UNIVERSITE DES SCIENCES ET TECHNIQUES
DE LILLE FLANDRE ARTOIS

ANNEE 1991

THESE

présentée par
Aline DERUYVER

en vue de l'obtention du titre de Docteur en Informatique

Deux aspects de la réécriture:
Un laboratoire pour les automates:
VALERIAN
Un système de démonstration automatique de théorèmes:
EMMY

JURY:

PRESIDENT: Mr DAUCHET M.
RAPPORTEURS: Mr DELAHAYE J.P.
Mr GALLIER J.
MEMBRES: Mr DEVIENNE P.
Mr COMON H.

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

REMERCIEMENTS

A Max DAUCHET
Professeur à l'université des sciences et techniques de Lille
Tu m'as fait l'honneur de diriger ce travail de thèse et tu m'as familiarisée avec certains aspects théoriques de l'informatique.
Trouve ici l'expression de ma gratitude et de mon amitié.

A Jean-Paul DELAHAYE
Professeur à l'université des sciences et techniques de Lille
Tu t'es toujours interessé à mon travail et tu as su me donner plus confiance en moi en me permettant d'enseigner.
Trouve ici l'expression de mes remerciements et de mon amitié.

A Jean GALLIER
Professeur à l'université de Pensylvanie (Philadelphie USA)
Directeur de recherche chez DEC.
Vous avez accepté de juger ce travail. Trouvez ici l'expression de mes remerciements.

A Philippe DEVIENNE
Chargé de recherche au CNRS
C'est avec plaisir que je te côtoyais à Lille. Tu as accepté de juger ce travail, trouve ici l'expression de mes remerciements et de ma sympathie.

A Hubert COMON
Chargé de recherche au CNRS
Tu as accepté de juger ce travail, trouve ici l'expression de mes remerciements.

Je remercie également Leo Bachmair, professeur à l'université de Stony Brook (USA), pour m'avoir gentiment accueillie dans son laboratoire pendant un an.
Enfin je remercie Harald Ganzinger, professeur à l'université de Dortmund (Allemagne) avec qui j'ai eu d'intéressantes et sympathiques discussions.

A mes parents, à qui je dédie cette thèse. Vous m'avez toujours soutenue tout au long de mes études et vous êtes pour beaucoup dans ma réussite.
A Yann, mon compagnon de vie. Tout au long de l'élaboration de ce travail, tu as toujours été là pour m'encourager et me donner confiance en moi.

Sommaire

Introduction

Dans ce mémoire nous étudions deux aspects de la réécriture. Le premier travail, réalisé au LIFL à Lille I est une contribution à l'étude structurelle de certaines classes de réécriture. Nous montrons comment certains problèmes d'accessibilité se traitent à la lumière des automates finis d'arbres.
Le second travail a été réalisé à l'université de Stony Brook (New York). Il consiste en la mise en oeuvre d'un système de démonstration automatique de théorèmes basé sur un calcul par superposition et fonctionnant pour des clauses logiques du premier ordre avec équations.

****

Dans cette introduction nous allons tout d'abord passer brièvement en revue les différents concepts de base utilisés dans ces travaux. Ensuite nous brosserons un rapide tableau des différents systèmes développés, respectivement Valerian et Emmy.

Notre domaine d'intérêt concernant les systèmes de réécriture est au coeur de la spécification algébrique et de la preuve de théorèmes. Sans entrer dans les détails, si un type ou une théorie est spécifié par un ensemble d'égalités entre termes, on obtient des règles de réécriture (c'est à dire de calcul) en orientant ces égalités. On peut alors espérer en déduire un procédé de calcul de formes normales (complétion à la Knuth-Bendix). On peut ainsi considérer les interpréteurs comme des systèmes de réécriture particuliers. La forme normale d'un programme et de son jeu de données est alors le résultat.

Cette formalisation permet d'énoncer certaines notions très naturelles qui permettent de mieux appréhender les propriétés d'un système de réécriture. Citons par exemples les notions de confluence, de terminaison ou d'accessibilité.

- La confluence des systèmes de réécriture (c'est à dire que "toutes les façons de calculer mènent au même résultat"), traduit l'indépendance du comportement d'un programme (du point de vue de l'utilisateur) par rapport à l'implantation.

- la terminaison des systèmes de réécriture (c'est à dire la terminaison des calculs). Ce problème est important notamment dans la démonstration automatique de théorèmes. Pour établir cette terminaison, il est souvent judicieux de définir un ordre de réduction sur les termes, c'est à dire un ordre de réécriture bien fondé. Il existe différents types d'ordres de réduction tels que les ordres de chemins récursifs, dont l'ordre de chemin lexicographique et l'ordre de chemin sur les multi-ensembles (Dershowitz [14])

- Les problèmes de l'accessibilité qui se posent à deux niveaux:
  L'accessibilité du 1er ordre consistant à décider pour un système de réécriture donné S et deux termes M et N, si M peut se transformer en N en appliquant les règles de S.
  L'accessibilité du "second ordre" c'est à dire est-ce-que telle sorte de donnée peut donner telle sorte de résultat? (Nous utilisons le terme sorte au sens des algèbres sortées).

Concernant le problème de l'accessibilité, nous nous attacherons plus particulièrement aux systèmes de réécriture clos, le problème n'étant pas décidable dans le cas général.
Un système de réécriture est clos si et seulement si aucune variable n'apparaît dans les règles.
Soit le système de réécriture S suivant:

Règle 1: q1 -> q(q1) Règle 3: b(q(q1)) -> c(p1,p(p1),p1)
Règle 2: q(q1) -> q1 Règle 4: a(b1,a(q1,b1)) -> b(q1)

où a, b, c, q, q1, p, p1 sont des symboles de fonctions.
Ce système est un système de réécriture clos.

Les automates finis d'arbre introduits dans les années 1960 par Tatcher et Brainer [4] peuvent être vus comme un type particulier de système de réécriture clos. Dans ce cas les états peuvent être vus comme des sortes.

En effet soit la sorte Horn représentant les polynômes de Hörner telle que:

sorte Horn, Hmul
utilise Mon, Cst, Xfv
o p +: Mon x Cst -> Horn
+: Hmul x Cst -> Horn
*: Horn x Xfv -> Hmul
*: Hmul x Xfv -> Horn
*: Cst x Xfv -> Horn
*: Mon x Xfv -> Horn
*: Hmul x Xfv -> Hmul
*: Horn x Xfv -> Horn

où Mon représente la sorte monôme, Cst représente la sorte constante et Xfv représente la sorte variable.
l'automate H reconnaissant la sorte Horn aura la forme suivante:

+(Mon, Cst) -> Horn *(Cst, Xfv) -> Horn
+(Hmul, Cst) -> Horn *(Mon, Xfv) -> Horn
*(Horn, Xfv) -> Hmul *(Hmul, Xfv) -> Hmul
*(Hmul, Xfv) -> Horn *(Horn, Xfv) -> Horn

Prenons la règle +(Mon,Cst) -> Horn. Celle-ci représente aussi bien la signature de l'opérateur + qu'une règle de transition sur les états Mon, Cst et Horn.
Cette approche par les automates des algèbres multi-sortées ordonnées commence à être utilisée. (Voir dernière version de REVE à Nancy et d'OBJ3 à Orsay)

Nous allons maintenant examiner deux grandes méthodes directrices utilisées dans la démonstration automatique de théorèmes. La première, appelée surréduction est propre à l'utilisation des systèmes de réécriture. La seconde, appelée la E-unification consiste à utiliser les propriétés inhérentes aux équations lorsque les objets que l'on manipule se présentent sous cette forme.

La surréduction, aussi appelée "narrowing" ou superposition est une notion utilisée dans les systèmes s'inspirant de la méthode de complétion à la Knuth Bendix. Il s'agit, de façon approximative, étant donné un terme t et une règle li ->ri d'un système de réécriture R, de trouver les paires critiques entre t et li. Puis pour une paire critique donnée, de remplacer le sous terme u de t pouvant s'unifier avec li par ri.
Plus exactement on a la définition suivante:
Soit R un système de réécriture, un terme t est surréductible en t' à la position u, en utilisant la règle li -> ri ssi

- u est une position dans t qui ne correspond pas à une variable
- t à la position u et li sont unifiables.
- $t' = \sigma(t[u <- ri]) = \sigma(t)[u <- \sigma(ri)]$

avec σ l'unificateur minimal de t à la position u et li.
Exemple:
Soit R un système de réécriture tel que

1- g(0,x) -> x
2- f(g(x,y)) -> f(y)

Soit t=f(g(0,0))
t peut être superposé avec la partie gauche de la règle 2. On obtient donc,

$$\frac{f(g(0,0)) \qquad f(g(x,y)) \rightarrow f(y)}{f(0)}$$

avec $\sigma = \{x <- 0, y <- 0\}$
De même, si on prend la règle 1, on obtient:

$$\frac{f(g(0,0)) \qquad g(0,x) \rightarrow x}{f(0)}$$

avec $\sigma = \{x <- 0\}$
Cette approche a inspirée la définition des règles d'inférences utilisées dans Emmy, comme nous le verrons ultérieurement dans la thèse.

Voyons maintenant la deuxième méthode appelée E-unification. On dit que deux termes t et t' sont E-unifiables si et seulement si il existe une substitution θ telle que $\theta(t) == \theta(t')$ soit une conséquence logique de l'ensemble d'axiomes SP, c'est à dire $\theta(t) =_{SP} \theta(t')$. Une telle substitution θ qui résoud l'équation t == t' par rapport à $E\text{-}Mod_{SP}$ est dite E-unificateur de t et t'.
Exemple:
Soit SP=(S, E) telle que E={f(0,x) == x, h(f(x,y))=h(y)}.
Considérons l'équation h(0) == h(z), l'ensemble des substitutions

$D=\{\{z <- 0\}\} \cup \{\{z <- f(vi(f(vi-1, \ldots f(v0,0) \ldots)\} \mid i \geq 0\}$

est un ensemble de E-unificateurs des termes h(0) et h(z).

****

Ayant brièvement développé quelques concepts de base, nous allons présenter le premier système développé, nommé Valerian.
L'idée générale de ce système est la suivante:
A un système de réécriture clos on associe, grace à une compilation préalable, des couples d'automates finis représentant les liens. Nous utilisons pour cela une nouvelle classe de transformations d'arbres appelée GTT (Ground tree transducers) introduite récemment par

(Dauchet et Tison [11], et Dauchet, Heuillard, Lescanne, Tison [10]) pour répondre au problème posé par Huet [29] : La décidabilité de la confluence des systèmes de réécriture clos. Les GTT ont été depuis repris dans le cadre de l'étude algébrique des arbres (Courcelle [8] et [9], Fülop et Vagvölgyi [22], Raoult)

L'utilisation de ces outils permet une conception modulaire de nos algorithmes ainsi que l'utilisation de méthodes uniformes d'optimisation (réduction du non-déterminisme, minimalisation et tout autre algorithme "fini" sur les automates d'arbres). Par ailleurs l'approche algébrique possède en plus l'avantage de l'efficacité, alors que l'on pourrait s'attendre à ce qu'une telle approche soit peu puissante et ne donne que des algorithmes naïfs.

Cela nous a permis de donner une méthode de compilation des systèmes de réécriture qui permet de décider en temps quadratique ou même linéaire (selon les niveaux de compilation) du problème de l'accessibilité.

Reprenons le système de réécriture clos précédent afin d'expliciter les différentes étapes de notre compilation:

Règle 1: q1 -> q(q1) Règle 3: b(q(q1)) -> c(p1,p(p1),p1)
Règle 2: q(q1) -> q1 Règle 4: a(b1,a(q1,b1)) -> b(q1)

A ce système on associe le GTT B formé des deux automates finis suivants:

Automate ascendant G	Automate descendant D
Règle 1: q1-> i1	i1 -> q(e4) e4 -> q1
Règle 2: q1 -> e5 q(e5) -> i2	i2 -> q1
Règle 3: q1-> e7 q(e7) -> e8 b(e8) -> i3	i3 -> c(e9, e10, e11) e9 -> p1 e12 -> p1 e11 -> p1 e10 -> p(e12)
Règle 4: q -> e14 b1 -> e15 b1 -> e17 a(e14, e15) -> e16 a(e17, e16) -> i4	i4 -> b(e18) e18 -> q1

A partir de là, afin que ces automates simulent le système de réécriture de départ on génère des ε-transitions. On obtient alors le GTT B*. Ainsi, si on considère les règles 1 et 2 on obtient la nouvelle règle i1 -> i2.
De cette manière au lieu de réaliser les étapes suivantes:

i3 -> q(e4) -> q(q1) -> q(e5) -> i4

B* passera directement de i1 à i2.

Ces résultats améliorent ceux obtenus par Oyamaguchi [35] et Togashi-Noguchi (Voir Figure 1), et sont à comparer avec ceux obtenus tout récemment par Snyder [39] et Gallier [25] et [24] sur les théories équationnelles closes (qui sont un cas particulier de notre étude)

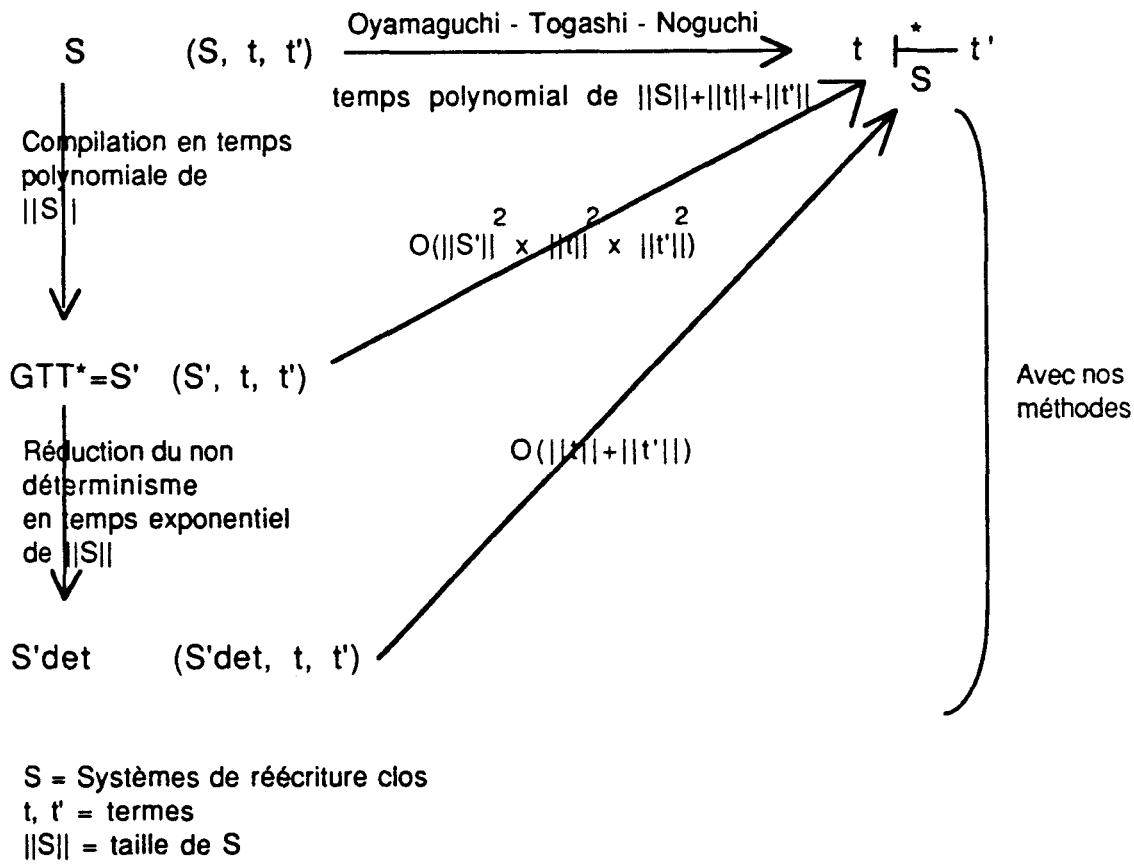


FIGURE 1

Il est possible d'étendre Valerian de façon à pouvoir manipuler des systèmes séparés sortés. Ces systèmes approximent des systèmes plus généraux, et permettent de donner des réponses plus où moins sûres à des problèmes posés par des systèmes de réécriture avec variables.
Prenons l'algorithme suivant:
Règle 1: +(*(x,y),*(z,y)) ->(+(x,z),y)
Règle 2: +(*(x,y),*(z',y)) -> *(+(x,z'),y)
Sur ces règles nous spécifions une sorte pour chaque variable:

$\tau(x)$=Horn, $\tau(y)$=Xfv, $\tau(z)$=Mon, $\tau(z')$=Cst.

Les sortes Horn, Mon, Xfv, et Cst, étant ensuite spécifiées sous forme d'automates on peut associer à l'algorithme précédent le système séparé sorté suivant:
Règle1: +(*(Horn,Xfv),*(Mon, Xfv)) -> *(+(Horn,Mon),Xfv)
Règle2: +(*(Horn,Xfv),*(Cst,Xfv)) -> *(+(Horn,Cst),Xfv)

Ce système pourra alors répondre aux types de questions suivantes:
Question du premier ordre:
Est-il possible que Horn3 $\in$ A(Npol3)?
où Npol3 = +(+(+(*(*(*(C,X),X),X),*(*(C,X),X)),*(C,X)),C)
et Horn3 = +(*(+(*(+(*(C,X),C),X),C),X),C)
Question du second ordre:
Est-il possible que $\text{Horn}_{\geq 3} \subseteq \text{Algoi}(\text{Npol}_{\geq 3})$?
où $\text{Npol}_{\geq 3}$=+(+(+(+(NPol,*(*(*(C,X),X),X)),*(*(C,X),X)),*(C,X)),C) et
$\text{Horn}_{\geq 3}$=+(*(+(*(+(*(Horn,X),C),X),C),X),C).

****

Etudions maintenant le second système appelé Emmy. Derrière la démonstration automatique se profile toujours la recherche d'un langage qui soit suffisamment puissant pour exprimer mieux une réalité complexe. Cette recherche se retrouve déjà chez Condillac [5] et les premiers logiciens avec l'utopie d'obtenir un langage sur le modèle des mathématiques permettant de valider tout jugement par un calcul. On peut citer le langage Prolog qui dérive partiellement de cette idée, à savoir représenter sous forme de clauses logiques les problèmes courant de la programmation. En dépit de sa puissance, Prolog ne permet pas toujours d'exprimer de manière aisée certains calculs pour lesquels la programmation fonctionnelle (Langage lisp et ses dérivés tels que ML) paraît plus naturelle. La réciproque est également vraie. De là on peut penser qu'un système qui serait capable de combiner les deux approches en un tout cohérent gagnerait dans la possibilité d'une représentation forte et concise des problèmes de programmation.
Par ailleurs, il s'agit d'obtenir une méthode de calcul consistante complète et efficace.
Le système Emmy dénote de cette recherche. Ce système est un système de démonstration automatique de théorèmes par réfutation basé sur un calcul par superposition décrit par Bachmair et Ganzinger [3] et fonctionne pour des clauses logiques du premier ordre avec équations.
On retrouve là quelque chose que l'on peut considérer comme la synthèse de deux approches: La logique du premier ordre (Prolog) et la logique équationnelle (OBJ où SBREVE)

De nombreuses techniques permettant de prouver des théorèmes dans les théories équationnelles ont été suggérées. Celle que nous avons retenue consiste à utiliser les équations s=t dans une seule direction, c'est à dire comme règle de réécriture s -> t. Cette méthode peut être très efficace comme le montre les nombreuses applications de la méthode de complétion de Knuth-Bendix. Ses deux principaux composants sont la superposition et la simplification par réécriture. Ici nous considérons une version restreinte de la paramodulation ordonnée, appelée aussi superposition stricte. La superposition stricte est complète par réfutation pour les clauses de Horn. Pour obtenir la complétude dans le cas général de clauses logiques du premier ordre, il faut utiliser de surcroît l'application restreinte et spécifique de la paramodulation ordonnée, appelée paramodulation fusionnante.
Le système d'inférence ainsi obtenu est compatible avec la suppression des tautologies et des techniques plus fortes de suppression et de simplification telles que la subsomption et la réécriture de termes.

****

Concernant ces différents travaux, plusieurs perspectives de recherche peuvent être mentionnées:

Dans le cas de Valerian, il serait intéressant d'améliorer la complexité de temps de l'étape de compilation du système de réécriture et d'étendre Valerian à la spécification de programmes par approximation de systèmes de réécriture généraux (Dauchet 1990 [13]).

Dans le cas de Emmy, la méthode utilisée n'est pas linéaire, ce qui interdit la possibilité de garder les substitutions et permet uniquement de donner une réponse négative ou positive à un problème donné. Il serait peut être possible d'utiliser la règle de superposition stricte de façon linéaire en se limitant aux clauses de Horn. On pourrait espérer ainsi garder l'efficacité obtenue grâce à l'ordre imposé sur les termes lors de l'application des règles d'inférence, cet ordre évitant de réaliser certaines superpositions inutiles. On obtiendrait alors, en n'utilisant que des méthodes inspirées de la surréduction, un langage de programmation logico-fonctionnel. Une tentative analogue se retrouve chez d'autres auteurs, mais avec une méthode différente, à savoir la SLDEI résolution (E-unification par surréduction). Si ces approches aboutissent, il serait intéressant d'en comparer l'efficacité.

Dans les deux cas, que ce soit Emmy ou Valerian, la mise à l'épreuve par la programmation a eu l'intérêt de mieux cerner l'écart entre complexité théorique et complexité constatée. Bien que des améliorations algorithmiques restent à faire les premiers résultats montrent d'ores et déjà la faisabilité de ces approches et l'intérêt potentiel des théories qui les sous-tendent.

VALERIAN
CONCEPTION ET IMPLANTATION D'ALGORITHMES D'ACCESSIBILITE EN REECRITURE UTILISANT LES AUTOMATES D'ARBRES

CHAPITRE I

Bases Théoriques

1. Les systèmes de réécriture et les automates d'arbres.

1.1. Les systèmes de réécriture comme paradigme de la programmation.

On pourra remarquer que les équations sont présentes dans de nombreux domaines scientifiques et plus particulièrement en mathématiques. Si parfois, on essaie de déterminer si une identité donnée peut être déduite d'un ensemble donné d'axiomes de façon logique, d'autres fois, on recherche les solutions d'une équation donnée. Ces différents types de raisonnements se rencontrent fréquemment dans de nombreuses applications informatiques. Entre autre choses, on peut citer le calcul symbolique, la vérification et la spécification de programme, les langages de programmation de haut niveau, l'environnement de programmation, etc . . .
C'est pourquoi les sytèmes de réécriture qui ne sont rien d'autre que des ensembles d'équations orientées, peuvent également jouer un grand rôle en informatique.
Sous la forme d'un programme informatique, les systèmes de réécriture ont fait leur apparition dans Gorn [28].
De nombreux programmes modernes réalisant des manipulations symboliques continuent d'utiliser des règles de réécriture pour faire des simplifications de manière ad-hoc. Un langage d'intelligence artificielle, nommé PROLOG, peut être considéré comme ayant un processus calqué sur la réécriture.
En tant que formalisme, les systèmes de réécriture ont la puissance des machines de Turing et peuvent être considérés comme des algorithmes de Markov non déterministes, fonctionnant sur des termes plutôt que sur des chaînes (Concernant les algorithmes de Markov voir Tourlakis [41]). La théorie de la réécriture est une théorie des formes normales. En effet, les équations orientées d'un système de réécriture sont

utilisées pour calculer, en remplaçant de façon répétitive des sous-termes d'un terme donné par d'autres termes, la forme la plus simple possible. Le résultat final d'une suite non extensible d'applications d'équations orientées, ou règles, est alors appelée une "forme normale". Les systèmes de réécriture définissant au plus une forme normale pour n'importe quel terme d'entrée donné, peuvent être considérés comme des programmes fonctionnels ou comme des interpréteurs de programmes équationnels (O'Donnel, [34]). De plus, lorque les calculs de termes équivalents se terminent toujours en une forme normale unique, un système de réécriture peut faire office de programme fonctionnel non déterministe (Goguen& Tardo, [27]). Un tel système peut également permettre de décider si deux termes sont égaux dans une théorie équationnelle définie par des règles, et résoud ce qu'on appelle le "problème du mot" pour cette théorie.
Ayant vu tout l'intérêt des systèmes de réécriture en informatique, nous allons maintenant pouvoir donner une définition formelle d'un système de réécriture, ainsi que quelques exemples.

Définition 1. *Un **système de réécriture** (SDR) S sur F est un ensemble de règles de réécriture orientées* $R = \{l_i \rightarrow r_i \mid i \in I\}$. *Ici, nous ne considérons que les systèmes de réécriture fini (où I est fini).*
l_i, r_i *appartenant à L(F) où L(F) est l'ensemble des termes sur un alphabet F.*

Définition 2. *Un **système de réécriture d'arbre** (SDR) S sur un alphabet gradué F est un ensemble de règles de réécriture orientées* $R = \{l_i \rightarrow r_i \mid i \in I\}$, *ou l_i et r_i appartiennent à T(F).*

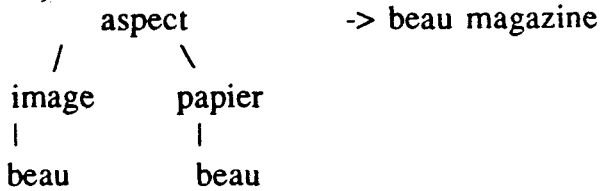
Définition 3. *S est un système de réécriture clos si et seulement si aucune variable n'apparaît dans les règles.*

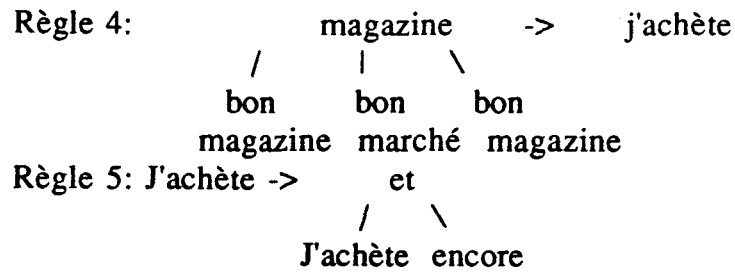
Exemples:
Système de réécriture clos:

Soit le système R1:

Règle 1: articles -> bon magazine
|
intéressants

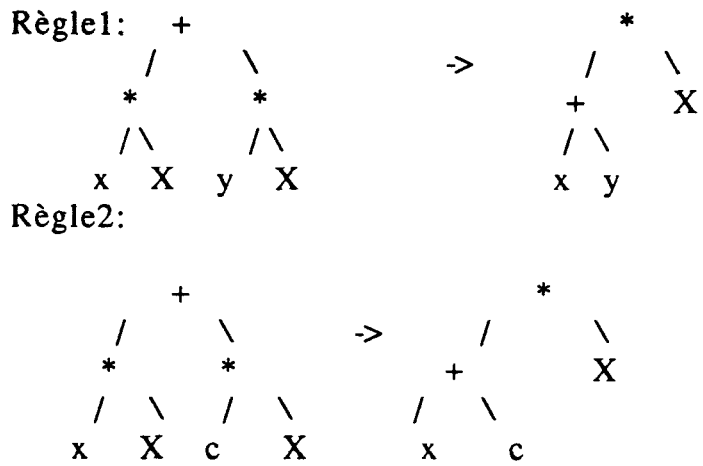
Règle 2: prix -> bon marché
|
petit

Règle 3:




Système de récriture avec variables:

Soit le système R2:



où x et y sont des variables et X et c sont des feuilles.

Parmi les nombreux problèmes posés par les systèmes de réécriture, on peut citer le problème de l'accessibilité, auquel nous nous attachons plus particulièrement dans cette thèse. Ce problème est pour les systèmes de réécriture, ce qu'est le "problème du mot" pour les théories équationnelles.
Ce problème peut se présenter sous deux formes différentes:

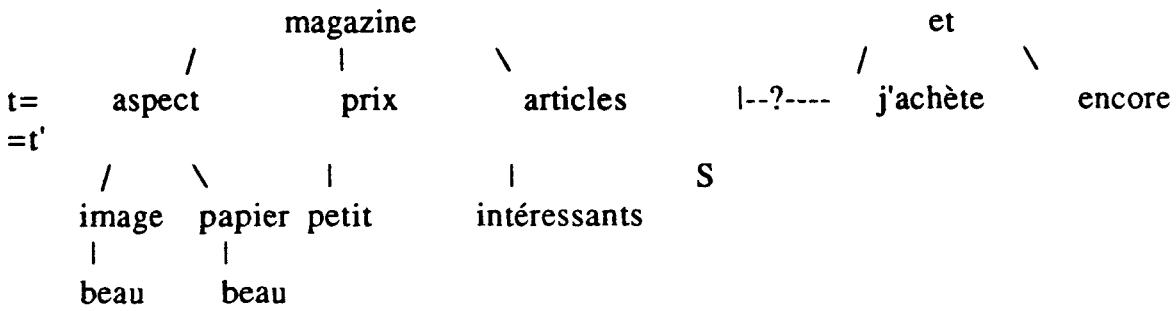
1- Le problème de l'accessibilité du premier ordre consistant à décider pour un système de réécriture donné S et deux termes M et N, si M peut être transformé en N en appliquant les règles de S.

2- Le problème de l'accessibilité du second ordre, qui peut être défini de la façon suivante:
Peut-on à partir d'un programme R, obtenir un résultat satisfaisant à la spécification G à partir d'une donnée satisfaisant à la spécification F?
On a, de façon plus formelle:

Soit un automate A_F reconnaissant la forêt F
Soit un système de réécriture R

Soit un automate A_G reconnaissant la forêt G
On appelle R(F) la transformée de la forêt F par le système R, et $A_{R(F)}$ l'automate reconnaissant cette transformée.
On va alors se demander si $R(F) \cap G \neq \varnothing$
Exemples:

1er ordre:



2ième ordre:

Soit le système R2 précédent, soit la forêt Npol reconnaissant les polynômes en forme normal, et soit la forêt Horn reconnaissant les polynômes de Hörner.
La transformation en polynôme de Hörner consiste à transformer, par exemple, un polynôme de la forme ax^2+bx+c sous la forme $c+x(a+bx)$ ainsi au lieu de faire 3 mulplications dans le cas d'un polynôme en forme normale, on en fera que 2 sous la forme d'un polynôme de Hörner.
La question sera donc:
Peut-on obtenir la forêt Horn à partir de la forêt Npol à l'aide de R2?

1.2. Les automates d'arbres.

Cette partie concerne les automates d'états finis d'arbres et la famille des forêts reconnues par eux. Ici les arbres sont définis comme des termes sur un domaine fini d'opérateurs (ou alphabet gradué fini) et une forêt (ou langage d'arbre) n'est qu'un ensemble d'arbres. Comme dans le cas des langages formels, il y a deux façons naturelles de définir effectivement une forêt: Une forêt peut être reconnue par un automate, ou peut être générée par une grammaire. Ici nous ne nous intéresserons qu'aux automates. La définition d'un automate d'arbres n'est rien d'autre que la généralisation naturelle de celle des automates de mots habituels. Il existe plusieurs types d'automates d'arbres: Ceux qui lisent les arbres des feuilles vers la racine (automates ascendants), ceux qui travaille dans la direction opposée (automates descendants), et dans chacun de ces deux cas, un automate peut être soit déterministe, soit non-déterministe. Nous avons donc au total, quatre types différents d'automates d'arbres. Trois de ces quatres types définissent la même

famille de forêts, la famille REC des forêts reconnaissables. Les automates déterministes descendants sont moins puissants et définissent une sous famille propre de REC.
Dans la suite nous adopterons les notations suivantes:

L'ensemble d'états d'un automate d'arbres fini est appelé E (E_G si l'on veut se référer au système de réécriture clos G) et ses éléments sont représentés par $\{e_1, \ldots, e_n, \ldots\}$

Soit F un alphabet gradué fini. T(F) représente l'ensemble des termes (ou arbres) sur F. $T_m(F)$ représente l'ensemble des termes sur F possédant m variables distinctes. Si $t \in T_m(F)$ et que $r_1 \ldots r_m$ sont des termes d'un autre ensemble alors $t(r_1, \ldots, r_m)$ représente l'arbre qui est obtenu en remplacant chaque x_i par r_i dans t pour chaque i compris entre 1 et m.

Nous définirons donc un automate d'arbre ascendant de la façon suivante.

Définition 4. *Un* ***automate d'arbre ascendant*** *G est un système de réécriture sur* $T(F \cap E_G)$ *qui contient uniquement les règles de la forme:*

(1) $c(e_1, \ldots, e_n) \rightarrow e_0$,
Si n = 0 la règle a la forme suivante: $c' \rightarrow e_0$,
ces règles sont appelées, ***règles de réduction.***

(2) $e \rightarrow e'$
qui sont appelées ***ε–transitions.***

pour $c \in F$ *et* $e_1, \ldots, e_n, e_0, e, e' \in E_G$.

Nous pouvons définir de manière duale l'automate d'arbre descendant.

Définition 5. *Une forêt F est dite* ***reconnaissable*** *si et seulement si il existe un automate d'arbre ascendant qui la reconnaît.*

Propriétés. *La famille REC des forêts reconnaissables est fermée par union, intersection, et complémentarité.*

Définition 6. *Un automate d'arbre M est dit* ***déterministe*** *si et seulement si M(t), représentant l'ensemble des états atteints par M au sommet du terme t, ne contient toujours qu'un élément au plus.*

Exemples:
1- La vérification de types.
Dans ce type d'automate, les états symbolisent les types.
Soit M un automate d'arbre ascendant définie par l'alphabet gradué A, l'ensemble E d'états, l'ensemble F d'états finaux (F est un sous-ensemble

de E) et l'ensemble R de règles (chaque règle peut être vu comme une signature):
A= {+;*,/,-, suc, 1, 0}. +, *, / et - sont des opérateurs binaires (rang 2); suc est unaire (rang 1); 1 et 0 sont des constantes (rang 0). E= {real, int, bool};

Règles:
0 -> real; 0 -> int; 0 -> bool; 1 -> real; 1-> int; 1 -> bool;
+(real,real) ->real; +(int, int) -> int; +(bool, bool) -> bool;
*(real, real) -> real; *(int, int) -> int; *(bool,bool) -> bool;
-(real,real) -> real; -(int,int) -> int; /(real,real)-> real; suc(int)->int;
int ->real; (l'inclusion de type est noté par un type particulier de règle appelée e-transition)

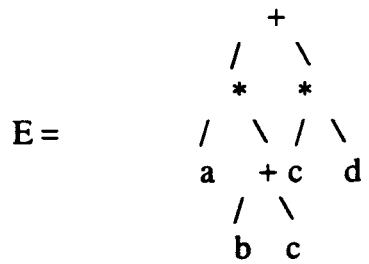
Intuitivement cet automate M calcule en partant des feuilles pour aller jusqu'à la racine, le type d'un terme t. Il tombe en echec si le terme n'est pas bien typé et il est non déterministe car un terme peut avoir plusieurs types.
Par exemple:
M[0] = {real,int,bool}; M[-(1,1)] = {real,int}; M[/(1,+(1,1)] = {real}; M[suc(/(1,+(1,1))] est vide.

2- Les nombres de Strahler.

Ils sont utilisés pour faire de la compilation modulaire. Le but est de traduire un langage source S en un langage brute B.
On définit S à l'aide de la grammaire G suivante:

$$E = E * E \mid E + E \mid I + E \mid I * E \mid I$$
$$I = a \mid \ldots \mid z$$

On aura donc par exemple l'expression suivante générée par G:

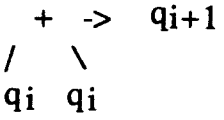


Le langage B, quant à lui sera formé des instructions suivantes:
LOAD i : Chercher le contenu [i] du registre i dans l'accumulateur.
STORE i : Stocker le contenu de l'accumulateur en i.
ADD i : Additionner [accu]+[i] en accumulateur.
MUL i: Multiplier [accu]*[i] en accumulateur.
En supposant que l'on a une machine à n registres, on examinera quelles sont les expressions générées par G que l'on pourra calculer.

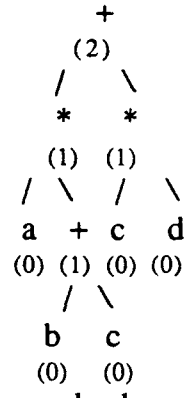
Pour cela on décorera l'expression de G donnée, à l'aide d'un automate d'arbres ascendant, qui aura les règles suivantes:

a -> q0, . . ., z -> q0,

si i>j + -> qi, si i<j + -> qj



on aura les même règles pour l'opérateur *.
L'expression E sera alors décorée de la façon suivante:



Il faudra donc 2 registres pour calculer cette expression. Ensuite il restera à implémenter cette expression dans le langage B à l'aide d'un transducteur d'arbres.
On utilisera par exemple un transducteur d'arbres T1 ayant les règles de la forme:

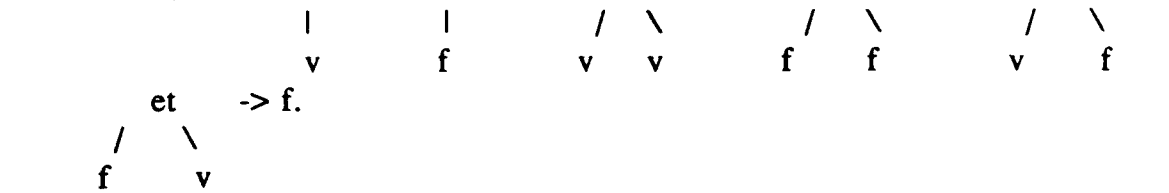
q étant l'état initial et l'unique état du transducteur.
k étant la valeur courante sur laquelle on applique l'opération en question (MUL ou ADD).

3-Evaluation d'une expression booléenne.

L'automate va calculer par ses états, la valeur des sous expressions.
Σ = { ou, et, non, V, F} où les deux premiers opérateurs sont binaires, le troisième est unaire et les deux derniers sont des constantes.
E= {v,f}

Les règles de l'automate sont:

V -> v, F-> f, non -> f, non -> v, et -> v, et -> f, et -> f,



On fera de même pour ou.

1.3. Les propriétés algorithmiques des automates d'arbre.

1.3.1. La réduction du non déterministe.

L'algorithme de réduction du non-déterminisme fonctionne de la même manière que dans le cas des mots. (Pour l'étude détaillée de l'algorithme de réduction du non-déterminisme voir annexe 1 p136)
Sa complexité de temps est donc exponentielle. Reprenons l'exemple 1 précédent. On obtient l'automate M' déterministe équivalent à M suivant:
0 -> rib; 1 -> rib; +(rib,rib)-> rib; *(rib,rib) -> rib; -(rib, rib) -> ri; /(rib) -> r; succ(rib) -> i; +(ri,rib) -> ri; +(rib,ri) -> ri; +(ri,ri) -> ri; *(ri,rib) -> ri; *(rib,ri) -> ri; *(ri,ri) -> ri; -(ri,rib) -> ri; -(rib,ri) ->ri; -(ri,ri) -> ri; -(rib,rib) -> ri; /(ri)-> r; succ(ri) ->i; +(i,rib) -> i; +(i,ri) -> i; +(ri,i) -> i; +(r,rib) -> r etc... etc... . (intuitivement, ri peut être identifié à {real,int} etc...).

1.3.2. L'algorithme de minimalisation.

Le nombre d'états est une mesure simple et naturelle de complexité d'un automate fini. Ici on considère les automates minimaux reconnaissant des forêts. dans le cas d'une forêt reconnaissable, le fait d'être minimal signifie simplement que l'automate possède un nombre minimal d'états, et il existe toujours un automate minimal unique. dans ce cas la procédure est effective et c'est le même algorithme que dans le cas des mots. (voir annexe 1 p140, pour l'étude des algorithmes de suppression des états inutiles)

1.3.3. Rappel de résultats de complexité.

Notation:
On note ||m|| le nombre de règles de l'automate m et $|m_Q|$ le nombre d'états de l'automate m.
On note $\underline{m}$ l'automate qui reconnait le complémentaire du langage reconnu par m.

a- Décision du vide (M = ∅)

Soit M un automate. Pour répondre au problème suivant:
Est-ce que le langage reconnu par M est vide?

la complexité de temps est:
*linéaire dans le cas des mots, avec un accés directe aux règles et en utilisant un algorithme naïf.
* en $O(\|M\| x |M_q|)$, pour les langages d'arbres.

b- Intersection de deux automates M et M', et décision du vide de cette intersection. ($M \cap M' \neq \varnothing$)
* dans le cas des mots, la complexité de temps pour répondre à ce problème est en $O(\|M\| x \|M'\|)$.
* dans le cas des langages d'arbres, la complexité de temps est plus grande, elle est en $O(\|M\| x \|M'\| x |M_q| x |M_q'|)$.

c- Equivalence de M et de M' (M=M')

$$M = M' \iff M \cap \underline{M'} = \varnothing \quad \text{et} \quad \underline{M} \cap M' = \varnothing$$

*Cas déterministe:
On transforme M en $\underline{M}$ en échangeant les états finaux avec les autres états, puis on revient au cas b.

*Cas non-déterministe:
La complexité de temps contient le temps de réduction du non-déterminisme qui est exponentiel.

2. Les transducteurs d'arbres clos.

L'idée intuitive qui se cache derrière le terme de transducteur d'arbres clos, consiste à transformer des termes clos en d'autres termes clos en deux étapes et ceci avec une mémoire qui utilise un nombre fini d'états. Dans la première étape on efface tout ou partie d'un terme et dans la seconde étape on générera un nouveau terme. L'effacement consiste à transformer certains sous-termes en constantes, ou états, et la génération consiste à transformer les états en sous-termes. ces opérations sont rationnelles, c'est à dire que la relation qu'elles définissent satisfait aux propriétés qui sont demandées pour les relations rationnelles, (ou, plus exactement, la stabilité par union, composition et itération). On peut faire un parallèle avec les piles dans le cas des mots. En effet, dans ce cas, l'effacement consisterait à dépiler un certain nombre de lettres et la génération consisterait à en empiler également un certain nombre.

Ici, les étapes d'effacement et de génération sont décrites par des automates d'arbres qui ne sont rien d'autre que des systèmes de réécritures clos qui fonctionnent sur des ensembles de termes clos enrichis par des états considérés comme des constantes.

Définition 7.
*- Une relation $\rightarrow$ est **F-compatible** ou est une précongruence si*

$$(\forall\, c \in F)\ s \to t \quad \Rightarrow \quad c(\ldots, s, \ldots) \to c(\ldots, t, \ldots)$$

- La relation $\to^*$ est la précongruence réflexive et transitive sur les termes clos contenant $\to$.

Définition 8. *Un **transducteur d'arbre clos** sur T(F) (en abrégé, GTT (de l'anglais Ground Tree Transducer)) est une relation T sur (G, D) associée à deux automates d'arbres G et D et définie comme suit:*

$$t \to^T \leftarrow t' \text{ ssi il existe } u \in T(F \cup E_G \cup E_D) \text{ tel que } t \to_G^* u \ {}^*\!\leftarrow_D t'$$

de façon à produire des couples de termes, les ensembles E_G *et* E_D *sont supposés non disjoints.* $E_G \cap E_D$ *est appelée l'**interface**.*

Il serait plus intuitif de considérer u comme un terme de la forme $c(e_1, \ldots, e_n)$ où $c(x_1, \ldots, x_n)$ est le contexte commun à t et à t' avec $t = c(t_1, \ldots, t_n)$, $t' = c(t'_1, \ldots, t'_n)$ et $t_1 \to_G^* e_1 \ {}^*\!\leftarrow_D t'_1, \ldots, t_n \to_G^* e_n \ {}^*\!\leftarrow_D t'_n$.

(Voir figure 2)

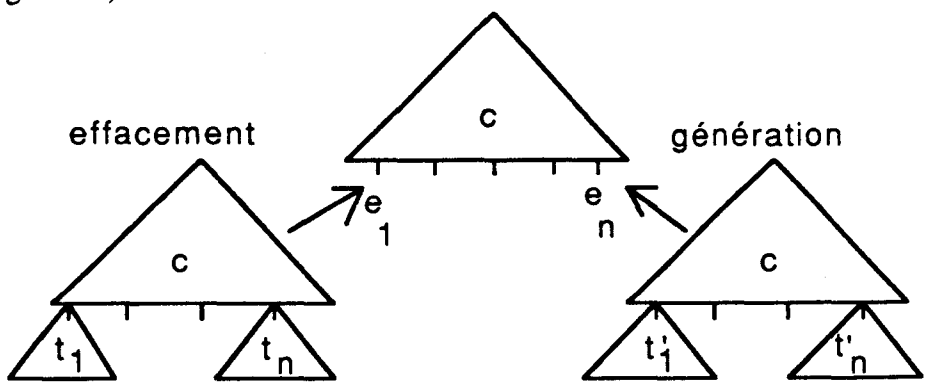


FIGURE 2

Une relation associée à un transducteur d'arbre clos est appelée une GTT-relation. La relation associée à un transducteur d'arbre clos T est notée par R(T).

3- Stabilité des GTT-relations.

(voir Dauchet -Tison [11] et Dauchet, Heuillard, Lescanne, Tison [10])

A partir de la définition il est facile de voir que l'inverse d'une GTT-relation est encore une GTT-relation, il suffit de commuter le rôle de G

et de D. L'union de deux GTT-relation R_1 et R_2 n'est pas vraiment une GTT-relation. Cependant la clôture précongruente de $R_1 \cup R_2$ qui est la plus petite précongruence contenant R_1 et R_2 est associée au GTT (G_3, D_3) défini par $G_3 \equiv G_1 \cup G_2$ et $D_3 \equiv D_1 \cup D_2$ ou (G_1, D_1) et (G_2, D_2) sont les GTT associé aux relations R_1 et R_2. Ceci suppose que $E_{G1} \cup E_{D1} \neq E_{G2} \cup E_{D2}$. Donc on peut établir les propositions suivantes.

Proposition 1. *L'inverse d'une GTT-relation est une GTT-relation.*

Proposition 2. *La clôture précongruente de l'union de deux GTT-relations est une GTT-relation.*

Etablissons maintenant une autre propriété des GTT-relations, c'est à dire la stabilité par itération. La preuve est basée sur un lemme particulier. Etant donné deux relations A et B nous définissons une relation $\rightarrow\{A, B\}$ à l'aide des deux règles d'inférences suivantes:

$$\frac{f(e_1, \ldots, e_n) \rightarrow^*_A e}{f(e_1, \ldots, e_n) \rightarrow^*_{\{A, B\}} e}$$

$$\frac{e_0 {}^*\!\leftarrow_B f(e_1, \ldots, e_n) \;\&\; e_1 \rightarrow^*_{\{A, B\}} e'_1, \ldots, e_n \rightarrow^*_{\{A, B\}} e'_n \;\&\; f(e_1, \ldots, e_n) \rightarrow^*_{\{A, B\}} e'_0}{e_0 \rightarrow^*_{\{A, B\}} e'_0}$$

Prouvons maintenant notre lemme de base:

Lemme 1. *Si A et B sont décrits par des automates d'arbres sans ε-transition et*

$$u_1 {}^*\!\leftarrow_{\{A, B\}} t \rightarrow^*_{\{B, A\}} u_2$$

avec $t \in T(F \cup E_A \cup E_B)$, *il existe* $s \in T(F \cup E_A \cup E_B)$ *tel que*

$$u_1 \rightarrow^*_{\{B, A\}} s {}^*\!\leftarrow_{\{A, B\}} u_2$$

en terme de relation, on a

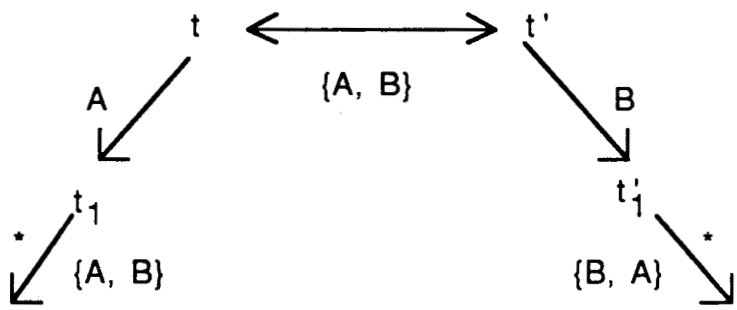
$$\rightarrow^*_{\{B, A\}} \circ {}^*\!\leftarrow_{\{A, B\}} \supseteq {}^*\!\leftarrow_{\{A, B\}} \circ \rightarrow^*_{\{B, A\}}$$

Preuve. Remarquons que $\leftarrow_{\{A, B\}}$ et $\rightarrow_{\{B, A\}}$ contiennent des ε-transitions (à cause des règles d'inférences définies ci-dessus). Notons également que si $u \ {}^*\!\leftarrow_{\{B, A\}} v$ utilise uniquement des ε-transitions, alors $u \ {}^*\!\leftarrow_{\{A, B\}} v$ utilise uniquement des ε-transitions et vice versa. Ecrivons $u \leftarrow\{A, B\}\rightarrow v$ dans ce cas. La preuve du lemme est réalisé par induction sur la taille des termes. Les cas de base sont les suivants:

1er cas: $u1 \leftarrow\{A, B\}\rightarrow t \rightarrow^*_{\{B, A\}} u2$, alors si s est pris comme étant u2 alors $u1 \rightarrow^*_{\{B, A\}} s \equiv u2$.

Second cas: $u1 \ {}^*\!\leftarrow_{\{A, B\}} t \leftarrow\{A, B\}\rightarrow u2$ la preuve est la même que la précédente avec $s \equiv u1$.

