67805

USTL FLANDRES ARTOIS

LIFL U.A. 369 du C.N.R.S.

CNRS

50376
1991
126

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

Numéro d'ordre: 738

Année 1991

THESE

présentée à

l'Université des Sciences et Techniques de LILLE FLANDRES-ARTOIS

Pour obtenir le titre de

Docteur en INFORMATIQUE

par:

Stéphane JANOT

Règles d'Evaluation Equitables en Programmation Logique

soutenue le 17 Juin 1991 devant la commission d'Examen

Membres du Jury:

Michel MERIAUX	Président
Yves BEKKERS	Rapporteur
Pierre DERANSART	Rapporteur
Jean-Paul DELAHAYE	Directeur de thèse
Philippe DEVIENNE	Examinateur
Pierre LECOUFFE	Examinateur

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

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

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

PROFESSEUR EMERITE

M. A. LEBRUN

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

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

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

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

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

PROFESSEURS - 1ère CLASSE

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

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

PROFESSEURS - 2ème CLASSE

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

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

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

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

Table des Matières

Introduction 7

Chapitre I : Résolution SLD, fondements et aspects théoriques des stratégies équitables 15

I.1 Introduction 17

I.2 Résolution SLD 17

I.2.1 Introduction 17

I.2.2 Définitions et notations 18

I.2.2.1 Programmes définis 18

I.2.2.2 Dérivation SLD 18

I.2.3 Sémantique déclarative des programmes définis 19

I.2.3.1 Base de Herbrand, modèles de Herbrand d'un programme 19

I.2.3.2 Réponses correctes et plus petit modèle de Herbrand 19

I.2.4 Justesse de la résolution SLD 20

I.2.4.1 Ensemble de succès d'un programme 20

I.2.4.2 Justesse de la résolution SLD 20

I.2.5 Complétude de la résolution SLD 21

I.2.6 Indépendance de la règle d'évaluation 21

I.2.6.1 Règle d'évaluation 21

I.2.6.2 Indépendance de la règle d'évaluation 22

I.2.7 Arbres de dérivation SLD 22

I.2.8 Echecs finis d'un programme 25

I.3 Résolution SLD équitable 27

I.3.1 Introduction 27

I.3.2 Dérivations équitables 27
I.3.2.1 Définitions 27
I.3.2.2 Propriétés des dérivations équitables 28
I.3.2.3 Propriétés des arbres de dérivation équitables 30
I.4 Interpréteurs PROLOG 31
I.4.1 Introduction 31
I.4.2 Interpréteurs Prolog 31
I.4.2.1 Définition d'un interpréteur 31
I.4.2.2 Interpréteurs standards 31
I.4.2.3 Interpréteurs équitables 33
I.4.3 Ensembles d'échecs et de succès 34
I.4.3.1 Echecs finis 34
I.4.3.2 Ensembles de succès 35
I.4.3.2.1 Ensemble de succès d'un programme 35
I.4.3.2.2 Ensemble de succès pour un interpréteur "en profondeur d'abord" .. 35
I.4.3.2.3 Ensemble de succès finis d'un programme 36
I.4.3.3 Boucles liées à la stratégie de dérivation standard 38
I.5 Résolution SLD avec négation par l'échec 41
I.5.1 Introduction 41
I.5.2 Définitions 41
I.5.2.1 Programmes normaux 41
I.5.2.2 Complétion d'un programme 42
I.5.3 Justesse de la Résolution SLDNF 43
I.5.3.1 Dérivations SLDNF, arbres de dérivation 43
I.5.3.2 Justesse de la résolution SLDNF 48
I.5.4 Complétude de la SLDNF 48
I.5.4.1 Résultats de complétude 48
I.5.4.2 Règles maximales récursives pour la SLDNF 50
Chapitre II : Définition de stratégies de dérivation équitables 53
II.1 Introduction 55
II.2 Stratégie de dérivation en file 55
II.3 Stratégie de dérivation "Pile-Indicée" 58
II.3.1 Introduction 58
II.3.2 Définitions 58
II.3.3 La règle d'évaluation pile-indicée 58
II.3.4 Equité de la règle d'évaluation pile-indicée 59
II.3.5 Exemples 60
II.3.6 Conclusion 62

