USTL
FLANDRES ARTOIS

UNIVERSITE DES SCIENCES ET TECHNIQUES
DE LILLE FLANDRES ARTOIS

N° d'ordre : 277

THÈSE

Nouveau Régime

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Patrick Lebègue

Contribution à l'étude de la Programmation Logique par les Graphes Orientés Pondérés.

Soutenue le 10 novembre 1988 devant la commission d'examen

Président :	Gérard	COMYN
Rapporteurs :	Jean-Paul	DELAHAYE
	Pierre	DERANSART
Directeur de thèse :	Max	DAUCHET
Examinateur :	Gérard	FERRAND
	Jean-Pierre	JOUANNAUD

IVERSITE DES SCIENCES
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. PARREAU, 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	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie végétale
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie-Physique
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	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 atmosphéri
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
Mlle 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 électroniques
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	Spectrochimie
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 électroniques optiques
M. CONYN 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
Mle DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBRABANT Pierre	Géologie appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELORME Pierre	Physiologie 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
Mle 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. FAUQUEMBERGUE Renaud	Composants électroniques
M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
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. LEFEVRE Christian	Pétrologie
Mle LEGRAND Denise	Algèbre
Mle 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 electroniques
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
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée

Tous mes remerciements à Gérard COMYN, Professeur à l'Université de Lille I, qui m'a fait l'honneur de présider le jury.

Je remercie aussi tout particulièrement Jean-Pierre JOUANNAUD, Professeur à l'Université d'Orsay, ainsi que Gérard FERRAND pour l'attention qu'ils ont portée à mes travaux.

J'exprime ma gratitude à Pierre DERANSART et Jean-Paul DELAHAYE qui ont bien voulu juger les résultats de cette thèse et en être les rapporteurs, leurs conseils féconds m'ont été très utiles.

Une attention toute particulière ira vers Max DAUCHET qui n'a ménagé ni son temps ni son énergie. Sa compétence et son imagination ont été les éléments moteurs de ce travail.

J'exprime bien sûr toute ma reconnaissance à Philippe DEVIENNE pour son étroite collaboration, ses conseils et ses critiques positives. J'ai eu beaucoup de plaisir à travailler avec lui.

Je tiens à exprimer ma profonde gratitude à tous ceux qui m'ont apporté aide et soutien au cours de l'élaboration de cette thèse, en particulier à tous mes collègues du département informatique de l'IUT.

Il serait injuste d'omettre dans mes remerciements Marylène, Christophe et Charlotte, ainsi que tous les menbres de ma famille pour leur patience et leurs encouragements.

Chap 1 - INTRODUCTION

Chap 2 -Travaux sur la terminaison

Chap 3 - Les Graphes Orientés Pondérés

Chap 4 -Hiérarchie de Programmes

Chap 5 -Décision de l'arrêt et complexité-

Chap 6 - Implémentation de résultats

Chap 1 - Introduction -

1) Avant propos.

2) Motivations et travail effectué.

3) Arrêt et complexité de programmes Prolog.

4) Exemple d'optimisation d'un interpréteur Prolog à l'aide des Gop's.

5) Résumés des chapitres.

6) Perspectives.

1 Avant propos

Le présent travail exploite et développe les résultats exposés dans la thèse de Philippe Devienne. L'ensemble des travaux contenus dans ces deux mémoires est le fruit d'une étroite collaboration entre Ph. Devienne et moi même, et a donné lieu par ailleurs à plusieurs communications communes.

Il s'agit d'une étude théorique de la complexité d'une classe de programmes Prolog. Nous considérons Prolog pur, sans prédicat standard, sans coupe-choix (cut) , ... etc , mais avec test d'occurrence. En effet, l'utilisation de solutions infinies [Prolog II] présente parfois des incohérences logiques. De notre point de vue, nous considèrerons toujours Prolog avec test d'occurrence, car ce dernier apporte une sémantique claire, indispensable à une étude théorique telle que celle présentée dans ce travail.

Le problème de l'application ou non du test d'occurrence est central [PLA 1984], bien que d'un complexité théorique raisonnable, les algorithmes d'unification effectuant ce test sont très peu utilisés dans la pratique en raison de leur difficulté d'implémentation (les structures de données utilisée sont très complexes) . C'est la raison pour laquelle peu d'interpréteurs effectuent ce test. Notons qu'il existe plusieurs méthodes donnant des conditions suffisantes pour que le test d'occurrence ne soit pas nécessaire [DER 1985][BEE 1988].

Dans la suite de cette introduction, nous allons présenter, dans un premier temps, la globalité du travail effectué ainsi que les motivations qui en sont à la base. Puis, nous étudierons plus en détails les différentes étapes chapitre par chapitre.

Le vocabulaire employé dans ce mémoire est un peu différent de celui habituellement rencontré dans la littérature sur Prolog. En fait, il s'agit plutôt du vocabulaire utilisé pour les systèmes de réécriture d'arbres [COU 1983]. En effet, nous considérons Prolog comme étant un système de réécriture particulier. Chaque arbre est étiqueté à la racine, par un symbole de prédicat. Ce qui oblige la réécriture à s'effectuer toujours à partir de la racine. De plus l'algorithme de filtrage est remplacé, dans Prolog, par celui d'unification. Ce qui le rapproche du "Narrowing".

Les clauses d'un programme Prolog sont donc appelées, par abus de langage, règles de réécriture, et le sous-but courant "terme réécrit".

L'ensemble des bases théoriques et pratiques nécessaires à l'étude de Prolog sont très largement détaillées dans [LLO 1984][APT 1987], et nous conseillons vivement au lecteur de s'y référer.

2 Motivations et travail effectué

Le mémoire de doctorat de Philippe Devienne [DEV 87] introduit un nouvel outil théorique : Le Graphe Orienté Pondéré. Les objets ainsi appelés (les Gop's), construits à partir de graphes orientés auquel on ajoute des pondérations entières d'arcs et des congruences de sommets (cf. Chap 3), sont munis d'un certain nombre d'opérations telles que substitutions, unifications ...

La simplicité des calculs, ainsi que leur capacité à schématiser le comportement dynamique de la réécriture d'arbres, du type de celle utilisée en programmation logique, en font des outils privilégiés pour l'étude de Prolog.

Les premiers résultats obtenus sont très prometteurs. Ils sont de deux types :

Premièrement : Il s'agit de conditions nécessaires et suffisantes simples, calculées sur un Gop associé à une règle récursive Prolog quelconque. La première condition nous indique si la règle récursive peut ou non être itérée une infinité de fois. Si c'est le cas, une deuxième condition nous permet de savoir si la règle admet ou non, une infinité de solutions. Nous avons, dans le même esprit, une condition suffisante pour qu'une règle récursive Prolog engendre systématiquement un échec, dû à une occurrence de variable lors de l'utilisation consécutive de la règle, un nombre connu de fois.

Deuxièmement : L'étude générale des gop's associés aux règles récursives simples Prolog (un seul prédicat n'apparaissant qu'une fois dans le corps de la règle), nous permet de prouver la linéarité en espace (taille des objets manipulés) par rapport au nombre d'appels consécutifs d'une telle règle, d'abord sous certaines conditions d'utilisation [DEV 87] telles que la linéarité des faits et des sous-buts (cf. Chap 3), puis bientôt (nous le pensons) dans le cas général.

Ces derniers résultats sur la complexité de Prolog, nous amènent tout naturellement à étudier, d'une part, l'arrêt et la complexité de programmes Prolog plus généraux, et d'autre part, dans quelles mesures on peut implémenter le calcul des Gop's dans un interpréteur Prolog en vue d'optimiser sont comportement. C'est ce que le présent travail se propose de faire. Les chapitres 2 à 5 sont consacrés à l'étude de l'arrêt et de la complexité d'une classe de programmes Prolog, le chapitre 6 à un exemple d'implémentation de la structure de Gop dans un interpréteur Prolog.

Nous nous bornons à étudier ici, les Gop's dans le cadre de l'analyse de programmes Prolog, mais le développement de l'étude des Gop's du point de vue algébrique [Courcelle] reste à faire.

Approximativement, nous pouvons dire que ce mémoire se résume ainsi :

"On peut tout décider et tout dire du comportement et de la complexité de programmes dont la structure se réduit à :

(i) $P( ) \rightarrow ;$

$P( ) \rightarrow P( ) ;$

ce qui peut paraître très restreint. Mais il faut savoir que nous montrons que tout programme Prolog peut s'exprimer sous la forme :

(ii) $P( ) \rightarrow ;$

$P( ) \rightarrow ;$

$Q( ) \rightarrow ;$

$Q( ) \rightarrow Q( ) P( ) ;$

et que tout programme ayant une structure plus simple, peut être réduit à un programme de la forme (i), ce qui explique qu'on ne peut rien en dire de général."

Ce trou entre "tout et rien" pour si peu de différence syntaxique illustre une fois de plus les limites de l'approche purement syntaxique. Notre analyse met en lumière des méthodes syntaxiques indécidables, mais naturelles, qui, grâce à des calculs sémantiques (attributs), aboutissent à des critères d'arrêt et de mesure de complexité pour de larges classes de programmes. Ainsi, la syntaxe retrouve son rôle d'aide dans un environnement de programmation.

3 Arrêt et complexité de programmes Prolog

La structure de base d'un programme Prolog sur laquelle s'appuie l'étude des Gop's, est de la forme :

$$P(\ ) \rightarrow P(\ );$$

pour laquelle nous avons prouvé le comportement uniforme. L'extension de ce résultat à des règles du type :

$$P(\ ) \rightarrow P(\ )\ Q_1(\ )\ Q_2(\ )\ \ldots\ Q_n(\ );$$

telles que aucun Q_i ne peut engendrer, directement ou indirectement, un sous-but de prédicat P, est immédiate sous certaines conditions énoncées plus loin.

Cette structure de règle Prolog peut paraître pauvre, en fait il n'en est rien, car comme déjà annoncé, nous prouvons dans le chapitre quatre, que tout programme Prolog peut s'écrire avec une seule règle récursive de cette forme, et une base de faits. Cette dernière pouvant être d'ailleurs réduite à trois faits.

Le type de règles décrit précédemment est facilement exploitable à l'aide des graphes orientés pondérés. C'est pourquoi, une première étape de notre travail consiste à classer les programmes Prolog, en fonction de leur complexité structurelle. Il s'agit donc de construire une hiérarchie de programmes en ne tenant compte que de la forme des règles (sans regarder les arguments de prédicat) ainsi que la façon dont elles s'appellent les unes les autres.

On définit dans cette étape, de manière tout à fait classique, comme pour les grammaires d'arbres, les notions de programmes expansifs et de programmes quasi-rationnels. Pour ce faire, on associe à chaque programme Prolog une grammaire algébrique de mots, représentant l'**appel** des différents prédicats. De manière identique, nous introduisons la notion de programmes déterministes.

Précisons que toutes ces propriétés de programmes ne sont que des raffinements construits à partir de la notion très classique de **graphe d'appel des prédicats** pour un programme Prolog.

La réduction de tout programme Prolog à une seule boucle (chap 4) nous donne un "méta-programme" [HIL 1988] pratiquement illisible, sans aucune utilité pratique pour le programmeur. En revanche, il est possible d'effectuer des transformations de programmes dans le but d'obtenir une structure plus simple, facile à analyser à l'aide des

Gop's, tout en conservant une forme et un graphe d'appel des prédicats ayant un sens du point de vue du programmeur. Nous exposons donc un certain nombre de transformations de base généralisant nos résultats de complexité à un ensemble de programmes plus large.

Dans une deuxième étape, nous introduisons, de manière formelle, une notion qu'utilise, parfois implicitement, tout programmeur. Cette notion, classique en programmation procédurale, l'est beaucoup moins en programmation logique. Il s'agit d'exprimer les convergences simple et uniforme des données (sous-buts) au cours de la résolution. Nous nous donnons deux critères de convergences (simple et uniforme), assurant l'arrêt des programmes les vérifiant. Ces critères sont établis à partir de la construction de fonctions décroissantes, de façon uniforme ou non, sur un espace bien fondé.

L'indécidabilité de tels critères est démontrée dans le cas général. En revanche, et ceci grâce à l'étude des Gop's associés aux règles récursives, nous montrons la décidabilité de ces critères pour les programmes quasi-rationnels et, lorsqu'un programme vérifie l'un ou l'autre des critères, nous évaluons sa complexité. Cette dernière est mesurée en nombre d'applications de règles du programme et est entièrement automatique. Il est important de noter que l'extension de ces calculs à une large classe de programme se fera sans problème particuliers de manière semi-automatique, c'est là sa véritable vocation.

4 Exemple d'optimisation d'un interpréteur Prolog à l'aide des GOP's

Il s'agit dans cette partie d'évaluer la difficulté d'implémenter la structure de Gop dans un interpréteur Prolog. Pour cela nous nous sommes posés pour objectif d'intégrer dans un interpréteur une partie des résultats présentés dans [DEV 1987].

En fait, dans une phase de précompilation, nous construisons le Gop associé à chacune des règles récursives, puis nous évaluons les différentes constantes relatives à la règle (périodicité, taux de croissance des arguments consommés et générés par la règle, test d'occurrence systématiquement bloquant). Ces calculs sont présentés dans [DEV 1987] uniquement par une analyse statique du Gop associé à chaque règle. L'expérience nous montre que le résultat n'est alors pas satisfaisant. C'est pourquoi, le calcul des constantes est fait en partie de manière statique, en partie de manière dynamique lors d'une simulation d'utilisation de la règle récursive. Cette simulation a l'inconvénient d'augmenter le temps de compilation, mais elle diminue, parfois très sensiblement, le temps d'exécution. Une preuve de la correction de la méthode utilisée ainsi que quelques exemples sont exposés au chapitre 6.

5 Résumés des chapitres

5.1 Chapitre 2

Il s'agit d'une étude bibliographique, très partielle, de travaux sur la terminaison et l'optimisation de programmes Prolog. En fait, nous avons, dans ce chapitre, placé en avant un certain nombre de problèmes plus ou moins liés à ceux exposés dans ce mémoire, ainsi que des travaux qui nous ont parus intéressant ou originaux.

Nous abordons dans un premier paragraphe, les problèmes relatifs au test d'occurrence. Puis, nous résumons quelques approches concernant la détection et l'élimination de boucles lors de la résolution. Deux principales méthodes sont à signaler, la première est statique, et utilise comme nous le faisons, la notion de graphe d'appel des prédicats. La seconde est dynamique, et agit directement sur le moteur de résolution Prolog. Quelques prédicats de contrôle sont étudiés dans ce chapitre à titre d'exemples.

Nous citons par la suite, les travaux de P. Deransart et J. Maluszynski sur le lien existant entre la programmation logique et les grammaires attribuées. Une notion que nous utilisons dans cette thèse y est introduite : les programmes dirigés par les données. Cette propriété concerne la propagation de la clôture des termes au cours de la résolution. Elle va nous être très utile par la suite et sera un prémisse à nos critères de convergence.

5.2 Chapitre 3

Dans ce chapitre, nous résumons le contenu du mémoire de Philippe Devienne. Cette partie est indispensable au lecteur par le lien étroit qu'il existe entre elle et le présent mémoire. De plus elle nous permet d'introduire le formalisme utilisé ainsi que le vocabulaire employé. Nous résumons les notions liées aux monoïdes, aux magmoïdes ainsi que les différentes grammaires les concernant. Nous rappelons aussi les notions de graphes orientés et de graphes orientés pondérés ainsi que les différents résultats exposés dans [DEV 1987].

Le "tant que" Prolog : Soit P un symbole de prédicat.

Un "tant que" Prolog est un programme de la forme :

$P(\alpha) \rightarrow ;$

$P(\beta) \rightarrow P(\gamma);$

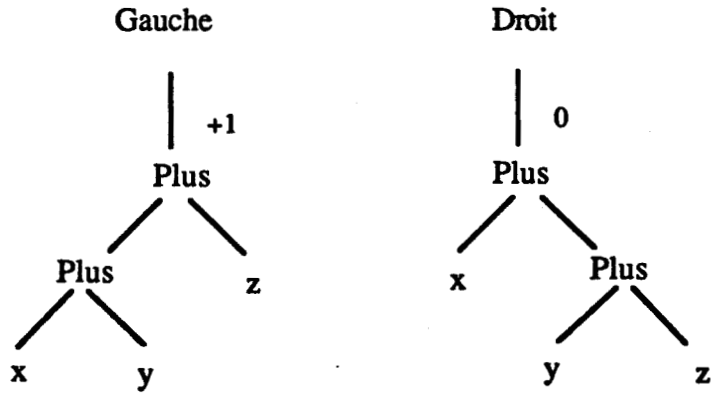
$P(t)?$

où α,β,γ sont des arbres construits sur un alphabet fini gradué Σ et un ensemble de variables V.

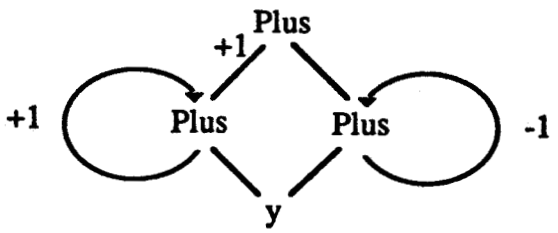
Exemple de règle récursive :

$$P(Plus(Plus(x,y),z)) \rightarrow P(Plus(x,Plus(y,z)));$$

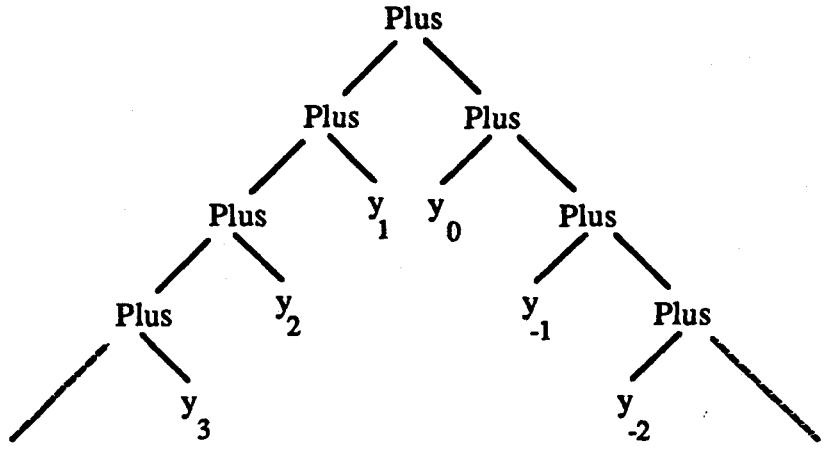
Pour utiliser une règle Prolog, il est nécessaire de renommer les variables afin d'éviter leur capture. Ceci est fait en associant un indice à chaque variable. Dans le cas du "tant que", cet indice est égal au nombre d'utilisations de la règle déjà effectuées. (3ème utilisation : les indices sont compris entre 0 et 2). Le calcul du Gop associé à la règle récursive s'effectue en une seule unification des termes gauche et droit de la règle, en additionnant toutes les contraintes sur les variables comme si la règle était utilisée indéfiniment : on unifie le terme de droite à l'indice i, avec le terme de gauche à l'indice i+1. Ceci pour tout i. Ce qui se schématise de la manière suivante :



après unification , on obtient le schéma du plus petit terme se réécrivant indéfiniment par la règle : Le GOP :



dont l'interprétation donne l'arbre infini :



Les informations que l'on déduit du Gop sont les suivantes :

- son existence ainsi que le fait qu'aucune boucle de poids cumulé nul ne peut être faite par parcours du Gop (sinon il y aurait "occurrence"), nous assure qu'il existe au moins un terme (question) pouvant être réécrit une infinité de fois (le test d'occurrence ne peut bloquer la résolution qu'à cause de la forme de la question).

- la valeur des poids de boucles ainsi que leur longueur, indiquent un taux de croissance moyen des branches des termes initiaux et réécrits lors de la résolution. En fait, les boucles correspondent à des branches poussantes en fonction du nombre de pas de réécriture, dans le terme initial (boucles de poids positif) ou dans le terme final (boucles de poids négatif), sachant qu'une branche qui ne pousse pas durant un temps suffisamment grand, ne poussera plus.

Dans l'exemple ci-dessus, la branche de gauche pousse effectivement d'un nœud à chaque pas dans le terme initial (poids de boucle = +1), celle de droite d'un nœud à chaque pas dans le terme final (poids de boucle = -1).

Le lecteur se convaincra vite que le présent propos, évident sur l'exemple, est inextricable de façon naïve dans le cas général, tant est trompeuse, comme dit précédemment, l'apparente simplicité structurelle d'une seule boucle. Les Gop's permettent de fonder l'étude dans le cas général.

De ce comportement uniforme, mis en évidence lors du calcul du Gop, on déduit la linéarité en espace (tailles des objets) du "tant que" Prolog lorsque le but et le fait sont linéaire (une seule occurrence de chaque variable).

C'est à dire que si T_n est la réponse à une question posée telle que U_n soit le terme unifié avec le fait après n utilisations de la règle récursive, alors :

$$\exists A,A',B,B' \in N \text{ tq} \quad An+B \leq \text{Taille}(Tn) \leq A'n+B'$$

$$\text{et} \quad An+B \leq \text{Taille}(Un) \leq A'n+B'$$

La taille des termes désigne, pour notre étude, la hauteur des arbres qui les représentent, les arbres étant représentés sous forme de Go's (Dag's).

5.3 Chapitre 4

Dans ce chapitre, nous abordons deux types de problèmes. Le premier concerne une classification structurelle des programmes Prolog. Le second invoque un certain nombre de transformations permettant de passer d'une classe de programmes à l'autre, tout en conservant les propriétés opérationnelles des programmes Prolog. Pour cette dernière raison, nous introduisons la notion de Prolog-équivalence, dont une idée intuitive est donnée dans la suite de ce paragraphe.

Les différentes classes de programmes Prolog que nous utilisons dans ce mémoire, sont fondées sur l'étude du classique graphe d'appel des prédicats.

Nous construisons, à partir du programme Prolog, une grammaire algébrique de mots dont les non terminaux sont les symboles de prédicats. L'ensemble des mots, constitués de lettres non-terminales, que l'on peut engendrer à partir du prédicat de la question, est l'ensemble des suites d'appels (effacements) valides de prédicats lors de la résolution Prolog, sans tenir compte des arguments de ces prédicats. Le graphe de précédence de cette grammaire est similaire au graphe d'appel des prédicats. A partir de ce graphe, nous définissons la relation de dépendance : Le prédicat P "peut appeler" le prédicat Q (on dira que Q est "dépendant" de P) s'il existe un chemin dans le graphe allant de P à Q. Cette relation définit une relation d'équivalence (référence croisée) sur l'ensemble des prédicats du programme : P équivaut à Q si et seulement si P peut appeler Q et Q peut appeler P. On définit classiquement, sur l'ensemble des classes d'équivalence induit par cette relation, une fonction entière : Ordre(classe(P)) ou plus brièvement Ordre(P), comme étant le nombre d'imbrications de boucles d'appel possibles à partir de P.

Dans l'exemple suivant, Add est d'ordre 1, Mul d'ordre 2 et Exp d'ordre 3.

Exemple : Pour le programme :

1 Add (0 , x , x) →;

2 Add (sx, y , sz) → Add (x , y , z);

3 Mul (0 , x , 0) →;

4 Mul (sx , y , z) → Mul (x , y , z') Add (y , z' , z);

5 Exp (x , 0 , s0) →;

6 Exp (x , sy , z) → Exp (x , y , z') Mul (x , z' , z);

La grammaire algébrique associée est :

Add –1→ e

Add –2→ Add

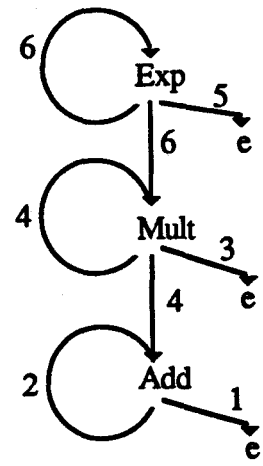
Mul –3→ e

Mul –4→ Mul Add

Exp –5→ e

Exp –6→ Exp Mul

Le graphe d'appel des prédicat, ou plus exactement, le graphe de précédence de la grammaire algébrique associée au programme est :



Lorsque cette grammaire est **quasi-rationnelle** (au sens de la théorie des langages), c'est à dire qu'aucun non-terminal A n'est tel que A engendre un mot de la forme uAvAw (le comportement expansif d'un tel non-terminal est immédiat), nous dirons que le programme est quasi-rationnel, par opposition aux programmes **expansifs**.

De plus, nous dirons que cette grammaire est récursivement **déterministe** (par la suite nous omettrons le mot "récursivement") si elle vérifie la propriété suivante :

Si A engendre uAv et u'Av', cela ne peut se faire que consécutivement. C'est à dire qu'il faut passer part l'un des deux mots pour obtenir l'autre. Autrement dit, il n'y a qu'une seule façon d'appeler un prédicat à partir de lui-même, d'où la terminologie de **"déterminisme récursif"** .

Les programmes quasi-rationnels déterministes ont la propriété intéressante de se comporter comme une suite finie de **"tant que"** Prolog. C'est pourquoi nous allons tenter de montrer quelques techniques de transformation de programmes Prolog afin de se ramener à cette structure.

Pour cela nous définissons deux **équivalences de programmes Prolog**. La première, très classique, est une équivalence au sens des solutions en stratégie complète, elle n'est utile qu'à fragmenter la transformation afin d'en simplifier l'exposé. On l'appelle l'**équivalence simple**. La seconde, beaucoup plus forte, est une équivalence opérationnelle pour la stratégie Prolog standard, elle assure que deux programmes équivalents auront le même ordre de complexité, et que l'utilisateur verra les deux programmes se dérouler de manière identique même dans le cas d'utilisation de prédicats d'entrée-sortie. On appelle cette équivalence **"Prolog-équivalence"**. Ces deux équivalences sont définies pour deux programmes et un ensemble de prédicats inclus dans celui commun aux deux programmes, cet ensemble est celui des "questions possibles".

Dans un premier temps, nous allons normaliser la forme des programmes quasi-rationnels déterministes. Pour cela nous allons réduire à 1 la longueur de chaque cycle dans l'appel des prédicats. Nous utilisons le résultat suivant :

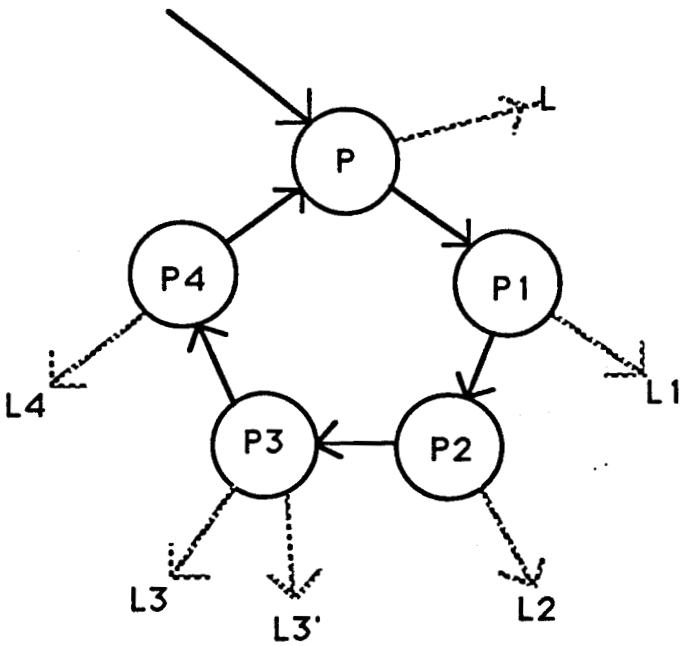
Théorème 1 : *Tout programme quasi-rationnel déterministe admet un programme équivalent (équivalence simple) ayant toutes ses règles de la forme i ou de la forme ii :*

i) $P() \rightarrow Q_1() \; Q_2() \; ... \; Q_n() \; ;$

ii) $P() \rightarrow P() \; Q() \; ;$

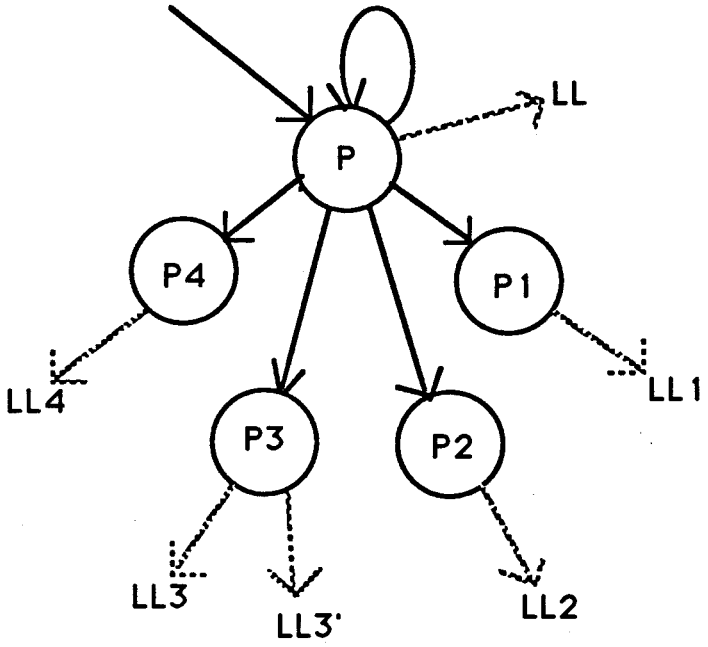
où les Q et Qi ne peuvent pas appeler, directement ou indirectement, le prédicat P.

Nous savons , puisque le programme Prolog à transformer est quasi-rationnel déterministe, que pour chaque prédicat récursif, il n'y a qu'une seule façon de boucler. La boucle est de longueur quelconque. Le graphe d'appel a donc la forme suivante :



où L et les Li désignent une liste de littéraux.

Il s'agit de le transformer pour obtenir le graphe suivant :



Ce qui donne les mêmes résultats pour toute question de prédicat P. (Notons qu'il ne peut y avoir qu'un seul prédicat, appartenant à l'ensemble distingué des questions possibles, par cycle d'appel. Cette contrainte n'enlève rien à la généralité de la transformation, car tous les prédicats d'un même cycle sont équivalents à une opération constante près).

Cette transformation s'effectue en ajoutant des règles simulant toutes les fractions initiales des cycles. Puis en retirant du programme la dernière règle de chaque cycle (celle qui contient dans le corps le symbole de prédicat choisi comme question possible).

Dans un premier temps, nous présentons cette transformation sans tenir compte de l'ordre des prédicats en partie droite des règles (ce qui donne bien, comme le dit le théorème, un équivalent au sens de l'équivalence simple). Cela permet d'obtenir des récursivités gauches uniquement.

Par la suite, nous reprenons la même technique, en respectant l'ordre des prédicats en partie droite, ce qui nous permet d'obtenir des programmes Prolog-équivalents. Nous utilisons alors le théorème suivant :

Théorème 2 : *Tout programme quasi-rationnel déterministe admet un programme Prolog-équivalent ayant toutes ses règles de la forme i ou de la forme ii :*

i) $P() \rightarrow Q_1() \; Q_2() \; ... \; Q_n()$;

ii) $P() \rightarrow Q() \; P() \; Q'()$;

où les Q , Q' *et* Q_i *ne peuvent pas appeler, directement ou indirectement, le prédicat* P.

La transformation obtenue est sensiblement plus complexe, mais l'idée de base est la même.

Ce chapitre se termine sur l'écriture d'un interpréteur Prolog en Prolog avec une seule règle et une base de fait. d'où le théorème :

Théorème 3 : *Tout programme Prolog possède un programme Prolog-équivalent quasi-rationnel déterministe ne contenant qu'une seule règle et des faits.*

Remarque : Le nombre de faits peut être réduit à trois [DEV].

Ce résultat n'a d'intérêt que dans la justification d'une étude très détaillée du simple "tant que" Prolog (simple en apparence). En effet, nous avons là une preuve que tout programme Prolog peut s'écrire sous une forme très proche du "tant que".

5.4 Chapitre 5

Nous allons, dans ce chapitre, formaliser pour la programmation logique, une notion de convergence utilisée couramment en programmation procédurale classique. Cette notion est fondée sur l'écriture d'une fonction numérique décroissante (de façon uniforme ou non) sur un espace bien fondé. En fait, il s'agit de quantifier l'information circulant dans le programme, via les différents cycles d'appels des prédicats.

Le contrôle de la circulation d'information est lié, dans notre étude, à la notion d'attribution de modes d'entrée/sortie. Pour cela nous faisons référence à la technique introduite dans [DER 1985], sur le lien existant entre la programmation logique et les grammaires attribuées. Chaque position d'argument d'un prédicat donné, reçoit l'annotation "hérité" ou "synthétisé". Un argument de prédicat est en entrée dans une règle donnée, s'il est hérité de la tête ou s'il est synthétisé du corps de la règle. Réciproquement, un argument d'un prédicat donné est en sortie d'une règle, s'il est synthétisé de la tête ou s'il est hérité du corps de la règle.

Un programme est dirigé par les données, si et seulement si, lorsque la question possède des termes clos à toutes les positions héritées, alors à chaque étape de la résolution, le sous-but courant possède des termes clos à toutes ses positions héritées au moment de l'appel, et à toutes ses positions synthétisées lorsqu'il est résolu.

Par la suite, nous considèrerons uniquement des programmes possédant cette propriété.

Critères de convergence

Les critères énoncés ici, ne sont pas originaux, il sont inspirés des classiques ordres de réduction dans les systèmes de réécriture, dont une synthèse nous est présentée dans [JOU 1986]. Il s'agit d'établir un ordre bien-fondé (sans chaîne infinie descendante) compatible avec la structure des termes. Ces techniques ne sont évidemment pas universelles, mais elles donnent, en général, de bons résultats dans la pratique. Notre étude a le mérite d'obtenir la décidabilité et de calculer la complexité pour une sous-classe de programmes.

Les deux critères que nous allons exposer, sont des critères purement syntaxique. Ils seront accompagnés de contraintes sémantiques sur les programmes étudiés, et s'intègreront sans doute dans un univers d'aide au programmeurs. En effet, une contrainte que l'on va utiliser est que les programmes soient "dirigés par les données". c'est à dire que le flot d'information que l'on mesure et qui doit vérifier une condition de convergence, est composé de termes clos (cette contrainte un peu forte, peut sans aucun doute être allégée par une étude sur la clôture partielle des données). Les programmes possédant l'un ou l'autre des critères s'arrêtent. Nous allons voir que la vérification d'une telle condition, assez naturelle, est loin d'être triviale, même dans des cas très simples. En revanche, grâce à une étude sur les Gop's, nous prouvons la décidabilité de ces critères sur la classe des programmes quasi-rationnels. De plus, cette étude nous permet de conclure par une évaluation numérique de la complexité des programmes de cette classe vérifiant ces critères.

Critère de convergence simple :

Ce critère effectue une vérification de convergence de la taille des données closes manipulées par le programme, pour tous les chemins d'appel ne sortant pas d'une classe de prédicats donnée (les classes sont définies aux chapitre 4). C'est à dire que nous allons vérifier la convergence stricte des données pour chaque classe de prédicats, sans regarder la façon dont la résolution se fait en dehors de la classe étudiée. Il s'agit donc d'un critère "hors-contexte".

Pour cela, nous définissons ce qu'est le projeté d'un programme Prolog sur une classe de prédicats. Il s'agit de l'ensemble des règles du programme initial dans lequel on a éliminé tous les littéraux n'ayant pas un symbole de prédicat de la classe.

Les données étant closes et convergeant strictement pour ce programme projeté, dans le programme complet, tout appel à un sous-but dont le prédicat est dans la classe ne peut engendrer qu'un nombre fini et linéaire d'appels à des prédicats de la même classe. Si toutes les classes de prédicats du programme vérifient cette propriété, alors le programme s'arrête.

Ce critère est indécidable dans le cas général. De plus il ne permet pas, à priori, de conclure sur la complexité des programmes.

Critère de convergence uniforme :

Ce critère, plus contraignant que le précédent, vérifie la convergence de la taille des données closes manipulées par le programme dans sa totalité, sans le fractionner classe par classe. Nous vérifions que l'ensemble des données converge strictement et de façon uniforme sur l'ensemble du programme. Un programme vérifiant ce critère vérifie donc forcément le critère précédent, donc il s'arrête. Ce que le nouveau critère apporte de plus, est une évaluation du temps d'exécution des programmes qui le vérifient. Ils sont tous uniformément linéaires.

Bien que les contraintes que doivent vérifier les programmes satisfaisant ce nouveau critère soient très fortes, elles sont ensembles indécidables.

L'intérêt d'un tel résultat est plutôt négatif. Il montre, s'il en est encore nécessaire, que des propriétés très strictes sur le comportement de programmes Prolog sont indécidables. Ce qui tente à montrer que la notion de programmation logique structurée n'est pas inutile.

Décidabilité du critère de convergence simple et complexité dans le cas quasi-rationnel.

Les résultats précédents sur les programmes quasi-rationnels nous montrent que tous programmes non déterministes peut être transformé en un programme Prolog-équivalent déterministe, et que tout programme quasi-rationnel déterministe pouvait être transformé en un programme Prolog-équivalent quasi-rationnel déterministe ayant toutes ses classes de prédicats réduites à un seul élément chacune (chaque boucle d'appel est de longueur 1).

Nous montrons, dans ce cas, l'équivalence entre la propriété du critère de convergence simple, et celle de l'existence, dans le Gop associé à chaque règle récursive, d'une boucle de pondération positive effectivement dépliée lors de la résolution (cf. Chap 3), cette boucle correspondant à un argument en entrée de la règle. Cette propriété est décidable.

De plus, pour chaque cycle d'appel, l'étude de la croissance des termes lors de la résolution étant possible au regard du Gop associé à la règle récursive, une évaluation de la complexité des programmes vérifiant le critère de convergence simple est possible. Nous en montrons le calcul en fin de ce chapitre 5.

Nous avons énoncé une méthode de vérification automatique de l'arrêt d'une classe restreinte de programmes Prolog accompagnée d'un calcul automatique de complexité. En fait, il est clair que cette méthode a vocation d'être intégrée dans un univers d'aide à la programmation, et ainsi, de façon interactive, se généraliser à une vaste classe de programme, rendant automatique une importante part des calculs servant à en vérifier l'arrêt et à en connaître la complexité.