Cas général: La prémisse du cas général est un diagramme de la forme suivante:



deux cas peuvent être considérés: soit les occurences sur lesquelles A et B sont utilisées sont différentes soit ce sont les mêmes.
Si *les occurences sont différentes*, on a

$$t = c(e_1, \ldots, e_n, f(a_1, \ldots, a_p), g(b_1, \ldots, b_q))$$

et

$$t' = c(e'_1, \ldots, e'_n, f(a'_1, \ldots, a'_p), g(b'_1, \ldots, b'_q))$$

et

$$t_1 = c(e_1, \ldots, e_n, a_0, g(b_1, \ldots, b_q))$$

et

$$t_2 = c(e_1, \ldots, e_n, a_0, g(b'_1, \ldots, b'_q))$$

et

$$t'_1 = c(e'_1, \ldots, e'_n, f(a'_1, \ldots, a'_p), b_0)$$

et

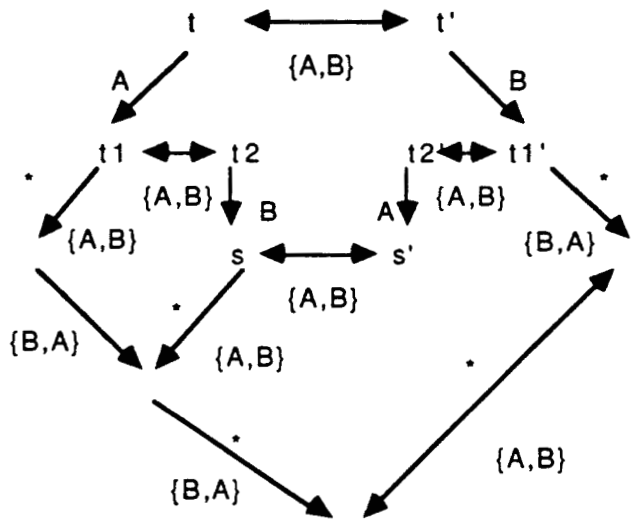
$$t'_2 = c(e'_1, \ldots, e'_n, f(a_1, \ldots, a_p), b_0)$$

où $e_1, \ldots, e_n, e'_1, \ldots, e'_n, a_1, \ldots, a_p, a'_1, \ldots, a'_p, a_0, b_1, \ldots, b_q, b'_1, \ldots, b'_q, b_0$ sont dans $E_A \cup E_B$.
Définissons

$$s = c(e_1, \ldots, e_n, a_0, b_0)$$
$$s' = c(e'_1, \ldots, e'_n, a_0, b_0)$$

On a $s \leftarrow\{A, B\}\rightarrow s'$.

Par induction sur la taille des termes, comme t_1, t'_1, s et s' sont de taille plus petite que celle de t, le diagramme peut être complété de la manière suivante:



Si *les occurences sont les mêmes*, on a,

$$t = c(e_1, \ldots, e_n, f(a_1, \ldots, a_p))$$

et

$$t' = c(e'_1, \ldots, e'_n, f(b_1, \ldots, b_p))$$

On définit t_1 et t'_1 ainsi:

$$t_1 = c(e_1, \ldots, e_n, a_0)$$

où

$$f(a_1, \ldots, a_p) \rightarrow A^* a_0,$$

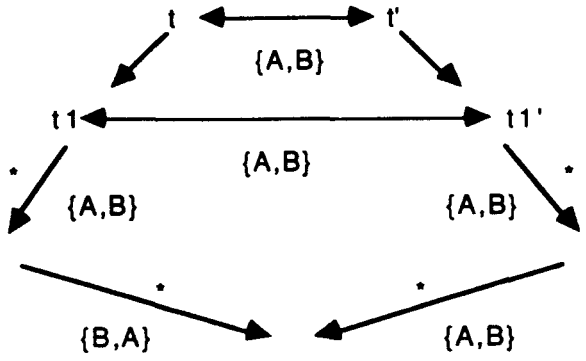
et

$$t'_1 = c(e'_1, \ldots, e'_n, b_0)$$

où

$$f(b_1, \ldots, b_p) \rightarrow B^* b_0,$$

On a $t_1 \leftarrow\{A, B\}\rightarrow t'_1$ à partir de la seconde règle d'inférence, de plus le diagramme peut être complété de la manière suivante:



Les propositions suivantes sont les conséquences du lemme précédent.◊

Proposition 2: $\rightarrow\{B, A\}^* o\ {}^*\leftarrow\{A, B\} \supseteq {}^*\leftarrow A\ o \rightarrow B^*$

Preuve: C'est évident à partir du lemme 1 et à partir du fait que $\leftarrow\{A,B\} \supseteq \leftarrow A$ et que $\rightarrow\{B, A\} \supseteq \rightarrow B$.◊

Proposition 3: $(\rightarrow B^* o\ {}^*\leftarrow A)^* = \rightarrow\{B, A\}^* o\ {}^*\leftarrow\{A, B\}$. L'itération d'une GTT-relation est une GTT-relation.

Preuve: L'inclusion $(\rightarrow B^* o\ {}^*\leftarrow A)^* \supseteq \rightarrow\{B, A\}^* o\ {}^*\leftarrow\{A, B\}$ est obtenue par induction à partir de la proposition 2 et du lemme 1. On a $(\rightarrow B^* o^* \leftarrow A)^* \supseteq \rightarrow\{B, A\}^*$ et $(\rightarrow B^* o^* \leftarrow A)^* \supseteq {}^*\leftarrow\{A, B\}$. Donc on obtient de manière évidente, $(\rightarrow B^* o\ {}^*\leftarrow A)^* \supseteq \rightarrow\{B, A\}^* o\ {}^*\leftarrow\{A, B\}$. Puisque $(\rightarrow\{B, A\}, \leftarrow\{A, B\})$ défini un GTT, ceci prouve facilement la stabilité par itération.◊

A partir de là on peut prouver une autre propriété, c'est à dire la stabilité par composition.
Pour cela on établit la proposition suivante:

Proposition 4: *Si* $E_A \cap E_B = \emptyset$ *alors*

$$\leftarrow A\ o \rightarrow B \supseteq \rightarrow B\ o \leftarrow A$$

et

$${}^*\leftarrow A\ o \rightarrow B^* \supseteq \rightarrow\{B, A\}^* o\ {}^*\leftarrow\{A, B\}$$

Preuve: Puisque les ensembles d'états sont disjoints, si $\rightarrow B$ précéde $\leftarrow A$, les réécritures ont lieu sur deux occurences différentes et donc elles peuvent commuter. Donc on obtient ${}^*\leftarrow A\ o \rightarrow B^* \supseteq \rightarrow B^*\ o\ {}^*\leftarrow A$. Nous concluons à l'aide de la proposition 3.◊

Proposition 5: *La composition de deux GTT-relations est une GTT-relation.*

Preuve: Supposons que les deux relations sont associées aux GTT (G, D) et (Γ, Δ) et que les ensembles d'états satisfont la condition $E_D \cap E_\Gamma = \varnothing$. La relation qui correspond à la composition des relations associées aux deux GTT est donnée par $\rightarrow_G{}^*$ o $*\leftarrow_D$ o $\rightarrow_\Gamma{}^*$ o $*\leftarrow_\Delta$. A partir de la proposition 2 et de la proposition 4, ceci est égal à $\rightarrow_G{}^*$ o $\rightarrow_{\{\Gamma, D\}}{}^*$ o $*\leftarrow_{\{D, \Gamma\}}$ o $*\leftarrow_\Delta$. En renommant les états ceci peut facilement être décrit comme un GTT.◊
Pour terminer cette partie, on peut grâce aux propositions précédentes établir la proposition importante suivante:

Proposition 6: *Soit S un système de réécriture clos, alors il existe un GTT U tel que* $\Rightarrow_S{}^* = R(U)$.

Preuve: Fülöp & Valgvölgyi[22]
Soit $S = \{t_i \rightarrow r_i \mid 1 \leq i \leq n\}$ un système de réécriture clos sur F. Pour chaque $1 \leq i \leq n$, construire les automates d'arbre A_i et B_i avec $E_{A_i} \cap E_{B_i} = \{e_i\}$ tel que $L(A_i(e_i)) = \{t_i\}$ et $L(B_i(e_i)) = \{r_i\}$. Supposons que $E_{A_i} \cap E_{A_j} = \varnothing$ et que $E_{B_i} \cap E_{B_j} = \varnothing$ chaque fois que $1 \leq i \neq j \leq n$. Alors, construire l'automate d'arbre A à partir des A_i en posant

$$E_A = \bigcup_{i=1}^{n} E_{A_i},$$

et si R_A est l'ensemble des règles de A et R_{A_i} l'ensemble des règles de A_i on pose

$$R_A = \bigcup_{i=1}^{n} R_{A_i},$$

et de la même manière construire l'automate B à partir des B_i. On établit alors que pour le GTT V= (A, B), $\Rightarrow_S{}^* = R(V)^*$ est vraie.

(a) Si $t, r \in T(F)$ sont tels que $t \Rightarrow_S r$, alors pour $s \in T_1(F)$ et $1 \leq i \leq n$ on a $t = s(t_i)$ et $r = s(r_i)$. donc, il est évident à partir de la construction de A et de B que $t = s(t_i) \Rightarrow_A{}^* s(a_i) *\Leftarrow_B s(r_i) = r$, donc, $(t, r) \in R(V)$. Par conséquent $R(V)^* \supseteq \Rightarrow_S{}^*$.

(b) Pour prouver l'inclusion inverse, notons par V_i le GTT (A_i, B_i), pour $1 \leq i \leq n$. On note, grâce à la proposition 2

$$R(V)^* = (\bigcup_{i=1}^{n} R(V_i))^*$$

Maintenant prenons $(t, r) \in R(V_i)$, supposons que pour un $m \geq 0$ et un $s \in T_m(F)$, on a

$$t = s(t_i, \ldots, t_i) \Rightarrow A_i^* \; s(a_i, \ldots, a_i)^* \Leftarrow B_i \; s(r_i, \ldots, r_i) = r.$$

Alors on peut voir que $(t, r) \in \Rightarrow S^*$ et donc

$$\Rightarrow S^* \supseteq \bigcup_{i=1}^{n} R(V_i),$$

puisque $\Rightarrow S^*$ est réflexive et transitive, on a également

$$\Rightarrow S^* \supseteq (\bigcup_{i=1}^{n} R(V_i))^*.$$

Donc par la proposition 3, il existe un GTT U avec $R(U) = R(V)^*$ ce qui termine la preuve.◊

4- Création de l'automate S" reconnaissant toutes les transformations possibles d'un système de réécriture clos.

L'idée consiste à inscrire la GTT-relation associée à un système de réécriture clos S dans un langage d'arbre régulier. Pour répondre à la question t peut-il se réécrire en t' (problème de reachabilité) ? nous n'aurons qu'à examiner si la transformation associée à cette question est un élément de ce langage.
La fonction est définie de la façon suivante:
à chaque couple

$$t = c(t_1, \ldots, t_n) \rightarrow G^* e(e_1, \ldots, e_n)$$
$$c(e_1, \ldots, e_n)^* \leftarrow D \; t' = c(t'_1, \ldots, t'_n)$$

On associe le terme $v\downarrow_{\#}$, ou $v\downarrow_{\#}$ est la forme normale du terme $v = c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n))$ pour le système de réécriture $R_{\#}$ définie par les règles

$$\#(f(x_1, \ldots, x_p), f(y_1, \ldots, y_p)) \rightarrow_{R_{\#}} f(\#(x_1, y_1), \ldots, \#(x_p, y_p))$$

Notons $\mathbb{L}(T)$, le langage formé par les $v\downarrow_{\#}$ et appelons le, le produit tensoriel du GTT T. Montrons que $\mathbb{L}(T)$ est reconnu par l'automate d'arbre $\mathbb{A}(T)$. L'ensemble des états est divisé en quatre parties E_G, $E_G{}^c$, {ok}, et $E_G \times E_D{}^c$. $E_D{}^c$ est une copie de E_D de façon à écarter toutes confusion. $e^c \in E_D{}^c$ correspond à $e \in E_D$. ok est l'unique état final,

autrement dit $v \in \mathbb{L}(T)$ si et seulement si $v \rightarrow \mathbb{A}(T)^*$ ok. Les règles de transition sont celles de G, plus celle de la forme:

$$f(e_1^c, \ldots, e_n^c) \rightarrow e^c \quad (1)$$
$$\#(e, e^c) \rightarrow ok \quad (2)$$
$$f(ok, \ldots, ok) \rightarrow ok \quad (3)$$
$$\#(e', e^c) \rightarrow < e', e^c > \quad (4)$$
$$< e', e^c > \rightarrow ok \quad (5)$$
$$f(< e'_1, e_1^c >, \ldots, < e'_n, e_n^c >) \rightarrow < e', e^c > \quad (6)$$

La première règle correspond à la règle

$$f(e_1, \ldots, e_n) \rightarrow_D e \quad (7)$$

La dernière règle correspond aux règles

$$f(e'_1, \ldots, e'_n) \rightarrow_G e' \quad (8)$$
$$f(e_1, \ldots, e_n) \rightarrow_D e \quad (9)$$

La première règle de transition transforme en $E_D{}^c$ les règles de transition E_D, la seconde accepte un terme #(s, t) où s et t sont correctement associés par le GTT T (c'est à dire: $s \rightarrow T \leftarrow t$), la troisième permet simplement de transmettre les précédentes acceptations à travers le contexte. Puisque $R_\#$ descend les # à travers les termes possèdant le même sommet, on peut se trouver en face du cas où un couple n'a pas été complétement reconnu quand apparaît un #. La quatrième règle permet de continuer la reconnaissance lorsque l'on se trouve dans un tel cas. Puisque les termes sont supposés avoir le même sommet, la reconnaissance du reste des deux termes de chacun des couples doit être faite sur chaque terme en même temps, et la reconnaissance doit fonctionner sur un couple de terme. C'est le rôle de la sixième règle. La cinquième règle correspond à la seconde lorsque l'on travaille sur des couples d'états, ceci signifie que ce qui a été reconnu jusqu'à ce point correspond à la forme normale $R_\#$ d'un terme #(s, t) où $s \rightarrow T \leftarrow t$. Notons que les règles (2) et (4) rendent $\mathbb{A}(T)$ non déterministe. Le théorème suivant exprime le fait que $\mathbb{L}(T)$ et $\mathbb{A}(T)$ sont correctement associés.

Théorème 1: *Les trois énoncés suivants sont équivalents*

(i) $c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n))\downarrow_\# \in \mathbb{L}(T)$
(ii) $c(t_1, \ldots, t_n) \rightarrow T \leftarrow c(t'_1, \ldots, t'_n)$
(iii) $c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n))\downarrow_\#$ *est reconnu par l'automate* $\mathbb{A}(T)$, *autrement dit* $c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n))\downarrow_\# \rightarrow_{\mathbb{A}(T)^*}$ ok.

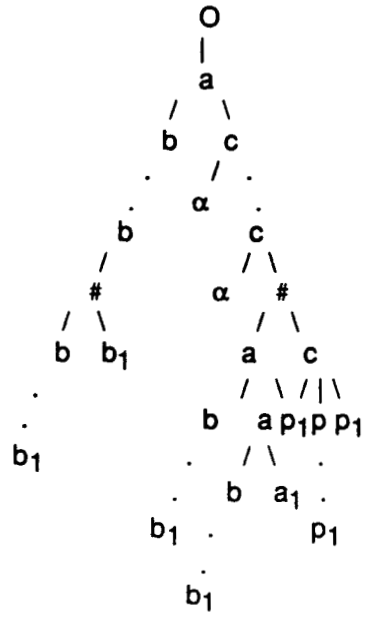
Preuve:
(i) $\Leftrightarrow$ (ii) est une conséquence directe de la définition de $\mathbb{L}(T)$. Pour la preuve de (ii) $\Rightarrow$ (iii), regardons les parties $\#(t_i, t'_i)$ de $c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n))\downarrow_\#$. Sa forme normale $\#(t_i, t'_i)\downarrow_\#$ est de la forme $c_i(\#(t_{i,1}, t'_{i,1}), \ldots, \#(t_{i,n}, t'_{i,n}))$. Si $c(t_1, \ldots, t_n) \rightarrow T \leftarrow c(t'_1, \ldots, t'_n)$, on a $t_i \rightarrow_G {}^*e_i {}^* \leftarrow_D t'_i$ pour l'état e_i de $\mathbb{A}(T)$. Pour reconnaître $\#(t_i, t'_i)\downarrow_\#$ c'est à dire, pour avoir $\#(t_i, t'_i)\downarrow_\# \rightarrow_{\mathbb{A}(T)^*}$ ok, on utilise la première règle pour dériver $t_{i,j} \rightarrow_{\mathbb{A}(T)^*} e_{i,j}$ et $t'_{i,j} \rightarrow_{\mathbb{A}(T)^*} e'_{i,j}$, ensuite la quatrième règle pour avoir $\#(e_{i,j}, e'_{i,j}) \rightarrow_{\mathbb{A}(T)^*} <e_{i,j}, e'_{i,j}>$, puis la sixième règle pour avoir

$$c_i(\#(t_{i,1}, t'_{i,1}), \ldots, \#(t_{i,n}, t'_{i,n})) \rightarrow_{\mathbb{A}(T)^*} <e_i, e_i>,$$

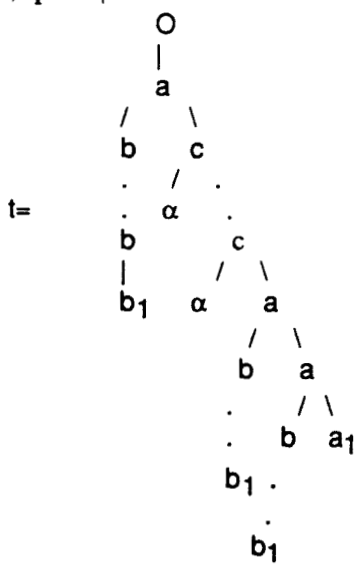
et enfin la cinquième règle pour avoir $<e_i, e_i> \rightarrow_{\mathbb{A}(T)^*}$ ok. Donc $c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n)) \rightarrow_{\mathbb{A}(T)^*}$ ok en utilisant la troisième règle. Pour la preuve de (iii) $\Rightarrow$ (ii), le problème consiste à reconstruire les termes $c(t_1, \ldots, t_n)$ et $c(t'_1, \ldots, t'_n)$ à partir de la reconnaissance de $c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n))\downarrow_\#$, autrement dit, il s'agit de reconstruire $c(\#(t_1, t'_1), \ldots, \#(t_n, t'_n))$ à partir de sa forme normale $R_\#$. L'idée consiste à remonter les # dans le terme jusqu'à ce que les règles $<e_i, e_i> \rightarrow_{\mathbb{A}(T)}$ ok soient utilisées.◊

Le langage $\mathbb{L}(T)$ est rationnel car il est reconnu par un automate ascendant $\mathbb{A}(T)$. La présentation est un peu différente de celle utilisée dans la littérature habituelle sur les automates, mais le lecteur familier avec ce type d'approche peut facilement se convaincre que $\mathbb{A}(T)$ est effectivement un automate ascendant.
Nous associerons donc au GTT S' qui simule un système de réécriture clos S, l'automate A(S') qui reconnaît une forêt symbolisant toutes les transformations qui peuvent être réalisées avec S' et donc, comme on l'a vu dans la partie précedente, avec S.

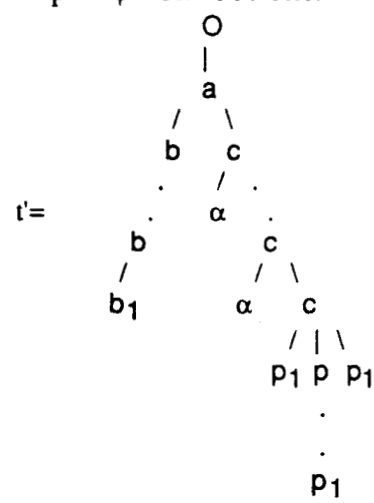
On pourra par exemple décrire un élément de cette forêt de la façon, suivante:



A l'intérieur de cet arbre, on peut mettre en évidence deux arbres t et t', à l'aide des deux morphismes ϕ et ϕ'.
ϕ supprime le fils droit de chaque noeud # et ϕ' supprime le fils gauche de chaque noeud #.
Donc, par ϕ on obtient:



par ϕ' on obtient:



Ainsi l'arbre A signifie que l'on peut transformer t en t' avec le système S.
Donc toutes les transformations en accord avec S sont codées dans une forêt reconnaissable F avec:

$$F = \{\, t \# t' \mid t \overset{S}{\vdash\!\!*} t' \,\}$$

On peut ainsi décider du problème de l'accessibilité. Cependant pour construire $\mathbb{A}(S')$ à partir de S', non seulement on crée les règles (1) à (6) définies dans la preuve précédente, mais, il faut en outre, tenir compte d'une difficulté cachée. En effet il faut descendre dans F les noeuds # le plus bas possible, afin que les lettres filles de ces noeuds soient toujours différentes. Afin que notre automate soit capable de reconnaitre la forêt F ainsi modifiée, on doit réaliser les deux opérations suivantes:

a- Garder dans les états la dernière lettre reconnue.
b- Ne pas créer les règles suivantes:

$\#(e, e^c) \rightarrow$ ok si e et e^c ont reconnu la même lettre.

Mais comme on l'a déjà remarqué précédemment, notre automate $\mathbb{A}(S')$ n'est pas déterministe et si on désire avoir la réponse au problème de l'accessibilité en temps réel, il faut rendre ce dernier déterministe, ce qui est réalisé en temps exponentiel dans le pire des cas.
Donc finalement,

$t \vdash^*_- t' \quad \Leftrightarrow \quad t \# t'$ est reconnu par $\mathbb{A}(S')$

De plus, on sait que l'inclusion de deux langages d'arbres rationnels est décidable. Donc on obtient le corollaire suivant:

Corollaire 1: *L'inclusion de deux GTT-relations est décidable.*

On sait également que la confluence d'une relation de réécriture $\rightarrow$ peut être exprimée par l'inclusion suivante:

$$\rightarrow^* \circ {}^*\!\leftarrow \supseteq {}^*\!\leftarrow \circ \rightarrow^*$$

Donc, à partir des résultats précédents, on obtient le corollaire suivant:

Corollaire 2: *La confluence d'un système de réécriture clos est décidable.*

Remarque:
L'automate $\mathbb{A}(S')$ possède beaucoup de règles ainsi que beaucoup d'états, si bien que la réduction du non-déterminisme pose très vite des problèmes d'espace mémoire et de temps d'exécution. Nous verrons dans le chapitre suivant comment nous avons résolu le problème.

5-Les algèbres multi-sortées avec sortes ordonnées.

5.1- Les signatures multi-sortées avec sortes ordonnées.

Définition 9:
Une signature multi-sortée (SMS en abrégé) est un couple (S,F) où S est un ensemble de noms de sortes (qui seront notée par $\underline{s}, \underline{s1}, \ldots$) et F est un ensemble de symboles de fonctions associé à une fonction de typage τ qui associe à chaque $f \in F$ une chaîne appartenant à S^+. Lorsque $\tau(f) = \underline{s1}, \underline{s2}, \ldots, \underline{sn}, s$ on ecrit $f: \underline{s1} \times \ldots \times \underline{sn} \rightarrow \underline{s}$ et on dit que f possède la signature $\underline{s1} \times \ldots \times \underline{sn} \rightarrow \underline{s}$.

Exemple: Soit la signature (S,F) telle que
$S=\{N,R,B\}$ $F=\{+,>\}$
on a alors $\tau(+) = \{NNN, RRR\}$ et $\tau(>) = \{BNN, BRR\}$

Définition 10:
Une signature avec sortes ordonnées (SSO en abrégé) est un triplet ($S, \geq$, F) où S est un ensemble de symboles de types, $\geq$ est un ordre sur S et F est un ensemble de symboles de fonction associé à une fonction de typage τ qui associe à chaque $f \in F$ un sous-ensemble non-vide de S^+. Tous les mots de T(F) ont la même longueur n+1 et |f|=n est l'arité de f. Comme dans le cas multi-sorté, on dit que f possède la signature $\underline{s1} \times \ldots \times \underline{sn} \rightarrow \underline{s}$ quand $\underline{s1}, \underline{s2}, \ldots, \underline{sn}, s \in \tau(f)$.

Dans chacun des cas, X est un ensemble de symboles de variables. Un type est affecté à chaque variable et on écrit $x: \underline{s}$ et $\tau(x) = \underline{s}$. On suppose qu'il y a un nombre infini de variables de chaque sorte.
Dans chaque cas, si SIG est une signature, T(SIG,X) (parfois on écrit T(F,X) lorsqu'il n'y a pas d'ambiguité) est l'ensemble des termes "bien formés" construits sur SIG et X de façon habituelle. Lorque X est vide on écrit T(SIG) (ou T(F)) au lieu de T(SIG,0).
Dans la suite, on suppose que pour tout $\underline{s} \in S$, il y a au moins un $t \in$ T(SIG) tel que t possède le type $\underline{s}$.

Propriété:
Une signature est finie lorsque à la fois S et F sont finis.

5.2-Algèbre avec sortes ordonnées.

Définition 11:
Soit SIG=($S, \geq$, F) une SSO. Une (SIG-)algèbre avec sortes ordonnées $\mathbb{A}$ est définie par:

-Une famille $(\mathbb{A}_{\underline{s}})_{\underline{s} \in S}$ d'ensembles non-vides telle que, lorsque $\underline{s} \leq \underline{s}'$, alors $\mathbb{A}_{\underline{s}} \subseteq \mathbb{A}_{\underline{s}'}$. Soit $C_{\mathbb{A}} = \cup_{\underline{s} \in S} \mathbb{A}_{\underline{s}}$.

-Pour chaque symbole de fonction f, une application $f_{\mathbb{A}}$ de $D_f{}^{\mathbb{A}} \subseteq C_{\mathbb{A}}{}^{|f|}$ dans $C_{\mathbb{A}}$ telle que si f possède la signature $w \rightarrow \underline{s}$, alors $\mathbb{A}_w \subseteq D_f{}^{\mathbb{A}}$ et $f(\mathbb{A}_w) \subseteq \underline{s}_{\mathbb{A}}$. (si $\underline{s}1, \underline{s}2, \ldots, \underline{s}n = w \in S^+$, $\mathbb{A}_w$ et le produit cartésien $\mathbb{A}_{\underline{s}1} x \ldots x \mathbb{A}_{\underline{s}n}$)

Etant donné une SMS (S,F), une (S,F)-algèbre multi-sortée est simplement une (S, ≥, F)-algébre avec sortes ordonnées, ou ≥ est l'ordre trivial sur S ($\underline{s} \geq \underline{s}'$ ssi $\underline{s} = \underline{s}'$). De cette façon, les algèbres avec sortes ordonnées généralisent strictement les algèbres multi-sortées.
Donc lorsque l'on parle de substitution, de règles de réécritures, . . ., sans rien mentionner d'autre, on doit comprendre "substitution avec sortes ordonnées", "règles de réécritures avec sortes ordonnées" . . ., etc.

Définition 12: *Une* ***règle de réécriture multi-sortée*** *est un couple de termes s, t* ∈ T(SIG, X) *tel que* Var(t) ⊆ Var(s), *on écrit alors* $s \rightarrow t$. *Un* ***système de réécriture*** *multi-sorté est un ensemble de règles de réécriture multi-sortées.*

Définition 13: Système de réécriture multi-sorté séparé (SRS)
Soit $\rho: l \rightarrow r$ *une règle de réécriture.* ρ *est séparée si aucune variable n'apparaît deux fois dans cette règle.*
Exemple: +(*(x,y),*(z,y) → *(+(x,z),y) n'est pas séparé, mais
+(*(x,y),*(z,y')) → *(+(x',z'),y") est séparé.
Un système de réécriture est SRS si chaque règle est séparée (et sortée).

CHAPITRE 2

Algorithmes et étude de la complexité

1- Introduction

Avant d'étudier en détail nos algorithmes permettant de décider du problème de l'accessibilité en temps réel, nous allons rappeler briévement une autre méthode permettant de résoudre le problème de façon à expliquer la raison pour laquelle nous n'avons pas retenu cette façon de procéder.

1.1- Les algorithmes décidant de l'équivalence de deux expressions dans une relation d'équivalence.

Nous allons commencer par étudier ce type d'algorithme dont Oyamaguchi-Togashi-Nogushi [35] se sont directement inspirés pour démontrer que le problème de reachabilité était décidable. De plus, dans un tel cadre, Peter Downey, Sethi et Tarjan [18] ont trouvé un algorithme en O(n log n), c'est pourquoi il est intéressant de bien le connaître afin de voir si il n'est pas possible d'avoir un résultat équivalent pour le problème de l'accessibilité.

Algorithme calculant la clôture congruente d'une relation d'équivalence:

Soit G= (V, E) un graphe orienté tel que pour chaque sommet v de G, les successeurs de v soient ordonnés. Soit C une relation d'équivalence sur V. La clôture congruente C* de C est la relation d'équivalence la plus fine sur V contenant C et satisfaisant la propriété suivante pour tous sommets v et w.
(i) Soit v et w ayant les successeurs $v1, v2, \ldots, vk$ et $w1, w2, \ldots, wl$, respectivement. Si $k = l \geq 1$ et $(v_i, w_i) \in C^*$ pour $1 \leq i \leq k$, alors $(v, w) \in C^*$. Autrement dit, si les successeurs correspondant de v et de w sont équivalent sous C*, alors v, et w sont eux-même équivalents sous C*.

Un algorithme simple:

Soit n le nombre de sommets de G, m le nombre d'arêtes. On suppose ici que n et en $O(m)$.
Cet algorithme est une extension directe de la méthode de Cocke et Scharwz pour trouver des sous-expressions communes.

Initialisation. Soit $C_0 = C$ et $i = 0$
Pas général:-Numéroter les classes d'équivalence de C_i à partir de 1.
-Affecter à chaque sommet de v le nombre $\alpha(v)$ de la classe d'équivalence contenant v.
-Pour chaque sommet v construire une signature $s(v) = (\alpha(v1), \alpha(v2), \ldots, \alpha(vk))$, où $v1, v2, \ldots, vk$ sont les successeurs de v. La signature de chaque sommet est une suite d'entiers dans l'intervalle $1 \leq i \leq n$.
-Grouper les sommets en classes de sommets ayant des signatures équivalentes. Soit C_{i+1} la relation d'équivalence la plus fine sur v telle que deux sommets équivalents sous C_i ou ayant la même signature, soient équivalents sous C_{i+1}.
-Si $C_{i+1} = C_i$ alors $C^* = C_{i+1}$. Sinon remplacer i par i+1 et répéter le pas général.

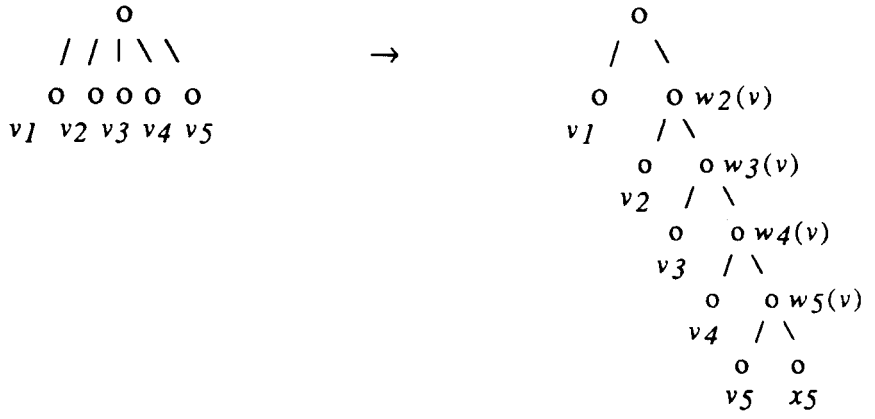
Complexité:

Puisque chaque exécution du pas général, excepté le dernier, peut fusionner au moins deux classes d'équivalence, le nombre d'exécutions du pas général est au plus égal à n.
Ce qui prend le plus de temps dans l'algorithme est le regroupement de sommets ayant la même signature. En utilisant un ordre lexicographique, ce pas est en $O(m)$ et l'algorithme complet est en $O(mn)$, dans le pire des cas. En utilisant une table de hachage, ce pas est en moyenne en $O(m)$ et l'algorithme complet est en moyenne en $O(mn)$

(car l'accés à un élément de la table de hachage est en moyenne en O(1)).
Remarque:
On peut obtenir un algorithme plus rapide en faisant la remarque suivante:
Cet algorithme considère trop de sommets, en fait on peut ne considérer que les sommets dont au moins un successeur a changé de classe d'équivalence pendant l'exécution précédente du pas général.

Un algorithme rapide:

Cet algorithme ne fonctionne que sur des graphes dont chaque noeud ne possède, au plus, que deux successeurs.
On montre (cf [18]) que l'on peut transformer n'importe quel graphe en un graphe dont les noeuds ne possèdent, au plus que deux successeurs. (voir figure)



Structures de données:

On utilise quatre structures de données principales:

i/ La première représente les classes d'équivalences définies par la relation d'équivalence courante. Chaque classe d'équivalence a un nom qui est un entier compris ente 1 et |V|.
on associe à chaque classe d'équivalence la liste des sommets qui ont au moins un successeur dans la classe.

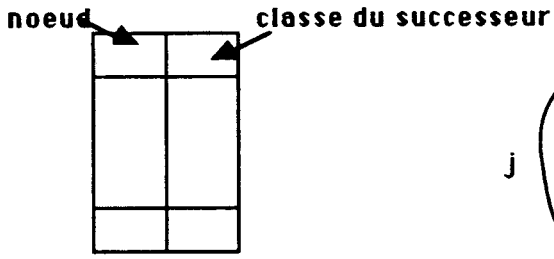
1	$v_i, \ldots, v_j$
2	$v'_i, \ldots, v'_j$
|V|	

$v_i, \ldots, v_j, v'_i, \ldots, v'_j$ sont des noeuds dont au moins un successeur appartient à la classse 1 et 2 respectivement.

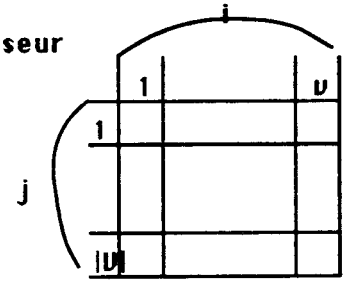
Opérations réalisées sur cette structure:

find(v): renvoit le nom de la classe d'équivalence contenant v.
list(e): renvoit la liste des sommets avec au moins un successeur dans la classe d'équivalence e.
union(e_1, e_2): Combine e_1 et e_2 en une seule classe d'équivalence s'appelant e_1.

ii/ la seconde structure est appelée table des signatures, dans laquelle on mémorise les sommets et leur signature. Chaque signature est soit un entier, soit un couple d'entier ordonné (i, j) dans l'intervalle $1 \leq i \leq v$, $1 \leq j \leq |V|$
On divise cette table en deux parties:



signature à un seul entier



signature à deux entiers

Opérations sur cette structure:

entier(v) : mémorise v avec sa signature courante dans la table des signatures
delete(v) : supprime v de la table des signatures si il est présent
query(v) : Si un sommet w de la table des signatures a la la même signature que v, alors renvoie w, sinon renvoie Λ.

iii/ Un ensemble appelé pending (en attente), qui correspond à la liste des sommets à entrer dans la table des signatures.
iv/ un ensemble appelé combine (combiner) qui correspond à la liste des couples de sommet dont les classes d'équivalence doivent être combinées.

L'algorithme est le même que le précédent avec les différences suivantes:
- Les sommets et leurs signatures sont mémorisés dans la table des signatures.
- Chaque fois que la signature d'un sommet change car la classe d'équivalence d'un de ses successeurs change, ce sommet est de nouveau entré dans la table.

- Si un autre sommet à la même signature, les classes d'équivalence des sommets sont combinées.
Dans ce cas on utilise la stratégie suivante:
Des deux anciennes classes, le nom de celle ayant le plus de prédécesseurs est donné à la nouvelle classe. Donc, les seules signatures qui sont modifiées lorsque deux classes sont combinées sont celles correspondant aux noeuds dont un successeur appartenait à l'ancienne classe ayant le moins de prédecesseurs.

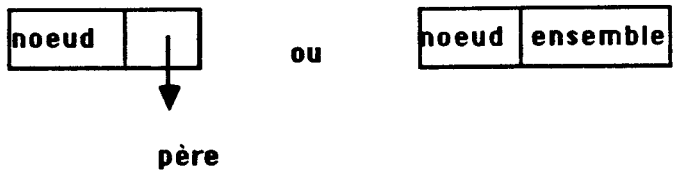
```
Algorithme:

pending := {v ∈ V | d(v) ≥ 1};
tant que pending ≠ ∅ faire
    combine := ∅
    pour chaque v ∈ pending faire
    si query(v) = Λ alors entier(v)
    sinon ajout(v, query(v)) à combine fsi
    fpour
    pending := ∅
    pour chaque (v, w) ∈ combine faire
    Si find(v) ≠ find(w) alors
      Si |list(find(v))| < |list(find(w))| alors
        pour chaque u ∈ list(find(v)) faire
        delete(u): ajouter u à pending
        fpour
        union(find(w), find(v))
      Sinon
        pour chaque u ∈ list(find(w)) faire
              delete(u); ajouter u à pending
        fpour
        union(find(v), find(w))
      fsi
    fsi
    fpour
ftq
```

Complexité

Nous allons étudier dans un premier temps la complexité des différentes opérations réalisées sur chacune des structures de données:

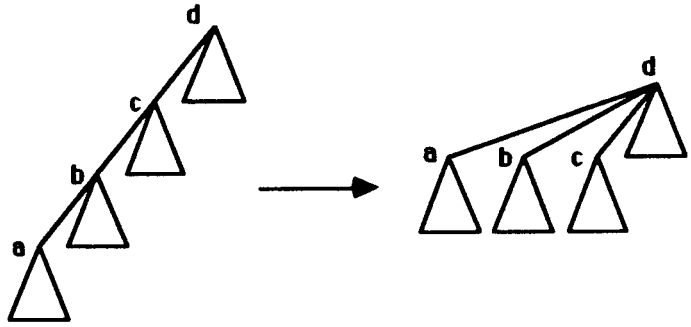
i/*Les opérations union et find (analysées par Tarjan):*
Chaque ensemble est représenté par un arbre. Chaque noeud de cet arbre représente un élément de l'ensemble et la racine de l'arbre

correspond à l'ensemble complet. Un noeud de l'arbre est formé du nom de l'élément et, soit d'un pointeur vers le père, soit du nom de l'ensemble, si il s'agit de la racine.



Cette notion a été introduite par Galler et Fischer [14].

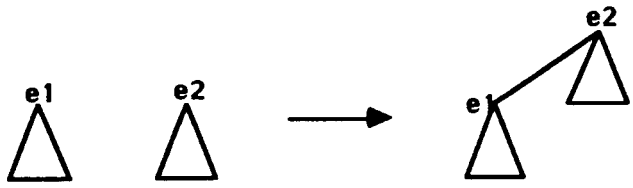
Exécution de find(x):
On localise la cellule contenant x, puis on suit les pointeurs jusqu'à la racine de l'arbre correspondant pour trouver le nom de l'ensemble. De plus on transforme l'arbre de la façon suivante:
Après une opération find, on rend tous les noeuds rencontrés pendant l'opération, fils de la racine de l'arbre (voir figure).



Cette transformation est due à Tritter.

Exécution de union(a,b):

On localise les racines appelées a et b, puis on rend l'une fille de l'autre, et le nom de la nouvelle racine est a.



A l'aide d'une telle implémentation, Tarjan [40] montre que le temps d'exécution de union et find est en O(1).
L'opération list possède également une complexité de temps en O(1), car elle consiste simplement à renvoyer le pointeur sur la liste de prédécesseurs.

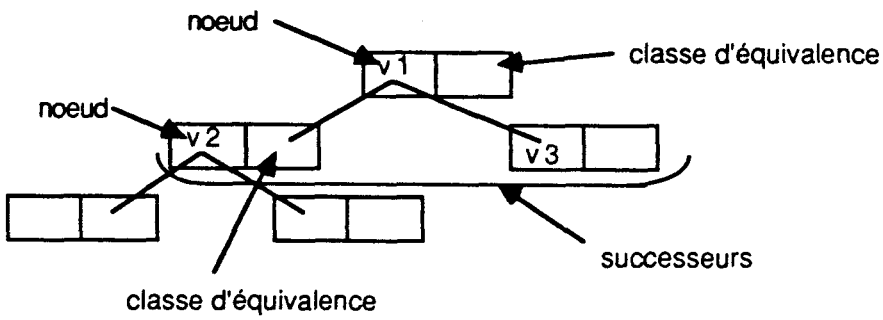
ii/ *Opérations utilisées sur la table des signatures:*

Comme on l'a vu, la table de signatures est divisées en deux parties:

* La première correspond aux signatures à 1 seul entier. C'est un tableau de n cases, dans ce cas chacune des opérations enter, delete et query a une complexité en O(1).

* La seconde peut être implémentée de trois façons différentes.

- Sous forme d'un arbre binaire équilibré.

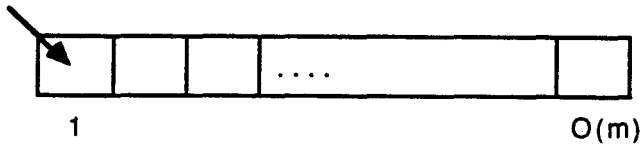


Dans ce cas on sait que la complexité de temps de chacune des opérations est O(n log n). Il n'y aura pas plus de cellules que de noeuds dans G dont l'espace mémoire sera en O(m).

- Sous forme d'une matrice n x $|V_1|$
Dans ce cas la complexité de temps de chacune des opérations est en O(1). L'espace mémoire est en O(mn).

- Avec une table de hachage.
On utilise une fonction de hachage f telle que si C correspond à l'ensemble des classes d'équivalence, on ait f: C x C $\rightarrow$ N.

nom du noeud dont f(signature) =1



Dans ce cas chaque opération est en O(1) en moyenne et l'espace mémoire est en O(m)

iii/*Complexité de l'algorithme complet.*

Bornons tous d'abord le nombre d'opération sur pending pendant l'éxécution de l'algorithme.
a- Au départ, on ajoute au plus $|V_1|$ éléments à pending.
b- Chaque sommet apparaît dans au plus deux listes de prédécesseurs, donc la taille totale de toutes les listes de prédecesseurs est de $2|V_1|$.
c- Chaque fois que deux listes de prédécesseurs sont fusionnées car les classes d'équivalences correspondantes sont combinées, seuls les sommets de la plus petite liste sont ajoutés à pending. Autrement dit chaque fois qu'un sommet est ajouté à pending, la longueur de l'une des listes de prédécesseurs double au moins.

Donc si on ajoute k fois le noeud v à pending, la liste de prédécesseurs dans la quelle il se trouvait va devenir égale à $2^k l$ si l est la dimension initiale de la liste en question. Cette liste pouvant au plus être égale à $2|V_1|$, on a au plus $2^k l = 2|V_1|$, donc k sera au plus égale à $\log 2|V_1|/l \approx \log 2|V_1|$ pour un noeud. Comme on a en tout $2|V_1|$ noeuds dans les listes de prédécesseurs on pourra au plus faire $2|V_1|\log 2|V_1|$ ajouts à pending. Donc finalement on pourra faire au plus $|V_1| + 2|V_1|\log 2|V_1|$ ajouts à pending durant l'éxécution de l'algorithme tout entier.

Bornons maintenant le nombre d'opération sur les classes d'équivalence. Il y a au plus $|V_1| - 1$ opérations d'union, puisque chaque opération d'union réduit le nombre de classes d'équivalence d'une, et il y a au plus $|V_1|$ classes d'équivalence à l'origine.
Le nombre d'opérations de list est borné par une constante multipliée par le nombre d'opérations d'union et donc de l'ordre de $|V_1| = O(m)$.
Le nombre d'opérations de find est borné par une constante multipliée par le nombre d'ajouts à combine, qui lui-même est borné par le nombre d'ajouts à pending. Donc le nombre d'opérations de find est en O(m log m). Le nombre d'opérations sur la table des signatures est borné par une constante multipliée par le nombre d'ajouts à pending et est donc en O(m log m).
D'après ce que l'on a dit précédemment, le temps d'exécution des O(m) opérations d'union et de list est en O(m). Les O(m log m) opérations de find se font en un temps de l'ordre de m log m.
Le nombre d'opérations sur la table des signatures fournit finalement une borne supérieure pour le temps d'exécution de l'algorithme complet. Cette borne dépendra donc de l'implantation de la table.
- Dans le premier type d'implantation, la complexité de temps d'une opération étant en O(log m). La borne sera donc en $O(m(\log m)^2)$.
- Dans le second cas, la complexité de temps d'une opération est en O(1). La borne sera donc en O(m log m).

-Dans le troisième cas la complexité de temps d'une opération est en moyenne en O(1). La borne sera donc en moyenne en O(m log m).

1.2- Algorithme calculant la clôture transitive d'une relation d'ordre (Oyamaguchi).

Le calcul de la clôture transitive d'une relation d'ordre, est très utile notamment pour résoudre le problème de l'accessibilité dans les systèmes de réécriture clos. L'algorithme que nous présentons ici est directement inspiré du précédent. Cependant on verra plus tard qu'il est impossible, en utilisant cette méthode, d'obtenir un algorithme aussi efficace que celui présenté juste avant, pour calculer la clôture transitive d'une relation d'ordre.
Soit un graphe acyclique G représentant un ensemble de termes. Chaque noeud d_i représente le sous terme dont il est la racine. On note $T_m(d_i)$ le sous terme représenté par d_i. Deux noeuds distincts dans le graphe représentent différents termes. $Lb(d_i)$ est égale au symbole représenté par le noeud d_i. $Ch(d_i)$ est la suite des fils de d_i. Ch(d_i, j) représente le jiéme fils de d_i. Notons que si $Ch(d_i) = e_1 \ldots e_n$ alors $T_m(d_i) = Lb(d_i)(T_m(e_1), \ldots, T_m(e_n))$.

Algorithme:

Soit M et N deux termes, et soit Eg un système clos. Notre graphe G représentera l'ensemble des termes $T_D = \{M, N\} \cup L_{Eg} \cup R_{Eg}$ où L_{Eg} et R_{Eg} sont les ensembles des termes gauches et droits des règles de Eg. Soit N_D l'ensemble des noeuds de G.
L'algorithme suivant calcule le sous-ensemble X de $\cup N_D \times N_D$ tel que $(d,d') \in X$ ssi $T_m(d) \rightarrow^*_{Eg} T_m(d')$ pour d et d' $\in N_D$ de façon à décider si $M \rightarrow^*_{Eg} N$.

(1) Soit X = Règle $\cup$ I où
    I = $\{(d,d') \mid d \in N_D\}$ et
    Règle = $\{(d,d') \in N_D \times N_D \mid T_m(d) \rightarrow T_m(d') \in Eg\}$
(2) Soit X = Clôture(X) où
    Clôture(X) est la clôture reflexive et transitive de X.
(3) **tant qu'il** existe d, d' $\in N_D$ tel que
    (d, d') $\notin$ X et Con(d, d')=vrai {Con(d,d') = vrai ssi Lb(d) = Lb(d') et pour Ch(d) = $d_1 \ldots d_n$ et Ch(d') = $d'_1 \ldots d'_n$, $(d_i, d'_i) \in X$ $1 \leq i \leq n$}
    faire X = clôture ($X \cup \{(d,d')\}$)
(4) **Si** (d, d') $\in$ X tel que $T_m(d) = M$ et $T_m(d') = N$
    alors vrai: accessible
    sinon faux.

Complexité:

Cet algorithme à une complexité de temps polynomiale de l'ordre de n ou n= |M| + |N| taille(Eg).
Cependant on peut montrer qu'il est impossible d'avoir un algorithme en n log n dans le cas de la clôture transitive.

Propriété: *Il n'existe pas d'algorithme calculant la clôture transitive en O(n log n).*

Preuve. On se donne le graphe g dont les noeuds sont définis par:

$Sg = \{e_{2n-1,\, n+i} \mid 0 \le i \le n\} \cup \{e_i \mid 0 \le i \le 2n\} \cup \{e_{n-i,\, i} \mid 0 \le i \le n\}$

et dont les arcs orientés et ordonnés sont les suivants:
$Ag = \{e_{i+1} \rightarrow e_i \mid 0 \le i \le 2n\} \cup \{$ e_{hl} pour les 2n+2 sommets doublement indicés$\}$

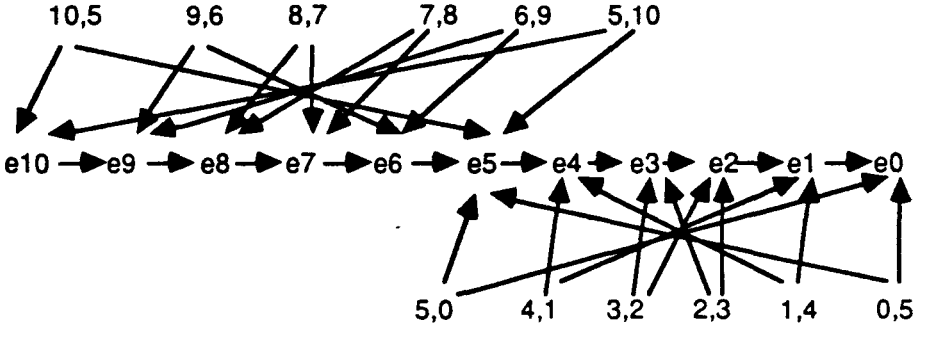
e_{hl} (1/ \2) : e_h e_l

On a donc environ 4n noeuds et 6n arêtes. On prend un ordre de départ vide et on fait la clôture transitive à l'aide de la règle d'inférence suivante:

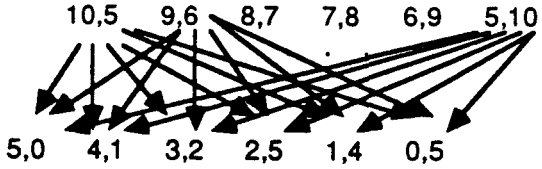
$$\frac{e \;(1/\ \backslash n)\; e_1 \ldots e_n \qquad e' \;(1/\ \backslash n)\; e_1' \ldots e_n' \qquad e_1 \text{ -*-> } e_1', \ldots, e_n \text{ -*-> } e_n'}{e \text{ -*-> } e'}$$

Ici on a dés le départ e_i -> e_j ssi i>j

On part donc du graphe suivant (exemple n=5)



La clôture transitive va nécessairement rajouter les flêches suivantes:



On est donc obligé d'ajouter environ n^2 flêches. L'algorithme de clôture transitive sera donc au minimum en $O(n^2)$ et ne pourra jamais être aussi simple que l'algorithme de clôture congruente.

