  disjointe, 49, 63
  hiérarchique, 49

Variable, 16

Zeugma, 47