II.4 Stratégie de dérivation "Arbre de Buts Indicés" 63
II.4.1 Introduction 63
II.4.2 Définitions 63
II.4.3 Mise à jour d'un arbre de buts indicés 64
II.4.4 Programmes partiellement indicés 64
II.4.5 La règle d'évaluation "Arbres de Buts Indicés" 65
II.4.6 Equité de la règle d'évaluation "Arbre de Buts Indicés" 66
II.4.7 Exemples 67
II.4.8 Conclusions 71

Chapitre III : Réalisation d'interpréteurs équitables 73

III.1 Introduction 75

III.2 Réalisations en Prolog 75
III.2.1 Méta-interpréteurs Prolog 75
III.2.2 Implémentation de la stratégie Pile-Indicée. 77
III.2.3 Implémentation de la stratégie Arbre de buts indicés 77
III.2.4 Interpréteurs avec "cut" 79

III.3 Réalisation d'interpréteurs équitables avec MALI 83
III.3.1 Introduction 83
III.3.2 La machine MALI 83
III.3.2.1 Les termes MALI 83
III.3.2.2 Gestion de l'indéterminisme 84
III.3.2.3 Le récupérateur de mémoire de MALI 84
III.3.3 PrologII/MALI 84
III.3.4 Implémentations 84
III.3.4.1 Modifications apportées à PrologII/MALI 84
III.3.4.2 Implémentation de la stratégie Pile-Indicée 85
III.3.4.3 Implémentation de la stratégie Arbres de buts indicés 86
III.3.5 Evaluation des interpréteurs équitables PrologII/PI et PrologII/ABI 86
III.3.5.1 PrologII/PI et PrologII/ABI 86
III.3.5.2 Performances de l'interpréteur PrologII/PI 87
III.3.5.3 Performances de l'interpréteur PrologII/ABI 89
III.3.6 Conclusions 90

III.4 Stratégies non standards et prédicats prédéfinis 91
III.4.1 Traitements des prédicats système avec les stratégies équitables 91
III.4.2 Utilisation du cut avec les interpréteurs équitables 92

III.5 Implémentation de la négation 94

III.5.1 Négation en Prolog standard 94
III.5.2 Négation avec la stratégie ABI 95
III.5.2.1 Négation équitable 95
III.5.2.2 Négation sure 96
III.5.2.3 Négation sure et équitable 97

Chapitre IV : Techniques de contrôle et règle d'évaluation 101

IV.1 Introduction 103

IV.2 Contrôle dynamique en Prolog 103
IV.2.1 Introduction 103
IV.2.2 Primitives de contrôle 103
IV.2.2.1 Les annotations de variables 104
IV.2.2.2 Geler 104
IV.2.2.3 Les déclarations WAIT 105
IV.2.3 MU-PROLOG 106
IV.2.3.1 Génération des déclarations WAIT 106
IV.2.3.2 Réordonnancement des sous-buts 107
IV.2.4 Conclusions 107

IV.3 Complexité de l'arbre SLD et règle d'évaluation 109
IV.3.1 Introduction 109
IV.3.2 Exemple 109
IV.3.3 Une bonne règle d'évaluation 110
IV.3.3.1 Minimiser l'arbre de dérivation SLD 110
IV.3.3.2 Les différents cas possibles 111

Chapitre V : Utilisation des interpréteurs équitables 115

V.1 Introduction 117

V.2 Utilisation de la stratégie Pile-Indicée 117
V.2.1 Contrôle avec la stratégie Pile-indicée 117
V.2.2 Utilisation "standard" de Pile-Indicée 119

V.3 Utilisation de la stratégie Arbres de Buts Indicés 120
V.3.1 Introduction 120
V.3.2 Exemple 1: Les 8-reines avec la stratégie ABI 120
V.3.3 Exemple 2 122
V.3.4 Exemple 3: Une utilisation particulière de la stratégie ABI 123

Conclusion 125

INTRODUCTION

A) PRESENTATION DU SUJET

La programmation logique consiste à utiliser la logique mathématique comme un langage de programmation. Kowalski [KOW] et Colmerauer [COL a] [ROU] sont à la base du développement de la programmation logique. Kowalski a montré qu'un ensemble de formules logiques pouvait avoir une interprétation procédurale. Dans le même temps, Colmerauer et son équipe créaient le langage PROLOG.