Exemple d'implémentation de l'algorithme de Oyamaguchi:

i/Structures de données

L'algorithme utilisera deux structures de données:
- La première représentera le graphe.
- La seconde exprimera la clôture transitive.

On pourra par exemple avoir deux matrices de la forme suivante:

Soit la matrice M:

	1			m
1				
.		a		
.				
m				

m = nombre de noeuds dans le graphe

a= X si il existe un lien dans le graphe
rien sinon

Soit la matrice M'

	d1		di	dn
d1				
.dj			b	
dn				

n = nombre de termes différents dans le système plus deux (M et N) soit le nombre de parties gauches +le nombre de parties droites + M+N

b= X si on peut passer de di à dj
rien sinon

Exemple:

Soit le système suivant:

S:

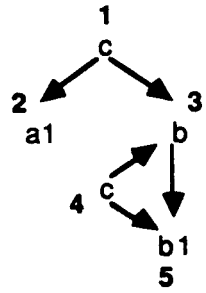
```
        c -> a1                b -> b1          b1 -> b
      /   \                    |                  |
      b    b1                  b1                 b1
      |
      b 1
```

```
        a1 -> b1          b -> a1
        |                 |
        b 1               b 1
```

et

```
M=  c                 et  N = b
   / \                       |
  a1  b                      b 1
      |
      b 1
```

On aura donc le graphe suivant:



La numérotation est telle que les fils soient numérotés de gauche à droite dans un ordre croissant.
Ce graphe sera représenté par:

	1	2	3	4	5
1					
2	X				
3	X			X	
4					
5			X	X	

comme chaque noeud du graphe représente un terme, la matrice M' repésentant la relation d'ordre aura la forme suivante:

	1	2	3	4	5
1	X				
2		X	X	X	
3		X	X		X
4				X	
5			X		X

ii/Procédure de calcul de la clôture transitive
On va procéder de la façon suivante:
A chaque fois que l'on ajoute une flêche d -> d' à la relation, on va regarder si il existe des flêches de la forme d' -> e, et de la forme e' ->d. L'ensemble de ces règles a une cardinalité, au plus égale à n^2-1.

1- Si on a des règles de la forme d' -> e on ajoute alors la règle d -> e et on applique récursivement la procédure sur cette nouvelle règle.

2- Si on a des règles de la forme e' -> d on ajoute la règle e' -> d' et on applique récursivement la procédure sur cette nouvelle règle.

Comme on ne pourra ajouter au plus que n^2 règles, on aura une complexité de temps totale pour l'algorithme entier en $O(n^4)$.

A l'aide de notre exemple la première opération de clôture transitive donnera la forme suivante à notre matrice M':

	1	2	3	4	5
1	X				
2		X	X	X	X
3		X	X	X	X
4				X	
5		X	X	X	X

Il ne reste plus qu'à examiner une par une les cases restantes de la forme (d,d') telle que Lb(d) = Lb(d') pour voir si les fils d1 . . . dl, d1' . . . dl' soient tels que d1 -> d1' . . . dl -> dl'. L'accés a une case que ce soit dans M ou dans M' est en O(1), la comparaison deux à deux des fils sera de l'ordre de l'arité maximale de l'alphabet. Comme on a au plus n^2 cases à examiner la complexité de temps de l'algorithme de Oyamaguchi ainsi implémenté sera en $O(n^6)$.

Conclusion:
Il est évident que ces méthodes, bien qu'intéressantes ne pourront jamais nous conduire à un algorithme de décision en temps réél du problème d'accéssibilité.
C'est pourquoi nous avons élaboré une autre méthode qui, comme nous allons le voir dans la section suivante, consiste à compiler le système de réécriture clos de façon à ce que l'on puisse, une fois la compilation faite, décider du problème d'accessibilité en temps réél.

2- Compilation du système de réécriture clos.

2.1-Création du GTT associé à un système de réécriture clos.

On va donc construire un transducteur d'arbre clos à partir d'un système de réécriture clos. En réalité, l'automate ascendant du GTT va reconnaître les parties gauches des règles de S et son automate descendant générera les parties droites des règles de S. Ses états interface assureront le lien entre les parties gauches et les parties droites.
Exemple:
Soit le système S suivant:

Σ ={ b1, q , q1' , p1 , b , q', p, a , c }
| | | / \ / | \

règles = 1- b(b1) -> b1 2- a(b1, q) -> q 3- q1' -> q'(q1')

4- q(q1') -> q1' 5- q1' -> a (q1', q1') 6- b(q'(q'1)) -> c(p1, p(p1), p1)

7-p1 -> p(p1) 8- a(b1, a(q, b1)) -> b(q1')

Nous aurons le GTT suivant formé des deux automates G et D associé au système S précédent.

Automate ascendant G	Automate descendant D
1- b1 -> e1 b(e1) -> i1	i1-> b1
2- b1-> e2 q-> e3 a(e2, e3) -> i2	i2 -> q
3- q1'-> i3	i3 -> q'(e4) e4 -> q1'
4- q1' -> e5 q'(e5) -> i4	i4 -> q1'
5- q1' -> i5	i5 -> a(e6, e6) e6 -> q1'
6- q1'-> e7 q'(e7) -> e8 b(e8) -> i6	i6 -> c(e9, e10, e11) e9 -> p1 e12 -> p1 e11 -> p1 e10 -> p(e12)
7- p1 -> i7	i7 -> p(e13) e13 -> p1
8- q -> e14 b1 -> e15 b1 -> e17 a(e14, e15) -> e16 a(e17, e16) -> i8	i8 -> b(e18) e18 -> q1'

Les états interfaces sont: I={ i1,i2,i3,i4,i5,i6,i7,i8}

En fait, on peut trouver une caractérisation simple d'un GTT qui est la suivante.

Caractérisation 1:

Soit B un GTT où ses automates associés sont G et D. On a

$\mathbb{B} = \{ (t(u_1, \ldots, u_n), t(v_1, \ldots, v_n) \mid$ *pour chaque* i $(1 \leq i \leq n)$ *il existe un* $q_i \in I$ *tel que* $u_i \in F(G, q_i)$ *et* $v_i \in F(q_i, D) \}$

ou $F(G, q_i) = \{t \in T(F) \mid t \vdash_G^* q_i\}$

et $F(q_i,D) = \{t \in T(F) \mid q_i \vdash_D^* t\}$

Complexité:

La création d'un tel GTT se fait en temps linéaire, soit en O(n), où n est le nombre de lettres présentent dans les règles du système de réécriture.

La complexité de l'espace mémoire est également en O(n), en effet, posons:

p le nombre de règles de l'automate G du GTT

et q le nombre de règles de l'automate D du GTT

On a en réalité autant de règles que de lettres présentes dans le système de réécriture. Donc si on a n lettres alors on a n règles, avec n = p+q.
De plus, on obtient également autant d'états distincts que de lettres - 1 (à cause de l'état interface qui apparaît dans les deux automates).
Comme ces états apparaissent deux fois dans les règles des automates, on obtient finalement 3n-1 symboles distincts, c'est à dire n lettres +2n-1 états.

2.2-Création du GTT qui simule le système de réécriture clos: calcul des ε-transitions.

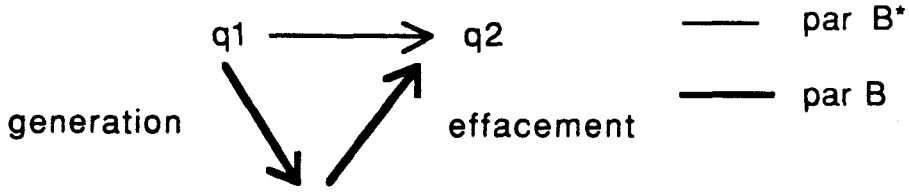
Pour obtenir le GTT qui simule le système de réécriture S, il faut faire l'itération du système obtenu à la première étape. D'après la proposition 6, l'itération de ce GTT simule S et de façon plus rapide que S lui-même.
En effet le principe est le suivant:

"Il ne sert à rien de générer pour effacer ensuite"

Etant donné le GTT B associé à S, nous allons créer le GTT B* en ajoutant des ε-transitions à l'aide des règles d'induction déjà citées précédemment.
Si G et D sont les deux automates associés à B, nous aurons alors les règles d'induction suivantes:

$$\frac{\begin{array}{c} e \text{ -D-> } f(e_1, \ldots, e_n) \\ e_1\text{-B*-> } e_1', \ldots, e_n\text{-B*-> } e_n' \\ f(e_1', \ldots, e_n') \text{ -G-> } e' \end{array}}{e \text{ -B*-> } e'}$$

Une telle transformation peut être illustrée par le diagramme suivant:



Algorithme:
1/ On prend une règle de l'automate descendant D.
2/ On examine, si on peut trouver la partie droite de la règle soit par exemple $f(e_1, \ldots, e_n)$, dans l'une des partie gauche des règles de l'automate ascendant G, soit par exemple $f(e_1', \ldots, e_n')$ telle que $e_1 \text{->} e_1', \ldots, e_n \text{->} e_n'$.

* Si c'est le cas, on crée une ε-transition avec en partie gauche, la partie gauche de la règle de D choisie (un état de D) et en partie droite, l'état dans lequel on arrive lorsque l'on applique la règle de G qui a été trouvée. Puis on calcule la clôture transitive sur l'ensemble des ε-

transitions déjà existantes à l'aide de la méthode étudiée p37. Enfin on prend la règle suivante de D et on recommence en 2/.
*Si ce n'est pas le cas, on recommence en 1/.

Complexité:

Posons
feuille(A)={ensemble des règles reconnaissant les feuilles dans A}
noeud(A)={ensemble des règles reconnaissant les noeuds de A}
card(feuille(A))=j_1 et card(noeud(A))=i_1.

Considérons tout d'abord les règles de feuille(D), pour chacune de ces règles, on examine au plus les p règles de G. De plus chaque fois que l'on ajoute une ε-transition on calcule la clôture transitive sur l'ensemble des ε-transitions déjà existante, ce qui se fait, comme on l'a vu, en $O(m^4)$, m étant le nombre de noeud du graphe, ce qui correspond ici au nombre d'états.

On a donc, si on ne considère que les règles de feuille(D), pour la création des ε-transitions une complexité égale à $j_1(p+m^4)$.

Considérons maintenant les règles de noeud(D). Pour chacune des i_1 règles de cet ensemble, on examine également les p règles de G, puis, lorsque l'on a trouvé une règle de G qui semble correspondre, on va examiner les k ε-transitions ajoutées lors de la première étape afin de trouver l'ε-transition (ou les ε-transitions) qui fera le lien entre la partie droite de la règle choisie de Noeud(D) et la partie gauche de la règle de G correspondante. En fait, l'accés à une ε-transition étant en O(1), cette étape aura donc une complexité de temps en O(1). Enfin la nouvelle ε-transition ajoutée, on calcul de nouveau la clôture transitive sur l'ensemble des ε-transitions. Donc une étape de ce type aura une complexité de temps égale à $i_1(p+m^4)$.

Si tous les états de D n'ont pas été examinés à la fin d'une telle étape, on recommence l'opération, soit au pire autant de fois qu'il y a d'états dans D, soit i_1 ce nombre d'états. Enfin, en supposant que l'on ajoute qu'une seule ε-transition à la fin de toutes les opérations précedentes, ce qui est le pire des cas et sachant que l'on ne pourra écrire au plus que m^2 ε-transitions, on recommencera donc les opérations précédentes au plus m^2 fois.

Finalement la complexité totale de cet algorithme est de l'ordre de

$$j_1(p+m^4) + i_1{}^2(p+m^4)m^2 \equiv O(m^6)$$

Exemple:

Reprenons l'exemple précedent et considérons les règles 3 et 4 de ce système.

La règle 3 a été décomposée comme suit: q_1' -> i_3 i_3 -> $q'(e_4)$
e_4 -> q_1'

et la règle 4 a été décomposée ainsi: $q1' \rightarrow e5$ $\quad i4 \rightarrow q1'$
$q'(e5) \rightarrow i4$
Considérons alors l'état e4, nous avons: $e4 \rightarrow q1'$, $q1' \rightarrow e5$ et $q1' \rightarrow i3$
Donc on a $e4 \rightarrow e5$ et $e4 \rightarrow i3$.
Considérons maintenant l'état i3 , on a: $i3 \rightarrow q'(e4)$
et par le dernier pas on a $e4 \rightarrow e5$ donc $i3 \rightarrow q'(e5)$ et on trouve $q'(e5) \rightarrow i4$ dans la décomposition de la règle 4. Finalement on en déduit l'ε-transition $i3 \rightarrow i4$.
Donc au lieu de faire les réécritures suivantes:
$i3 \rightarrow q'(e4) \rightarrow q'(q1') \rightarrow q'(e5) \rightarrow i4$
B* passera directement de i3 à i4.
Pour obtenir le système S', il nous reste maintenant à prouver le lemme suivant:
Lemme2: *Une GTT-relation peut toujours être associée à un GTT sans ε-transition* .
Preuve:
A partir du GTT (G,D) nous construisons le GTT (G',D') défini comme suit: EG' =EG et ED'=ED les règles de G' sont de la forme

$$f(e1', \ldots, en') \text{ -G'-> } e0'$$

si et seulement si

$$e1' \text{ -*-> } e1,$$
$$G$$
$$\ldots$$
$$en' \text{-*-> } en,$$
$$G$$
$$e0' \text{-*-> } e0,$$
$$G$$
$$f(e1, \ldots, en) \text{ -G-> } e0.$$

et la même chose pour D. Il est facile de voir que $t \rightarrow\!\!<\!\!- t'$ si et seulement (G,D)
si $t \rightarrow\!\!<\!\!- t'.\Diamond$
(G',D')
l'algorithme utilisé pour réalisé la suppression des ε-transitions est le même que celui supprimant les ε-règles de la forme $A \rightarrow B$ où A et B sont des variables, dans les grammaires. (Pour l'étude de ces algorithmes voir chapitre3 p80)
On peut donc à partir de B* obtenir le système S' qui correspondra au GTT B* sans ε-transition.
La réponse à notre problème pourra alors à l'aide de S' être donné en un temps quadratique.
Remarque: Le système S' n'est pas déterministe, la réduction du non déterminisme pourra être envisagée cependant, celle-ci comme nous l'avons vu a une complexité de temps exponentielle, ce qui augmente considérablement le temps de compilation de notre système.

3-Approche algébrique de la décidabilité du problème de l'accessibilité.

3.1- Le problème de l'accessibilité du 1er ordre.

On peut considérer deux cas distincts:

-Notre GTT B* est non déterministe (S')

-Notre GTT B* est déterministe (S'_{det})

Pour décider du problème de l'accessibilité nous devrons alors considérer une méthode différente selon le cas.

3.1.1- Décidabilité du problème de l'accéssibilité avec S'.

On peut remarquer que l'on peut répondre à la question t $\vdash$-*- t' ? en utilisant la forêt F définie en page 22 du chapitre 1.

En fait t $\vdash$-*- t' $\Longleftrightarrow$ $\phi^{-1}(t) \cap \phi'^{-1}(t') \cap F \neq \varnothing$

ϕ et ϕ' étant les morphismes définis au chapitre 1. Ceux-ci étant indépendants de t, t', S.

Donc, dans ce cas, on peut obtenir une complexité de temps égale à K(S) x $\| Mt \|^2$x $\| Mt' \|^2$ en omettant le temps d'accés.

De plus, on peut procéder de la même façon pour exprimer la forêt de toutes les transformations possibles de t par S. Nous appellerons cette forêt $\mathbb{F}(t)$ et on a:

$$\mathbb{F}(t) = \phi'(\phi^{-1}(t) \cap F)$$

La création de l'automate reconnaissant $\mathbb{F}(t)$ est réalisé avec l'algorithme suivant:

Notations:

Soient G et D les automates associés à S'.

Soit Mt et Mt' les automates reconnaissant t et t'.

Nous appellerons D_{inv}, l'automate obtenu à partir de D en renversant les flêches, donc D_{inv} est un automate ascendant.

1ere étape:

On réalise le produit de l'automate Mt par l'automate G, mais ceci en gardant les règles reconnaissant t.

La complexité de temps de cette opération est $O(\|mt\| x \|G\| x \|Mt_q\| x \|G_q\|)$. (Pour l'étude détaillée de l'algorithme réalisant le produit de deux automates, voir annexe 1 p128)

2ième étape:

On recherche à l'intérieur de l'automate ainsi obtenu, les règles qui mène à un couple d'état (q,i) où i est un état interface de S' et où q est un état quelconque de S'. A partir de là on ajoute toutes les règles de D_{inv} qui mènent à l'état interface en question (en remplaçant l'état interface par le couple d'état (q,i) correspondant). On obtient ainsi l'automate reconnaissant $\mathbb{F}(t)$, nous l'appellerons $\mathbb{A}(\mathbb{F}(t))$. La complexité de temps de l'ajout des règles de D_{inv} est en $O(\|D_{inv}\|)$.

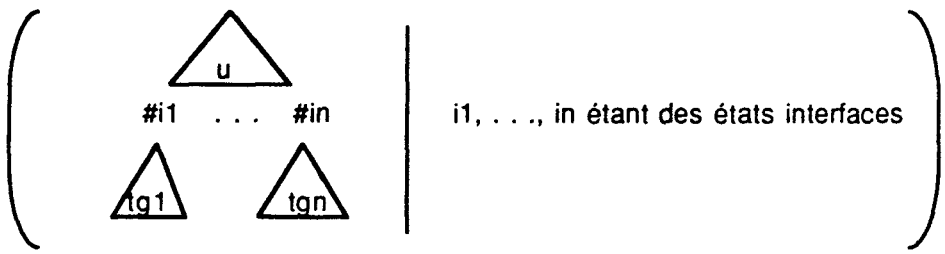
L'automate $\mathbb{A}(\mathbb{F}(t))$ obtenu, il ne nous reste plus qu'à décider si $\mathbb{F}(t) \cap t' \neq \varnothing$.

Pour cela on réalise l'intersection entre l'automate Mt' et $\mathbb{A}(\mathbb{F}(t))$, dont la complexité de temps est en $O(\|\mathbb{A}(\mathbb{F}(t))\| x \|Mt'\| x |\mathbb{A}(\mathbb{F}(t))_q| x |Mt'_q|)$. Ensuite il ne reste plus qu'à regarder si on a un couple d'état formé de l'état final de Mt et de l'état final de Mt' parmi les états de l'automate résultant de cette intersection. Ce qui est une opération en O(1).
Par conséquent, le temps de réponse totale à notre problème, en utilisant ce type de méthode est en:

$$O((|Mt|\|^2 x \|G\|^2 x \|D\|)+(\|Mt'\|^2 x \|\mathbb{A}(\mathbb{F}(t))\|^2)).$$

3.1.2-Décidabilité du problème de l'accessibilité avec S'_{det}.
On appelle S'_{det}, le système S' dont les automates associés G et D sont déterministes. En effet, on pourra répondre à notre problème en un temps beaucoup plus rapide si on se trouve dans un tel cas de figure.
Dans le premier chapitre, on a vu comment on pouvait construire l'automate reconnaissant la forêt F de toutes les transformations réalisables avec notre système de réécriture clos. On avait appelé un tel système, S''. Après l'obtention de ce dernier, il ne restait plus, pour répondre à la question t $|-^*-_S$ t'?, qu'à fusionner nos deux termes t et t' sous la forme de l'arbre t#t' et à examiner si cet arbre appartient à la forêt F. Mais on a pu remarqué que l'obtention d'un tel système était, en pratique, très longue. Cependant, il est possible de déterminiser G et D en un temps raisonnable et dans ce cas on peut reprendre le même type d'idée. Cette fois, on essaie, à partir de t et de t' de mettre en évidence une arborescence possèdant les même propriétés que celles des éléments appartenant à la forêt F, sans pour autant créer l'automate reconnaissant cette forêt.
a-Introduction des fonctions de marquage θ et θ'.
Soient t1 et t2 deux arbres donnés.
Soit $\Delta = F \cup \{\#i \mid i \in E_G \cap E_D\}$
Soit θ une fonction telle que:

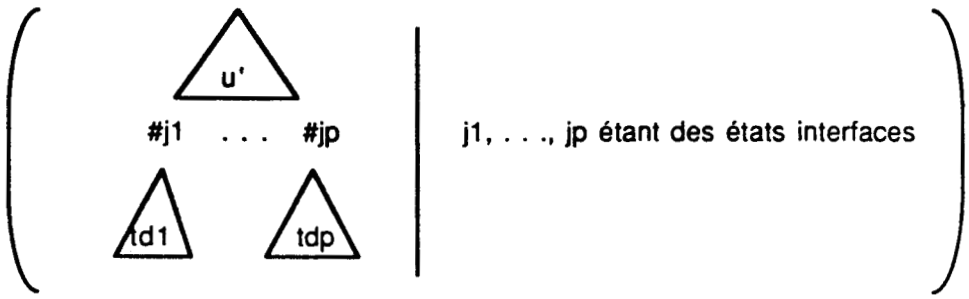
θ: T(F) -> T(Δ)
t1 |--> θ(t1) =



Les sous arbres tg1, . . ., tgn ont été reconnus par G. Cependant, a l'intérieur de ces sous-arbres, d'autres sous-arbres peuvent également avoir été reconnus par G, dans ce cas d'autres marques #ij peuvent s'y trouver.

Si t1 |-*-S t2, on doit trouver dans l'ensemble des arbres marqués, générés par θ, l'arbre obtenu dans le premier chapitre lorque l'on appliquait ϕ sur A.
Soit θ' une fonction telle que:

θ': T(F) -> T(Δ)
t2 |--> θ'(t2) =



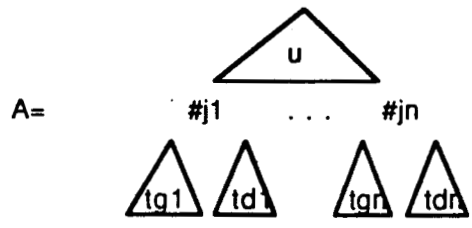
Les td1, . . ., tdp ont été reconnus par D. Ici aussi les différents éléments de θ'(t2) ne différent que par la position des marques #ij.
De même que pour θ, si t1 |-*-S t2, on doit trouver parmi les éléments de θ'(t2), l'arbre obtenu dans le premier chapitre lorque l'on appliquait ϕ' sur A.
θ(t1) et θ'(t2) étant obtenus, il ne nous reste plus qu'à les comparer de manière à mettre en évidence la proposition suivante:
Proposition 4:
t1 |--St2 ssi ∃t ∈ θ(t1) et t' ∈ θ'(t2) tels que:*

où u1=u2 et où les feuilles étiquetées par les suites #i1 . . . #in et #i1' . . . #in' sont telles que:

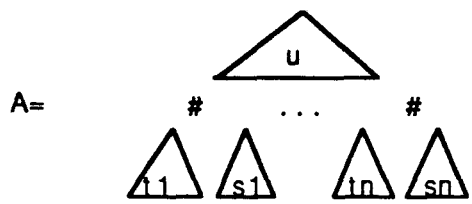
$$i1=i1', \ldots, in=in'$$

où i1, . . ., in et i1', . . ., in' ∈ EG∩ED.

Preuve:
(1)⇐ évidente, car si on trouve deux arbres t et t' vérifiant la propriété, on pourra par superposition des deux arbres, obtenir l'arbre A.



où tg1, . . .,tgn ∈ F(G)
td1, . . ., tdn ∈ F(D)
j1, . . ., jn ∈ $E_G \cap E_D$.
Or cette arborescence possède les propriétés des éléments de la forêt F reconnue par S". donc on peut en déduire que t $\vdash^*_S$ t'.
(2)⇒ si t $\vdash^*_S$ t', alors, en superposant t et t' on peut obtenir le terme A = t#t' ∈ F, forêt reconnue par S".
L'arbre A à la forme suivante:



Or à partir de A, on peut mettre en évidence notre propriété:

1- ∃u commun a t et à t'

2- Si A ∈ F alors t1, . . ., tn peuvent être reconnus par G, et s1, . . ., sn peuvent être reconnus par D_{inv}. Après reconnaissance des sous-arbres t1, . . ., tn par G, on obtient une liste i1, . . ., in d'états interface.
De même après reconnaissance des sous-arbres s1, . . ., sn par D_{inv} on obtient une liste i1', . . ., in' d'états interface.
Mais comme A ∈ F, on doit avoir i1=i1', . . ., in=in'.
Finalement, on peut remarquer que ces listes d'états correspondent aux marques déposées par nos fonctions θ et θ'. donc la propriété est vérifiée.◊

b-Algorithmes et étude de la complexité.
On peut remarquer que les opérations de marquages ont une complexité en temps très réduite. En effet, comme G est déterministe, l'opération de marquage réalisée avec la fonction θ, s'exécute en un temps linéaire, simplement en appliquant un algorithme de reconnaissance de t par G. De même, comme D_{inv} est déterministe, l'opération réalisée avec la fonction θ', s'exécute en un temps linéaire, en appliquant un algorithme de reconnaissance de t' par D_{inv}.
Nous allons donc à l'aide de l'algorithme suivant marquer tous les sous-arbres de t reconnus par G (resp. tous les sous-arbres de t' reconnus par D_{inv}) afin d'obtenir deux nouveaux automates qui reconnaîtrons respectivement, la forêt engendrée par θ et la forêt engendrée par θ'.
L'algorithme de marquage est le suivant:

MARQUAGE(X,L,Y,E)
x: noeud
l: liste des fils du noeud x
y: état dans lequel on arrive lorque l'on a reconnu le noeud x (avec les règles de l'automate Mt, ou de l'automate Mt')
e: état dans le quel on arrive lorsque l'on a reconnu le noeud x (avec les règles de l'automate G ou de l'automate D_{inv})
Début
Si il existe une règle de G (ou de D_{inv}) reconnaissant le noeud x avec la liste l **alors**
(1) On garde l'état e de G (ou de D_{inv}) dans lequel on arrive après avoir fait la reconnaissance

(2) On regarde si cet état est un étqt interface.
Si oui **alors** on ajoute la règle suivante:
#(e) -> etqt(y) en début de la liste de règle de l'automate Mt (ou Mt')
sinon rien.
fsi
sinon on garde un état fictif 'p' de façon à pouvoir continuer l'exploration de l'arbre. (Remarque: Les pères des noeuds x, ne peuvent plus être reconnu par G (ou D_{inv}))
fsi
fin

ETUDE DES NOEUDS(X,L,Y,E)
Début
si la lettre x est une feuille **alors** marquage(x,l,y,e)
sinon si la lettre x est un noeud **alors**
pour chaque fils fi de x **faire**
(1) prendre les règles de Mt (ou Mt') qui conduisent à ce fils: <xi,li> -> etat(fi)
(2) étude des noeuds(xi,li,fi,ei)
(3) garder chaque ei dans une liste l'
fin
marquage(x,l',y,e)
fsi
fsi
fin

PROGRAMME PRINCIPAL
Début
Prendre la règle qui reconnaît le noeud racine de t ou de t'
<x,l> ->etat(y) /* x: noeud racine l: liste des fils.*/
étude des noeuds(x,l,y,e)
/* exploration de l'arbre t (ou t') avec l'opération de marquage*/
fin

On appelle les deux automates obtenus, en appliquant cet algorithme sur t et t', Mt_m et Mt_m'.
On peut remarquer que nous faisons une et une seule opération de marquage sur chaque noeud de l'arbre considéré (c'est à dire, pour chaque règle de l'automate associé)
De plus, on a vu tout à l'heure que l'opération de marquage était réalisée en un temps linéaire. Donc on peut en déduire que la création des automates Mt_m et Mt_m' est réalisée avec une complexité de temps en O(||Mt||+||Mt'||).
Maintenant il ne nous reste plus qu'à comparer les deux forêts reconnues par Mt_m et Mt_m' de façon à trouver deux arbres t1 et t1' qui se superposent. Cette opération de comparaison est réalisée en un temps très court à l'aide de l'algorithme suivant:

```
COMPARER NOEUD(Y,Y1,E,E')
Début
si y=<x,L> et y1=<x1,L1> alors
     si x=x1 alors considérons chaque liste
                L = e1, e2, . . ., en et
                L1 = e1', e2', . . ., en'
                (ei et ei' sont des états)
                pour chaque couple d'états (ei,ei') faire
                (1) rechercher les règles qui conduisent à ces deux états
                c. à d.: yi -> état(ei)
                      yi' -> état(ei')
                (2) comparer noeud(yi,yi',ei,ei')
                finpour
     sinon echec
     fsi
sinon si y =#(e1) et y1 =<x1, L1> alors
                (1) prendre la règle telle que y =<x,L>
                (Cette règle existe toujours)
                (2) comparer noeud(<x,L>, <x1,L1>,e,e')
     sinon si y = <x,L> et y1 = #(e1) alors
                (1) prendre la règle telle que y1 = <x1,L1>
                (Cette règle existe toujours)
                (2) comparer noeud(<x,L>,<x1,L1>,e,e')
     sinon si y = #(e1) et y1=#(e1) alors OK
          sinon prendre les règles telles que y = <x,L> et y1 = <x1,L1>
                comparer noeud(<x,L>,<x1,L1>,e,e')
          fsi
     fsi
     fsi
fin
```

On peut remarquer que, ici aussi, on ne considère qu'une seule et une seule fois chaque règle des automates Mt_m et Mt_m'. Donc cet algorithme est exécuté en un temps linéaire. La réponse à notre problème sera donc donnée avec une complexité en temps de l'ordre de $||Mt_m|| + ||Mt_m'||$. On peut comparer ce résultat avec le résultat de complexité obtenu dans le papier de Oyamaguchi M [35]. Leur algorithme opère en un temps polynomial de n ou $n=||Mt||+||Mt'||+||S||$. Dans notre cas, la complexité est devenue linéaire et la taille de S n'est pas considérée. Ce fait peut être expliqué car nous avons fait une première opération de compilation sur notre système de réécriture (celle-ci n'est faite qu'une seule fois). Donc après la réalisation de cette opération, on peut poser autant de questions que l'on veut sans être obligé de la refaire, c'est pourquoi nous gagnons énormément de temps.

Remarque: Si G et D_{inv} ne sont pas déterministes, la réduction du non-déterminisme sur ceux-ci est plus faisable en pratique que sur S'', car ils ont un plus petit nombre de règles que S'' et donc le temps d'exécution de cette opération est réduite.

3.2-Les différents problèmes de l'accessibilité du 2nd ordre.

Soient deux forêts F et F'.
Soit $\mathbb{F}(F)$ la forêt de toutes les transformées, réalisables avec le système de réécriture clos S, de la forêt F.
Deux types de problème peuvent être considérés:
- Est-ce que $\mathbb{F}(F) \cap F' \neq \emptyset$? Soit est-ce qu'il existe au moins un élement commun à $\mathbb{F}(F)$ et à F'?
- Est-ce que $\mathbb{F}(F) \subset F'$? Soit est ce qu'il est possible de transformer la forêt F en la forêt F' à l'aide du système de réécriture?

Dans le premier cas de figure, il suffit d'utiliser la méthode étudiée en 3.1.1. Décider si l'intersection est différente du vide ne pose pas de problème, il suffit de voir si il existe dans notre automate produit un état de la forme (i,j) où i est un etat final de $\mathbb{A}(\mathbb{F}(F))$ et où j est un état final de l'automate reconnaissant F'.
Au contraire, dans le deuxième cas de figure, le problème pose quelques difficultés. En effet pour décider si $\mathbb{F}(F)$ est incluse dans F', cela revient à décider si l'intersection entre la forêt $\mathbb{F}(F)$ et la forêt complémentaire de F' est vide, pour cela on est amené à déterminiser l'automate produit afin de voir si il n'existe pas d'état de la forme (i',j') où i' est un état final de $\mathbb{A}(\mathbb{F}(F))$ et j' est un état final de l'automate reconnaissant le complémentaire de F', ce qui accroît de façon exponentielle le temps de réponse à notre problème.

4-Les explications: Quelles règles sont utilisées pour passer de t à t'?

A l'aide de notre algorithme, nous avons maintenant la possiblité de répondre à la question: Est-ce que t peut se transformer en t' à l'aide des règles de S? en temps réél. Mais, si il est possible de le faire, on ne dit rien sur la façon dont on transforme t en t'. Soit, quelle règles doit-on utiliser, et quel est le plus court chemin, pour passer de t à t'?

C'est pourquoi, nous avons établie un algorithme qui permet de justifier la réponse de notre logiciel dans le cas où on travail avec S'_{det}.

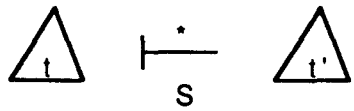
L'idée est la suivante:

Il s'agit de faire le raisonnement à l'envers. Soit, sachant que l'on peut effectivement transformer t en t', et sachant par quel état interface (puisque ceci nous est fourni par l'algorithme de comparaison des arbres marqués), retrouver la règle du système de réécriture clos correspondante puis connaissant les parties gauche et droite de cette règle, recommencer le processus afin de descendre de plus en plus bas dans l'arborescence jusqu'à ce que l'on n'ait plus de transformation à faire.

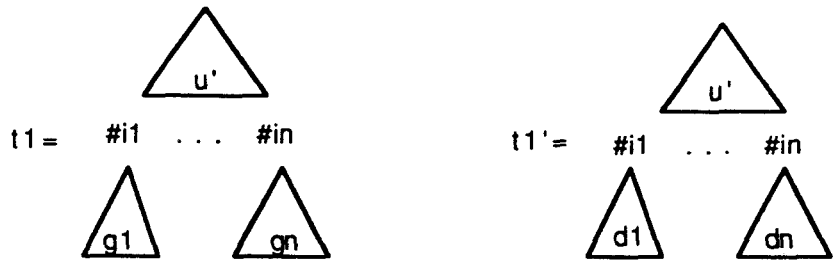
L'idée ainsi exposée, on peut constater que l'on va de nouveau utiliser nos algorithmes de marquage et de comparaison.

Plus exactement, voici comment on procède:

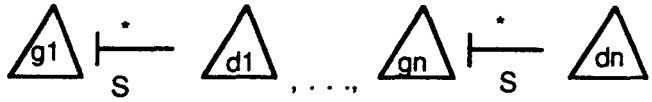
On part de la situation suivante:



Après marquage et comparaison, on obtient deux arbres superposables:



on va alors se trouver en face des nouvelles situations suivantes:

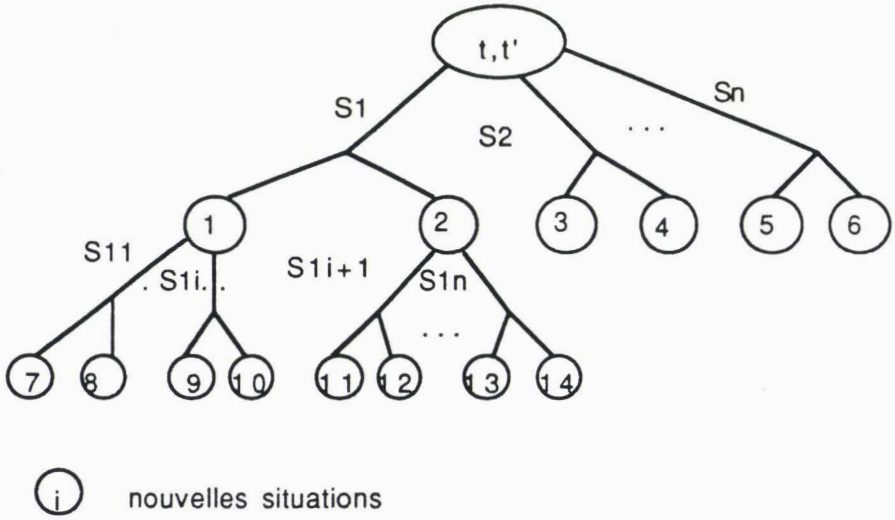


On va donc rechercher les règles correspondant aux états interfaces i1, . . ., in. Car on sait qu'à un état interface correspond une règle du système de réécriture clos.

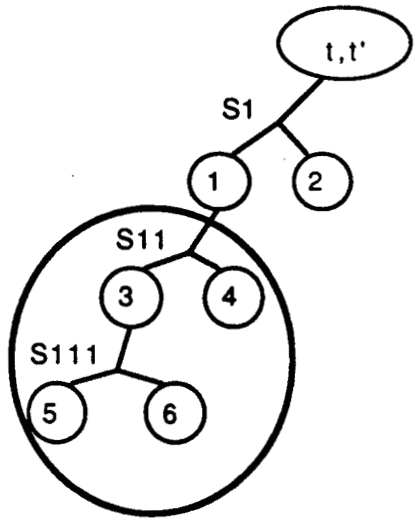
Soit r1, . . ., rn ces règles.

Soit (P11,P12), . . ., (Pn1,Pn2) les couples de parties gauches et droites de ces règles.
On va alors engendrer de nouvelles situations qui sont les suivantes:

Pour chacune de ces situations, on va alors appliquer récursivement les mêmes opérations que celles appliquées à la situation t|-*-St'.
On peut alors remarquer que l'on va développer une arborescence dont les noeuds correspondent aux différentes situations rencontrées, et dont les liens correspondent aux règles utilisées.
On peut représenter cette arborescence de la manière suivante:
Soitent t et t' les données de départ.
Soient S1, S2, . . ., Sn et S11, . . ., S1i, S1i+1, . . ., S1n des règles appartenant au système de réécriture clos S, on a alors:



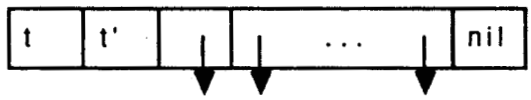
Cependant, pour que notre algorithme s'arrête, il ne faut pas que l'on passe deux fois par la même situation. Soit, si on parle en terme d'arborescence, il ne faut pas que celle-ci possède une branche qui possède plusieurs fois le même noeud.
pour résoudre notre problème on va mémoriser cette arborescence, afin que chaque fois que l'on rencontre une nouvelle situation on puisse vérifier si celle-ci n'a pas déjà été rencontrée dans la branche que l'on est en train de développer.
On peut alors se trouver dans la situation suivante:



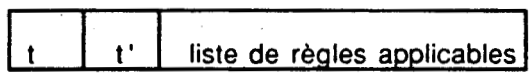
Si on a (4) = (1) alors il faut changer de règle, car la règle S11 fait passer deux fois par la même situation. dans ce cas il faut également supprimer tout ce qui à été engendré par S11. (Soit ce qui est entouré).

Algorithme:

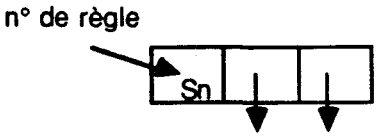
Structures de données:
Notre structure de données va être constituée de deux types d'objets qui sont les suivants:
Le type situation de départ:
c'est un enregistrement formé des champs suivants:
-Arbre de départ
-Arbre d'arrivé
-liste de pointeurs pointant sur des objets de type sous-situation



Le type sous-situation:
C'est également un enregistrement formé des champs suivants:
-Sous-arbre de départ
-Sous-arbre d'arrivée
-Liste des règles applicables



Le type règle applicable:
enregistrement formé de trois champs:
-numéro de la règle
-deux pointeurs pointant sur deux objets de type situation de départ.



On a donc une structure récursive, comme on pouvait s'y attendre.
Algorithme:

EXPLICATION(T1,T2)

Début
/*t1 et t2 sont les arbres de départ*/

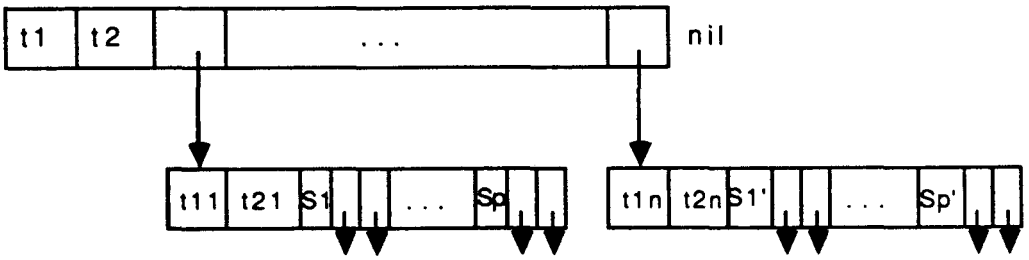
marquage(t1)
marquage(t2)
comparaison(t1,t2)

/*La comparaison génére l'ensemble $C_{t1\text{-}t2}$ pour la situation courante:

$$C_{t1\text{-}t2} = \{(t1_k, t2_k, e_{ik}, e_{jk} \mid k = 1, \ldots, n\}$$

où $t1_k$ et $t2_k$ sont les kièmes sous-arbres de t1 et de t2 et où e_{ik} et e_{jk} sont les états interfaces auxquels on arrive lorque l'on reconnaît $t1_k$ et $t2_k$ respectivement. */

Pour chaque couple (e_{ik}, e_{jk}) on définit l'ensemble elc_k des règles que l'on peut appliquer.
/* $elc_k(e_{ik}, e_{jk})$ = {Sz = numéro de règles | z = 1 . . . p} */
On crée alors l'objet de type situation de départ suivant:



Pour k = 1 . . . n **faire** /* k correspond à un sous-arbre */
Pour z = 1 . . . p **faire** /* z correspond à une règle*/
- Soit Sz: gz ->dz
à chaque pointeur de Sz on fait correspondre les deux nouvelles situations

$t1_k$, g_z, et d_z, $t2_k$,

-Vérifier en remontant de pointeur en pointeur dans la structure si ces situations n'ont pas déjà été rencontrées.
Si non **alors**
Si $t1_k \neq g_z$ **alors** explication($t1_k$, g_z)

sinon création de l'objet situation de départ:

t1k	gz	nil

fsi
Si $d_z \neq t_{2k}$ **alors** explication(d_z, t_{2k})
sinon création de l'objet situation de départ:

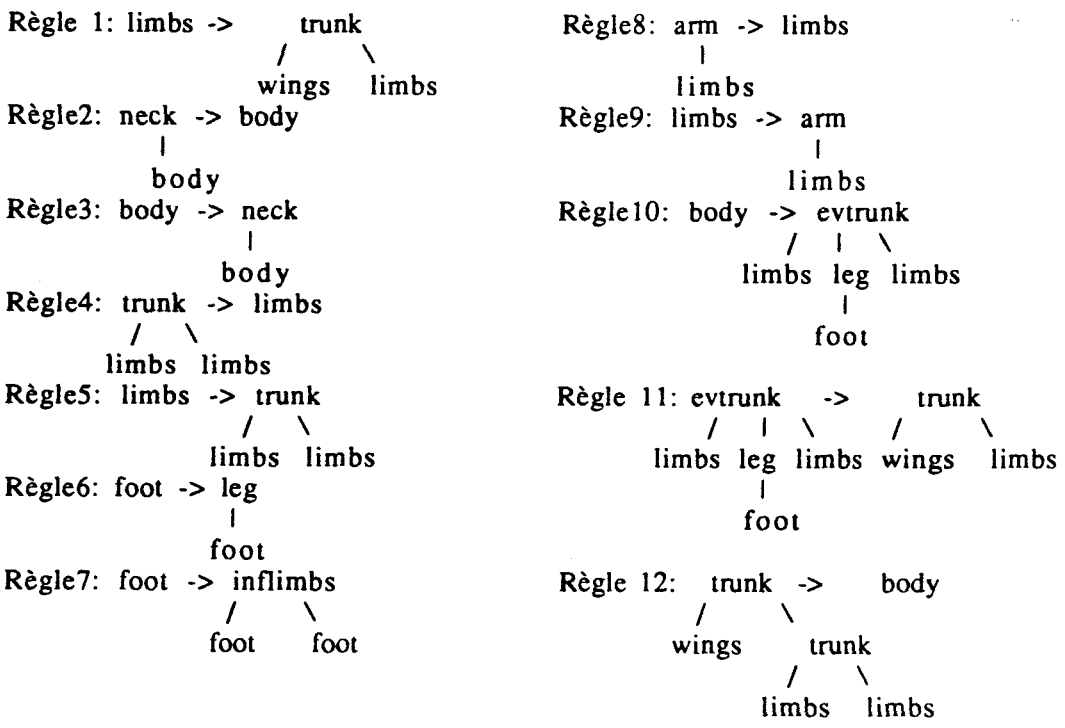
dz	t2k	nil

fsi
sinon mettre les pointeurs de la règle Sz à nil
tantque z + 1 > p et qu'aucune règle n'a été trouvée **alors**
remonter de deux pointeurs afin d'aller chercher la règle précédemment choisie, mettre ses pointeurs à nil et effacer tous ce qui correspond à ces pointeurs. Soit z' le numéro de cette règle. z = z', k=k', et p = p'.
ftq
fp
fp

fin

Remarque: Une fois cet algorithme exécuter, pour retrouver les règles dans l'ordre de leur utilisation, il suffit de relire notre structure récursive de bas en haut et de gauche à droite.

Exemple:
Soit le système de réécriture suivant:



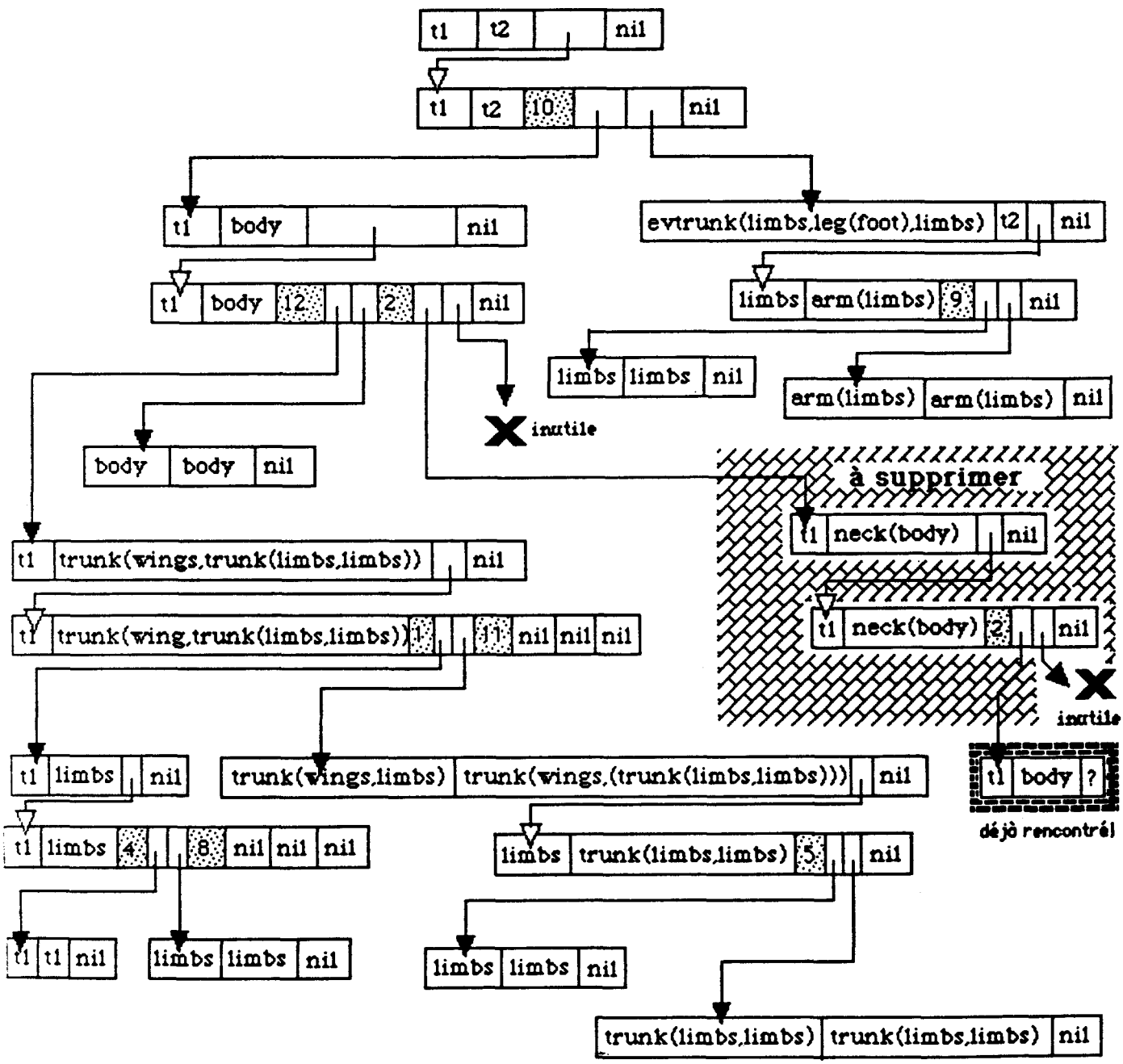

t1 t2 nil
t1 t2 10 nil
t1 body nil
evtrunk(limbs,leg(foot),limbs) t2 nil
t1 body 12 2 nil
limbs arm(limbs) 9 nil
limbs limbs nil
arm(limbs) arm(limbs) nil
inutile
body body nil
à supprimer
t1 neck(body) nil
t1 trunk(wings,trunk(limbs,limbs)) nil
t1 neck(body) 2 nil
t1 trunk(wing,trunk(limbs,limbs)) 1 11 nil nil nil
inutile
t1 limbs nil
trunk(wings,limbs) trunk(wings,(trunk(limbs,limbs))) nil
t1 body ?
déjà rencontré
t1 limbs 4 8 nil nil nil
limbs trunk(limbs,limbs) 5 nil
t1 t1 nil
limbs limbs nil
limbs limbs nil
trunk(limbs,limbs) trunk(limbs,limbs) nil

Figure 3

On se demande alors quelles sont les réécritures à effectuer pour passer de l'arbre t à l'arbre t', t et t' étant les arbres suivants:

Pour cela on va développer la structure de données décrite sur la figure 1.
En relisant cette structure de bas en haut et de gauche à droite on obtient la suite de règles suivante:

4, 1, 5, 12, 10, 9.