5.5 Chapitre 6

Dans ce chapitre nous détaillons une implémentation de calcul de graphes orientés pondérés dans le cadre d'un interpréteur Prolog. Nous calculons le Gop associé pour chaque règle récursive simple détectée dans le programme interprété. Ce calcul associé à une simulation d'utilisation nous permet d'obtenir des renseignements précis sur le comportement de certaines règles récursives (périodicité, solutions possibles ..etc). En modifiant légèrement le moteur Prolog, on obtient alors des résultats encourageant sur l'utilité d'une telle méthode.

Nous montrons un certain nombre d'exemples de programmes Prolog dont le comportement est amélioré grâce à ces calculs.

Le travail réalisé dans cette partie est très restreint. Il nous parait évident qu'il faille l'approfondir et l'intégrer dans un univers complet d'aide à la programmation. Il serait judicieux d'y associer des algorithmes de détection de boucles, ainsi qu'une attribution de modes automatique permettant d'exploiter, de manière automatique, le critère simple énoncé plus haut sur des parties de programmes dont il serait alors aisé de prévoir le comportement.

6 Perspectives

Un certain nombre d'approfondissements de ce travail peuvent être envisagés.

- Il nous semble intéressant d'affiner **l'étude théorique sur la structure des programmes Prolog**. Des travaux sont en cours en collaboration avec Philippe Devienne qui vient de publier un article sur les Gop's et la programmation logique [DEV 88].

- L'étude des graphes orientés pondérés en temps qu'**outils algébriques** possédant de bonnes propriétés est, sans doute, une perspective essentielle.

- La facilité d'**implémentation**, au sein d'un interpréteur Prolog, des résultats obtenus sur la réécriture, ouvrent la possibilité d'utiliser ces outils comme aide aux programmeurs. En effet, il est envisagé d'intégrer nos résultats sur l'étude des boucles dans un interpréteur Prolog utilisé en travaux pratique par les étudiants en tant qu'environnement de programmation (visualiser le comportement et la complexité de programmes).

- Les critères de convergence que nous énonçons, suppose préétablie une condition de clôture sur les termes, condition qui dépend de la notion d'attribution de mode [MEL 81]. En fait il y a convergence, donc arrêt d'un programme quasi-rationnel, si il y a compatibilité entre **l'attribution des modes** et la structure des Gop's associés aux règles récursives. Il y a donc là une possibilité d'aider le programmeur en proposant l'ensemble des attributions de modes valides, s'il en existe, telles que le programme vérifie le critère, et donc s'arrête.

Chap 2 - Travaux sur la terminaison de programmes PROLOG -

1) Introduction.

2) Travaux sur la terminaison.

1 Introduction

De nombreuses publications concernant le problème de l'arrêt de programmes Prolog ont vu le jour ces dernières années, et il serait vain d'en faire une liste exhaustive. Aussi, allons nous nous contenter de citer, plutôt que des résultats, un certain nombre de démarches ayant chacune son originalité, en essayant d'en démontrer les avantages et les inconvénients.

Il est notoire que le problème de l'arrêt d'un programme est indécidable. L'étude de la terminaison se réduit donc en général à une condition suffisante à la terminaison.

Ce problème a été très largement étudié pour les systèmes de réécriture pour lesquels l'indécidabilité de l'arrêt a été prouvée pour 3 règles [LIP 1977], puis pour 2 [DERS 1985], puis récemment pour une seule [DAU 1987].

L'étude de la terminaison se borne donc, en général, à trouver une condition suffisante pour que la réécriture s'arrête. Nous nous inspirons des techniques utilisées dans ce domaine pour les appliquer à la programmation logique.

2 Travaux sur la terminaison

La non terminaison lors de la résolution (le problème de la non-terminaison lors de l'unification est lié à la non application du test d'occurrence, nous n'en parlerons donc pas) doit sa cause au parcours, dans l'arbre de résolution, d'une branche infinie. Les branches infinies peuvent être de différentes natures, et il est parfois difficile, voire impossible, de les détecter. En général, la détection de ces branches revient à trouver les boucles intervenant lors de la résolution. De très nombreux travaux sont relatif à la détection de boucles :

[COV 1985] : Covington nous dit que si un sous-but est plus général qu'un sous-but apparaissant plus haut dans l'arbre de preuve, alors la branche peut être bloquée. En effet l'évaluation de ce sous-but engendrera une boucle infinie.

Dans [BRO 1979] un échec est générée lorsque les sous-buts sont "syntaxiquement identiques".

Les limites de telles méthodes sont exposées dans [POO 1985] .

Deux courants distincts sont à noter : le premier consiste à faire cette détection de boucles de manière dynamique, et donc à intervenir sur l'algorithme de résolution en vue d'éliminer les causes de bouclages (soit en éliminant des sous-buts, soit en modifiant la règle de choix), le second consiste en une analyse statique du programme en vue de reconnaître les schémas de bouclage possibles. Ces deux approches sont complémentaires.

Dans l'approche dynamique, il y a bien sûr un compromis à trouver entre le type de boucles détectées et le coût de la détection, ce dernier étant loin d'être négligeable. L'insertion de prédicats de contrôle est largement étudiée et, si l'efficacité d'une telle méthode n'est plus à mettre en cause, elle enlève une bonne partie de l'intérêt de Prolog en tant que langage déclaratif (Cut, Geler, ..).

L'arrêt des programmes Prolog est étudié, par ailleurs, grâce à un contrôle d'origine sémantique. On associe statiquement à chaque symbole de prédicat un certain nombre de propriétés sémantiques qui doivent être vérifiées lors de la résolution.

Citons :

Le WAIT [NAI 1982] est associé à un prédicat et non à une occurrence de prédicat. Il retarde toute évaluation de celui-ci tant que certains arguments ne sont pas suffisamment instanciés. Les WAIT peuvent être insérés automatiquement lors d'une analyse statique

du programme [NAI 1985].

L'exemple le plus classique est écrit sur la concaténation de liste : Append:

Wait Append (1 , 1 , 0)

Wait Append (0 , 1 , 1)

Append([ ] , A , A) → ;

Append (A.B , C , A.D) → Append (B , C , D) ;

Lorsque le but courant de prédicat Append n'a pas l'un des deux premiers arguments clos, ou l'un des deux derniers arguments clos, alors l'effacement du sous-but est retardé. Il est alors placé à la fin de la liste des sous-buts.

Il existe bien d'autres structures de contrôle. Notre but n'est pas d'en faire une étude complète mais de mettre en évidence l'idée sous-jacente : la réduction du non-déterminisme (choix dans les règles) lors de la résolution. Il s'agit de diminuer le plus possible l'espace de recherche par élagage des branches ne portant aucune solution, ou des solutions déjà trouvées.

Bien que certaines structures comme le WAIT peuvent parfois être insérées automatiquement dans le programme [NAI 1985], cette tache incombe le plus souvent au programmeur, ce qui suppose une maîtrise parfaite de la stratégie employée par l'interpréteur, le réordonnancement dynamique des littéraux lors de l'exécution étant parfois complexe. C'est à ce niveau qu'il faut y voir un handicap, le programmeur doit absolument avoir une idée itérative de son programme.

Une autre méthode dynamique consiste à étudier intéractivement et dans une phase de mise au point le déroulement complet du programme. On y détecte alors les différentes boucles, leurs causes, et si possible la façon d'y remédier. Citons Le Debugger [SHA 1983] ainsi qu'une extension de ce debugger à l'aide d'une approche en terme de "dénotation relative" [FER 1985].

Dans l'approche statique, il est souvent fait appel à la notion de mode [MEL 1981] : on associe à chaque argument d'un prédicat un mode indiquant si l'argument est en entrée ou en sortie. L'attribution de ces modes est en général indécidable [KAT 1986]. Souvent c'est le programmeur qui en a la charge. Toutefois l'attribution automatique des modes, est très largement étudiée : [MEL 1981] [DEB 1988].

L'approche formelle présentée dans [DER 1985] nous a parue très intéressante. Elle

nous montre le lien étroit qu'il y a entre la programmation logique et les grammaires attribuées. Ce lien y est utilisé, à titre d'exemples, pour déterminer la non nécessité du test d'occurrence, puis pour donner une condition suffisante pour qu'un programme logique puisse être dirigé par les données (à chaque étape de la résolution, si avant l'effacement d'un littéral tous ses arguments en entrée sont clos, alors tous les arguments en sortie seront clos lors de l'effacement), puis finalement pour apporter une condition suffisante pour qu'un programme Prolog puisse être exécuté sans unification (le filtrage suffit alors). Cette méthode est fondée sur la notion de Schémas de Dépendance d'Attributs (SDA).

Chaque position d'argument des prédicats du programme prolog est muni d'une annotation de direction indiquant si l'argument est hérité ou synthétisé (↑ pour hérité, ↓ pour synthétisé). Une flèche apparait entre deux arguments dans le schéma s'il y a transfert d'information d'une position d'entrée vers une position de sortie, (en entrée si hérité d'une tête de clause ou synthétisé d'une queue de clause, inversement en sortie si synthétisé d'une tête de clause ou hérité d'une queue de clause), c'est à dire si une même variable apparait aux deux positions.

Le schéma induit par le SDA dans un arbre de preuve d'un but fermé donné pour un programme donné, nous renseigne sur la manière dont le transport d'information s'effectue au cours de la résolution. L'idée qui s'en suit est qu'il est sans doute possible, en utilisant cette technique, d'obtenir des renseignements précis sur l'arrêt et peut être sur la complexité de certains programmes Prolog.

L'exemple qui suit est tiré de [DER 1985]

Pour les clauses

1 Plus (0, x, x)

2 Plus (sx, y, sz) -> Plus (x, y, z) ;

munies de l'annotation Plus(↓ ↓ ↑)

SDA associé: 1 Plus (0, x, x)

2 Plus (sx, y, sz)

↓ ↓ ↑

Plus (x, y, z)

d'où le schéma de dépendances induit pour un arbre de résolution particulier :

	Plus (	sss0,	x,	sssx)
		↓	↓	↑
2	Plus (	ss0,	x,	ssx)
		↓	↓	↑
2	Plus(	s0,	x,	sx)
		↓	↓	↑
1	Plus(	0,	x,	x)
		↓	↑	

On comprend bien comment ce schéma peut nous donner des renseignements sur la clôture des différents arguments des sous-buts lors de la résolution.

D'autre part, citons [BAU 1988] où une approche sémantique est faite des propriétés de terminaison des programmes Prolog. Un point de vue dénotationnel y est présenté : à chaque programme on associe un système d'équations fonctionnelles dont la solution en tant que plus petit point fixe est caractéristique du programme. C'est sur ce système d'équation que les raisonnement formels sont bâtis, et que des propriétés sur la terminaison sont élaborées. L'intérêt principal de la méthode présentée est sa capacité d'être implémentée de manière automatique et la possibilité d'intégrer le Cut et la négation.

Citons aussi l'approche théorique de [POT 1986] dans lequel une classification des questions de programmes en fonction de différents types d'arrêts souhaités .

Chap 3 - Les Graphes Orientés Pondérés (GOP's) -

1) Introduction.

2) Le "Tant que" Prolog.

3) Définition et unification des Gop's.

4) Interprétation finie des Gop's.

5) Terminaison et complexité des "Tant que" Prolog.

6) Exemples.

1 Introduction

1.1 Contenu du chapitre

Ce chapitre est un résumé du doctorat de Philippe Devienne [DEV 87], résultat d'un travail sur l'étude de l'arrêt et de la complexité en temps de calcul d'une classe simple de programmes Prolog. En fait il s'agit de la première partie d'une étude plus générale de la complexité des programmes Prolog, dont les chapitres suivant constituent la deuxième partie.

Cette présentation de précédents travaux nous permettra par ailleurs de fixer les notations ainsi que d'introduire le vocabulaire nécessaire à la bonne compréhension de la suite de ce document.

Nous allons dans un premier temps introduire la classe des programmes concernés par ces premiers résultats, puis présenter la notion de Graphes Orientés Pondérés, utile à l'étude du comportement récursif des programmes. Grâce à cette notion ainsi qu'aux opérations syntaxiques qui lui sont rattachées (substitutions, unification, interprétations finies), nous pourrons énoncer les résultats de décidabilité et de complexité des programmes étudiés. Pour terminer, nous illustrerons à l'aide d'exemples ces résultats.

1.2 Notations , formalisme, définitions

1.2.1 Alphabets de lettres et mots

X est un **alphabet** fini de lettre (ex : X = {a,b,c,d,e,f}), l'ensemble des mots ou **monoïde libre**, noté X^*, engendré par X est l'ensemble de tous les mots finis que l'on peut écrire à l'aide des lettres de X, ε désigne le mot vide $\varepsilon \in X^*$.

On note mm' le mot composé du mot m suivit du mot m'. Si $m \in X^*$ et $m' \in X^*$ alors $mm' \in X^*$.

Nous étendons la définition de X^* aux mots infinis. Ce nouvel ensemble s'appelle X^ω. Le mot infini composé uniquement de la lettre a sera noté a^ω. Le mot composé d'une infinité de fois le mot m est noté m^ω.

Soit un mot m de X^*, on définit l'ensemble des **préfixes** ou **facteurs gauches** de m de la manière suivante :

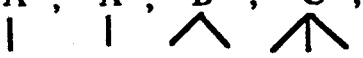
$$FG(m) = \{ m' \in X^* / \exists m'' \in X^* \quad m = m'm'' \}$$

1.2.2 Alphabets gradués, arbres et substitutions

Soit Σ est un **alphabet fini gradué** (ensemble fini de symboles munis d'une arité entière π, ex: { A,A',B,C,F } $\pi(A)=1, \pi(A')=1, \pi(B)=2, \pi(C)=3, \pi(F)=0$).

On note en général :

$$\Sigma = \{ \quad A \quad , \quad A' \quad , \quad B \quad , \quad C \quad , \quad F \quad \}$$



Soit V un ensemble dénombrable de variables (les variables sont d'arité nulle).

Par convention on note $V = \{ x, y, z, t, x_0, x_1, \ldots \}$

N est l'ensemble des entiers naturels, Z celui des entiers relatifs.

- $M^{\infty}(\Sigma, V)$ désigne l'ensemble des arbres finis ou infinis construits sur l'alphabet $\Sigma \cup V$. Les éléments de $M^{\infty}(\Sigma, V)$ seront aussi appelés **termes.**

Un exemple de terme est :

Une autre notation sous forme parenthésée pour le même arbre est :

$$C(B(A'(x), A(y_0)) , F, A(F)).$$

Chaque occurrence d'un symbole dans l'arbre représente un **nœud** ou **sommet** de l'arbre. Le nœud d'étiquette C est, dans cet exemple, le nœud **racine** de l'arbre. Chaque nœud a autant de **successeurs** (ou **fils**) que son arité (un nœud est le **père** de ses successeurs). Les nœuds d'étiquette x , y_0 et F sont les **feuilles** de l'arbre (d'arité nulle).

Chaque nœud de l'arbre est repéré de façon non ambiguë par son **chemin**, mot de X^* ou $X = \{ 1, 2, .. , \max(\pi(L)) / L \in \Sigma \}$ X est un alphabet composé de lettres étant les nombres de 1 au maximum des arités des symboles de Σ.

Le chemin de la racine est, par convention, le mot vide ε. Le chemin de son premier successeur (dans l'exemple le nœud d'étiquette B) est le mot 1. Le chemin du mot d'étiquette A' est 11. celui d'étiquette y_0 est 121 ... etc. Les chemins parcourent les **branches** de l'arbre, une branche est une suite de nœuds allant de la racine à une feuille.

Plus formellement :

- Un **arbre** T de $M^\infty(\Sigma, V)$ est une fonction partielle de l'ensemble des mots de (mot de $(N-\{0\})^*$), appelés chemins, dans $(\Sigma \cup V)$ telle que :

i) Son **domaine** $Dom(T) = \{ m \in (N-\{0\})^* / T(m) \text{ est défini } \}$ est clos par préfixe i.e. si $mm' \in Dom(T)$ alors $m \in Dom(T)$.

Dans l'exemple $Dom(T) = \{\varepsilon,1,11,111,12,121,2,3,31\}$

ii) $\forall m \in Dom(T)$ $mi \in Dom(T) \Leftrightarrow 1 \leq i \leq \pi(T(m))$.

On note T.m le sous-arbre de T issu du nœud T(m) pour $m \in Dom(T)$, $Dom(T.m) = \{ m' \in (N-\{0\})^* / T.mm' \in Dom(T) \}$.

Dans l'exemple $T.31 = T(31) = F$

Un chemin m d'un arbre T est dit **maximal** dans T si et seulement si T.m est une feuille.

On note $| m |$ la **longueur** du chemin m et $\| T \|$ la **taille** de l'arbre T, c'est à dire la longueur du plus long chemin de T.

- **Var(T)** désigne l'ensemble des variables étiquettes de nœuds de T.

Dans l'exemple : $Var(T) = \{ x, y_0 \}$

De la même manière que dans les mots, on généralise cette notion d'arbre aux arbres infinis en considérant les chemins sur Xw au lieu de X*, ce qui autorise les branches infinies. Il est bien entendu qu'il est difficile de dessiner un arbre infini (le nombre de symboles à écrire est en général infini), nous ne donnerons, pour l'instant, pas d'exemple.

Une branche d'arbre est dite **close**, si et seulement si elle ne se termine pas par une variable.

Un arbre est **clos** si et seulement si toutes ses branches sont closes.

Une **substitution** σ dans $M^{\infty}(\Sigma, V)$ est un ensemble dénombrable de couples de la forme (u,t) où

. u est une variable de V qui apparaît au plus une fois en partie gauche des couples de σ.

Exemple : $\sigma = \{ (x, A(z)), (y_0 = F) \}$ est une substitution finie.

. t est un arbre quelconque de $M^{\infty}(\Sigma, V)$.

on notera $Dom(\sigma) = \{ u \in V / (u,t) \in \sigma \}$.

Ex : $Dom(\sigma) = \{ x, y_0 \}$

$Var(\sigma) = \cup (Var(t) / (u,t) \in \sigma)$.

Ex : $Var(\sigma) = \{ z \}$

En fait, il s'agit de substitutions du **premier ordre** (les variables substituées n'apparaissent qu'en feuille).

Le composé $\sigma(T)$ (noté aussi $T.\sigma$) d'un arbre par une substitution, est l'arbre T dont toutes les occurrences des $x \in Dom(\sigma)$ ont été remplacées par l'arbre t_x associé dans σ ($(x, t_x) \in \sigma$).

La substitution σ de l'exemple, appliquée sur l'arbre T de l'exemple donne le nouvel arbre $T.\sigma$:

Une substitution σ est dite **idempotente**, si elle vérifie $Dom(\sigma) \cap Var(\sigma) = \emptyset$.

On note $\sigma\downarrow_E$ la **projection** de σ sur l'ensemble de variables E (ou l'ensemble des variables du terme E, si E est un terme).

On définit, par extension, le **composé** de deux substitutions σ et σ' comme suit :

$\sigma.\sigma' = \{ (u,\sigma'(t)) / (u,t) \in \sigma \} \cup \sigma\downarrow_{V\text{-}Dom(\sigma)}$.

On a $(T.\sigma).\sigma' = T.(\sigma.\sigma')$

Une **permutation** est une substitution bijective sur l'ensemble des variables.

1.2.3 Relation d'ordre

Soient T et $T' \in M^{\infty}(\Sigma, V)$.

on définit la **relation de préordre** classique sur les termes de la façon suivante :

$T \leq T' \Leftrightarrow \exists \sigma$ tel que $\sigma(T) = T'$.

On dira par la suite que T est plus général que T', ou qu'il contient moins d'information que T'.

La relation d'ordre induit la **relation d'équivalence** suivante :

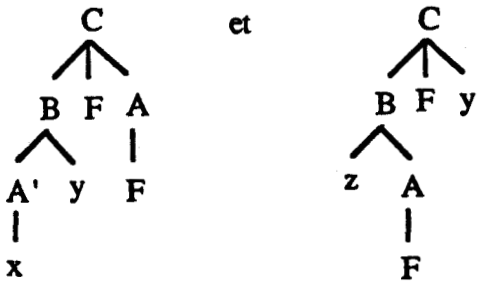
$T \equiv T' \Leftrightarrow T \leq T'$ et $T' \leq T$.

On dira par la suite que T et T' sont égaux aux noms des variables près.

Ces deux relations sont étendues aux substitutions de même domaine de manière classique : $\sigma \leq \sigma'$ si et seulement si pour toute variable $x \in Dom(\sigma)=Dom(\sigma')$, si $(x, t) \in \sigma$ et $(x', t') \in \sigma'$ alors $t \leq t'$.

1.2.4 Unification

Deux termes T et T' sont dits **unifiables** si et seulement si il existe une substitution s telle que $T.\sigma = T'.\sigma$

Les deux termes :  et

sont unifiables sans renommage des variables, avec la substitution :

$$\sigma = \{ (x, x), (y, A(F)), (z, A'(x)) \}$$

La plus petite substitution σ (au sens de l'ordre défini plus haut) s'appelle le **plus général unificateur** (most general unifier ou mgu) de T et T' et sera noté **mgu(T, T')**.

Dans l'exemple ci-dessus, la substitution σ est le mgu des deux arbres unifiés.

Deux termes T et T' sont **unifiables avec renommage** (weak unification [EDE 1985]) des variables si et seulement si il existe deux substitutions σ et σ' telles que $T.\sigma \equiv T'.\sigma'$

Les deux termes :  et

Sont unifiables avec renommage des variables par les substitution σ et σ' :

$$\sigma = \{ (x, A(x)) \} \quad \text{et} \quad \sigma' = \{ (x, x) \}$$

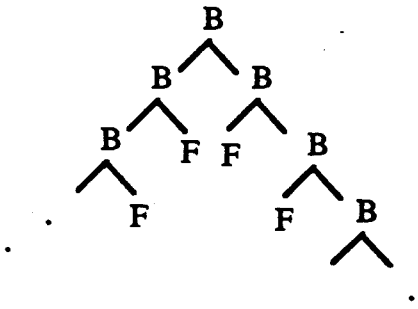
Tout se passe comme si l'unification des deux termes se fait après avoir renommé les variables de telle façon que $Var(T) \cap Var(T') = \emptyset$, d'où le nom donné à cette unification.

En revanche, ces termes ne sont pas unifiables sans renommage des variables. En effet, il n'existe pas de substitution dans les arbres finis rendant les deux termes égaux.

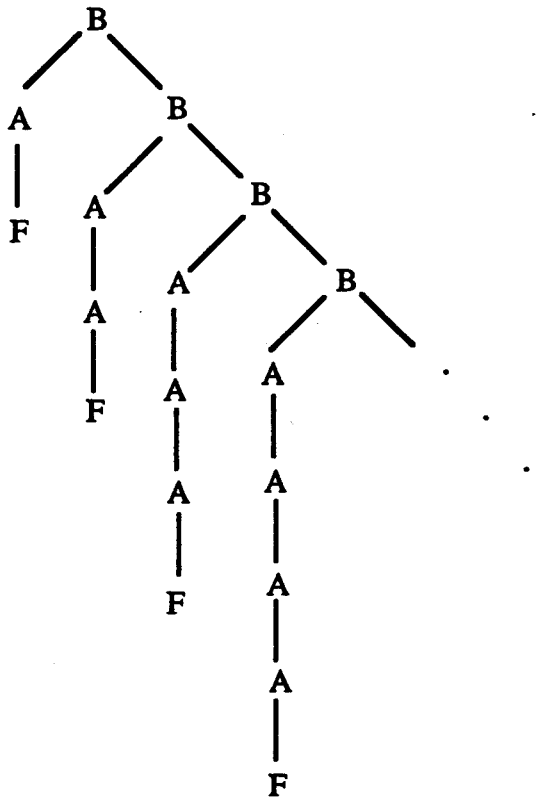
1.2.5 Graphes

La notion de graphe est un moyen de dessiner un sous-ensemble des arbres finis ou infinis [COU 1983] [FAG 1983]: les **arbres rationnels** ou **réguliers**. Ce sont tous les arbres n'ayant qu'un nombre fini de sous-arbres distincts. Les arbres finis sont de toute évidence rationnels.

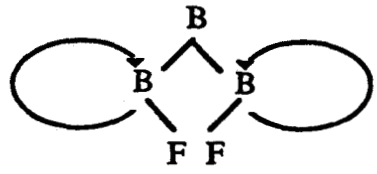
Par exemple l'arbre T_1 suivant est rationnel :



en revanche, l'arbre T_2 ne l'est pas, il contient une infinité de sous-arbres distincts :



Le graphe représentant T_1 est :



En plus des notions de chemin, de racine (qui doit être explicite contrairement au cas des arbres), d'étiquette et de successeurs, le graphe a besoin, pour être défini, de la notion d'emplacement. En effet, dans les arbres , un nœud est parfaitement défini par son chemin, dans les graphes ce n'est plus le cas. La notion de nœud doit être mieux définie, elle correspond à une **place** dans le graphe, un nœud pouvant être le fils de plusieurs autres nœuds. (Dans l'exemple, le nœud de chemin 1 est à la fois fils de la racine, et son propre fils).

Un graphe sur $\Sigma \cup V$ est donc défini formellement par un quadruplet : $G = < X , Lab , Succ, Rac >$ où :

. X est un ensemble fini de sommets ou places.

. Lab est une fonction de X dans $\Sigma \cup V$ associant à chaque nœud un étiquette unique.

. Succ est une fonction de $X \times N$ dans X associant à un nœud sont $k^{ième}$ successeur.

. Rac est un élément de X appelé racine.

On appelle **déplié** d'un graphe, l'arbre fini ou infini de même racine, de même fonction label et de même fonction successeur.

2 Le "TANT QUE" Prolog

Il s'agit d'une structure de programmes Prolog composés d'une seule règle récursive et d'un seul fait.

Tant que : f Prédicat (. .) → ;

r Prédicat (. .) → Prédicat (. .) ;

Prédicat (. .) ?

Exemple : f Entier (Zéro) → ;

r Entier (Succ (x)) → Entier (x) ;

Entier (Succ(Succ(Zéro))) ?

signifiant que : f : 0 est un entier,

r : x+1 est un entier si x est un entier,

Question : Deux est-il un entier ?

La syntaxe choisie est proche de celle de Prolog II , avec les variables écrites en minuscules, et les symboles de fonctions commençant toujours par une majuscule.

Le fonctionnement d'un tel programme est, comme son nom l'indique, semblable à celui d'un "tant que" de langage procédural. En effet, tant que le sous-but courant n'est pas unifiable avec le fait, on applique l'unique règle récursive. Ce processus pouvant être infini.

Le choix de la flèche pour l'écriture des règles n'est pas innocent. Le programme du type TANT QUE peut être assimilé à un système de réécriture d'arbres finis (Prolog avec le test d'occurrence) dans lequel on ne s'autoriserait que les réécritures "en tête", c'est à dire à partir de la racine de l'arbre réécrit.

f : $\alpha \rightarrow$

r : $\beta \rightarrow \gamma$

T ?

T,α,β et γ étant des arbres définis sur un alphabet fini gradué Σ, et un ensemble de variables d'arité nulle V. L'alphabet Σ correspond à l'ensemble des symboles de fonctions qu'utilise le programme Prolog, V correspond à l'ensemble infini dénombrable des variables manipulées par l'interpréteur Prolog. Nous omettons le symbole de prédicat, car il est unique.

Cette similitude nous amène à prendre les notations suivantes : Si la question T provoque n utilisations consécutives de la règle r et si le sous-but courant est, après n pas T', nous notons

$$T \vdash_{r^n} T'$$

r^n désignant le mot constitué de n fois la lettre r.

La simplicité d'un tel système n'est qu'apparente. En fait il suffit d'étudier la littérature qui est consacrée à des systèmes de réécritures du même style, pour s'en convaincre.

Par exemple, il a été montré que la propriété de terminaison (arrêt) est indécidable pour des systèmes de réécriture classiques dans les arbres ne contenant qu'une seule règle [DAU 1987]. C'est à dire qu'il est indécidable, étant donné une règle de réécriture, de savoir si oui ou non, il existe un terme clos engendrant une réécriture infinie par la règle. Cette même propriété d'indécidabilité avait été montrée précédemment pour trois règles [LIP 1977], pour deux règles [DERS 1985].

Pour parfaire les limites dans lesquelles se trouve notre travail, nous montrons dans le chapitre 4 que l'arrêt d'un programme Prolog dont la structure est très proche de celle du TANT QUE, est indécidable. En effet, il suffit d'avoir plusieurs faits et une seule règle de la forme P() → P() Q() ; pour obtenir des programmes d'une complexité arbitrairement élevée.

3 Définition et unification des GOP's

3.1 Approche intuitive

Les graphes orientés pondérés sont des outils syntaxiques d'aide à la démonstration de propriétés sur le comportement récursif d'une règle. Intuitivement, ils nous renseignent sur la façon dont l'information est consommée ou générée par l'application répétée d'une même règle.

L'application d'une règle Prolog sous-entend, pour éviter le problème de capture de variables, le renommage de toutes les variables de la règle. Ceci est fait en général en indiçant les symboles de variables avec un entier indiquant le numéro d'application de la règle (1 pour une première application, 2 pour la deuxième, ...etc..). Reprenons l'exemple du paragraphe précédent.

Soit le but Entier (Succ(Succ(Zéro))) ?

a - on applique l'unique règle récursive avec l'indice 1 aux variables :

Entier (Succ (x_1)) $\rightarrow$ Entier (x_1) ;

ce qui donne la substitution (x_1 = Succ (Zéro)) ;

Le sous-but courant est alors Entier (Succ (zéro)) ?

b - on applique l'unique règle récursive avec l'indice 2 aux variables :

Entier (Succ (x_2)) $\rightarrow$ Entier (x_2) ;

ce qui donne la substitution (x_2 = Zéro) ;

c - le sous-but courant Entier (Zéro) ? s'unifie avec le fait.

Si on tente de généraliser ce comportement, on s'aperçoit que le sous but au bout du $i^{ème}$ pas Entier (x_i) ? est de la forme Entier(Succ(x_{i+1})) avec Entier (x_{i+1}) ? sous-but au $(i+1)^{ème}$ pas.

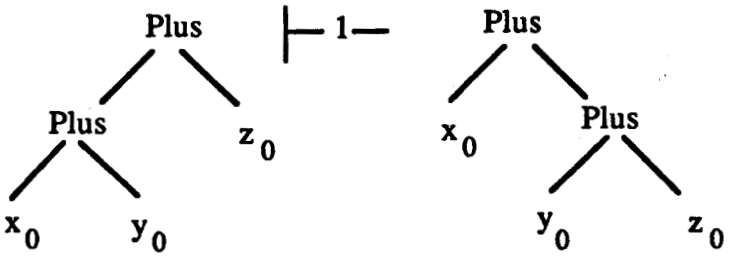
Par la suite, nous allons utiliser la notion de **substitutions élémentaires** (binding [LAS 1987]) . Il s'agit d'un ensemble de couples (variable, terme) tels que les termes soient sous la forme la plus élémentaire possible, au sens des substitutions. C'est à dire que dans les termes, aucune substitution n'a été effectuée. Une présentation formelle de ces objets sera faite par la suite. L'exemple qui suit, illustre cette notion.

Prenons l'exemple de l'associativité de l'addition. Considérons la règle :

1 : Assoc (Plus (Plus (x , y) , z) → Assoc (Plus (x, Plus (y , z)) ;

De la même manière que pour l'exemple précédent, nous allons numéroter les variables à l'aide d'indices égaux au nombre de pas successifs déjà réalisés avec la règle.

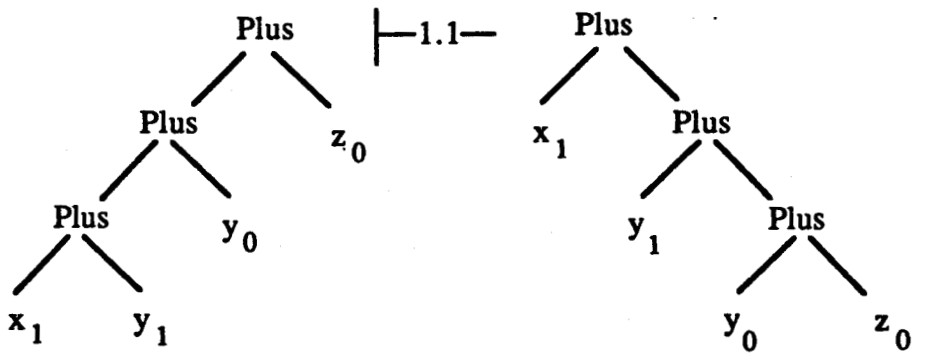
La première application nous donne :



L'ensemble des substitutions élémentaires est alors:

{(x_0= libre), (y_0=libre), (z_0=libre)}

Deux applications successives donnent :

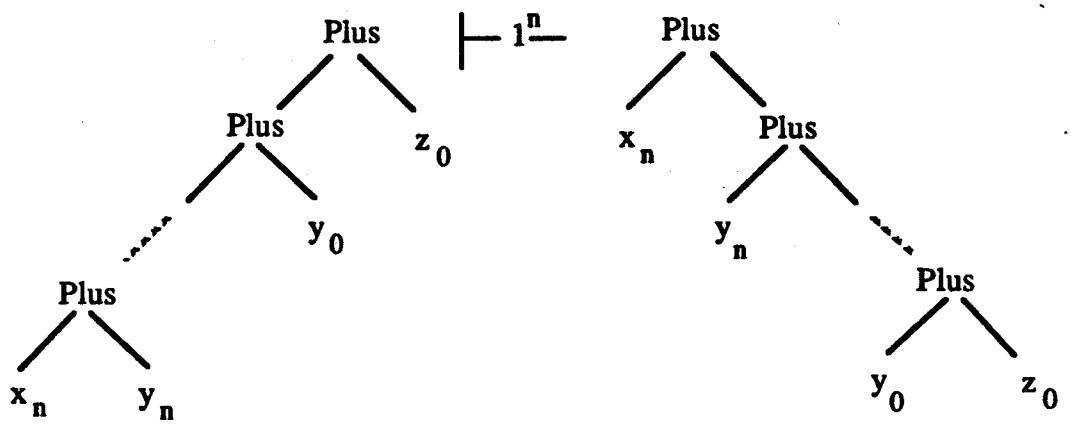


L'ensemble des substitutions élémentaires est alors:

{(x_0= Plus(x_1,y_1)), (y_0=libre), (z_0=libre),(x_1= libre), (y_1=libre), (z_1=Plus(y_0,z_0))}

Remarque : Dans l'ensemble des substitutions élémentaires, toutes les variables intermédiaires apparaissent, et aucune application de substitution n'est effectuée sur les termes. On retrouve donc pour chaque symbole de variable tous les indices de 0 à i, i étant le nombre d'applications de la règle moins un.

n applications successives donnent :



L'ensemble des substitutions élémentaires est alors:

$\{(x_0 = \text{Plus}(x_1,y_1)), (y_0=\text{libre}), (z_0=\text{libre}),$

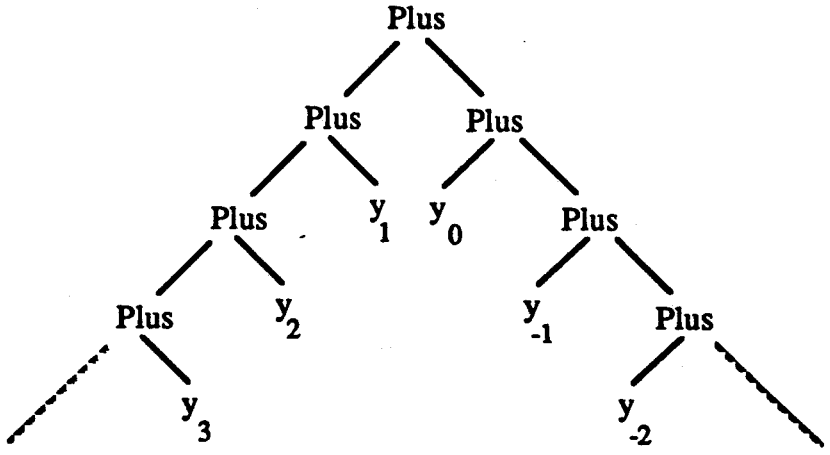
$(x_1 = \text{Plus}(x_2,y_2)), (y_1=\text{libre}), (z_1=\text{Plus}(x_0,y_0))$

$(x_2 = \text{Plus}(x_3,y_3)), (y_2=\text{libre}), (z_2=\text{Plus}(x_1,y_1))$

..

$(x_n = \text{Libre}), (y_n=\text{libre}), (z_n=\text{Plus}(y_{n-1},z_{n-1}))\}$

Si on généralise cette réécriture en supposant un comportement infini, c'est à dire que la règle a été utilisée une infinité de fois et sera encore utilisée une infinité de fois, le terme manipulé est alors de la forme: (à un décalage des indices près selon l'endroit où l'on choisit de mettre l'indice 0). C'est le plus petit point fixe de la règle.

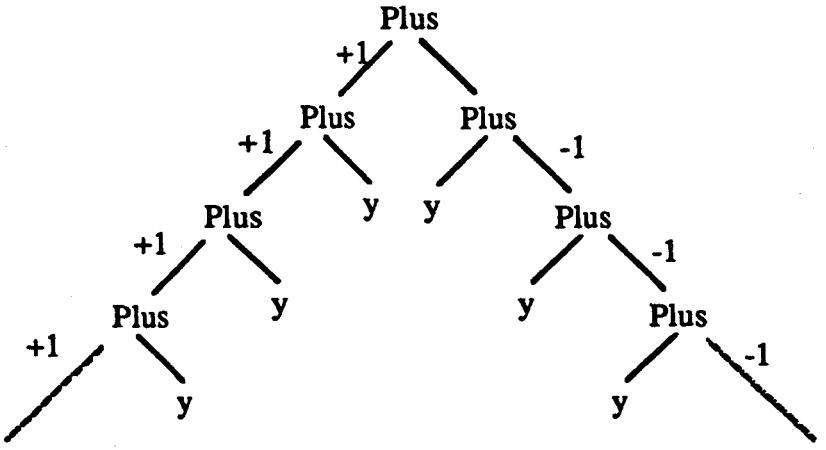


L'arbre qui précède est un arbre presque rationnel, seuls les indices des variables en font un arbre non rationnel. Or ces indices sont organisés de façon très régulière, ce qui nous fait dire que cet arbre est rationnel moyennant la définition d'une opération de décalage d'indices.