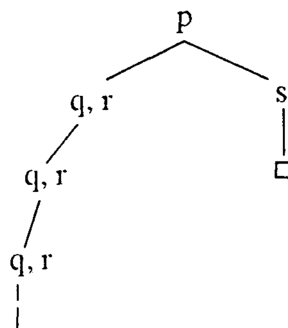
L'idéal de la programmation logique est de permettre une programmation purement déclarative: le programmeur spécifie le problème à résoudre à l'aide de formules logiques, le système détermine comment résoudre le problème posé. Le langage Prolog (si l'on se restreint à un "Prolog pur", c'est-à-dire sans prédicats prédéfinis) est un langage déclaratif, cependant les systèmes Prolog actuels sont encore loin de cet idéal, en particulier à cause de la procédure de résolution utilisée qui n'est pas toujours adaptée.

Un interpréteur PROLOG est la mise en oeuvre d'une procédure de réfutation particulière pour la résolution SLD. La résolution SLD est une application du principe de résolution de Robinson [ROB] à un sous-ensemble de la logique du premier ordre, les clauses de Horn. La résolution SLD est correcte et complète: toutes les réponses calculées sont des conséquences logiques du programme, et toutes les conséquences logiques peuvent être calculées.

Malheureusement, les résultats de correction et de complétude ne s'appliquent pas aux interpréteurs standards. Considérons par exemple le programme suivant:

```
p :- q,r.
p :- s.
q :- q.
s.
```

Pour prouver le but $\leftarrow$p, un interpréteur Prolog parcourt l'arbre de dérivation suivant:



Un interpréteur standard ne trouvera pas de succès car la branche gauche est infinie et l'arbre est parcouru en profondeur d'abord. La branche gauche est infinie alors que l'un des

sous-buts, r, donne un échec: le sous-but sélectionné est toujours le premier sous-but de la liste. L'incomplétude d'un interpréteur Prolog standard provient donc de deux points:

- le choix d'un atome dans une liste de buts
- le type de parcours de l'arbre de dérivation

Les résultats de complétude de la résolution SLD (voir par exemple [APT a] ou [LLO]) ne s'appliquent qu'à des interpréteurs "idéaux" [DEL], c'est-à-dire utilisant une stratégie de recherche équitable (tous les noeuds de l'arbre de dérivation sont parcourus), par exemple en largeur d'abord, et une règle de choix de l'atome dans une liste de buts (ce que nous désignons par règle d'évaluation ou stratégie de dérivation) équitable (tout atome est sélectionné lorsque la dérivation n'échoue pas).

Un parcours de l'arbre en largeur d'abord est trop coûteux pour être effectivement utilisé. Par contre, il est possible d'envisager l'utilisation d'une règle de choix équitable: l'objectif de notre travail est de proposer des interpréteurs Prolog plus proches de l'interpréteur "idéal" en définissant des règles de choix équitables.

B) PLAN DE LA THESE ET RESUME DES CHAPITRES

Notre travail a consisté à étudier les stratégies de dérivation équitables, des aspects théoriques aux aspects pratiques.

Dans le chapitre I, nous donnons les définitions et résultats de base concernant la résolution SLD et montrons l'intérêt théorique des stratégies équitables. Dans le deuxième chapitre, nous introduisons deux stratégies de dérivation équitable, Pile-indicée et Arbres-de-Buts-Indicés. Le troisième chapitre décrit la réalisation d'interpréteurs équitables basés sur ces stratégies. Dans le quatrième chapitre, nous comparons les stratégies équitables avec les techniques de contrôle ayant pour effet de modifier la règle d'évaluation utilisée. Le chapitre V est consacré à l'utilisation pratique des stratégies Pile-Indicée et Arbres-de-buts-indicés.