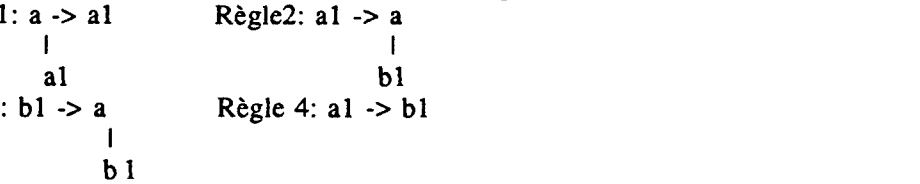
Remarque:
La comparaison de deux termes marqués peut s'effectuer de deux manières différentes.
En effet supposons que l'on se trouve en face du cas suivant:

C'est à dire que l'on a rencontré une marque commune aux deux arbres, celle-ci décorant le même noeud dans chacun des deux arbres.
Deux méthode peuvent alors s'appliquer.

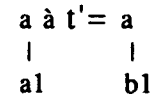
1- Soit, on considère que l'on a trouver ce que l'on cherchait et on passe au sous arbre frère.

2- Soit on considère que dans ce cas, il est possible que l'on fasse une étape de réécriture inutile (Puisque l'on ne modifie pas le sommet de l'arbre) on garde alors la solution trouvée en attente et on continue de parcourir les deux arbres afin d'examiner si il n'y a pas d'autres marques communes plus bas.

Exemple:
Si on prend le système de réécriture simple suivant:



et que l'on veut savoir comment passer de t= a à t'= a



Après compilation le système devient:

```
a1 -> 5                    4 ->b1
a -> 3                     5 -> a
|                              |
1                              6
a -> 7                     7 -> b1
|                          3 -> a
1                              |
a -> 1                         5
|                          5 -> a
1                              |
                               5
```

Les états interfaces étant les états 2, 3, 5, et 7.
Après réduction du non déterminisme on obtient alors les deux automates suivants:

```
a1->{1,5,7,3}              a1 ->{2}
b1-> {5}                   b1 -> {7,6,4}
a -> {1,3,7,2}             a ->{3,5}
|                          |
{1,5,7,3}                  {7,6,4}
a -> {1,3,7,2}             a -> {3,5}
|                          |
{1,3,7,2}                  {3,5}
```

Après marquage des deux arbres t et t' on obtient:

```
a {1,3,7,2}                a {3,5}
|                          |
a1 {1,5,7,3}               b1 {7,6,4}
```

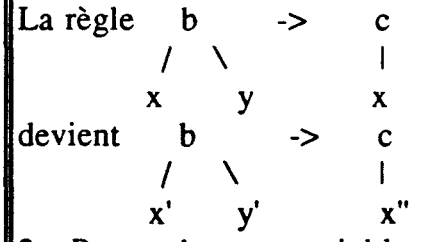
Avec la première méthode on a les marques {1,3,7,2} et {3,5}, ce qui correspond à l'état interface 3 qui lui-même correspond à la règle 2. Pour obtenir le chemin complet il faudra alors effectuer une autre étape de ce genre, ce qui nous fournira alors les deux règles suivantes: 1, 2.
Avec la deuxième méthode on trouve les marques {1,5,7,3} et {7,6,4}, ce qui correspond à l'état interface 7 qui lui-même correspond à la règle 4.
On a alors déjà terminer notre recherche, pour passer de t à t' il suffira d'appliquer la règle 4.

Comme on peut le remarquer dans l'exemple précédent on obtient le plus court chemin à l'aide de la méthode 2. Cependant on ne peut pas pour cela en déduire que la deuxième méthode est meilleure que la première. En effet si on prend un autre système de réécriture le phénomène contraire peut très bien se produire.
Dans chacun des cas, on obtiendra donc des chemins différents et incomparables car on ne pourra pas passer de l'un a l'autre en supprimant simplement des boucles (retour à des situations précédemment recontrées (a(a1) -> a1-> a(a1) -> a1 -> b1 est équivalent à a(a1) -> a1 -> b1)). Il faudra donc pour obtenir tous les chemins possibles et par conséquent le plus court chemin, appliquer les deux méthodes (ce qui revient à examiner toutes les façons possibles de passer d'un sous-arbre à un autre)

5- Application: Approximation du problème de l'accessiblité pour un s.d.r quelconque dans une algèbre avec sortes ordonnées.

Si on reprend la définition d'un système de réécriture séparé donnée dans le premier chapitre, on peut remarquer que, du point de vue théorique celui-ci n'est pas très différent d'un système de réécriture clos. En effet si on identifie les sortes à des nouveaux symboles de l'alphabet sur lequel est défini notre système, une règle sortée séparée devient une règle close. D'une autre façon, on peut également considérer ces noms de sortes comme des états finaux d'automates reconnaissant les sortes en question.
Dans ce cas, lors de la construction du GTT associé, il suffit de greffer les règles des automates reconnaissant les sortes sur les règles reconnaissant les parties gauches et droites du système de réécriture. On obtient ainsi un GTT à partir duquel on pourra réaliser les opérations de compilation habituelles d'un système de réécriture.
L'intérêt de considérer ce type de système de réécriture est le suivant:
Du point de vue pratique, on peut approximer n'importe quel système de réécriture par un système de réécriture séparé et spécifier des sortes ainsi que d'autres propriétés de ce type. L'un des buts de notre logiciel est alors d'aider à établir, prouver ou réfuter des propriétés de programmes.
Nous appliquerons l'algorithme suivant:

Soit R un système de réécriture d'arbres,
1- Pour chaque règle de R, distinguer les différentes occurrences des variables.
Exemple:



2- Pour chaque variable les remplacer à gauche par les sortes ascendantes et à droite par les sortes descendantes.

Théorème d'appoximation:
Soit RG le système obtenu après application de l'algorithme précédent.
Si t et u sont des termes, on alors

$$t \xrightarrow[R]{*} u \quad \Rightarrow \quad t \xrightarrow[RG]{*} u$$

La preuve de ce théorème est évidente.
A partir de là, nous obtenons en conséquence les propriétés suivantes:

Propriété 1: *Si u est un terme en forme normale pour RG alors u l'est aussi pour R.*

Propriété 2: *Si R est linéaire gauche (aucune variable n'apparaît plus d'une fois dans n'importe quelle partie gauche de règle) alors u est en forme normale pour RG ⇔ u est enforme normale pour R.*

Propriété 3: *Soit F une forêt, on a $R(F) \subset RG(F)$*

Propriété 4: *Si R est linéaire gauche alors*
R(F) ∩ ensemble des formes nomales de $R \subset RG(F)$ ∩ ensemble des formes normales de RG.

Ceci nous permet de résoudre plusieurs types de problèmes d'accessibilité du premier et du second ordre.
Problèmes du premier ordre:
1- t -*-> u ⇒ il se peut que t -*-> u
RG R
2- non(t -*-> u) ⇒ non(t -*-> u)
RG R
Problèmes du second ordre:
Soit F et S deux forêt, on a alors
1- $RG(F) \supset S \Rightarrow$ il se peut que $R(F) \supset S$
2- $RG(F) \subset S \Rightarrow R(F) \subset S$
3- $RG(F) \cap S = \varnothing \Rightarrow R(F) \cap S = \varnothing$
Si F est linéaire gauche on peut énoncé les mêmes questions pour les formes normales.

On peut également établir le théorème suivant.
Théorème de contrôle:
Sachant que le calcul par valeur ("innermost control") peut être exprimé par un contrôle reconnaissable si R est un système de réécriture linéaire gauche, on a:
Quelque soit R un système de réécriture linéaire gauche, il existe R' qui ne diffère que par les sortes, tel que,

$t \xrightarrow{R\ \text{"innermost"}} u \quad \Leftrightarrow \quad t \xrightarrow{R'} u$

Preuve:
Comme R est linéaire gauche, l'ensemble Norm(R) des formes normales est reconnaissable.
On peut donc définir les nouvelles sortes du système R' de la façon suivante:
Soit S la sorte obtenue après application de la règle g -> d de R.

Soit Semi-Norm(R)={t | t n'est pas réductible ailleurs qu'à la racine}
La nouvelle sorte S' deviendra alors égale à S ∩ Semi-Norm(R).◊

L'algorithme précédent peut être illustré par l'exemple suivant.
Exemple:
Soit la signature (S,F) multi-sortée telle que
S = {Mon, Pol, Xfv, Npol, Horn, Cst, Hmul}
F={+, *}
Chaque sorte est définie ainsi:
La sorte Mon des monômes.
Elle est générée par l'opérateur binaire * , une variable X et les constantes A, B, . . ., Z.

$X \rightarrow Xfv$ $A \rightarrow Cst, \ldots, Z \rightarrow Cst$
$* \; Cst \times Xfv \rightarrow Mon$ $* \; Mon \times Xfv \rightarrow Mon$

La sorte Pol des polynômes.
Elle est générée par la somme des monômes à l'aide de l'opérateur + .

$Mon \rightarrow Pol$ (inclusion de sortes)
$+ \; Pol \times Pol \rightarrow Pol$ $+ \; Pol \times Cst \rightarrow Pol$
$+ \; Cst \times Pol \rightarrow Pol$

La sorte Npol des polynômes en forme normale.
On considère les polynômes en forme normale dans le sens habituel.

$Mon \rightarrow Npol$
$+ \; Npol \times Mon \rightarrow Npol$ $+ \; Npol \times Cst \rightarrow Npol$

La sorte Horn des polynômes sous forme de Hörner

$+ \; Mon \times Cst \rightarrow Horn$ $+ \; Hmul \times Cst \rightarrow Horn$
$* \; Hmul \times Xfv \rightarrow Hmul$ $* \; Horn \times Xfv \rightarrow Hmul$
$Mon \rightarrow Horn$ $Hmul \rightarrow Horn$

Dans cet exemple le but est d'écrire un système de réécriture qui permettra de transformer un polynôme sous forme normale en polynôme de Hörner.

Exemple:
$ax^3 + bx^2 + cx + d$ se transformera en polynôme ayant la forme suivante:

$$d + x (c+x (b + ax))$$

Pour cela nous donnons trois algorithmes différents.

Algo1:
Règle1: +(*(x,y),*(z,y)) → *(+(x,z),y)
Règle2: +(*(x,y),*(z',y)) → *(+(x,z'),y)
Sur ces règles nous spécifions une sorte pour chaque variable:
τ(x) = Horn, τ(y) = Xfv, τ(z) = Mon, τ(z') = Cst.

Algo 2:
Règle1: +(*(x,y),*(z,y)) → *(+(x,z), y)
Règle2: +(*(x,y),*(z',y)) → *(+(x,z'),y)
Les sortes seront alors:
$\tau(x)$ = NPol, $\tau(y)$ = Xfv, $\tau(z)$ = Mon, $\tau(z')$ = Cst.

Algo 3:
Règle1: +(*(x,y),*(z,y)) → *(+(x,z), y)
Règle2: +(*(x,y),*(z',y)) → *(+(x,z'),y)
Les sortes seront alors:
$\tau(x)$ = Mon, $\tau(y)$ = Xfv, $\tau(z)$ = Mon, $\tau(z')$ = Cst.

Notre but est donc d'approximer tout problème par un problème clos qui peut être exprimé et résolu dans notre théorie de la réécriture close. Les sortes sont toujours spécifiées par des réécritures de termes clos.
Tout d'abord, on associe un algorithme clos GAlgo1 à notre algorithme 1.

GAlgo 1:
Règle1: +(*(x,y),*(z,y')) → *(+(x',z'),y'')
avec $\tau(x)=\tau(x')$=Horn
$\tau(y)=\tau(y')=\tau(y'')$=Xfv
$\tau(z)=\tau(z')$=Mon
Les sortes Horn, Xfv, et Mon vont être explicitées en termes d'automates.
On pourra également écrire cette règle de la façon suivante:
+(*(Horn,Xfv),*(Mon,Xfv))→*(+(Horn,Mon),Xfv)
Règle2:+(*(x,y),(z,y')) → *(+(x',z'),y'')
avec $\tau(x)=\tau(x')$=Horn
$\tau(y)=\tau(y')=\tau(y'')$=Xfv
$\tau(z)=\tau(z')$=Cst
Ici aussi , les sortes Horn, Xfv et Cst seront explicitées en termes d'automates.
De même on pourra l'écrire de cette façon:
+(*(Horn,Xfv),*(Cst,Xfv)) → *(+(Horn,Cst),Xfv)

Maintenant nous pouvons transformer GAlgo1 en un GTT à l'aide du formalisme des automates d'arbres.
Pour simplifier nous remplaçons les constantes A, . . ., Z par l'unique caractère C. Nous allons obtenir les spécifications suivantes:

Spécifications de gauche:
Automate gauche reconnaissant la sorte Mon
X → Xfv C → Cst *(Cst, Xfv) → Mon *(Mon,Xfv) → Mon

Automate gauche reconnaissant la sorte Horn

+(Mon,Cst) →Horn +(Hmul, Cst) → Horn *(Hmul,Xfv) → Hmul

*(Horn,Xfv) → Hmul *(Hmul,Xfv) → Horn *(Horn,Xfv) → Horn

*(Cst,Xfv) → Horn *(Mon,Xfv) → Horn

Automate reconnaissant la partie gauche de la Règle 1

I est l'état interface associé à la Règle 1, Q1 et Q2 sont les états intermédiaires

*(Horn,Xfv) → Q1 *(Mon,Xfv) → Q2 +(Q1,Q2) → I

Automate reconnaissant la partie gauche de la Règle 2

J est l'état interface associé à la Règle 2, R1 et R2 sont les états intermédiaires

*(Horn,Xfv) → R1 *(Cst,Xfv) →R2 +(R1,R2) → J

Spécifications de droite:

Automate droit reconnaissant la sorte Mon

Il s'agit du même automate qu'à gauche en renversant les flêches et en renommant les états

Xfv' → X Cst' → C Mon' → *(Cst', Xfv') Mon'→*(Mon',Xfv')

Automate droit reconnaissant la sorte Horn

Comme pour la sorte Mon il s'agit du même automate en reversant les flêches et en renommant les états

Horn'→+(Mon',Cst') Horn'→ +(Hmul', Cst') Hmul'→*(Hmul',Xfv')

Hmul'→ *(Horn',Xfv') Horn'→ *(Hmul',Xfv') Horn'→ *(Horn',Xfv')

Horn'→ *(Cst',Xfv') Horn'→ *(Mon',Xfv')

Automate reconnaissant la partie droite de la Règle 1

Q' est un état intermédiaire

I → *(Q',Xfv') Q' → +(Horn',Mon')

Automate reconnaissant la partie droite de la Règle 2

R' est un état intermédiaire

J → *(R',Xfv') R' → +(Horn',Cst')

Ce GTT ainsi obtenu, il nous reste à le compiler, c'est à dire à calculer les ε-transitions à l'aide de notre règle d'inférence puis à le réduire à l'aide des algorithmes classiques de réduction des automates d'arbres.

Génération des ε-transitions:

$$\frac{Cst' \rightarrow C \qquad C \rightarrow Cst}{Cst' \rightarrow Cst} \qquad \frac{Xfv' \rightarrow X \qquad X \rightarrow Xfv}{Xfv' \rightarrow Xfv}$$

$$\frac{Mon' \rightarrow *(Cst',Xfv') \quad Cst' \xrightarrow{*} Cst \quad Xfv' \xrightarrow{*} Xfv \quad *(Cst,Xfv) \rightarrow Mon}{Mon' \xrightarrow{*} Mon}$$

De la même manière nous obtenons

Mon' → Mon | R2| Q2| Horn | Hmul | Q1 | R1;
Hmul' → Hmul | Q1 | R1 | Horn | Mon | Q2;
Horn' → Horn | R2 | Q2| Hmul | Q1 | R1| Mon;
R'→ Horn; Q' → J | I ; I → Hmul | Q1 | R1 | Horn; J → Hmul | Q1 | R1 | Horn

Réduction:
Le réduction du GTT consiste à supprimer les ε-transitions et si on le désire à effectuer la réduction du non-déterminisme ce qui comme on l' a vu n'est pas obligatoire.
On procède de même pour Algo2 et Algo3.

Ces opérations effectuées, nous allons montrer quel type de question nous pourrons poser:
Soit Npol3 = +(+(+(*(*(*(C,X),X),X),*(*(C,X),X)),*(C,X)),C)
et Horn3 = +(*(+(*(+(*(C,X),C),X),C),X),C)
Npol3 est le schéma le plus général d'un polynôme de degré 3 en forme normale, Horn3 est le schéma le plus général d'un polynôme de degré 3 sous forme de Hörner.

Soit $Npol_{\geq 3}$=+(+(+(+(NPol,*(*(*(C,X),X),X)),*(*(C,X),X)),*(C,X)),C) et
$Horn_{\geq 3}$=(+(*(+(*(+(*(Horn,X),C),X),C),X),C).
$Npol_{\geq 3}$ est l'ensemble des polynômes en forme normal de degré ≥3. $Horn_{\geq 3}$ est l'ensemble des polynômes de Hörner de degré ≥3. Ce sont tous les deux des forêts reconnaissables.
Le tableau suivant énumère les réponses de valerian en fonction du problème posé.

algorithme	réponse de Valérian	conséquence pour le programme approximé
Problème du premier ordre		
Galgo1	Npol3 -*-> Horn3	Il se peut que Npol3 -*-> Horn3
Galgo2	Npol3 -*-> Horn3	Il se peut que Npol3 -*-> Horn3
Galgo3	non(Npol3 -*-> Horn3)	impossible
Problème du second ordre		
Galgo1	$Horn_{\geq 3} \cap$ Galgo1($Npol_{\geq 3}$)≠ ∅	Il se peut que $Horn_{\geq 3} \subset$ algo1($Npol_{\geq 3}$)
Galgo2	$Horn_{\geq 3} \cap$ Galgo2($Npol_{\geq 3}$)≠ ∅	Il se peut que $Horn_{\geq 3} \subset$ algo2($Npol_{\geq 3}$)
Galgo1	Norm(Galgo1($Npol_{\geq 3}$)) $\not\subset$ $Horn_{\geq 3}$	Il se peut que Norm(algo1($Npol_{\geq 3}$)) $\not\subset$ $Horn_{\geq 3}$

algorithme	réponse de Valérian	conséquence pour le programme approximé
Galgo2	Norm(Galgo2(Npol$_{\geq 3}$)) $\notin$ Horn$_{\geq 3}$	Il se peut que Norm(Galgo2(Npol$_{\geq 3}$)) $\notin$ Horn$_{\geq 3}$
Problèmes du premier ordre traités avec une stratégie "innermost" (pas implanté)		
Galgo1	Npol3 -*-> Horn3	il se peut que Npol3 -*-> Horn3
Galgo2	non(Npol3 -*-> Horn3)	impossible

CHAPITRE 3

Le système VALERIAN

1. Présentation du logiciel

VALERIAN est un logiciel qui permet de décider du problème de l'accessibilité du premier ordre et du second ordre, ceci dans le cadre des systèmes de réécriture clos ainsi que des systèmes de réécriture séparés. Ce logiciel est implanté en PROLOG 2 et fonctionne sur SUN 3/50. Nous allons dans un premier temps examiner comment fonctionne VALERIAN et comment se déroule une session d'utilisation de celui-ci.
Pour lancer le logiciel il faut entrer la commande:

prolog valeriann.psv

Le système PROLOG 2 ainsi que le logiciel se chargent en mémoire interne, puis apparaît à l'écran le menu principal suivant:

Ecran 1

```
                VALERIAN

    1: creation of a ground rewriting system
    2: compilation
    3: execution
    4: save
    5: load
    6: remove a rule
    7: end


your choice?:
```

- La première option correspond à la création des règles d'un système de réécriture.
- La seconde correspond à la compilation du système de réécriture.
- La troisième permet de résoudre le problème de l'accessibilité.
- La quatrième permet de sauvegarder un système de réécriture compilé.
- La cinquième permet de charger un système de réécriture compilé.
- La sixième permet de supprimer une règle du système de réécriture courant.
- La dernière permet de quitter VALERIAN.

1.1- Création d'un système de réécriture.

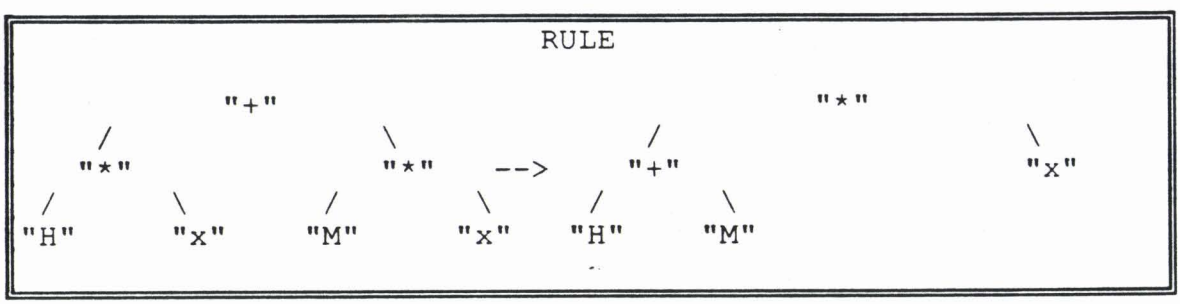
Lors de la sélection de cette option, le système demande si l'utilisateur désire créer une nouvelle règle.
Si c'est le cas, les règles sont saisies sous forme de termes arborescents. A chaque nouveau noeud le système demande l'arité de celui-ci. Lorsque l'arité est nulle, deux possibilités peuvent se présenter:

-Soit il s'agit d'une constante (système de réécriture clos)
-Soit il s'agit d'un nom de sorte (système de réécriture séparé)

On peut voir sur les copies d'écran qui suivent comment cela se passe réellement.

Ecran 2

```
                DIALOGUE
do you want to create a new rule? y
arity?0
is it a leaf (1) or a variable
(2)?2
```



Ecran 3

```
DIALOGUE
do you want to create a new rule? y
arity?0
is it a leaf (1) or a variable
(2)?1
```

```
RULE

        "+"                                   "*"
       /   \                                 /   \
     "*"    "*"          -->              "+"    "x"
    /   \  /   \                         /   \
  "H"  "x" "c"  "x"                    "H"  "c"
```

Une fois la saisie des règles terminée, le système examine si des noms de sorte ont été saisis. Si c'est le cas, le système demande à l'utilisateur quelles sont les règles de l'automate reconnaissant la sorte en question.
Les règles de l'automate sont saisies sous forme naturelle.

Ecran 4

```
DIALOGUE
Automaton which recognizes the M
sort:
do you want to create a new rule? y
```

```
RULE

        "*"
       /   \
     "q0"   "q1"        -->        "q2"
```

Puis VALERIAN demande quels sont les états finaux de notre automate.

Ecran 5

```
DIALOGUE
Please, give the final states of
this automaton.
final state 1: "q2"
```

Les opérations de saisie sont alors terminées. Si il existait déjà un système en mémoire, les nouvelles informations sont ajoutées aux anciennes.

1.2-Compilation d'un système de réécriture.

Pour que l'on puisse décider du problème de l'accessibilité, il faut réaliser la compilation du système courant.
Il se produit alors une première étape de compilation sans réduction du non-déterminisme. Cette opération effectuée, VALERIAN demande à l'utilisateur si il désire déterminiser le système. Ceci étant particulièrement long, il n'est pas obligatoirement nécessaire de l'effectuer. Cependant, si on l'effectue, le temps de réponse au problème de l'accessibilité devient instantané (linéaire). Sinon, ce temps de réponse sera quadratique.

1.3-Décision du problème de l'accessibilité.

Lorsque l'on choisit cette opération, l'écran suivant apparaît:

Ecran 6

```
VALERIAN tries to give an answer to the
following question:
Can a given tree T (or a forest F) be reduced
to another given tree T'(or another forest
F')?

Do you want to give two trees (1) or two
forests (2)?
```

On peut alors résoudre:

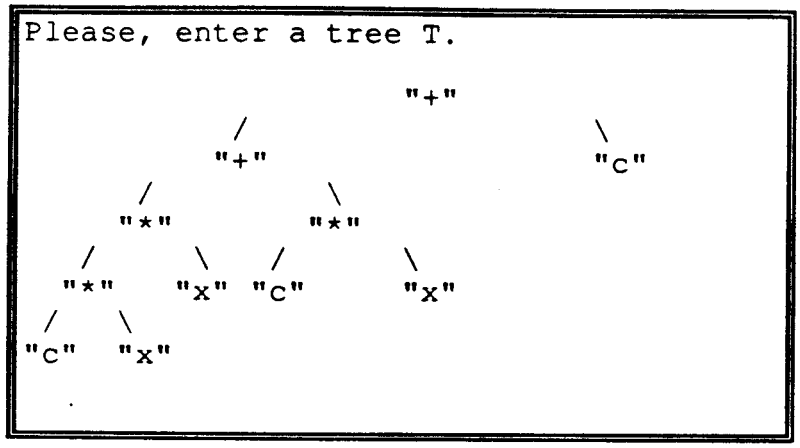
-Le problème de l'accessibilité du premier ordre: saisie de deux arbres.

-Le problème de l'accessibilité du second ordre: saisie de deux forêts.

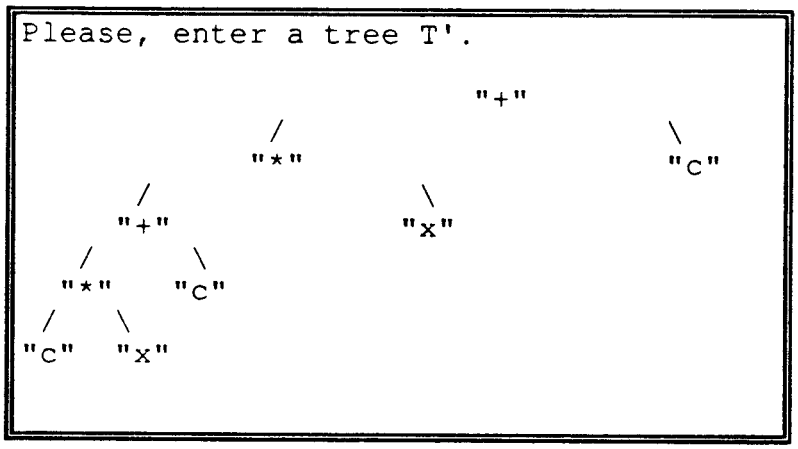
1.3.1-Saisie de deux arbres.

La saisie des deux arbres se fait sous la forme d'arborescences comme lors de la saisie des règles.
Voici un exemple de saisie de deux arbres.

Ecran 7



Ecran 8



Une fois la saisie réalisée, le système demande si l'utilisateur désire lancer l'exécution. Si oui, dans le cas où le système n'est pas déterministe la réponse vient au bout d'un temps quadratique et l'écran suivant apparaît:

Ecran 9

```
computation time: 90240
T can be reduced to T' with respect to
our rewriting system.
Strike s<return> to continue.
```

On obtient ainsi le temps d'exécution (en milliseconde) et la réponse au problème de décision.
Si le système est déterministe, on peut obtenir en outre la suite des réécritures à effectuer pour pouvoir transformer T en T'. Dans ce cas l'écran suivant apparaît:

Ecran 10

```
Computation time: 640
Strike s<return> to continue
s


Do you want any explanation (y/n) y
T -- 8 --> -- 4 --> -- 1 --> -- 5 --> --
12 --> -- 10 --> -- 9//9 --> T'
Strike s<return> to continue
```

- Les numéros correspondent aux règles à appliquer.
- Le symbole // signifie que les règles peuvent être appliquées en parallèle.
- Le symbole -- --> signifie que les règles doivent s'appliquer en séquence.

1.3.2-Saisie de deux forêts.

La saisie des deux forêts s'effectue en donnant les règles des automates les reconnaissant. La saisie de ces automates s'effectue de la même manière que la saisie des automates reconnaissant les sortes d'un système de réécriture séparé.
Dans ce cas le système dit si oui ou non $R(F) \cap F' \neq \emptyset$. C'est à dire, est-il possible que F se transforme en F' à l'aide du système de réécriture?

1.4- Suppression d'une règle de réécriture.

Dans ce cas le système affiche les règles du système de réécriture et demande à l'utilisateur le numéro de la règle à supprimer.

Ecran 11

```
rule 1:
from
<c,<b<b1,nil>.nil>.<b1.nil>.nil>
to
<a1,nil>

rule 2:
from <b1,nil> to <b,<b1,nil>.nil>

rule 3:
from <a1,nil>  to  <a,<a1,nil>.nil>
Which rule do you want to remove? 2
```

Une fois le numéro de la règle saisi, VALERIAN recompile le système de réécriture ainsi modifié afin de pouvoir à nouveau décider du problème de l'accessibilité.

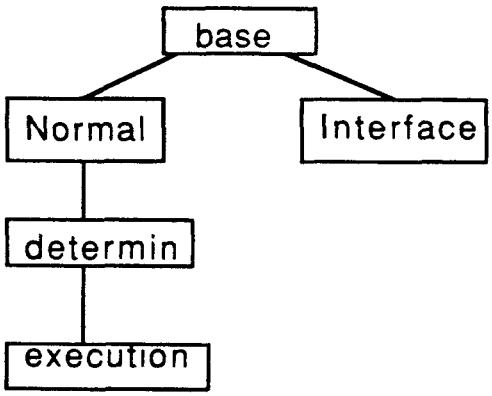
2- Implantation PROLOG.

2.1-Structure générale de VALERIAN.

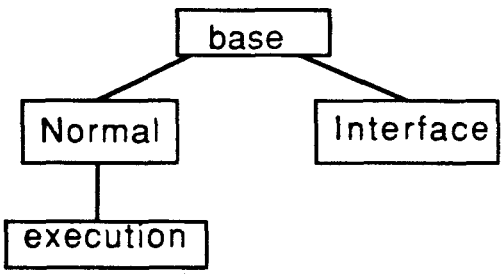
VALERIAN étant implanté en PROLOG 2, ce logiciel est donc structuré par des mondes.
Il utilise trois mondes:
- Le monde "Normal" contenant les différents programmes qui composent VALERIAN.
- Le monde "determin" dans lequel sera mémorisé le système de réécriture compilé. (N'existe que si le système est déterminisé, sinon le système compilé se trouve dans le monde "Normal")
- Le monde "execution", à l'intérieur duquel seront exécutées les différentes opérations nécessaires à la décision du problème de l'accessibilité.

Donc, VALERIAN pourra avoir deux configurations différentes:

Cas où le système est déterminisé



Cas où le système n'est pas déterminisé



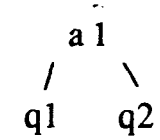
2.2- Formalisme utilisé pour représenter les règles.

Un noeud est représenté par un couple d'éléments:
-Un nom de noeud (chaîne de caractères)
-Une liste de fils qui seront soit des états (automates) soit eux-même des noeuds (liste PROLOG).
Les états des automates sont représentés par le prédicat état(y), y étant un nom d'état.

Exemple:

< "a1", états(1).état(2).nil>

représente le terme



Ici les fils du noeud a1 sont des états.

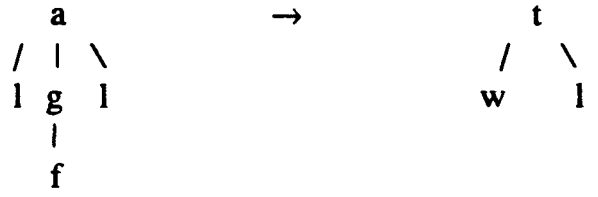
Donc, les règles du système non compilé sont représentées de la façon suivante:

ss(<x,L>,<y,L'>)->;

x et y étant des noms de noeud et L et L' sont des listes de fils, les fils étant eux-même des noeuds.

Exemple:

La règle de réécriture



sera représentée dans VALERIAN par:

ss(<"a",<"l",nil>.<"g",<"f",nil>.nil>.<"l",nil>.nil>, <"t",<"w",nil>.<"l".nil>.nil>)->;

Les règles des automates ascendant et descendant du GTT associé au système de réécriture S sont représentées de la façon suivante:

Règles de l'automate ascendant:

gg(<x,L>,etat(y))->;

x: nom du noeud
L:liste de fils (états)
y:nom d'état.

Règles de l'automate descendant:

dd(etat(y), <x,L>)->;

Ayant vu la structure des objets manipulés par VALERIAN, nous pouvons examiner en détail les principaux programmes de notre logiciel, afin d'étudier les problèmes rencontrés et les solutions qui ont été apportées.

3- Implantation des algorithmes de compilation.

3.1- Transformation des règles du système de réécriture en un GTT.

Cette opération s'effectue afin de pouvoir modifier facilement le système de réécriture ultérieurement. (ajout ou suppression de règles)

Pour cela, voici comment on procède.

Pour supprimer une règle il est nécessaire de savoir à quelles règles du système correspondent les règles des automates ascendant et descendant du GTT associé. Ainsi lorsque l'on décidera de supprimer une règle de S, il suffira de supprimer les règles du GTT correspondantes, puis d'enchainer avec les autres opérations de compilation (calcul des ε-transitions et réduction)

A chaque règle nous associerons donc un GTT, c'est à dire qu'à une règle ri → li on associe le GTT GGi dont les automates Ai et Di seront tels que Ai reconnaitra le terme ri et Di générera le terme li.
C'est le prédicat *formattête(n,<x,y>,q)* qui est utilisé pour transformer la partie gauche (ri_n) de la règle n en un automate ascendant. Les arguments ont la signification suivante:

n:numéro de la règle

<x,y>: terme à transformer en automate.

q: état dans lequel on est après avoir reconnu le terme <x,y>.

Le deuxième argument peut également être une liste de termes et le troisième être une liste d'états associés à chacun des termes de la liste du second argument.

```
formattete(n,nil,nil) ->
    /;
formattete(n,<x,y>,q) ->
    formattete(n,y,q1)
    assert(gg2(n,<x,q1>,etat(q)),nil)
    /;
formattete(n,y.q,etat(y1).q1) ->
    incrementer(compteur)
    val(compteur,y1)
    formattete(n,y,y1)
    formattete(n,q,q1)
    /;
```

Prédicat utilisé pour transformer la partie droite (li_n) de la règle n en un automate descendant.
Les arguments ont la même signification que pour le prédicat précédent.

```
formatqueue(n,nil,nil) ->
    /;
formatqueue(n,q,<x,y>) ->
    formatqueue(n,q1,y)
    assert(dd2(n,etat(q),<x,q1>),nil)
    /;
formatqueue(n,etat(y1).q1,y.q) ->
    incrementer(compteur)
    val(compteur,y1)
    formatqueue(n,y1,y)
    formatqueue(n,q1,q)
    /;
```

Le nom des états intermédiaires est généré automatiquement à l'aide d'un compteur. Chaque état reconnaissant une partie de règle complète est un état interface du GTT.
On utilise donc ces deux prédicats pour chacune des règles du système de réécriture.
Au cours de cette opération, il est nécessaire de garder certains paramètres en mémoire. Ceux-ci nous servirons lors de l'ajout ou de la suppression de règles, ainsi que pour les opérations de compilation ultérieures.

* Le nombre de règles: gardé dans le prédicat nbregl(n)
* Le nombre d'états: nbetat(e)
* Le dernier nom d'état généré: dernom(m)
* La liste des états interface: inter(l)

L'appel des prédicats *formattete* et *formatqueue* pour chacune des règles du système ainsi que la sauvegarde des paramètres précédents sont réalisés par le prédicat *sdrgtt*.

```
sdrgtt ->
    assign(compteur,0)
    assign(nr,0)
    sdrgtt'(l,0)
    val(compteur,n)
    assert(nbet(n),nil)
    assert(dernom(n),nil)
    assert(inter(l),nil)
    ;
```

Le prédicat *sdrgtt'* récupère dans ses arguments la liste des états interface ainsi que le nombre d'états engendrés.

```
sdrgtt'(etat(q).l,n) ->
    rule(ss(x,y),nil)
    suppress(l)
    incrementer(compteur)
    val(compteur,q)
    val(add(n,1),n1)
    formattete(n1,x,q)
    formatqueue(n1,q,y)
    sdrgtt'(l,n1)
    incrementer(nr)
    val(nr,n2)
    assert(ss'(n1,x,y,q),nil)
    /;
sdrgtt'(nil,n) ->ajoutnbregl(n);
```

```
ajoutnbregl(n)->
 rule(nbregl(m),nil)
 suppress(1)
 assert(nbregl(n),nil)/;
ajoutnbregl(n) ->
 assert(nbregl(n),nil);
```

Le GTT ainsi calculé est gardé en mémoire auxiliaire de façon à pouvoir le réutiliser lors de la modification du système de réécriture. La version gardée en mémoire interne va subir diverses modifications. Nous allons voir comment dans les parties qui suivent.

3.2- Cas d'un système de réécriture séparé: Ajout des règles des automates reconnaissant les sortes.

Dans le cas d'un système de réécriture séparé, notre GTT comprendra des règles qui seront:

-Soit de la forme gg(<x,nil>,y)->;
-Soit de la forme dd(y,<x,nil>)->;

où x correspond à un nom de sorte.

Dans ce cas, il faut greffer sur ces règles, les règles de l'automate reconnaissant la sorte x, celles-ci sont représentées par le prédicat *tt(x,l,q1)* où *x* est le nom de la sorte, *l* la partie gauche de la règle et *q1* la partie droite. Les états finaux de cet automate sont

représentés par les faits ff(x,q) où x est le nom de la sorte et q est le nom de l'état final.
Pour cela, on prend toutes les règles de cet automate dont la partie droite est un état final. Puis, dans le cas de l'automate gg, on substituera la partie gauche <x,nil> par *l* (partie gauche de la règle choisie de l'automate reconnaissant la sorte *x*.). Ainsi on peut créer une nouvelle règle de la forme gg(l,y)->;, en ayant pris soin de supprimer l'ancienne. Ceci est effectué par le prédicat *jonctiong*.

```
jonctiong ->var(x)
            gg(<x,nil>,etat(q))
            jonction'(x,q) echec;
jonctiong ->;
jonction'(x,q) ->ff(x,q1)
                 tt(x,l,q1)
                 ajoutgg(x,l,q);
```

Ajout de la nouvelle règle gg:

```
ajoutgg(x,l,q)->rule(gg(<x,nil>,etat(q)),nil)
                suppress(1)
                assert(gg(l,etat(q)),nil)/;
ajoutgg(x,l,q)->assert(gg(l,etat(q)),nil);
```

Enfin, on termine en ajoutant toutes les autres règles de l'automate reconnaissant la sorte *x*.

```
copiertt ->
    tt(x,l,q)            ajoutgg'(l,q)->gg(l,q)/;
    ajoutgg'(l,q)        ajoutgg'(l,q)->assert(gg(l,q),nil);
    echec;
copiertt ->;
```

Dans le cas de dd, il est nécessaire de renommer les états des automates reconnaissant les sortes de façon à ce que l'intersection entre l'ensemble des états non-interface de gg et l'ensemble des états non-interface de dd reste vide.
Ce renommage est réalisé à l'aide des prédicats *prime(l,ll)* où *l* est une liste d'état et *ll* est la même liste avec renommage, et *copiertt'*.

```
prime(etat(x).l,etat(l.x).ll) ->prime(l,ll);
prime(nil,nil)->;

copiertt'->tt(x,<y,l>,etat(q))
           prime(l,ll)
           ajoutdd'(<y,ll>,etat(l.q)) echec;
copiertt'->;

ajoutdd'(l,q)->dd(q,l)/;
ajoutdd'(l,q)->assert(dd(q,l),nil);
```

Mis à part ce renommage, les opérations sont quasi-identiques à celles effectuées sur gg. Plus exactement, on substituera la partie droite <x,nil> où x est un nom de sorte, par le couple <y,l> trouvé dans la règle tt(x,<y,l>,q)->; où q est un état final, ceci après avoir renommé les états de la liste l.
Ceci est effectué par *jonctiond*

```
jonctiond-> var(x)                          jonctiond'(x,q)->
            dd(etat(q),<x,nil>)                  ff(x,q1)
            jonctiond'(x,q) echec;               tt(x,<y,l>,q1)
jonctiond->;                                     ajoutdd(x,y,l,q);
```

Ajout de la nouvelle règle dd:

```
ajoutdd(x,y,l,q)->rule(dd(etat(q),<x,nil>),nil)
                  suppress(1)
                  prime(l,l1)
                  assert(dd(etat(q),<y,l1>),nil)/;
ajoutdd(x,y,l,q)->prime(l,l1)
                  assert(dd(etat(q),<y,l1>),nil)/;
```

Ensuite comme pour gg on ajoute toutes les autres règles de l'automate reconnaissant la sorte x, après avoir renommé tous les états présents dans ces règles et inversé les parties gauches et les parties droites.

3.3- Calcul des ε-transitions et réduction.

L'algorithme calculant l' itération du GTT (calcul des ε-transitions) a déjà été étudié dans le Chapitre 2. Voici comment il est implanté en PROLOG2:
Le prédicat *gttétoile* appelle deux prédicats. Le premier manipule les règles reconnaissant les feuilles, le second manipule les règles reconnaissant les noeuds.
Le premier prédicat appelé *début(l)* récupère dans son argument les états rencontrés en partie gauche des règles de l'automate dd reconnaissant les feuilles.

```
gttetoile -> debut(l)
             suitefin'(l,l1);

debut(y.l) -> rule(dd(y,<x,nil>),nil)
              suppress(1)
              chercherg1(x,y)
              debut(l)
              assert(dd(y,<x,nil>),nil)
              /;
debut(nil) ->;
```

Ce prédicat appelle *cherchergl(x,y)* qui va rechercher les règles de l'automate gg correspondant aux règles de l'automate dd trouvées dans *début(l)*. Ensuite, il ajoute une ε-transition à l'aide du prédicat *nvtrans(y,yl)*, *y* et *yl* étant des noms d'états.
Ici le premier argument de *cherchergl* correspond au nom du noeud rencontré dans la règle de dd choisie et le second correspond au nom de l'état présent en partie gauche de cette même règle.

```
chercherg1(x,y) ->
        gg(<x,nil>,y1)
        nvtrans(y,y1)
        echec;
chercherg1(x,y) ->;
nvtrans(y,y1)->gg(y,y1)/;
nvtrans(y,y1)->assert(gg(y,y1),nil);
```

Le second prédicat appelé *suitefin'(l,ll)* est tel que *l* correspond à la liste des anciens états rencontrés et *ll* correspond à la liste des nouveaux états rencontrés au cours de la recherche des ε-transitions.
Dans ce prédicat, *compteurl* est utilisé pour vérifier si on a ajouté au moins une nouvelle ε-transition ou non. Si on en a ajouté une, on recommence la recherche des ε-transitions récursivement, sinon on arrête.

```
suitefin'(l,ll) ->
      assign(compteur1,0)
      suitefin(l,ll)
      val(compteur1,x)
      dif(x,0)
      suitefin'(l,l2)
      /;
suitefin'(l,ll) ->;
```

Le prédicat *suitefin'(l,ll)* appelle le prédicat *suitefin(l,ll)*. Ce dernier permet de monter dans les arborescences présentent dans les parties droites des règles du système de réécriture. Avant d'exécuter un appel récursif de *suitefin*, on examine si la concaténation des deux listes *l* et *ll* contient tous les états interface. Si c'est le cas on est arrivé à la racine de toutes les arborescences et on a terminé, sinon il faut passer à un niveau supérieur.

```
suitefin(l,ll) ->
      suite(l,ll)
      concat(l,ll,l2)
      verif(l2)
      suitefin(l2,l3)
      /;
suitefin(l,l) ->;

verif(l2)->
  inter(i)
  elt(i,l2)
  /
  echec;
verif(l2)->;
```

Ce prédicat appelle le prédicat *suite(l,l1)*, dont les arguments ont même signification que pour les deux prédicats précédents. Ce prédicat prend, une à une, les règles reconnaissant les noeuds de l'automate dd et vérifie si les états présents dans la liste de fils ont déjà été étudiés, c'est à dire si ils sont présents dans la liste *l*. Si c'est le cas, on va chercher à l'aide du prédicat *chercherg2(x,y,y1)* la règle de l'automate gg correspondante qui sera de la forme gg(<x,y3>,y2)->;. Sinon on passe à la règle suivante de l'automate dd.

```
suite(l,y1.l1)->
        dd(y1,<x,y>)
        dif(y,nil)
        exam'(y,l)
        /
        rule(dd(y1,<x,y>),nil)
        suppress(1)
        chercherg2(x,y,y1)
        suite(l,l1)
        assert(dd(y1,<x,y>),nil);
suite(l,nil) ->;

exam'(nil,l)->;
exam'(x.y,l)->
    elementde(x,l)
    exam'(y,l);

elementde(x,x.y)->;
elementde(x,u.y)->
    elementde(x,y);
```

```
chercherg2(x,y,y1) ->
        gg(<x,y3>,y2)
        comp(y,y3)
        ajouttrans(y1,y2)
        echec;
chercherg2(x,y,y1) ->;

ajouttrans(y1,y2) ->
    gg(y1,y2)
    /;
ajouttrans(y1,y2) ->
incrementer(compteur1)
assert(gg(y1,y2),nil);
```

Les arguments de *chercherg2(x,y,y1)* sont tels que *x* représente le nom du noeud courant, *y* représente la liste de fils du noeud *x* et *y1* est l'état rencontré en partie gauche de la règle choisie de l'automate dd. Il appelle le prédicat *comp(y,y3)* qui va regarder si on peut passer des états de *y* aux états de *y3* (*y* et *y3* étant des listes d'états) par les ε-transitions déjà existantes. Pour chaque couple d'états, on appelle le prédicat *cherchgg* qui va regarder si l'ε-transition existe.

```
comp(nil,nil) ->
comp(x.y,x1.y1) ->
    cherchgg(x,x1)
    comp(y,y1);
```

Le prédicat *cherchgg* est composé de deux clauses. La première concerne le cas où l'ε-transition existe telle quelle. Dans le cas contraire, la seconde va recherche l'ε-transition en procédant par transitivité.

```
cherchgg(e1,e2)-> gg(e1,e2)
                  /supold;
cherchgg(e1,e2)->
      assert(old'(e1),nil)
      gg(e1,x)
      not(old'(x))
      cherchgg(x,e2);

supold->
  rule(old'(x),nil)
   suppress(1)
   supold/;
supold->;
```

Remarque:
On pourrait procéder autrement, en calculant, chaque fois que l'on ajoute une nouvelle ε-transition, la clôture transitive du graphe formé par les ε-transitions existantes. Ainsi *cherchgg* ne serait composé que de la première clause.
L'étude de la compléxité de cette méthode à été vue dans le Chapitre 2. Cependant, la recherche d'une ε-transition par transitivité, en procédant en profondeur d'abord, possède une complexité linéaire en O(max(n,e)) où n est le nombre de sommets et e le nombre de flêches.
En effet, le temps passé dans cherchgg(e1,e2) est proportionnel au nombre de sommets adjacents à e1. De plus, cherchgg(e1,e2) est appelé une seule fois pour un noeud donné e1, puisque e1 est marqué par old' la première fois que cherchgg(e1,e2) est appelé. D'où le résultat.
Ainsi, la complexité de cette méthode est inférieure à la complexité de l'algorithme calculant la clôture transitive ($O(n^4)$). C'est pourquoi nous avons gardé la recherche d'une ε-transition par transitivité.

Si toutes les ε-transitions recherchées existent, on peut créer une nouvelle ε-transition de la forme gg(y1,y2)->;.

3.4-Suppression des boucles à l'intérieur du graphe formé par les ε-transitions.

On a pu remarquer que dans certains cas, il y avait présence de boucles dans le graphe formé par les ε-transitions. Dans ce cas tous les états présents dans ces boucles sont équivalents, et il n'est donc pas nécessaire de les garder tous, mais simplement d'en garder un seul équivalent à tous les autres. Ainsi nos automates contiendront moins d'états, ce qui diminuera le temps d'exécution de nos algorithmes de décision du problème de l'accessibilité.

Nous avons, pour cela, implanté l'algorithme détaillé dans le livre de Aho-Hopcroft et Ullman [1] qui possède une compléxité linéaire en O(max(n,e)) où n est le nombre de sommets et e le nombre de flêches. Cet algorithme utilise une recherche en profondeur d'abord afin de trouver les composants fortement connexes du graphe formé par les ε-transitions.
Chacune des boucles détectées sont sauvegardées dans le prédicat *alias(x,l)* où l représente la liste des états équivalents à l'état x.
Grâce à ces faits on peut substituer dans les ε-transitions tous les états présents dans les listes l par l'état x correspondant.(Ceci à l'aide du prédicat *remplace(x)*).