L'ensemble des substitutions élémentaires est alors:

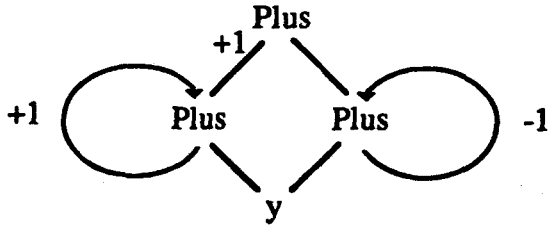
$\{(x_i = \text{Plus}(x_{i+1}, y_{i+1})), (y_i = \text{libre}), (z_i = \text{Plus}(y_{i-1}, z_{i-1}))\}$ pour tout i entier relatif.

On peut schématiser cet arbre comme suit :



L'indice d'une variable se retrouve en effectuant la somme des pondérations d'arcs depuis la racine jusqu'à la variable.

Puis en repliant les branches nous obtenons une représentation sous la forme d'un graphe de manière non ambiguë :



Nous verrons dans le paragraphe suivant une manière d'obtenir directement ce graphe à partir de la règle récursive.

L'étude de ce comportement infini nous permet de mettre en évidence le caractère uniforme des substitutions calculées lors de la résolution pour une application finie. En effet, pour un nombre de pas suffisamment grand (au pire de l'ordre de 2^n où n est le

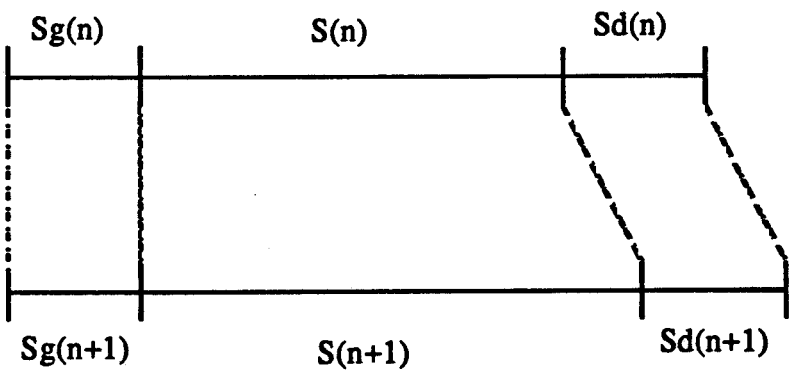
nombre de symboles constituant la règle) la substitution $\sigma(n)$ courante peut être décomposée en trois sous-systèmes distincts, que l'on note Sg(n), Sd(n) et S(n).

Sg(n), appelé "effets de bords à gauche" et représentant un ensemble de substitutions élémentaires de variables d'indices proches de zéro, est constant par rapport à n : pour toute application supplémentaire : Sg(n) = Sg(n+k).

Sd(n), appelé "effets de bords à droite", représente de façon symétrique un ensemble de substitutions élémentaires de variables d'indices proches de n. Cette partie est constante relativement à n pour toute application supplémentaire : Sd(n+k) = Δ(Sd(n),k) où Δ est un opérateur de décalage d'indice, ayant comme premier argument un élément de n'importe quel type. C'est à dire que pour un objet O quelconque, Δ(O, k) désigne l'objet O dans lequel chaque variable a son indice augmenté de k.

S(n) est l'ensemble des substitutions non contenues dans Sg(n) et Sd(n). La forme de ces substitutions est très particulière. Si (x_i = t) et (x_{i+j}=t') sont deux substitutions élémentaires de S(n), alors t' = Δ(t , j). En fait S(n) est un sous ensemble des substitutions élémentaires lors de la réécriture infinie. On a donc bien mis en évidence un phénomène uniforme lors de la réécriture finie, ce phénomène étant "encadré" par des effets de bords constants.

On peut illustrer ce phénomène par le schéma suivant :



Sg(n) et Sd(n) sont constants, S(n) est la partie uniforme des substitutions, cette partie étant extensible.

Il est difficile de se faire une idée du comportement d'une règle récursive si l'on se contente d'observer le système dans sa généralité, les effets de bords rendent en général le système trop complexe. D'où l'idée d'isoler le sous-système S(n) en regardant le résultat d'une réécriture infinie telle qu'elle est décrite plus haut. Les effets de bord sont alors reportés à l'infini, on obtient alors une expression claire du comportement uniforme.

Exemples : La règle suivante illustre un comportement périodique :

Commut (Plus (x , y)) → Commut (Plus (y , x)) ;

La première application nous donne les substitutions élémentaires suivantes :

$\{(x_0$= libre), (y_0=libre)$\}$

Une deuxième application donne :

$\{(x_0$= libre), (y_0=libre),(x_1= y_0), (y_1=x_0)$\}$

Une troisième application nous donne l'identité:

$\{(x_0$= libre), (y_0=libre),(x_1= y_0), (y_1=x_0), (x_2= x_0), (y_2=y_0)$\}$

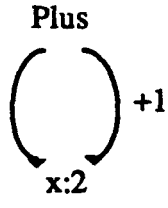
Une application de n réécritures donne:

$\{(x_0$= libre), (y_0=libre),(x_1= y_0), (y_1=x_0), (x_i= x_{i-2}), (y_i=y_{i-2})$\}$ pour tout i de 2 à n.

Le résultat de l'application infinie est dans la substitution suivante :

$\{(x_i$= x_{i-2}), (y_i=y_{i-2})$\}$

ce que l'on retrouve dans le graphe :



Où x:2 désigne que la variable x est indicée modulo 2 . Ce graphe nous indique donc que la règle est périodique, car toutes les variables (dans cet exemple il n'y en a qu'une) ont des indices modulo 2. On dira que le graphe est de congruence 2.

Un troisième exemple concerne le test d'occurrence lors de l'utilisation récursive d'une règle. En effet, l'étude du comportement infini de la règle nous renseigne aussi sur l'existence ou non d'une occurrence liée à la règle seule :

Soit la règle :

Occur (B (x , x)) → Occur (B (A (x) , x)) ;

Cette règle engendre de toute évidence une occurrence lors de deux applications consécutives. Nous allons voir que le graphe calculé sur le comportement infini de la règle va nous le dire .

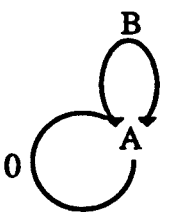
une application :

$\{(x_0 = \text{libre})\}$

Deux applications :

$\{(x_0 = A^{\omega}), (x_1 = A^{\omega})\}$

Le graphe calculé est :



L'occurrence est ici symbolisée par l'obtention d'une pondération de boucle nulle indiquant que pour n suffisamment grand, dans la substitution S(n) (donc dans la substitution globale $\sigma(n)$), une variable sera substituée par un terme contenant la même variable au même indice.

3.2 Définition du graphe orienté pondéré

Dans ce paragraphe, nous allons présenter une approche plus formelle des objets que sont les GOP's. La définition, ainsi que la méthode de calcul, justifie l'emploie de ces objets pour comprendre le comportement d'une règle récursive, aussi compliquée soit-elle.

Définition : *Un Graphe Orienté Pondéré est la donnée d'un quadruplet (X,Lab,Succ,Clas) définissant le graphe, et d'un couple Rac = (s_0,p_0) nous donnant un point d'entrée ainsi qu'une pondération d'entrée dont nous verrons l'utilité par la suite.*

- X est l'ensemble fini des sommets (ou nœuds) du graphe.

- Lab est une fonction Label associant à chaque sommet une étiquette symbole de fonction ou variable.

- Succ est la fonction successeur du graphe, de X×N dans X×Z, associant à un nœud de X son $i^{ème}$ successeur muni de la pondération d'arc $p \in Z$.

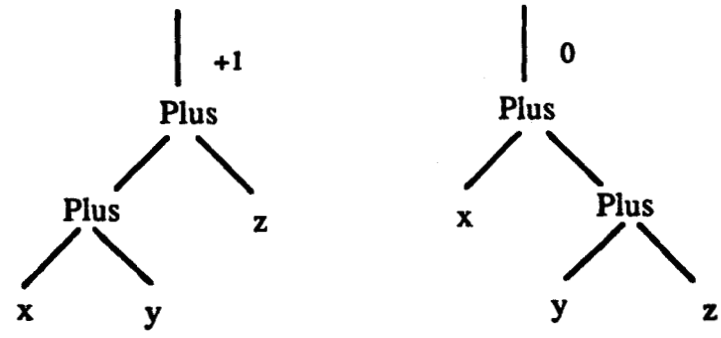
$Succ(s , i) = (s', p)$

- Clas est une fonction de X dans N qui associe à chaque nœud sa période.

- Rac est un couple (s_0,p_0) singularisant un nœud s_0 comme point d'entrée, ou racine, du graphe, et p_0 est un entier relatif, pondération en tête du graphe (d'un arc fictif menant à la racine).

3.3 Construction du graphe orienté pondéré associé à un règle récursive

Le calcul du graphe orienté pondéré associé à une règle récursive est une simulation, comme nous l'avons vu dans le paragraphe précédent, d'une réécriture infinie par une seule règle. En fait, il s'agit d'effectuer l'opération de renommage des variables de la règle par décalage d'indices, ceci est fait par une pondération en tête de la partie gauche (tête) de la règle valant 1, alors que la partie droite (corps) est munie d'une pondération en tête de 0. Les termes sont donc, dans le cas de l'associativité de l'addition :

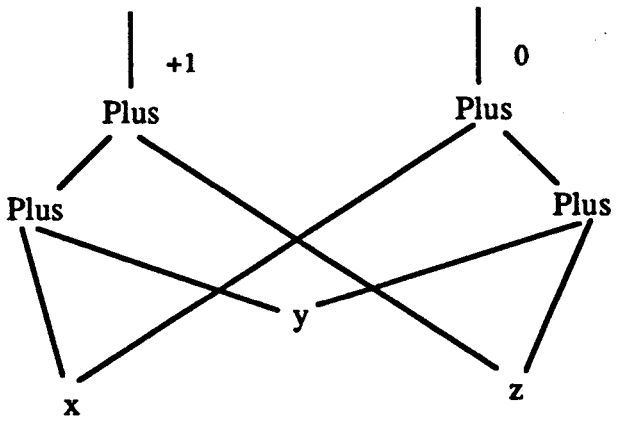


L'unification Prolog et la réécriture Prolog sont simulées simplement par une unification des deux objets obtenus, sans renommage des variables. Le calcul des indices

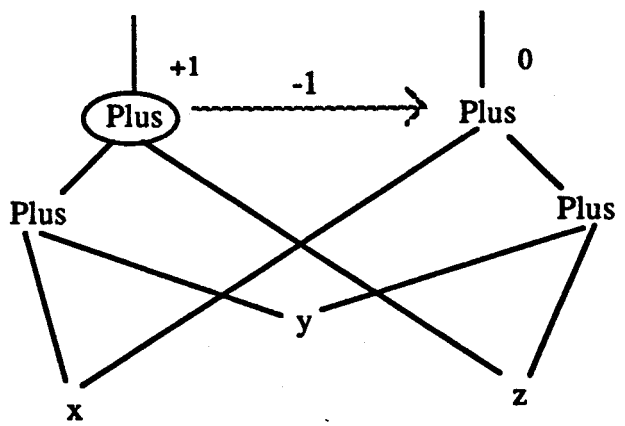
des arcs de l'unifié et que celui des congruences d'indices viennent s'ajouter à l'unification classique dans les graphes, présentée par Fages dans [FAG 1983].

Cet algorithme d'unification est fondé sur la notion d'indirection : un nœud unifié à un autre nœud est redirigé vers ce dernier. Il ne sera donc plus accessible lors d'unifications ultérieures. De plus, on met en commun les nœuds dont l'étiquette est une variable de telle sorte qu'il n'y ait pas à recopier plusieurs fois la substitution associée à une variable.

étape 1 :

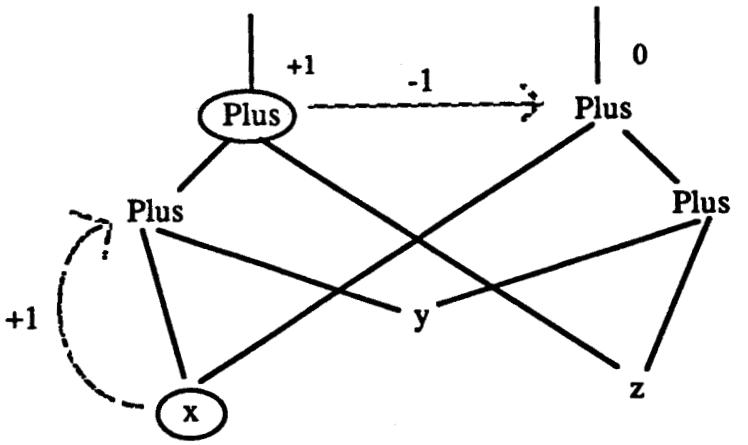


étape 2 : unification des deux racines.



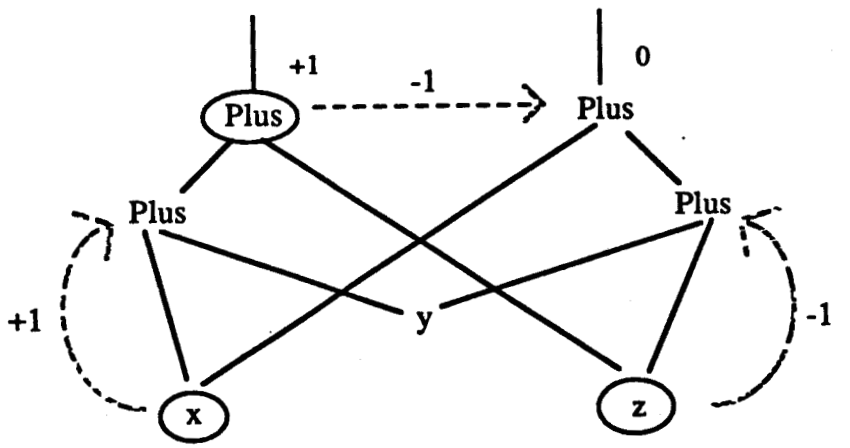
Il y a donc indirection d'une racine vers l'autre, cette indirection étant munie d'une pondération rétablissant l'égalité des pondérations cumulées que l'on entre d'un coté ou de l'autre dans le graphe. Le nœud encerclé ne sera plus accessible par la suite car il est "redirigé".

étape 3 : unification des deux fils gauche :



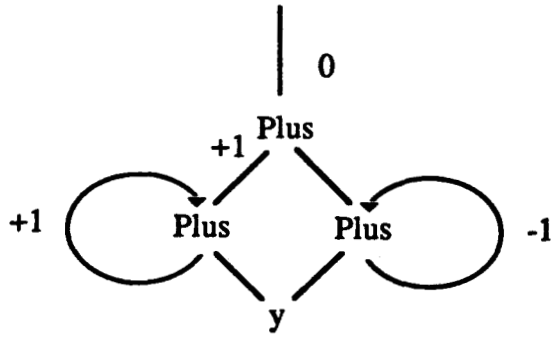
x est une variable, le nœud x est redirigé vers le nœud Plus, la pondération d'indirection est +1 pour rétablir l'équilibre entre une pondération cumulée à gauche de 1 et à droite de 0. Il y a création d'un élément de substitution : $x_i = Plus(x_{i+1}, y_{i+1})$.

étape 4 : unification des deux fils droits :



Même phénomène que dans l'étape précédente avec une pondération d'indirection de -1 car l'indirection va d'un sommet de pondération cumulée 1 dans le terme de gauche, vers un sommet de pondération cumulée de 0 dans le terme de droite.

Le GOP unifié est donc :



Nous retrouvons le schéma présenté dans le paragraphe précédent, représentant l'application infinie de la règle.

L'algorithme général d'unification des GOP's est le suivant:

Soient deux Gop's G et G' de racines respectives (s,p) et (s',p').Dans un premier temps on construit un graphe unique contenant les deux Gop's partageant leurs variables. L'algorithme modifie alors ce graphe jusqu'à ce que les Gop's soient les mêmes, ou alors il termine par un échec.

L'algorithme d'unification est proposé à la page suivante.

```
Procédure UNIFICATION ( (s,p),(s',p') )
Début
  Si (s=s') alors  % le nœud s devient de période | p-p' | %
  |     Clas(s) ← PGCD ( Clas(s), | p-p' | )
  sinon
  |     Si  ( Lab(s') = x' :  symbole de variable )
  |     |     % homogénéisation des périodes %
  |     |     Clas( s ) ← PGCD ( Clas( s ) , Clas (s'))
  |     |     % création d'une indirection de pondération (p-p') de x' vers s %
  |     |     ∀ s" ∈ X, si Succ(s",i ) = (s',p_i ) alors Succ( s",i ) ← (s,p_i+p-p')
  |     Sinon
  |     |     Si  ( Lab(s) = x :  symbole de variable )
  |     |     |     % cas symétrique du précédent %
  |     |     Sinon
  |     |     |     Si (Lab(s) = Lab(s'))
  |     |     |     |     % on crée une indirection de s' vers s avec la
  |     |     |     |     pondération (p-p') %
  |     |     |     |     % unification de chacun des successeurs de s avec les
  |     |     |     |     successeurs correspondants de s' la pondération étant
  |     |     |     |     cumulée%
  |     |     |     |     Pour i de 1 au nombre de fils de s
  |     |     |     |     |     % (s_i,p_i) i^ème fils de s, (s'_i,p'_i) celui de s' %
  |     |     |     |     |     Unification ( ( s_i, p+pi ) , ( s'_i , p'+p'i ) )
  |     |     |     |     Fin Pour
  |     |     |     sinon Echec
  |     |     |     Fsi
  |     |     Fsi
  |     Fsi
  Fsi
FIN
```

L'unification étant faite, il faut "homogénéiser" la fonction Clas, sachant qu'un nœud doit avoir une période qui divise celle de chacun de ses pères.

En effet, une période sur un nœud du graphe représente un phénomène cyclique sur l'ensemble du sous-graphe issu de ce nœud. C'est à dire que tous les nœuds fils vont avoir la même période (ou une période diviseur) que le père. Un nœud appartenant à deux

sous-graphes distincts doit donc avoir simultanément deux périodes. Il est donc d'une période PGCD des deux périodes des sous-graphes.

Cette opération, que nous ne détaillerons pas, est effectuée par parcours successifs du graphe.

3.3 Problème du test d'occurrence

Définition : *On appelle **cycle** d'un graphe orienté pondéré, tout chemin non vide du graphe menant d'un nœud quelconque à lui même.*

Définition : *On appelle **pondération cumulée locale** d'un chemin m d'un gop, la somme des pondérations rencontrées le long du chemin m.*

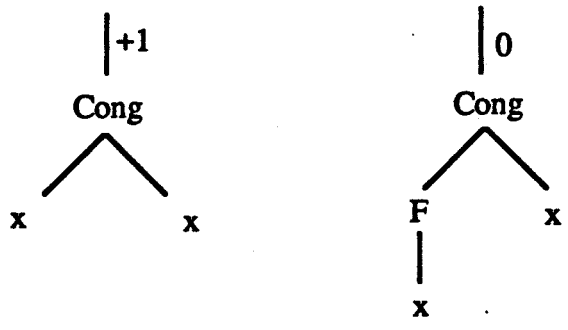
Propriété : Un Gop vérifie la propriété de **restriction sur les pondérations de boucles (RPB)** si et seulement si aucun cycle n'a une pondération cumulée locale nulle.

Théorème : [DEV 1987] *Une règle récursive Prolog dont le gop ne vérifie pas la condition RPB ne pourra pas engendrer une résolution infinie car il y aura une occurrence de variable en moins de 2^n pas récursifs (n est le nombre d'occurrence de symboles de fonctions et de variables constituant la règle).*

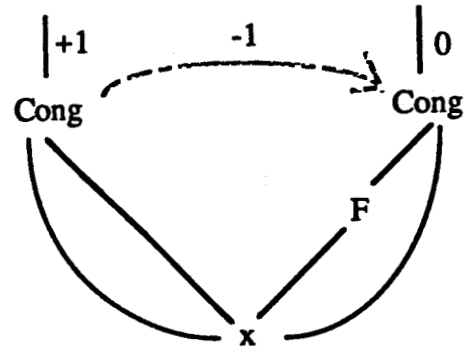
Exemple : L'exemple suivant illustre la présence d'un nœud de période 1

soit la règle : Cong (x , x) → Cong (F(x) , x) ;

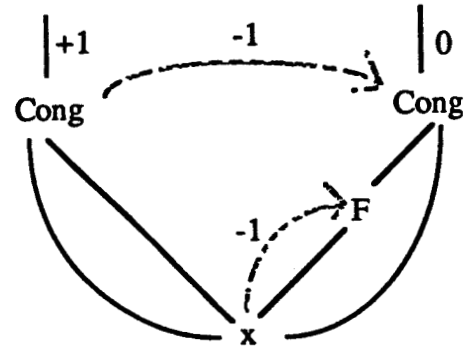
étape 1 : pondérations en tête



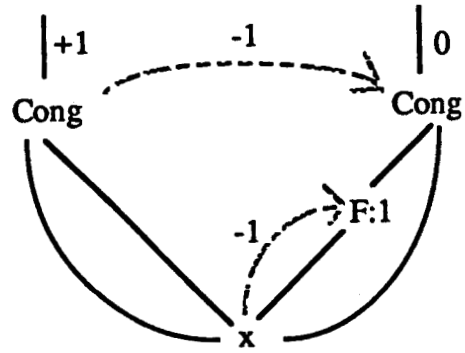
étape 2 : mise en commun des variables et unification des racines



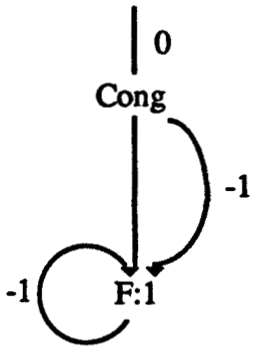
étape 3 : unification des fils gauches



étape 4 : Unification des fils droits : Dans le terme gauche, x est redirigé vers F avec la pondération cumulée de +1-1 = 0. Dans le terme droit x est redirigé vers F avec la pondération cumulée de 0-1 = -1. L'unification est donc des deux côtés sur le même nœud, d'une part avec la pondération 0 d'autre part avec -1. Ce qui donne une période de 1 égale à la valeur absolue de la différence des pondérations: ($|0-1|$).



Le Gop unifié est donc :



Cette **périodicité d'un nœud dans une boucle** nous indique qu'au cours de la résolution prolog, dans le système S(n) associé à la règle pour n applications, n étant assez grand, il y aura une variable $x_{i \text{ modulo } 1}$ soit x_0 substituée par $F(x_{(i-1) \text{ modulo } 1})$ soit $F(x_0)$. Il y aura donc **occurrence.**

En fait, le graphe associé à une règle récursive représente très fidèlement le système S(n) de la règle pour n assez grand (le plus petit n est facilement calculable en fonction de la règle : cf chap 6). Ce qui nous permet d'obtenir les résultats décrits dans le paragraphe 5 de ce chapitre.

4 Interprétation finie des GOP's

4.1 La réécriture finie Prolog

Soit la règle récursive $r : P(\beta) \rightarrow P(\gamma)$;

Nous allons noter $T_{n,m}$ le $n^{ième}$ sous-but d'une suite de m+1 sous-buts "les plus généraux possibles" contenant toutes les contraintes dues à la réécriture. La suite est telle que :

$$T_{0,m} \vdash_r^n T_{n,m} \vdash_r^{m-n} T_{m,m}$$

A la question P(x) avec le fait $P(x) \rightarrow$; (aucune contrainte ne peut alors venir du fait ou de la question), le programme donnera la solution $(x = T_{0,m})$ au bout de m pas, le dernier élément unifié avec le fait étant $T_{m,m}$.

Exemple : Reprenons le programme de l'associativité de l'addition :

Le programme que l'on étudie, pour n'avoir aucune contrainte sur les termes manipulés en dehors de celles exercées par la règle seule, est le suivant :

f : Assoc (x) → ;

r : Assoc (Plus (Plus (x , y) , z) → Assoc (Plus (x, Plus (y , z)) ;

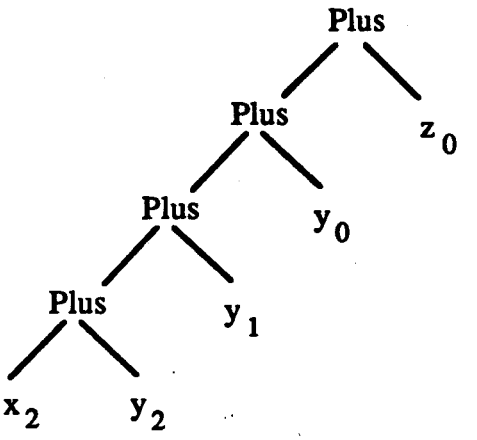
question : Assoc (x) ?

Le plus petit terme qui se réécrit 3 fois par la règle est évidemment Assoc(x). Mais ce terme doit vérifier la contrainte, au bout de 3 pas:

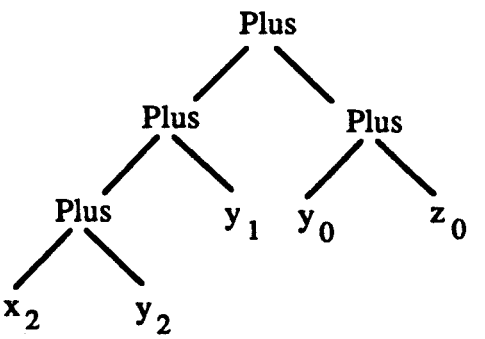
$x = Plus(Plus(Plus(Plus(x_2,y_2),y_1),y_0),z_0)$

On obtient donc la suite des termes formant la réécriture en 3 pas sans contrainte supplémentaire sur leurs variables:

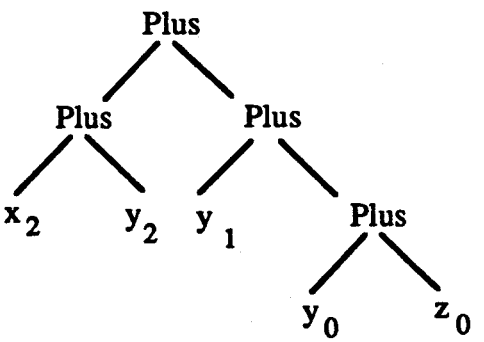
$T_{0,3}$ = Plus(Plus(Plus(Plus(x_2,y_2),y_1),y_0),z_0)



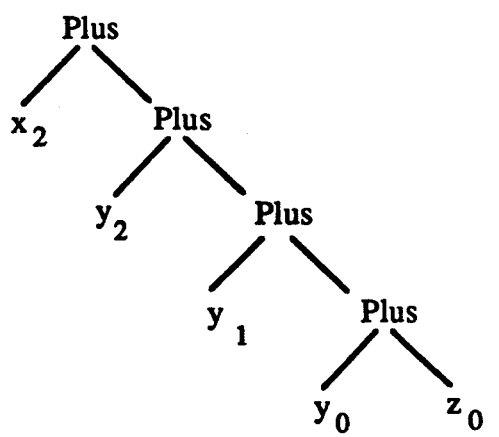
$T_{1,3}$ = Plus(Plus(Plus(x_2,y_2),y_1),Plus(y_0,z_0))



$T_{2,3}$ = Plus(Plus(x_2,y_2),Plus(y_1,Plus(y_0,z_0)))



$T_{3,3}$ = Plus(x_2,Plus(y_2,Plus(y_1,Plus(y_0,z_0))))



Nous avons donc bien les plus petits termes tels que :

$$T_{0,3} \vdash_r T_{1,3} \vdash_r T_{2,3} \vdash_r T_{3,3}$$

On appelera par la suite $\Sigma(n)$ la substitution calculée par l'interpréteur Prolog dans les conditions énoncées précédemment en n réécritures récursives.

Soit β_i le terme β où les variables ont été munies d'indices tous égaux à i. De la même manière, γ_i est le terme γ où les variables ont été munies d'indices tous égaux à i.

Théorème [DEV 1987] : $T_{n,m} = \beta_n . \Sigma(m) = \gamma_{n-1} . \Sigma(m)$

En effet, supposons que lors de la première utilisation de la règle, on indice les variables avec 0, la règle est donc "$P(\beta_0) \rightarrow P(\gamma_0)$;". A chaque utilisation, nous incrémentons les indices pour renommer les variables. C'est à dire qu'à la $n^{ième}$ utilisation, la règle s'écrit : "$P(\beta_n) \rightarrow P(\gamma_n)$;" .

Les différentes unifications effectuées par l'interpréteur Prolog sont :

$$\beta_1 \vee \gamma_0 \ , \beta_2 \vee \gamma_1 \ , \beta_3 \vee \gamma_2 , .. \ , \beta_m \vee \gamma_{m-1}$$

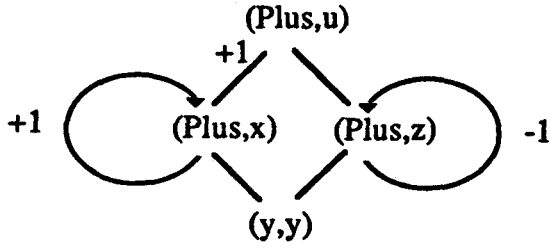
La réécriture complète est donc déduite de ces unifications locales en considérant qu'elles n'aboutissent qu'à une seule substitution commune : $\Sigma(m)$. Le terme réécrit après n pas est donc $\gamma_{n-1}.\Sigma(m)$ qui s'unifie avec β_n sans aucune contrainte (substitution) nouvelle, puisqu'elles sont déjà contenues dans $\Sigma(m)$. C'est donc $\beta_n.\Sigma(m)$.

4.2 Interprétation des Gop's sur un intervalle fini

L'interprétation des Gop's sur un intervalle fini est facile à comprendre, mais en revanche un peu plus difficile à mettre en œuvre. En effet, il s'agit de déplier l'arbre que le gop représente, sans que les pondérations cumulées depuis la racine ne sortent d'un intervalle donné. Pour cela il faut pouvoir couper une branche à n'importe quel endroit. Or, une branche coupée doit avoir en feuille un symbole de variable. Il est donc nécessaire de compléter l'étiquette de chaque nœud du Gop par un symbole de variable n'apparaissant qu'une seule fois dans le Gop. Lorsque l'on déplie une branche, si un nœud à un fils tel que la pondération cumulée vers ce fils sort de l'intervalle d'interprétation, alors la sous-branche est remplacée par la variable associée au nœud fils munie d'un indice égal au cumul des pondérations du chemin allant de la racine au fils.

Exemple : l'interprétation du gop G sur l'intervalle [0,4] noté $I_{[0,4]}(G)$ est une substitution:

soit G :



On peut remarquer que l'on retrouve les variables x et z étiquetant les nœuds qui leurs sont associés à la fois dans le terme de gauche : β, et dans le terme de droite γ. Ce n'est pas un hasard, car cela va nous permettre de retrouver, en terme de substitutions, le système S(n) associé à la règle.

La substitution associée est $I_{[0,4]}(G)$, elle est égale à :

{

$x_0 = Plus(Plus(Plus(Plus(Plus(x_5,y_4),y_3),y_2),y_1),y_0)$

$x_1 = Plus(Plus(Plus(Plus(x_5,y_4),y_3),y_2),y_1)$

$x_2 = Plus(Plus(Plus(x_5,y_4),y_3),y_2)$

$x_3 = Plus(Plus(x_5,y_4),y_3)$

$x_4 = Plus(x_5,y_4)$

$y_0=y_1=y_2=y_3=y_4=$ libre

$z_0 = Plus(y_0,z_{-1})$

$z_1 = Plus(y_1,Plus(y_0,z_{-1}))$

$z_2 = Plus(y_2,Plus(y_1,Plus(y_0,z_{-1})))$

$z_3 = Plus(y_3,Plus(y_2,Plus(y_1,Plus(y_0,z_{-1}))))$

$z_4 = Plus(y_4,Plus(y_3,Plus(y_2,Plus(y_1,Plus(y_0,z_{-1})))))$

}

elle est déduite du gop par dépliage des sous-arbres issus des nœuds dont les symboles sont les différentes variables, avec toutes les pondérations initiales appartenant à l'intervalle d'interprétation.

Théorème [DEV 1987] : *Soit G le gop associé à la règle "$P(\beta) \rightarrow P(\gamma)$;", et $\Sigma(n)$ la substitution minimale pour n pas récursifs telle que définie précédemment.*

Il existe trois entiers positifs a , b et A tels que pour tout $n > A$:

$$I_{[a,n-b]}(G) \leq \Sigma(n) \leq I_{[-a,n+b]}(G)$$

Où $\leq$ est l'ordre classique sur les substitutions.

Ce théorème est central dans le travail réalisé sur l'étude d'une règle récursive unique. En effet, il nous permet de prouver l'existence des deux sous-systèmes gauche et droit de $\Sigma(n)$, puis de prouver qu'ils sont constants. De plus, il nous apprend qu'une branche finie (non bouclante) dans le gop est forcément finie et de longueur inférieure dans les termes manipulés lors de la résolution. En effet, les termes manipulés sont de la forme $\beta_i.\Sigma(n)$ ou $\gamma_i.\Sigma(n)$, ces termes sont donc encadrés par deux termes issus d'une interprétation finie du gop. Donc en aucun cas, ils ne peuvent posséder une branche plus longue que la branche correspondante dans le déplié du gop.

5 Terminaison et complexité des "Tant que" Prolog

5.1 Décidabilité de l'arrêt d'une règle récursive

Théorème [DEV 1987] : *Une règle récursive simple engendre une infinité de réécritures avec application du test d'occurrence si et seulement si :*

i) Il existe un Gop associé à la règle. C'est à dire que les parties gauche et droite de la règle sont unifiables sans renommage.

ii) Le Gop associé à la règle vérifie la restriction sur les pondérations de boucles (RPB) (cf §3.2).

Une idée de la démonstration ainsi qu'une justification intuitive de ce théorème vous est proposée au paragraphe 3.2 de ce chapitre. En fait, puisque le Gop simule la réécriture infinie par la règle, son existence est nécessaire. De plus, nous savons que toutes les substitutions apparaissant dans le gop (théorème du paragraphe précédent) sont effectivement calculées, pour des indices quelconques, par la résolution Prolog et ceci en un temps fini. Or, si un chemin bouclant de pondération cumulée nulle est dans le gop, il correspond à une substitution de variable d'un indice donné par un terme où se trouve la même variable de même indice. Il y a donc occurrence, la résolution s'arrête.

La suite des résultats concerne des règles vérifiant ce théorème.

5.2 Etude de la croissance des termes réécrits

Dans ce paragraphe, nous nous intéressons plus particulièrement aux deux termes $T_{0,n}$ et $T_{n,n}$, le premier représentant la réponse à la question posée, le second étant le terme final unifié avec le fait.

Remarque : $T_{0,n} \leq T_{0,n+1}$ et $T_{n,n} \leq T_{n+1,n+1}$

En effet, $T_{0,n}$ est le plus petit terme se réécrivant n fois par la règle. $T_{0,n+1}$ se réécrit n+1 fois par la même règle donc à fortiori se réécrit n fois. Il est donc de taille supérieure ou égale à celle de $T_{0,n}$. Il en va de même pour le terme final.

Si $T_{0,n} < T_{0,n+k}$ le terme croit strictement, on dit alors qu'il s'agit d'une règle consommatrice car elle supprime à chaque utilisation (ou toutes les k utilisations) au moins un nœud de la question.

De même si $T_{n,n} < T_{n+k,n+k}$ il s'agit d'une règle productrice car elle ajoute régulièrement au moins un nœud au terme final que l'on unifie avec le fait.

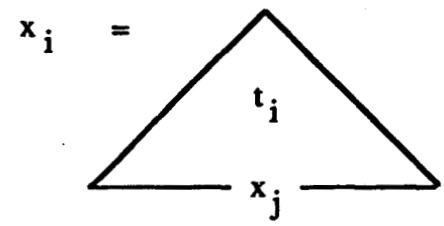
Une règle peut à la fois être consommatrice et productrice. Les branches mises en cause par la consommation étant évidemment différentes de celles mises en cause par la production.

Le but suivant est donc d'étudier de quelle façon s'effectue cette croissance (production ou consommation).

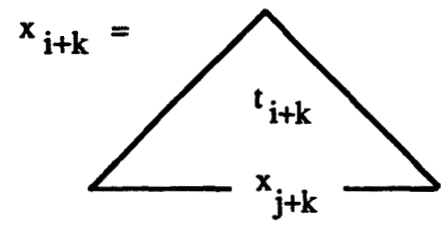
Nous étudions plus particulièrement la croissance des branches bouclantes du Gop, car elles sont les seules qui peuvent pousser lors de la résolution après un nombre de pas constant ne dépendant que de la règle (ce nombre peut être calculé lors de la construction du gop associé à la règle).

Nous savons que la substitution générale au bout du $n^{ième}$ pas, pour n assez grand, se décompose en trois sous-substitutions. Deux sous-substitutions de tailles constantes (les effets de bords à gauche et à droite), et une substitution S(n) qui est une partie finie de la substitution infinie associée au gop. Or cette dernière a une forme tout à fait particulière, toutes les substitutions élémentaires qui la composent sont de la forme $x_i = t_i$ où t_i est égal à l'arbre t_0 (substitué à la variable x_0 dans le gop), où toutes les variables ont leur indice augmenté de i. De plus, si $x_i = t_i$ et $x_{i+k} = t_{i+k}$ sont des substitutions de S(n) alors S(n) contient $x_j = t_j$ pour tous les $i \leq j \leq i+k$. On a donc un phénomène uniforme, extensible en fonction de n.

Si



est une substitution de S(n), alors il est immédiat que



est une substitution de S(n+k);

alors pour i+k =j la hauteur du sous arbre substitué à x_i a doublé. La généralisation de ce résultat nous a donné un moyen d'obtenir de façon précise le taux de croissance linéaire de chaque branche des termes $T_{0,n}$ et $T_{n,n}$ en fonction de n.

5.3 Décidabilité de l'arrêt du "tant que" pour des but et fait linéaires

Il s'agit, dans ce titre, de la linéarité dans le sens d'une seule occurrence de chaque variable dans les termes considérés. En effet, dans le cas contraire, nous ne pouvons pas conclure, car il y a interférence entre la croissance liée à la règle récursive seule, et la croissance liée à l'unification de termes non linéaires. Nous supposons seulement que le résultat qui suit peut s'étendre au cas de termes non linéaires, sans en avoir de preuve (conjecture).