• Intérêt théorique des interpréteurs équitables (Chapitre I)

La notion de dérivation équitable a été introduite par Lassez et Maher [L, M]: un arbre de dérivation est équitable si tout atome apparaissant dans une liste de buts d'une branche infinie est effectivement sélectionné au bout d'un nombre fini d'étapes. Ils ont montré que de tels arbres sont complets pour les échecs finis: s'il existe un arbre de dérivation fini et sans clause vide, alors tout arbre de dérivation est fini et sans clause vide. Un interpréteur Prolog est équitable si tout arbre de dérivation envisagé par cet interpréteur est un arbre équitable.

La négation par l'échec est basée sur les échecs finis: si un atome clos At est dans l'ensemble des échecs finis du programme, alors on déduit ←At. L'utilisation d'une règle d'évaluation équitable permet donc une implémentation correcte et complète de la négation par l'échec.

Nous avons étendu le résultat de Lassez et Maher en montrant qu'il est vrai également pour les arbres de dérivation contenant la clause vide: s'il existe un arbre de dérivation SLD fini pour un programme P et un but ←At, alors tout arbre de dérivation équitable pour P et ←At est fini et contient la clause vide. Ce résultat accroit l'intérêt des règles d'évaluation équitables: s'il existe une règle d'évaluation qui donne un arbre de dérivation SLD fini, alors toute règle d'évaluation équitable donne un arbre de dérivation fini.

Une conséquence de cette propriété est que, avec un interpréteur équitable, la terminaison d'une dérivation SLD ne dépend pas de l'ordre des atomes dans les corps de clause, ce qui n'est pas toujours le cas avec un interpréteur standard. De ce fait, certains prédicats (par exemple naïve-reverse/2 ou permut/2) sont réversibles avec une stratégie de dérivation équitable alors qu'ils ne le sont pas avec la stratégie standard. Une règle d'évaluation

équitable permet donc d'éviter toutes les boucles liées à l'interpréteur, ce qui autorise une programmation plus déclarative.

• Définition de stratégies équitables (Chapitre II)

La définition d'un interpréteur Prolog nécessite le choix d'une stratégie de dérivation. La stratégie de dérivation ou règle d'évaluation ("computation rule") est le mécanisme utilisé pour choisir un atome dans une liste de sous-buts.

Un interpréteur Prolog standard utilise une stratégie de dérivation "en pile": l'atome sélectionné est toujours le premier élément de la liste et les nouveaux sous-buts obtenus sont ajoutés en tête. De ce fait, si l'un des sous-buts de la liste donne une dérivation infinie, les sous-buts qui se trouvent derrière ce but ne seront jamais examinés.

Une stratégie de dérivation est équitable si tout arbre de dérivation SLD obtenu en utilisant cette stratégie est équitable, c'est à dire si aucun atome apparaissant dans une liste de buts n'est laissé indéfiniment non sélectionné. Le moyen le plus simple pour obtenir un interpréteur équitable consiste à gérer la liste de buts "en file": L'atome sélectionné est le premier élément de la liste, mais les éventuels nouveaux sous-buts obtenus sont ajoutés en queue de la liste.

Nous avons étudié cette stratégie et montré qu'elle est inadaptée à la programmation Prolog dans de nombreux cas: la complexité de la résolution augmente et le programmeur perd toute possibilité de contrôle.

Nous proposons deux stratégies de dérivation originales basées sur l'introduction d'indices de dérivation dans les programmes et montrons que ces stratégies sont équitables.

La première stratégie définie, appelée "Pile-Indicée" consiste en quelque sorte à mixer la stratégie standard "en pile" et la stratégie équitable "en file" en contrôlant ce mixage à l'aide d'indices. Cette stratégie, simple à mettre en oeuvre, permet d'apporter l'équité à faible coût.

La seconde stratégie, appelée "Arbre de Buts Indicés", a été tout d'abord proposée par Delahaye et Paradinas [D, P]. Nous avons étendu cette stratégie de manière à la rendre plus souple et plus facilement utilisable en pratique. Cette stratégie qui utilise une structure d'arbre pour représenter la résolvante offre de nouveaux moyens de contrôle. Il est par exemple très facile de définir un mécanisme de coroutine entre deux sous-buts.