Implantation PROLOG2 de cet algorithme:

```
boucle -> assign(count,1)
          gg(etat(v),etat(w))
          not(old(etat(v)))
          searchc(etat(v))
          echec;
boucle -> remplace;
"-----------recherche des boucles---------------"
searchc(etat(v)) -> assert(old(etat(v)),nil)
                    val(count,c)
                    assert(dfn(etat(v),c),nil)
                    incrementer(count)
                    ajoutlink(etat(v),c)
                    assert(pile(etat(v)),nil)
                    search'(etat(v))
                    teste(etat(v))/;

search'(v) -> gg(v,w)
              search''(v,w) echec;
search'(v) ->;

search''(v,w) -> not(old(w))
                 searchc(w)
                 lowlink(v,c1) lowlink(w,c2)
                 min(c1,c2,c3)
                 ajoutlink(v,c3)/;
search''(v,w) -> dfn(v,c1) dfn(w,c2) val(inf(c2,c1),1)
                 pile(w)
                 lowlink(v,c3)
                 min(c2,c3,c4)
                 ajoutlink(v,c4)/;
"-------verifie si on trouve une boucle---------"
teste(etat(v)) ->lowlink(etat(v),b) dfn(etat(v),b)
                 teste'(etat(v),l)
                 assert(alias(v,l),nil)/;
teste(etat(v))) ->;
"-------stocke les elements de la boucle dans une liste----"
teste'(v,x.l) ->cherchpile(x)
                dif(etat(x),v)
                teste'(v,l);
teste'(v,nil) ->;

cherchpile(x) ->rule(pile(etat(x)),nil)
                suppress(1)/;
"------mise a jour du predicat lowlink pour le noeud v-----"
ajoutlink(v,c) ->rule(lowlink(v,x),nil)
                 suppress(1)
                 assert(lowlink(v,c),nil)/;
ajoutlink(v,c) ->assert(lowlink(v,c),nil);

"--renommer tous les etats equivalents--"

remplace ->alias(v,l)
           remplace'(v,l) echec;
remplace ->;

remplace'(v,x.l) ->remplace1'(x,v) remplace'(v,l);
remplace'(v,nil) ->;
```

```
remplace1'(y,x) ->remplacer1(y,x) echec;
remplace1'(y,x) ->;

remplacer1(y,x) ->
            rule(gg(etat(y),etat(z)),nil)
            suppress(1)
            nvgg(etat(x),etat(z));
remplacer1(y,x) ->
            rule(gg(etat(z),etat(y)),nil)
            suppress(1)
            nvgg(etat(z),etat(x));
remplacer1(y,x) ->
            rule(gg(z,z),nil)
            suppress(1);

nvgg(x,y) -> gg(x,y) /;
nvgg(x,y) -> assert''(gg(x,y),nil);
```

Complexité:

Soit m le nombre d'ε-transitions (en terme de graphes, cela correspond aux flêches) borné par n^2 (n étant le nombre d'états, ou en terme de graphes, le nombre de sommets). On a vu que la détection des boucles s'effectue en temps linéaire, l'algorithme possédant une complexité en O(max(n,m)).

Ensuite, il reste à effectuer le renommage des états équivalents. Ceci se fait en trois étapes.

- Substitution à gauche.
- Substitution à droite.
- Suppression des ε-transitions dont la partie gauche est égale à la partie droite.

A chaque fois on parcourt les m ε-transitions. Donc la complexité du renommage est égale à 3m.

Finalement, on aura une complexité de temps en:

$$O(max(n,m)+ 3m)$$

La suppression des boucles étant terminée, on peut alors réaliser la substitution des états équivalents à l'intérieur des règles des automates gg et dd. Ceci est réalisé à l'aide des prédicats *substituerg* et *substituerd*.

```
substituerg ->
  rule(gg(<x,l>,etat(y)),nil)
  dif(x,etat)
  suppress(1)
  substituerliste(l,ll)
  cherchealias(y,yl)
  substituerg
  nvgg(<x,ll>,etat(yl))
  /;
substituerg ->;
```

```
substituerd ->
  rule(dd(etat(y),<x,l>),nil)
  suppress(1)
  substituerliste(l,ll)
  cherchealias(y,yl)
  substituerd
  nvdd(etat(yl),<x,ll>)
  /;
substituerd ->;
```

```
nvdd(x,y) -> dd(x,y) /;
nvdd(x,y) -> assert(dd(x,y),nil);

substituerliste(etat(x).l,etat(y).l2) -> cherchealias(x,y)
    substituerliste(l,l2) /;
substituerliste(nil,nil) ->;

"--recherche du nom equivalent--"

cherchealias(x,y) -> alias(y,l) elementde(x,l) /;
cherchealias(x,x) ->;
```

3.5- Suppression des ε-transitions.

De façon à pouvoir déterminiser nos nouveaux automates, il est nécessaire de supprimer les ε-transitions. Pour cela, on ajoute les règles nécessaires dans les deux automates gg et dd.
Ceci est réalisé dans gg à l'aide du prédicat *etransg*.
Il est inutile de faire les substitutions d'états en partie gauche des règles de l'automate gg, car les ε-transitions sont de la forme i → e où i est soit un état appartenant à l'ensemble des états de l'automate dd, soit un état interface. Or les états interface n'apparaissent qu'en partie droite des règles de l'automate gg. Nous ne ferons donc les substitutions qu'en partie droite des règles de gg. Donc *etransg* examine si il existe une règle de la forme gg(<x1,l>,etat(x))->; et une ε-transition de la forme gg(etat(x),etat(y))->;. Si c'est le cas, alors la règle gg(<x1,l>, etat(y))->; peut être ajoutée.

```
etransg ->
    gg(<x1,l>,etat(x))
    dif(x1,etat)
    gg(etat(x),etat(y))
    cregleg(x1,l,etat(y))
    echec;
etransg ->;

cregleg(y,l,x) ->
    gg(<y,l>,x)
    /;
cregleg(y,l,x) ->
 assert''(gg(<y,l>,x),nil);
```

Si m correspond au nombre d'ε-transitions et p au nombre de règles de l'automate gg, on aura une complexité en O(mp).

Dans dd, la substitution des états en partie droite est un peu plus compliquée. On utilise pour cela le prédicat *etd*. Le compteur est utilisé pour voir si on a ajouté une nouvelle règle. Si c'est le cas, on recommence, sinon on arrête.

```
etd ->
    assign(compteur,0)
    etransd'
    val(compteur,n)
    dif(n,0)
    etd
    /;
etd ->;
```

Ce prédicat appelle le but *etransd'*. Ce dernier prend une à une toutes les règles de dd de la forme dd(x,<y,l>)->; avec l non vide. Pour chaque élément i appartenant à l, on regarde si il existe une ε-transition de la forme i → z, avec z appartenant à l'ensemble des états de dd. Si c'est le cas on substitue i par z. Etant donné qu'il existe plusieurs façons de réaliser les substitutions, il est nécessaire de générer toutes les combinaisons d'états possibles. Celles-ci sont générées par le prédicat *combinl(l,l2)*, où l est la liste d'origine et l2 est la nouvelle liste avec substitutions. Pour chacune de ces nouvelles listes l2 on crée alors une nouvelle règle de la forme dd(x,<y,l2>)->; à l'aide du prédicat *crègle(x,y,l2)*.

```
etransd' ->
    dd(x,<y,l>)
    dif(l,nil)
    combinl(l,l2)
    cregle(x,y,l2)
    echec;
etransd' ->;

combinl(nil,nil) ->/;
combinl(x.l,y.l2) ->
    gg(x,y)
    combinl(l,l2);
combinl(x.l,x.l2) ->
    pasgg(x)
    combinl(l,l2);
```

```
cregle(x,y,l) ->
    dd(x,<y,l>)
    /;
cregle(x,y,l) ->
    assert(dd(x,<y,l>,nil)
    incrementer(compteur);

pasgg(x) -> gg(x,y) / echec;
pasgg(x) ->;
```

Il est évident que la complexité de cet algorithme est combinatoire.

La suppression des ε-transitions effectuée, on peut alors, si on le désire déterminiser nos deux automates gg et dd. (La réduction du non déterminisme est étudiée en annexe 1 p136)

4- Implantation des algorithmes de décision du problème de l'accessibilité.

4.1- Décision avec S'.

Le prédicat principal, décidant du problème de l'accessibilité, est appelé *créationm'(t)* et a la forme suivante:
t peut prendre soit la valeur 1 soit la valeur 2, ceci suivant que l'on étudie deux forêts ou deux arbres, respectivement.

```
creationm'(t) ->
    intersection'
    rechercheetat'(l1)
    subst(l1)
    testerinter(t)
    /;
creationm'(p) ->
    page
    cpu-time(t)
    outm("computation time:")
    out(t)
    line
    repneg(p)
    line
    in-char'(r)
    climb("Normal");
```

```
testerinter(p) ->
    intersection''
    trouveretat''(p)
    page
    cpu-time(t)
    outm("computation time:")
    out(t)
    line
    reppos(p)
line
    in-char'(h)
    climb("Normal")
    /;
testerinter(p) ->
    page
    cpu-time(t)
    outm("computation time:")
    out(t)
    line
    repneg(p)
    line
    in-char'(h)
    climb("Normal");
```

Comme on l'a vu la décision s'effectue en trois étapes:

-Calcul du produit de l'automate gg et de l'automate appelé aa0 reconnaissant la donnée de départ, ceci en gardant toutes les règles de l'automate aa0. (C'est le prédicat *intersection'* qui réalise ce calcul, on étudiera celui-ci en annexe)
-Ajout des règles de l'automate dd.
Cette opération est réalisée à l'aide des prédicats suivants:
On recherche tout d'abord à l'intérieur de l'automate produit, les états couples de la forme etat(<q,i>) où i est un état interface. Ceci avec le prédicat *rechercheetat'(l1)*, *l1* étant la liste des états de l'automate produit de ce type.

```
rechercheetat'(l1) ->
    etatinterf(l1)
    dif(l1,nil);

etatinterf(etat(x).l1) ->
    rule(couple(x),nil)
    suppress(1)
    exametat(etat(x))
    etatinterf(l1)
    /;
etatinterf(nil) ->;
```

```
exametat(etat(<x,y>)) ->
    inter(i)
    elementde(etat(y),i);
```

Ensuite, en greffera les règles de l'automate dd à l'automate produit. On utilise pour cela le prédicat *subsd'(l)* où *l* est la liste formée par le prédicat *rechercheetat'(l)*. Pour chaque état couple de la forme etat(<x,y>), on prend les règles de l'automate dd dont la partie

gauche est l'état interface etat(y) à l'aide du prédicat gardercouple'(x,y).
Soit dd(etat(y),<x1,y1>)->; la règle trouvée. On ajoute alors à l'automate produit la nouvelle règle na(<x1,y1>,etat(<x,y>))->;, ceci avec le prédicat *ajou'(x1,y1,etat(<x,y>))*.

```
subsd'(etat(<x,y>).l1) ->
    gardercouple'(x,y)
    subsd'(l1)
    /;
subsd'(nil) ->;

gardercouple'(x,y) ->
    dd1(etat(y),<x1,y1>)
    ajou'(x1,y1,etat(<x,y>))
    echec;
gardercouple'(x,y) ->;

ajou'(x,y2,y1) -> na(<x,y2>,y1) /;
ajou'(x,y2,y1) -> assert(na(<x,y2>,y1),nil);
```

Puis, on ajoute toutes les autres règles de dd à l'automate produit. Ceci se fait par simple recopie des règles de l'automate dd en les renversant (la partie gauche devient la partie droite et vice et versa).

```
copierddna ->
    dd1(x,y)
    assert(na(y,x),nil)
    echec;
copierddna ->;
```

L'ajout des règles de l'automate dd à l'automate produit possède une complexité de l'ordre du nombre de règles de l'automate dd.
-La dernière étape consiste enfin à calculer le produit de l'automate créé par les deux étapes précédentes avec l'automate reconnaissant la donnée d'arrivée, appelé aa2. (On réalise cet opération avec *intersection"* que l'on étudiera également en annexe)
Pour déterminer si la forêt reconnue par cet automate est non vide, on utilise le prédicat *trouveretat"(t)*, t pouvant prendre 2 valeurs:

* 1, dans le cas où on essaie de transformer une forêt F' en une forêt F" à l'aide du système de réécriture.
*2, dans le cas où on essaie de transformer un arbre t en un arbre t'.

Dans le premier cas *trouveretat''(t)* appelle le prédicat *trouveretat'(x,y)* qui examine si il existe un état couple formé de deux états finaux.

Dans les faits ff1(1,q) se trouvent les états finaux de l'automate aa0 et dans les faits ff1(2,q) se trouvent ceux de l'automate aa2.

```
trouveretat''(2) -> trouveretat'(x,y)/;
trouveretat''(1) -> trouveretat(1.0,2.0);

trouveretat'(x,y) ->couple(<y,x>)
                    ff1(1,etat(y))
                    ff1(2,etat(x))/;
trouveretat'(x,y) ->couple(<x,y>)
                    ff1(2,etat(x))
                    ff1(1,etat(y))/;
trouveretat'(x,y) ->couple(<y,<x,x2>>)
                    ff1(2,etat(y))
                    ff1(1,etat(x));
```

dans le second cas *trouveretat''(t)* appelle *trouveretat(x,y)* qui examine si il existe un état couple formé des états 1.0 et 2.0. Ces deux états étant respectivement, l'état final de l'automate reconnaissant l'arbre t et l'état final de l'automate reconnaissant l'arbre t'.

```
trouveretat(x,y) -> na(x1,etat(<y,x>)) /;
trouveretat(x,y) -> na(x1,etat(<x,y>)) /;
trouveretat(x,y) -> na(x1,etat(<y,<x,x2>>));
```

4.2- Décision avec S'det.

Les algorithmes utilisés dans ce cas on déjà été amplement étudiés au Chapitre 2. Nous allons voir ici comment ils ont été implantés en PROLOG2.
Le prédicat principal *créationm'* a cette fois la forme suivante:

```
creationm' ->
    etg(l1)
    et(l2)
    marquage(l1,1.0)
    marquaged(l2,2.0)
    comparaison
    /;
```

Là aussi, la décision se fait en trois étapes:

-Marquage de l'arbre t.
Il est réalisé par le prédicat *marquage(l1,x)*, l1 étant la liste des états interface de l'automate gg déterminisé et x étant l'état final de l'automate reconnaissant t.

```
marquage(l1,x) ->
    aa1(t,etat(x))
    /
    manip(l1,t,e,x);
```

```
manip(l1,<x,nil>,e,y) ->
    verif''(l1,x,nil,e,y)
    /;
manip(l1,<x,y1>,e,y) ->
    dif(y1,nil)
    manipliste'(l1,y1,y2)
    verif''(l1,x,y2,e,y);

manipliste'(l1,nil,nil) ->
    /;
manipliste'(l1,etat(y).q,y1.q1) ->
    aa1(y2,etat(y))
    /
    manip(l1,y2,y1,y)
    manipliste'(l1,q,q1);

verif''(l1,x,y1,e,y) ->
    gg1(<x,y1>,e)
    verif'(l1,e,y)
    /;
verif''(l1,x,y1,etat("p"),y) ->;

verif'(l1,e,y) ->
    elementde(e,l1)
    ajout'(e,y)
    /;
verif'(l1,e,y) ->;

ajout'(e,y) ->
    aa1(vt(e),etat(y))
    /;
ajout'(e,y) ->
    assert(aa1(vt(e),etat(y)),nil);
```

-Marquage de l'arbre t'

C'est le même algorithme que le précédent. A la différence que l'on utilise l'automate dd déterminisé pour marquer l'arbre t'.
Le prédicat utilisé s'appelle *marquaged(l2,x)*, l2 étant la liste des états interface de l'automate dd déterminisé et x l'état final de l'automate reconnaissant l'arbre t'.

```
marquaged(l2,x) ->
    aa3(t1,etat(x))
    /
    manipd(l2,t1,e,x);

manipd(l2,<x,nil>,e,y) ->
    verifd''(l2,x,nil,e,y)
    /;
manipd(l2,<x,y1>,e,y) ->
    dif(y1,nil)
    maniplisted'(l2,y1,y2)
    verifd''(l2,x,y2,e,y);

maniplisted'(l2,nil,nil) ->
    /;
```

```
maniplisted'(l2,etat(y).q,y1.q1) ->
    aa3(y2,etat(y))
    /
    manipd(l2,y2,y1,y)
    maniplisted'(l2,q,q1);

verifd''(l2,x,y1,e,y) ->
    dd1(<x,y1>,e)
    verifd'(l2,e,y)
    /;
verifd''(l2,x,y1,etat("p"),y) ->;

verifd'(l2,e,y) ->
    elementde(e,l2)
    ajoutd'(e,y)
    /;
verifd'(l2,e,y) ->;

ajoutd'(e,y) ->
    aa3(vt(e),etat(y))
    /;
ajoutd'(e,y) ->
    assert(aa3(vt(e),etat(y)),nil);
```

Chaque fois qu'un sous-arbre de t ou de t' est reconnu respectivement par l'automate gg et l'automate dd, on ajoute une règle à l'automate reconnaissant la donnée en cours de marquage (t ou t') qui aura la forme suivante:

aai(vt(e),etat(y))->; i = 1, 3

y étant l'état atteint après reconnaissance du sous-arbre par l'automate reconnaissant la donnée en cours de marquage et e étant l'état interface atteint après reconnaissance du même sous-arbre par l'automate dd ou gg.

-Comparaison des deux arbres marqués.

Ceci est réalisé par le prédicat *comparaison*.

```
comparaison ->
    aa1(z1,etat(1.0))
    aa3(z2,etat(2.0))
    /
    compar(etat(1.0),etat(2.0),z1,z2);
```

Celui-ci appelle le prédicat *compar(e1,e2,t1,t2)* où t1 et t2 sont les termes reconnus respectivement par e1 et e2. Ces termes peuvent être des marques de la forme vt(e) ou des noeuds de la forme <x,L>. Si on n'a pas trouvé de marques communes à deux sous-arbres de la forme <x,L1> et<x,L2> alors on appelle récursivement *compar* sur chacun des éléments des listes L1 et L2. Ceci à l'aide du prédicat *comparliste(L1, L2)*.

```
compar(e1,e2,vt(x1),vt(x2)) ->
    et(l2)
    cherch(x1,l2,l3)
    elementde(x2,l3)
    /;
compar(e1,e2,vt(x1),vt(x2)) ->
    aa1(<x,l2>,e1)
    aa3(<x1,l1>,e2)
    /
    comparliste(l2,l1);
compar(e1,e2,vt(x1),<x,l>) ->
    aa1(<x,l2>,e1)
    /
    comparliste(l2,l);
compar(e1,e2,<x,l>,vt(x1)) ->
    aa3(<x,l2>,e2)
    /
    comparliste(l,l2);
compar(e1,e2,<x,l1>,<x,l2>) ->
    comparliste(l1,l2);

comparliste(x1.l1,x2.l2) ->
    aa1(z1,x1)
    aa3(z2,x2)
    compar(x1,x2,z1,z2)
    comparliste(l1,l2)
    /;
comparliste(nil,nil) ->;

cherch(etat(y),etat(x).l2,etat(x).l3) ->
    elt'(y,x)
    cherch(etat(y),l2,l3)
    /;
cherch(y,x.l2,l3) -> cherch(y,l2,l3) /;
cherch(y,nil,nil) ->;
```

5-Modification d'un système de réécriture.

5.1- Ajout d'une règle.

Il est intéressant de pouvoir ajouter une règle à un système de réécriture sans avoir à recommencer complétement la compilation. Pour cela, nous procédons de la façon suivante:
Soit B* le GTT d'origine, Soitent gg et dd ses automates associés, et soit ri → li la nouvelle règle à ajouter.

1ère étape:
On associe à la règle ri → li, un GTT B' dont l'automate gauche gg' reconnaîtra le terme ri et l'automate droit dd' génèrera le terme li. Le GTT B' est tel que $E_{B'} \cap E_{B*} = \emptyset$ $E_{B'}$ étant l'ensemble des états de B' et E_{B*} étant l'ensemble des états de B*.

2ième étape:
On fusionne les deux GTT B* et B' afin d'obtenir un nouveau GTT $\mathbb{B}$ dont les automates associés sont **gg**=gg $\cup$ gg' et $\mathbb{dd}$ = dd $\cup$ dd'.

3ième étape:
Il reste alors à calculer l'itération de $\mathbb{B}$ en générant les ε-transitions correspondantes. Nous appellerons ce système $\mathbb{B}$*.

Ces différentes tâches sont remplies par le prédicat *sdrgtt''*.

```
sdrgtt'' -> rule(dernom(m),nil) suppress(1)
            assign(compteur,m)
            nbregl(n1) assign(nr,n1)
            sdrgtt'(1,n1)
            val(compteur,p)
            rule(nbet(r),nil)
            suppress(1)
            val(sub(p,m),r1)
            val(add(r,r1),r2)
            assert(nbet(r2),nil)
            assert(dernom(p),nil)
            rule(inter(l1),nil) suppress(1)
            concat(l1,l,l2)
            assert(inter(l2),nil);
```

Ce prédicat transforme la nouvelle règle en un GTT comme en l'a vu dans la partie 3.1, puis, met à jour les paramètres mentionnés dans cette même partie, soit:

-incrémentation du nombre de règles.
-augmentation du nombre d'états.
-modification du dernier nom d'état généré.
-ajout du nouvel état interface à la liste des états interfaces.

Le nombre de règles est nécessaire pour éviter que plusieurs règles soient numérotées de la même façon. De même, le dernier nom d'état généré est utile pour éviter que deux états différents portent le même nom à l'intérieur des automates modifiés. Le nombre d'états ainsi que la liste des états interface sont utilisés dans les algorithmes de décision du problème de l'accessibilité.

5.2- Suppression d'une règle.

La suppresion d'une règle est plus compliqué que l'ajout. En effet, le calcul des ε-transitions provoque la dispersion des informations un peu partout dans les règles des automates du GTT et il est impossible de retrouver à l'intérieur de ces règles, les informations qui ne concernent que la règle que l'on désire supprimer.
L'unique moyen est de modifier le GTT obtenu avant le calcul des ε-transitions. Comme on l'a vu dans la partie 3.1, le calcul de ce GTT

est réalisé de façon à ce que l'on sache à quelle règle du système de réécriture correspondent les règles du GTT.
On peut décomposer la suppression d'une règle en quatre étapes:

1ère étape:
Suppression des règles du GTT correspondant à la règle du système de réécriture que l'on désire supprimer. Ceci est effectué par le prédicat *suppgtt(n,l)*, *n* étant le numéro de la règle à supprimer et *l* la liste des états qui ne sont plus dans le GTT après suppression des informations concernant la règle *n*.

```
suppgtt(n,y.l)->rule(gg2(n,x,y),nil) suppress(1)
                incrementer(rs)
                suppgtt(n,l)/;
suppgtt(n,x.l)->rule(dd2(n,x,y),nil) suppress(1)
                incrementer(rs)
                suppgtt(n,l)/;
suppgtt(n,nil)->decrementer(rs);
```

2ième étape:
Rénumérotation des règles.
Elle est effectuée dans le GTT par le prédicat *renuméroter'(n)* et dans le système de réécriture par le prédicat *renuméroter(n)*, *n* étant le numéro de la règle supprimée.

```
renumeroter(n) -> val(add(n,1),n1)
                  rule(ss'(n1,x,y,z),nil)
                  suppress(1)
                  assert(ss'(n,x,y,z),nil)
                  renumeroter(n1)/;
renumeroter(n) ->;

renumeroter'(n) -> nbregl(m) val(add(m,1),m1)
                   dif(m1,n) val(add(n,1),n1)
                   numero(n,n1) numero'(n,n1)
                   renumeroter'(n1)/;
renumeroter'(n) ->;

numero(n,n1) -> rule(gg2(n1,x,y),nil)
                suppress(1)
                assert(gg2(n,x,y),nil) echec;
numero(n,n1) ->;
numero'(n,n1)-> rule(dd2(n1,x,y),nil)
                suppress(1)
                assert(dd2(n,x,y),nil) echec;
numero'(n,n1)->;
```

3ième étape:
Mise à jour des trois paramètres suivants:
-diminution du nombre d'états.
-décrémentation du nombre de règles.

-suppression de la liste des états interface, de l'état interface associé à la règle supprimée. Cette opération est effectuée par le prédicat *miseajourinter(i,l)*, *i* étant la liste des états interface et *l* la liste d'états générée par *suppgtt(n,l)*.

```
miseajourinter(i,x.l) -> elementde(x,i)
                            supp(x,i)/;
miseajourinter(i,x.l) -> miseajourinter(i,x);

supp(x,i) -> supp'(x,i,l) rule(inter(i),nil)
             suppress(l)
             assert(inter(l),nil);

"suppression de l'element x de la liste l"
supp'(x,x.i,i)->/;
supp'(x,y.i,y.l)->supp'(x,i,l);
```

4ième étape:
Calcul de l'itération du GTT ainsi modifié.

EMMY
UN SYSTEME DE DEMONSTRATION AUTOMATIQUE PAR REFUTATION POUR LA LOGIQUE DU PREMIER ORDRE AVEC EQUATIONS

1- INTRODUCTION

Emmy est une implantation en Quintus Prolog d'un système de démonstration automatique par réfutation. Il est basé sur un calcul par superposition décrit par Bachmair et Ganzinger [3] et fonctionne pour la logique du premier ordre avec équations. Ce calcul par superposition est complet par réfutation et compatible avec divers mécanismes de simplification et de suppression tels que la réécriture de termes et la suppression de tautologies.

On définit les *clauses* en termes de multi-ensembles. Un *multi-ensemble* sur un ensemble X est une fonction M de X vers l'ensemble des entiers naturels N. Intuitivement M(x) spécifie le nombre d'occurences de x dans M. On dit que x est un *élément* de M si et seulement si M(x)>0.
Pour plus de clarté on utilise souvent une notation ensembliste pour décrire les multi-ensembles. Par exemple, {x,x,x} décrit le multi-ensemble M tel que M(x)=3 et M(y)=0 pour $y \neq x$.

Une *équation* est un multi-ensemble {s,t}, où s et t sont des termes (du premier ordre) construits à partir de variables et de symboles de fonctions. On écrit $s \approx t$ pour exprimer l'équation {s,t}.

Une *clause* est un couple de multi-ensembles d'équations, écrit $\Gamma \rightarrow \Delta$. Le multi-ensemble Γ est appelé *l'antécédent*, et le multi-ensemble Δ, le *successeur*. Les équations qui apparaissent dans l'antécédent sont dites *négatives* et celles qui apparaissent dans le successeur sont dites *positives*. Habituellement, on écrit $\Gamma 1, \Gamma 2$ au lieu de $\Gamma 1 \cup \Gamma 2$; Γ,A ou A,Γ au lieu de $\Gamma \cup \{A\}$; et A1, . . ., Am → B1, . . ., Bn au lieu de {A1, . . ., Am}→{B1, . . ., Bn}.
Une clause A1, . . ., Am→ B1, . . ., Bn peut être assimilée à la représentation d'une implication $A1 \wedge \ldots \wedge Am \supset B1 \vee \ldots \vee Bn$. La clause vide indique une contradiction.
Soit > un *ordre de réduction* sur les termes (on utilisera soit un ordre de chemin lexicographique soit un ordre de chemin sur les multi-ensembles).
En général les littéraux apparaissant dans les clauses sont des équations. Dans le cadre de notre formalisme, on peut représenter des formules atomiques non équationnelles de la façon suivante:
P(t1, . . ., tn)=0, où 0 est un symbole particulier pris comme étant minimal dans l'ordre de réduction > donné.

Si C est une clause

$$s1 \approx t1, \ldots, sm \approx tm \rightarrow u1 \approx v1, \ldots, un \approx vn,$$

on définit sa complexité γ_C par le multi-ensemble (de multi-ensembles)

$$\{\{s1, t1, 1\}, \ldots, \{sm, tm, 1\}, \{u1, v1, 0\}, \ldots, \{un, vn, 0\}\}$$

Autrement dit, la complexité d'une équation si ≈ ti appartenant à l'antécédent est le multi-ensemble {si, ti, 1}. De même, la complexité d'une équation ui ≈ vi appartenant au successeur est le multi-ensemble {ui, vi, 0}.

Enfin, l' *ordre sur les littéraux* dans les clauses est le suivant:
On sait que n'importe quel ordre sur un ensemble S peut être étendu à l'ordre $>_{mul}$ sur les multi-ensembles pris sur S comme suit:
$M >_{mul} N$ si et seulement si (i) $M \neq N$ et (ii) chaque fois que $N(x) > M(x)$ alors $M(y) > N(y)$ pour un y tel que $y > x$.
On définit l'ordre $>_C$ sur les clauses par $C >_C C'$ si et seulement si $\gamma_C (>_{mul})_{mul} \gamma_{C'}$.
On dit qu'une équation A est *maximale* dans une clause $\Gamma \rightarrow \Delta$ si et seulement si $A \geq_{mul} B$, pour tout $B \in \Gamma \cup \Delta$. On dit que A est *strictement maximale* si et seulement si elle est maximale et si et seulement si elle n'apparaît qu'une seule fois dans la clause.

On écrit u[s] pour indiquer que s est un sous-terme de u et (de façon ambiguë) on écrit u[t] le résultat provenant du remplacement d'une occurence particulière de s par t. On note par $t\sigma$ le résultat provenant de l'application de la *substitution* σ à t, et on appelle $t\sigma$ une *instance* de t.
Le calcul par superposition que l'on utilise est constitué par les règles d'inférences suivantes. Celles-ci sont définies en fonction d'un ordre de réduction > donné.

Résolution sur égalités: $\Gamma, s \approx t \rightarrow \Delta$

$$\frac{}{\Gamma\sigma \rightarrow \Delta\sigma}$$

Où σ est le plus grand unificateur de s et de t et $s\sigma \approx t\sigma$ est maximale dans $\Gamma\sigma, s\sigma \approx t\sigma \rightarrow \Delta\sigma$.

Exemple:
Soit $g > f$ un ordre sur les symboles. Alors

$$\frac{g(x) \approx g(y) \rightarrow f(x,y) \approx 0}{\rightarrow f(x,x) \approx 0}$$

est une inférence par résolution sur égalités.

Factorisation ordonnée: $\Gamma \rightarrow \Delta, A, B$

$$\frac{}{\Gamma\sigma \rightarrow \Delta\sigma, A\sigma}$$

où σ est le plus grand unificateur de A et de B, $A\sigma$ est maximale dans $\Gamma\sigma \rightarrow \Delta\sigma, A\sigma$.
Remarquons que pour que la preuve par réfutation soit complète, la factorisation ordonnée n'a besoin d'être appliquée que sur les atomes appartenant au successeur des clauses. Cependant il est également possible d'appliquer la factorisation ordonnée sur les atomes appartenant à l'antécédent.

Exemple:
Soit $f > g > h > p > a$ un ordre sur les symboles. Alors

$$\frac{p(y)=0 \rightarrow f(y,g(a))=0,\ f(h(x),x)=0}{p(h(g(a))=0 \rightarrow f(h(g(a)),g(a))=0}$$

est une inférence par factorisation ordonnée.

Superposition stricte gauche: $\dfrac{\Gamma \rightarrow \Delta, s \approx t \quad u[s'] \approx v, \Lambda \rightarrow \Pi}{u[t]\sigma \approx v\sigma, \Gamma\sigma, \Lambda\sigma \rightarrow \Delta\sigma, \Pi\sigma}$

où (i) σ est le plus grand unificateur de s et de s'.
(ii) $t\sigma \not\geq s\sigma$ et l'équation signée $s\sigma \approx t\sigma$ est strictement maximale dans
$\Gamma\sigma \rightarrow \Delta\sigma, s\sigma \approx t\sigma$.
(iii) $v\sigma \not\geq u\sigma$ et l'équation signée $u\sigma \approx v\sigma$ est maximale dans
$u\sigma \approx v\sigma, \Lambda\sigma \rightarrow \Pi\sigma$.
et (iv) s' n'est pas une variable .

Exemple:
Soit $p > q > r > a > b$ un ordre sur les symboles. Alors

$$\frac{\rightarrow p(a)=0,\ q(b)=0 \qquad p(x)=0,\ r(x)=0 \rightarrow}{r(a)=0 \rightarrow q(b)=0}$$

est une inférence par superposition stricte gauche.

Superposition stricte droite: $\Gamma \rightarrow \Delta, s \approx t \quad \Lambda \rightarrow u[s'] \approx v, \Pi$

$$\frac{}{\Gamma\sigma, \Lambda\sigma \rightarrow u[t]\sigma \approx v\sigma, \Delta\sigma, \Pi\sigma}$$

où (i) σ est le plus grand unificateur de s et de s',
(ii) $t\sigma \not\geq s\sigma$ et l'équation signée $s\sigma \approx t\sigma$ est strictement maximale dans
$\Gamma\sigma \rightarrow \Delta\sigma, s\sigma \approx t\sigma$.
(iii) $v\sigma \not\geq u\sigma$ et l'équation signée $u\sigma \approx v\sigma$ est strictement maximale dans
$\Lambda\sigma \rightarrow u\sigma \approx v\sigma, \Pi\sigma$.
(iv) s' n'est pas une variable.
et (v) $u\sigma \approx v\sigma >_{mul} s\sigma \approx t\sigma$.

Exemple:
Soit $f > g > a$ un ordre sur les symboles. Alors

$$\frac{\rightarrow f(g(x),x)=a \qquad \rightarrow f(f(x1,y),z)=f(x1,f(y,z)}{\rightarrow f(a,z)=f(g(x),f(x,z))}$$

est une inférence par superposition stricte droite.

Paramodulation fusionnante: $\Gamma \rightarrow \Delta, s \approx t \quad \Lambda \rightarrow u \approx v[s'], u' \approx v', \Pi$

$$\frac{}{\Gamma\sigma, \Lambda\sigma \rightarrow u\sigma \approx v[t]\sigma, u\sigma \approx v'\sigma, \Delta\sigma, \Pi\sigma}$$

où (i) σ est la composition $\tau\rho$ du plus grand unificateur τ de s et de s', et du plus grand unificateur ρ de $u\tau$ et de $u'\tau$,
(ii) $t\sigma \not\geq s\sigma$ et l'équation signée $s\sigma \approx t\sigma$ est strictement maximale dans $\Gamma\sigma \rightarrow \Delta\sigma, s\sigma \approx t\sigma$.
(iii) $v\sigma \not\geq u\sigma$ et l'équation signée $u\sigma \approx v\sigma$ est strictement maximale dans
$\Lambda\sigma \rightarrow u\sigma \approx v\sigma, \Pi\sigma$.
(iv) $u\tau > v\tau$ et $v'\sigma \not\geq v\sigma$,
(v) s' n'est pas une variable.

Exemple:
Soit $a > b > c > d$ un ordre sur les symboles. Alors

$$\frac{\rightarrow b=d \qquad \rightarrow a=b,\ a=d}{\rightarrow a=d,\ a=d}$$

est une inférence par paramodulation fusionnante.

En effet, la paramodulation fusionnante est construite de telle façon que son application répétée à des clauses sans variable (en conjonction avec la factorisation ordonnée), provoque la fusion de certains atomes contenant le terme maximal.
Cette règle d'inférence n'est pas implémentée actuellement.

De plus on suppose qu'aucune des clauses utilisées dans les règles d'inférence précédentes n'est une tautologie. Enfin, on suppose que les règles d'inférence précédentes ne sont appliquées qu'à des clauses qui ne partagent aucune variable. (Dans notre implantation les variables sont systématiquement renommées)

2- IMPLANTATION.

Emmy est écrit en Quintus Prolog version 2.4.2. Il fonctionne sur Sun 3 (système Unix).

Taille du fichier source: 113.872 K.
Taille du fichier après compilation: 75.596 K

Les fichiers sources sont:

mainprogram.pl	maximalequa.pl
tools.pl	unification.pl
rpo.pl	substitution.pl
simplification.pl	overlap.pl
engine.pl	

2.1- Aperçu général du système.
La structure générale du système peut être illustrée par l'algorithme suivant:

1- Initialisation.
On a trois ensembles de clauses équationelles:
Clo = ∅, Clnn = ∅, et Cln est l'ensembles des clauses d'entrée.

2- Calculer l'équation maximale pour chaque clause de Cln.

3- Moteur d'inférence.
Tant que Cln ≠ ∅ et que la clause vide n'est pas générée **faire**

3.1- Trier les clauses de Cln.
3.2- Sélectionner la plus petite clause de Cln.
3.3- Appliquer les règles d'inférence à la clause sélectionnée.
Les nouvelles clauses sont mises dans l'ensemble Clnn et leur équation maximale est calculée.
3.4- Mettre la clause sélectionnée dans Clo.
3.5- Simplifier les clauses de Clo, Cln, et Clnn.
3.6- Mettre les clauses de Clnn dans Cln.

4- Si la clause vide est générée, l'ensemble des clauses d'entrée est insatisfiable. Si elle n'est pas générée, l'ensemble des clauses d'entrée est satisfiable.

Expliquons plus en détail cet algorithme:
1- On a différents types d'ensemble de clauses.
a- L'ensemble Clo des clauses marquées.
b- L'ensemble Cln des clauses non marquées.
c- L'ensemble Clnn des nouvelles clauses générées à la fin d'un cycle.

2-Le prédicat *clausemax(T,C)* calcule l'équation maximale de chaque clause de l'ensemble Cln.

3-L'ensemble Cln est trié dans l'ordre croissant. Ce tri est fait de telle façon que l'on puisse à chaque cycle, toujours prendre la plus petite

règle. Ceci doit diminuer l'espace de recherche puisque l'on essaie toujours de ne générer que des petites clauses

Pour le moment les clauses peuvent être triées en fonction de deux critères différents:
-en fonction du nombre de variables et de symboles de fonctions apparaissant dans les clauses.
-en fonction du nombre de symboles de fonctions + 2*(nombre de symboles de variables).
D'autres critères possibles peuvent être ajoutés, par exemple, le nombre de variables distinctes, le nombre d'un symbole de fonction donné . . . etc.

4- Les règles d'inférences de notre système sont appliquées à la clause sélectionnée C.
On applique tout d'abord la résolution sur égalités et la factorisation ordonnée. Ces opérations sont réalisées par les prédicats Prolog *nvuneseuleclause1(T,C)* et *nvuneseuleclause2(T,C)*.
Ensuite, on applique les règles de superposition strictes. On superpose la clause sélectionnée avec chaque clause de Cln (à l'aide du prédicat Prolog *nvdeuxclause2(T,C)*) puis avec chaque clause de Clo (à l'aide du prédicat Prolog *nvdeuxclause1(T,C)*) pourvu que cela n'est pas déjà été fait précédemment.
Lorsque l'on applique les règles de superposition, on superpose les équations maximales de chaque clause avec le prédicat Prolog *unifierequa(E1,E2,R)*, où R est le résultat provenant de la superposition des deux équations E1 et E2.
Parfois ces équations sont maximales si et seulement si certaines conditions sont satisfaites (mémorisées dans les faits Prolog *posmax2*). Après l'unification, on propage les substitutions à l'intérieur de la liste de conditions et on vérifie si elles sont encore vérifiées.

Si ce n'est pas le cas, la nouvelle clause n'est pas générée. Sinon, elle est générée si ce n'est pas une tautologie ou si elle n'existe pas déjà. *Verifiertautologie(L1,R1)* et *clexistedeja(L1,R1,L)* réalisent ces opérations (L1 → R1 est la nouvelle clause et L correspond à sa taille). Chaque nouvelle clause est renommée (avec le prédicat Prolog *renommer(L1,R1,L2,R2)*), de telle façon qu'elle ne partage aucune variable avec les autres clauses.

5-La nouvelle clause générée est mise dans l'ensemble Clnn. La plus petite clause C devient une clause marquée.

6- Les clauses de chaque ensemble sont simplifiées de façon à ce que les clauses redondantes soient supprimées.

2.2- Les structures de données.

On utilise les structures de données suivantes pour représenter les clauses équationnelles.

Une équation est un terme Prolog clos. Les variables sont représentées par des chaînes de caractères commençant par une lettre majuscule. Par exemple

f('X',f('Y','Z')) = f(f('X','Y'),'Z')

est une équation, où 'X', 'Y', et 'Z' sont des variables.

Une clause est un fait Prolog de la forme suivante:

[liste d'équations] → [liste d'équations].

La clause vide est représentée par [] → [].

Lorsque l'utilisateur donne l'ensemble des clauses équationnelles, Emmy lui demande de donner un ordre sur les symboles qui apparaissent dans la théorie. Cet ordre est mémorisé dans des faits Prolog comme suit:

sup1(a,b) signifie que a>b

Cet ordre doit être total. Lorsque l'on a des symboles de prédicats à l'intérieur de nos clauses il apparaît une lettre particulière notée 0. Cette lettre doit être la plus petite.
Par exemple, si on a p('X')=0, cette expression signifie que le prédicat p('X') est vrai.

Emmy demande également à l'utilisateur le nom des symboles de prédicats, de façon à ce que l'on puisse faire la distinction entre les symboles de fonctions et les symboles de prédicats.
Ces noms sont mémorisés dans le fait Prolog pred(F), où F est un symbole de prédicat.

2.3- L'ordre de réduction.

Deux types d'ordres de chemin récursifs sont implémentés dans Emmy:

- l'ordre de chemin lexicographique.
- l'ordre de chemin sur les multi-ensembles.

L'implantation de l'ordre de chemin sur les multi-ensembles est basé sur la définition suivante, qui utilise la distinction entre les symboles de prédicats et les symboles de fonctions.

Soient s et t deux termes.

Alors $s >_{rpo} t$ ssi
soit s est un terme composé et t est une variable qui apparaît dans s.
soit s=f(s1, . . ., sn) et t=g(t1, . . ., tm) et l'une des conditions suivantes est satisfaite:

1- f et g sont des symboles de prédicats et f >g.
2- f est un symbole de prédicat et g ne l'est pas.
3- f et g sont des symboles de fonctions et
si $\geq_{rpo}$ t pour un i, n≥ i ≥ 1
4- f et g sont des symboles de fonctions et
f >g et s $>_{rpo}$ tj pour tout j, m≥ j ≥ 1
5- f et g sont tous deux des symboles de prédicats ou des symboles de fonctions et
f = g et {s1, . . ., sn} $>_{rpomul}$ {t1, . . .,tm}.

La définition de l'ordre de chemin lexicographique est similaire, seule la cinquième condition est différente:

5- f et g sont tous deux des symboles de prédicats ou des symboles de fonctions, f = g et
(i) (s(1), . . ., s(n)) $>_{rpolex}$(t(1),. . ., t(m)).
(ii) f(s1, . . ., sn) $>_{rpo}$ ti pour tout i avec n≥ i ≥ 1

Ces ordres sont calculés par le prédicat prolog *rpo(C,M,N,M1,N1)*, où C est l'ordre choisi (1 pour l'ordre de chemin sur les multi-ensembles, 2 pour l'ordre de chemin lexicographique), M et N les deux termes à ordonner. Si les deux termes M et N sont comparables, alors M1 est le plus grand terme et N1 est le plus petit.
Si les termes sont incomparables, M1 et N1 sont mis à Nil, et des faits représentant une disjonction de conditions (sur les variables de M et de N) sont générés de façon à ce que M soit plus grand que N.
Ces conditions seront utiles pour l'application des règles d'inférence par superposition.
Les règles suivantes génèrent des conditions pour l'ordre de chemin lexicographique:

si s=f(t1, . . .,tn) et t=g(u1, . . ., um),
alors
1- si f > g,
f(t1,. . ., tn) > g(u1, . . ., um)
ssi f(t1, . . ., tn) >u1 et . . . et f(t1,. . ., tn) > um.

2- si g >f,
f(t1,. . ., tn) > g(u1, . . ., um)
ssi t1 ≥ g(u1, . . ., um) ou . . . ou tn ≥ g(u1, . . ., um)
3- si f = g
f(t1,. . ., tn) > g(u1, . . ., um)
ssi (t1 > u1 et f(t1, . . ., tn) >u2 et . . . et f(t1, . . ., tn)>un)
ou (t1 = u1 et t2 > u2 et . . . et f(t1, . . ., tn) >un)
. . .
ou(t1=u1 et t2=u2 et . . . et tn>un)

Pour l'ordre de chemin sur les multi-ensembles les règles sont similaires, mais la troisième règle est plus complexe, du aux permutations que l'on peut faire sur les arguments incomparables.

Ces conditions sont mémorisées dans le fait *necccond(X,Y,C)* où C représente un ensemble de conditions pour que X soit plus grand que Y.

Exemple:

Soit f > g > h > a un ordre sur les symboles.
Soient f('Y',g(a)) et f(h('X'),'X') deux termes.
Alors rpo(2, f('Y',g(a)),f(h('X'),'X'),M,N) dit que ces deux termes sont incomparables et donne la réponse suivante:
M = N = nil avec les deux faits suivants:
necccond(f('Y',g(a)), f(h('X'),'X'), ['Y'>h('X'), f('Y',g(a))>'X')]) et
necccond(f('Y',g(a)),f(h('X'),'X'),['Y'=h('X'), g(a)>'X']).
rpo(1,f('Y',g(a)),f(h('X'),'X'),M,N) dit également que ces deux termes sont incomparables (M = N = nil) et donne les faits suivants:
necccond(f('Y',g(a)),f(h('X'),'X'), [g(a)>'Y', g(a)>'X']),
necccond(f('Y',g(a)),f(h('X'),'X'), ['Y'>g(a), 'Y'>h('X')]) et
necccond(f('Y',g(a)),f(h('X'),'X'), ['Y'>g(a), 'Y'=h('X'), g(a)>'X']).

Mais comme on peut le remarquer sur le troisième fait de cet exemple, ces conditions peuvent être insatisfiables. Donc, une fois que l'ensemble de conditions Z est généré, on vérifie si elles sont satisfiables en utilisant le prédicat *verifiercond(T,Z)*, où T est l'ordre choisi.

Ce test de satisfiabilité procède comme suit. Tout d'abord, on distingue deux ensembles de conditions:
-la partie équationnelle: EP(Z)
-la partie inéquationnelle: IP(Z)

Donc, pour chaque équation de EP(Z) on essaie d'unifier ses deux parties, et si c'est possible, on propage les substitutions aux autres éléments de EP(Z) et de IP(Z).
Si certaines unifications sont impossibles, la condition ne peut pas être satisfaite.
Une fois que cette opération est faite, on regarde si la partie inéquationnelle peut être satisfaite. Pour le moment on contrôle deux choses:

1- Si on n'a pas de condition de la forme suivante:
$s > t[s]$
2- Si on n'a pas deux conditions comme suit:
$x > t$ et $t' > s[x]$ avec $t \geq t'$
ou $s > x$ et $t[x] > s'$ avec $s \geq s'$
ou $s > t$ et $t > s$

en dépit de ces contrôles, il est possible que certaines conditions restent insatisfiables.
Par exemple, si on a $F=\{ \ldots, s, 0\}$ avec $>_F s >_F 0$ alors
$s(x) > y$ et $y > x$
n'a pas de solution puisque, pour tous termes clos x, il n'y a pas de terme entre x et s(x).

Cependant une analyse plus détaillée semble être trop coûteuse.

2.4- Calcul de l'équation maximale.

Soit $\Gamma \rightarrow \Delta$, une clause.

On essaie de trouver l'équation maximale de l'ensemble $\Gamma \cup \Delta$, en utilisant le prédicat *equamax(T,L)*, où T est l'ordre choisi (1 ou 2) et L est une liste d'équations.
Parfois, on peut avoir plusieurs équations maximales possibles. Dans ce cas, pour chaque équation maximale, *equamax* génère une liste de conditions.
Ces conditions sont mémorisées dans le fait *def(N,X,Y,L1)* où N correspond à une possibilité d'orienter les équations (lorsqu'il y a des termes incomparables à l'intérieur de certaines équations). X et Y sont les équations et L1 est l'ensemble des conditions telles que X puisse être plus grande que Y.
Ensuite, ces faits sont formatés d'une façon plus appropriée avec *etudiermax(T, $\Gamma \rightarrow \Delta$)* et *rassemblerposmax($\Gamma \rightarrow \Delta$)*. A la fin on obtient des faits ayant la forme suivante.

posmax1(Γ→ Δ, équation maximale, disjonction de listes de conditions).

Exemple:
Soit f > g un ordre sur les symboles, et soit C=[f('X')='Y', g('Y')='Y'] → [f('X')=g('Y')] une clause.
Alors equamax(2,[f('X')='Y',g('Y')='Y',f('X')=g('Y')]), etudiermax(2,C), et rassemblerposmax(C) génèrent les deux faits *posmax1* suivants:

posmax1([f('X')='Y',g('Y')='Y'] → [f('X')=g('Y')], f('X')=g('Y'), [[f('X')>'Y']]).
posmax1([f('X')='Y',g('Y')='Y'] → [f('X')=g('Y')], g('Y')='Y', [['Y'>f('X')]]).

2.5-Simplification et suppression.

A la fin de chaque cycles, on simplifie les éléments de chaque ensemble Cln, Clo, et Clnn.
On a deux types de simplifications.

- suppression des clauses subsumées.
- simplification par équations.

a- Suppression des clauses subsumées.

Une clause C= Γ → Δ subsume une clause D = Λ → Π si il existe une substitution σ telle que Λ⊃Γσ et Π⊃ Λσ. On dit que C subsume proprement D (et on écrit D ⊳ C) si C subsume D mais pas le contraire. On sait que si une clause C dans N est proprement subsumée par une autre clause C' dans N, où N est un ensemble de clauses, alors elle est redondante et on peut la supprimer.

Exemple:
Soit C=[p('X')=0]→[f('X')='X']. et D=[p(a)=0]→[f(a)=a, r(a)=0]. deux clauses. Alors il est facile de voir que D est subsumée par C.