Théorème [DEV 1987]: *Pour une règle* $r = P(\beta) \rightarrow P(\gamma)$ *; et une question* $P(t)$ *finie et linéaire, la propriété de terminaison est décidable. De plus, si la propriété est vérifiée, l'arrêt intervient en au plus* $\varphi(|t|)$ *réécritures,* φ *est une fonction linéaire, et* $|t|$ *est le nombre de nœuds de t.*

Cela vient du fait que les effets de bords sont constants, et ne dépendent que de la règle. Ce qui fait que lorsque le nombre de pas est suffisamment grand (supérieur à un entier n_0 , calculable lors de la construction du gop), il n'y a plus que le système central S(n) qui soit modifié par de nouvelles réécritures. Or S(n), nous venons de le voir, fait obligatoirement pousser les termes de façon linéaire. Une branche correspondant à une boucle du gop et qui n'a pas poussé dans $T_{0,n}$ (resp. $T_{n,n}$) au bout de n_0 pas, ne poussera jamais dans $T_{0,n}$ (resp. $T_{n,n}$). (n'oublions pas que cette branche pousse forcément dans un des $T_{i,n}$) .

Connaissant l'ensemble des branches poussant de façon linéaire dans $T_{0,n}$(resp. $T_{n,n}$) les autres ne poussant jamais, il est alors facile de conclure :

Trois cas sont à considérer:

A) Lorsque la hauteur d'une de ces branches dépasse la hauteur d'une branche fermée de la question (resp. du fait), il n'y aura plus de solution.

sinon:

B) Lorsque toutes les branches poussantes ont dépassé la hauteur des branches non closes (obligatoirement sinon la condition A est vérifiée) de la question (resp. du fait)

alors il y a une infinité de solutions distinctes, toutes les solutions à venir peuvent être incomparables entre elles. (ex: fait : Entier(0) →; règle : Entier(Succ(x)) → Entier(x);).

sinon :

C) Aucune branche ne pousse, le nombre de termes distincts que la règle peut engendrer est fini.

6 Exemples

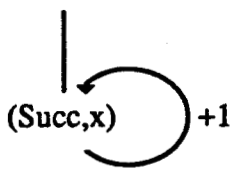
6.1 Règle monotone

Entier(0) →;

Entier(Succ(x)) → Entier (x) ;

Cette règle est, de toute évidence, consommatrice, la question perdant un nœud à chaque appel.

Le Gop associé est :



Cet exemple n'a pas d'effet de bords. On obtient immédiatement le taux de croissance du terme initial en calculant le nombre de nœuds de la boucle divisé par la pondération cumulée de la boucle. Dans ce cas, le taux vaut 1. Cela veut dire que le terme initial poussera d'un nœud pour chaque pas.

Pour une question de hauteur k, la résolution s'arrêtera en au plus k pas.

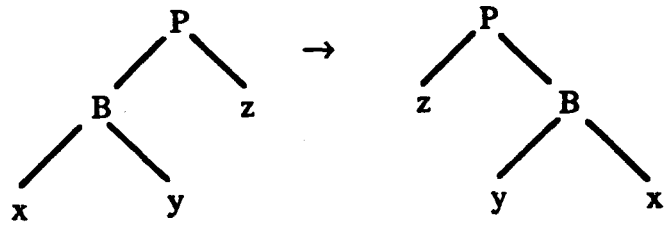
6.2 Règle périodique

La règle qui suit n'est ni consommatrice ni productrice, en revanche elle possède des effets de bords non nuls et une période non nulle.

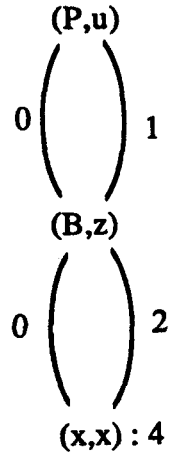
P(B(C,D),B(E,F)) →;

P(B(x,y),z) → P(z,B(y,x));

Pour plus de clarté, nous noterons la règle récursive:



Le gop unifié associé à cette règle est:



Il satisfait à la condition RPB, donc pas d'occurrence due à la règle.

L'unique variable est de période 4, donc la règle est de période 4.

Pas de boucle, donc le nombre de termes possibles est fini.

Effectivement, si l'on effectue les premières réécritures à l'aide de cette règle, on obtient les termes suivants :

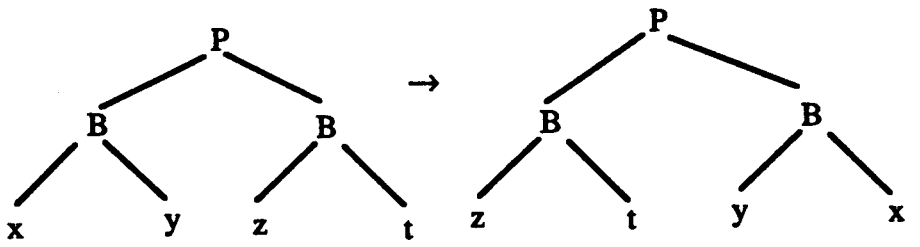
$. P(B(x_1, y_1), z_1) \xrightarrow{r} P(z_1, B(y_1, x_1))$

$. P(B(x_1, y_1), B(x_2, y_2)) \xrightarrow{r^2} P(B(y_1, x_1), B(y_2, x_2))$

$. P(B(x_1, x_3), B(x_2, y_2)) \xrightarrow{r^3} P(B(y_2, x_2), B(x_1, x_3))$

$. P(B(x_1, x_3), B(x_2, x_4)) \xrightarrow{r^4} P(B(x_1, x_3), B(x_2, x_4))$

Au bout de 4 pas on obtient l'identité. Donc la règle est de période 4. Au delà, elle se comporte comme la règle :



Chap 4 - Hiérarchie de Programmes PROLOG -

1) Introduction.

2) Grammaire algébrique associée à un programme Prolog.

3) Grammaires expansives et grammaires quasi-rationnelles.

4) Grammaires déterministes.

5) Ordre d'une grammaire déterministe quasi-rationnelle.

6) Equivalence de programmes Prolog.

7) Transformations normales de programmes.

1 Introduction

Il s'agit de décrire ici, sous la forme d'une grammaire puis d'un graphe de dépendance, l'imbrication logique des différents prédicats d'un programme Prolog.

Cela nous permettra :

- 1 - D'obtenir pour un prédicat donné, grâce à son coefficient d'imbrication, une idée de son facteur de complexité.

- 2 - D'obtenir un critère de *terminaison possible* pour un programme et un but donné.

- 3 - D'élaborer une méthode de *normalisation* de programmes vérifiant de "bonnes propriétés".

- 4 - Puis, pour une classe de programmes logiques, d'obtenir un facteur de complexité.

L'idée de construire une grammaire algébrique puis de définir un graphe de dépendance des prédicats est tout à fait classique (ce dernier correspond au graphe de précédance pour la grammaire algébrique) :

a) à chaque clause on associe une règle de la grammaire en ne considérant que les symboles de prédicats.

b) le graphe est composé d'un noeud par symbole de prédicats. Un arc relie un noeud dont le symbole de prédicat est en partie gauche d'une règle, à tous les noeuds des symboles se trouvant en partie droite de la même règle.

Exemple : Programme

1	Add (0 , x , x)	→ ;
2	Add (sx, y , sz)	→ Add (x , y , z) ;
3	Mul (0 , x , 0)	→ ;
4	Mul (sx , y , z)	→ Mul (x , y , z') Add (y , z' , z) ;
5	Exp (x , 0 , s0)	→ ;
6	Exp (x , sy , z)	→ Exp (x , y , z') Mul (x , z' , z) ;

Grammaire algébrique

Add –1→ e

Add –2→ Add

Mul –3→ e

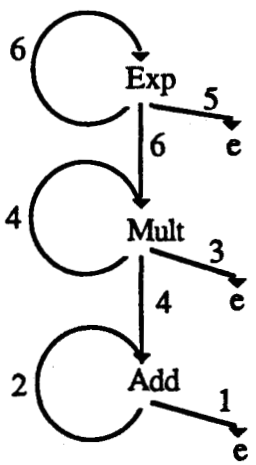
Mul –4→ Mul Add

Exp –5→ e

Exp –6→ Exp Mul

e est l'unique lettre terminale de la grammaire.

Graphe de précédance ou de dépendances des prédicats



2 Grammaire algébrique associée à un programme Prolog

2.1 Notation sur les grammaires

Une grammaire algébrique G est la donnée d'un triplet $G = <\Sigma, N, R>$ tel que:

Σ est un alphabet fini terminal.

N est un alphabet fini non-terminal (par convention nous noterons les non-terminaux avec des majuscules).

R est un ensemble fini de règles de dérivation de la forme :

$$A \text{ –i→ } m$$

où A est un symbole non-terminal, et m un mot de $\Sigma \cup N$. L'indice i ne sert qu'à numéroter les règles de G.

Nous noterons $m \vdash_G m'$, m et m' étant deux mots de $\Sigma \cup N$, si m se dérive en m' par la grammaire G.

Lorsqu'il n'y aura pas d'ambiguïté sur la grammaire nous noterons $m \vdash_{ij..} m'$ si m se dérive en m' par l'utilisation des règles de numéros i j .. dans l'ordre.

ij... formant un mot sur l'alphabet des numéros de règles, nous adoptons la notation $m \vdash_i^k m'$ si m se dérive en m' par k utilisations consécutives de la règle numéro i.

2.2 Définition et construction

Soit π un programme Prolog. Nous numérotons chacune de ses règles dans l'ordre d'apparition. Nous associons de manière classique à π une grammaire G_π algébrique, dont les éléments non-terminaux sont les noms des prédicats :

. Soit $P(a) \rightarrow Q_1(\beta_1) .. Q_n(\beta_n)$ la $i^{ème}$ règle de π , alors :

$P \text{ –i→ } Q_1 .. Q_n$ est une règle de G_π

. Si la $i^{ème}$ règle est un fait de la forme $P(a) \rightarrow$; alors :

$P \text{ –i→ } e$ est une règle de G_π

Les non-terminaux sont les P, Q_1, Q_2, et l'unique symbole terminal est e.

Le ou les axiomes sont les noms des prédicats pouvant apparaître dans la question.

Nous distinguons les règles (P –i→Q R) et (P –j→ Q R) par leur numéro i ou j.

Exemple : Le tri combinatoire [Llo 1984]

1 TRI (x,y) → PERMUTATION (x,y) EST-TRIE (y) ;
2 EST-TRIE (Nil) → ;
3 EST-TRIE (x.y.z) → INF (x,y) EST-TRIE (y.z) ;
4 PERMUTATION (Nil,Nil) → ;
5 PERMUTATION (x.y,u.v) → SUPPRIMER (u,x.y,z)
PERMUTATION (z,v) ;
6 SUPPRIMER (x,x.y,y) → ;
7 SUPPRIMER (x,y.z,y.w) → SUPPRIMER (x,z,w) ;
8 INF (Zero,y) → ;
9 INF (Succ(x),Succ(y)) → INF (x,y) ;

Question : TRI( ) ?

La grammaire associée d'axiome TRI est :

1 TRI –1→ PERMUTATION EST-TRIE

2 EST-TRIE –2→ e

3 EST-TRIE –3→ INF EST-TRIE

4 PERMUTATION –4→ e

5 PERMUTATION –5→ SUPPRIMER PERMUTATION

6 SUPPRIMER –6→ e

7 SUPPRIMER –7→ SUPPRIMER

8 INF –8→ e

9 INF –9→ INF

2.3 Relation de dépendance sur les prédicats

Les classiques graphes d'appel se définissent comme suit dans notre formalisme. Ce dernier nous permettra de raffiner agréablement les notions dans les paragraphes suivants.

Définition : *Soient G_π une grammaire de programmes et A, B deux non-terminaux. On dira que B est dépendant de A (ou appelé par A), si on peut obtenir B en dérivant A.*

$$B \leq A \quad \Leftrightarrow \quad A \vdash_G u B v$$

On associe la relation :

$$A \approx B \quad \Leftrightarrow \{ (A \leq B \text{ et } B \geq A) \text{ ou } A = B \} \quad (\leq \text{n'est pas réflexive})$$

$\approx$ est une relation d'équivalence.

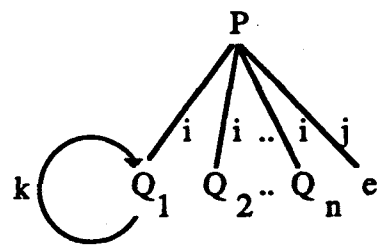
On note $A < B \Leftrightarrow A \leq B$ *et non* $B \geq A$.

Notation : Le Graphe de précédance d'une grammaire (ou graphe de dépendance, au sens des prédicats en logique) se notera sous la forme d'un graphe dont les arcs seront étiquetés du numéro de la régle.

Exemples : $P \xrightarrow{i} Q_1 \; .. \; Q_n$

$P \xrightarrow{j} e$

$Q_1 \xrightarrow{k} Q_1$



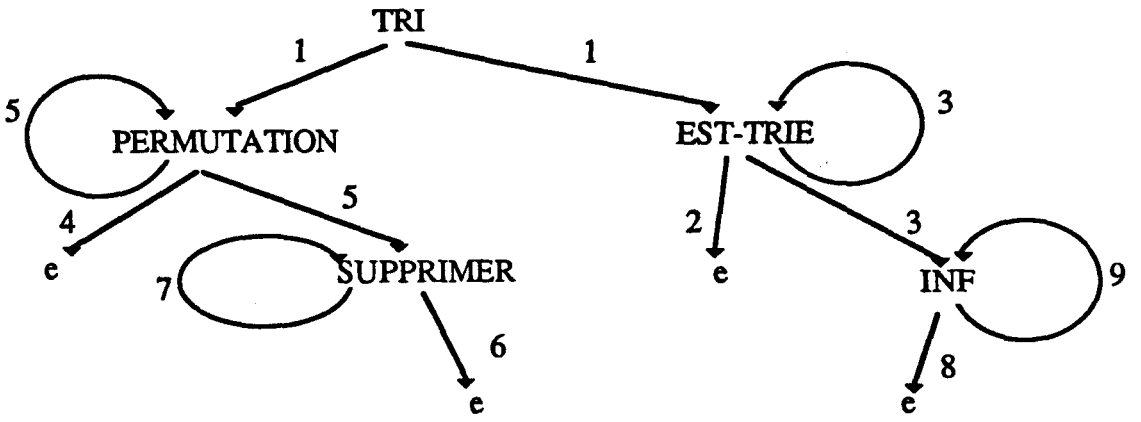
2.4 Graphe de dépendance d'une grammaire

Rappel : On appelle déplié d'un graphe, l'arbre fini ou infini construit à partir du graphe, tel que tout chemin du graphe soit un chemin du déplié, et réciproquement, tout chemin du déplié est un chemin du graphe. Les nœuds d'un chemin ont les mêmes étiquettes dans le graphe et dans le déplié.

Intuitivement, le déplié d'un graphe est son écriture en extension.

Le déplié du graphe de dépendance d'une grammaire associée à un programme Prolog est une représentation générale de l'arbre de recherche et/ou associé aux prédicats les plus externes du programme, et où l'on n'a gardé que les symboles de prédicats.

Reprenons l'exemple du Tri combinatoire :



2.5 Propriété de terminaison possible

Il est évident qu'un programme Prolog ne pourra s'arrêter quel que soit le but demandé, que si, pour chaque prédicat du programme, il existe au moins une règle susceptible de l'effacer. Ce qui nous donne la définition récursive suivante sur le graphe associé au programme :

Définition : *Nous dirons qu'un noeud du graphe vérifie la propriété de succès possible si et seulement si :*

- soit il possède un noeud fils terminal.

- soit il existe un numéro d'arc pour lequel tous les noeuds accessibles vérifient la propriété de succès possible.(Les points d'entrée du graphe sont les nœuds correspondant aux questions).

Propriété : Soit G une grammaire algébrique, les deux propriétés suivantes sont équivalentes :

i) Les (ou le) nœuds correspondant aux axiomes ne vérifient pas la propriété de succès possible

ii) Tout programme ayant même grammaire et mêmes axiomes n'admet aucune solution.

Exemple : Le programme limité à $P(\beta) \rightarrow P(\gamma)$ n'admet aucune solution. Dans le Tri combinatoire, si l'on retire l'une des règles 2,4 ou 6, le prédicat TRI n'admet aucune solution.

En effet : Il est clair qu'un noeud n'ayant aucun noeud fils terminal, correspond à un symbole de prédicat qui n'apparaît pas comme fait. Donc le seul moyen d'effacer un sous-but de tel symbole est d'utiliser une règle non récursive dont tous les termes de la partie droite correspondent à des noeuds effaçables, c'est à dire qu'il doit exister un numéro donnant accès à un ensemble de noeuds effaçables.

Dans la suite de ce travail nous supposerons que les programmes considérés vérifient ce critère.

3 Grammaires expansives et grammaires quasi-rationnelles.

Définition : *Une grammaire est dite* ***expansive*** *si et seulement si il existe un non-terminal A tel que :*

$$A \vdash_G uAvAw$$

Par opposition, une grammaire non expansive sera dite ***quasi-rationnelle*** *(au sens de la théorie des langages).*

Remarque : G est une grammaire quasi-rationnelle si et seulement si, pour toute règle $A \rightarrow B_1 B_2 .. B_n$ de G, il existe au plus un B_i équivalent à A.

Exemples :

(1) La programmation du problème des tours d'HANOI peut se faire de la façon suivante:

HANOI (. .) → HANOI (. .) DEPLACER (. .) HANOI(. .) ;

ce qui génère une grammaire expansive.

(2) Le Quick Sort est souvent écrit de la façon suivante :

QUICKSORT (e.l , lt) → PARTITIONNER (e , l , l_1 , l_2)

QUICKSORT (l_1 , lt_1)

QUICKSORT (l_2 , lt_2)

CONCATENER (lt_1 , e.lt_2 , lt) ;

ce qui est aussi de nature expansive.

(3) La grammaire du Tri combinatoire n'est pas expansive.

Remarque : Dans la suite de ce chapitre, les grammaires que nous étudierons sont supposées quasi-rationnelles.

4 Grammaires déterministes.

4.1 Ordre d'un prédicat

Intuitivement, l'ordre d'un prédicat représente, dans le cas de grammaires quasi-rationnelles, le nombre d'imbrications de boucles accessibles à partir de ce prédicat. Etant donné un prédicat récursif, il faut différencier les boucles accessibles après être sortie de la récursion de celles accessibles sans être sorti de la récursion. Seules ces dernières sont appelées boucles imbriquées.

Soient les seules règles récursives du programme ayant P ou Q en tête:

P( ) → P( ) ;

P( ) → Q( ) ; seul chemin de P à Q

Q( ) → Q( );

Dans l'exemple ci-dessus P et Q sont à priori (cela dépend uniquement des autres prédicats dépendant de P) de même ordre car la boucle sur Q n'est pas imbriquée dans celle de P. Dans l'exemple suivant, P est d'ordre strictement supérieur à l'ordre de Q:

P( ) → P( ) Q( );

Q( ) → Q( );

Définition : *Soit G une grammaire déterministe et quasi-rationnelle. Un prédicat Q est dit* ***étroitement dépendant*** *d'un prédicat P si et seulement si il existe un chemin de dérivation m tel que :*

$P \vdash_m P \ldots Q \ldots$

Intuitivement, les prédicats étroitement dépendants d'un prédicat récursif P sont ceux qui lors de la résolution seront générés dans le but courant à chaque utilisation de la régle récursive ayant en partie gauche le prédicat P.

Exemples:

1) PERMUTATION $-5\rightarrow$ PERMUTATION SUPPRIMER (du tri combinatoire)

Si, lors de la résolution PROLOG, on applique k fois cette règle récursive sur la question PERMUTATION(-) alors :

PERMUTATION (-) $\vdash 5^k -$ PERMUTATION(-) SUPPRIMER($-_1$)

... SUPPRIMER($-_k$)

SUPPRIMER est étroitement dépendant de PERMUTATION

2) Q $-1\rightarrow e_1$

P $-2\rightarrow$ Q G est déterministe car Q < P

P $-3\rightarrow$ P mais Q n'est pas étroitement dépendant de P

Définition: *De l'ordre d'un prédicat P (non-terminal de G)*

Pour un prédicat quelconque P, l'ordre de P est le maximum de l'ordre des prédicats dépendants de P ou, s'il en existe, le maximum de l'ordre des prédicats étroitement dépendants de P plus 1.

a) P est non récursif : Ordre(P) = Sup{ ordre(Q) / Q < P }

(0 s'il n'existe aucun Q < P)

b) P est récursif : k = Sup { ordre (Q) / Q < P }

(k = 0 s'il n'existe aucun Q < P)

j = Sup { ordre (Q) / Q étroi. dép. de P}

(j = 0 s'il n'existe aucun Q étroi. dép. de P)

Ordre (P) = Sup { j+1 , k } ($\geq$ 1)

Dans le cas général, l'ordre d'un prédicat correspond au nombre maximum de boucles imbriquées accessibles par ce prédicat. (exemple §1: Add:1, Mul:2, Exp:3, ...).

L'ordre naturel serait de confondre "prédicats dépendants" et "prédicats étroitement dépendants" en augmentant de 1 dans tous les cas l'ordre de P. Mais dans certains cas, l'ordre naturel serait pessimiste. Par exemple si le prédicat récursif n'a pas de prédicat étroitement dépendant on peut effectuer un calcul plus proche de la réalité :

$Q(.) \rightarrow ;$

$Q(.) \rightarrow Q(.) ;$

$P(.) \rightarrow Q(.) ;$ Ordre(Q) = Ordre(P) = 1

$P(.) \rightarrow P(.) ;$

Deux autres exemples sont présentés (cf §5.3), liés aux langages :

$\{ a^n b^n / n \in N \}$ et $\{ a^n b^n c^n / n \in N \}$.

Remarque : $Q < P \Rightarrow Ordre(Q) \leq Ordre(P)$.

4.2 Grammaires déterministes

Nous rappelons que les résultats suivants concernent des grammaires quasi-rationnelles.

Définition : *Une grammaire est dite c l a s s é e si et seulement si dans le corps des règles, les non-terminaux apparaissent de gauche à droite par ordre décroissant :*

pour toute règle $P \rightarrow Q_1 \ldots Q_n$ *de G, si* $i < j$ *alors* $Ordre(Q_i) \geq Ordre(Q_j)$

Exemple : La grammaire du Tri combinatoire n'est pas classée. En réordonnant les termes en partie droite des régles 3 et 5 , on peut construire la grammaire classée suivante:

1 TRI –1→ PERMUTATION EST-TRIE
2 EST-TRIE –2→ e
3 EST-TRIE –3→ EST-TRIE INF
4 PERMUTATION –4→ e
5 PERMUTATION –5→ PERMUTATION SUPPRIMER

6 SUPPRIMER $-6\rightarrow$ e
7 SUPPRIMER $-7\rightarrow$ SUPPRIMER
8 INF $-8\rightarrow$ e
9 INF $-9\rightarrow$ INF

Remarque : On ne change rien aux solutions d'un programme π si on modifie l'ordre des prédicats dans le corps des règles. Mais, en revanche, cette modification peut rendre des solutions inaccessibles si l'on utilise la stratégie classique de Prolog.

Définition : *Une grammaire G classée est dite* ***déterministe*** *si et seulement si pour tout non-terminal A de G et pour tout couple de chemins de dérivation gauche en stratégie classique Prolog II* (m_1 ,m_2) :

$A \vdash_{m_1} Au$ *et* $A \vdash_{m_2} Av$ *alors* m_1 *est un préfixe (facteur gauche) de* m_2 *ou* m_2 *est un préfixe de* m_1. *C'est à dire qu'il existe m tel que soit* $m_1 = m_2 m$, *soit* $m_2 = m_1 m$.

(RAPPEL : En dérivation gauche, on effectue toujours la réécriture du non-terminal, le plus à gauche.)

Remarque : Le mot **déterministe** est ici employé dans un sens très particulier. Il s'agit en fait de déterminisme lors du choix d'un règle récursive. On peut parler de **déterminisme récursif** par opposition au sens classique où le déterminisme doit s'appliquer à tous les symboles de prédicats.

Cette notion de grammaire classée nous permet de différencier les programmes réellement non-déterministes,tels que :

A $-1\rightarrow$ A;

A $-2\rightarrow$ A;

de programmes "pseudo-déterministes" tels que :

B $-1\rightarrow$ e;

B $-2\rightarrow$ e; (A $\vdash_{3.1-}$ e A et A $\vdash_{3.2-}$ e A)

A $-3\rightarrow$ B A ;

en classant cette grammaire, la règle 3 devient A $-3\rightarrow$ A B; elle est alors déterministe.

Plutôt que d'opérer une dérivation gauche, on pourrait appliquer une dérivation du prédicat d'ordre le plus élevé. Il ne serait pas nécessaire alors de la classer.

Convention : On dira par extension qu'un programme π est classé, déterministe, et quasi-rationnel si sa grammaire vérifie les mêmes propriétés.

Exemples: (1) Le Quick Sort , sous sa forme classique, est non déterministe, puisqu'il utilise le prédicat PARTITIONNER qui s'écrit :

PARTITIONNER (e , Nil , Nil , Nil) $\rightarrow$;

PARTITIONNER (e , f.l , f.l_1 , l_2) $\rightarrow$ PARTITIONNER (e , l , l_1 , l_2)
INF (f , e) ;

PARTITIONNER (e , f.l , l_1 , f.l_2) $\rightarrow$ PARTITIONNER (e , l , l_1 , l_2)
INF (e , f) ;

L'arrêt de ce type de programmes est directement lié à la diminution de la taille des données à chaque utilisation d'une des règles.

(2) La conjecture de COLLATZ (ou conjecture tchèque), plus connue sous le nom de "problème de Syracuse", non encore démontrée, s'exprime pour tout n par l'arrêt du programme suivant :

```
Tant Que n ≠ 1
  Si n pair Alors      n ← n/2
       Sinon           n ← 3*n+1
  Fsi
Fin tant que
```

En Prolog II, sous la forme synthétique :

COLLATZ (1) → ;

COLLATZ (n) → PAIR (n) MUL(n',2,n) COLLATZ (n') ;

COLLATZ (n) → IMPAIR (n) MUL(n,3,n') COLLATZ (s(n')) ;

La conjecture de COLLATZ est équivalente à l'arrêt de la résolution du but COLLATZ(n), quel que soit la valeur de n , entier positif et non nul. La grammaire de ce programme est non déterministe .

Nous verrons par la suite, que tout programme non déterministe admet un équivalent déterministe. Pour l'exemple ci-dessus voici le programme logiquement équivalent déterministe:

...

Q (n' , n) → PAIR(n) MUL(n',2,n) ;

Q (s(n') , n) → IMPAIR (n) MUL(n,3,n') ;

COLLATZ (1 , x) → ;

COLLATZ (n , x) → COLLATZ (y , n) Q (y , n) ;

(3) Le programme relatif au Tri combinatoire est quasi-rationnel, classé et déterministe, sous la forme suivante :

1 TRI (x,y) → PERMUTATION (x,y) EST-TRIE (y) ;
2 EST-TRIE (Nil) → ;
3 EST-TRIE (x.y.z) → EST-TRIE (y.z) INF (x,y) ;
4 PERMUTATION (Nil,Nil) → ;
5 PERMUTATION (x.y,u.v) → PERMUTATION (z,v) SUPPRIMER (u,x.y,z) ;
6 SUPPRIMER (x,x.y,y) → ;
7 SUPPRIMER (x,y.z,y.w) → SUPPRIMER (x,z,w) ;
8 INF (Zero,y) → ;
9 INF (Succ(x),Succ(y)) → INF (x,y) ;

5 Ordre d'une grammaire déterministe quasi-rationnelle.

5.1 Ordre d'une grammaire ou d'un programme

Définition : *Une grammaire déterministe quasi-rationnelle, et par extension le programme correspondant , a pour ordre le Sup de l'ordre de ses axiomes.*

Remarque : Toute grammaire déterministe et quasi-rationnelle admet un ordre. Dan la plupart des cas, il se déduit facilement de son graphe de précédance.

5.2 Exemples de grammaires

Par convention, dans un programme Prolog, les variables seront toujours notées en minuscules, et les Fonctions avec au moins une majuscule, les constantes étant considérées comme des fonctions d'arité nulle.

D'autres exemples de programmes d'ordre 1 et 2 sont proposés en annexe.

1) Ordre(G_π) = 0

Un des exemples les plus utilisés pour introduire PROLOG est le suivant :

PERE (Jean,Jacques) →;

PERE (Alain,Jean) →;

MERE (Helene,Jacques) →;

MERE (Marie,Jean) →;

PARENT (x,y) → PERE (x,y);

PARENT (x,y) → MERE (x,y);

GRAND-PERE (x,y) → PERE (x,z) PARENT (z,y);

Aucun prédicat récursif , c'est à dire pas de récursivité. Donc le nombre de réécritures possibles est borné. Nous dirons de ces programmes qu'ils sont de "Complexité constante".

2) Ordre(G_π) = 1

a) Sur le schéma du TANT QUE PROLOG (cf chap 2), on peut citer de nombreux exemples. (cf annexe)

Si on représente les entiers sous la forme de " $Succ^n$ (Zero) " :

* Langage rationnel des entiers :

ENTIER (Zero) → ;
ENTIER (Succ(x)) → ENTIER (x) ;

* Prédicat vérifiant si " 1^{er} argument < $2^{ème}$ argument "

INF (Zero,Succ(x)) → ; [0 < (x+1)].
INF(Succ(x),Succ(y)) → INF (x,y) ; [(x+1) < (y+1)] si [x < y].

3) Ordre(G_π) = 2

* Tri combinatoire :

TRI	–1→	PERMUTATION EST-TRIE
EST-TRIE	–2→	e
EST-TRIE	–3→	EST-TRIE INF
PERMUTATION	–4→	e
PERMUTATION	-5→	PERMUTATION SUPPRIMER
SUPPRIMER	–6→	e
SUPPRIMER	–7→	SUPPRIMER
INF	–8→	e
INF	–9→	INF

Ordre(INF) = Ordre(SUPPRIMER) = 1

Ordre(PERMUTATION) = Ordre(EST-TRIE) = Ordre(TRI) = 2

4) Ordre(G_π) = 3

On peut redéfinir certaines des fonctions arithmétiques pré-citées par des programmes d'ordre 3. Ceux-ci correspondent à leur expression la plus courante (et d'ailleurs souvent la meilleure). En utilisant les prédicats ADD et MUL2 , les prédicats Factoriel , Exponentiel et Fermat s'écrivent donc:

FACTORIEL3 (Zero , Succ(Zero)) → ;

FACTORIEL3 (Succ(n) , résultat) → FACTORIEL3 (n , sous-résultat)

MUL2 (n , sous-résultat , résultat) ;

EXP3 (x , Zero , Succ(Zero)) → ;

EXP3 (x , Succ(y) , z) → EXP3 (x , y , w) MUL2 (x , w , z) ;

FERMAT3 (x , y , z , n) → EXP3 (x , n , x^n) EXP3 (y , n , y^n)

EXP3 (z , n , z^n)

INF(Succ(Zero) , x) INF(Succ(Zero) , y) INF(Succ(Zero) , z)

INF(Succ(Succ(Zero)) , n) ADD (x^n, y^n , z^n) ;

6 Equivalence de programmes PROLOG

L'équivalence des programmes est un problème très important, tant en programmation logique, qu'en tout autre mode de programmation. C'est donc un domaine très étudié. Chaque approche sémantique d'un langage de programmation, et il y en a en général de nombreuses, possède sa propre relation d'équivalence. Il est donc nécessaire, pour aborder ce problème de l'équivalence de programme, de préciser clairement la sémantique utilisée.

Les sémantiques couramment rencontrées en programmation logique sont nombreuses, citons les sémantiques dénotationnelles ou fonctionnelles correspondant à l'énoncé de la ou des fonctions calculées par le programme en un nombre quelconque d'inférences (cf. la célèbre fonction Tp [KOW 1976]), les sémantique logiques pour lesquelles le programme est un ensemble complet de déductions logiques (différentes techniques de complétion donnent différentes interprétations), puis les semantiques opérationnelles, pour lesquelles on évalue les différents ensembles de succès et d'échecs. Chacune de ces différents sémantique donne lieu à plusieurs relations d'équivalence. Pour plus d'information, nous conseillons au lecteur de se reporter à [MAH 1986], dans lequel une hiérarchie entre ces différentes relations est présentée.

Pour notre part, nous utiliserons deux notions d'équivalences, décrite ci-dessous.

6.1 Equivalence simple

Définition : *Deux programmes Prolog π_1 et π_2 sont dits **équivalents si et seulement si** leur exécution donne le même ensemble de solutions, pour toute stratégie complète.*

Si on considère cette équivalence sans se restreindre à un ensemble de buts initiaux, il s'agit de l'équivalence par égalité des plus petits points fixes des deux fonctions T_{π_1} et T_{π_2} (fonctions "déduction directe" introduites par Van Emden et Kowalski [KOW 1976], voir aussi [MAH 1986]).

Exemples:

1) Les prédicats MUL1 et MUL2 sont deux façons différentes de programmer la multiplication de deux entiers :

π_1 : MUL1 (x,y,z) → CALCUL (x,y,z,Zero,Zero,Zero) ;

% initialisation de trois compteurs à 0 %

CALCUL (x,y,z,z,Zero,x) → ;

CALCUL (x,y,z,c0,c1,c2) → CALCUL (x,y,z,Succ(c0),c1',c2')

COMPTEUR (x,y,c1,c1',c2,c2') ;

% c0 compteur résultat, c1 pour y, c2 pour x, on fait x fois la somme de y %

COMPTEUR (x,Succ(y),y,Zero,c2,Succ(c2)) → ;

% si le compteur c1 vaut y-1, on ajoute 1 à c2 et on repart avec 0 dans c1 %

COMPTEUR (x,y,c1,Succ(c1),c2,c2) → ;

% sinon on ajoute 1 à c1, c2 et inchangé %

π_2 : ADD (Zero,x,x) → ;

ADD (Succ(x),y,Succ(z) → ADD (x,y,z) ;

MUL2 (Zero,x,Zero) → ;

MUL2 (Succ(x),y,z) → MUL2 (x,y,z') ADD (y,z',z) ;

π_1 correspond à une programmation typiquement itérative; π_2 correspond plutôt à une programmation déclarative , π_1 équivalent à π_2 pour les buts respectifs Mul1 et Mul2.

2) Certains prédicats dans un programme ne peuvent pas être appelés à partir des prédicats axiomes. On peut donc supprimer toutes les règles pour lesquelles ces prédicats

apparaissent en tête. Ce sont les non-terminaux de la grammaire non inférieurs à aucun des axiomes.

1 - Q (.) → ;

2 - P (.) → ;

3 - P (.) → P (.) Q (.) ;

4 - R (.) → P (.) ;

Question : P (.) ou/et Q (.) ?

La règle 4 n'est pas nécessaire, pour ces questions, dans le programme (R n'est inférieur ni à P ni à Q) .

3) Certaines règles sont non utilisables, c'est à dire qu'elles ne peuvent jamais permettre l'obtention d'une solution. Nous en avons vu un exemple pour le critère de "terminaison possible" (cf ...). Voici un autre type d'exemple plus difficile à détecter:

1 - ENTIER (Zero) → ;

2 - ENTIER (Succ(x)) → ENTIER (x) ;

3 - ENTIER(x) → ENTIER (F(x)) ;

La règle 3 est inutilisable pour l'obtention d'une solution .

Les différentes opérations citées précédemment correspondent à une simplification du programme. Les transformations sont parfois syntaxiques (ex. 2), mais la plupart d'entre elles demandent une analyse fine, dans certains cas indécidable (chap. 5).

6.2 Prolog-équivalence

La sémantique opérationnelle associée au programmes Prolog en stratégie "en profondeur d'abord" est la suivante:

Nous associons à chaque programme un ensemble de prédicats appelés *Questions possibles*, pour lesquels nous définirons une suite de systèmes fortement réduits d'équations (cf définition plus bas) S_1, S_2, ... , S_n , ... dont les solutions nous donnerons, dans l'ordre, les substitutions calculées par l'interpréteur pour ces prédicats.

Définition [DEV 88] : *Un système réduit d'équation rs [COL 84] est dit Fortement réduit si :*

. *$\forall (U=t), (V=t') \in rs$, if t' non réduit à une variable alors t' n'est pas un sous-arbre de t.*

. *$\forall (U=V) \in rs$, la variable U n'apparait pas dans une autre équation.*

La propriété principale de tels systèmes est l'unicité. En effet,à chaque étape de la résolution, la substitution courante s'exprime d'une manière unique en un système fortement réduit d'équations.

Deux programmes sont dit **Prolog-équivalents** pour deux ensembles de questions possibles donnés (un par programme), si et seulement si les suites de systèmes d'équations associées sont équivalents aux notations près, et si les système S_i sont les systèmes calculés après un nombre de pas de résolution respectif inférieur ou égal à $(A \times i)$, A est une constante entière ne dépendant que du programme. Ce type d'équivalence opérationnelle conserve la complexité (ordre du nombre d'unifications) de la résolution des buts de l'ensemble des questions possibles.

Intuitivement, Les deux programmes auront le même comportement vis-à-vis des prédicats sélectionnés. C'est à dire que la résolution "réduites à ces ensembles de prédicats" sera identique dans l'un et l'autre programme. Un certain nombre "d'étirements" de tailles bornées pouvant intervenir dans un programme comme dans l'autre lors de la résolution. Ces "étirements" sont dus à l'utilisation, bornée en nombre (au plus A), de prédicats intermédiaires.

Exemple :

Les programmes suivants sont de toute évidence opérationnellement équivalents pour toute question de prédicat Plus ( ) pour P1 et P2 Ψ(Plus( )) pour P3 :

P1 : Plus(0, x , x) →;

Plus(sx , y , sz) → Plus (x , y , z) ;

P2 : Plus (0 , x , x) → ;

Plus (sx , y , sz) → Q (x , y , z) ;

Q (x , y , z) → Plus (x , y , z) ;

P3 : Ψ (Plus(0, x , x))→;

Ψ (Plus(sx , y , sz))→ Ψ (Plus (x , y , z)) ;

La Prolog-équivalence est la plus "forte" que l'on puisse espérer, car elle implique les mêmes calculs dans le même ordre.