• Réalisation d'interpréteurs équitables (Chapitre III)

Deux implémentations des stratégies équitables Pile-Indicée et Arbres-de-Buts-Indicés ont été réalisées.

Une implémentation a été faite en Prolog, sous la forme d'un méta-interpréteur pouvant utiliser l'une ou l'autre des stratégies. Les prédicats prédéfinis ainsi que le "cut" peuvent être utilisés dans les programmes indicés.

Une deuxième implémentation a été réalisée à partir de l'interpréteur PrologII/MALI [LEHa], en collaboration avec S. Le Huitouze (IRISA-RENNES). La machine MALI est une machine abstraite dédiée à l'implémentation des langages logiques ([BEK a], [RID]). PrologII/MALI est un interpréteur pour le langage PrologII implémenté avec MALI.

Deux interpréteurs équitables pour les stratégies Pile-Indicée et Arbres-de-Buts-Indicés ont été réalisés en modifiant PrologII/MALI. Ceci nous a permis d'hériter de tout l'environnement de programmation ainsi que de la gestion de mémoire de PrologII/MALI.

Dans les deux cas, les langages implémentés sont des sur-ensembles de Prolog: il est possible de mettre des indices seulement pour certains sous-buts ou même de ne pas en mettre du tout, auquel cas la stratégie effectivement utilisée correspond à la stratégie standard.

• Comparaison avec les techniques de contrôle (Chapitre IV)

De nombreux travaux ont été faits pour améliorer le contrôle Prolog. Nous étudions les méthodes ayant pour effet de modifier la règle d'évaluation utilisée.

Ces méthodes sont efficaces pour optimiser des programmes qui sont exécutés de manière très inefficace avec la stratégie standard, en ce sens qu'elles permettent de minimiser l'arbre de dérivation envisagé. Elles permettent de supprimer certaines boucles liées à la stratégie standard. Cependant, contrairement aux stratégies équitables, elles ne sont pas complètes et ne permettent pas d'éviter toutes les boucles liées à la règle d'évaluation.

• Utilisation des stratégies Pile-indicée et Arbres-de-buts-indicés (Chapitre 5)

Dans ce chapitre, nous étudions l'utilisation des stratégies Pile-Indicée et Arbres de Buts Indicés.

La stratégie Pile-Indicée est simple à mettre en oeuvre et peut être implémentée efficacement. Par contre, l'usage des indices de dérivation comme outil de contrôle n'est pas satisfaisant. Aussi, le principal usage de cette stratégie est d'apporter l'équité à la règle standard en modifiant le moins possible sa sémantique procédurale: en ajoutant automatiquement des indices égaux à N aux sous-buts, la règle d'évaluation est équitable mais ne diffère de la règle standard que pour des dérivations de longueur supérieure à N.

En revanche, la stratégie Arbres de Buts Indicés est plus coûteuse, mais est plus souple d'utilisation et permet de programmer différemment. Des exemples de programmes simples qui exploitent le mécanisme de la stratégie ABI (et qui donc ne fonctionnent pas en Prolog standard) sont présentés. Le problème du choix des indices de dérivation est ensuite abordé.

• Conclusion

Le principal intérêt des stratégies équitables et d'apporter la complétude pour les échecs finis et les succès finis. Dans certains cas, l'usage d'une règle équitable permet en plus d'améliorer l'efficacité (en produisant un arbre SLD plus petit), cependant, dans d'autres cas, cela peut-être une source d'inefficacité, en particulier lorsque la règle standard donne une dérivation optimale. Pour palier à cet inconvénient, il serait intéressant de pouvoir caractériser les programmes pour lesquels l'équité est effectivement utile. Il serait également intéressant d'essayer de combiner l'usage d'une règle équitable avec d'autres techniques de contrôle ou des techniques d'optimisation de programmes.

Notre travail montre que l'utilisation effective de règles d'évaluation différentes et équitables est possible.

Chapitre I :

Résolution SLD, fondements et aspects théoriques des stratégies équitables

I.1 Introduction

Dans ce chapitre, nous donnons les notations et définitions concernant la résolution SLD pour les programmes définis.