Dans notre implantation on contrôle si les clauses de Cln et de Clo sont subsumées ou non par l'une des nouvelles clauses de Clnn (Ceci avec le prédicat *subsumero*).
Ensuite, on regarde si les nouvelles clauses de Clnn sont subsumées par l'une des clauses de Clo, Cln et Clnn (avec le prédicat *subsumern*).
Cependant, la suppression des clauses subsumées possède une complexité exponentielle. En effet, prenons par exemple, une clause C de la forme suivante:

[] → [s1=t1, s2=t2, . . ., s6=t6].

et une clause C' de la forme suivante:

[] → [u1=v1,u2=v2,u3=v3].

Il faut trouver les trois équations de C qui peuvent être unifiées avec les équations de C'. Pour cela il faut considérer toutes les permutations de 3 éléments prises parmi les 6 éléments de la partie droite de C.
A cause de ce problème, un seuil a été défini. C'est à dire que l'on ne considère, comme clauses pouvant en subsumer une autre, que les clauses ne possèdant au plus que 3 équations de chaque coté.

b-Simplification par équation.

Pour réaliser ces simplifications, on considère deux nouveaux ensembles de clauses:

-L'ensemble Cltpo des clauses appartenant à Clo ∪ Cln pouvant être utilisées pour les simplifications.

-L'ensemble Cltp des clauses appartenant à Cln du cycle précédent, pouvant être utilisées pour les simplifications.

Les éléments de ces ensembles on la forme suivante:

[]→[X=Y].

où X et Y sont des termes comparables. C'est à dire que X est plus grand que Y.
Avec ces équations on peut simplifier les autres clauses en remplaçant chaque instance de X par l'instance correspondante de Y.
Cette normalisation des clauses est faite à l'aide d'une stratégie en leftmost-innermost, par le prédicat Prolog *changercl(C,C')*.

Exemple:

Soit []→[f(a,'X')='X'] une clause. Notons que f(a,'X') $>_{rpolex}$ 'X'.
Soit [g('X')='X']→[f('Y','Z')=f(a,f('Y','Z'))] une autre clause.
Par simplification, on obtient une nouvelle clause:

[g('X')='X']→[f('Y','Z')=f('Y','Z')].

Cette clause étant une tautologie, elle peut être supprimée.

Tout d'abord, on simplifie les clauses de Clo ∪ Cln par les éléments de Cltp (avec le prédicat *simplclocln1*).
Ensuite, on simplifie les clauses de Clnn par les éléments de Cltp ∪ Cltpo (avec le prédicat *simplclnn1*).
Enfin, on ne simplifie jamais de façon répétée une clause par une autre lorsque ceci a déjà été fait.

Après ces simplifications, on met les éléments de Cltp à l'intérieur de Cltpo, et le nouvel ensemble Cltp est constitué des clauses de la forme []→ [X=Y] appartenant à Clnn.

3 - EXEMPLES.

Les exemples qui suivent sont pris du papier de Francis Jeffry Pelletier[37].
A chaque cycle Emmy donne le temps de calcul correspondant aux superpositions et aux simplifications en millisecondes.
A la fin d'un calcul, on obtient le temps total passé à faire les superpositions et les simplifications, ainsi que le temps de calcul total, (également en millisecondes).

Les exemples sont classés ainsi:

- logique propositionelle.
- logique des prédicats monadiques.
- logique des prédicats (sans l'identité en sans fonction).
- logique des prédicat avec l'identité (sans fonction).
- logique des prédicats avec l'identité et des fonctions arbitraires.

3.1- Logique propositionnelle.

problème 1:

[p=0]→[q=0]. avec p > q
[q=0]→[p=0].
[q=0]→[].
[]→[p=0].

Temps total pour les superpositions: 0.116s
Temps total pour les simplifications: 0.417s
Temps de calcul total: 1.284s
nombre de paires critiques: 0 nonbres de cycles: 1

problème 2:

[p=0]→[].
[]→[p=0].

Temps total pour les superpositions: 0.016s
Temps total pour les simplifications: 0.084s
Temps de calcul total: 0.350s
nombre de paires critiques: 0 nombre de cycles: 1

problème 3:
[]→[p=0]. avec p > q
[q=0]→[].
[]→[q=0].
[p=0]→[].

Temps total pour les superpositions: 0.033s
Temps total pour les simplifications: 0.100s
Temps de calcul total: 0.633s
nombre de paires critiques: 0 nombre de cycles: 1

problème 4:
[]→[p=0,q=0]. avec p > q
[q=0,p=0]→[].
[q=0]→[].
[p=0]→[].

Temps total pour les superpositions: 0.301s
Temps total pour les simplifications: 0.334s
Temps de calcul total: 1.383s
nombre de paires critiques: 2 nombre de cycles: 2

problème 5:
[q=0]→[p=0,r=0]. avec p > q > r
[p=0]→[].
[]→[q=0].
[r=0]→[].

Temps total pour les superpositions: 0.350s
Temps total pour les simplifications: 0.733s
Temps de calcul total: 1.817ms
nombre de paires critiques: 1 nombre de cycles: 1

problème 9:
[]→[p=0,q=0]. avec p > q
[p=0]→[q=0].
[q=0]→[p=0].
[p=0,q=0]→[].

Temps total pour les superpositions: 0.500s
Temps total pour les simplifications: 0.983s
Temps de calcul total: 2.500s

nombre de paires critiques: 1 nombre de cycles: 3

problème 10:

[q=0]→[r=0]. avec r > p > q > r
[r=0]→[p=0].
[r=0]→[q=0].
[p=0]→[q=0,r=0].
[]→[p=0,q=0].
[p=0,q=0]→[].
Temps total pour les superpositions: 2.118s
Temps total pour les simplifications: 2.250s
Temps de calcul total: 6.700s
nombre de paires critiques: 5 nombre de cycles: 8

problème 12:

[]→[p=0,q=0,r=0]. avec p > q > r
[q=0,p=0]→[r=0].
[r=0,p=0]→[q=0].
[r=0,q=0]→[p=0].
[q=0]→[p=0,r=0].
[r=0]→[p=0,q=0].
[p=0]→[q=0,r=0].
[r=0,q=0,p=0]→[].

Temps total pour les superpositions: 3.300s
Temps total pour les simplifications: 4.383s
Temps de calcul total: 11.000s
nombre de paires critiques: 4 nombre de cycles: 10

problème 13:

[]→[p=0,q=0]. avec p > q > r
[]→[p=0,r=0].
[p=0]→[].
[q=0,r=0]→[].

Temps total pour les superpositions: 0.383s
Temps total pour les simplifications: 0.433s
Temps de calcul total: 1.517s
nombre de paires critiques: 2 nombre de cycles: 1

problème 14:
[p=0]→[q=0]. avec p > q
[q=0]→[p=0].
[q=0,p=0]→[].
[]→[p=0,q=0].

Temps total pour les superpositions: 0.350s
Temps total pour les simplifications: 0.834s
Temps de calcul total: 2.183s
nombre de paires critiques: 1 nombre de cycles: 2

problème 15:
[p=0]→[q=0]. avec p > q
[]→[p=0].
[q=0]→[].

Temps total pour les superpositions: 0.117s
Temps total pour les simplifications: 0.317s
Temps de calcul total: 1.050s
nombre de paires critiques: 1 nombre de cycles: 1

problème 17:
[p=0]→[q=0,s=0]. avec s > p > q > r
[p=0,r=0]→[s=0].
[]→[p=0].
[q=0]→[r=0].
[s=0]→[].

Temps total pour les superpositions: 0.866s
Temps total pour les simplifications: 1.583s
Temps de calcul total: 3.716s
nombre de paires critiques: 3 nombre de cycles: 3

3.2- Logique des prédicats monadiques.
problème 18:
[]→[f('X')=0]. avec f > g
[f(g('X1'))=0]→[].

Temps total pour les superpositions: 0s
Temps total pour les simplifications: 0.117s
Temps de calcul total: 0.367s
nombre de paires critiques: 0 nombre de cycles: 1

problème 19:
[p(f('X'))=0]→[q(g('X'))=0]. avec p > q > f > g
[]→[p('X1')=0].
[q('X2')=0]→[].

Temps total pour les superpositions: 0.150s
Temps total pour les simplifications: 0.400s
Temps de calcul total: 1.166s
nombre de paires critiques: 1 nombre de cycles: 1

problème 20:
[p('Y')=0,q('Z')=0]→[r(f('Y','Z'))=0]. avec p > q > r > s > f > a > b
[p('Y1')=0,q('Z1')=0]→[s('X1')=0].
[]→[p(a)=0].
[]→[q(b)=0].
[r('W')=0]→[].

Temps total pour les superpositions: 3.267s
Temps total pour les simplifications: 2.750s
Temps de calcul total: 7.767s
nombre de paires critiques: 5 nombre de cycles: 8

problème 21:
[p=0]→[f(a)=0]. avec p > f > a
[f(b)=0]→[p=0].
[]→[p=0,f('X')=0].
[f('X1')=0,p=0]→[].

Temps total pour les superpositions: 8.801s
Temps total pour les simplifications: 3.951s
Temps de calcul total: 14.533s
nombre de paires critiques: 7 nombre de cycles: 9

problème 22:
[f('X')=0]→[p=0]. avec p > f > a
[p=0]→[f(X1')=0].
[]→[f('Y')=0,p=0].
[p=0,f(a)=0]→[].
[f(a)=0]→[f('Y1')=0].

Temps total pour les superpositions: 9.017s
Temps total pour les simplifications: 3.267s

Temps de calcul total: 14.617s
nombre de paires critiques: 3 nombre de cycles: 7

problème 23:
[]→[p=0,f('X')=0,f('Y')=0]. avec p > f > a > b
[]→[p=0,f('X1)=0,f(b)=0].
[p=0]→[].
[f(a)=0]→[p=0,f('Y1')=0].
[f(a)=0,f(b)=0]→[].

Temps total pour les superpositions: 7.283s
Temps total pour les simplifications: 2.217s
Temps de calcul total: 10.783s
nombre de paires critiques: 5 nombre de cycles: 3

problème 24:
[s('X')=0,q('X')=0]→[]. avec p > q > r > s > a > b
[p('X1')=0]→[q('X1')=0,r('X1')=0].
[]→[p(a)=0,q(b)=0].
[q('X2')=0]→[s('X2')=0].
[r('X3')=0]→[s('X3')=0].
[p('X4')=0,r('X4')=-0]→[].

Temps total pour les superpositions: 30.649s
Temps total pour les simplifications: 8.101s
Temps de calcul total: 43.934s
nombre de paires critiques: 17 nombre de cycles: 21

problème 25:
[]→[p(a)=0]. avec p > q > r > f > g > a > b
[f('X')=0,g('X')=0,r('X')=0]→[].
[p('X1')=0]→[f('X1')=0].
[p('X2')=0]→[g('X2')=0].
[p('X3')=0]→[q('X3')=0,p(b)=0].
[p('X4')=0]→[q('X4')=0,r(b)=0].
[q('X5')=0,p('X5')=0]→[].

Temps total pour les superpositions: 26.485s
Temps total pour les simplifications: 13.951s
Temps de calcul total: 46.084s
nombre de paires critiques: 19 nombre de cycles: 18

problème 27:

[]→[f(a)=0]. avec f > g > h > i > j > a > b
[g(a)=0]→[].
[f('X')=0]→[h('X')=0].
[j('X1')=0,i('X1')=0]→[f('X1')=0].
[h('X2')=0,I('Y')=0,h('Y')=0]→[g('X2')=0].
[]→[j(b)=0].
[]→[i(b)=0].

Temps total pour les superpositions: 5.268s
Temps total pour les simplifications: 4.199s
Temps de calcul total: 12.600s
nombre de paires critiques: 6 nombre de cycles: 8

problème 28:

[p('X')=0]→[q('Y')=0]. avec g > f > s > q > p > r > d > c > b > a
[q(b)=0]→[q(c)=0].
[q(b)=0]→[s(c)=0].
[r(b)=0]→[q(c)=0].
[r(b)=0]→[s(c)=0].
[s('X1')=0,f('Y1')=0]→[g('Y1')=0].
[]→[p(d)=0].
[]→[f(d)=0].
[g(d)=0]→[].

Temps total pour les superpositions: 3.264s
Temps total pour les simplifications: 2.716s
Temps de calcul total: 8.533s
nombre de paires critiques: 5 nombre de cycles: 6

problème 30:

[f('X')=0,h('X')=0]→[]. avec i > f > h > g > a
[]→[g('X1')=0,f('X1')=0].
[g('X2')=0,h('X2')=0]→[].
[]→[g('X3')=0,h('X3')=0].
[]→[i('X4')=0,f('X4')=0].
[]→[i('X5')=0,h('X5')=0].
[i(a)=0]→[].

Temps total pour les superpositions: 2.484s
Temps total pour les simplifications: 1.717s

Temps de calcul total: 5.767s
nombre de paires critiques: 4 nombre de cycles: 3

problème 31:
[f('X')=0,g('X')=0]→[]. avec f > h > j > i > g > a
[f('X1')=0,h('X1')=0]→[].
[]→[i(a)=0].
[]→[f(a)=0].
[]→[h('X2')=0,j('X2')=0].
[i('X3')=0,j('X3')]→[].
Temps total pour les superpositions: 3.933s
Temps total pour les simplifications: 3.234s
Temps de calcul total: 9.183s
nombre de paires critiques: 4 nombre de cycles: 8

problème 32:
[f('X')=0,g('X')=0]→[i('X')=0]. avec f > k > h > j > r > g > a
[f('X1')=0,h('X1')=0]→[i('X1')=0].
[i('X2')=0,h('X2')=0]→[j('X2')=0].
[k('X3')=0]→[[h('X3')=0].
[]→[f(a)=0].
[]→[k(a)=0].
[j(a)=0]→[[].

Temps total pour les superpositions: 5.949s
Temps total pour les simplifications: 5.368s
Temps de calcul total: 14.284s
nombre de paires critiques: 5 nombre de cycles: 10

problème 33:
[p(a)=0]→[p('X')=0,p(c)=0,p('Y')=0]. avec p > a > b > c > d > e
[p(c)=0]→[].
[p(a)=0,p(d)=0]→[p('X1')=0,p(c)=0,p(b)=0].
[p(a)=0,p(b)=0]→[p(c)=0].
[]→[p(a)=0].
[p(e)=0,p(a)=0]→[p(b)=0,p('X2')=0,p(c)=0].
[p(e)=0,p(d)=0]→[p(b)=0].

Temps total pour les superpositions: 8.500s
Temps total pour les simplifications: 14.766s
Temps de calcul total: 41.900s
nombre de paires critiques: 4 nombre de cycles: 4

3.3-Logique des prédicats (sans l'identité et sans fonctions).
problème 35:
[]→[p('X','Y')=0]. avec p > f > g
[p(f('X','Y'),g('X','Y'))=0]→[].

Temps total pour les superpositions: 0s
Temps total pour les simplifications: 0.150s
Temps de calcul total: 0.467s
nombre de paires critiques: 0 nombre de cycles: 1

problème 36:
[]→[f('X',i('X'))=0]. avec h > f > g > i > j > a
[]→[g('X1',j('X1'))=0].
[f('X2','Y')=0,f('Y','Z')=0]→[h('X2','Z')=0].
[f('X3','Y1')=0,g('Y1','Z1')=0]→[h('X3','Z1)=0].
[g('X4','Y2')=0,f('Y2','Z2')=0]→[h('X4','Z2')=0].
[g('X5','Y3')=0,g('Y3','Z3')=0]→[h('X5,'Z3')=0].
[h(a,'X6')=0]→[].
Temps total pour les superpositions: 11.051s
Temps total pour les simplifications: 4.833s
Temps de calcul total: 23.366s
nombre de paires critiques: 6 nombre de cycles: 4

problème 37:
[p('Y','X')=0]→[p(f('X','Y'),g('X'))=0]. avec r > q > p > f > g > h > i > a
[]→[p(f('X1','Y1'),'X1')=0].
[p(f('X2,'Y2),g('X2))=0]→[q(h('X2,'Y2'),g('X2))=0].
[]→[p('X3,'Y3)=0,q(i('X3,'Y3'),'X3)=0].
[q('X4','Y4')=0]→[r('Z','Z')=0].
[r(a,'X5)=0]→[].

Temps total pour les superpositions: 2.017s
Temps total pour les simplifications: 2.017s
Temps de calcul total: 5.517s
nombre de paires critiques: 3 nombre de cycles: 2

problème 39:
[f('X,a)=0,f('X','X')=0]→[]. avec f > a
[]→[f('X1,'X1)=0,f('X1,a)=0].

Temps total pour les superpositions: 2.851s

Temps total pour les simplifications: 0.700s
Temps de calcul total: 6.733s
nombre de paires critiques: 1 nombre de cycles: 2

problème 40:
[f('X',a)=0]→[f('X','X')=0]. avec f > g > a
[f('X1','X1')=0]→[f('X1',a)=0].
[f('X2',g('X2'))=0,f('Y','X2')=0→[].
[]→[f('Y1','X3')=0,f('X3',g('X3'))=0].

Temps total pour les superpositions: 143.984s
Temps total pour les simplifications: 22.935s
Temps de calcul total: 174.250s
nombre de paires critiques: 31 nombre de cycles: 14

problème 41:
[f('X',g('Y'))=0]→[f('X','Y')=0]. avec f > g > a
[f('X1',g('Y1'))=0, f('X1','X1')=0]→[].
[f('X2','Y2')=0]→[f('X2','X2')=0,f('X2,g('Y2'))=0].
[]→[f('X3',a)=0].

Temps total pour les superpositions: 66.784s
Temps total pour les simplifications: 9.101s
Temps de calcul total: 80.683s
nombre de paires critiques: 16 nombre de cycles: 6

problème 42:
[f('X',a)=0,f('X','Y')=0,f('Y','X')=0]→[]. avec f > g > a
[]→[f('X1',g('X1'))=0,f('X1',a)=0].
[]→[f(g('X2'),'X2')=0,f('X2',a)=0].

Temps total pour les superpositions: 11.166s
Temps total pour les simplifications: 1.417s
Temps de calcul total: 19.283s
nombre de paires critiques: 6 nombre de cycles: 5

problème 44:
[f('X')=0]→[g(i('X'))=0]. avec j > f > g >l > h > i > a
[f('X1')=0]→[h('X1',i('X1'))=0].
[f('X2')=0]→[g(j('X2'))=0].
[f('X3')=0,h('X3',j('X3'))=0]→[].
[]→[j(a)=0].

[g('X4')=0]→[h(a,'X4')=0].
[j('X5')=0]→[f('X5')=0].

Temps total pour les superpositions: 6.017s
Temps total pour les simplifications: 5.301s
Temps de calcul total: 14.983s
nombre de paires critiques: 6 nombre de cycles: 6

problème 45:
[f('X')=0,g('Y')=0,h('X','Y')=0]→[g(i('X'))=0,k('Y')=0]. avec f > g > h > j > k > l >i > o > a
[f('X1')=0,g('Y1')=0,h('X1,'Y1')=0]→[h('X1,i('X1))=0,k('Y1')=0].
[f('X2')=0,j('X2,i('X2'))=0,g('Y2')=0,h('X2','Y2')=0]→[k('Y2')=0].
[l('X3')=0,k('X3')=0]→[].
[]→[f(a)=0].
[h(a,'X4')=0]→[l('X4')=0].
[g('X5')=0,h(a,'X5')=0]→[j(a,'X5')=0].
[f('X6')=0]→[g(o('X6'))=0].
[f('X7')=0]→[h('X7',o('X7'))=0].

Temps total pour les superpositions: 50.901s
Temps total pour les simplifications: 23.233s
Temps de calcul total: 83.450s
nombre de paires critiques: 18 nombre de cycles: 23

problème 46:
[f('X')=0]→[f(i('X'))=0,g('X')=0]. avec f > g > h > j > l > a > b
[f('X1')=0]→[h(i('X1'))=0,g('X1')=0].
[f('X2')=0,g(i('X2'))=0]→[g('X2')=0].
[f('X3')=0]→[g('X3')=0,f(a)=0].
[f('X4')=0,g(a)=0]→[g('X4')=0].
[f('X5')=0,f('Y')=0]→[g('X5')=0,g('Y')=0,j(a,'Y')=0].
[f('X6')=0,f('Y1')=0,h('X6','Y1')=0,j('Y1','X6')=0]→[].
[]→[f(b)=0].
[g(b)=0]→[].

Temps total pour les superpositions: 9321.096s
Temps total pour les simplifications: 827.978s
Temps de calcul total: 10326.967s
nombre de paires critiques: 192

En triant en fonction du nombre de symboles de fonctions+ 2*(le nombre de variables), on obtient:
Temps total pour les superpositions: 3780.898s
Temps total pour les simplifications: 398.728s
Temps de calcul total: 4282.817s
nombre de paires critiques: 128 nombre de cycles: 90

problème 47:
[]→[p1(a)=0]. avec p1 > p2 > p3 > p4 > p5 > q1 > r > q0 > p0 > s > i > j > a > b > c > d > e > f
[]→[p2(b)=0].
[]→[p3(c)=0].
[]→[p4(d)=0].
[]→[p5(e)=0].
[]→[q1(f)=0].
[p1('X')=0]→[p0('X')=0].
[p2('X1')=0]→[p0('X1')=0].
[p3('X2')=0]→[p0('X2')=0].
[p4('X3')=0]→[p0('X3')=0].
[p5('X4')=0]→[p0('X4')=0].
[q1('X5')=0]→[p0('X5')=0].
[p3('X6')=0,p5('Y')=0,r('X6','Y')=0]→[].
[p0('X7')=0,q0('Y1')=0,p0('Z')=0,s('Z','X7')=0,q0('W')=0,r('Z','W')=0]→
[r('X7','Y1')=0,r('X7,'Z')=0].
[p4('X8')=0,p3('Y2')=0]→[s('X8','Y2')=0].
[p5('X9')=0,p3('Y3')=0]→[s('X9','Y3')=0].
[p3('X10')=0,p2('Y4')=0]→[s('X10','Y4')=0].
[p2('X11')=0,p1('Y5')=0]→[s('X11','Y5')=0].
[p3('X12')=0,p4('Y6')=0]→[r('X12','Y6')=0].
[p4('X13')=0]→[q0(i('X13'))=0].
[p4('X14')=0]→[r('X14',i('X14'))=0].
[p5('X15')=0]→[q0(j('X15'))=0].
[p5('X16')=0]→[r('X15',j('X16'))=0].
[p1('X17')=0,p2('Y7')=0,r('X17','Y7')=0]→[].
[p1('X18')=0,q1('Y8')=0,r('X18','Y8')=0]→[].
[p0('X19')=0,p0('Y9')=0,q1('Z1')=0,r('Y9','Z1')=0,r('X19','Y9')=0]→[].

En triant en fonction du nombre de symboles de fonctions+ 2*(le nombre de variables), on obtient:
Temps total pour les superpositions: 3835.923s
Temps total pour les simplifications: 1147.067s

Temps de calcul total: 5072.817s
nombre de paires critiques: 152 nombre de cycles: 73

3.4-Logique des prédicats avec l'identité(sans fonctions).

problème 48:
[]→[a=b,c=d]. avec a > b > c > d
[]→[a=c,b=d].
[a=d]→[].
[b=c]→[].

Temps total pour les superpositions: 3.249s
Temps total pour les simplifications: 2.049s
Temps de calcul total: 6.883s
nombre de paires critiques: 7 nombre de cycles: 8

problème 49:
[]→['X'=c,'X'=d]. avec p > a > b > c > d > e
[]→[p(a)=0].
[]→[p(b)=0].
[a=b]→[].
[p(e)=0]→[].

Temps total pour les superpositions: 49.501s
Temps total pour les simplifications: 24.734s
Temps de calcul total: 84.533s
nombre de paires critiques: 29 nombre de cycles: 25

problème 50:
[]→[f(a,'X')=0,f('X','Y')=0]. avec f > a > b
[f('X1',i('X1'))=0]→[].

Temps total pour les superpositions: 1.267s
Temps total pour les simplifications: 0.150s
Temps de calcul total: 4.083s
nombre de paires critiques: 2 nombre de cycles: 1

problème 55:
[]→[l(d)=0], avec k ≥ l > h > r >f > a > b > c > d
[]→[k(d,a)=0].
[]→[l(a)=0].
[]→[l(b)=0].

[]→[l(c)=0].
[l('X')=0]→['X'=a,'X'=b,'X'=c].
[k('X1','Y')=0]→[h('X1','Y')=0].
[k('X2','Y1')=0,r('X2,'Y1')=0]→[].
[h(a,'X3')=0,h(c,'X3')=0]→[].
[]→['X4'=b,h(a,'X4')=0].
[]→[r('X5,a)=0,h(b,'X5')=0].
[h(a,'X6')=0]→[h(b,'X6')=0].
[h('X7',f('X7'))=0]→[].
[a=b]→[].
[k(a,a)=0]→[].

Temps total pour les superpositions: 174.271s
Temps total pour les simplifications: 106.615s
Temps de calcul total: 333.167s
nombre de paires critiques: 37 nombre de cycles: 35

3.5-Logique des prédicats avec l'identité et des fonctions arbitraires.

problème 57:

[]→[f(g(a,b),g(b,c))=0]. avec f > g > a > b > c
[]→[f(g(b,c),g(a,c))=0].
[f('X','Y')=0,f('Y','Z')=0]→[f('X','Z')=0].
[f(g(a,b),g(a,c))=0]→[].

Temps total pour les superpositions: 20.749s
Temps total pour les simplifications: 1.017s
Temps de calcul total: 27.150s
nombre de paires critiques: 3 nombre de cycles: 3

problème58:

[]→[f('X')=g('Y')]. avec f > g > a > b
[f(f(a))=f(g(b))]→[].

Temps total pour les superpositions: 4.999s
Temps total pour les simplifications: 0.400s
Temps de calcul total: 6.666s
nombre de paires critiques: 4 nombre de cycles: 3

problème59:

[f('X')=0,f(g('X'))=0]→[]. avec f > g

[]→[f(g('X1'))=0,f('X1')=0].
[f('X2')=0]→[f(g('X2'))=0].
Temps total pour les superpositions: 0.800s
Temps total pour les simplifications: 0.483s
Temps de calcul total: 2.116s
nombre de paires critiques: 3 nombre de cycles: 2

problème 60:
[f('Y',b)=0]→[f(a,g(a))=0,f('Y',g(a))=0]. avec f > a > b > g > h
[]→[f(a,g(a))=0,f(a,b)=0].
[f(a,'X1')=0,f('Y1',b)=0→[f('Y1',g(a))=0, f(h('X1'),'X1')=0].
[f(a,'X2')=0]→[f(h('X2'),'X2')=0,f(a,b)=0].
[f(a,'X3')=0,f(a,g(a))=0]→[f(h('X3')'X3')=0].
[f(h('X4'),g(a))=0,f(a,'X4')=0,f('Y2',b)=0]→[f('Y2',g(a))=0].
[f(h('X5'),g(a))=0,f(a,'X5')=0]→[f(a,b)=0].
[f(h('X6'),g(a))=0,f(a,'X6')=0, f(a,g(a))=0]→[].

Temps total pour les superpositions: 224.818s
Temps total pour les simplifications: 32.300s
Temps de calcul total: 312.367s
nombre de paires critiques: 21 nombre de cycles: 12

problème 61:
[]→[f('X',f('Y','Z'))=f(f('X','Y'),'Z')]. avec f > a > b > c > d
[f(a,f(b,f(c,d)))=f(f(f(a,b),c),d)]→[].

Temps total pour les superpositions: 0.016s
Temps total pour les simplifications: 0.567s
Temps de calcul total: 1.300s
nombre de paires critiques: 0 nombre de cycles: 1

problème 63:
[]→[f(f('X','Y'),'Z')=f('X',f('Y','Z'))]. avec g > f > a > b > c > d
[]→[f(a,'X1')='X1'].
[]→[f(g('X2'),'X2')=a].
[]→[f(b,c)=f(d,c)].
[b=d]→[].

Temps total pour les superpositions: 92.532s
Temps total pour les simplifications: 65.301s
Temps de calcul total: 162.134s
nombre de paires critiques: 54 nombre de cycles: 16

problème 64:
[]→[f(f('X','Y'),'Z')=f('X',f('Y','Z'))]. avec g > f > a > b > c
[]→[f(a,'X1')='X1'].
[]→[f(g('X2'),'X2')=a].
[]→[f(c,b)=a].
[f(b,c)=a]→[].

Temps total pour les superpositions: 53.766s
Temps total pour les simplifications: 36.466s
Temps de calcul total: 93.366s
nombre de paires critiques: 30 nombre de cycles: 14

problème 65:
[]→[f(f('X','Y'),'Z')=f('X',f('Y','Z'))]. avec g > f > a > b > c
[]→[f(a,'X1')='X1'].
[]→[f('X2','X2')=a].
[f(b,c)=f(c,b)]→[].

Temps total pour les superpositions: 83.585s
Temps total pour les simplifications: 70.966s
Temps de calcul total: 157.184s
nombre de paires critiques: 63 nombre de cycles: 9
Les exemples suivants viennent du papier de McCharen, Overbeek and Wos [32].

Théorie des groupes:
Problème G3:
[]→[f(f('X','Y'),'Z')=f('X',f('Y','Z'))]. avec g > f > a > b
[]→[f(a,'X1')='X1'].
[]→[f(g('X2'),'X2')=a].
[f(b,a)=b]→[].

Temps total pour les superpositions: 12.353s
Temps total pour les simplifications: 8.967s
Temps de calcul total: 28.583s
nombre de paires critiques: 15 nombre de cycles: 8

Problème G4:
[]→[f(f('X','Y'),'Z')=f('X',f('Y','Z'))]. avec g > f > a > b
[]→[f(a,'X1')='X1'].
[]→[f(g('X2'),'X2')=a].

[f(b,'Y1')=a]→[].

Temps total pour les superpositions: 19.881s
Temps total pour les simplifications: 14.215s
Temps de calcul total: 43.750s
nombre de paires critiques: 24 nombre de cycles: 12

Problème G5:
[]→[f(f('X','Y'),'Z')=f('X',f('Y','Z'))]. avec g > k > f > a
[]→[f(a,'X1')='X1'].
[]→[f(g('X2'),'X2')=a].
[f(k('Y1'),'Y1')=k('Y1')]→[].

Temps total pour les superpositions: 13.867s
Temps total pour les simplifications: 10.101s
Temps de calcul total: 31.900s
nombre de paires critiques: 17 nombre de cycles: 9

Les Modèles de Henkin:

Problème Hp1:
[le('X','Y')=0]→[f('X','Y')=e]. le > f > e > d > a
[f('X1','Y1')=e]→[le('X1,'Y1')=0].
[]→[le(f('X2','Y2'),'X2')=0].
[]→[le(f(f('X3','Z'),f('Y3','Z')),f(f('X3','Y3'),'Z'))=0].
[]→[le(e,'X4')=0].
[le('X5,'Y5')=0,le('Y5','X5)=0]→['X5'='Y5'].
[]→[le('X6',d)=0].
[f(a,d)=e]→[].

Temps total pour les superpositions: 0.600s
Temps total pour les simplifications: 0.750s
Temps de calcul total: 6.183s
nombre de paires critiques: 1 nombre de cycles: 1

Problème Hp2:
[le('X','Y')=0]→[f('X','Y')=e]. le > f > e > d > a
[f('X1','Y1')=e]→[le('X1,'Y1')=0].
[]→[le(f('X2','Y2'),'X2')=0].
[]→[le(f(f('X3','Z'),f('Y3','Z')),f(f('X3','Y3'),'Z'))=0].
[]→[le(e,'X4')=0].
[le('X5,'Y5')=0,le('Y5','X5)=0]→['X5'='Y5'].

[]→[le('X6',d)=0].
[]→[f('X7',d)=e].
[f(e,a)=e]→[].

Temps total pour les superpositions: 0.850s
Temps total pour les simplifications: 0.950s
Temps de calcul total: 7s
nombre de paires critiques: 1 nombre de cycles: 1

Problème Hp3:
[le('X','Y')=0]→[f('X','Y')=e]. le > f > e > d > a
[f('X1','Y1')=e]→[le('X1,'Y1')=0].
[]→[le(f('X2','Y2'),'X2')=0].
[]→[le(f(f('X3','Z'),f('Y3','Z')),f(f('X3','Y3'),'Z'))=0].
[]→[le(e,'X4')=0].
[le('X5,'Y5')=0,le('Y5','X5)=0]→['X5'='Y5'].
[]→[le('X6',d)=0].
[]→[f(e,'X7')=e].
[f(a,a)=e]→[].

Temps total pour les superpositions: 242.834s
Temps total pour les simplifications: 98.033s
Temps de calcul total: 357.100s
nombre de paires critiques: 114 nombre de cycles: 28

Problème Hp4:
[le('X','Y')=0]→[f('X','Y')=e]. le > f > e > d > a
[f('X1','Y1')=e]→[le('X1,'Y1')=0].
[]→[le(f('X2','Y2'),'X2')=0].
[]→[le(f(f('X3','Z'),f('Y3','Z')),f(f('X3','Y3'),'Z'))=0].
[]→[le(e,'X4')=0].
[le('X5,'Y5')=0,le('Y5','X5)=0]→['X5'='Y5'].
[]→[le('X6',d)=0].
[]→[f('X7','X7')=e].
[f(a,e)=a]→[].

Temps total pour les superpositions: 496.737s
Temps total pour les simplifications: 199.012s
Temps de calcul total: 718.450s
nombre de paires critiques: 208 nombre de cycles: 35

Problème Hp5:

[le('X','Y')=0]→[f('X','Y')=e]. le > f > e > d >a > a1 > a2
[f('X1','Y1')=e]→[le('X1,'Y1')=0].
[]→[le(f('X2','Y2'),'X2')=0].
[]→[le(f(f('X3','Z'),f('Y3','Z')),f(f('X3','Y3'),'Z'))=0].
[]→[le(e,'X4')=0].
[le('X5,'Y5')=0,le('Y5','X5)=0]→['X5'='Y5'].
[]→[le('X6',d)=0].
[]→[f('X7',e)='X7'].
[]→[le(a,a1)=0].
[]→[le(a1,a2)=0].
[le(a,a2)=0]→[].

Temps total pour les superpositions: 366.482s
Temps total pour les simplifications: 106.567s
Temps de calcul total: 492.700s
nombre de paires critiques: 112 nombre de cycles: 37

Les exemples précédents sont énumérés dans la table qui suit. Ils sont triés en fonction du nombre de paires critiques générées

temps pour les superpositions en sec.	temps pour les simplifications en sec.	temps total en secondes	paires critiques	nombre de cycles	categorie	problème
0.016	0.084	0.350	0	1	1	2
0.000	0.117	0.367	0	1	2	18
0.000	0.150	0.467	0	1	3	35
0.033	0.100	0.633	0	1	1	3
0.116	0.417	1.284	0	1	1	1
0.016	0.567	1.300	0	1	5	61
0.117	0.317	1.050	1	1	1	15
0.150	0.400	1.166	1	1	2	19
0.350	0.733	1.817	1	1	1	5
0.350	0.843	2.183	1	2	1	14
0.500	0.983	2.500	1	3	1	9
0.600	0.750	6.183	1	1	5	Hp1
2.851	0.700	6.733	1	2	3	39
0.850	0.950	7.000	1	1	5	Hp2
0.301	0.334	1.383	2	2	1	4
0.383	0.433	1.517	2	1	1	13
1.267	0.150	4.083	2	1	4	50
0.800	0.483	2.116	3	2	5	59
0.866	1.583	3.716	3	3	1	17
2.017	2.017	5.517	3	2	3	37
9.017	3.267	14.617	3	7	2	22
20.749	1.017	27.150	3	3	5	57
2.484	1.717	5.767	4	3	2	30
4.999	0.400	6.666	4	3	5	58
3.933	3.234	9.183	4	8	2	31
3.300	4.383	11.000	4	10	1	12
8.500	14.766	41.900	4	4	2	33
2.118	2.250	6.700	5	8	1	10
3.267	2.750	7.767	5	8	2	20
3.264	2.716	8.533	5	6	2	28
7.283	2.217	10.783	5	3	2	23
5.949	5.368	14.284	5	10	2	32
5.268	4.199	12.600	6	8	2	27
6.017	5.301	14.983	6	6	3	44
11.116	1.417	19.283	6	5	3	42
11.051	4.833	23.366	6	4	3	36
3.249	2.049	6.883	7	8	4	48
8.801	3.951	14.533	7	9	2	21
12.353	8.967	28.583	15	8	5	G3
66.784	9.101	80.683	16	6	3	41
13.867	10.101	31.900	17	9	5	G5
30.649	8.101	43.934	17	21	2	24
50.901	23.233	83.450	18	23	3	45
26.485	13.951	46.084	19	18	2	25
224.818	32.300	312.367	21	12	5	60
19.881	14.215	43.750	24	12	5	G4
49.501	24.734	84.533	29	25	4	49
53.766	36.466	93.366	30	14	5	64
143.984	22.935	174.250	31	14	3	40
174.271	106.615	333.167	37	35	4	55
92.532	65.301	162.134	54	16	5	63
83.585	70.966	157.184	63	9	5	65
366.482	106.567	492.700	112	37	5	Hp5
242.834	98.033	357.100	114	28	5	Hp3
3780.898	398.728	4282.817	128	90	3	46
3835.923	1147.067	5072.817	152	73	3	47
496.737	199.012	718.450	208	35	5	Hp4

4 - UNE SESSION AVEC EMMY.

Dans cette partie on décrit le processus du système de démonstration automatique de théorème pour le problème 42.
On peut remarquer qu'à chaque cycle, l'ensemble de clauses est divisé en deux parties. La première partie correspond à l'ensemble Clo des clauses sélectionnées et la deuxième partie correspond à l'ensemble Cln des clauses non sélectionnées.

```
> emmy.
                    MENU
          1:creation of a theory
          2:execution
          3:saving a theory
          4:loading a theory
          5:end

          Your Choice?
|: 4.
Name of the file? pb42.
[consulting /u5/sr/leo/GUEST/aline/pb42. . .]
[pb42 consulted 0.250 sec 1,176 bytes]
                    MENU
          1:creation of a theory
          2:execution
          3:saving a theory
          4:loading a theory
          5:end

          Your Choice?
|: 2.
which kind of ordering do you want?
1:multiset path ordering
2:lexicographic path ordering
|: 2.
which kind of criterion do you want
to sort the clauses:
1: in function of the number of symbols(function symbols + variables)
2: in function of the number of function symbols + 2*(the number of
variables)
|: 1.
[]--->[f(Z,g(Z))=0, f(Z,a)=0]

[]--->[f(g(W),W)=0, f(W,a)=0]
```

```
[f(X,a)=0,f(Y,X)=0,f(X,Y)=0]--->[]

time for superposition: 233 ms
time for simplification: 117 ms

[]--->[f(Z,g(Z))=0, f(Z,a)=0]
[]--->[f(g(W),W)=0, f(W,a)=0]

[f(X,a)=0,f(Y,X)=0,f(X,Y)=0]--->[]

time for superposition: 250 ms
time for simplification: 116 ms
[]--->[f(Z,g(Z))=0, f(Z,a)=0]
[]--->[f(g(W),W)=0, f(W,a)=0]
[f(X,a)=0,f(Y,X)=0,f(X,Y)=0]--->[]

[f(g(W1),a)=0,f(W1,g(W1))=0]--->[f(W1,a)=0]
[f(Y1,g(a))=0,f(g(a),Y1)=0]--->[f(a,a)=0]

time for superposition: 6016 ms
time for simplification: 200 ms

[]--->[f(Z,g(Z))=0, f(Z,a)=0]
[]--->[f(g(W),W)=0, f(W,a)=0]
[f(X,a)=0,f(Y,X)=0,f(X,Y)=0]--->[]
[f(g(W1),a)=0,f(W1,g(W1))=0]--->[f(W1,a)=0]

[f(a,g(a))=0]--->[f(a,a)=0]
[f(Y1,g(a))=0,f(g(a),Y1)=0]--->[f(a,a)=0]

time for superposition: 767 ms
time for simplification: 267 ms

[]--->[f(Z,g(Z))=0, f(Z,a)=0]
[]--->[f(g(W),W)=0, f(W,a)=0]
[f(X,a)=0,f(Y,X)=0,f(X,Y)=0]--->[]
[f(g(W1),a)=0,f(W1,g(W1))=0]--->[f(W1,a)=0]

[]--->[f(a,a)=0]

time for superposition: 550 ms
time for simplification: 334 ms
success
Total computation time for superpositions: 11166 ms
```

```
Total computation time for simplifications: 1417 ms
Total computation time: 19283ms
number of critical pairs: 6                    number of cycles: 5

Strike the key <return>
                    MENU
          1:creation of a theory
          2:execution
          3:saving a theory
          4:loading a theory
          5:end

          Your Choice?
I: 5.
bye!
```

ANNEXE 1

Outils de manipulation d'automates

1- Calcul du produit de deux automates.
Le produit de deux automates A et B, où Q_A et Q_B sont leurs ensembles d'états respectifs, se calcule en appliquant les deux règles suivantes:

-Calcul du produit sur les règles reconnaissant les feuilles.
Règle 1:

$$\frac{a \rightarrow q \in A \qquad a \rightarrow s \in B \qquad \forall\, q \in Q_A \text{ et } \forall\, s \in Q_B}{a \rightarrow (q,s) \in A \times B \qquad \text{et } (q,s) \in Q_{A \times B}}$$

- Calcul du produit sur les règles reconnaissant les noeuds.
Règle 2:

$$\frac{\begin{array}{ccc} a & \rightarrow q \in A & a \rightarrow s \in B \qquad \forall i\ (qi,si) \in Q_{A \times B} \\ / \quad \backslash & & / \quad \backslash \\ q1 \ldots qn & & s1 \ldots sn \end{array}}{\begin{array}{c} a \rightarrow (q,s) \in A \times B \quad \text{et} \quad (q,s) \in Q_{A \times B} \\ / \qquad \backslash \\ (q1,s1) \ldots (qn,sn) \end{array}}$$

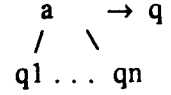
L'implantation de ces deux règles doit être réalisée de façon à ce que la complexité soit quadratique.
Pour cela nous allons générer une structure de données dont les éléments auront la forme suivante:

(n1, n2, l)

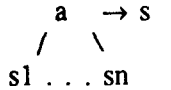
n1: numéro de la règle appartenant à A
n2: numéro de la règle appartenant à B
l: liste de conditions

La liste de conditions est en fait une liste d'états couple qui doivent nécessairement exister pour que l'on puisse associer les deux règles n1 et n2 de la façon décrite dans la Règle2
Exemple:

Si n1 est le numéro de la règle



Si n2 est le numéro de la règle



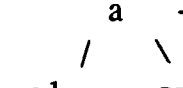
La liste de conditions sera

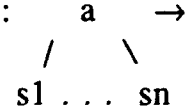
(q1,s1), (q2,s2), . . ., (qn,sn)

Il est évident que dans le cas de règles reconnaissant les feuilles, la liste de conditions sera vide.
Cette structure de données étant remplie, on applique l'algorithme suivant:

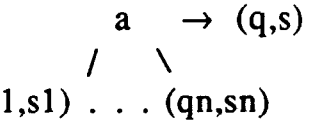
tantque il existe un élement de la forme (n1, n2, nil) (condition nulle)

alors

- soit n1 : 

et n2 : 

générer la règle produit



- mise à jour de la structure de données:
  C'est à dire, suppression de la condition (q,s) de toutes les listes de conditions.

fintq

Complexité:
Le nombre d'éléments présents dans la structure de données est majoré par ||A|| x ||B|| (||M|| étant le nombre de règles de l'automate M). Mais en réalité ce chiffre est toujours très inférieur a cette borne. Ceci, d'autant plus que lors de la création de la structure de données, on peut voir facilement que certaine liste de conditions ne pourront jamais être satisfaites.

Exemple:
Soit la liste de conditions:

(q1,s1), . . ., (qn,sn)

Si il existe un couple (qi,si) tel que qi reconnaît une feuille et si reconnaît un noeud, la liste de conditions ne sera jamais satisfaite. Dans ce cas il est inutile de générer l'élément formé de cette liste de conditions.

Donc le nombre de règle de l'automate produit sera également très inférieure à ||A||x||B||.

Chaque fois que l'on génèrera une nouvelle règle produit, on aura besoin, dans le pire des cas, de parcourir deux fois la structure de données.

- Une fois pour trouver un élément dont la condition est vide.
- Une fois pour mettre à jour la structure de données.

Donc la complexité de temps est au plus égale à 2 ||A||x||B||
Finalement, la complexité de temps de l'algorithme entier est dans le pire des cas en $O((||A||x||B||)^2)$.

Implantation en PROLOG 2 de notre algorithme.

On numérote tout d'abord les règles de deux automates dont on va faire le produit. Après cette opération, nos deux automates sont représentés par les prédicats *aa1(n,x,y)* et *bb1(n,x,y)* n étant le numéro de la règle, x la partie gauche de la règle et y la partie droite.
Voici les prédicats *numéroteraa(n)* et *numéroterbb(n)* qui numérotent les règles des automates aa0 et gg dont on a déjà parlé dans le Chapitre 3.

```
numerotera(n)->  rule(aa0(<x,1>,y),nil)
                 suppress(1)
                 val(add(n,1),n1)
                 assert(aa1(n1,<x,1>,y),nil)
                 numeroteraa(n1)/;
numeroteraa(n)->;

numeroterbb(n)-> rule(gg1(<x,1>,y),nil)
                 suppress(1)
                 val(add(n,1),n1)
                 assert(bb1(n1,<x,1>,y),nil)
                 numeroterbb(n1)/;
numeroterbb(n))->;
```

On fera de même pour les automates aa2 et na, na étant l'automate produit, résultat de l'intersection des automates aa0 et gg.