Exemple : reprenons l'exemple de COLLATZ ou "conjecture tchèque" ou encore problème de Syracuse:

π_1 COLLATZ (1) → ;

COLLATZ (n) → PAIR (n) MUL(n',2,n) COLLATZ (n') ;

COLLATZ (n) → IMPAIR (n) MUL(n,3,n') COLLATZ (s(n')) ;

π_2 COLLATZ' (1 , x) → ;

Q (n' , n) → PAIR(n) MUL(n',2,n) ;

Q (s(n') , n) → IMPAIR (n) MUL(n,3,n') ;

COLLATZ' (n , x) → Q (y , n) COLLATZ' (y , n) ;

COLLATZ (n) → COLLATZ' (n , x) ;

Les deux programmes ci-dessus sont Prolog-équivalents car, hors mises les substitutions sur x et y, la pile des substitutions en stratégie classique Prolog évolue de façon strictement identique. La suite des sous-buts du premier programme se retrouve comme sous-suite des sous-buts du second. En fait l'arbre de résolution est identique à quelques "étirements" près, lors de l'utilisation du prédicat Q.

Déplions une boucle dans l'arbre de résolution :

π_1	π_2
COLLATZ (5)	COLLATZ (5)
.	COLLATZ' (n , x)
.	{n=5 , x = }
.	Q (y , n) COLLATZ' (y , n)
	{n=5 , y = }
IMPAIR (n) MUL(n,3,n') COLLATZ (s(n'))	IMPAIR (n) MUL(n,3,n') COLLATZ'(y ,n)
{ n=5 , n'= }	{ n=5 , y= s(n') , n'= }
MUL(n,3,n') COLLATZ (s(n'))	MUL(n,3,n') COLLATZ' (y , n)
{ n=5 , n'= }	{ n=5 , y= s(n') , n'= }
COLLATZ (s(n'))	COLLATZ' (y , n)
{n'=15}	{ n=5, y=s(n') , n'=15 }

... etc ...

Il est immédiat que l'équivalence ainsi définie, nous assure que les deux programmes concernés donnerons les réponses à la question dans le même ordre et dans le même ordre de temps (même degré de complexité).

Cette définition, nous est utile à bien imaginer l'ensemble des différences pouvant exister entre deux programmes dits "Prolog-équivalents". Les transformations que nous allons présenter, et qui conservent la Prolog-équivalence, le feront souvent de manière évidente au regard de cette définition.

7 Transformations normales de programmes

Les transformations de programmes par pliage/dépliage présentées dans ce paragraphe, s'apparente à la méthode de Burstall-Darlington [BUR 1977] appliquée à la programmation logique [AZI 1986]. Dans cette dernière publication, une méthode de transformation de programmes Prolog en programmes Quasi-itératifs est développée. Cette notion de programme Quasi-itératif est très proche de la notion, utilisée dans cette thèse, de programmes quai-rationnels.

7.1 Forme normale pour l'équivalence simple

Propriété : Tout programme quasi-rationnel et déterministe d'ordre p admet un programme équivalent de même ordre dont les règles ne peuvent avoir que deux formes :

Type 1 : $P(\beta) \rightarrow Q_1(\gamma_1) \ldots Q_n(\gamma_n)$;

avec pour tout i : $Q_i < P$

Type 2 : $P(\beta) \rightarrow P(\gamma)\ Q(\delta)$;

avec $Q < P$ et une et une seule règle de ce type par prédicat P récursif.

Nous dirons que, dans ce cas, le programme est **normalisé**.

En effet :

1) Ce schéma de programme type est quasi-rationnel et déterministe.

* quasi-rationnel, car les règles de type 1 sont non récursives, et dans les règles de type 2 une seule récursion n'est possible.

* déterministe, car la seule façon de dériver A à partir de lui-même est d'utiliser l'unique règle récursive de tête A. Les chemins de dérivation possibles appartiennent donc à {n°règle}*, c'est à dire que pour tout m1 et m2 $\in$ {n°règle}*, m1 est préfixe de m2 ou m2 est préfixe de m1.

2) Montrons que tout programme déterministe et quasi-rationnel peut être mis sous cette forme.

La démonstration est purement constructive, elle se décompose en quatre étapes. En fait, il s'agit d'éliminer les boucles d'appel de plus d'une règle. Ce qui nous donne :

Etape 1 : Réaliser la simulation de l'utilisation d'une partie de boucle à l'aide d'une seule règle non récursive. Il y aura donc une nouvelle règle pour chaque partie de boucle.

Etape 2 : Toute utilisation non complète d'une boucle étant simulée par une règle non récursive, l'utilisation d'une boucle dans sa totalité peut être simulée par une seule règle récursive.

Etape 3 : normalisation de la forme des règles récursives.

Etape 4 : Emondage. C'est à dire suppression des règles non accessibles.

Ces différentes étapes sont détaillées ci-dessous.

1ère Etape : Soit P un prédicat récursif. Comme le programme est déterministe, il existe un et un seul chemin minimal non vide permettant d'obtenir P à partir de lui-même. Il existe donc une suite unique de règles $i_0, i_1, i_2 \ldots i_n$, toutes différentes telles que :

$P(\beta_0) \ -i_0\rightarrow \ P_1(\gamma_0)$ liste-0-de-prédicats ;

$P_1(\beta_1) \ -i_1\rightarrow \ P_2(\gamma_1)$ liste-1-de-prédicats ;

.. ..

$P_n(\beta_n) \ -i_n\rightarrow \ P(\gamma_n)$ liste-n-de-prédicats ;

où chacune des listes de prédicats est formée de prédicats d'ordre strictement inférieur à Ordre(P) (=Ordre(P_i)).

Comme $P \approx P_1 \approx \ldots \approx P_n$ (voir la définition de l'équivalence $\approx$ en début de chapitre), on se choisit un représentant de cette classe d'équivalence, par exemple P .

Pour toute règle non récursive portant sur un des P_k équivalents à notre représentant canonique P : $P_k(\,.\,) \ -j\rightarrow$ liste-de prédicats $< P_k$, liste éventuellement vide, on

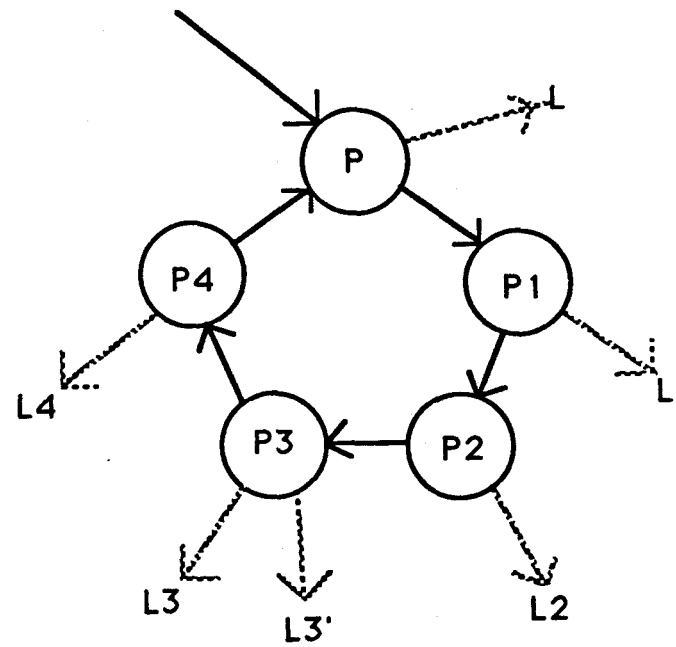
ajoute une règle correspondant à l'application successive des règles $i_0, i_1, \ldots, i_{k-1}, j$, règle qui sera donc de la forme :

P(.) → liste-de-prédicats ;

qui est une règle non récursive où chaque prédicat du corps est strictement inférieur à P. Les règles ainsi obtenues sont donc de type 1.

Preuve d'équivalence : Comme toutes les règles de départ sont conservées par cette étape et comme les nouvelles régles sont déduites par k applications successives de règles du programme, l'équivalence des deux programmes est immédiate.

Les règles que nous venons d'ajouter au programme Prolog correspondent à l'utilisation, dans le graphe ci-dessus, des chemins d'appels PP_1L_1; $PP_1P_2L_2$; $PP_1P_2P_3L_3$; $PP_1P_2P_3L_{3'}$; $PP_1P_2P_3P_4L_4$; où les L_i sont les listes de prédicats engendrées à l'appel des P_i



2ème Etape : Génération de la règle " P(β) → P(γ) liste-de-prédicats ; "

On remplace $P(\beta_0) -i_0\rightarrow P_1(\gamma_0)$; par la règle obtenue par application successive des règles $i_0, \ldots, i_n$:

P(β) → P(γ) liste0 ,1. . n -de-prédicats;

Dans ce nouveau programme, les prédicats $P_1, P_2, \ldots, P_n$ ne sont plus récursifs, puisque la règle i_0 a été supprimée.

Preuve d'équivalence : Toutes les façons d'effacer P(-) sans jamais revenir sur un sous-but de prédicat P sont prises en compte par les règles générées à l'étape 1. Il nous reste donc une seule possibilité, celle d'utiliser toutes les règles $i_0, \ldots, i_n$ consécutivement, ce qui est fait par la règle ci-dessus.

Les suites d'appels de la forme $P_i\ P_{i+1} \ldots P_n\ P\ P_1 P_2 \ldots P_n\ P\ P_1\ P_2 \ldots P_k$ sont remplacées par des suites de la forme $P_i\ P_{i+1} \ldots P_n\ P\ P$

3ème Etape : Après ces transformations, certaines règles sont de la forme :

$P(\beta) \rightarrow P(\gamma)$ liste-de-prédicats ;

avec liste-de-prédicats d'ordre inférieur strictement à Ordre(P).

On peut remplacer liste-de-prédicats par un seul, notons le Q, en ajoutant une règle de type (1) : $Q(\alpha) \rightarrow$ liste-de-prédicats ; avec α un arbre contenant toutes les variables de liste-de-prédicats, appartenant aussi à β ou à γ.

$\alpha = \#(x_1, x_2, \ldots, x_p)$ $\quad x_i \in$ Var(liste2-de-prédicats)) $\cap$ Var(β,γ)

Preuve d'équivalence : immédiat.

4ème Etape : Pour simplifier le programme engendré, on peut supprimer toutes les règles non accessibles à partir des axiomes, c'est à dire dont le prédicat tête n'est pas inférieur à un des axiomes, au sens de la relation de préordre.

7.2 Exemple de transformation

Exemple : Soit π ,le programme suivant, d'axiome R :

$S(\beta_1)$ $-1\rightarrow$;

$S(\beta_2)$ $-2\rightarrow$ $S(\gamma_2)$;

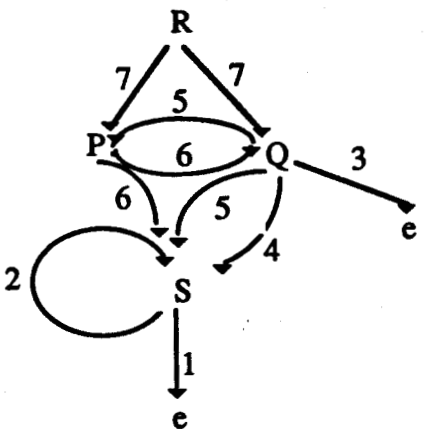
$Q(\beta_3)$ $-3\rightarrow$;

$Q(\beta_4)$ $-4\rightarrow$ $S(\gamma_4)$;

$Q(\beta_5)$ $-5\rightarrow$ $P(\gamma_5)$ $S(\Phi_5)$;

$P(\beta_6)$ $-6\rightarrow$ $Q(\gamma_6)$ $S(\Phi_6)$;

$R(\beta_7)$ $-7\rightarrow$ $P(\delta_7)$ $S(\Phi_7)$;



Ordre(R) = Ordre(P) = Ordre(Q) = 2 et Ordre(S) = 1

Etape 0 :

. π est quasi-rationnel. Seules les règles 2,5,6 sont récursives et elles ne possèdent qu'un seul prédicat équivalent dans le corps de la règle.

. π est classé.

. π est déterministe.

Etape 1 : Les seuls prédicats récursifs sont P et Q. Il existe une et une seule façon d'obtenir P à partir de lui-même (déterministe) :

. $P(\beta_6)$ $-6\rightarrow$ $Q(\gamma_6)$ $S(\Phi_6)$;

. $Q(\beta_5)$ $-5\rightarrow$ $P(\gamma_5)$ $S(\Phi_5)$;

On ajoute les régles correspondant à l'application des règles suivantes :

. $P(\beta_6)$ $-6\rightarrow$ $Q(\gamma_6)$ $S(\Phi_6)$ et $Q(\beta_3)$ $-3\rightarrow$ $\equiv$ $P(\beta_8)$ $-8\rightarrow$ $S(\Phi_8)$;

. $P(\beta_6)$ $-6\rightarrow$ $Q(\gamma_6)$ $S(\Phi_6)$ et $Q(\beta_4)$ $-4\rightarrow$ $S(\gamma 4) \equiv$ $P(\beta_9)$ $-9\rightarrow$ $S(\gamma_9)$ $S(\Phi_9)$;

π : $S(\beta_1)\ -1\rightarrow\ ;$ $Q(\beta_5)\ -5\rightarrow\ P(\gamma 5)\ S(\Phi_5)\ ;$

$S(\beta_2)\ -2\rightarrow\ S(\gamma 2)\ ;$ $P(\beta_6)\ -6\rightarrow\ Q(\gamma 6)\ S(\Phi_6);$

$Q(\beta_3)\ -3\rightarrow\ ;$ $R(\beta_7)\ -7\rightarrow\ P(\delta 7)\ S(\Phi 7)\ ;$

$Q(\beta_4)\ -4\rightarrow\ S(\gamma_4)\ ;$ $P(\beta_8)\ -8\rightarrow\ S(\Phi_8)\ ;$

$P(\beta_9)\ -9\rightarrow\ S(\gamma_9)\ S(\Phi_9)\ ;$

Etape 2 : On remplace la régle 6 par celle obtenue par application de 6 et 5 :

cad $P(\beta_6)\ \rightarrow\ Q(\gamma_6)\ S(\Phi_6);$ par $P(\beta'_6)\ -6\rightarrow\ P(\gamma'_6)\ S(\delta'_6)\ S(\Phi'_6)\ ;$

π : $S(\beta_1)\ -1\rightarrow\ ;$ $Q(\beta_5)\ -5\rightarrow\ P(\gamma 5)\ S(\Phi_5);$

$S(\beta_2)\ -2\rightarrow\ S(\gamma 2)\ ;$ $P(\beta'_6)\ -6\rightarrow\ P(\gamma'_6)\ S(\delta'_6)\ S(\Phi'_6)\ ;$

$Q(\beta_3)\ -3\rightarrow\ ;$ $R(\beta_7)\ -7\rightarrow\ P(\delta 7)\ S(\Phi 7)\ ;$

$Q(\beta_4)\ -4\rightarrow\ S(\gamma_4)\ ;$ $P(\beta_8)\ -8\rightarrow\ S(\Phi_8)\ ;$

$P(\beta_9)\ -9\rightarrow\ S(\gamma_9)\ S(\Phi_9)\ ;$

Etape 3 : La seule règle non standard est la règle 6.

$P(\beta'_6)\ -6\rightarrow\ P(\gamma'_6)\ S(\delta'_6)\ S(\Phi'_6)\ ;$ devient $P(\beta'_6)\ -6\rightarrow\ P(\gamma'_6)\ SS(\alpha_6);$

$SS(\alpha_6)\ -10\rightarrow\ S(\delta'_6)\ S(\Phi'_6)\ ;$

où α_6 est un arbre contenant toutes les variables appartenant à l'ensemble suivant :

$(\ Var(\beta'_6) \cup Var(\gamma'_6)\ ) \cap (\ Var(\delta'_6) \cup Var(\Phi'_6)\ ).$

π : $S(\beta_1)\ -1\rightarrow\ ;$ $P(\beta'_6)\ -6\rightarrow\ P(\gamma'_6)\ SS(\alpha_6)\ ;$

$S(\beta_2)\ -2\rightarrow\ S(\gamma 2)\ ;$ $R(\beta_7)\ -7\rightarrow\ P(\delta 7)\ S(\Phi 7)\ ;$

$Q(\beta_3)\ -3\rightarrow\ ;$ $P(\beta_8)\ -8\rightarrow\ S(\Phi_8)\ ;$

$Q(\beta_4)\ -4\rightarrow\ S(\gamma_4)\ ;$ $P(\beta_9)\ -9\rightarrow\ S(\gamma_9)\ S(\Phi_9)\ ;$

$Q(\beta_5)\ -5\rightarrow\ P(\gamma 5)\ S(\Phi_5);$ $SS(\alpha_6)\ -10\rightarrow\ S(\delta'_6)\ S(\Phi'_6)\ ;$

Etape 4 : Par hypothèse le seul prédicat axiome de π est R. Après ces modifications le prédicat Q n'est plus accessible on peut donc supprimer les règles le concernant : 3,4 et 5.

Si on renumérote les règles , on obtient donc un programme normalisé :

π :

$S(\beta_1)\ -1\rightarrow\ ;$

$S(\beta_2)\ -2\rightarrow\ S(\gamma 2)\ ;$

$SS(\alpha_6)\ -3\rightarrow\ S(\delta'_6)\ S(\Phi'_6)\ ;$

$P(\beta_8)\ -4\rightarrow\ S(\Phi_8)\ ;$

$P(\beta_9)\ -5\rightarrow\ S(\gamma_9)\ S(\Phi_9)\ ;$

$P(\beta'_6)\ -6\rightarrow\ P(\gamma'_6)\ SS(\alpha_6)\ ;$

$R(\beta_7)\ -7\rightarrow\ P(\delta 7)\ S(\Phi 7)\ ;$

7.3 Forme normale pour la Prolog-équivalence

La transformation d'un programme prolog quasi-rationnel déterministe en un programme sous forme normale vue au paragraphe précédent, nous assure l'équivalence simple mais ne nous permet pas de conserver des propriétés opérationnelles telles que "dirigé par les données" (cf chap 5). Pour cela nous sommes obligés de modifier cette transformation de façon à ce que les sous buts soient évalués dans le même ordre que dans le programme original.

1ère étape : La forme des règles est maintenant celle-ci :

$P(\beta_0)$ $-i_0\rightarrow$ liste0-1-de-prédicats $P_1(\gamma_0)$ liste0-2-de-prédicats ;

$P_1(\beta_1)$ $-i_1\rightarrow$ liste1-1-de-prédicats $P_2(\gamma_1)$ liste1-2-de-prédicats ;

.. ..

$P_n(\beta_n)$ $-i_n\rightarrow$ liste n-1-de-prédicats $P(\gamma_n)$ liste n-2-de-prédicats ;

où chacune des listes de prédicats est formée de prédicats d'ordre strictement inférieur à Ordre(P) (=Ordre(P_i)).

Pour toute règle non récursive portant sur un des P_k équivalents à notre représentant canonique P : $P_k(.) -j\rightarrow$ liste-de-prédicats < Pk , liste éventuellement vide, on ajoute une règle correspondant à l'application successive des règles $i_0, i_1, \ldots, i_{k-1}, j$, règle qui sera donc de la forme :

$P(.) \rightarrow$ liste-de-prédicats ;

Ces nouvelles règles ne modifient pas l'ordre d'évaluation des littéraux. Le programme conserve donc la propriété "dirigé par les données" s'il l'avait au départ.

2ème étape : Génération de la règle " $P(\beta) \rightarrow$ liste1-de-prédicats $P(\gamma)$ liste2-de-prédicats ; "

On remplace $P(\beta_0) -i_0\rightarrow$ ... $P_1(\gamma_0)$ ... ; par la règle obtenue par application successive des règles $i_0, \ldots, i_n$:

$P(\beta) \rightarrow$ liste0-1 ,1-1. . n-1-de-prédicats $P(\gamma)$ liste0-2 ,1-2. . n-2-de-prédicats;

Dans ce nouveau programme, les prédicats $P_1, P_2, \ldots, P_n$ ne sont plus récursifs, puisque la règle i_0 a été supprimée.

3ème étape : Après ces manipulations, certaines règles sont de la forme :

$P(\beta) \rightarrow$ liste-de-prédicats $P(\gamma)$ liste-de-prédicats ;

avec liste-de-prédicats d'ordre inférieur strictement à Ordre(P).

On peut remplacer chacune des deux "liste-de-prédicats" par un prédicat unique, notons les Q et Q', en ajoutant une règle de type (1) : $Q(\alpha) \rightarrow$ liste-de-prédicats ;

avec α un arbre contenant toutes les variables de liste-de-prédicats, appartenant aussi à β ou à γ.

$\alpha = \#(x_1, x_2, .., x_p)$ $\quad x_i \in$ Var(liste-de-prédicats)) $\cap$ Var(β,γ)

(idem pour Q')

Les arguments en entrée de Q ou Q' sont ceux contenant les variables en entrée des liste-de-prédicats, il en va de même pour les arguments en sortie.

La propriété "dirigé par les données" reste conservée.

4ème étape : inchangée.

7.4 Réduction du non-déterminisme récursif

Les transformations qui vont être exposées dans ce paragraphe ne conservent pas, en général, l'équivalence logique des programmes (ajout de symboles de prédicats ...) . En revanche les transformés de programmes sont des équivalents opérationnels du programme d'origine.

7.4.1 Cas des programmes quasi-rationnels

Résultat 1 : On ne change rien aux solutions d'un programme, si on applique la transformation suivante : Soit un ensemble E de symboles de prédicats. Tous les littéraux de prédicat P appartenant à l'ensemble E, notons P(α) α étant quelconque, sont remplacés dans le programme par des littéraux de la forme Ψ(P(α)). Les symboles de E ne sont alors plus des symboles de prédicats, mais des symboles fonctionnels. Ψ est un nouveau symbole de prédicat.

Exemple : ADD (Zero,x,x) $\rightarrow$;

ADD (Succ(x),y,Succ(z) $\rightarrow$ ADD (x,y,z) ;

MUL2 (Zero,x,Zero) $\rightarrow$;

MUL2 (Succ(x),y,z) $\rightarrow$ MUL2 (x,y,z') ADD (y,z',z) ;

est identique à :

Ψ(ADD (Zero,x,x)) → ;

Ψ(ADD (Succ(x),y,Succ(z)) → Ψ(ADD (x,y,z)) ;

MUL2 (Zero,x,Zero) → ;

MUL2 (Succ(x),y,z) → MUL2 (x,y,z') Ψ(ADD (y,z',z)) ;

et à :

Ψ(ADD (Zero,x,x)) → ;

Ψ(ADD (Succ(x),y,Succ(z)) → Ψ(ADD (x,y,z)) ;

Ψ(MUL2 (Zero,x,Zero)) → ;

Ψ(MUL2 (Succ(x),y,z)) → Ψ(MUL2 (x,y,z')) Ψ(ADD (y,z',z));

Les nouvelles questions seront alors de la forme Ψ(ancienne question).

La démonstration est simple. En effet, nous montrons dans la suite que la résolution reste inchangée, et si E ne contient pas le symbole de prédicat de la question, les solutions sont alors identiques. Dans le cas contraire, les solutions du transformé seront de la forme Ψ(s), où s est une solution du programme original.

Il est facile de se convaincre que la résolution reste inchangée, car deux littéraux de prédicat Ψ ne sont unifiables que s'ils sont construits à partir du même symbole de prédicat du programme original. L'ensemble des substitutions engendré lors de l'unification est alors, de toute évidence, le même que dans le programme original.

De plus, la complexité n'est pas fondamentalement modifiée : on ne trouvera la règle applicable à un moment donné, qu'après plusieurs tentatives d'unification qui échoueront au premier niveau, c'est à dire tout en haut de l'arbre (surcoût en temps : constant).

Remarque : Il est amusant de noter que tout programme Prolog peut s'écrire à l'aide d'un seul symbole de prédicat.

Etape 1 : Cette étape consiste à appliquer la transformation décrite ci-dessus pour toutes les classes de prédicats (pour la relation d'équivalence définie §2), à raison d'un nouveau symbole de prédicat par classe. On obtient donc facilement un programme Prolog

"Prolog-équivalent" au programme de départ, ne contenant que des règles de la forme (1) ou (2):

(1) . $P(\beta) \rightarrow Q_1(\gamma_1) \ldots Q_n(\gamma_n)$;

avec $Q_i < P$

(2) . $P(\beta) \rightarrow P(\gamma)\ Q(\delta)$;

avec $Q < P$ et une ou plusieurs règles de ce type par prédicat P récursif.

Résultat 2 : On ne change pas l'ensemble des solutions d'un programme Prolog en ajoutant le fait : Egal (x , x) → ; puis en remplaçant chacune des règles de prédicat en tête P :

$P(t_1, t_2 \ldots t_n) \rightarrow$ Queue ; Par la règle

$P(x_1, x_2 \ldots x_n) \rightarrow$ Egal ($ff(x_1, x_2 \ldots x_n)$, $ff(t_1, t_2 \ldots t_n)$) Queue;

Où les x_i sont de nouvelles variables, et où ff est un symbole de fonction quelconque. Queue reste inchangé. En fait, nous avons retardé de façon artificielle l'opération d'unification entre le sous-but courant de prédicat P, et la tête de clause.

Etape 2 : On applique la remarque précédante à toutes les règles récursives. Ceci nous permet de "banaliser" l'ensemble des têtes des clauses récursives d'un même symbole de prédicat.

Etape 3 : On transfère le non déterminisme dans le choix d'une règle récursive parmi un nombre quelconque, vers le choix des règles d'ordre inférieur, de la manière suivante :

Soit l'ensemble de toutes les règles récursives sur P :

$P(x_1, x_2 \ldots x_n) \ -i_0\rightarrow \ P(a_1, a_2 \ldots a_n)$ liste-0-de-prédicats ;

$P(x_1, x_2 \ldots x_n) \ -i_1\rightarrow \ P(b_1, b_2 \ldots b_n)$ liste-1-de-prédicats ;

.. ..

$P(x_1, x_2 \ldots x_n) \ -i_k\rightarrow \ P(k_1, k_2 \ldots k_n)$ liste-k-de-prédicats ;

les a_i , b_i .. k_i étant des termes quelconques les prédicats EGAL de l'étape précédente font partie des liste-i-de-prédicats.

(a) On double de façon artificielle le nombre d'arguments de P dans toutes les règles du programme, sauf dans les règles récursives sur P, à l'aide de nouvelles variables.

$P(t_1, t_2) \rightarrow \ldots$; se transforme en $P(t_1, t_2, z_1, z_2) \rightarrow \ldots$;

$\ldots \rightarrow \ldots P(t_1, t_2) \ldots$; se transforme en $\ldots \rightarrow \ldots P(t_1, t_2, z_1, z_2) \ldots$;

(b) On supprime toutes les règles récursives sur P en introduisant un nouveau symbole Q puis les remplaçant par les règles :

$P(x_1, x_2 \ldots x_n, z_1, z_2 \ldots z_n) \rightarrow P(y_1, y_2 \ldots y_n, x_1, x_2 \ldots x_n)$
$Q(y_1, y_2 \ldots y_n, x_1, x_2 \ldots x_n)$;

$Q(a_1, a_2 \ldots a_n, x_1, x_2 \ldots x_n) \; -i_0\rightarrow$ liste-0-de-prédicats ;

$Q(b_1, b_2 \ldots b_n, x_1, x_2 \ldots x_n) \; -i_1\rightarrow$ liste-1-de-prédicats ;

.. ..

$Q(k_1, k_2 \ldots k_n, x_1, x_2 \ldots x_n) \; -i_k\rightarrow$ liste-k-de-prédicats ;

Cette dernière transformation est la plus délicate à justifier, pour cela nous allons utiliser un exemple où n vaut 1. Ceci n'enlève rien à la généralité du résultat.

Soit le programme suivant :

$P(a) \rightarrow$;

$R(b) \rightarrow$;

$R'(c) \rightarrow$;

r1 : $P(\alpha) \rightarrow P(\beta)\, R(\gamma)$;

r2 : $P(\alpha') \rightarrow P(\beta')\, R'(\gamma')$;

Il vérifie déjà l'existence d'un seul symbole de prédicat par classe. La transformation de l'étape 1 n'est donc pas nécessaire.

Etape 2 :

P(a) → ;
R(b) → ;
R'(c) → ;
Egal (x , x) → ;
P(x) → P(β) Egal (x , α) R(γ);
P(x) → P(β') Egal (x , α') R'(γ');

x n'appartient à aucun des α, α', β, β', γ, γ'.

Etape 3 :

P(a , z) → ;
R(b) → ;
R'(c) → ;
Egal (x , x) → ;
P(x , z) → P(y , x) Q(y , x) ;
Q(β , x) → Egal (x , α) R(γ);
Q(β' , x) → Egal (x , α') R'(γ');

z n'appartient à aucun des α, α', β, β', γ, γ'.

Q(y , x) peut être tantôt unifié à Q(β,x), ce qui simule l'utilisation de la règle r1, tantôt unifié à Q(β',x) ce qui simule l'utilisation de la règle r2. On peut appliquer cette technique pour plus de règles récursives sans aucune complication.

Soit la question P (t) ?

si t = σ(α) , le sous-but de prédicat P issu de l'utilisation de la règle r1 est P(σ'(β)), où σ' est le résultat sur σ de la résolution de R(γ). La même question transformée pour le programme final est P(t , z) ?

Ce qui donne comme sous-buts après une utilisation de la règle récursive : P(y , t) et Q(y , t). C dernier s'unifie à Q (β , x). On a donc comme résultat de cette unification Q(β , t) . Grâce au prédicat Egal, t s'unifie avec α, ce qui nous donne la substitution σ qui, complétée par le résolution de R(γ), nous donne σ'. Le sous-but Q(y, t) est donc effacé avec la substitution σ' à laquelle on ajoute la substitution élémentaire de y par β. Pour finir nous retrouvons bien, par rétroaction, le sous-but P(s'(β) , x) .

Cette technique est illustrée par un exemple au paragraphe 6.2 de ce chapitre.

7.4.2 Cas général

Dans le même esprit qu'au paragraphe précédent, nous allons construire un programme équivalent, du point de vue opérationnel, à celui d'origine.

Voici, comme on peut s'en convaincre assez facilement, un interpréteur Prolog Pur, écrit en Prolog :

Règle ($Tête_1$, $Q_1.Q_2..Q_{K1}$) → ;

Règle ($Tête_2$, $Q_1.Q_2..Q_{K2}$) → ; Programme en donnée: Tête est un littéral,

les $Q_1.Q_2..Q_{Ki}$ sont des listes de littéraux formant le corps des règles.

Règle ($Tête_n$, $Q_1.Q_2..Q_{Kn}$) → ;

Traiter_liste (nil) → ;

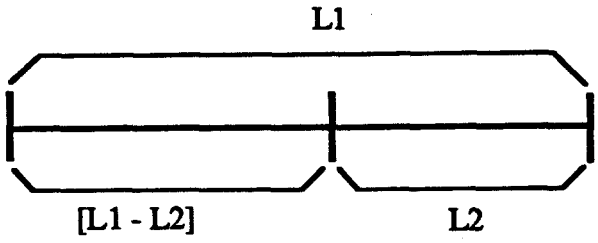
Traiter_liste (but . l) → Traiter (but) Traiter_liste (l) ;

Traiter (but) → Règle (but , sous_buts) Traiter_liste (sous_buts);

Comme on le voit, cet interpréteur est expansif (Traiter appelle Traiter_liste qui appelle Traiter et Traiter_liste) donc il est facile de voir que

Traiter ⊢ .. Traiter .. Traiter .. Traiter ...

Or, ce non déterminisme est du au seul parcours de liste par le prédicat Traiter_liste. Il peut donc être réduit grâce à l'utilisation de la notion de **listes différentielles.** On note DD (L1 , L2) la liste composée des éléments de L1 avec une suppression en queue des éléments de L2.



Ce qui nous donne l'interpréteur suivant :

Règle ($Tête_1$, $[Q_1.Q_2..Q_{K1}.1-1]$) → ;

Règle ($Tête_2$, $[Q_1.Q_2..Q_{K2}.1-1]$) → ;

.

Règle ($Tête_n$, $[Q_1.Q_2..Q_{Kn}.1-1]$) → ;

Traiter (([1 - 1]) →;

Traiter ([but.l1 - l2]) → Règle (but , [l3 - l1])
Traiter ([l3 - l2]);

Programme en donnée : $Tête_i$ est un littéral, $Corps_i$ est une liste de littéraux exprimée sous la forme d'une différence.

Le nouveau programme est donc un interpréteur Prolog avec la stratégie standard, quasi-rationnel déterministe d'ordre 1, n'ayant qu'une seule règle, et des faits.

D'où le théorème suivant :

Théorème : *Tout programme Prolog peut être écrit sous la forme d'un programme opérationnellement équivalent et ayant la forme :*

Règle ( ) → ;

Règle ( ) → ;

.

.

Règle ( ) → ;

Traiter ( ) → ;

Traiter ( ) → Règle ( ) Traiter ( ) ;

Exemple :

Le programme suivant permet de donner l'ensemble des listes constituées d'une permutation des éléments d'une liste donnée. Deux symboles de prédicats sont utilisés, l'un "Perm" est le prédicat principal, il sélectionne tour à tour chacun des éléments de la liste initiale, le second est "Ajouter", et ajoute l'élément sélectionné aux différentes positions possible dans la liste résultat.

Perm (Nil , Nil) → ;
Perm (x.l1, l2) → Perm (l1, l3) Ajouter (x, l3, l2) ;

Ajouter (x , l1, x.l1) → ;
Ajouter (x, y.l1, y.l2) → Ajouter (x, l1 , l2) ;

Ce qui nous donne comme programme opérationnellement équivalent et constitué d'une seule règle d'ordre 1, et d'une base de faits :

(Le symbole de fonction DD désigne une Liste Différentielle).

```
Règle( Perm(nil,nil) , [l-l] ) ->;
Règle( Perm(x.l1,l2), [ Perm(l1,l3).Ajouter(x,l3,l2).l - l ] ) ->;

Règle( Ajouter ( x , l1 , x.l1) , [ l - l ] ) ->;
Règle( Ajouter ( x , y.l1, y.l2 ) , [ Ajouter ( x , l1 , l2 ).l - l ] ) ->;

Traiter( [l - l] ) ->;
Traiter( [b.l1 - l2] ) -> Règle(b, [l3 - l1]) Traiter( [l3 - l2] );
```

Chap 5 -Décision de l'arrêt et complexité-

1) Introduction.

2) Indécidabilité de l'arrêt des programmes d'ordre 1.

3) Programmes de complexité constante.

4) Programmes récursifs : critère d'arrêt et convergence simple.

5) Critère de convergence uniforme.

6) Critère de décision de l'arrêt et complexité de programmes quasi-rationnels déterministes.

1 Introduction

Le Problème de l'arrêt d'un programme étant indécidable, il s'agit dans ce travail de mettre en évidence un critère *formel* et *naturel* assurant l'arrêt d'un programme logique.

- Critère *formel* car il s'agit d'un outil de démonstration de l'arrêt d'un programme logique, critère sur lequel un certain nombre de calculs pourront être faits ainsi qu'une mesure de complexité théorique et pratique lorsque cela est possible.

- Critère *naturel* car il doit refléter l'idée qu'a le programmeur lorsqu'il s'assure que son programme s'arrête. Dans bien des cas l'intuition que l'on se fait sur le comportement d'un programme logique est purement opérationnelle. Le programme pour s'arrêter doit vérifier un *critère de convergence* lors de la résolution, c'est à dire qu'à chaque opération on doit pouvoir se convaincre d'une consommation d'information en "entrée", et d'une création d'information en "sortie".

Nous débordons à ce niveau de l'aspect purement logique des programmes pour en avoir une approche opérationnelle. En effet, le critère cherché fait appel à des notions d'entrée/sortie sur les programmes logiques, donc sur les prédicats, souvent rencontrées dans la littérature sous le nom de "modes" [Mel 1981] [Eud 1986] [Nak 1986].

La notion de *quantité d'information* en entrée ou en sortie est très classique. Pratiquement elle peut être confondue avec la taille des termes manipulés, c'est à dire par exemple, le nombre de caractères nécessaire à leur écriture en extension ou le nombre de noeuds des termes lors d'une écriture sous forme d'arbres.

Comme pour la programmation procédurale, la vérification "à la main" d'un critère d'arrêt pour un programme logique donné revient à associer à chaque clause une fonction strictement décroissante sur un ensemble bien fondé, comme par exemple l'ensemble des tailles des termes manipulés.

Dans le paragraphe n°2, nous présentons une preuve de l'indécidabilité de l'arrêt des programmes ne contenant qu'une seule règle récursive, ce qui explique les limites de notre étude. En fait , nous avons là une confirmation, si besoin est, qu'il est très difficile de maîtriser le comportement des programmes logiques aussi simples soient-ils.

1.1 Exemples

Exemple 1 : Prenons l'exemple du calcul de l'exponentielle. s étant la fonction Successeur sur les entiers

1 Add (0 , x , x) → ;

2 Add (s(x), y , s(z)) → Add (x , y , z) ;

3 Mul (0 , x , 0) → ;

4 Mul (s(x) , y , z) → Mul (x , y , z') Add (y , z' , z) ;

5 Exp (x , 0 , s(0)) → ;

6 Exp (x , s(y), z) → Exp (x , y , z') Mul (x , z' , z) ;

chacune des règles récursives vérifie un critère de convergence sur la taille d'au moins un argument:

Le premier argument pour les prédicats Add et Mul,

Le second argument pour Exp.

Il reste à vérifier qu'il y a bien critère d'arrêt. En effet la taille (nombre de noeuds) des arguments cités plus haut diminue bien d'au moins 1 à chaque utilisation des règles correspondantes. Il faut maintenant vérifier que ces arguments sont bien en entrée, c'est à dire qu'il sont clos au moment de l'appel.

Rappel : Un terme est clos s'il est sans variable.

Cette propriété de clôture est de loin la plus difficile à vérifier "manuellement". Une idée intuitive de la manière dont elle est obtenue s'énonce de la manière suivante :

Pour chaque prédicat on peut vérifier les propriétés suivantes en commençant cette vérification par le prédicat le plus interne, c'est à dire dans l'exemple 1 :

Pour le prédicat Add

1) Si les deux premiers arguments sont clos à l'appel initial, ils le sont toujours.

2) Les deux premiers arguments étant clos à l'appel, le troisième argument est clos au moment de l'effacement.

La propriété 1 se vérifie au niveau de la règle récursive, au regard des variables apparaissant à gauche et à droite.