L'obtention ou non d'une clause vide dans un arbre de dérivation SLD ne dépend pas de la règle d'évaluation utilisée. Par contre, l'obtention ou non d'un échec peut dépendre de la règle d'évaluation. Lassez et Maher ont introduit la notion de dérivation équitable - une dérivation est équitable si elle est finie ou si tout atome est sélectionné - et ont montré que les arbres de dérivation équitables donnent tous les échecs finis [L, M].

Nous étendons le résultat de Lassez et Maher en montrant qu'il est vrai également pour des arbres de dérivation contenant des clauses vides: s'il existe un arbre de dérivation SLD fini pour un programme P et un but $\leftarrow$B, alors tout arbre de dérivation équitable pour $P \cup \{\leftarrow B\}$ est fini. Une conséquence de ce résultat est que la terminaison d'un but avec une règle d'évaluation équitable ne dépend pas de l'ordre des sous-buts dans un corps de clause.

Dans le cadre des programmes définis, l'utilisation d'une règle d'évaluation équitable permet de réaliser une implémentation correcte et complête de la négation par l'échec. Il n'en est pas de même lorsque l'on considère des programmes comportant des littéraux négatifs dans les corps de clause (programme normaux). Des résultats de complétude pour la SLDNF ont été obtenus sous des hypothèses assez restrictives. Cependant, et ce contrairement au cas des programmes définis, il n'est pas possible de concevoir facilement des interpréteurs Prolog auxquels ces résultats soient applicables.

I.2 Résolution SLD

I.2.1 Introduction

La résolution SLD est une méthode de résolution particulière pour les clauses de Horn, c'est à dire des clauses comportant exactement un littéral positif. Nous donnons tout d'abord les définitions classiques concernant les programmes définis et la résolution SLD, puis nous rappelons les principaux résultats de correction (toute réponse calculée est correcte) et de complétude (toute réponse correcte est calculée) de la résolution SLD. Ces résultats sont dus essentiellement à Hill [HIL], Clark [CLA a], Apt et Van Emden [A, VE] et Lloyd [LLO] et sont rassemblés dans [LLO]. Nous suivons en général la présentation de Lloyd.

I.2.2 Définitions et notations

I.2.2.1 Programmes définis

Définition: Une *clause définie* est une clause de la forme:

$A \leftarrow B_1, B_2, \dots , B_n$ où $A, B_1, B_2, \dots ,B_n$ sont des atomes.

A est la tête de la clause et $B_1, B_2, \dots , B_n$ est le corps de la clause.

Un *programme défini* est une suite finie de clauses définies.

Définition: Un *but défini* est une clause négative, c'est à dire une clause de la forme :

$\leftarrow B_1, B_2, \dots ,B_n$ où $B_1, B_2, \dots , B_n$ sont des atomes.

Définition: Un *atome* est une formule de la forme $p(t_1, t_2, \dots , t_n)$, où p est un symbole de prédicat et $t_1, t_2, \dots, t_k$ sont des *termes*.

Un *terme* est :

-soit une constante

-soit une variable

-soit une formule de la forme $f(t_1, t_2, \dots, t_k)$, où $t_1, t_2, \dots, t_k$ sont des termes.

Exemple:

```
p(X):- q(X),r(X).
q(X):- q(f(X)).
r(f(f(a))).
```

est un programme défini, $\leftarrow$`p(X)` est un but défini.

I.2.2.2 Dérivation SLD

Définition: Soient G le but défini: $\leftarrow A_1, \dots, A_m, \dots, A_k$ et C la clause $A \leftarrow B_1, \dots , B_q$.

Alors G' dérive de G et C par la résolution SLD en utilisant l'unificateur le plus général θ si:

- θ est un plus grand unificateur de Am et A

- G' est le but $\leftarrow (A_1, \dots ,A_{m-1}, B_1, \dots ,B_q, A_{m+1}, \dots A_k)\theta$

A_m est appelé *l'atome choisi* et G' est une *résolvante* de G et C.