Ensuite on crée la structure de données à l'aide des prédicats *debut'* et *creerliste'*. Un élément de la structure de données est représenté par le prédicat *intera(n1,n2,l)* n1 et n2 étant les numéros des deux règles et l étant la liste de conditions.
Le prédicat *debut'* va rechercher les règles reconnaissant les feuilles.

```
debut'->aa1(n,<m,nil>,y)
        trouverbb1(n,m,y) echec;
debut'->;

"fourni la règle de bb1 correspondante"

trouverbb1(n,m,y)->bb1(n1,<m,nil>,y1)
                    ajouter'(m,n)/;
trouverbb1(n,m,y)->;

"ajout d'un élément à la structure de données
 dont la liste de conditions est vide"

ajouter'(m,n)->bb1(n1,<m,nil>,y1)
               ajouterstr(n,n1,nil) echec;
ajouter'(m,n)->;
```

Le prédicat *creerliste'* va rechercher les règles reconnaissant les noeuds.

```
creerliste'->aa1(n,<m,l>,y) lettre(m)
            trouverbb1'(m) creerliste(m) echec;
creerliste'->;

creerliste(m)->aa1(n,<m,l>,y)
               bb1(n1,<m,l1>,y1)
               cond(l,l1,l2)
               ajouterstr(n,n1,l2) echec;
creerliste(m)->;

"vérifie si le noeud a déjà été rencontré"

lettre(m) -> let(m)/echec;
lettre(m)->assert(let(m),nil);

"fourni la règle de bb1 correspondante
 pour les noeuds"

trouverbb1'(m)->bb1(n1,<m,l>,y)/;
trouverbb1'(m)->;
```

Le prédicat *cond(l,l1,l2)* crée la liste de conditions. A chaque fois que l'on génère un couple d'états on examine si les deux états ont une chance d'être couplés. Ceci à l'aide du prédicat *verifposs(x,x1)*, x et x1 étant les deux états susceptibles d'être couplés.

```
cond(etat(x).l,etat(x1).l1,<x,x1>.l2)->
        verifposs(x,x1) cond(l,l1,l2);
cond(nil,nil,nil)->;
"vérifie si les deux états ont une chance de
pouvoir être couplés"

verifposs(x,x1)->aa1(n,<m,nil>,etat(x)) non''(x)/
                 bb1(n1,<m1,nil>,etat(x1));
verifposs(x,x1)->aa1(n,<m,nil>,etat(x))
                 aa1(n1,<m1,nil>,etat(x)) dif(l,nil)/;
verifposs(x,x1)->bb1(n,<m,l>,etat(x1)) dif(l,nil);

non''(x)->aa1(n,<m,nil>,etat(x)) dif(l,nil) / echec;
non''(x)->;
```

Le prédicat *ajouterstr(n,n1,l2)* ajoute un élément *intera(n,n1,l2)* à la structure de données.

```
ajouterstr(n,n1,l2)->intera(n,n1,l2)/;
ajouterstr(n,n1,l2)->assert''(intera(n,n1,l2),nil);
```

Le produit est ensuite calculé avec le prédicat *intersection*. Il génère les règles produit à l'aide du prédicat *génèreprod(m,l,ll,<q,q1>)* où <q,q1> est le nouvel état couple. Puis, il met à jour la structure de données à l'aide du prédicat *miseajour'*.

```
intersection->rule(intera(n,n1,nil),nil)
              suppress(1) ajoutrep
              aa1(n,<m,l>,etat(q))
              bb1(n1,<m,l1>,etat(q1))
              genereprod(m,l,l1,<q,q1>)
              echec;
intersection->verifrep miseajour' intersection/;
intersection->;

"ajoute un repert chaque fois que l'on crée une
nouvelle règle produit"

ajoutrep->rep/;
ajoutrep->assert(rep,nil);

"vérifie si un repert a été ajouté"

verifrep->rule(rep,nil)
          suppress(1);
```

Dans *genereprod(m,l,ll,<q,q1>)* chaque fois que l'on ajoute une nouvelle règle produit, on sauvegarde le nouvel état <q,q1> dans le prédicat *couple(x)*. Ce prédicat sera utilisé lors de la mise à jour des listes de conditions à l'intérieur de la structure de données.

```
genereprod(m,nil,nil,<q,q1>)->ajout(m,nil,q,q1);
genereprod(m,l,l1,<q,q1>)->liste(l,l1,l2)
                           ajout(m,l2,q,q1);
```

```
"création de la liste d'états couple, ces états étant
les fils du noeud m"

liste(etat(x).l,etat(x1).l1,etat(<x,x1>).l2)->
                                        liste(l,l1,l2);
liste(nil,nil,nil)->;

"ajout de la nouvelle règle produit"

ajout(m,l2,q,q1)->na(<m,l2>,etat(<q,q1>))/;
ajout(m,l2,q,q1)->assert(na(<m,l2>,etat(<q,q1>)),nil)
                  nvetat(couple(<q,q1>));

"prédicat utilisé pour sauvegarder le nouvel état
couple dans le prédicat couple"

nvetat(p)->p/;
nvetat(p)->assert(p,nil);
```

Le prédicat suivant effectue la mise à jour de la structure de données. La modification des listes de conditions proprement dite est effectuée par le prédicat *nvliste(l,l1)*, l1 étant le liste l mise à jour.

```
miseajour'-> rule-nb(intera,n) assign(compteur,n)
             miseajour;

miseajour->intera(n,n1,l)
           nvliste'(l,n,n1)
           decrementer(compteur) val(compteur,c)
           testercompteur(c)/;
miseajour->;

"vérifie si le compteur est à zero"

testercompteur(0)->/;
testercompteur(n)-> echec;

"modification des listes de conditions dans la
structure de données"

nvliste'(l,n,n1)->nvliste(l,l1)
                  dif(l,l1)
                  rule(intera(n,n1,l),nil)
                  suppress(1)
                  ajouterstr(n,n1,l)/;
nvliste'(l,n,n1)->;


nvliste(<q,q1>.l,l1)->couple(<q,q1>) nvliste(l,l1)/;
nvliste(<j,j1>.l,<j,j1>.l1)->nvliste(l,l1);
nvliste(nil,nil)->;
```

Application au problème de l'accessibilité

Pour décider du problème de l'accessibilité avec S', on a vu que l'automate produit, résultat de l'intersection des automates aa0 et gg, devait continuer de reconnaître la donnée de départ (t reconnu par l'automate aa0) même si il n'en est rien en réalité.
Plus exactement, si l'automate produit ne reconnaît pas la donnée de départ, il faut alors ajouter les règles de l'automate aa0 nécessaires pour que l'on puisse tout de même la reconnaître.
Ceci est effectué par les prédicats *transfaa* et *completaa1*. *transfaa* prend toutes les règles de l'automate aa1 (aa0 numéroté) et examine si on peut substituer tous les états formant la liste de fils du noeud m par les états couples de l'automate produit ne comportant pas de nom d'états interface.
Pour cela, il appelle le prédicat *transfl'(m,y,l,l1)* qui lui même appelle le but *transfl''(l,l1)* qui génère toutes les substitutions possibles, ainsi que le but *verifcouple'(l1,m,y)* qui ajoute une nouvelle règle à l'automate produit si la liste l1 n'est pas uniquement composée d'état couple sans état interface.

```
transfaa->aa1(n,<m,l>,y) transfl'(m,y,l,l1) echec;
transfaa->;

transfl'(m,y,l,l1)->transfl''(l,l1)
                    verifcouple'(l1,m,y)
                    echec;
transfl'(m,y,l,l1)->;

transfl''(nil,nil)->/;
transfl''(etat(x).l,etat(x).l1)->transfl''(l,l1);
transfl''(etat(x).l,etat(<x,y>.l1)->couple(<x,y>)
                                    transfl''(l,l1);

"Ajoute une nouvelle règle à l'automate produit si la
liste l1 n'est pas uniquement formée d'états couple
dont le deuxième élément n'est pas un état interface"

verifcouple'(l1,m,y)->verifcouple(l1)
                      ajoutna'(<m,l1>,y)/;
verifcouple'(l1,m,y)->;

ajoutna'(x,y)->na(x,z)/;
ajoutna'(x,y)->assert(na(x,y),nil);

"tombe en echec si la liste n'est formée que de couple
dont le deuxième élément n'est pas un état interface"

verifcouple(etat(<x,y>).l1)->inter(i)
                             elementde(etat(y),i)/;
verifcouple(etat(<x,y>).l1)->/verifcouple(l1);
verifcouple(etat(x).l1)->/;
verifcouple(nil)->echec;
```

Le prédicat *completaal* ajoute les règles de aa0 reconnaissant les feuilles, si celles-ci ne sont pas déjà présentes dans l'automate produit.

```
completaa1->aa1(n,<m,nil>,e) ajoutna(<m,nil>,e) echec;
completaa1->;

ajoutna(x,y)->na(x,y)/;
ajoutna(x,y)->assert(na(x,y),nil);
```

Donc le prédicat *intersection'* réalisant la première intersection à la forme suivante:

```
intersection'->numeroteraa(0)
               copiergg'
               numeroterbb(0)
               debut'
               creerliste'
               intersection
               transfaa completaa1;

copiergg'->gg(x,y)
           assert(gg1(x,y),nil)
           echec;
copiergg'->;
```

Donc, ce prédicat,

-numérote les automates aa0 et gg.
-crée la structure de données.
-calcul l'automate produit.
-ajoute les règles nécessaires pour que l'automate produit continue de reconnaître la donnée de départ.

La prédicat intersection" réalisant la seconde intersection à la forme suivante:

```
intersection''->supintera suplet supaa1
                supbb1 supcouple
                numeroteraa'(0)
                numeroterbb'(0)
                debut'
                creerliste'
                intersection;
```

Finalement, ce prédicat,

-supprime les informations inutiles générés par l'intersection précédente. (les faits intera, aa1, bb1, couple, et let)
-numérote les automates aa2 et na.
-crée la nouvelle structure de données.
-calcule l'automate produit.

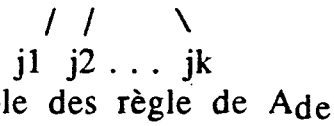
2- Réduction du non déterminisme.

Soit F l'alphabet gradué sur lequel est définie la forêt reconnue par l'automate A.
F est tel que $F = F_0 \cup F_{n>0}$ où F_0 est l'ensemble des lettres d'arité 0 et $F_{n>0}$ est l'ensemble des lettres d'arité supérieure à 0.
Soit E_{det} l'ensemble des états générés par l'algorithme de réduction du non-déterminisme.

L'algorithme de réduction du non-déterminisme est donc le suivant:

Début
/*Initialisation*/
$E_{det} = \varnothing$
Pour chaque élément $m \in F_0$ **faire**
*Prendre l'ensemble des règles de A reconnaissant l'élément m.
Soit $m \rightarrow q1$, $m \rightarrow q2$, . . ., $m \rightarrow qn$ ces règles.
*On crée alors une unique règle de la forme
$m \rightarrow (q1.q2. \ldots . qn.nil)$
qui appartient à l'ensemble des règles de A_{det}.
*$E_{det} = E_{det} \cup \{(q1.q2. \ldots .qn.nil)\}$
fpour
/* Boucle principale */
tantque au moins 1 nouvel état à été ajouté à E_{det} **faire**
Pour chaque élément $p \in F_{n>0}$ **faire**
*calculer l'arité de p, soit n son arité.
*générer toutes les combinaisons possibles de n états appartenant à E_{det}.

Pour chaque combinaison c = j1 j2 . . . jk **faire**
/*j1 j2 . . . jk $\in E_{det}$*/
*Former le nouvel état e qui appartiendra à E_{det}, obtenu lorsque p possède la combinaison c comme liste de fils.
*ajouter



à l'ensemble des règle de A_{det}.
*$E_{det} = E_{det} \cup \{e\}$.
fpour
fpour
ftq
fin
Remarque:

Il est évident que cet algorithme possède une complexité exponentielle. De plus la formation du nouvel état e à l'intérieur de la boucle principale s'effectue en temps exponentiel. En effet, posons,

$$j1 = q_{11}\, q_{12} \ldots q_{1p_1}$$
$$j2 = q_{21}\, q_{22} \ldots q_{2p_2}$$
$$\vdots$$
$$jk = q_{k1}\, q_{k2} \ldots q_{kp_k}$$

Les q_{ij} étant des états appartenant à l'ensemble des états de l'automate A, on appellera ces états, des états élémentaires.
Il faut alors décomposer les ji en états élémentaires. On génère ensuite les différentes combinaisons de k états élémentaires, le ième état q_{il} faisant partie de la liste d'états définissant ji.

Exemple:
Soit j1 = 1.2.5.nil
j2 = 4.6.nil
On obtient les différentes combinaisons suivantes:
(1.4.nil), (1.6.nil), (2.4.nil), (2.6.nil), (5.4.nil), (5.6.nil).

Si p est le noeud étudié, il faut alors examiner, pour chacune de ces combinaisons c, si il existe une règle appartenant à A dont la partie gauche est formée du noeud p avec la combinaison c comme liste de fils. Si c'est la cas, on collecte l'état qui se trouve en partie droite de la règle trouvée.
Il reste alors à rassembler tous les états collectés en une liste, ce qui nous fournit un nouvel état, élément de E_{det}.

Implantation PROLOG 2 de cet algorithme.

La boucle "Pour" décrite dans la partie initialisation de l'algorithme précédent est effectué par le prédicat *detfeuille* .
Ce prédicat forme le nouvel état *etat(y)* où y est une liste d'états élémentaires, à l'aide du prédicat *formeretat(m,z,y)*, m étant la lettre d'arité 0 étudiée. Cet état est sauvegarder dans le prédicat *et(etat(y))*.

```
detfeuille -> rule(aa(<m,nil>,etat(z)),nil)
              suppress(1)
              down("execution")
              formeretat(m,z,y)
              detfeuille
              assert(aa1(<m,nil>,etat(y)),nil)
              assert(et(etat(y)),nil)/;
detfeuille-> down("execution");
```

```
formeretat(m,z,z.y.l)-> rule(aa(<m,nil>,etat(y)),nil)
                        suppress(1)
                        down("execution")
                        suiteetat(m,l)/;
formeretat(m,z,z.nil)->;

suiteetat(m,y.l)-> rule(aa(<m,nil>,etat(y)),nil)
                   suppress(1)
                   down("execution")
                   suiteetat(m,l)/;
suiteetat(m,nil)->;
```

La boucle principale est assurée par le prédicat *detnoeudgene*. ce prédicat appelle le but *detnoeud* qui étudie toutes les lettres d'arité supérieure à 0. L'arité de la lettre étudiée est calculée par le prédicat *compterfils(l,n)*, n étant l'arité de la lettre et l une des listes de fils de la lettre en question, trouvée dans les règles de A. La génération de toutes les combinaisons possibles de n éléments de E_{det} ainsi que leur étude est effectuée par le prédicat *manipliste(m,n)*, m étant la lettre étudiée et n son arité.

```
detnoeudgene->detnoeud
               et1(x)
               creatnvetat
               supsais
               detnoeudgene/;
detnoeudgene->;

detnoeud->aa(<m,l>,y)
          verifier(m)
          assign(compteur,0)
          compterfils(l,n)
          manipliste(m,n)
          echec;
detnoeud->;
```

Le prédicat *manipliste(m,n)* appelle le but *combin(n,l3)* qui génère une à une, dans l'argument l3, les combinaisons possibles de n éléments de E_{det}. Puis, il appelle le but *calcul(n,m,l3)* qui va former l'état obtenu lorsque le noeud m possède la liste l3 comme liste de fils. Ceci avec le prédicat *examgene(n,m,l3)*.

```
manipliste(m;n)->combin(n,l3)
                 calcul(n,m,l3);

combin(0,nil)->/;
combin(n,y.l3)->val(sub(n,1),n1)
                et(y)
                combin(n1,l3);

calcul(n,m,l3)->aa1(<m,l3>,y)/;
calcul(n,m,l3)->examgene(n,m,l3);
```

Le prédicat *examgene(n,m,l)* est composé de deux clauses.
La première clause génère le nouvel état à l'aide du prédicat *exam(n,m,l,l2)*, où l2 est le nouvel état en question, et le sauvegarde, si il n'existe pas déjà, dans le prédicat *et1(l2)*.
La seconde clause concerne le cas où l'on arrive dans un état puit.

```
examgene(n,m,l)->exam(n,m,l,l2)
                 estcevide(l2)
                 suptermedouble(l2,l4)
                 existedeja(l4,l6)
                 assert(aa1(<m,l>,etat(l6)),nil)
                 assert(et1(etat(l6)),nil)/;
examgene(n,m,l)->;
```

Le prédicat *exam(n,m,l,l2)* est également composé de deux clauses.
La première décompose les états formant la liste l en états élémentaires comme on l'a vu précédemment, puis génère une à une les différentes combinaisons possibles à l'aide du prédicat *combin2(n,nl,ll)* qui récupère ces combinaisons dans l'argument ll. Enfin, pour chacune des combinaisons ll, le prédicat *calcul2(m,ll,y)* récupère dans l'argument y l'état obtenu lorsque le noeud m a la combinaison ll comme liste de fils. y est sauvegardé dans le prédicat *etint(y)* .
La seconde clause rassemble les états sauvegardés dans les faits etint(y) par la première clause, en une liste qui correspondra au nouvel état élément de E_{det}. Ceci grace au prédicat *concatgene1(nil,l2)*, l2 étant le nouvel état en question.

```
exam(n,m,l,l2)->new-subworld("intermed")
                garderetat(0,l)
                combin2(n,0,ll)
                calcul2(m,ll,y)
                echec;
exam(n,m,l,l2)->concatgene1(nil,l2)
                climb("execution")
                kill-subworld("intermed");

concatgene1(l1,l2)->rule(etint(l),nil)
                    suppress(l)
                    concat(l1,l,l3)
                    concatgene1(l3,l2)/;
concatgene1(l1,l1)->;
```

Le prédicat *garderétat(n,l)* décompose, par exemple, la liste etat(1.2.5.nil).etat(4.6.nil) en faits qui auront la forme suivante:

```
et2(1, etat(1))->;      et2(2,etat(4))->;
et2(1,etat(2))->;       et2(2,etat(6))->;
et2(1,etat(5))->;
```

```
garderetat(n,nil)->/;
garderetat(n,etat(x).l)->val(add(n,1),n1)
                         copieretat(n1,x)
                         garderetat(n1,l);
copieretat(n,nil)->/;
copieretat(n,x.q)->assert''(et2(n,etat(x)),nil)
                   copieretat(n,q);
```

Si on reprend l'exemple précedent, *combin2(n,n1,l1)* générera les combinaisons suivantes:

(1.4.nil), (1.6.nil), (2.4.nil), (2.6.nil), (5.4.nil), (5.6.nil).

```
combin2(0,n1,nil)->/;
combin2(n,n1,x.l)->val(sub(n,1),n2)
                   val(add(n1,1),n3)
                   et2(n3,x)
                   combin2(n2,n3,l);
```

Le prédicat *calcul2(m,l3,y)* appelle le prédicat *noeud(m,l3,y)* qui va rechercher toutes les règles de A ayant en partie gauche le noeud m avec la liste de fils l3, puis concatène les parties droites de façon à former l'état obtenu dans ce cas.

```
calcul2(m,l3,y)->noeud(m,l3,y)
                 down("execution")
                 down("intermed")
                 dif(y,nil)
                 assert(etint(y),nil);

noeud(m,l1,y.l)->rule(aa(<m,l1>,etat(y)),nil)
                 suppress(1)
                 down("execution")
                 noeud(m,l1,l)
                 assert(aa(<m,l1>,etat(y)),nil)/;
noeud(m,l1,nil)->;
```

3- Suppression des états inutiles dans les automates d'arbres.

3.1- Suppression des états inaccessibles.

Définition: *Un état q est* ***inaccessible*** *ssi la forêt reconnue par q est vide.*

Soit A_q l'ensemble des états de l'automate A.
A' sera l'automate obtenu après suppression des états inaccessibles.

Algorithme:

Debut
/*Initialisation*/
U' = { x | ∋ a → x ∈ A où a est une lettre d'arité 0 et x ∈ Aq }
U = ∅
/*Boucle principale*/
Tant que U' ≠ U **faire**
{recherche de nouveaux états accessibles}
U ← U'
U' ← U ∪ {x ∈ A | ∋ b → x ∈ A, q1, . . ., qp ∈ U}
/ / \ \
q1 . . . qp
ftq
A'q← U'
A' ← { t → x ∈ A | x ∈ A'q et t est un terme dont les feuilles appartiennent à A'q }
fin

Implantation PROLOG2
La partie initialisation est assurée par le prédicat *Supinacc* .

```
supinacc->aa(<y,nil>,x)
          assert(pris(x),nil) echec;
supinacc->;
```

Les états accessibles sont sauvegardés dans le prédicat *pris(x)*.
La recherche des nouveaux états accessibles à l'intérieur de la boucle principale et la suppression des règles comportant au moins un état pris est effectué par le prédicat *supinacc''*.

```
supinacc''->supinacc'
            verifrep
            supinacc''/;
supinacc''->aa(<y,l>,x)
            eltpris''(y,l,x) echec;
```

La première clause de *supinacc''* va rechercher les nouveaux états accessibles à l'aide du prédicat *supinacc'*.
suppinacc' prend chaque règle de A de la forme aa(<y,l>,x) et examine avec eltpris'(l,x) si tous les éléments de l sont des états accessibles. Si c'est la cas x devient également un état accessible.

```
supinacc'->aa(<y,l>,x)
           eltpris'(l,x) echec;
supinacc'->;

eltpris'(l,x)->eltpris(l) ajoutpris(x)/;
```

```
eltpris(x.l)->pris(x) eltpris(l);
eltpris(nil)->;

ajoutpris(x)->pris(x)/;
ajoutpris(x)->assert(pris(x),nil) ajoutrep;
```

La seconde clause de *supinacc''* supprime les règles contenant au moins un état inaccessible. Ceci à l'aide du prédicat *eltpris''(y,l,x).*

```
eltpris''(y,l,x)->eltpris(l) pris(x)/;
eltpris''(y,l,x)->rule(aa(<y,l>,x),nil) suppress(l);
```

3.2- Suppression des états ne menant pas à un état final.(Etats pris)

Définition: *Un état q est non-pris ssi il exite un état final f tel que*

$$\begin{array}{l} t \Rightarrow_A f \\ | \\ q \end{array}$$

Soit A_f l'ensemble des états finaux de A.
A'' sera l'automate obtenu après suppression des états pris.

Algorithme:

Debut
{Initialisation}
$S \leftarrow \varnothing$
$S' \leftarrow A_f$
{Boucle principale}
Tantque $S \neq S'$ **faire**
{recherche de nouveaux états non-pris}
$S \leftarrow S'$
$S' \leftarrow S \cup \{x \in A'_q \mid \exists y \in S,\ b \rightarrow y \in A'\}$

$$/\ |\ .\ .\ \backslash$$
$$x$$

ftq
$A''_q \leftarrow S'$
$A'' \leftarrow \{t \rightarrow x \in A' \mid x \in A''_q\}$
fin

Implantation PROLOG 2

La recherche des états non-pris est effectuée par le prédicat *suppris'(l),* l est l'ensemble des états finaux de l'automate A.
Pour chaque élément de l, *suppris'(l)* appelle le prédicat *suppris(x)* ou x est un état final.

```
suppris'(x.l)-> suppris(x)
                suppris'(l);
suppris'(nil)->;
```

Suppris(x) va rechercher toutes les règles de la forme aa(<y,z>,x) appartenant à A et sauvegarde, à l'aide du prédicat *supprisl(z)*, tous les états présents dans la liste z dans le prédicat *etats(q)*. Ces états étant des états non-pris, on appelle récursivement Suppris(x) sur chacun de ces états.

```
suppris(x)->aa(<y,z>,x) ajoutetat(x)
            supprisl(z) echec;
suppris(x)->;

supprisl(x.l)->ajoutetat(x)
               supprisl(l);
supprisl(nil)->;

ajoutetat(x)->inter(l) elt(x,l)/;
ajoutetat(x)->etats(x)/;
ajoutetat(x)->assert(etats(x),nil)
              suppris(x);
```

Ensuite on supprime toutes les règles dont l'état présent en partie droite est pris. Ceci est effectué par le prédicat *inut*.

```
inut->aa(y,x)
      nuti(x)
      supprimerregle(x) echec;
inut->;

nuti(x)->inter(l) elt(x,l)/echec;
nuti(x)->etats(x)/echec;
nuti(x)->;

supprimerregle(x)->rule(aa(y,x),nil) suppress(1)
                   supprimerregle(x)/;
supprimerregle(x)->;
```

ANNEXE 2

Exemples

1- Exemple d'un système de réécriture clos.

Reprenons l'exemple décrit au chapitre 2 à la page 52. Nous apellerons ce système R2.
Lors de la saisie du système de réécriture, celui-ci est sauvegardé dans la mémoire de travail sous la forme suivante:

```
ss(12,<"trunk",<"wings",nil>.<"trunk",<"limbs",nil>.<"limbs",nil>.nil>.nil>,<
   "body",nil>,1) ->;
ss(11,<"evtrunk",<"limbs",nil>.<"leg",<"foot",nil>.nil>.<"limbs",nil>.nil>,<
   "trunk",<"wings",nil>.<"limbs",nil>.nil>,6) ->;
ss(10,<"body",nil>,<"evtrunk",<"limbs",nil>.<"leg",<"foot",nil>.nil>.<"limbs",
   nil>.nil>,13) ->;
ss(9,<"limbs",nil>,<"arm",<"limbs",nil>.nil>,18) ->;
ss(8,<"arm",<"limbs",nil>.nil>,<"limbs",nil>,20) ->;
ss(7,<"foot",nil>,<"inflimbs",<"foot",nil>.<"foot",nil>.nil>,22) ->;
ss(6,<"foot",nil>,<"leg",<"foot",nil>.nil>,25) ->;
ss(5,<"limbs",nil>,<"trunk",<"limbs",nil>.<"limbs",nil>.nil>,27) ->;
ss(4,<"trunk",<"limbs",nil>.<"limbs",nil>.nil>,<"limbs",nil>,30) ->;
ss(3,<"body",nil>,<"neck",<"body",nil>.nil>,33) ->;
ss(2,<"neck",<"body",nil>.nil>,<"body",nil>,35) ->;
ss(1,<"limbs",nil>,<"trunk",<"wings",nil>.<"limbs",nil>.nil>,37) ->;
```

Les états interface sont les suivants:

```
inter(etat(1).etat(6).etat(13).etat(18).etat(20).etat(22).etat(25).etat(27).etat
   (30).etat(33).etat(35).etat(37).nil) ->;
```

Le GTT associé à ce système après suppression des ε-transitions possède l'automate gauche suivant:

```
gg(<"wings",nil>,etat(2)) ->;
gg(<"limbs",nil>,etat(4)) ->;
gg(<"limbs",nil>,etat(5)) ->;
gg(<"trunk",etat(4).etat(5).nil>,etat(3)) ->;
gg(<"trunk",etat(2).etat(3).nil>,etat(16)) ->;
gg(<"limbs",nil>,etat(6)) ->;
gg(<"limbs",nil>,etat(10)) ->;
gg(<"foot",nil>,etat(12)) ->;
gg(<"leg",etat(12).nil>,etat(11)) ->;
gg(<"limbs",nil>,etat(13)) ->;
gg(<"evtrunk",etat(10).etat(11).etat(13).nil>,etat(16)) ->;
gg(<"body",nil>,etat(16)) ->;
gg(<"limbs",nil>,etat(21)) ->;
gg(<"limbs",nil>,etat(25)) ->;
gg(<"limbs",nil>,etat(26)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(21)) ->;
gg(<"limbs",nil>,etat(30)) ->;
gg(<"arm",etat(30).nil>,etat(21)) ->;
gg(<"foot",nil>,etat(31)) ->;
gg(<"foot",nil>,etat(34)) ->;
gg(<"body",nil>,etat(39)) ->;
gg(<"neck",etat(39).nil>,etat(16)) ->;
gg(<"trunk",etat(2).etat(3).nil>,etat(39)) ->;
gg(<"limbs",nil>,etat(16)) ->;
gg(<"evtrunk",etat(10).etat(11).etat(13).nil>,etat(39)) ->;
gg(<"limbs",nil>,etat(3)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(3)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(4)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(5)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(6)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(10)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(13)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(25)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(26)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(30)) ->;
gg(<"arm",etat(30).nil>,etat(3)) ->;
gg(<"arm",etat(30).nil>,etat(4)) ->;
gg(<"arm",etat(30).nil>,etat(5)) ->;
gg(<"arm",etat(30).nil>,etat(6)) ->;
gg(<"arm",etat(30).nil>,etat(10)) ->;
gg(<"arm",etat(30).nil>,etat(13)) ->;
gg(<"arm",etat(30).nil>,etat(25)) ->;
gg(<"arm",etat(30).nil>,etat(26)) ->;
gg(<"arm",etat(30).nil>,etat(30)) ->;
gg(<"foot",nil>,etat(11)) ->;
gg(<"neck",etat(39).nil>,etat(39)) ->;
gg(<"limbs",nil>,etat(39)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(16)) ->;
gg(<"arm",etat(30).nil>,etat(16)) ->;
gg(<"trunk",etat(25).etat(26).nil>,etat(39)) ->;
gg(<"arm",etat(30).nil>,etat(39)) ->;
```

Son automate droit est le suivant:

```
dd(etat(21),<"arm",etat(16).nil>) ->;
dd(etat(21),<"trunk",etat(22).etat(16).nil>) ->;
dd(etat(21),<"trunk",etat(16).etat(23).nil>) ->;
dd(etat(21),<"trunk",etat(16).etat(16).nil>) ->;
dd(etat(21),<"trunk",etat(16).etat(6).nil>) ->;
dd(etat(21),<"trunk",etat(16).etat(21).nil>) ->;
dd(etat(21),<"trunk",etat(6).etat(16).nil>) ->;
dd(etat(21),<"trunk",etat(21).etat(16).nil>) ->;
dd(etat(16),<"trunk",etat(14).etat(16).nil>) ->;
dd(etat(6),<"trunk",etat(7).etat(16).nil>) ->;
dd(etat(16),<"evtrunk",etat(17).etat(18).etat(16).nil>) ->;
dd(etat(16),<"evtrunk",etat(16).etat(18).etat(20).nil>) ->;
dd(etat(16),<"evtrunk",etat(16).etat(18).etat(16).nil>) ->;
dd(etat(16),<"evtrunk",etat(16).etat(18).etat(6).nil>) ->;
dd(etat(16),<"evtrunk",etat(16).etat(18).etat(21).nil>) ->;
dd(etat(16),<"evtrunk",etat(6).etat(18).etat(16).nil>) ->;
dd(etat(16),<"evtrunk",etat(21).etat(18).etat(16).nil>) ->;
dd(etat(16),<"evtrunk",etat(21).etat(18).etat(21).nil>) ->;
dd(etat(16),<"evtrunk",etat(21).etat(18).etat(6).nil>) ->;
dd(etat(16),<"evtrunk",etat(21).etat(18).etat(20).nil>) ->;
dd(etat(16),<"evtrunk",etat(6).etat(18).etat(21).nil>) ->;
dd(etat(16),<"evtrunk",etat(6).etat(18).etat(6).nil>) ->;
dd(etat(16),<"evtrunk",etat(6).etat(18).etat(20).nil>) ->;
dd(etat(16),<"evtrunk",etat(17).etat(18).etat(21).nil>) ->;
dd(etat(16),<"evtrunk",etat(17).etat(18).etat(6).nil>) ->;
dd(etat(6),<"trunk",etat(7).etat(21).nil>) ->;
dd(etat(6),<"trunk",etat(7).etat(6).nil>) ->;
dd(etat(16),<"trunk",etat(14).etat(21).nil>) ->;
dd(etat(16),<"trunk",etat(14).etat(6).nil>) ->;
dd(etat(18),<"leg",etat(34).nil>) ->;
dd(etat(18),<"leg",etat(31).nil>) ->;
dd(etat(21),<"trunk",etat(21).etat(21).nil>) ->;
dd(etat(21),<"trunk",etat(21).etat(6).nil>) ->;
dd(etat(21),<"trunk",etat(21).etat(23).nil>) ->;
dd(etat(21),<"trunk",etat(6).etat(21).nil>) ->;
dd(etat(21),<"trunk",etat(6).etat(6).nil>) ->;
dd(etat(21),<"trunk",etat(6).etat(23).nil>) ->;
dd(etat(21),<"trunk",etat(22).etat(21).nil>) ->;
dd(etat(21),<"trunk",etat(22).etat(6).nil>) ->;
dd(etat(21),<"arm",etat(21).nil>) ->;
dd(etat(21),<"arm",etat(6).nil>) ->;
dd(etat(31),<"inflimbs",etat(34).etat(34).nil>) ->;
dd(etat(31),<"inflimbs",etat(34).etat(31).nil>) ->;
dd(etat(31),<"inflimbs",etat(34).etat(33).nil>) ->;
dd(etat(31),<"inflimbs",etat(31).etat(34).nil>) ->;
dd(etat(31),<"inflimbs",etat(31).etat(31).nil>) ->;
dd(etat(31),<"inflimbs",etat(31).etat(33).nil>) ->;
dd(etat(31),<"inflimbs",etat(32).etat(34).nil>) ->;
dd(etat(31),<"inflimbs",etat(32).etat(31).nil>) ->;
dd(etat(34),<"leg",etat(34).nil>) ->;
dd(etat(34),<"leg",etat(31).nil>) ->;
dd(etat(16),<"neck",etat(16).nil>) ->;
dd(etat(16),<"neck",etat(37).nil>) ->;
dd(etat(34),<"leg",etat(35).nil>) ->;
dd(etat(31),<"inflimbs",etat(32).etat(33).nil>) ->;
```

```
dd(etat(21),<"arm",etat(28).nil>) ->;
dd(etat(21),<"trunk",etat(22).etat(23).nil>) ->;
dd(etat(18),<"leg",etat(19).nil>) ->;
dd(etat(16),<"trunk",etat(14).etat(15).nil>) ->;
dd(etat(6),<"trunk",etat(7).etat(8).nil>) ->;
dd(etat(16),<"evtrunk",etat(17).etat(18).etat(20).nil>) ->;
dd(etat(37),<"body",nil>) ->;
dd(etat(35),<"foot",nil>) ->;
dd(etat(33),<"foot",nil>) ->;
dd(etat(32),<"foot",nil>) ->;
dd(etat(28),<"limbs",nil>) ->;
dd(etat(21),<"limbs",nil>) ->;
dd(etat(23),<"limbs",nil>) ->;
dd(etat(22),<"limbs",nil>) ->;
dd(etat(20),<"limbs",nil>) ->;
dd(etat(19),<"foot",nil>) ->;
dd(etat(17),<"limbs",nil>) ->;
dd(etat(15),<"limbs",nil>) ->;
dd(etat(14),<"wings",nil>) ->;
dd(etat(8),<"limbs",nil>) ->;
dd(etat(7),<"wings",nil>) ->;
dd(etat(16),<"body",nil>) ->;
```

Ce GTT n'est pas déterministe, après réduction du non-déterminisme on obtient le GTT G' composé des automates gg1 et dd1 suivants:

```
gg1(<"leg",etat(25.22.9.8.nil).nil>,etat(8.nil)) ->;
gg1(<"neck",etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).nil>,etat(36.33.
   13.35.6.nil)) ->;
gg1(<"neck",etat(1.13.33.36.6.35.nil).nil>,etat(36.33.13.35.6.nil)) ->;
gg1(<"arm",etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).nil>,etat(1.36.35
   .33.13.6.37.32.31.27.21.18.3.30.20.nil)) ->;
gg1(<"trunk",etat(2.nil).etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).nil
   >,etat(1.13.33.36.6.35.nil)) ->;
gg1(<"trunk",etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).etat(1.36.35.33
   .13.6.37.32.31.27.21.18.3.30.20.nil).nil>,etat(1.36.35.33.13.6.37.32.31.27.21
   .18.3.30.20.nil)) ->;
gg1(<"neck",etat(36.33.13.35.6.nil).nil>,etat(36.33.13.35.6.nil)) ->;
gg1(<"limbs",nil>,etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)) ->;
gg1(<"body",nil>,etat(36.33.13.35.6.nil)) ->;
gg1(<"foot",nil>,etat(25.22.9.8.nil)) ->;
gg1(<"wings",nil>,etat(2.nil)) ->;
gg1(<"evtrunk",etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).etat(25.22.9.
   8.nil).etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).nil>,etat(1.13.33.
   36.6.35.nil)) ->;
gg1(<"evtrunk",etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).etat(8.nil).
   etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil).nil>,etat(1.13.33.36.6.35
   .nil)) ->;
```

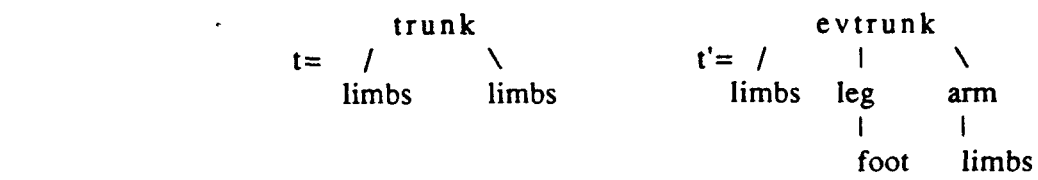
```
ddl(<"evtrunk",etat(35.34.1.nil).etat(15.25.nil).etat(39.30.29.28.20.19.17.14.12
   .nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(35.34.1.nil).etat(15.25.nil).etat(18.nil).nil>,etat(13.nil))
    ->;
ddl(<"evtrunk",etat(35.34.1.nil).etat(15.25.nil).etat(6.37.nil).nil>,etat(13.nil
   )) ->;
ddl(<"evtrunk",etat(5.34.13.nil).etat(15.25.nil).etat(27.nil).nil>,etat(13.nil))
    ->;
ddl(<"evtrunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(15.25.nil).etat(35.34.1
   .nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(35.34.1.nil).etat(15.25.nil).etat(35.34.1.nil.nil).nil>,etat
   (13.nil)) ->;
ddl(<"evtrunk",etat(6.37.nil).etat(15.25.nil).etat(35.34.1.nil).nil>,etat(13.nil
   )) ->;
ddl(<"evtrunk",etat(18.nil).etat(15.25.nil).etat(35.34.1.nil).nil>,etat(13.nil))
    ->;
ddl(<"evtrunk",etat(27.nil).etat(15.25.nil).etat(35.34.1.nil).nil>,etat(13.nil))
    ->;
ddl(<"evtrunk",etat(33.nil).etat(15.25.nil).etat(39.30.29.28.20.19.17.14.12.nil)
   .nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(33.nil).etat(15.25.nil).etat(18.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(33.nil).etat(15.25.nil).etat(6.37.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(33.nil).etat(15.25.nil).etat(27.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(15.25.nil).etat(33.nil)
   .nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(33.nil).etat(15.25.nil).etat(35.34.1.nil.nil).nil>,etat(13.
   nil)) ->;
ddl(<"evtrunk",etat(33.nil).etat(15.25.nil).etat(13.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(18.nil).etat(15.25.nil).etat(33.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(35.34.1.nil).etat(15.25.nil).etat(33.nil).nil>,etat(13.nil))
    ->;
ddl(<"evtrunk",etat(6.37.nil).etat(15.25.nil).etat(33.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(27.nil).etat(15.25.nil).etat(33.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(13.nil).etat(15.25.nil).etat(33.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(33.nil).etat(15.25.nil).etat(33.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(13.nil).etat(15.25.nil).etat(39.30.29.28.20.19.17.14.12.nil)
   .nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(13.nil).etat(15.25.nil).etat(18.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(13.nil).etat(15.25.nil).etat(6.37.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(13.nil).etat(15.25.nil).etat(27.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(15.25.nil).etat(13.nil)
   .nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(13.nil).etat(15.25.nil).etat(35.34.1.nil.nil).nil>,etat(13.
   nil)) ->;
ddl(<"evtrunk",etat(13.nil).etat(15.25.nil).etat(13.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(18.nil).etat(15.25.nil).etat(13.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(35.34.1.nil).etat(15.25.nil).etat(13.nil).nil>,etat(13.nil))
    ->;
ddl(<"evtrunk",etat(6.37.nil).etat(15.25.nil).etat(13.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(27.nil).etat(15.25.nil).etat(13.nil).nil>,etat(13.nil)) ->;
ddl(<"trunk",etat(38.11.nil).etat(33.nil).nil>,etat(6.37.nil)) ->;
```

```
ddl(<"trunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(33.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(33.nil).etat(6.37.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(27.nil).etat(33.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(18.nil).etat(33.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(33.nil).etat(18.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(33.nil).etat(35.34.1.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(33.nil).etat(27.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(33.nil).etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(6.37.nil).etat(33.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(13.nil).etat(33.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(35.34.1.nil).etat(33.nil)>,etat(27.nil)) ->;
ddl(<"trunk",etat(33.nil).etat(13.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(33.nil).etat(33.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(38.11.nil).etat(13.nil).nil>,etat(6.37.nil)) ->;
ddl(<"trunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(13.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(13.nil).etat(6.37.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(27.nil).etat(13.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(18.nil).etat(13.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(13.nil).etat(18.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(13.nil).etat(35.34.1.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(13.nil).etat(27.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(13.nil).etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(6.37.nil).etat(13.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(13.nil).etat(13.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(35.34.1.nil).etat(13.nil)>,etat(27.nil)) ->;
ddl(<"trunk",etat(38.11.nil).etat(35.34.1.nil).nil>,etat(6.37.nil)) ->;
ddl(<"trunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(35.34.1.nil).nil>,etat(27
   .nil)) ->;
ddl(<"trunk",etat(35.34.1.nil).etat(6.37.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(27.nil).etat(35.34.1.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(18.nil).etat(35.34.1.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(35.34.1.nil).etat(18.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(35.34.1.nil).etat(35.34.1.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(35.34.1.nil).etat(27.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(35.34.1.nil).etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(27
   .nil)) ->;
ddl(<"trunk",etat(6.37.nil).etat(35.34.1.nil).nil>,etat(27.nil)) ->;
ddl(<"neck",etat(13.nil).nil>,etat(33.nil)) ->;
ddl(<"evtrunk",etat(18.nil).etat(15.25.nil).etat(39.30.29.28.20.19.17.14.12.nil)
   .nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(18.nil).etat(15.25.nil).etat(18.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(18.nil).etat(15.25.nil).etat(6.37.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(18.nil).etat(15.25.nil).etat(27.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(15.25.nil).etat(39.30.
   29.28.20.19.17.14.12.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(15.25.nil).etat(18.nil)
   .nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(15.25.nil).etat(6.37.
   nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(15.25.nil).etat(27.nil)
   .nil>,etat(13.nil)) ->;
```

```
ddl(<"evtrunk",etat(6.37.nil).etat(15.25.nil).etat(39.30.29.28.20.19.17.14.12.
   nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(6.37.nil).etat(15.25.nil).etat(18.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(6.37.nil).etat(15.25.nil).etat(6.37.nil).nil>,etat(13.nil))
    ->;
ddl(<"evtrunk",etat(6.37.nil).etat(15.25.nil).etat(27.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(27.nil).etat(15.25.nil).etat(39.30.29.28.20.19.17.14.12.nil)
   .nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(27.nil).etat(15.25.nil).etat(18.nil).nil>,etat(13.nil)) ->;
ddl(<"evtrunk",etat(27.nil).etat(15.25.nil).etat(6.37.nil).nil>,etat(13.nil)) ->
   ;
ddl(<"evtrunk",etat(27.nil).etat(15.25.nil).etat(27.nil).nil>,etat(13.nil)) ->;
ddl(<"trunk",etat(38.11.nil).etat(18.nil).nil>,etat(6.37.nil)) ->;
ddl(<"trunk",etat(38.11.nil).etat(6.37.nil).nil>,etat(6.37.nil)) ->;
ddl(<"trunk",etat(38.11.nil).etat(27.nil).nil>,etat(6.37.nil)) ->;
ddl(<"trunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(18.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(6.37.nil).nil>,etat(27.
   nil)) ->;
ddl(<"trunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(27.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(18.nil).etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(18.nil).etat(18.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(18.nil).etat(6.37.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(18.nil).etat(27.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(6.37.nil).etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(27.
   nil)) ->;
ddl(<"trunk",etat(6.37.nil).etat(18.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(6.37.nil).etat(6.37.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(6.37.nil).etat(27.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(27.nil).etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(27.nil)
   ) ->;
ddl(<"trunk",etat(27.nil).etat(18.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(27.nil).etat(6.37.nil).nil>,etat(27.nil)) ->;
ddl(<"trunk",etat(27.nil).etat(27.nil).nil>,etat(27.nil)) ->;
ddl(<"leg",etat(22.nil).nil>,etat(15.25.nil)) ->;
ddl(<"leg",etat(15.25.nil).nil>,etat(15.25.nil)) ->;
ddl(<"arm",etat(18.nil).nil>,etat(18.nil)) ->;
ddl(<"arm",etat(6.37.nil).nil>,etat(18.nil)) ->;
ddl(<"arm",etat(27.nil).nil>,etat(18.nil)) ->;
ddl(<"inflimbs",etat(26.24.23.16.nil).etat(22.nil).nil>,etat(22.nil)) ->;
ddl(<"inflimbs",etat(26.24.23.16.nil).etat(15.25.nil).nil>,etat(22.nil)) ->;
ddl(<"inflimbs",etat(22.nil).etat(26.24.23.16.nil).nil>,etat(22.nil)) ->;
ddl(<"inflimbs",etat(22.nil).etat(22.nil).nil>,etat(22.nil)) ->;
ddl(<"inflimbs",etat(22.nil).etat(15.25.nil).nil>,etat(22.nil)) ->;
ddl(<"inflimbs",etat(15.25.nil).etat(26.24.23.16.nil).nil>,etat(22.nil)) ->;
ddl(<"inflimbs",etat(15.25.nil).etat(22.nil).nil>,etat(22.nil)) ->;
ddl(<"inflimbs",etat(15.25.nil).etat(15.25.nil).nil>,etat(22.nil)) ->;
ddl(<"neck",etat(33.nil).nil>,etat(33.nil)) ->;
ddl(<"neck",etat(6.37.nil).nil>,etat(33.nil)) ->;
ddl(<"neck",etat(35.34.1.nil).nil>,etat(33.nil)) ->;
ddl(<"inflimbs",etat(26.24.23.16.nil).etat(26.24.23.16.nil).nil>,etat(22.nil))
    ->;
```

```
dd1(<"arm",etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(18.nil)) ->;
dd1(<"leg",etat(26.24.23.16.nil).nil>,etat(15.25.nil)) ->;
dd1(<"trunk",etat(38.11.nil).etat(39.30.29.28.20.19.17.14.12.nil).nil>,etat(6.37
   .nil)) ->;
dd1(<"trunk",etat(39.30.29.28.20.19.17.14.12.nil).etat(39.30.29.28.20.19.17.14.
   12.nil).nil>,etat(27.nil)) ->;
dd1(<"limbs",nil>,etat(39.30.29.28.20.19.17.14.12.nil)) ->;
dd1(<"wings",nil>,etat(38.11.nil)) ->;
dd1(<"body",nil>,etat(35.34.1.nil)) ->;
dd1(<"foot",nil>,etat(26.24.23.16.nil)) ->;
```

Avec le GTT G' on peut résoudre le problème de l'accessibilité en temps linéaire.
Prenons par exemple les deux termes suivants:



Ceux-ci sont représentés en mémoire interne par:

```
aa1(<"trunk",etat(1.1).etat(1.2).nil>,etat(1.0))->;
aa1(<"limbs",nil>,etat(1.2))->;
aa1(<"limbs",nil>,etat(1.1))->;


aa3(<"evtrunk",etat(2.1).etat(2.2).etat(2.3).nil>,etat(2.0))->;
aa3(<"arm",etat(2.5).nil>,etat(2.3))->;
aa3(<"limbs",nil>,etat(2.5))->;
aa3(<"leg",etat(2.4).nil>,etat(2.2))->;
aa3(<"foot",nil>,etat(2.4))->;
aa3(<"limbs",nil>,etat(2.1))->;
```

Après marquage, les automates aa1 et aa3 deviennent:

```
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(1.0)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(1.2)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(1.1)) ->;
aa1(<"trunk",etat(1.1).etat(1.2).nil>,etat(1.0)) ->;
aa1(<"limbs",nil>,etat(1.2)) ->;
aa1(<"limbs",nil>,etat(1.1)) ->;

aa3(vt(etat(13.nil)),etat(2.0)) ->;
aa3(vt(etat(18.nil)),etat(2.3)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(2.5)) ->;
aa3(vt(etat(15.25.nil)),etat(2.2)) ->;
aa3(vt(etat(26.24.23.16.nil)),etat(2.4)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(2.1)) ->;
aa3(<"evtrunk",etat(2.1).etat(2.2).etat(2.3).nil>,etat(2.0)) ->;
aa3(<"arm",etat(2.5).nil>,etat(2.3)) ->;
aa3(<"limbs",nil>,etat(2.5)) ->;
aa3(<"leg",etat(2.4).nil>,etat(2.2)) ->;
aa3(<"foot",nil>,etat(2.4)) ->;
aa3(<"limbs",nil>,etat(2.1)) ->;
```

On voit que l'état 2.0 est marqué par l'état 13.nil, et l'état 1.0 est marqué par l'état 1.36.35.13.6.37.32.31.27.21.18.3.30.20.nil. Ces deux marques ont en commun l'état 13 qui est un état interface et qui correspond à la règle 10.
Comme on l'a vu, il est possible de demander à VALERIAN quelles sont les règles à appliquer pour transformer t en t'.
Voici ce que deviennent les automates aa1 et aa3.

```
aa1(vt(etat(1.13.33.36.6.35.nil)),etat(4.9)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(4.11)) ->;
aa1(<"limbs",nil>,etat(4.11)) ->;
aa1(<"wings",nil>,etat(4.10)) ->;
aa1(<"trunk",etat(4.10).etat(4.11).nil>,etat(4.9)) ->;
aa1(vt(etat(1.13.33.36.6.35.nil)),etat(4.6)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(4.8)) ->;
aa1(<"limbs",nil>,etat(4.8)) ->;
aa1(<"wings",nil>,etat(4.7)) ->;
aa1(<"trunk",etat(4.7).etat(4.8).nil>,etat(4.6)) ->;
aa1(vt(etat(1.13.33.36.6.35.nil)),etat(4.1)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(4.4)) ->;
aa1(vt(etat(25.22.9.8.nil)),etat(4.5)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(4.2)) ->;
aa1(<"limbs",nil>,etat(4.4)) ->;
aa1(<"foot",nil>,etat(4.5)) ->;
aa1(<"leg",etat(4.5).nil>,etat(4.3)) ->;
aa1(<"limbs",nil>,etat(4.2)) ->;
aa1(<"evtrunk",etat(4.2).etat(4.3).etat(4.4).nil>,etat(4.1)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(1.0)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(1.2)) ->;
aa1(vt(etat(1.36.35.33.13.6.37.32.31.27.21.18.3.30.20.nil)),etat(1.1)) ->;
aa1(<"trunk",etat(1.1).etat(1.2).nil>,etat(1.0)) ->;
aa1(<"limbs",nil>,etat(1.2)) ->;
aa1(<"limbs",nil>,etat(1.1)) ->;

aa3(vt(etat(35.34.1.nil)),etat(3.19)) ->;
aa3(<"body",nil>,etat(3.19)) ->;
aa3(vt(etat(33.nil)),etat(3.17)) ->;
aa3(vt(etat(35.34.1.nil)),etat(3.18)) ->;
aa3(<"body",nil>,etat(3.18)) ->;
aa3(<"neck",etat(3.18).nil>,etat(3.17)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(3.16)) ->;
aa3(<"limbs",nil>,etat(3.16)) ->;
aa3(vt(etat(18.nil)),etat(3.14)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(3.15)) ->;
aa3(<"limbs",nil>,etat(3.15)) ->;
aa3(<"arm",etat(3.15).nil>,etat(3.14)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(3.13)) ->;
aa3(<"limbs",nil>,etat(3.13)) ->;
aa3(vt(etat(35.34.1.nil)),etat(3.12)) ->;
aa3(<"body",nil>,etat(3.12)) ->;
aa3(vt(etat(13.nil)),etat(3.7)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(3.10)) ->;
aa3(vt(etat(15.25.nil)),etat(3.9)) ->;
aa3(vt(etat(26.24.23.16.nil)),etat(3.11)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(3.8)) ->;
aa3(<"limbs",nil>,etat(3.10)) ->;
aa3(<"foot",nil>,etat(3.11)) ->;
aa3(<"leg",etat(3.11).nil>,etat(3.9)) ->;
aa3(<"limbs",nil>,etat(3.8)) ->;
```

```
aa3(<"evtrunk",etat(3.8).etat(3.9).etat(3.10).nil>,etat(3.7)) ->;
aa3(vt(etat(6.37.nil)),etat(3.2)) ->;
aa3(vt(etat(27.nil)),etat(3.4)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(3.6)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(3.5)) ->;
aa3(vt(etat(38.11.nil)),etat(3.3)) ->;
aa3(<"limbs",nil>,etat(3.6)) ->;
aa3(<"limbs",nil>,etat(3.5)) ->;
aa3(<"trunk",etat(3.5).etat(3.6).nil>,etat(3.4)) ->;
aa3(<"wings",nil>,etat(3.3)) ->;
aa3(<"trunk",etat(3.3).etat(3.4).nil>,etat(3.2)) ->;
aa3(vt(etat(35.34.1.nil)),etat(3.1)) ->;
aa3(<"body",nil>,etat(3.1)) ->;
aa3(vt(etat(13.nil)),etat(2.0)) ->;
aa3(vt(etat(18.nil)),etat(2.3)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(2.5)) ->;
aa3(vt(etat(15.25.nil)),etat(2.2)) ->;
aa3(vt(etat(26.24.23.16.nil)),etat(2.4)) ->;
aa3(vt(etat(39.30.29.28.20.19.17.14.12.nil)),etat(2.1)) ->;
aa3(<"evtrunk",etat(2.1).etat(2.2).etat(2.3).nil>,etat(2.0)) ->;
aa3(<"arm",etat(2.5).nil>,etat(2.3)) ->;
aa3(<"limbs",nil>,etat(2.5)) ->;
aa3(<"leg",etat(2.4).nil>,etat(2.2)) ->;
aa3(<"foot",nil>,etat(2.4)) ->;
aa3(<"limbs",nil>,etat(2.1)) ->;
```

On obtient finalement cet ensemble de faits:

```
rr(1.nil,10,100,1.nil)->;
rr(1.1.nil,9,150,1.1.nil)->;
rr(1.1.nil,12,50,1.1.nil)->;
rr(1.1.1.nil,1,25,2.1.1.nil)->;
rr(1.1.1.1.nil,5,37,1.2.1.1.nil)->;
rr(1.1.1.1.nil,4,13,1.2.1.1.nil)->;
```

Le premier argument du prédicat *rr* permet de se repérer dans l'arborescence développée dynamiquement.
Le deuxième argument correspond au numéro de la règle à appliquer.
Le troisième permet de placer les règles dans l'ordre de leur utilisation.
Le dernier permet de voir si la règle choisie correspondait au nième choix possible.