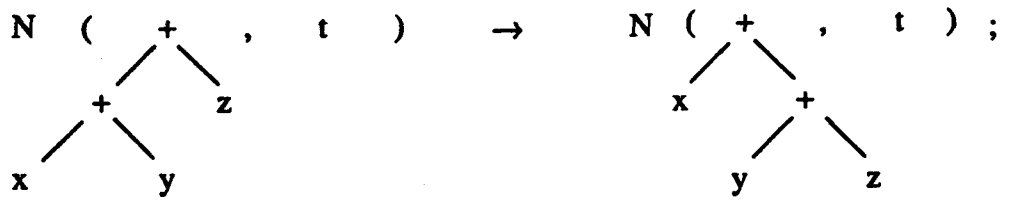
La propriété 2 se vérifie dans le fait.

Cette démarche intuitive est parfaitement formalisée à l'aide du schéma de dépendance d'attributs SDA présenté dans [Der 1985].

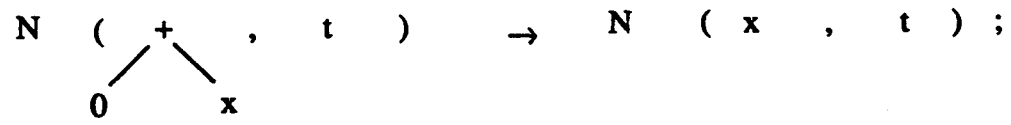
Exemple 2 : Soit le programme d'évaluation d'une expression arithmétique n'employant que les symboles d'addition +, la fonction successeur sur les entiers s et la constante 0. L'évaluation consiste à appliquer tant que c'est possible l'associativité de l'addition, puis d'effectuer l'addition la plus à gauche selon la valeur du premier argument. N (e , t) est vrai si t est le résultat de l'expression arithmétique e.

Nous prenons la notation sous forme d'arbres.

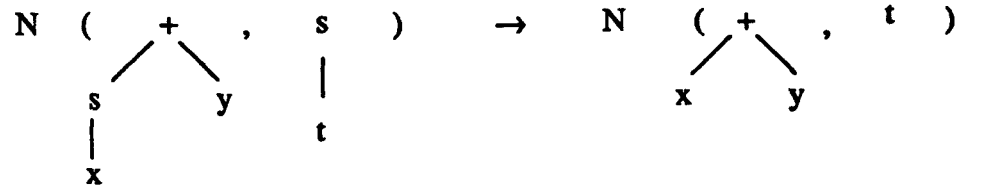
Règle 1 : Associativité de l'addition : $t = (x + y) + z = x + (y + z)$



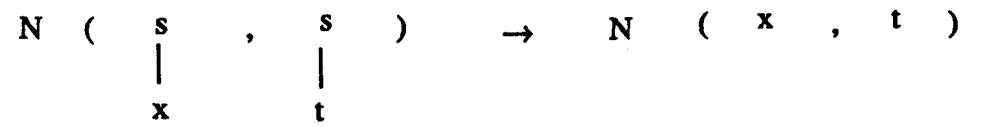
Règle 2 : élément neutre de l'addition : $t = 0 + x$ si $t = x$



Règle 3 : règle d'inférence : t + 1 = (x+1) + y si t = x+y



Règle 4 : t + 1 = s + 1 si t = x



Fait : 0 = 0

N (0 , 0) → ;

Ce programme est linéaire en nombre de pas de résolution par rapport au nombre d'éléments contenus dans le terme à normaliser. La preuve de la linéarité n'est pas immédiate. Elle consiste à trouver une fonction φ de T dans N (T désigne l'ensemble des termes manipulés par le programme Prolog et N l'ensemble des entiers naturels) ayant comme argument le terme courant (ou sous-but courant), et dont le résultat diminue linéairement à chaque pas de résolution. En fait, nous allons tenter de formaliser le fait que le sous arbre gauche du premier argument de chaque littéral diminue d'au moins un noeud à chaque pas de résolution.

Cette fonction φ est définie de la manière suivante :

ν (t) désigne le nombre de noeuds du terme t.

$$\varphi\ (\ +\ ) = \nu\ (t_1) + \varphi\ (t_2)$$

t_1 t_2

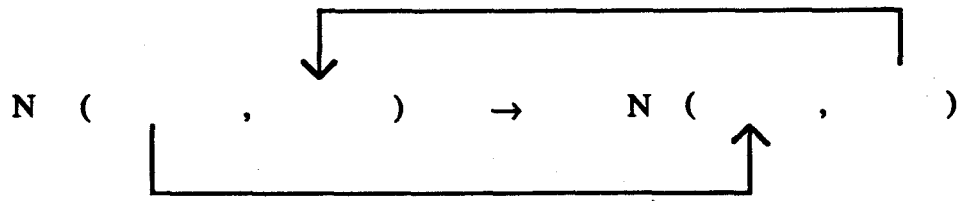
$$\varphi\ (\ s\ ) = \nu\ (s(t))$$

t

$$\varphi\ (\ 0\ ) = 1$$

On remarque que φ (t) ≤ ν (t) pour tout t , et que le φ associé au premier paramètre de chaque sous-but, décroît strictement à chaque pas.

L'annotation associée au programme ci-dessus est : N(↓ , ↑), ce qui donne comme SDA: (cf Chap. 2)



pour chacune des règles récursives.

Notation : Pour simplifier l'expression des entiers , nous noterons sss0 au lieu de s(s(s(0))).

Le schéma induit par le SDA dans l'arbre de preuve du but : N(+(+(ss0,s0) , 0) , y) ? est le suivant : (cf chap. 2)

	N(+ (+ (ss0 , s0) , 0) ,	sss0)
1	↓	↑
	N(+ (ss0 , + (s0 , 0) ,	sss0)
3	↓	↑
	N(+ (s0 , + (s0 , 0) ,	ss0)
3	↓	↑
	N(+ (0 , + (s0 , 0) ,	s0)
2	↓	↑
	N(+ (s0 , 0) ,	s0)
3	↓	↑
	N(+ (0 , 0) ,	0)
2	↓	↑
	N(0 ,	0)

2 Indécidabilité de l'arrêt des programmes d'ordre 1

On pourrait être tenté de déterminer la complexité des programmes Prolog uniquement en fonction de leur structure (ordre pour les programmes quasi-rationnels) sans tenir compte de la façon dont l'information circule lors de la résolution. Le résultat suivant nous montre que c'est insuffisant. Il faut, en effet, aller plus loin dans l'étude du comportement récursif des programmes, pour en déterminer le temps approximatif d'exécution. La hiérarchie établie sur les programmes quasi-rationnels n'aura d'utilité que sur des programmes vérifiant d'autres contraintes que nous allons exposer dans ce chapitre.

Théorème : *L'arrêt de programmes d'ordre 1 est un problème indécidable.*

En effet : Nous avons vu dans le chapitre 4, paragraphe 7, sur la réduction du non-déterminisme récursif, un interpréteur Prolog pur ne contenant qu'une seule règle d'ordre 1. Donc tout programme Prolog peut s'écrire sous la forme d'un programme d'ordre 1. Comme l'arrêt d'un programme Prolog quelconque est indécidable, l'arrêt d'un programme d'ordre 1 est à fortiori indécidable.

La conclusion que nous pouvons apporter à ce théorème, est que le schéma de programme vu ci-dessus, le plus simple possible en dehors du schéma TANT QUE, possède toute la complexité d'un langage de programmation. On en déduit que les résultats généraux que l'on va pouvoir établir sur ce genre de schémas, sont très peu nombreux, et que les résultats d'arrêt et de complexité de programmes Prolog vont sans aucun doute être très restrictifs, l'étude de la structure d'appel des prédicat étant à elle seule très insuffisante.

3 Programmes de complexité constante.

Le définition que nous allons prendre de la complexité ne dépend pas

Définition :

Définition : *Un programme Prolog est de complexité constante si et seulement si toute question admet une solution obtenue en un nombre borné de réécritures ne dépendant que du programme, c'est à dire indépendant de la question.*

Il s'agit de la notion de programmes **bornés** (bounded). Ce sont tous les programmes dont on peut éliminer la récursion.

Propriété : Tout programme d'ordre nul (cf chap. 4) admet un programme logiquement équivalent ne contenant que des faits.

Exemple : $\pi 1 = \pi 2$.

$\pi 1$
PERE (Jean,Jacques) → ;
PERE (Alain,Jean) → ;
MERE (Helene,Jacques) → ;
MERE (Marie,Jean) → ;
PARENT (x,y) → PERE (x,y) ;
PARENT (x,y) → MERE (x,y) ;
GRAND-PERE (x,y) → PERE (x,z) PARENT (z,y) ;

$\pi 2$
PERE (Jean,Jacques) → ;
PARENT (Jean,Jacques) → ;
PERE (Alain,Jean) → ;
PARENT (Alain,Jean) → ;
MERE (Helene,Jacques) → ;
PARENT (Helene,Jacques) → ;
MERE (Marie,Jean) → ;
PARENT (Marie,Jean) → ;
GRAND-PERE (Alain,Jacques) → ;

Propriété : Un programme Prolog est de complexité constante si et seulement si il existe un programme logiquement équivalent d'ordre 0.

Exemples : π_1 est un programme qui vérifie qu'un entier est strictement inférieurs à 10

INF10 (Succ^9(Zero)) → ; [9 < 10].

INF10 (x) → INF10(Succ(x)) ; [x < 10] si [(x+1) < 10].

π_3 est une application d'un prédicat commutatif.

Ces deux exemples donnent une résolution infinie dans une stratégie classique. Bien que l'ensemble des solutions est obtenu en un temps fini par chaînage avant :

$\pi 1 \approx \pi 2$ et $\pi 3 \approx \pi 4$

$\pi 1$

INF10 (Succ9(Zero)) → ;

INF10 (x) → INF10(Succ(x)) ;

$\pi 2$

INF10 (Succ9(Zero)) → ;

INF10 (Succ8(Zero)) → ;

INF10 (Succ7(Zero)) → ;

...

INF10 (Succ(Zero)) → ;

INF10 (Zero) → ;

$\pi 3$

FRERE (Jean,Gaston) → ;

FRERE (Georges,Xavier) → ;

FRERE (x,y) → FRERE (y,x) ;

$\pi 4$

FRERE (Jean,Gaston) → ;

FRERE (Georges,Xavier) → ;

FRERE (Gaston,Jean) → ;

FRERE (Xavier,Georges) → ;

En effet : Soit N le nombre maximum de réécritures pour obtenir toutes les solutions, il nous suffit, pour tout prédicat axiome, d'énumérer les Faits qui peuvent être obtenus en moins de N réécritures. Dans l'exemple $\pi 1$, $N \leq 10$: les Faits du programme d'ordre 0 équivalents sont les réponses à INF10(x) en au plus 10 réécritures.

Dans l'exemple $\pi 3$, $N \leq 2$, les Faits sont les réponses à FRERE(x,y) obtenues après au plus 2 réécritures.

4 Programmes récursifs : critère d'arrêt et convergence simple.

4.1 Rappels et Définitions

Rappel des notations :

- Un programme π est muni de sa grammaire algébrique de dépendance de prédicats G_π associant à chaque prédicat P la classe d'équivalence C_p (cf chap.3).

- Un programme π est muni d'une annotation hérité/synthétisé sous la forme de deux fonctions Her et Syn qui à chaque prédicat P associent les ensembles Her(P) et Syn(P) des positions d'arguments respectivement hérités et synthétisés.

Définition : *A tout prédicat P d'un Programme Prolog on associe le terme p plus grand minorant, au sens de l'ordre classique dans les arbres munis de variables, de l'ensemble des termes t tels que P (t) est une tête de clause.*

$$p = \wedge \{ t \ / \ P(t) \text{ est une tête de clause dans } \pi \}$$

Ceci correspond à la notion d'anti-unification [Lassez,Maher,Marriot].

Notations :

On note V_p l'ensemble des variables de p. (cet ensemble est supposé disjoint de l'ensemble des variables du programme).

On note VH_p le sous-ensemble de V_p des variables apparaissant aux positions héritées du prédicat P.

Pour tout littéral $P(\alpha)$, si $\sigma = mgu(\alpha, p)$ on note:

$H_p(\alpha)$ la substitution σ réduite à VH_p :

si $VH_p = \{ x_1, x_2, .., x_k \}$ on note alors $H_p(t) = \{ t_1, t_2, .., t_K \}$

la propriété suivante est immédiate :

Propriété : P (t) est clos sur Her (p) alors H_p (t) est clos ou alors P(t) n'est pas applicable.

On note alors $||H_p(t)||$ la valeur $||t_1|| + ||t_2|| + ... + ||t_k||$

Définition [Der Mal, 1985]: *Un programme Prolog annoté est dit **dirigé par les données** si et seulement si pour toute **question** dont les arguments en **entrée** sont **clos**, à chaque pas de la SLD-résolution, le **sous-but** courant a ses arguments en **entrée** clos.*

Cette définition est classique [Eud, 1986]. Dans la suite de ce travail nous supposerons toujours, sauf contradiction explicite, que les programmes Prolog étudiés sont annotés et dirigés par les données, et que la question a ses arguments en entrée clos.

Cette contrainte peut sembler forte à priori, mais si l'on y regarde de plus près, elle est satisfaite par la plupart des programmes courants. En effet, classiquement, un prédicat peut être utilisé indifféremment en tant que vérificateur ou générateur. Par exemple

Entier (0) → ;

Entier (sx) → Entier (x) ;

peut aussi bien générer l'ensemble des entiers que vérifier si un élément donné est entier. On y voit en général une puissance d'expression de Prolog. On est alors tenté de dire que le programmeur dans ce cas n'associe pas de position d'entrée et de sortie à son prédicat : en fait il en associe plusieurs , et il est facile de dupliquer l'ensemble des clauses concernées par un prédicat donné autant de fois qu'il y a de façons d'utiliser le prédicat. Ceci pouvant se faire de manière transparente pour le programmeur.

De plus, nous le verrons par la suite, cette contrainte peut être allégée en ne vérifiant les propriétés de clôture que sur des sous-termes des arguments en entrée, les sous-termes étant ceux directement liés au critère de convergence uniforme, c'est à dire ceux apparaissant en argument du critère d'arrêt.

4.2 Enoncé du critère de convergence simple

Le critère que nous allons énoncer est relatif aux différentes classes d'équivalence de prédicats d'un programme prolog au sens de la grammaire algébrique associée. De plus ce critère se vérifie sur les règles dont la tête est dans une classe donnée, ces règles étant sorties de leur contexte, c'est à dire que pour chaque classe on considère le programme Prolog réduit aux seuls prédicats de la classe concernée.

Exemple:

Reprenons l'exemple 1: le programme de calcul de l'exponentielle sera étudié en trois étapes car il y a trois classes distinctes, chacune ne contenant qu'un seul prédicat. Les programmes réduits sont construits en supprimant simplement tous les littéraux dont le prédicat ne fait pas partie de la classe.

étape 1 on étudie le critère pour le programme réduit :

1 Add (0 , x , x) → ;
2 Add (sx, y , sz) → Add (x , y , z) ;

étape 2

3 Mul (0 , x, 0) → ;
4 Mul (sx , y , z) → Mul (x , y , z') ;

étape 3

5 Exp (x , 0 , s0) → ;
6 Exp (x , sy , z) → Exp (x , y , z');

Critère de convergence simple:

Dans un programme Prolog π dirigé par les données, une classe C_p vérifie le critère de convergence simple si et seulement si :

π' désignant le programme π réduit aux seuls prédicats de C_p (π' est le projeté de π sur l'ensemble des prédicats C_p)

Pour tout prédicat P de C_p, il existe un entier A et il existe un sous-ensemble U_p de VH_p tels que si P(t) et P(t') sont deux sous-buts de la même branche de l'arbre de résolution séparés par au moins A noeuds, P(t) étant clos sur tous ses arguments en entrée, alors la taille de la substitution H_p(t) réduite aux variables de U_p est strictement plus grande que celle de H_p(t') réduite à U_p.

Plus formellement:

$\forall$ P prédicat de π ,$\exists$ A $\in$ N et $\exists$ $U_p \subseteq VH_p$ tels que $\forall$ P(t) clos en entrée si P(t) $-B\rightarrow$ P(t') par π réduit aux seuls prédicats de C_p avec B > A, alors

$$||H_p(t)/U_p|| > ||H_p(t')/U_p||$$

Exemples:

A) Le programme qui suit ne vérifie pas le critère de convergence simple, bien qu'il s'arrête en un temps linéaire par rapport à la taille de la question :

Q(0) →;

R(0 , 0) →;

Q(sx) → R (x , y) Q (y);

R(sx, sy) → R (x, y);

En effet, le programme réduit à la classe du prédicat Q est :

Q (0) →;

Q (sx) → Q (y);

et ne vérifie pas la condition du critère. La branche close de la question est perdue en un pas de résolution dans ce programme réduit, alors que le programme complet est bien dirigé par les données pour l'annotation Q(↓) , R(↓,↑).

B) L'exemple 1 du paragraphe 1 de ce chapitre vérifie le critère de convergence simple. En effet, dans chacune des boucles Prolog, un argument en entrée décroît indépendamment des autres prédicats.

1 Add (0 , x , x) →;

2 Add (s(x), y , s(z)) → Add (x , y , z);

3 Mul (0 , x , 0) →;

4 Mul (s(x) , y , z) → Mul (x , y , z') Add (y , z' , z);

5 Exp (x , 0 , s(0)) →;

6 Exp (x , s(y), z) → Exp (x , y , z') Mul (x , z' , z);

Il est aisé de vérifier que l'annotation Add(↓,↓,↑) Mul(↓,↓,↑) Exp(↓,↓,↑) donne au programme la propriété "dirigé par les données". De plus, on peut voir que :

- le second argument de Exp est hérité et diminue d'au moins un nœud par pas de résolution,

- Le premier argument de Mul et de Add vérifie la même propriété.

notons $\wedge Exp = Exp(u,v,w)$ le plus grand minorant de l'ensemble des têtes de clauses de prédicat Exp, $VH_{exp} = \{ u,v \}$, $U_{exp} = \{v\}$. $\forall$ Exp(t) close en entrée, si Exp(t') est conséquence immédiate (après une utilisation de la règle 6) de Exp(t), alors :

$$|| t / \{v\} || = || t' / \{v\} || + 1$$

C'est à dire que la taille du deuxième argument du prédicat Exp diminue de 1 à chaque pas de résolution.

On peut vérifier qu'il en est de même pour le premier argument de Add et Mul.

Théorème : *Si toutes les classes de prédicats d'un programme Prolog π vérifient le critère de convergence simple, et si π est dirigé par les données, alors quel que soit la question $Q(\alpha)$ close en entrée, la résolution Prolog s'arrête. On dira alors que le programme Prolog vérifie le critère de convergence simple.*

Remarque: Nous parlons dans ce théorème de l'arrêt de la **résolution classique Prolog II** car le critère n'est applicable que sur des programmes **dirigés par les données**, or cette dernière contrainte est par définition étroitement liée à la stratégie de résolution.

En effet : la preuve s'effectue par récurrence sur l'ordre des classes de prédicats.

Définition : Soit C une classe d'équivalence de prédicat, elle est dite d'ordre k si et seulement si :

- k = 0 : la classe est réduite à un seul prédicat n'apparaissant qu'en fait.

- k > 0 : k-1 est le plus grand ordre de classe des prédicats dépendants de ceux de C.

Il s'agit en fait de l'ordre naturel sur les prédicats présenté au chapitre 3.

Il est immédiat que pour tout prédicat de classe zéro la résolution s'arrête en un temps constant.

Soit P un prédicat récursif de classe k. Supposons que pour tout prédicat de classe inférieure à k la résolution soit finie et que pour la question P (t) close sur Her (P) la résolution soit infinie. Ceci implique qu'il existe au moins un prédicat R engendré une infinité de fois par la question P(t), R étant de classe k. Or π est dirigé par les données, c'est à dire que tous les sous-buts de la forme R (u) sont clos en entrée, et il en existe

une infinité. Soit $u_1,u_2,...,u_n,...$ une suite infinie de termes tels que $R(u_i) \xrightarrow{-A_i} R(u_{i+1})$ avec $A_i \geq A$. Comme Cp vérifie le critère de convergence simple, pour tout i :

$$||H_R(u_{i+1})|| < ||H_R(u_i)||$$

or $H_R(u_1)$ est fini d'où contradiction.

4.3 Complexité

Remarque : Le critère de convergence simple nous permet de nous assurer de l'arrêt d'un programme Prolog, mais pas d'en borner le temps de résolution.

Exemple :

```
Q(0,y)      →  P(y);
Q(sx,y)     →  Q(x,y);
P(sy)       →  R(x,y) Q(sx,y);
P(sy)       →  P(y);
```

avec $C_p = \{ P, Q \}$ et l'annotation $P(\downarrow)$; $Q(\downarrow,\uparrow)$; $R(\uparrow,\downarrow)$.

Il est facile de vérifier que le programme est bien dirigé par les données si la partie concernant R l'est, et qu'il vérifie le critère de convergence simple pour la classe C_p. Pourtant, il est difficile voire impossible de borner la résolution d'une question sur Q, car elle dépend étroitement de la taille du résultat $\sigma(x)$ du prédicat R(x , y), cette taille pouvant être inversement proportionnelle à celle de $\sigma(y)$, σ étant la substitution locale lors de l'effacement de R.

4.4 Indécidabilité du critère de convergence simple

Le critère que nous venons d'énoncer nous permet, dans certains cas, de vérifier l'arrêt de programmes Prolog "à la main". Une question se pose alors immédiatement: Est-il possible d'établir automatiquement qu'un programme donné vérifie ou non le critère ?

A priori le critère de convergence simple est assez contraignant. Nous allons voir qu'il est malgré tout indécidable.

Théorème : *Il est indécidable de savoir si un programme Prolog quelconque dirigé par les données vérifie ou non le critère de convergence simple.*

En effet : Nous allons pour cette preuve utiliser un schéma de programme Prolog codant le problème de Post.

Le théorème de Post nous dit qu'il est indécidable, étant données deux suites finies quelconques S1 = $(m_1,m_2,....,m_K)$ et S2 = $(n_1,n_2, ..., n_K)$ de mots sur un alphabet fini Σ, de savoir s'il existe une suite finie d'indice $(i_1,i_2,....,i_l)$ tels que les deux mots $m_{i_1}m_{i_2}...m_{i_l}$ et $n_{i_1}n_{i_2}...n_{i_l}$ soient égaux.

Nous dirons qu'un mot donné vérifie Post pour les deux suites de mots S1 et S2 s'il se décompose effectivement en deux découpages selon les conditions du théorème.

Nous nous proposons d'écrire un programme Prolog acceptant en question un prédicat dont l'argument est un mot, possédant comme base de faits la description des deux suites S1 et S2, et vérifiant le critère de convergence simple si et seulement si le mot de la question ne vérifie pas Post pour les suites S1 S2. Si un tel programme existe, c'est une preuve de l'indécidabilité du critère.

Pour plus de clarté, nous reprenons les notations du théorème, les mots étant codés sous la forme de listes de lettres.

Le programme contient les deux clauses :

```
Q(nil, nil, z)   →   P(z);

P(x)             →   Q(x,x,x);
```

pour chaque couple (m_i , n_i) le programme Prolog contient une clause de la forme :

Q (m_i.x , n_i.y , z) → Q (x , y , z) ; le z sert à conserver la liste initiale.

La question est P(m); m est clos

Dans le cas où m vérifie Post pour les suites S1 S2, le programme ne s'arrête pas et donc à fortiori ne vérifie pas le critère de convergence simple. Dans le cas contraire le seul prédicat effectivement récursif, c'est à dire qui s'appelle effectivement lui même, est Q. La tête de clause de la forme Q (nil , nil , .) ne pouvant jamais être unifiée avec le but courant. Q vérifie alors le critère de convergence simple avec A valant 1.

5 Critère de convergence uniforme.

5.1 introduction

Il s'agit dans ce paragraphe de restreindre la notion de convergence afin de permettre de borner, dans le cas où un programme vérifie le nouveau critère, le temps de résolution pour un but donné.

5.2 Enoncé du critère de convergence uniforme

Le nouveau critère que nous allons énoncer est, contrairement au précédent, relatif au programme et non à une classe de prédicats du programme. En effet le problème que nous avons rencontré lorsque nous avons voulu évaluer le temps de résolution, est que le critère simple est, comme son nom l'indique, vérifié sur le programme partie par partie, indépendamment du contexte. Le critère de convergence uniforme vérifie la convergence de la taille des données manipulées toutes les A étapes de la résolution, sans tenir compte du prédicat du sous-but courant.

Critère de convergence uniforme

Un programme Prolog π vérifie le critère de convergence uniforme si et £seulement si :

Il existe un entier A tel que pour tout couple de prédicats P,Q de π il existe un sous-ensemble non vide U_p de VH_p, et un sous-ensemble non vide U_q de VH_q tels que si P(t) et Q(t') sont deux sous-buts de la même branche de l'arbre de résolution séparés par au moins A noeuds, P(t) étant clos sur tous ses arguments en entrée, alors la taille de la substitution H_p(t) réduite aux variables de U_p est strictement plus grande que celle de H_q(t') réduite à U_q.

Plus formellement:

$\exists$ A,a $\in$ N tel que $\forall$ P et Q prédicats de π , $\exists$ $U_p \subseteq VH_p$ et $U_q \subseteq VH_q$ tels que si P(t) est clos en entrée et si P(t) $-A\rightarrow$ Q(t') par π , alors

$$||H_p(t)/U_p|| \geq ||H_q(t')/U_q|| + a$$

Théorème : *Si un programme Prolog dirigé par les données vérifie le critère de convergence uniforme, la résolution est alors de profondeur au plus égale à* $||H_p(t)|| \times A + a$ *. La question* $P(t)$ *étant close en entrée.*

En effet : La démonstration de ce théorème est immédiate, car tous les A descentes d'un noeud dans l'arbre de résolution, on enlève la constante a à une valeur finie inférieure à $||E_p(t)||$.

Théorème : *Il est indécidable, étant donné un programme Prolog quelconque dirigé par les données, de savoir s'il vérifie le critère de convergence uniforme.*

En effet : la démonstration est strictement la même que pour le critère de convergence simple.

Ce nouveau critère est très restrictif, et ne présente qu'un intérêt mineur. Il est vérifié par un sous ensemble des programmes **uniformément linéaires**, ce qui est de toute évidence exceptionnel. Il est quand même à noter le caractère indécidable d'un tel critère.

L'exemple 1 du paragraphe 1.1 qui nous donne un programme de calcul de l'exponentielle vérifie le critère de convergence simple sans pour autant vérifier le critère de convergence uniforme. En effet, le calcul séparé de chaque opération converge sur au moins un argument en entrée, le premier pour les opérations d'addition et de multiplication, le second pour l'opération d'exponentiation. Mais il est immédiat que la taille des entrées à chaque appel de la multiplication par le prédicat Exp est strictement croissante. Le premier appel est de la forme Mul (t , s0 ,), le second Mul (t , t ,) puis vient Mul (t , t^2 ,) etc ...

Remarque : Les critères de convergences simple et uniforme sont uniquement fondés sur une consommation d'information close de la question. Il est immédiat que nous aurons les même résultats si on considère le cas dual où l'information est croissante en sortie, c'est à dire croissante vers un argument fermé du fait. L'unification ne pourra alors plus donner un résultat positif au delà d'un certain nombre de pas de résolution.

6 Décidabilité de l'arrêt et complexité de programmes Quasi-rationnels Déterministes.

6.1 Introduction

Dans ce paragraphe, nous allons étudier plus particulièrement la classe des programmes Quasi-rationnels et déterministes (QRD). Cette classe est décrite en détail dans le chapitre 3. Pour simplifier l'exposé nous ne considérons que les programmes constitués de clauses du type

$$P() \rightarrow Q_1() Q_2() \ldots Q_k()$$

et du type

$$P() \rightarrow Q() P() Q'()$$

Où Q, Q' et les Q_i sont strictement plus petits que P au sens de l'ordre d'appel des prédicats.

Nous avons vu que tout programme QRD pouvait être transformé en un programme équivalent vérifiant cette propriété.

De plus, nous supposons établi que les programmes étudiés sont dirigés par les données. Ces différentes restrictions peuvent paraître à priori excessives, mais lorsqu'on y regarde de plus près, il s'avère qu'elles sont vérifiées par la plupart des applications classiques. En fait cela répond à une façon de programmer en prolog qui, si elle s'éloigne de l'aspect purement logique des programmes, n'en est pas moins répandue. On peut interpréter chaque prédicat comme une procédure ayant des paramètres d'entrée et de sortie à la place des arguments.

6.2 Rappel des résultats sur la boucle simple Prolog

L'étude d'une régle récursive simple Prolog [Dau 86] [Dev 87] nous a apporté les résultats suivant vus plus en détail au chapitre 3 :

Soit un programme Prolog de la forme :

$P(\alpha) \rightarrow$;

$P(\beta) \rightarrow P(\gamma)$;

P(t) ?

Résultat 1 : La résolution classique sans test d'occurrence est susceptible de ne pas s'arrêter si et seulement si l'unifié sans renommage des variables de β et γ existe.

Résultat 2 : La résolution classique avec test d'occurrence est susceptible de ne pas s'arrêter si et seulement si le GOP unifié sans renommage des variables de β^1 et de γ^0 existe (on le note g) et vérifie la condition de restriction sur les pondérations de boucles (RPB).

Résultat 3 : Soit $t_{0,n}$, $t_{1,n}$,..., $t_{n,n}$ les termes les plus généraux tels que: (sous-entendu: sans ajout d'information supplémentaire par la propriété suivante)

$$P(t_{0,n}) \rightarrow P(t_{1,n}) \rightarrow \ldots \rightarrow P(t_{n,n})$$

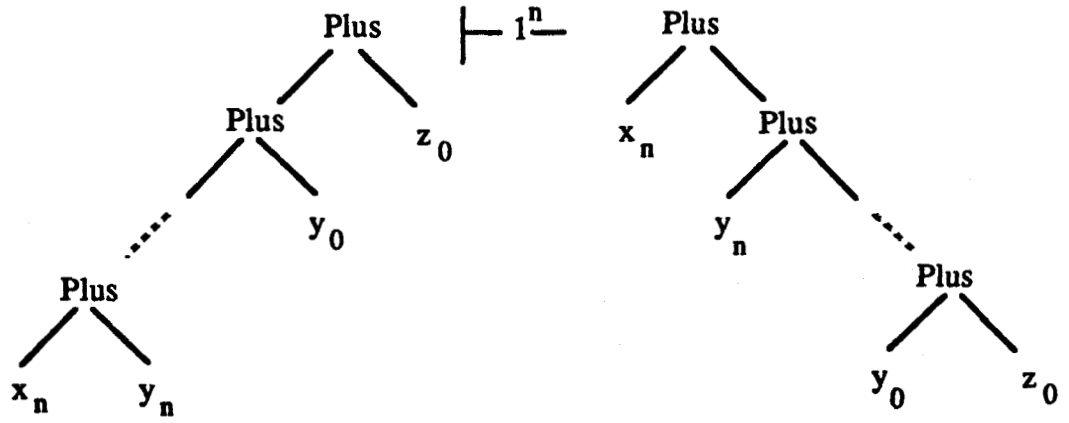
Soit CH(g) l'ensemble infini des chemins finis de $\beta^1 \vee \gamma^0$, CH(g) se décompose en trois sous-ensembles distincts qui sont :

.Gauche(g) l'ensemble des chemins déroulant une boucle positive de g et effectivement poussants dans la suite $t_{0,0}$, $t_{0,1}$,..., $t_{0,n}$ c'est à dire que si m est un chemin de Gauche(g) maximal dans $t_{0,k}$ alors il existe k' > k tel que m n'est pas maximal dans $t_{0,k'}$.

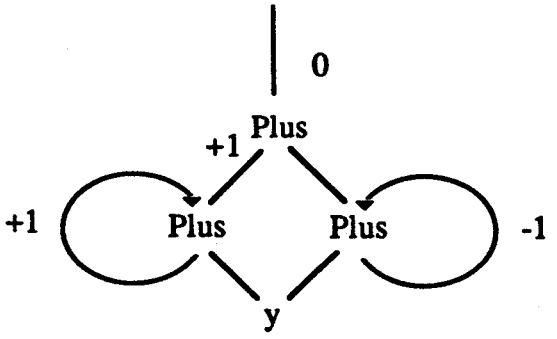
Prenons l'exemple de l'associativité de l'addition. Considérons la règle :

Assoc (Plus (Plus (x , y) , z) → Assoc (Plus (x, Plus (y , z)) ;

n applications successives donnent :



Le GOP unifié g associé à la règle est :



Gauche(g) = {11, 111 , 1111, ... }

.Droite(g) l'ensemble des chemins déroulant une boucle positive de g effectivement poussant dans la suite $t_{0,0}$, $t_{1,1}$,..., $t_{n,n}$ c'est à dire que si m est un chemin de Droite(g) maximal dans $t_{k,k}$ alors il existe k' > k tel que m n'est pas maximal dans $t_{k',k'}$.

Dans l'exemple : Droite(g) = {22 , 222 , 2222 , ... }

.Constant(g) l'ensemble des chemins bouclants dans g qui ne sont dépliés dans aucun des $t_{0,i}$ ni des $t_{i,i}$, union celui des chemins non bouclants de g.

C'est à dire Constant(g) = CH(g) - (Gauche(g) + Droite(g))

Dans l'exemple : Constant (g) = { ε, 12, 112 , 1112 , ... , 21 , 221 , 2221 , ... }

Si on note G(g,n) l'ensemble des chemins de Gauche(g) maximaux dans $t_{0,n}$ (rem: Gauche(g) est l'union des G(g,n) pour tout n positif)

Dans l'exemple : G(g,n) = { 1^n }

alors la longueur des chemins de G(g,n) est linéaire par rapport à n. C'est à dire qu'il existe quatre constantes a,a',b,b' telles que

$$\forall m \in G(g,n) \quad \text{on a} \quad an+b \leq |m| \leq a'n+b'$$

La même remarque peut être faite pour D(g,n) l'ensemble des chemins de Droite(g) maximaux dans $t_{n,n}$.

6.3 Critère d'arrêt et complexité de programmes QRD

Nous avons énoncé précédemment tous les éléments nécessaires à la décidabilité de l'arrêt des programmes quasi-rationnels déterministes. En effet, le critère de convergence simple est vérifié par un programme si chaque cycle d'appel de prédicat est *contrôlé* par la consommation d'information en entrée de la boucle. Ce critère indécidable dans le cas général devient décidable dans le cas de programmes quasi-rationnels déterministes car chaque cycle de tels programmes est comparable à une boucle *tant que* élémentaire dont on connait les arguments en entrée, et dont il est facile, grâce au calcul du GOP associé, de déterminer la place de l'information qui va être consommée par la boucle. De plus, le calcul du Gop associé nous renseigne sur la taux de consommation, c'est à dire la vitesse à laquelle l'information en entrée va être consommée, ainsi que le taux de génération, c'est à dire la vitesse à laquelle l'information en sortie va être générée. Il est alors possible de calculer de manière précise la complexité de tels programmes. C'est ce que nous nous proposons de faire dans ce paragraphe.

6.3.1 Décidabilité de l'arrêt des programmes QRD.

Théorème : *Le critère de convergence simple est décidable pour les programmes quasi-rationnels déterministes dirigés par les données.*

En effet, soit π un programme dirigé par les données, quasi-rationnel déterministe. Les régles de π sont soit de la forme :

$P() \rightarrow Q_1() \; Q_2() \; ... \; Q_n();$

soit de la forme :

$P() \rightarrow Q() \; P() \; Q'();$

où il n'existe qu'une seule régle récursive pour un prédicat P donné, et où les Qi, Q et Q' ne peuvent pas avoir P comme sous-but.

Pour avoir une idée plus intuitive du style de programmes envisagé, nous dirons qu'il s'agit des programmes dont le graphe de dépendance des prédicats associé au programme est linéaire avec une seule boucle au plus par noeud:

Exemple : Le programme repris de l'introduction du chapitre 3.

1 Add (0 , x , x) → ;

2 Add (sx, y , sz) → Add (x , y , z) ;

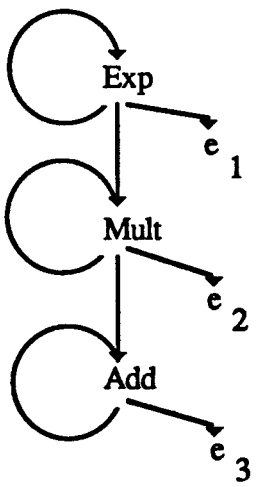
3 Mul (0 , x , 0) → ;

4 Mul (sx , y , z) → Mul (x , y , z') Add (y , z' , z) ;

5 Exp (x , 0 , s0) → ;

6 Exp (x , sy , z) → Exp (x , y , z') Mul (x , z' , z) ;

De graphe de précédance ou de dépendances des prédicats



Ces programmes sont supposés dirigés par les données, c'est à dire que chaque prédicat possède des arguments en entrée toujours clos à l'appel, et des arguments en sortie toujours clos au moment de l'effacement. Cette contrainte pouvant être allégée en ne considérant qu'une clôture partielle des arguments en entrée.

Définition : *L'argument d'un sous-but est dit consommé par une régle récursive de GOP g, si et seulement si il contient au moins un chemin de Gauche(g).*

Exemple : Pour la règle de l'associativité, tout sous-but ayant au moins un chemin (maximal ou non) de la forme 11m , m étant quelconque, est consommé.

Cette définition entraîne la propriété suivante sur les arguments consommés d'un prédicat récursif:

Propriété : Si un prédicat récursif possède un argument en entrée consommé, alors il vérifie le critère de convergence simple, et le nombre d'appels récursifs de ce prédicat lors de la résolution est linéairement borné en fonction de la hauteur de l'argument consommé au moment de l'appel initial.

En effet : L'argument en entrée consommé possède par définition au moins une branche m fermée appartenant à Gauche(g), donc il existe un n (le plus grand possible) tel que m soit maximal dans $t_{0,n}$. n est le plus grand nombre d'appels récursifs possible pour cet argument. n est de l'ordre de $(a/b)|m|$, a et b étant des constantes et $|m|$ la longueur de m.

Lemme 1 : Un programme Prolog quasi-rationnel déterministe dirigé par les données vérifie le critère de convergence simple si et seulement si chaque prédicat récursif possède au moins un argument en entrée consommé.