Exemple:
Prenons le fait rr(1.1.nil,1,25,2.1.1.nil). Dans ce cas 2.1.1.nil signifie que la règle 1 était le deuxième choix possible. De plus, on voit que les deux règles précédemment choisies correspondaient au premier choix possible.
Finalement le plus court chemin est inscrit dans le fait *chemin'*.

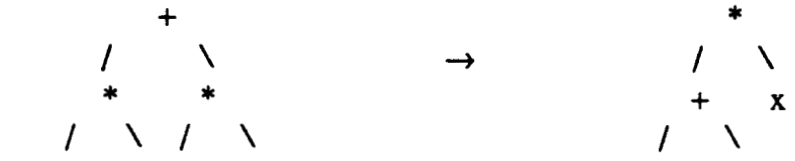
```
chemin'(1,<4,13>.<1,25>.<5,37>.<12,50>.<10,100>.<9,150>.nil)->;
```

Donc les règles à appliquer sont:

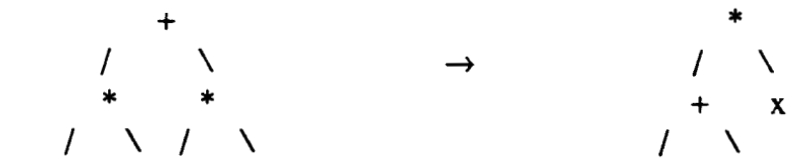
4, 1, 5, 12, 10, 9.

2- Exemple de système de réécriture séparé.

Soit le système R3 suivant:
Règle 1:



Règle 2:



Les symboles H et M étant des noms de sortes.
On obtient en mémoire interne les faits suivants:

```
ss'(2,<"+",<"*",<"H",nil>.<"x",nil>.nil>.<"*",<"c",nil>.<"x",nil>.nil>.nil>,<"*"
   ,<"+",<"H",nil>.<"c",nil>.nil>.<"x",nil>.nil>,12) ->;
ss'(1,<"+",<"*",<"H",nil>.<"x",nil>.nil>.<"*",<"M",nil>.<"x",nil>.nil>.nil>,<"*"
   ,<"+",<"H",nil>.<"M",nil>.nil>.<"x",nil>.nil>,1) ->;
```

Les états interface étant:

```
inter(etat(1).etat(12).nil)->;
```

Les automates reconnaissant les sortes M et H sont représentés en mémoire interne par:

```
tt("M",<"c",nil>,etat("2.0")) ->;
tt("M",<"x",nil>,etat("2.1")) ->;
tt("M",<"*",etat("2.0").etat("2.1").nil>,etat("2.2")) ->;
tt("M",<"*",etat("2.2").etat("2.1").nil>,etat("2.2")) ->;
tt("H",<"c",nil>,etat("1.0")) ->;
tt("H",<"x",nil>,etat("1.1")) ->;
tt("H",<"+",etat("2.2").etat("1.0").nil>,etat("1.2")) ->;
tt("H",<"*",etat("1.2").etat("1.1").nil>,etat("1.3")) ->;
tt("H",<"*",etat("1.3").etat("1.1").nil>,etat("1.3")) ->;
tt("H",<"+",etat("1.3").etat("1.0").nil>,etat("1.2")) ->;
tt("H",<"*",etat("1.2").etat("1.1").nil>,etat("1.2")) ->;
tt("H",<"*",etat("1.3").etat("1.1").nil>,etat("1.2")) ->;
tt("H",<"*",etat("1.0").etat("1.1").nil>,etat("1.2")) ->;
tt("H",<"*",etat("2.2").etat("1.1").nil>,etat("1.2")) ->;
```

Finalement le GTT G associé à ce système après suppression des ε-transitions possède les automates gg et dd suivants:

```
gg(<"*",etat("2.2").etat("1.1").nil>,etat(14)) ->;
gg(<"*",etat("1.0").etat("1.1").nil>,etat(14)) ->;
gg(<"*",etat("1.3").etat("1.1").nil>,etat(14)) ->;
gg(<"*",etat("1.2").etat("1.1").nil>,etat(14)) ->;
gg(<"+",etat("1.3").etat("1.0").nil>,etat(14)) ->;
gg(<"+",etat("2.2").etat("1.0").nil>,etat(14)) ->;
gg(<"+",etat(13).etat(16).nil>,etat(12)) ->;
gg(<"*",etat(17).etat(18).nil>,etat(16)) ->;
gg(<"x",nil>,etat(18)) ->;
gg(<"c",nil>,etat(17)) ->;
gg(<"*",etat(14).etat(15).nil>,etat(13)) ->;
gg(<"x",nil>,etat(15)) ->;
gg(<"*",etat("2.2").etat("1.1").nil>,etat("1.2")) ->;
gg(<"*",etat("1.0").etat("1.1").nil>,etat("1.2")) ->;
gg(<"*",etat("1.3").etat("1.1").nil>,etat("1.2")) ->;
gg(<"*",etat("1.2").etat("1.1").nil>,etat("1.2")) ->;
gg(<"+",etat("1.3").etat("1.0").nil>,etat("1.2")) ->;
gg(<"*",etat("1.3").etat("1.1").nil>,etat("1.3")) ->;
gg(<"*",etat("1.2").etat("1.1").nil>,etat("1.3")) ->;
gg(<"+",etat("2.2").etat("1.0").nil>,etat("1.2")) ->;
gg(<"x",nil>,etat("1.1")) ->;
gg(<"c",nil>,etat("1.0")) ->;
gg(<"*",etat("2.2").etat("2.1").nil>,etat("2.2")) ->;
gg(<"*",etat("2.0").etat("2.1").nil>,etat("2.2")) ->;
gg(<"x",nil>,etat("2.1")) ->;
gg(<"c",nil>,etat("2.0")) ->;
gg(<"*",etat("2.2").etat("1.1").nil>,etat(3)) ->;
gg(<"*",etat("1.0").etat("1.1").nil>,etat(3)) ->;
gg(<"*",etat("1.3").etat("1.1").nil>,etat(3)) ->;
gg(<"*",etat("1.2").etat("1.1").nil>,etat(3)) ->;
gg(<"+",etat("1.3").etat("1.0").nil>,etat(3)) ->;
gg(<"+",etat("2.2").etat("1.0").nil>,etat(3)) ->;
gg(<"*",etat("2.2").etat("2.1").nil>,etat(6)) ->;
gg(<"*",etat("2.0").etat("2.1").nil>,etat(6)) ->;
gg(<"+",etat(2).etat(5).nil>,etat(1)) ->;
gg(<"*",etat(6).etat(7).nil>,etat(5)) ->;
gg(<"x",nil>,etat(7)) ->;
gg(<"*",etat(3).etat(4).nil>,etat(2)) ->;
gg(<"x",nil>,etat(4)) ->;
gg(<"+",etat(13).etat(16).nil>,etat(2)) ->;
gg(<"+",etat(13).etat(16).nil>,etat(3)) ->;
gg(<"+",etat(13).etat(16).nil>,etat("1.3")) ->;
gg(<"+",etat(13).etat(16).nil>,etat("1.2")) ->;
gg(<"+",etat(13).etat(16).nil>,etat(13)) ->;
gg(<"+",etat(13).etat(16).nil>,etat(14)) ->;
gg(<"+",etat(2).etat(5).nil>,etat(2)) ->;
gg(<"+",etat(2).etat(5).nil>,etat(3)) ->;
gg(<"+",etat(2).etat(5).nil>,etat("1.3")) ->;
gg(<"+",etat(2).etat(5).nil>,etat("1.2")) ->;
gg(<"+",etat(2).etat(5).nil>,etat(13)) ->;
gg(<"+",etat(2).etat(5).nil>,etat(14)) ->;
```

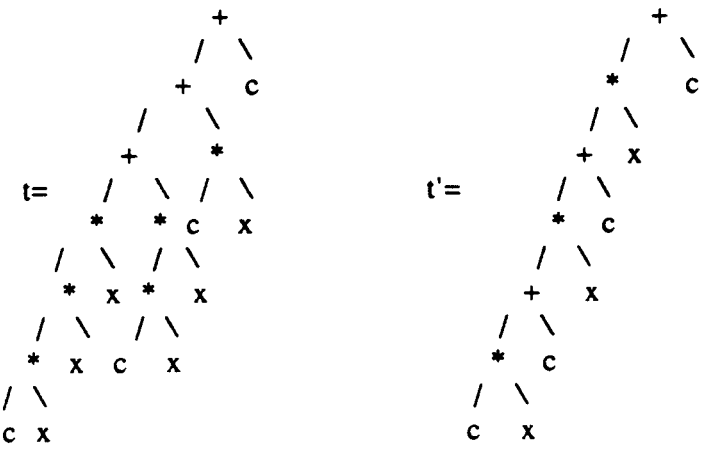
```
dd(etat(1),<"*",etat(12).etat(11).nil>) ->;
dd(etat(20),<"*",etat(1."1.0").etat(1."1.1").nil>) ->;
dd(etat(1."1.2"),<"*",etat(1."1.0").etat(1."1.1").nil>) ->;
dd(etat(1."2.2"),<"*",etat(1."2.0").etat(1."2.1").nil>) ->;
dd(etat(9),<"*",etat(1."1.0").etat(1."1.1").nil>) ->;
dd(etat(10),<"*",etat(1."2.0").etat(1."2.1").nil>) ->;
dd(etat(20),<"*",etat(1."2.2").etat(1."1.1").nil>) ->;
dd(etat(20),<"*",etat(1."1.2").etat(1."1.1").nil>) ->;
dd(etat(20),<"+",etat(1."2.2").etat(1."1.0").nil>) ->;
dd(etat(19),<"+",etat(20).etat(21).nil>) ->;
dd(etat(1."1.2"),<"*",etat(1."2.2").etat(1."1.1").nil>) ->;
dd(etat(1."1.2"),<"*",etat(1."1.2").etat(1."1.1").nil>) ->;
dd(etat(1."1.3"),<"*",etat(1."1.2").etat(1."1.1").nil>) ->;
dd(etat(1."1.2"),<"+",etat(1."2.2").etat(1."1.0").nil>) ->;
dd(etat(1."2.2"),<"*",etat(1."2.2").etat(1."2.1").nil>) ->;
dd(etat(9),<"*",etat(1."2.2").etat(1."1.1").nil>) ->;
dd(etat(9),<"*",etat(1."1.2").etat(1."1.1").nil>) ->;
dd(etat(9),<"+",etat(1."2.2").etat(1."1.0").nil>) ->;
dd(etat(10),<"*",etat(1."2.2").etat(1."2.1").nil>) ->;
dd(etat(8),<"+",etat(9).etat(10).nil>) ->;
dd(etat(20),<"*",etat(1."1.3").etat(1."1.1").nil>) ->;
dd(etat(20),<"+",etat(1."1.3").etat(1."1.0").nil>) ->;
dd(etat(12),<"*",etat(19).etat(22).nil>) ->;
dd(etat(1."1.2"),<"*",etat(1."1.3").etat(1."1.1").nil>) ->;
dd(etat(1."1.2"),<"+",etat(1."1.3").etat(1."1.0").nil>) ->;
dd(etat(1."1.3"),<"*",etat(1."1.3").etat(1."1.1").nil>) ->;
dd(etat(9),<"*",etat(1."1.3").etat(1."1.1").nil>) ->;
dd(etat(9),<"+",etat(1."1.3").etat(1."1.0").nil>) ->;
dd(etat(1),<"*",etat(8).etat(11).nil>) ->;
dd(etat(22),<"x",nil>) ->;
dd(etat(21),<"c",nil>) ->;
dd(etat(1."1.1"),<"x",nil>) ->;
dd(etat(1."1.0"),<"c",nil>) ->;
dd(etat(1."2.1"),<"x",nil>) ->;
dd(etat(1."2.0"),<"c",nil>) ->;
dd(etat(11),<"x",nil>) ->;
dd(etat(1),<"*",etat(1).etat(11).nil>) ->;
```

Comme dans l'exemple précédent on peut déterminiser ce GTT. On obtient le GTT G' suivant:

```
ggl(<"+",etat(2.14."1.2"."1.3".3.13.nil).etat(17."1.0"."2.0".nil).nil>,etat(14.
   "1.2".3.nil)) ->;
ggl(<"+",etat(2.14."1.2"."1.3".3.13.nil).etat("2.2".6.14."1.2".3.16.nil).nil>,
   etat(12.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"+",etat(2.14."1.2"."1.3".3.13.nil).etat(2.14."1.2"."1.3".3.13.5."2.2".6.
   nil).nil>,etat(1.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"+",etat(12.2.3."1.3"."1.2".13.14.nil).etat(17."1.0"."2.0".nil).nil>,etat(
   14."1.2".3.nil)) ->;
ggl(<"+",etat(12.2.3."1.3"."1.2".13.14.nil).etat("2.2".6.14."1.2".3.16.nil).nil>
   ,etat(12.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"+",etat(12.2.3."1.3"."1.2".13.14.nil).etat(2.14."1.2"."1.3".3.13.5."2.2".6
   .nil).nil>,etat(1.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"+",etat(1.2.3."1.3"."1.2".13.14.nil).etat(17."1.0"."2.0".nil).nil>,etat(14
   ."1.2".3.nil)) ->;
ggl(<"+",etat(1.2.3."1.3"."1.2".13.14.nil).etat("2.2".6.14."1.2".3.16.nil).nil>,
   etat(12.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"+",etat(1.2.3."1.3"."1.2".13.14.nil).etat(2.14."1.2"."1.3".3.13.5."2.2".6.
   nil).nil>,etat(1.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"*",etat(2.14."1.2"."1.3".3.13.nil).etat(18.15."1.1"."2.1".7.4.nil).nil>,
   etat(2.14."1.2"."1.3".3.13.nil)) ->;
ggl(<"*",etat(12.2.3."1.3"."1.2".13.14.nil).etat(18.15."1.1"."2.1".7.4.nil).nil>
   ,etat(2.14."1.2"."1.3".3.13.nil)) ->;
ggl(<"*",etat(1.2.3."1.3"."1.2".13.14.nil).etat(18.15."1.1"."2.1".7.4.nil).nil>,
   etat(2.14."1.2"."1.3".3.13.nil)) ->;
ggl(<"*",etat(14."1.2".3.nil).etat(18.15."1.1"."2.1".7.4.nil).nil>,etat(2.14.
   "1.2"."1.3".3.13.nil)) ->;
ggl(<"*",etat(2.14."1.2"."1.3".3.13.5."2.2".6.nil).etat(18.15."1.1"."2.1".7.4.
   nil).nil>,etat(2.14."1.2"."1.3".3.13.5."2.2".6.nil)) ->;
ggl(<"+",etat(2.14."1.2"."1.3".3.13.5."2.2".6.nil).etat(17."1.0"."2.0".nil).nil>
   ,etat(14."1.2".3.nil)) ->;
ggl(<"+",etat(2.14."1.2"."1.3".3.13.5."2.2".6.nil).etat("2.2".6.14."1.2".3.16.
   nil).nil>,etat(12.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"+",etat(2.14."1.2"."1.3".3.13.5."2.2".6.nil).etat(2.14."1.2"."1.3".3.13.5.
   "2.2".6.nil).nil>,etat(1.2.3."1.3"."1.2".13.14.nil)) ->;
ggl(<"+",etat("2.2".6.14."1.2".3.16.nil).etat(17."1.0"."2.0".nil).nil>,etat(14.
   "1.2".3.nil)) ->;
ggl(<"*",etat("2.2".6.14."1.2".3.16.nil).etat(18.15."1.1"."2.1".7.4.nil).nil>,
   etat(2.14."1.2"."1.3".3.13.5."2.2".6.nil)) ->;
ggl(<"*",etat(17."1.0"."2.0".nil).etat(18.15."1.1"."2.1".7.4.nil).nil>,etat(
   "2.2".6.14."1.2".3.16.nil)) ->;
ggl(<"x",nil>,etat(18.15."1.1"."2.1".7.4.nil)) ->;
ggl(<"c",nil>,etat(17."1.0"."2.0".nil)) ->;
```

```
ddl(<"*",etat(1.20.(1."1.2").(1."1.3").9.nil).etat(22.(1."1.1").(1."2.1").11.nil
   ).nil>,etat(1.20.(1."1.2").(1."1.3").9.nil)) ->;
ddl(<"*",etat(12.nil).etat(22.(1."1.1").(1."2.1").11.nil).nil>,etat(1.nil)) ->;
ddl(<"+",etat(1.20.(1."1.2").(1."1.3").9.nil).etat(21.(1."1.0").(1."2.0").nil).
   nil>,etat(19.20.(1."1.2").9.nil)) ->;
ddl(<"+",etat(1.20.(1."1.2").(1."1.3").9.nil).etat((1."2.2").10.20.(1."1.2").9.
   nil).nil>,etat(8.nil)) ->;
ddl(<"+",etat(1.20.(1."1.2").(1."1.3").9.nil).etat(20.(1."1.2").(1."1.3").9.(1.
   "2.2").10.nil).nil>,etat(8.nil)) ->;
ddl(<"+",etat(20.(1."1.2").(1."1.3").9.12.nil).etat(21.(1."1.0").(1."2.0").nil).
   nil>,etat(19.20.(1."1.2").9.nil)) ->;
ddl(<"+",etat(20.(1."1.2").(1."1.3").9.12.nil).etat((1."2.2").10.20.(1."1.2").9.
   nil).nil>,etat(8.nil)) ->;
ddl(<"+",etat(20.(1."1.2").(1."1.3").9.12.nil).etat(20.(1."1.2").(1."1.3").9.(1.
   "2.2").10.nil).nil>,etat(8.nil)) ->;
ddl(<"*",etat(20.(1."1.2").(1."1.3").9.12.nil).etat(22.(1."1.1").(1."2.1").11.
   nil).nil>,etat(1.20.(1."1.2").(1."1.3").9.nil)) ->;
ddl(<"*",etat(1.nil).etat(22.(1."1.1").(1."2.1").11.nil).nil>,etat(1.nil)) ->;
ddl(<"*",etat(19.nil).etat(22.(1."1.1").(1."2.1").11.nil).nil>,etat(12.nil)) ->;
ddl(<"*",etat(19.20.(1."1.2").9.nil).etat(22.(1."1.1").(1."2.1").11.nil).nil>,
   etat(20.(1."1.2").(1."1.3").9.12.nil)) ->;
ddl(<"*",etat(8.nil).etat(22.(1."1.1").(1."2.1").11.nil).nil>,etat(1.nil)) ->;
ddl(<"*",etat(20.(1."1.2").(1."1.3").9.(1."2.2").10.nil).etat(22.(1."1.1").(1.
   "2.1").11.nil).nil>,etat(20.(1."1.2").(1."1.3").9.(1."2.2").10.nil)) ->;
ddl(<"+",etat((1."2.2").10.20.(1."1.2").9.nil).etat(20.(1."1.2").(1."1.3").9.(1.
   "2.2").10.nil).nil>,etat(8.nil)) ->;
ddl(<"+",etat(19.20.(1."1.2").9.nil).etat(21.(1."1.0").(1."2.0").nil).nil>,etat(
   19.nil)) ->;
ddl(<"+",etat(19.20.(1."1.2").9.nil).etat((1."2.2").10.20.(1."1.2").9.nil).nil>,
   etat(8.nil)) ->;
ddl(<"+",etat(19.20.(1."1.2").9.nil).etat(20.(1."1.2").(1."1.3").9.(1."2.2").10.
   nil).nil>,etat(8.nil)) ->;
ddl(<"+",etat(20.(1."1.2").(1."1.3").9.(1."2.2").10.nil).etat(21.(1."1.0").(1.
   "2.0").nil).nil>,etat(19.20.(1."1.2").9.nil)) ->;
ddl(<"+",etat(20.(1."1.2").(1."1.3").9.(1."2.2").10.nil).etat((1."2.2").10.20.(1
   ."1.2").9.nil).nil>,etat(8.nil)) ->;
ddl(<"+",etat(20.(1."1.2").(1."1.3").9.(1."2.2").10.nil).etat(20.(1."1.2").(1.
   "1.3").9.(1."2.2").10.nil).nil>,etat(8.nil)) ->;
ddl(<"+",etat((1."2.2").10.20.(1."1.2").9.nil).etat(21.(1."1.0").(1."2.0").nil).
   nil>,etat(19.20.(1."1.2").9.nil)) ->;
ddl(<"+",etat((1."2.2").10.20.(1."1.2").9.nil).etat((1."2.2").10.20.(1."1.2").9.
   nil).nil>,etat(8.nil)) ->;
ddl(<"*",etat((1."2.2").10.20.(1."1.2").9.nil).etat(22.(1."1.1").(1."2.1").11.
   nil).nil>,etat(20.(1."1.2").(1."1.3").9.(1."2.2").10.nil)) ->;
ddl(<"*",etat(21.(1."1.0").(1."2.0").nil).etat(22.(1."1.1").(1."2.1").11.nil).
   nil>,etat((1."2.2").10.20.(1."1.2").9.nil)) ->;
ddl(<"x",nil>,etat(22.(1."1.1").(1."2.1").11.nil)) ->;
ddl(<"c",nil>,etat(21.(1."1.0").(1."2.0").nil)) ->;
```

A partir du GTT G on peut, comme on l'a vu, répondre au problème de l'accessibilité en temps quadratique.
Prenons par exemple les deux termes suivants:



Ces deux termes sont repésentés en mémoire interne par les automates aa0 et aa2 suivants:

```
aa0(<"+",etat(1.1).etat(1.2).nil>,etat(1.0))->;
aa0(<"c",nil>,etat(1.2))->;
aa0(<"+",etat(1.3).etat(1.4).nil>,etat(1.1))->;
aa0(<"*",etat(1.17).etat(1.18).nil>,etat(1.4))->;
aa0(<"x",nil>,etat(1.18))->;
aa0(<"c",nil>,etat(1.17))->;
aa0(<"+",etat(1.5).etat(1.6).nil>,etat(1.3))->;
aa0(<"*",etat(1.13).etat(1.14).nil>,etat(1.6))->;
aa0(<"x",nil>,etat(1.14))->;
aa0(<"*",etat(1.15).etat(1.16).nil>,etat(1.13))->;
aa0(<"x",nil>,etat(1.16))->;
aa0(<"c",nil>,etat(1.15))->;
aa0(<"*",etat(1.7).etat(1.8).nil>,etat(1.5))->;
aa0(<"x",nil>,etat(1.8))->;
aa0(<"*",etat(1.9).etat(1.10).nil>,etat(1.7))->;
aa0(<"x",nil>,etat(1.10))->;
aa0(<"*",etat(1.11.etat(1.12).nil>,etat(1.9))->;
aa0(<"x",nil>,etat(1.12))->;
aa0(<"c",nil>,etat(1.11))->;
```

```
aa2(<"+",etat(2.1).etat(2.2).nil>,etat(2.0)) ->;
aa2(<"c",nil>,etat(2.2)) ->;
aa2(<"*",etat(2.3).etat(2.4).nil>,etat(2.1)) ->;
aa2(<"x",nil>,etat(2.4)) ->;
aa2(<"+",etat(2.5).etat(2.6).nil>,etat(2.3)) ->;
aa2(<"c",nil>,etat(2.6)) ->;
aa2(<"*",etat(2.7).etat(2.8).nil>,etat(2.5)) ->;
aa2(<"x",nil>,etat(2.8)) ->;
aa2(<"+",etat(2.9).etat(2.10).nil>,etat(2.7)) ->;
aa2(<"c",nil>,etat(2.10)) ->;
aa2(<"*",etat(2.11).etat(2.12).nil>,etat(2.9)) ->;
aa2(<"x",nil>,etat(2.12)) ->;
aa2(<"c",nil>,etat(2.11)) ->;
```

Pour effectuer le produit des automates aa0 et gg, on les numérote et on obtient ainsi les deux automates aa1 et bb1 suivants:

```
aa1(19,<"c",nil>,etat(1.11)) ->;
aa1(18,<"x",nil>,etat(1.12)) ->;
aa1(17,<"*",etat(1.11).etat(1.12).nil>,etat(1.9)) ->;
aa1(16,<"x",nil>,etat(1.10)) ->;
aa1(15,<"*",etat(1.9).etat(1.10).nil>,etat(1.7)) ->;
aa1(14,<"x",nil>,etat(1.8)) ->;
aa1(13,<"*",etat(1.7).etat(1.8).nil>,etat(1.5)) ->;
aa1(12,<"c",nil>,etat(1.15)) ->;
aa1(11,<"x",nil>,etat(1.16)) ->;
aa1(10,<"*",etat(1.15).etat(1.16).nil>,etat(1.13)) ->;
aa1(9,<"x",nil>,etat(1.14)) ->;
aa1(8,<"*",etat(1.13).etat(1.14).nil>,etat(1.6)) ->;
aa1(7,<"+",etat(1.5).etat(1.6).nil>,etat(1.3)) ->;
aa1(6,<"c",nil>,etat(1.17)) ->;
aa1(5,<"x",nil>,etat(1.18)) ->;
aa1(4,<"*",etat(1.17).etat(1.18).nil>,etat(1.4)) ->;
aa1(3,<"+",etat(1.3).etat(1.4).nil>,etat(1.1)) ->;
aa1(2,<"c",nil>,etat(1.2)) ->;
aa1(1,<"+",etat(1.1).etat(1.2).nil>,etat(1.0)) ->;

bb1(51,<"*",etat("2.2").etat("1.1").nil>,etat(14)) ->;
bb1(50,<"*",etat("1.0").etat("1.1").nil>,etat(14)) ->;
bb1(49,<"*",etat("1.3").etat("1.1").nil>,etat(14)) ->;
bb1(48,<"*",etat("1.2").etat("1.1").nil>,etat(14)) ->;
bb1(47,<"+",etat("1.3").etat("1.0").nil>,etat(14)) ->;
bb1(46,<"+",etat("2.2").etat("1.0").nil>,etat(14)) ->;
bb1(45,<"+",etat(13).etat(16).nil>,etat(12)) ->;
bb1(44,<"*",etat(17).etat(18).nil>,etat(16)) ->;
bb1(43,<"x",nil>,etat(18)) ->;
bb1(42,<"c",nil>,etat(17)) ->;
bb1(41,<"*",etat(14).etat(15).nil>,etat(13)) ->;
bb1(40,<"x",nil>,etat(15)) ->;
bb1(39,<"*",etat("2.2").etat("1.1").nil>,etat("1.2")) ->;
bb1(38,<"*",etat("1.0").etat("1.1").nil>,etat("1.2")) ->;
bb1(37,<"*",etat("1.3").etat("1.1").nil>,etat("1.2")) ->;
bb1(36,<"*",etat("1.2").etat("1.1").nil>,etat("1.2")) ->;
bb1(35,<"+",etat("1.3").etat("1.0").nil>,etat("1.2")) ->;
bb1(34,<"*",etat("1.3").etat("1.1").nil>,etat("1.3")) ->;
bb1(33,<"*",etat("1.2").etat("1.1").nil>,etat("1.3")) ->;
bb1(32,<"+",etat("2.2").etat("1.0").nil>,etat("1.2")) ->;
bb1(31,<"x",nil>,etat("1.1")) ->;
bb1(30,<"c",nil>,etat("1.0")) ->;
bb1(29,<"*",etat("2.2").etat("2.1").nil>,etat("2.2")) ->;
bb1(28,<"*",etat("2.0").etat("2.1").nil>,etat("2.2")) ->;
bb1(27,<"x",nil>,etat("2.1")) ->;
bb1(26,<"c",nil>,etat("2.0")) ->;
bb1(25,<"*",etat("2.2").etat("1.1").nil>,etat(3)) ->;
bb1(24,<"*",etat("1.0").etat("1.1").nil>,etat(3)) ->;
bb1(23,<"*",etat("1.3").etat("1.1").nil>,etat(3)) ->;
bb1(22,<"*",etat("1.2").etat("1.1").nil>,etat(3)) ->;
bb1(21,<"+",etat("1.3").etat("1.0").nil>,etat(3)) ->;
bb1(20,<"+",etat("2.2").etat("1.0").nil>,etat(3)) ->;
bb1(19,<"*",etat("2.2").etat("2.1").nil>,etat(6)) ->;
bb1(18,<"*",etat("2.0").etat("2.1").nil>,etat(6)) ->;
bb1(17,<"+",etat(2).etat(5).nil>,etat(1)) ->;
bb1(16,<"*",etat(6).etat(7).nil>,etat(5)) ->;
bb1(15,<"x",nil>,etat(7)) ->;
bb1(14,<"*",etat(3).etat(4).nil>,etat(2)) ->;
```

```
bb1(13,<"x",nil>,etat(4)) ->;
bb1(12,<"+",etat(13).etat(16).nil>,etat(2)) ->;
bb1(11,<"+",etat(13).etat(16).nil>,etat(3)) ->;
bb1(10,<"+",etat(13).etat(16).nil>,etat("1.3")) ->;
bb1(9,<"+",etat(13).etat(16).nil>,etat("1.2")) ->;
bb1(8,<"+",etat(13).etat(16).nil>,etat(13)) ->;
bb1(7,<"+",etat(13).etat(16).nil>,etat(14)) ->;
bb1(6,<"+",etat(2).etat(5).nil>,etat(2)) ->;
bb1(5,<"+",etat(2).etat(5).nil>,etat(3)) ->;
bb1(4,<"+",etat(2).etat(5).nil>,etat("1.3")) ->;
bb1(3,<"+",etat(2).etat(5).nil>,etat("1.2")) ->;
bb1(2,<"+",etat(2).etat(5).nil>,etat(13)) ->;
bb1(1,<"+",etat(2).etat(5).nil>,etat(14)) ->;
```

Enfin, voici quelques règles de l'automate produit obtenu. Celui-ci ayant un nombre important de règles, nous ne l'avons pas inscrit ici dans son intégralité.

```
na(<"+",etat(<1.1,"1.3">).etat(<1.2,"1.0">).nil>,etat(<1.0,3>)) ->;
na(<"+",etat(<1.1,"1.3">).etat(<1.2,"1.0">).nil>,etat(<1.0,"1.2">)) ->;
na(<"+",etat(<1.1,"1.3">).etat(<1.2,"1.0">).nil>,etat(<1.0,14>)) ->;
na(<"+",etat(<1.3,13>).etat(<1.4,16>).nil>,etat(<1.1,14>)) ->;
na(<"+",etat(<1.3,13>).etat(<1.4,16>).nil>,etat(<1.1,13>)) ->;
na(<"+",etat(<1.3,13>).etat(<1.4,16>).nil>,etat(<1.1,"1.2">)) ->;
na(<"+",etat(<1.3,13>).etat(<1.4,16>).nil>,etat(<1.1,"1.3">)) ->;
na(<"+",etat(<1.3,13>).etat(<1.4,16>).nil>,etat(<1.1,3>)) ->;
na(<"+",etat(<1.3,13>).etat(<1.4,16>).nil>,etat(<1.1,2>)) ->;
na(<"+",etat(<1.3,13>).etat(<1.4,16>).nil>,etat(<1.1,12>)) ->;
na(<"+",etat(<1.5,2>).etat(<1.6,5>).nil>,etat(<1.3,14>)) ->;
na(<"+",etat(<1.5,2>).etat(<1.6,5>).nil>,etat(<1.3,13>)) ->;
```

La deuxième étape s'effectue à peu près de la même façon. Nous ne la detaillerons pas ici, car les automates obtenus sont particulièrement gros.

A partir du GTT G' (G déterminisé), on peut résoudre également le problème de l'accessibilité, mais cette fois ce sera en temps linéaire. Reprenons les deux arbres t et t' précedents. Ils sont représentés par les automates aa1 et aa3 suivants:

```
aa1(<"+",etat(1.1).etat(1.2).nil>,etat(1.0))->;
aa1(<"c",nil>,etat(1.2))->;
aa1(<"+",etat(1.3).etat(1.4).nil>,etat(1.1))->;
aa1(<"*",etat(1.17).etat(1.18).nil>,etat(1.4))->;
aa1(<"x",nil>,etat(1.18))->;
aa1(<"c",nil>,etat(1.17))->;
aa1(<"+",etat(1.5).etat(1.6).nil>,etat(1.3))->;
aa1(<"*",etat(1.13).etat(1.14).nil>,etat(1.6))->;
aa1(<"x",nil>,etat(1.14))->;
aa1(<"*",etat(1.15).etat(1.16).nil>,etat(1.13))->;
aa1(<"x",nil>,etat(1.16))->;
aa1(<"c",nil>,etat(1.15))->;
aa1(<"*",etat(1.7).etat(1.8).nil>,etat(1.5))->;
aa1(<"x",nil>,etat(1.8))->;
aa1(<"*",etat(1.9).etat(1.10).nil>,etat(1.7))->;
aa1(<"x",nil>,etat(1.10))->;
```

```
aa1(<"*",etat(1.11.etat(1.12).nil>,etat(1.9))->;
aa1(<"x",nil>,etat(1.12))->;
aa1(<"c",nil>,etat(1.11))->;

aa3(<"+",etat(2.1).etat(2.2).nil>,etat(2.0))->;
aa3(<"c",nil>,etat(2.2))->;
aa3(<"*",etat(2.3).etat(2.4).nil>,etat(2.1))->;
aa3(<"x",nil>,etat(2.4))->;
aa3(<"+",etat(2.5).etat(2.6).nil>,etat(2.3))->;
aa3(<"c",nil>,etat(2.6))->;
aa3(<"*",etat(2.7).etat(2.8).nil>,etat(2.5))->;
aa3(<"x",nil>,etat(2.8))->;
aa3(<"+",etat(2.9).etat(2.10).nil>,etat(2.7))->;
aa3(<"c",nil>,etat(2.10))->;
aa3(<"*",etat(2.11).etat(2.12).nil>,etat(2.9))->;
aa3(<"x",nil>,etat(2.12))->;
aa3(<"c",nil>,etat(2.11))->;
```

Après marquage, ces deux automates prennent la forme suivante:

```
aa1(vt(etat(12.2.3."1.3"."1.2".13.14.nil)),etat(1.1)) ->;
aa1(vt(etat(1.2.3."1.3"."1.2".13.14.nil)),etat(1.3)) ->;
aa1(<"+",etat(1.1).etat(1.2).nil>,etat(1.0)) ->;
aa1(<"c",nil>,etat(1.2)) ->;
aa1(<"+",etat(1.3).etat(1.4).nil>,etat(1.1)) ->;
aa1(<"*",etat(1.17).etat(1.18).nil>,etat(1.4)) ->;
aa1(<"x",nil>,etat(1.18)) ->;
aa1(<"c",nil>,etat(1.17)) ->;
aa1(<"+",etat(1.5).etat(1.6).nil>,etat(1.3)) ->;
aa1(<"*",etat(1.13).etat(1.14).nil>,etat(1.6)) ->;
aa1(<"x",nil>,etat(1.14)) ->;
aa1(<"*",etat(1.15).etat(1.16).nil>,etat(1.13)) ->;
aa1(<"x",nil>,etat(1.16)) ->;
aa1(<"c",nil>,etat(1.15)) ->;
aa1(<"*",etat(1.7).etat(1.8).nil>,etat(1.5)) ->;
aa1(<"x",nil>,etat(1.8)) ->;
aa1(<"*",etat(1.9).etat(1.10).nil>,etat(1.7)) ->;
aa1(<"x",nil>,etat(1.10)) ->;
aa1(<"*",etat(1.11).etat(1.12).nil>,etat(1.9)) ->;
aa1(<"x",nil>,etat(1.12)) ->;
aa1(<"c",nil>,etat(1.11)) ->;


aa3(vt(etat(19.20.(1."1.2").9.nil)),etat(2.0)) ->;
aa3(vt(etat(21.(1."1.0").(1."2.0").nil)),etat(2.2)) ->;
aa3(vt(etat(20.(1."1.2").(1."1.3").9.12.nil)),etat(2.1)) ->;
aa3(vt(etat(22.(1."1.1").(1."2.1").11.nil)),etat(2.4)) ->;
aa3(vt(etat(19.20.(1."1.2").9.nil)),etat(2.3)) ->;
aa3(vt(etat(21.(1."1.0").(1."2.0").nil)),etat(2.6)) ->;
aa3(vt(etat(20.(1."1.2").(1."1.3").9.12.nil)),etat(2.5)) ->;
aa3(vt(etat(22.(1."1.1").(1."2.1").11.nil)),etat(2.8)) ->;
aa3(vt(etat(19.20.(1."1.2").9.nil)),etat(2.7)) ->;
aa3(vt(etat(21.(1."1.0").(1."2.0").nil)),etat(2.10)) ->;
aa3(vt(etat((1."2.2").10.20.(1."1.2").9.nil)),etat(2.9)) ->;
aa3(vt(etat(22.(1."1.1").(1."2.1").11.nil)),etat(2.12)) ->;
aa3(vt(etat(21.(1."1.0").(1."2.0").nil)),etat(2.11)) ->;
aa3(<"+",etat(2.1).etat(2.2).nil>,etat(2.0)) ->;
```

```
aa3(<"c",nil>,etat(2.2)) ->;
aa3(<"*",etat(2.3).etat(2.4).nil>,etat(2.1)) ->;
aa3(<"x",nil>,etat(2.4)) ->;
aa3(<"+",etat(2.5).etat(2.6).nil>,etat(2.3)) ->;
aa3(<"c",nil>,etat(2.6)) ->;
aa3(<"*",etat(2.7).etat(2.8).nil>,etat(2.5)) ->;
aa3(<"x",nil>,etat(2.8)) ->;
aa3(<"+",etat(2.9).etat(2.10).nil>,etat(2.7)) ->;
aa3(<"c",nil>,etat(2.10)) ->;
aa3(<"*",etat(2.11).etat(2.12).nil>,etat(2.9)) ->;
aa3(<"x",nil>,etat(2.12)) ->;
aa3(<"c",nil>,etat(2.11)) ->;
```

L'état 1.1 est marqué par l'état 12.2.3."1.3"."1.2".13.14.nil, et l'état 2.1. est marqué par l'état 20.(2."1.2").(1."1.3").9.12.nil. On peut remarquer que l'état interface commun est l'état 12 qui correspond à la règle 2 du système de réécriture. Finalement, la réponse au problème est positive.

3- Tableau de comparaison de temps d'exécution suivant les deux méthodes étudiées.

Nous considérons ici les quatres systèmes de réécriture suivants.
Le système R1 possède les 9 règles suivantes:

Règle 1:

```
      c   → a1
    /  \
    b   b1
    |
    b1
```

Règle 2:

```
b  → b1
|
b1
```

Règle 3:

```
b1  → b
      |
     b1
```

Règle 4:

```
a1  → a
      |
     a1
```

Règle 5:

```
a   → a1
|
a1
```

Règle 6:

```
a1  → a
      |
     a2
```

Règle 7:

```
a   → a3
|
a
|
a2
```

Règle 8:

```
a2  → a
      |
     a2
```

Règle 9:

```
a1    → b
        |
       b1
```

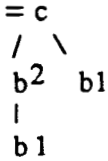
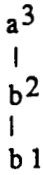
On reprend les systèmes R2 et R3 vus précédemment. Et enfin le système R4 dont les règles sont les suivantes:

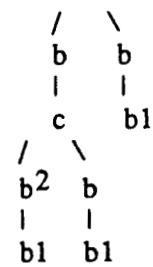
Règle 1:

Règle 2:

Pour chacun des ces systèmes nous avons poser les questions suivantes:

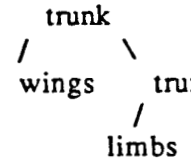
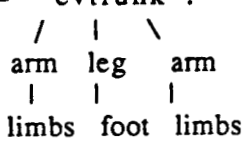
1/Avec R1:

Peut-on passer du terme t1 =  au terme t1'=  ?

Peut-on passer du terme t1=  au terme t2'= t1'?

2/Avec R2:

Peut-on passer du terme a1=limbs au terme a1'=body?

Peut-on passer du terme a2=  au terme a2'=  ?

3/Avec R3 et R4:

On a posé la question vue à la page 159.

Finalement on obtient le tableau suivant:

Les temps d'execution étant en millisecondes.

	R1		R2		R3	R4
	t1,t1'	t2,t2'	a1,a1'	a2,a2'	t , t'	t , t'
S'	21008	57936	3392	1520192	431152	287008
S'_{det}	240	368	496	560	656	554

BIBLIOGRAPHIE

[1] Aho A.V., Hopcroft J.E. et Ullman J.D., *The design and analysis of computer algorithms*, Addison-Wesley Publishing Company (1974)

[2] Bachmair L. and Dershowitz N. *Equational inference, canonical proofs and proof orderings* (1989).

[3] Bachmair L. and Ganzinger H. *On restrictions of ordered paramodulation with simplification.* Proc. 10th Int. Conf. Aut. Ded. LNCS vol 449, pp. 427-441 (M. Stickel, ed.) (1990).

[4] Brainers, "Tree-generating regular systems", *Information and control* (1969)

[5] Bréhier E. *Histoire de la philosophie XVIIième et XVIIIième siècles.* Quadrige/PUF PARIS 1983 Tome 2 pp338-354

[6] Comon, H. "Inductive Proofs by Specification Transformations". *Rewriting Technics and Applications*, Chapell Hill, North Carolina, April 1989, lecture Notes in Computer Science. (Dershowitz ed.)

[7] Comon, H. "Solving Inequations in Term Algebras". *Fifth annual IEEE symposium on Logic in Computer Science,* Philadelphia, PA, June 1990, IEEE Computer Society Press, pp 62-69

[8] Courcelle B. "Equivalences and transformations of regular systems. application to recursive schemes and grammars", *Theor. Comp. Sci.* 42 (1986), pp1-122.

[9] Courcelle, B. "On recognizable sets and tree automata", *Theor. Comp. Sci.*

[10] Dauchet, M., T. Heuillard, P. Lescanne & S. Tison "Decidability of the confluence of ground term tewriting systems", *2nd Symposium on Logic In Computer Science*, New-York, IEEE Computer Society Press (1987), pp 353-360.

[11] Dauchet, M. & Tison, S. "Decidability of confluence in ground term rewriting systems", *Fondations of Computation Theory ' 85*, Cottbus, Lecture Notes in Computer Science., 199 (1985), PP 80-84.

[12] Dauchet, M. & Deruyver, A." "VALERIAN": Compilation of Ground Term Rewriting Systems and Applications". *Rewriting Technics and Aplications*, Chapell Hill, North Carolina., April 1989, lecture Notes in Computer Science 355. (Dershowitz ed.)

[13] Dauchet, M. & Tison, S.
"The theory of ground rewrite systems is decidable",
Fifth annual IEEE symposium on Logic in Computer Science,
Philadelphia, PA, June 1990, IEEE Computer Society Press.

[14] Dershowitz N. *Termination* Journal of Symbolic Computation (1986).

[15] Dershowitz, N. & Jouannaud, J.P., "Rewrite Systems", *Handbook of Theoretical Computer Science*, J.V. Leeuwen editor, North-Holland, to appear. (1989)

[16] Deruyver, A., Gilleron, R., "Compilation of term rewriting systems" *CAAP'89*, Lec. Notes Comp. Sci. (Diaz ed.).

[17] Deruyver, A, "Emmy, a refutational theorem prover for first order logic with equations", Technical report 26/90 SUNY at Stony Brook (New York).

[18] Downey P.J., Sethi R., Tarjan R. E., "Variations on the Common Subexpression Problem". *Journal of ACM* , Vol 27 n°4 October 1980 pp. 758-771.

[19] Echahed R. *Sur l'intégration des langages algébrique et logique.* Thèse Grenoble 1990.

[20] Eilenberg s. & Wreight, J. "Automata in General Algebra", *Information and Control,* 11, pp 52-70 (1967).

[21] Engelfriet, J. "Bottom-up and Top-down tree transformations, a comparison", *Math. systems theory,* 9, 198-231 (1975).

[22] Fülöp,Z & Vàgvölgyi S., "Ground Term Rewriting rules for the Word Problem of Ground term Equations", submitted paper (1989).

[23] Galler B.A. & Fischer M.J. "An improved equivalence algorithm". *Comm. ACM.* 7, 5 (May 1964) pp. 301-303.

[24] Gallier, J. H., Raatz, S. & Snyder, W. "Theorem Proving using Rigid E-Unification: Equational Mating", *2nd Symposium on logic computer Science*, IEEE Computer Society Press (1987), pp 338-346.

[25] Gallier, J.H., Narendran, P., J.H., Raatz, S. Snyder, W. "Theorem Proving Using Equational Matings and Rigid E-Unification.", To appear.

[26] Gecseg F. & Steinby M., *Tree Automata* Akadémiai Kiado, Budapest (1984)

[27] Goguen J.A. & Tardo J.J., "An introduction to OBJ: A langage for rewriting and testing formal algebraic specifications", *Proc. of the specification of Reliable Software conference*, pp. 170-189, April 1979.

[28] Gorn S., "Explicit definitions and linguistic dominoes". In J. Hart, S. Takasu, ed., *Systems ans Computer Science*, pp. 77-115, University of Toronto Press, 1967.

[29] Huet.G & Oppen D.C. (1980) "Equations and rewrite rules: A survey", in R. V. Book, ed., New York: Academic Press. *Formal Language Theory: Perspectives and Open Problems*, pp. 349-405.

[30] Jouannaud, J.P. (1987). Editorial of *J. Symbolic computation*, 3, pp 2-3.

[31] Kozen, D. "Complexity of finitely presented algebras" *9th ACM Symp. on Theory of computing*, Boulder, Colorado, (1977), pp. 164-177.

[32] Mccharen John D., Overbeek Ross A. and Wos Laurence A. *Problems and experiments for and with automated Theorem-proving programs.* IEEE transactions on computers. Vol C-25 No 8 August 1976.

[33] Nelson G. & Oppen D. C. "Fast decision Procedures Based on Congruence Closure" *Journal of ACM*, 27(2), pp 356-364 (1980).

[34] O'Donnell M.J., "Computing in systems described by equations". In *Lecture Notes in Computer Science*, Vol 58, Springer, Berlin, West Germany, 1977.

[35] Oyamaguchi M., "The reachability problem for quasi-ground Term Rewriting Systems", *Journal of Information Processing* , vol 9 , n°4 (1986).

[36] Oyamaguchi M., "The Church-Rosser property for Ground Term Rewriting systems is decidable", *TCS* 49 pp. 43-79 (1987).

[37] Pelletier Francis Jeffry *Seventy five problems for testing automatic theorem provers.* Journal of automated reasoning (1986) pp. 191-216.

[38] Salomaa A. *Introduction à l'informatique théorique, Calcul et complexité.* Ed. Armand Colin 1989.

[39] Snyder, E.W., "Efficient Ground Completion: an O(n log n) Algorithm for generating Reduced sets of Ground Rewrite Rules Equivalent to a Set of Ground Equations." *Rewriting Technics and Applications*, Chapell Hill, North Carolina, April 1989, lecture Notes in Computer Science (Dershowitz ed.).

[40] Tarjan R.E., "Efficiency of a good but not linear set union algorithm". *Journal of ACM*, Vol 22 n°2 April 1975 PP. 215-225.

[41] Tourlakis G.J., *Computability*, Reston, VA, 1984.

NOM: DERUYVER PRENOM: ALINE

TITRE DE LA THESE:
DEUX ASPECTS DE LA REECRITURE:
UN LABORATOIRE POUR LES AUTOMATES: VALERIAN
UN SYSTEME DE DEMONSTRATION AUTOMATIQUE DE THEOREMES: EMMY.

Thèse, Informatique, LILLE I, 1991

Mots clés:
SYSTEME DE REECRITURE; AUTOMATE D'ARBRES; ACCESSIBILITE; TRANSDUCTEUR D'ARBRES CLOS; DEMONSTRATION AUTOMATIQUE; LOGIQUE EQUATIONNELLE; LOGIQUE DU PREMIER ORDRE; PROLOG.

RESUME:
Dans ce mémoire nous étudions deux aspects de la réécriture. Le premier travail est une contribution à l'étude structurelle de certaines classes de réécriture. Nous montrons comment certains problèmes d'accessibilité se traitent à la lumière des automates finis d'arbres.
De plus nous avons construit un logiciel nommé VALERIAN résolvant certains de ces problèmes en temps réél.
Le second travail consiste en la mise en oeuvre d'un système de démonstration automatique de théorèmes basé sur un calcul par superposition et fonctionnant pour des clauses logiques du premier ordre avec équations.