La démonstration de ce lemme est immédiate et découle directement de la propriété énoncée ci-dessus, car si un prédicat possède un argument en entrée consommé, celui-ci vérifie le critère de convergence simple, et dans l'autre sens si dans tout cycle une branche en entrée est strictement consommée, les cycles étant dans le cas de programme QRD réduits à une seule règle, le prédicat récursif de chaque règle récursive doit avoir au moins un argument en entrée consommé. Ceci est une conséquence des résultats sur les Gop's. En effet toute branche poussante dans la suite $t_{0,0}$, $t_{0,1}$,...., $t_{0,n}$ (resp. $t_{0,0}$, $t_{1,1}$,...., $t_{n,n}$) est obligatoirement une boucle positive (resp. négative) du Gop associé à la règle.

Preuve du théorème : On peut décider pour chaque prédicat récursif si au moins un argument en entrée est consommé. La méthode appliquée à une régle récursive simple s'applique sans problème à chaque règle récursive.

6.3.2 Calcul de la complexité des programmes QRD.

Le calcul de la complexité des programmes QRD est déduit du calcul des taux de croissance des branches dépliées des boucles positives et négatives du Gop associé aux différentes règles récursives. En effet, à chaque règle récursive nous pouvons associer six constantes A, B, C, A', B' et C' telles que toute branche dépliée d'une boucle positive (resp. négative) dans $t_{0,n}$ pour $n > C$ (resp. dans $t_{n,n}$ pour $n > C'$), alors:

$$An+B \leq |m| \leq A'n+B'.$$

m est dit de taille linéairement bornée.

Nous allons fixer les idées en traitant l'exemple de la multiplication:

1 Add (0 , x , x) → ;

2 Add (sx, y , sz) → Add (x , y , z) ;

3 Mul (0 , x , 0) → ;

4 Mul (sx , y , z) → Mul (x , y , z') Add (y , z' , z) ;

Ce programme est dirigé par les données pour les modes (entrée,entrée,sortie) des deux prédicats (ce résultat est exposé au chapitre 2).

La complexité du prédicat Add (le nombre de pas de résolution en fonction du but de prédicat Add) est immédiatement déduite des constantes calculées sur le Gop associé à la règle 2. En effet, le premier argument est en entrée, il est consommé par la règle récursive à raison d'un noeud par pas. La complexité est donc égale à T , T étant la hauteur du premier argument du sous-but de prédicat Add.

De même le prédicat Mul est de complexité égale à T , T étant la taille du premier argument du but initial.

La complexité globale se déduit du fait que la taille de l'argument en entrée consommé par Add et issu de l'application de la règle 4 est constante. Elle est égale à la hauteur du second argument du but initial. On obtient donc $\| arg1 \| \times \| arg2 \|$ produit des hauteurs des deux premiers arguments du but initial.

Nous associons à chaque prédicat une fonction *temps d'effacement* indiquant le nombre de pas de résolution maximum nécessaire à l'effacement d'un sous-but de symbole de prédicat P, en fonction de la taille de l'argument en entrée consommé de P. Cette fonction est *linéairement bornée* pour les programmes du type **Tant que**.

Pour simplifier les démonstrations, nous considèrerons que la fonction temps d'effacement est linéaire pour le TANT QUE. Cela ne changera rien au principe du calcul, mais au lieu de calculer deux fonctions du même type, l'une minorant l'autre majorant le résultat, nous n'en calculerons qu'une.

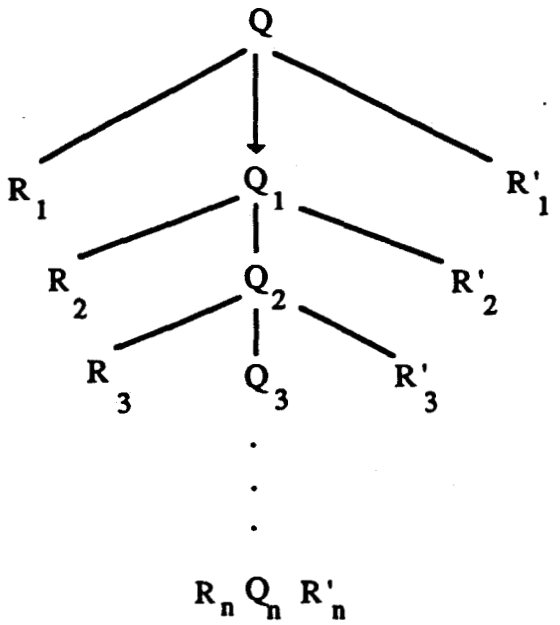
Soit la règle générale d'un programme QRD :

$r : Q(\alpha) \rightarrow R(\gamma)\, Q(\beta)\ R'(\gamma')$;

Nous supposons connues les fonctions temps d'effacement φ_R et $\varphi_{R'}$ des prédicats R et R'. Il s'agit donc de déduire la fonction φ_Q du prédicat Q.

Il existe deux constantes A et B, telles que si m est la branche fermée en entrée et consommée du sous-but Q(t), alors le nombre d'utilisations consécutives de la règle r est $A \times |m| + B$. De plus il existe deux constantes C et D telles que la taille maximum des substitutions obtenues sur les variables de γ ou de γ' au bout de n pas ($n \leq A \times |m| + B$) est $C \times n + D$.

Nous pouvons schématiser la résolution d'une question Q(t)? ayant une branche close et consommée par r, de la manière suivante:



L'ordre d'effacement des littéraux est de la gauche vers la droite, puis la racine. Sachant que n est une fonction linéaire de la taille de la question sur Q(t). Sachant que la suite $R_1, R_2, .., R_k, .. , R_n$ est fermée en entrée et que la taille des substitutions qui lui sont associées est linéairement croissante par rapport à k. Nous pouvons déduire la taille des substitutions associées aux éléments de la suite des R'_i de la manière suivante:

On note $\| Q_i(e) \|$ la taille de la substitution en entrée du littéral Q_i. De la même manière, on note $\| Q_i(s) \|$ la taille des substitutions du littéral Q_i lors de son effacement.

Le programme vérifie le critère de convergence simple. Donc $\| Q_i(e) \| \leq Ci + D$ et $n \leq A \| Q(t) \| + B$.

Les substitutions en entrée de R_i sont issues des substitutions en entrée sur Q_{i-1}. Ce qui donne la formule $\| R_i(e) \| \leq Ci + D$. D'où $\| R_i(s) \| \leq \varphi_R(Ci + D)$

Le nombre de pas de résolution nécessaire à l'effacement de R_j est donc $\varphi_R(C \times j + D)$. Et la substitution finale lors de son effacement est aussi de l'ordre de $\varphi_R(C \times j + D)$. Ce qui donne $\| R'_i(e) \| \leq \varphi_R(C \times j + D)$. Donc le nombre de pas nécessaire à l'effacement de R'_i est de l'ordre de $\varphi_{R'}(\varphi_R(C \times j + D)))$.

Ce qui donne :

$$\varphi_Q(k) = \sum_{j=1}^{A \times k+B} (\varphi_R(C \times j + D) + \varphi_{R'}(\varphi_R(C \times j + D)) + 1)$$

Si R et R' se réduisent à un TANT QUE, c'est à dire sont d'ordre 1. Alors φ_R et $\varphi_{R'}$ sont linéaires, donc φ_Q est de l'ordre de n^2. Par récursion on obtient facilement,

Si Q est un prédicat d'ordre p alors $\varphi_Q(n)$ est de l'ordre de $n^{(p-1)^2}$.

Remarque : Si le programme dirigé par les données est de la forme d'un représentant canonique pour l'équivalence logique définie au paragraphe 6 et 7 du chapitre 4, alors le prédicat R n'existe pas et la formule de calcul de φ_Q est la suivante:

$$\varphi_Q(k) = \sum_{j=1}^{A \times k+B} (\varphi_R(C \times j + D) + 1)$$

Si Q est un prédicat d'ordre p alors $\varphi_Q(n)$ est de l'ordre de n^p.

Chap 6 - Implémentation de résultats sur le "tant que" Prolog

1) Introduction.

2) Détection et calcul du GOP associé à une règle récursive.

3) Systèmes d'équations.

4) Calcul dynamique du temps d'invariance.

5) Algorithme de simulation.

6) Quelques exemples.

1 Introduction

Le présent chapitre fait état d'un travail réalisé sur un interpréteur Prolog écrit en Pascal. Le choix du Pascal, sans doute peu judicieux, est du à l'utilisation, à la base, de l'interpréteur Micro-Prolog présenté dans la thèse d'état de M. Van Caneghem. Mais très vite, pour simplifier l'ensemble des modifications que nous devions y apporter, nous avons, presque totalement, réécrit l'interpréteur en privilégiant la clarté des structures de données au dépend, peut être, de l'efficacité (aucun test de performance n'a pu, à l'heure actuelle, être fait).

Le travail effectué est une application partielle des résultats présentés dans [DEV 87], et a vocation d'être utilisé dans un univers d'aide à la programmation et d'optimisation de programmes.

Il s'agit de

- détecter les règles du programme susceptibles d'avoir un comportement du type "Tant que" (détection naïve),

- calculer le graphe orienté pondéré simulant l'exécution infinie de chacune de ces règles, (on détecte alors si une occurrence systématique aura lieu),

- calculer dynamiquement (cf § suivant) les différents taux de croissance des termes manipulés par la règle,

- dans le cadre de l'aide au programmeur, visualiser graphiquement le comportement de la règle,

- stopper, lors de la résolution, l'évaluation d'un littéral par une règle ne pouvant plus (voir critère plus loin) donner de solution.

J'ai réalisé les trois premières parties ainsi que l'interpréteur Prolog structuré dans le cadre de ce mémoire, les deux dernières ont été faites par Melle Rajoharison, étudiante de DESS Génie informatique option intelligence artificielle et génie logiciel, lors d'un stage que j'ai encadré.

2 Détection et calcul du GOP associé à une règle récursive

La détection des règles susceptibles de se comporter en "tant que" Prolog se fait de manière naïve : même symbole de prédicat à gauche et à droite de la règle, une seule occurrence à droite, une seule règle récursive ayant en tête le symbole de prédicat.

Le calcul du Gop se fait par unification entre les deux littéraux de tête et du corps de même symbole de prédicat, avec les pondérations en tête respectivement 1 et 0. Il s'effectue selon l'algorithme présenté au chapitre 3.

L'existence de ce Gop ainsi que la vérification du test de restriction sur les pondérations de boucles (test RPB) nous assure la possibilité d'itérer la règle une infinité de fois. Dans ce cas, l'interpréteur Prolog classique entre, en général, dans une boucle infinie.

Le détail des deux opérations ci-dessus est effectué au chapitre 2.

3 Systèmes d'équations

L'itération d'une régle récursive Prolog se décompose en deux phases (cf chap3). La première, appelée "effets de bords", est une évaluation partielle du phénomène d'invariance de la règle dont l'existence est montré dans [DEV 1987], la deuxième est constituée par l'ajout régulier d'information constante. Dans [DEV 1987] une évaluation statique du nombre d'itérations nécessaires à l'obtention de l'invariance est proposée. Or le résultat obtenu est très souvent éloigné des valeurs constatées dans la pratique. Cette technique est inexploitable. C'est pourquoi, une évaluation dynamique du temps d'invariance a été mise au point.

L'état d'invariance se caractérise par la propriété suivante : à toute boucle du Gop on peut associer un facteur de croissance (nombre de nœuds par pas), soit dans le terme initial, soit dans le terme final, tel que la branche associée pousse exactement à ce rythme lors de la résolution.

Le support théorique de l'évaluation dynamique est le suivant :

- Si l'ensemble des substitutions élémentaires est identique, à un décalage d'indice des variables près, entre l'itération numéro 2^k et celle numéro 2^{k+1}, l'état d'invariance est atteint à partir de l'itération numéro 2^k.

Ce qui entraîne :

- Une boucle du Gop de pondération positive non dépliée dans le terme initial entre l'itération numéro 2^k et celle numéro 2^{k+1}, ne se dépliera plus.

- Une boucle du Gop de pondération négative non dépliée dans le terme final entre l'itération numéro 2^k et celle numéro 2^{k+1}, ne se dépliera plus.

- Les boucles non concernées par les précédents critères se dépliera régulièrement ("longueur de boucle" nœuds tout les "pondération de boucle" pas).

La preuve s'effectue sur les systèmes fortement réduits d'équations (SFR) associés à chaque pas de résolution du programme Prolog. Les propriétés sont les suivantes :

On note Σ_n le SFR au bout de n itérations. (le SFR contient un ensemble d'équations ayant en partie gauche une variable indicée, et en partie droite un terme non réduit à une variable).

Définition 1 : *Un système d'équations est un ensemble de couples de termes de $\Sigma(F \cup V)$ de la forme ($t = t'$). (t et t' sont des arbres).*

Une **solution** d'un tel système est une substitution σ telle que pour toute équation $(t = t')$ du système, $t.\sigma$ et $t'.\sigma$ soient clos et égaux.

Deux systèmes d'équations sont **équivalents** si et seulement si ils admettent le même ensemble de solutions.

Définition 2 : [COL 84] *Un système d'équations est dit sans fin si toute variable apparaissant comme terme droit d'une équation apparait aussi comme terme gauche d'une équation.*

Définition 3 : [COL 84] *Un système d'équation est dit **réduit**, si les termes gauches d'équations sont des variables distinctes, et s'il ne contient pas de sous-système sans fin.*

Soit sr un système réduit d'équations, on appelle **dom(sr)** l'ensemble des variables apparaissant à gauche d'une équation.

Notations : Dans la suite t, t' ... désignerons des termes non vide et non réduits à une variable de $\Sigma(F \cup V)$, U, V .. désignerons des variables, $t_1, ..., t_n, t'_1, ..., t'_n$ des termes quelconques.

L'algorithme de réduction est , répéter récursivement :

Etape 1 :

$f(t_1, ..., t_n) = f(t'_1, ..., t'_n)$	remplacé par $t_1 = t'_1, ..., t_n = t'_n$
$f(t_1, ..., t_n) = g(t'_1, ..., t'_n)$	echec
$U = U$	détruire l'équation
$t = U$	remplacer par $U = t$
$(U = V)$ $(U \neq V)$	remplacer U par V dans toutes les autres équations
$U = t, U = t'$	remplacer $(U = t')$ par $(t = t')$

Etape 2 : le système ne contient pas de sous-systèmes sans fin.

Résultat 1 : [COL 84] L'algorithme de réduction appliqué à un système d'équation s'arrête, si et seulement si le système admet une solution. L'algorithme retourne alors un système réduit équivalent à celui d'origine.

Définition 4 : [DEV 88] *Un système réduit d'équations est dit fortement réduit si et seulement si :*

$\forall (U = t)$ et $(V = t')$ du système, si t' n'appartient pas à Var alors t' n'est pas un sous-terme de t.

Algorithme de réduction forte , répéter récursivement :

(U = t) , (V = t') t' sous-terme de t remplacer t' par V dans t.

Théorème 1 : [DEV 88] *Deux systèmes fortement réduits sont équivalent si et seulement si ils sont égaux au nom des variables près.*

L'unicité est le caractère le plus important des systèmes fortement réduits d'équations.

Définition 5 : *Deux systèmes s_1 et s_2 d'équations sont dits orthogonaux si et seulement si il existe deux systèmes réduits sr_1 et sr_2 (sr_1 équivalent à s_1, sr_2 équivalent à s_2) tels que l'union de sr_1 et de sr_2 soit un système réduit.*

C'est à dire qu'aucune nouvelle information n'est issue de l'union des deux systèmes. (aucune contrainte nouvelle sur une variable n'apparait qui n'apparaissait déjà dans l'un des deux systèmes).

Théorème 2 : *Deux systèmes s_1 et s_2 sont orthogonaux si et seulement si il existe deux systèmes fortement réduits srs_1 et srs_2 (srs_1 équivalent à s_1, srs_2 équivalent à s_2) tels que $s_1 \cup s_2$ soit équivalent à $srs_1 \oplus srs_2$:*

$$srs_1 \oplus srs_2 = srs_1 \cup \{ (U = t) \in srs_2 \ / \ \text{il n'existe pas } (U = t') \in srs_1 \}$$

4 Calcul dynamique du temps d'invariance

Dans la suite, nous considèrerons toujours le système fortement réduit (il existe toujours) équivalent au système d'équations calculé par Prolog après n pas récursifs sur une même règle. Notons

Σ_n possède les propriétés suivantes :

- $\Sigma_{n+1} = \Sigma_n \cup \Delta(\Sigma_1,n)$ par application de la règle avec des indices de variables de n+1

de manière plus générale :

- $\Sigma_{n+k} = \Sigma_n \cup \Delta(\Sigma_k,n)$ pour tout k,

- $\Sigma_n = \Sigma_{n-k} \cup \Delta(\Sigma_{n-k'},k')$ pour tout k k' tels que $k+k' > n$ et $k<n$ et $k'<n$. Il y a redondance de contraintes.

Définition : *L'état d'invariance d'une règle récursive est atteinte après n itérations si et seulement si pour tout* $k \geq 0$:

$$\Sigma_{n+k+1} = \Sigma_{n+k} \oplus \Delta(\Sigma_{n+k},1)$$

Un système se déduit du précédent par un simple union d'équations. C'est à dire qu'aucune contrainte nouvelle n'apparait lors de la dernière unification, sauf si elle est déjà apparue précédemment à l'indice inférieur.

Notation :

$\forall\ v \in V\ \ Ind_n(v) = \{ i / \exists t,\ (v_i = t) \in \Sigma_n \} = \{ i / v_i \in Dom(\Sigma_n) \}$ ensemble des indices de v pour lesquels une équation existe dans Σ_n.

$\forall\ v \in V\ \ I_n(v) = Inf\ (Ind_n(v))$ le plus petit indice de x pour lequel il existe une équation dans Σ_n.

$\forall\ v \in V\ \ S_n(v) = Sup\ (Ind_n(v))$ le plus petit indice de x pour lequel il existe une équation dans Σ_n.

Théorème d'invariance :

L'état d'invariance est atteint au bout de m pas au plus, si et seulement si pour toute variable v les trois conditions suivantes sont vérifiées :

i) $Ind_m(v)$ *est un intervalle.*

ii) $I_m(v) = I_{m+1}(v)$

iii) $m - S_m(v) = m + 1 - S_{m+1}(v)$

Ce théorème nous permet donc de tester dynamiquement si l'état d'invariance de la règle est atteint. La démonstration s'effectue en deux étapes, la première consiste à prouver que l'existence de l'état d'invariance implique l'existence d'un nombre d'itération m tel que le théorème soit vrai, la deuxième consiste à montrer que lorsque le théorème est vérifié pour l'étape m alors l'état d'invariance est atteint.

Démonstration :

Premièrement ($\Rightarrow$): supposons l'état d'invariance atteint à l'étape n.

$\forall k \geq 0\ \Sigma_{n+k+1} = \Sigma_{n+k} \oplus \Delta(\Sigma_{n+k},1) \Rightarrow \forall m > n$: ii , iii et $\exists m \geq n$ tq i.

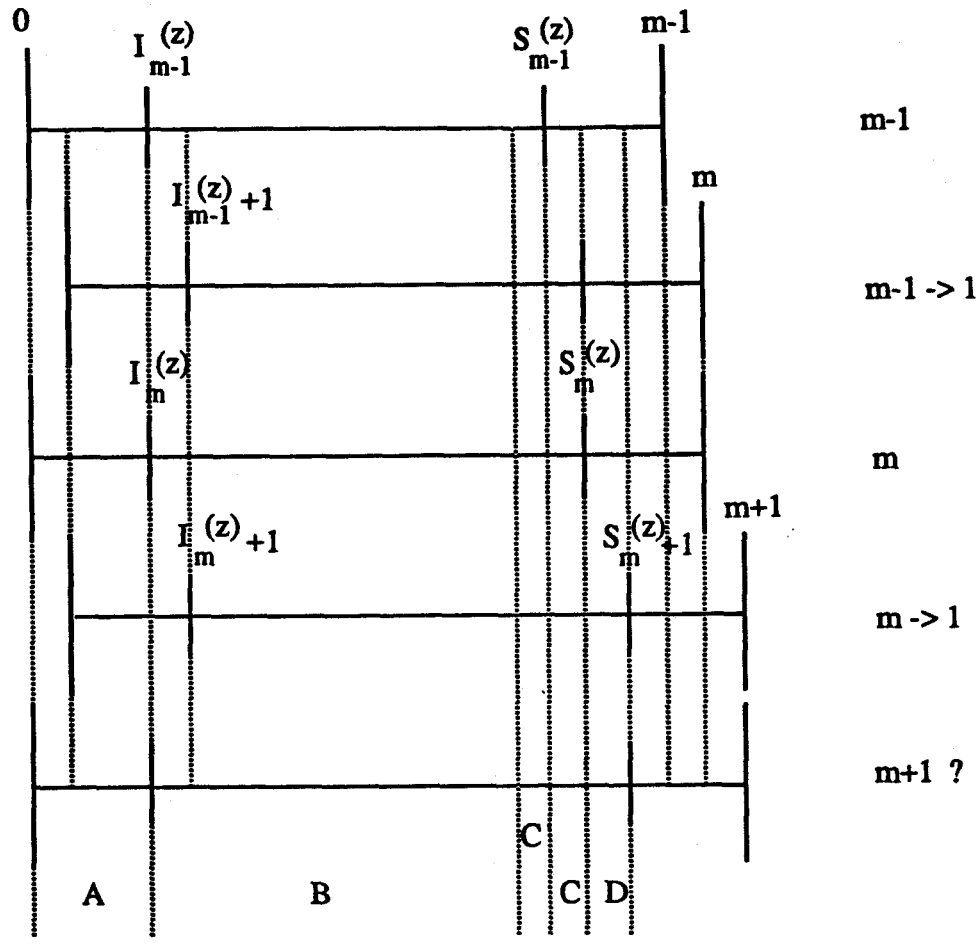
En effet : par définition, l'opération $\oplus$ peut se faire variable par variable de manière indépendante. Donc, le plus petit indice de v dans $Dom(\Sigma_m)$ restera inchangé. Il en est de même, de façon symétrique, pour le plus grand. De plus, si $Ind_{n+k}(v)$ n'est pas un intervalle, (il existe un "trou" de taille 1 dans la succession d'indice) à chaque application de l'opération $\oplus$, la taille 1 du plus grand trou diminue d'au moins 1, donc il existe un m plus grand que n tel que $Ind_m(v)$ soit un intervalle pour toutes les variables v.

Deuxièmement ($\Leftarrow$): supposons les conditions du théorème vrai pour m. prouvons le pour m+1.

$\Sigma_{m+1} = \Sigma_m \cup \Delta(\Sigma_m,1)$.

Si le théorème est faux pour m+1, alors il existe un entier i et une variable x tels que $x_i \in Dom(\Sigma_{m+1})$ et $x_i \notin Dom(\Sigma_m) \cup Dom(\Delta(\Sigma_m,1))$

Le schéma suivant représente les systèmes d'équations pour une variable z : les segments de 0 à $I_k(z)$ et de $S_k(z)$ à k (k égal à m-1, m et m+1) représente les indices pour lesquels la variable est libre, entre les deux ceux pour lesquels la variable est dans le domaine de Σ_k.



La question est donc de savoir si le schéma ci-dessus est correct pour chacune des variables de la règle (pour z quelconque de V):

- En zone A et B, le calcul effectué pour l'étape m+1 est strictement le même qu'à l'étape précédente. Donc aucune nouvelle équation ne peut être générée. Le problème reste donc au niveau des zones C et D. En fait les systèmes d'équations solution de l'unification des couples d'équations des zones C et D à l'étape n a pour solution un système S égal au système S' solution de l'unification des systèmes des zones C' et C à l'étape m-1 pour lesquels l'ensemble des indices gauches et droits ont été augmentés de 1. Donc aucune nouvelle équation n'appartenant pas à $\Sigma_m \oplus \Delta(\Sigma_m,1)$ n'a pu être introduite. On peut remarquer que toutes les équations d'une variable donnée sont égales à un décalage d'indice près.

5 Algorithme de simulation

La pile des substitutions de notre interpréteur est, comme pour le micro-Prolog de Van-Canneghem, composée de trois types d'information :

- LIBRE : pour une variable non substituée.

- REFERENCE : pour une variable ayant une contrainte d'égalité avec une autre variable, la variable concernée peut être soit libre si la variable de référence est libre, soit liée si la variable de référence est liée.

- LIEE : pour une variable substituée (associés à un terme non vide) non référencée.

Le seul état que l'on peut changer au court d'une unification est l'état LIBRE.

Dans l'algorithme qui suit, nous utiliserons une fonction ETAT retournant l'état, LIBRE ou LIEE d'une variable (indépendamment du fait qu'elle soit référencée ou non).

Appliquer (r: règle, ind:entier) est une procédure qui effectue l'application de la règle r en renommant au préalable les variables en leur affectant un indice ind.

Analyse (v : variable, k : entier) : etat_var est une fonction qui, étant donné un symbole de variable v, effectue le calcul de $I_k(v)$, $k\text{-}S_k(v)$ et vérifie que $Ind_k(v)$ est un intervalle.

VAR est l'ensemble des symboles de variables.

```
Simulation (r : règle) : entier ;  % r est une règle RPB, la fonction retour le nombre
                                     d'itération calculé pour atteindre l'invariance %.
Debut
        initialiser la pile des substitutions.
        pour tous les symboles de variable v de VAR
                                initialiser table_d_etat(v).
        fin_pour
        but_courant <- tete (r)
        nb_pas <- 0
        fini <- faux
        tant que non fini faire
              fini <- vrai
              nb_pas <- nb_pas +1
```

```
            appliquer (r,nb_pas)
            pour tous les symboles de variables v de VAR
                si  Analyse (v,nb_pas) ≠ table_d_etat(v) alors
                        table_d_etat(v) <- Analyse(v,nb_pas)
                        fini <- faux
                fin_si
            fin_pour
        fin_tant_que
        resultat : nb_pas - 1
Fin
```

Pour chaque règle récursive d'un programme Prolog, on obtient :

- Le nombre d'itérations nécessaires pour obtenir un comportement stable,

- Lors du comportement stable, les coefficient de croissance ou de décroissance de chaque branche des arguments des prédicats.

Ces différentes constantes sont utile à détecter de manière semi-automatique, l'arrêt de programmes.

6 Quelques exemples

Les exemples triviaux tels que la règle associative ou commutative nous donne bien, de manière tout à fait automatique pour la première un temps d'invariance égal à 2 (le minimum détectable) et pour la seconde une période de 2 à partir de 2 pas.

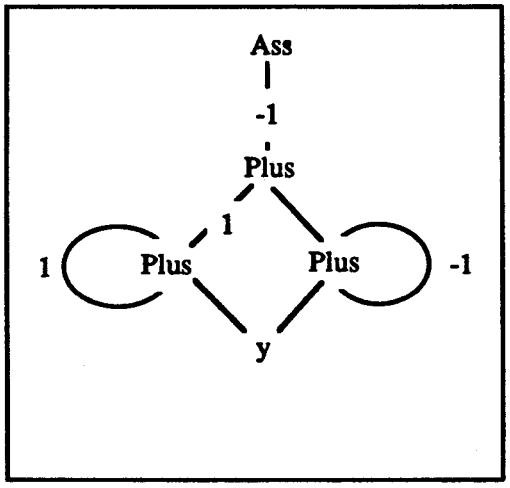
Exemple 1 : associativité :

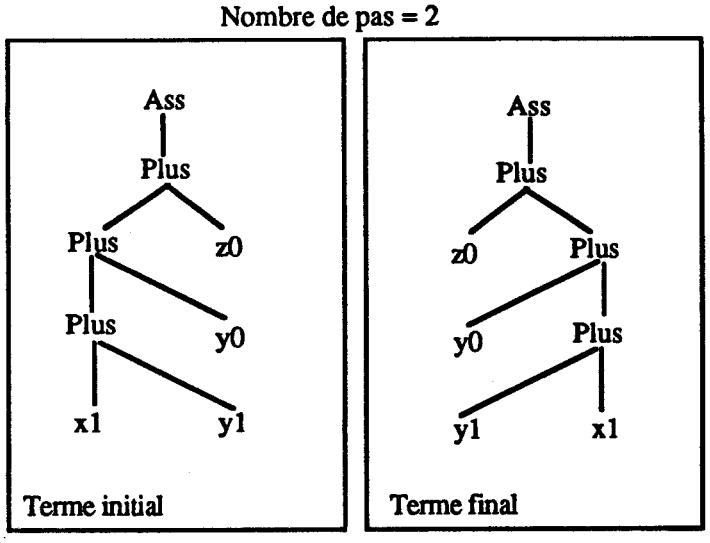
Nom du programme : Assoc

Assoc(Plus(X,Plus(Y,Plus(Z,Plus(T,Plus(V,Plus(W,0)))))) ->; le fait clos de hauteur 6

Assoc(Plus(Plus(X,Y),Z)) -> Assoc(Plus(X,Plus(Y,Z)));

Calcul du Gop :

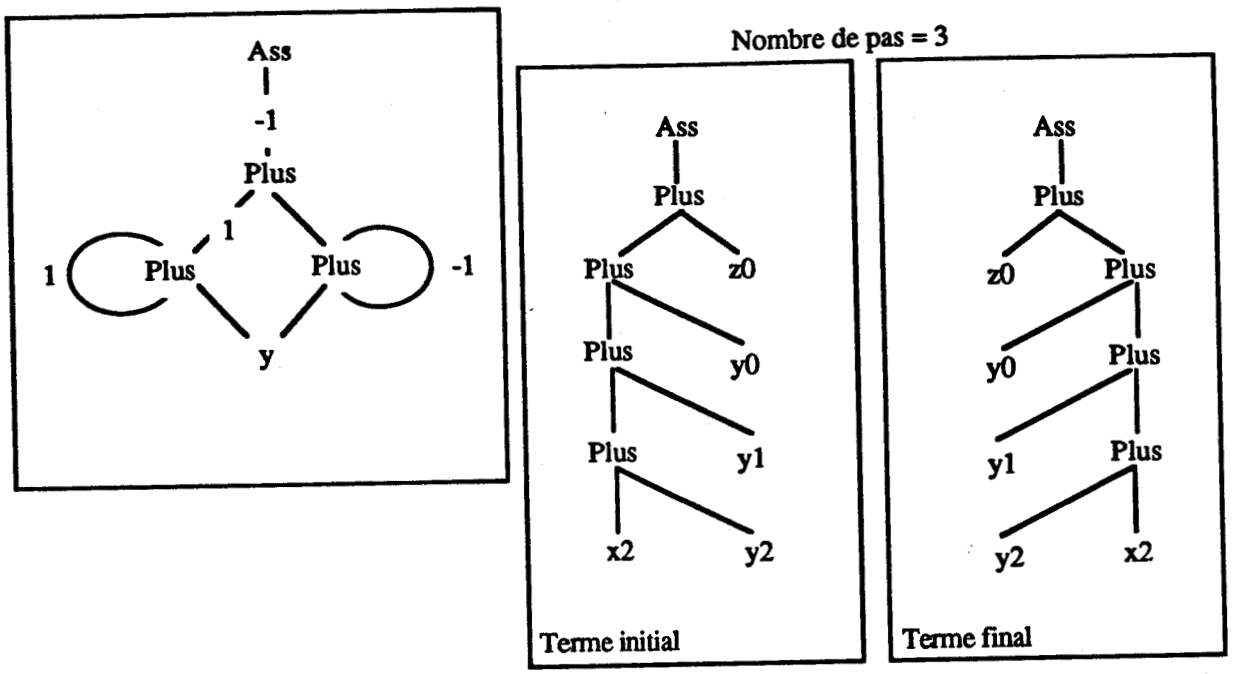




- Ce Gop vérifie la restriction sur les pondérations de boucles (pas d'arrêt par un occurrence de variable systématique)

- Nombre de boucles : 2 de couples (pondérations,longueurs) respectives : (+1,1) et (-1,1)

- pas de période.



- Nombre de pas avant il'invariance détectée : 2

- Nombre d'applications maximum au regard du fait : 6.

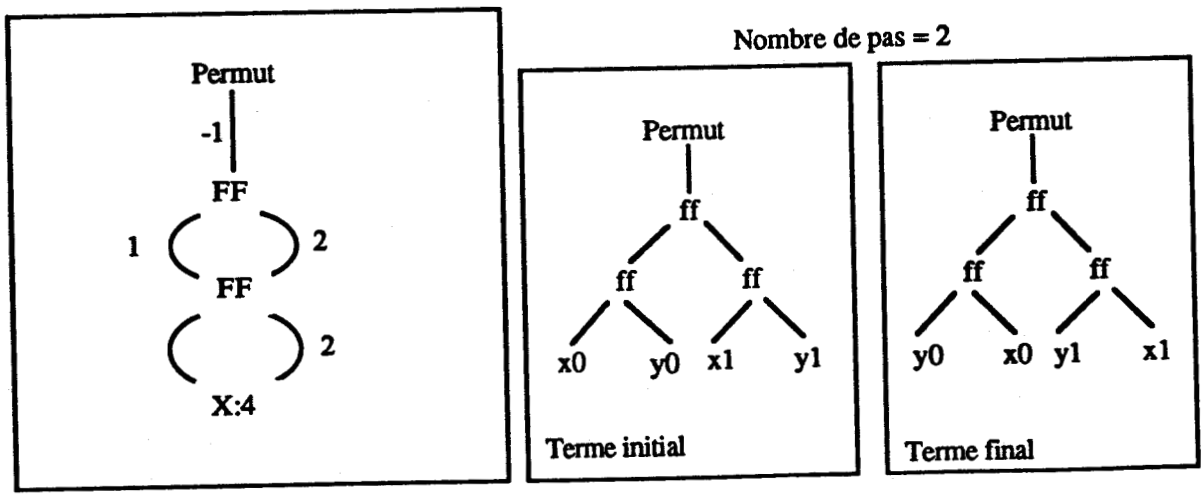
Pour bien illustrer l'intérêt de ces calcul, nous allons prendre des exemples moins évidents.

Exemple 2 : Permutation

Permut (X) -> ;

Permut (FF(FF(X,Y),Z)) -> Permut (FF(Z,FF(Y,X))) ;

Calcul du Gop :



- Ce Gop vérifie la restriction sur les pondérations de boucles (pas d'arrêt par un occurrence de variable systématique)

- Nombre de boucles : 0

- période = 4.

- Nombre de pas avant il'invariance détectée : 2 + 4 (période) = 6

- Nombre d'applications maximum : 6.

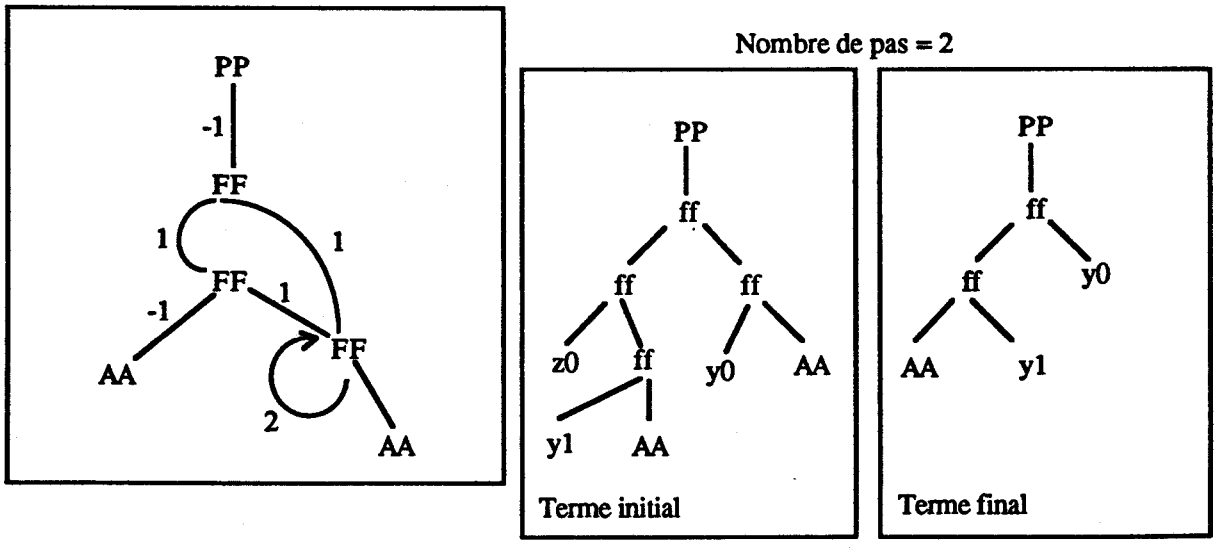
Exemple 3 :

Nom du programme : PP

PP(FF(FF(AA,AA),FF(AA,AA))) -> ;

PP(FF(FF(Z,X),FF(Y,AA))) -> PP(FF(FF(AA,Y),X)) ;

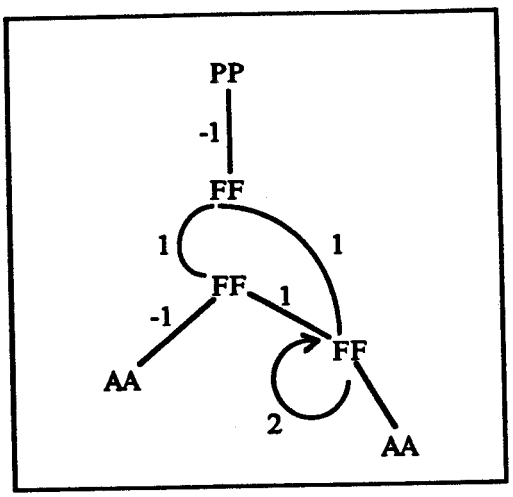
Calcul du Gop :



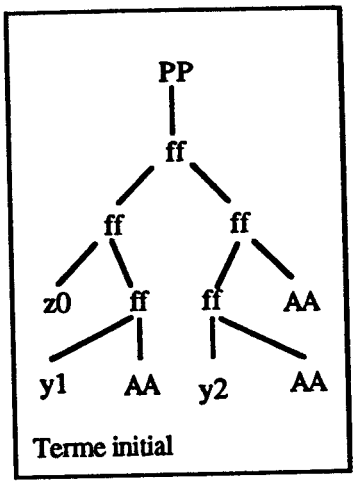
- Ce Gop vérifie la restriction sur les pondérations de boucles (pas d'arrêt par un occurrence de variable systématique)

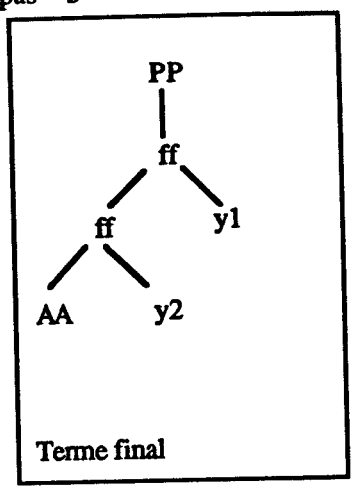
- Nombre de boucles : 2 de couples (pondérations,longueurs) respectives : (+2,1) et (+2,1)

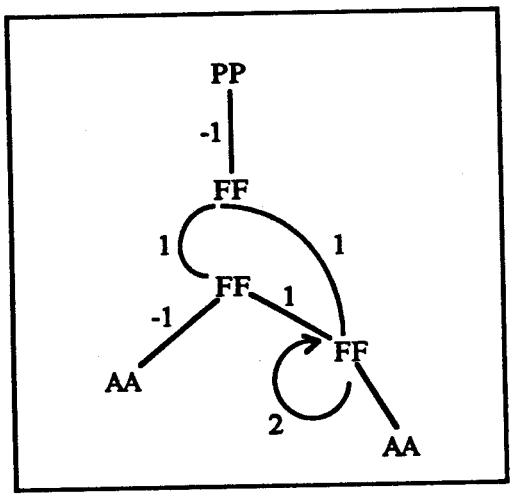
- pas de période.



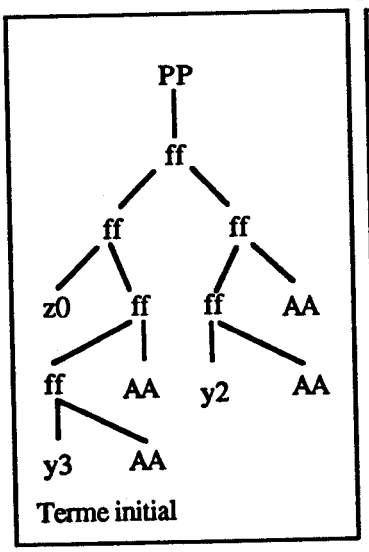
Nombre de pas = 3

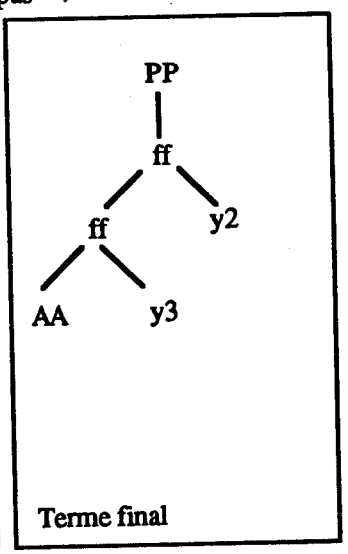


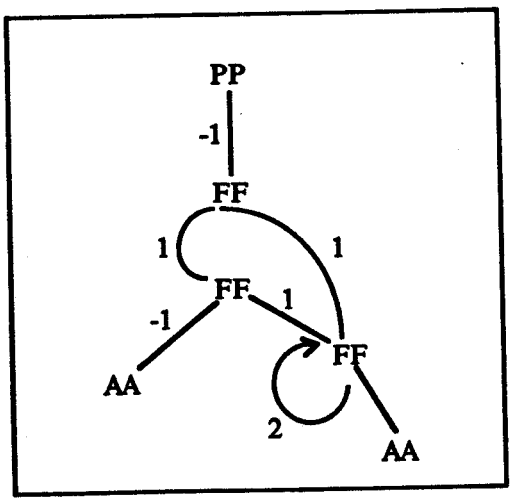




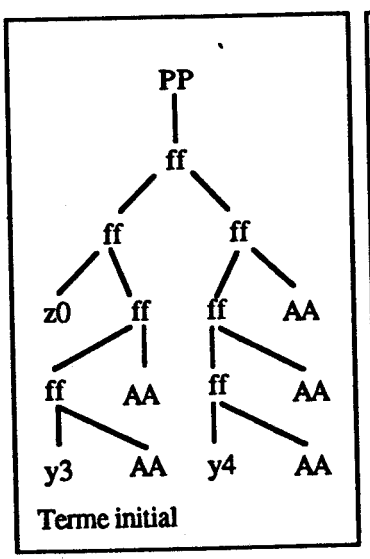
Nombre de pas =4

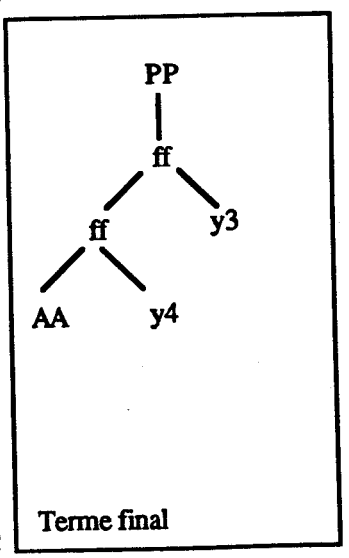






Nombre de pas = 5





- Nombre de pas avant il'invariance détectée : 4

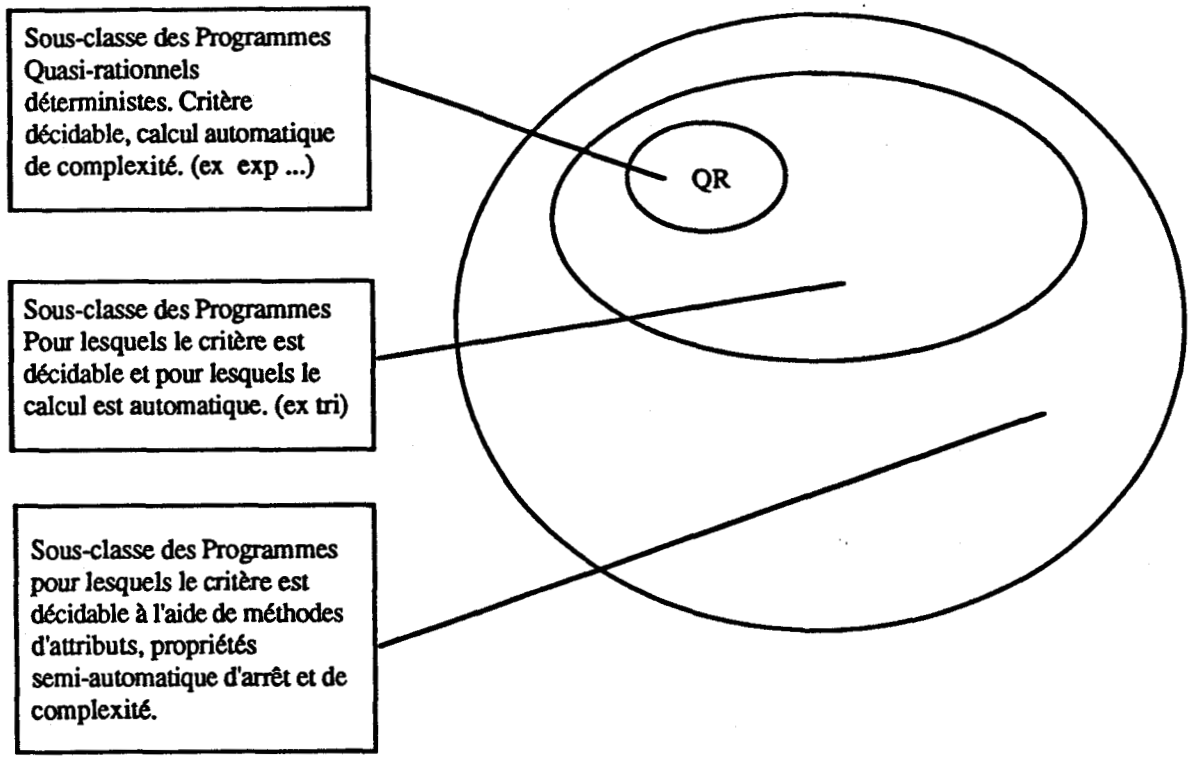
CONCLUSION

En conclusion, nous pouvons dire que l'apport de ce présent travail, est l'étude de critères de convergence et d'analyse de complexité de programmes logiques. Les critères énoncés, bien que très restricttifs, sont indécidables. La cause que l'on peut avancer, est que l'approche purement syntaxique est trop pauvre, le gap entre la structure de programmes sur laquelle on peut tout dire et tout décider, et celle pour laquelle tout est indécidable est trop mince.

Le critère le plus important que nous énonçons, qui est "hors contexte"[1] devient décidable dans une classe évidemment restreinte de programmes Prolog, celle des programmes Quasi-rationnels déterministes. Ce qui est important à noter, c'est que ce critère, lorsqu'il est vérifié, assure l'arrêt du programme. Ce qui ne veut pas dire que l'arrêt est décidable pour cette classe restreinte de programme, en effet, un programme ne vérifiant pas le critère peut s'arrêter. Mais nous avons défini une procédure entièrement automatique de vérification du critère, accompagnée, dans le cas positif, d'une évaluation automatique de la complexité des programmes concernés.

Il est clair que la finalité de ce travail n'est pas limitée à ce résultat. En effet, l'analyse de programmes Prolog peut être beaucoup plus générale lorsqu'on associe cette technique à une autre technique permettant d'inclure l'information contextuelle. C'est, par exemple, le cas des méthodes d'analyses par attributs présentées dans [DER 85]. L'union de ces deux techniques permet d'évaluer, avec ou non une aide semantique de la part de l'utilisateur, la complexité d'une large classe de programmes. Nous pouvons schématiser ceci de la manière suivante :

1 L'étude du critère se fait dans une classe donnée de prédicats, sans tenir compte du type d'informations transmis pas le contexte.



L'exemple qui suit donne une intuition de la façon dont un système d'aide et d'analyse de la complexité peut être construit.

Soit le programme classique de tri : Le Quick sort :

Tri ([] , []) → ;
Tri ([A] , [A]) → ;
Tri ([A,B/L1], L2) → Eclate ([A,B/L1], U, V)
Tri (U, U1)
Tri (V, V1)
Interclasse (U1, V1, L2) ;

Eclate ([], [], []) →;
Eclate ([A], [A], []) →;
Eclate ([A, B/L], [A/U], [B/V]) → Eclate (L, U, V) ;

Interclasse ([], L, L) → ;
Interclasse ([A/U], [], [A/U]) → ;
Interclasse ([A/U], [B/V], [A/L]) → Inf (A, B) Interclasse (U, [B/V], L) ;
Interclasse ([A/U], [B/V], [B/L]) → Infeg (B, A) Interclasse ([A/U], V, L) ;

De manière tout à fait automatique, le calcul des Gop's associés aux règles récursives Eclate et Interclasse nous donne les renseignement schématisés comme suit :

Eclate (-2 , -1 , -1) qui signifie que l'information est uniformément consommée de 2 nœuds par itération sur le premier argument, et de 1 nœud par itération sur le deuxième et le troisième. A la vue des faits, on en déduit que le premier argument est de taille double (à un nœud près) de celle des deux arguments suivants.

Interclasse ((-1) , -1) la somme des nœuds des deux premiers arguments diminue de 1 à chaque itération. A la vue des faits on obtient que le dernier argument est de taille égale à la somme des tailles des deux premiers arguments.

Ce qui donne, du point de vue de la complexité calculée automatiquement : l étant la taille de la liste en entrée de Eclate : Eclate($\downarrow, \uparrow, \uparrow$)

$E(l) = l/2$.

De même la complexité du prédicat Interclasse est : Interclasse ($\downarrow, \downarrow, \uparrow$)

$I(l/2,l/2) = l$. On notera $I(l) = l$.

Il est facile de vérifier par récurence sur la taille de la liste en entrée du prédicat Tri, Tri ($\downarrow$, $\uparrow$) que l'argument en sortie est de même taille, d'où la formule :

$$T(l) = E(l) + 2T(l/2) + I(l) = E(l) + I(l) + 2E(l/2) + 2I(l/2) + 4E(l/4) + 4I(l/4) + ...$$

$$= 3/2\, l \log_2(l)$$

Dans ce calcul, la part de l'utilisateur est limitée à l'attribution de mode, mais nous savons que cette attribution peut être automatisée. Ce programme se situe donc dans la sous-classe des programmes non Quasi-rationnels, dont l'étude est automatisable. Les règles sont simples, mais ce n'est qu'un exemple, il est bien entendu que l'analyse à l'aide des Gop's peut se faire sur des règles d'une complexité quelconque.

L'exemple qui suit est une programmation du célèbre problème des tours d'Hanoï :

Hanoï ([D] , A , B , C) → Ecrire (déplacer (D , A , C)) ;

Hanoï ([D/P] , A , B , C) → Hanoï (P, A , C , B)

Hanoï ([D] , A , B , C)

Hanoï (P , B , A , C) ;

On obtient de façon entièrement automatique, si H désigne la fonction de complexité de Hanoï :

$$H(p+1) = H(p)+1+ H(p) \quad = \sum_{i=0}^{p} 2^i$$

L'exemple qui suit nous montre le cas d'un programme arbitrairement compliqué, que l'on peut résoudre par transformation syntaxique:

$P\ (\ ff(x'_0,x_0),\ ff(x'_1,x_1),\ \dots\ ,\ ff(x'_n,x_n),\ ff(\alpha,x_{n+1})\ ) \rightarrow$

$\quad Q\ (\ ff(x'_0,x_0),\ ff(x'_1,x_1),\ ff(x'_1,x_1),\dots\ ,\ ff(x'_n,x_n),\ ff(x'_n,x_n),\ ff(\alpha,z)\ );$

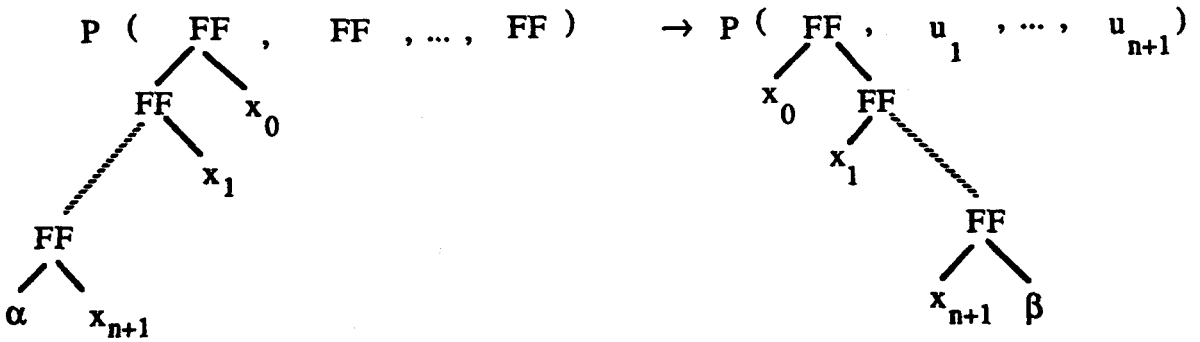
$Q\ (\ ff(t_0,x_0),\ t_0,\ ff(t_1,x_1),\ t_1,\ \dots\ ,\ ff(t_n,x_n),\ t_n) \rightarrow$

$\quad R\ (\ ff(x_0,t'_0),\ ff(x_1,t'_1),\ \dots\ ,\ ff(\ x_{n+1},\beta\ )\ )\ ;$

$R\ (\ ff(x_0,t'_0),\ t'_0,\ ff(x_1,t'_1),\ t'_1,\ \dots\ ,\ ff(\ x_n,t'_n\ )\ ,\ t'_n\ ) \rightarrow$

$\quad P\ (\ ff(x_0,t'_0)\ ,\ u_1\ ,\ u_2\ ,\ \dots\ ,\ u_{n+1}\ )\ ;$

qui a le même comportement que l'unique règle :



dont le cacul du gop nous enseigne une périodicité de n+1 avec obligation pour α et β d'être unifiable. les effets de bords calculés sont de taille 2^{n+1}.

ANNEXE

Exemples de programme quasi-rationnels d'ordre 1 et 2.

Ordre(G_π) = 1

* Prédicat vérifié pour tous les entiers strictement inférieurs à 10

INF10 ($Succ^9$(Zero)) → ; [9 < 10].

INF10 (x) → INF10(Succ(x)) ; [x < 10] si [(x+1) < 10].

* Addition de deux entiers : " ADD(x,y,x+y) "

ADD (Zero,x,x) → ; [0 + x = x].

ADD (Succ(x),y,Succ(z)) → ADD (x,y,z) ; [(x+1) + y = z+1] si [x + y = z] .

Une autre forme, plus efficace :

ADD (Zero,y,y) → ; [0 + y = y].

ADD (Succ(x),Zero,Succ(x)) → ; [x+1 + 0 = x+1].

ADD (Succ(x),Succ(y),Succ(Succ(z))) → ADD(x,y,z) ; [x+1 + y+1 = z+2]
si [x + y = z].

Ou encore, quelques exemples classiques de manipulation de listes :

* Concaténation de deux listes " CONC (l_1,l_2,l_1+l_2) "

CONC (Nil,liste,liste) → ;

CONC (e.liste1,liste2,e.liste) → CONC (liste1,liste2,liste) ;

* Prédicat d'appartenance d'un élément à une liste

APPARTIENT (elt,elt.liste) → ;

APPARTIENT(elt,e.liste) → APPARTIENT(elt,liste) ;

* Prédicat d'inversion d'une liste

REV (Nil , liste , liste) → ;

REV (a.liste , liste1 , liste2) → REV (liste , a.liste1 , liste2) ;

REVERSE (liste , liste-inv) → REV (liste , Nil , liste-inv) ;

b) Avec plusieurs faits.

* La commutativité :

FRERE (Jean,Gaston) → ; [Jean est le frère de Gaston]

FRERE (Georges,Xavier) → ; [Georges est le frère de Xavier]

FRERE (x,y) → FRERE (y,x) ; [x est le frère de y] si [y est le frère de x]

* Prédicat vérifiant que deux entiers sont differents :

DIFFERENT (Zero,Succ(y)) → ; [$0 \neq y+1$].

DIFFERENT (Succ(x),Zero) → ; [$x+1 \neq 0$].

DIFFERENT (Succ(x),Succ(y)) → DIFFERENT (x,y) ; [$(x+1) \neq (y+1)$] si [$x \neq y$].

Une autre formulation possible de même ordre est :

DIFFERENT (x,y) → INF (x,y); [x ≠ y] si [x < y]

DIFFERENT (x,y) → INF (y,x); [x ≠ y] si [y < x].

c) Coloration d'une carte, composée de 5 régions qui doivent être de couleurs différentes si elles sont voisines (cf figure 1).

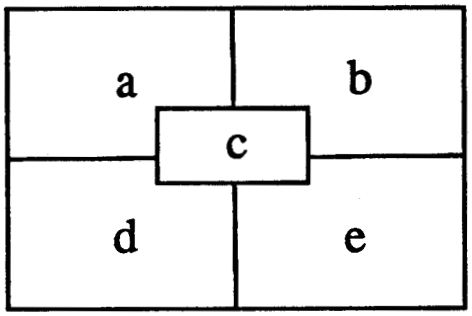


figure 1

```
COLORIER (a,b,c,d,e)  →  COULEUR(a,na)      COULEUR(b,nb)
                         COULEUR(c,nc)      COULEUR(d,nd)
                         COULEUR(e,ne)
                         DIFFERENT(na,nb)   DIFFERENT(na,nc)
                         DIFFERENT(na,nd)   DIFFERENT(nb,nc)
                         DIFFERENT(nb,ne)   DIFFERENT(nc,nd)
                         DIFFERENT(nc,ne)   DIFFERENT(nd,ne);

COULEUR(Vert,Zero)  →;

COULEUR(Bleu,Succ(Zero))  → ;

COULEUR(Rouge,Succ(Succ(Zero)))  → ;
```

d) Détaillons certains programmes de reconnaissance de Langages classiques :

. Langage de Dyck (bien parenthésé) sur X = { A,B }

```
DYCK (mot)  →  LIREMOT (mot,Zero) ;

LIREMOT (Nil,Zero)  → ;
```

LIREMOT (lettre.mot , compteur) → LIREMOT (mot , compteur')

COMPTEUR (lettre , compteur , compteur') ;

COMPTEUR (A , compteur , Succ(compteur)) → ; [compteur incrémenté]

COMPTEUR (B , Succ(compteur) , compteur) → ; [compteur décrémenté]

Dans LIREMOT , le 2ème argument désigne l'expression, toujours positive, suivante :

(nombre de A lus) - (nombre de B lus)

. Langage L1 = { $a^n b^n$ / $\forall \in N$ }.

L1 (Nil) → ;

L1 (mot) → LIRE-A (mot , Zero) ;

LIRE-A (A.mot , n) → LIRE-A (mot , Succ(n)) ;

LIRE-A (mot , n) → LIRE-B (mot , n) ;

LIRE-B (Nil , Zero) → ;

LIRE-B (B.mot , Succ(n)) → LIRE-B (mot , n) ;

Le 2ème argument de LIRE-A désigne le nombre de A déja lus

Le 2ème argument de LIRE-B désigne le nombre de B restant à lire.

LIRE-A et LIRE-B sont réflexifs , L1 est non réflexif

Aucun prédicat n'est étroitement dépendant de LIRE-A donc

Ordre(LIRE-A) = Ordre(LIRE-B)

Comme Ordre(LIRE-B) = 1 alors Ordre(L1) = 1

. Langage L2 = { $a^n b^n c^n$ / $\forall \in N$ } .

L2 (Nil) → ;

L2 (mot) → LIRE-A (mot , Zero) ;

LIRE-A (A.mot , n) → LIRE-A (mot , Succ(n)) ;

LIRE-A (mot , n) → LIRE-B (mot , n , n) ;

LIRE-B (B.mot , n_a , Succ(n)) → LIRE-B (mot , n_a , n) ;

LIRE-B (mot , n , Zero) → LIRE-C (mot , n) ;

LIRE-C (nil , Zero) → ;

LIRE-C (C.mot , Succ(n)) → LIRE-C (mot,n) ;

LIRE-A - 2ème argument : nombre de A déja lus

LIRE-B - 2ème argument : nombre total de A qui ont été lus

3ème argument : nombre de B restant à lire

LIRE-C - 2ème argument : nombre de C restant à lire.

e) Exemples de programmes polynomiaux d'ordre 1.

* Calcul du carré d'un entier :

CARRE (n , n_2) → CALCUL (n , n_2 , Zero , Zero , Zero) ;

CALCUL (n , n_2 , n_2 , Zero , n) → ;

CALCUL (n , n_2 , c_0 , c_1 , c_2) → CALCUL (n , n_2 , Succ(c_0) , c_1' , c_2')

COMPTEURS (n , c_1 , c_1' , c_2 , c_2') ;

COMPTEURS (Succ(n) , n , Zero , c_2 , Succ(c_2)) → ;

COMPTEURS (n , c_1 , Succ(c_1) , c_2 , c_2) → ;

On utilise ici trois compteurs :

. c_0 compteur général qui contiendra le carré de n

. $c_1 = c_0$ modulo n si $c_1 < n$

. $c_2 = c_0$ div n si $c_1 < n$

c_0 est le résultat ssi $c_2 = n$ et $c_1 =$ Zero

A chaque étape, on incrémente le compteur général c_0.

Si $c_1 = n-1$ alors la nouvelle valeur de c_1 (cad c_1') est Zero et la nouvelle valeur de c_2 (cad c_2') est $c_2 + 1$.

Ou alors on incrémente simplement c_1, c_2 est inchangé.

Sur le même principe, en augmentant le nombre de compteurs (p+1), on peut construire un programme "calculant" à partir d'une donnée n la puissance pième de n ($p \in N$).

* Multiplication de deux entiers :

MUL1 (x , y , z) → CALCUL (x , y , z , Zero , Zero , Zero) ;

CALCUL (x , y , z , z , Zero , x) → ;

CALCUL (x , y , z , c_0 , c_1 , c_2) → CALCUL (x , y , z , Succ(c_0) , c_1' , c_2')

COMPTEURS (x , y , c_1 , c_1' , c_2 , c_2');

COMPTEURS (x , Succ(y),y , Zero , c_2 , Succ(c_2)) → ;

COMPTEURS (x , c_1 , Succ(c_1) , c_2 , c_2) → ;

. c_0 compteur général qui contiendra x*y

. $c_1 = c_0$ modulo y si $c_1 < y$

. $c_2 = c_0$ div y si $c_1 < y$

c_0 est le résultat ssi c_2=x et c_1 = Zero

Ordre(G_π) = 2

On peut décrire aussi sur les listes :

* la fonction classique d'inversion de listes

REVERSE (Nil , Nil) → ;

REVERSE ($e.l_1$, l_2) → REVERSE (l_1 , l_3) CONC (l_3 , e.Nil , l_2) ;

* le minimum d'une liste " MINIMUM (minimum , liste d'entiers) "

MINIMUM (e , l) → APPARTIENT (e , l) MINORANT (e , l) ;

MINORANT (e , Nil) → ;

MINORANT (e , f.l) → MINORANT (e , l) INF-EGAL (e , f) ;

INF-EGAL (Zero , x) → ;

INF-EGAL (Succ(x) , Succ(y)) → INF-EGAL (x , y) ;

Le minimum e d'une liste l d'entiers est

. un entier appartenant à la liste : APPARTIENT(e , l)

. tout élément de la liste est inférieur ou égal à ce minimum : MINORANT (e , l)

On peut caractériser certaines fonctions arithmétiques :

* Calcul de la puissance $n^{ième}$ de 2 " PUISSANCE (n , 2^n) "

PUISSANCE (Zero , Succ(Zero)) → ;

PUISSANCE (Succ(n) , résultat) → PUISSANCE (n , sous-résultat)

ADD (sous-résultat , sous-résultat , résultat) ;

[$2^0 = 1$] et [$2^n = 2^{n-1} + 2^{n-1}$]

* Suite de Fibonnacci " FIBONNACCI (n , f(n)) "

FIBONNACCI (Zero , Zero) → ;

FIBONNACCI (n , f^n) → CALCUL (n , f^n , f^{n-1}) ;

CALCUL (Succ(Zero) , Succ(Zero) , Zero) → ;

CALCUL (Succ(n) , f^{n+1} , f^n) → CALCUL(n , f^n , f^{n-1})

ADD (f^{n-1} , f^n , f^{n+1}) ;

Le prédicat Calcul a pour arguments dans l'ordre : n , f(n) , f(n-1)

. pour n=1 f(n)=1 , f(n-1)=0

. f(n+1) = f(n) + f(n-1)

* Multiplication d'entiers " MUL2 (x , y , x*y) "

MUL2 (Zero , y , Zero) → ; [0*x = 0]

MUL2 (Succ(x) , y , z) → MUL2 (x , y , w) ADD (y , w , z) ;

[0*x = 0] et [(x+1)*y = z] si ([x*y = w] et [y + w = z]).

Une autre forme plus efficace :

MUL2 (Zero , y , Zero) → ; [0*y = 0]

MUL2 (Succ(x) , Zero , Zero) → ; [(x+1)*0 = 0]

MUL2 (Succ(x) , Succ(y) , z) → MUL2 (x , y , zz) ADD (x , zz , zzz)

ADD (Succ(y) , zzz , z) ;

[(x+1)*(y+1) = z] si [x*y = zz] et [x+(y+1)+zz = z]

* Calcul de Factorielle " FACTORIEL (n , n!) "

FACTORIEL (Zero , Succ(Zero)) → ;

FACTORIEL (Succ(x) , y) → FACTORIEL (x , z) MUL1 (Succ(x) , z , y) ;

[0! = 1] et [(n+1)! = y] si ([n! = z] et [(n+1)*z = y]).

* Calcul d'Exponentielle " EXP (x , y , x^y)"

EXP (x , Zero , Succ(Zero)) → ;

EXP (x , Succ(y) , z) → EXP (x , y , w) MUL1 (x , w , z) ;

[$x^0 = 1$] et [$x^{y+1} = z$] si ([$x^y = w$] et [$x*w = z$])

* La Propriété arithmétique du Théorème de FERMAT

Trouver x , y , z , n tel que x , y , z > 1 , n > 2 et $x^n+y^n = z^n$

FERMAT (x , y , z , n) → EXP (x , n , x^n) EXP (y , n , y^n) EXP (z , n , z^n)

INF (Succ(Zero) , x) INF (Succ(Zero) , y)

INF (Succ(Zero) , z) INF (Succ(Succ(Zero)) , n)

ADD (x^n , y^n , z^n) ;

BIBLIOGRAPHIE

[APT 1980] **Apt K.R. Van Emden M.H.**, *Contribution to the theory of Logic Programming*. Technical Report CS-80-12, Departement of Computer Science, University of Waterloo.

[APT 1987] **Apt K.R.**, *Introduction to Logic Programming*. Technical report TR-87-35, Departement of Computer Science, University of Texas at Austin. Septembre 87.

[AZI 1986] **Azibi N. Costa E.J. Kodratoff Y.**, *Méthode de transformation de programmes de Burstall-Darlington appliquée à la programmation logique*, Acte des jounées Programmation Logique, Tregastel pp 327-343.

[BAU 1988] **Baudinet M.**, *Proving Termination properties of PROLOG Programs : A semantic Approach*. Technical Report STAN-CS-88-1202, Department of computer science, Stanford University, California.

[BEE 1988] **Beer J.**, *The occur-check problem revisited*. J. of Logic Programming Vol 5 n°3 sept 88, pp. 243-261.

[BRO 1979] **Brough D.R.**, *Loop trappping in Logic Programs*.Dep.Rep 79/9, Dep. of Computing and control, Impérial College, London.

[BRO 1986] **Brough D.R. Hogger C.J.**, *The treatment of loops in Logic Programming*. Dep.Rep , Dep. of Computing and control, Impérial College, London.

[BRO 1987] **Brough D.R. Hogger C.J.**, *Compiling Associativity into logic programs*. Journal of Logic Programming 4, pp. 345-359.

[BUR 1977] **Burstall R.M. Darlington J.**, *A transformation system for developing recursive programs*, JACM, vol 24 n°1 Jan 77.

[CLA 1979] **Clark K.L. Mac Cabe F.G.**, *The control facilities of IC-Prolog*. Expert System in the Micro Electronic Age, Edinburgh University Press, Scotland, pp. 153-167.

[COL 1979] **Colmerauer A.**, *Prolog II : manuel de référence et modèle théorique*. G.I.A, Université d'Aix-Marseille.

[COL 1982] **Colmerauer A.**, *Prolog and infinite trees*, in Clark and Tarlund.

[COL 1983] **Colmerauer A. Kanoui H. Van Caneghem M.**, *Prolog, Bases théoriques et développements actuels* , TSI Vol. 2 n° 4, pp 43-62.

[COL 1984] **Colmerauer A.**, *Equations and inequetions on finite and infinite trees*, FGCS'84 Proceedings, pp 85-99.

[COR 1983] **Corbin J. Bidoit M.**, *Une réhabilitation de l'algorithme d'unification de Robinson*, Information Processing 83, recueil des conférences IFIP, pp 253-259.

[COU 1983] **Courcelle B.** , *Fundamental properties of infinite tree*, Theor. Comp. Sci., 17, pp 95-169.

[COU 1986] **Courcelle B.**, *Equivalence and transformations of regular systems. Applications to recursive program schemes and grammars*. Theor. comp. Sci., vol 42, pp. 1-122.

[COV 1985] **Covington M.A.**, *A further note on looping in Prolog*. SIGPLAN notices, pp. 28-31.

[COV 1985] **Covington M.A.**, *Eliminating unwanted loops in Prolog*. SIGPLAN notices, pp.20-26.

[DAU 1987] **Dauchet M. Devienne P. Lebègue P.**, *Décidabilité de la terminaison d'une règle de réécriture en tête*, Journées AFCET-GROPLAN, Rouen France.

[DAU 1987] **Dauchet M.**, *Termination of rewriting is undecidable in the one rule case*, Internal Report IT110, LIFL Lille.

[DEB 1986] **Debray S.K.**, *Mode Inference and Abstract Interpretation in Logic Programs*. Technical report 86-05 February 1986, Departement of Computer Science, State University of N.Y., at Stony Brook.

[DEB 1988] **Debray S.K. Warren D.S.**, *Automatic mode inferencing for Logic Programs*. Journal of Logic Programming, 1988 n°5, pp. 207-229.

[DEL 1986] **Delahaye J.P.**, *Outils logiques pour l'intelligence artificielle*. Editions Eyrolles.

[DEL 1988] **Delahaye J.P.**, *Cours de Prolog avec Turbo Prolog, éléments de bases*. Editions Eyrolles.

[DER 1985] **Deransart P. Maluszynski J.**, *Relating logic programs and attribute grammars*. The journal of logic programming 1985,vol 2,pp 119-155.

[DERS 1985] **Dershowitz N.**, *Termination*, Proc. 1st Conf. Rewriting techniques and applications, Lect. Notes in Comp. Sci. vol 202, pp. 180-224, Springer Verlag, Dijon Mai 85.

[DEV 1986] **Devienne P. Lebègue P.**, *Weighted Graph, A tool for logic programming*, CAAP'86, Lecture notes in Computer Science 214, Springer verlag, pp 100,111.

[DEV 1987] **Devienne P.**, *Les Graphes Orientés Pondérés, un outils pour la programmation logique, Thèse de doctorat*, Université de Lille I.

[DEV 1988] **Devienne P.**, *Weighted Graphs, a tool for expressing the behaviour of recursive rules in Logic Programming*. FGCS conferences NOV. 1988.

[EDE 1985] **Eder E.**, *Properties of substitutionss and unifications*, Journal of symb. Comp., 1, pp 31-46.

[ESE 1988] **Sestoft P. Sondergaard H.**, *A bibliography on partial evaluation*. Sigplan Notices V23 n°2 Fev 88, pp. 19-27.

[EUD 1986] **Eudes J.**, *Résolution et dépendance entre données dans un Programme PROLOG*. Programmation en logique, Actes du séminaire, Trégastel 86.

[FAG 1983] **Fages F.**, *Notes sur l'unification des termes du premier ordre finis ou infinis*, Rapport interne INRIA-LITP, France.

[FER 1985] **Ferrand G.**, *Error diagnosis in Logic Programming, an adaptation of E.Y. Shapiro's method.* Rappot technique n° 375, Mars 1985, INRIA Rocquencourt.

[HIL 1988] **Hill P.M. Lloyd J.W.**, *Analysis of Meta-Programs.* Technical Report CS-88-08, Departement of computer scince, University of Bristol.

[HUE 1976] **Huet G.**, *Résolution d'équations dans les langages d'ordre 1,2,..., Ω,* Thèse de doctorat d'état, Université de Paris VI.

[JOU 1986] **Jouannaud J.P. Lescanne P.**, *La réécriture.* Synthèse dans TSI Vol 5 n°6 p. 433-452.

[KAT 1986] **Katsuhiko Nakamura**, *Control of Logic Program execution based on the functional relations.* Third International Conference on Logic Programming, Lecture notes in Comptuter Science 225, pp 505-512].

[KOW 1976] **Van Emden M.H. Kowalski R.A.**, *The semantics of Predicate Logic as a Programming Language,* Journal of the ACM 23, 4 (1976), pp 733-742.

[LAS 1987] **Lassez J.L. Maher M.J. Marriot K.**, *Unification Revisited,* Workshop on Logic and Data Bases, J. Minker.

[LIP 1977] **Lipton R. Snyder L.**, *On the halting of tree replacement systems,* Conference on Theoretical Computer Science, Waterloo Canada, pp 43-46.

[LLO 1984] **Lloyd J.W.** , *Foundations of Logic Programming,* 1984 1987, Springer-verlag, Berlin.

[MAH 1986] **Maher M.J.** , *Equivalences of Logic Programs,* Third International Conference on Logic Programming, Lecture Notes in Computer Science 225 Springer-Verlag, pp 410-424.

[MEL 1981] **Mellish C.S.** , *Some Global Optimisation for a Prolog Compiler,* Journal of Logic Programming, Vol 2, No 1.

[MEL 1981] **Mellish C.S.** , *The Automatic Generation of Mode Declaration for Prolog Programs,* Research Report 163, Dept. of Artificial Intelligence Edindurgh.

[NAI 1982] **Naish L.** , *An Introduction to MU-PROLOG,* Technical Report 82/2, Dept. of Computer Science, Univ. Melbourne.

[NAI 1985] **Naish L.** , *Automating control for logic programming.* The Jounal of Logic Programming, Vol 2, No 3, pp 167-183.

[PLA 1984] **Plaisted D.A.**, *The occurs-check problem in PROLOG.* New generation Computing, 2(4). pp.309-322.

[POO 1985] **Poole D. Goebel R.**, *On elimination loops in PROLOG.* Sigplan Notices, V20 (8), pp.38-40.

[POT 1986] **Potter J. Vasak T.**, *Chacarterisation of terminating Logic Programs.* Third International Symposium on Logic Programming, 1986, pp. 140-147.

[SHA 1983] **Shapiro E.Y.**, *Algorithmic Program Debugging.* M.I.T. Press, Cambridge.

[SMI 1986] **Smith D.E. Genesereth M.R. Ginsberg M.L.** , ***Controlling Recursive Inference.*** Artificial Intelligence 30. pp. 343-389.

[VIE 1987] **Vieille L.,** ***Data-complete proof procedure based on SLD resolution.*** 4th conference on Logic Programming, Melbourne.

Contribution à l'étude de la programmation logique par les graphes orientés pondérés.

L'étude de la complexité de programmes logiques ne peut pas se fonder sur des critères uniquement syntaxiques. En effet, nous montrons que toute structure de programmes Prolog plus complexe que celle du "tant que", introduite par P. Devienne et moi-même : P(a) -> ;
P(b) -> P(c) ;
(pour laquelle nous pouvons tout dire), contient toute la complexité d'un langage de programmation. Ce qui montre une fois de plus, que l'approche purement syntaxique est insuffisante. C'est donc sur des critères sémantiques liés à la notion d'attributs, de programmes dirigés par les données, que nous nous basons pour énoncer une hiérarchie syntaxique de programmes pour lesquels nous pouvons décider de l'arrêt et de la complexité en temps d'exécution ainsi qu'en espace. Des méthodes de calculs automatiques pour une classe restreinte de programmes et semi-automatiques pour une classe beaucoup plus large, sont énoncées.

MOTS CLES :

Système de réécriture
Narrowing
Graphe
Complexité
Récursivité
Terminaison
Graphe orienté pondéré
Programmation Logique