Si G n'a qu'un seul atome et que C est une clause ne comportant qu'un seul littéral, alors G' est égal à la clause vide notée ❑.

<u>Définition:</u> Une *dérivation SLD* est une suite (finie ou infinie) de triplets (L_i, θ_i, C_i) telle que L_{i+1} dérive de L_i et C_i en utilisant l'unificateur θ_i.

Une dérivation SLD pour un programme défini P et un but défini B est une dérivation SLD telle que L_1=B et les clauses C_i sont des clauses de P.

Une *réfutation SLD* est une dérivation SLD qui se termine par la clause vide.

I.2.3 Sémantique déclarative des programmes définis

La sémantique déclarative des programmes définis est donnée par les modèles de Herbrand: l'ensemble des conséquences logiques d'un programme est égal à son plus petit modèle de Herbrand.

I.2.3.1 Base de Herbrand, modèles de Herbrand d'un programme

<u>Définition:</u> On note l(P) le langage associé à un programme défini P. l(P) est l'ensemble des symboles de prédicats et de fonctions apparaissant dans P.

La *base de Herbrand* d'un programme défini P, notée B(P), est l'ensemble des atomes sans variables construits à partir du langage associé à P, l(P).

<u>Définition:</u> Une *interprétation de Herbrand* d'un programme P est un sous-ensemble de la base de Herbrand de P (Ce sous-ensemble correspond à l'ensemble des atomes clos vrais dans cette interprétation).

Un *modèle de Herbrand* d'un programme P est une interprétation de Herbrand de P qui est un modèle de P

I.2.3.2 Réponses correctes et plus petit modèle de Herbrand

<u>Définition:</u> Soient P un programme défini et G un but défini. Une *réponse* pour P∪{G} est une substitution pour les variables de G.

<u>Définition:</u> Soient P un programme défini, G un but défini $\leftarrow A_1, ..., A_k$.

Soit θ une réponse pour $P \cup \{G\}$. On dit que θ est une *réponse correcte* pour $P \cup \{G\}$ si:

$$\forall((A_1 \wedge A_2 \wedge \ldots \wedge A_n)\theta) \text{ est conséquence logique de P.}$$

Autrement dit, θ est une réponse correcte si et ssi:

$$P \cup \sim(\forall(A_1 \wedge A_2 \wedge \ldots \wedge A_n)\theta) \text{ est insatisfiable.}$$

Théorème:

Soient P *un programme défini et* G *un but défini* $<-A_1, A_2, \ldots, A_k$. *Soit* θ *une réponse pour* $P \cup \{G\}$ *telle que* $(A_1 \wedge A_2 \wedge \ldots \wedge A_k)\theta$ *soit clos. Alors les propositions suivantes sont logiquement équivalentes:*

(a) θ *est correcte*

(b) $(A_1 \wedge A_2 \wedge \ldots \wedge A_k)\theta$ *est vraie dans tout modèle de Herbrand de* P.

(c) $(A_1 \wedge A_2 \wedge \ldots \wedge A_k)\theta$ *est vraie dans le plus petit modèle de Herbrand de* P.

I.2.4 Justesse de la résolution SLD

I.2.4.1 Ensemble de succès d'un programme

Définition: Soit P un programme défini.

L'*ensemble des succès* du programme P, noté SS(P) (Set of Success), est l'ensemble des atomes At appartenant à la base de Herbrand de P tels qu'il existe une réfutation SLD pour $P \cup \{\leftarrow At\}$.

I.2.4.2 Justesse de la résolution SLD

Définition: Soient P un programme défini et G un but défini. Une réponse calculée θ pour $P \cup \{\leftarrow G\}$ est la substitution obtenue en restreignant la composition $\theta_1, \theta_2, \ldots, \theta_n$ aux variables de G, où $\theta_1, \theta_2, \ldots, \theta_n$ est la suite d'unificateurs les plus généraux utilisés dans une réfutation SLD de $P \cup \{\leftarrow G\}$.

Théorème:

Soient P *un programme défini et* G *un but défini. Alors toute réponse calculée pour* $P \cup \{G\}$ *est une réponse correcte pour* $P \cup \{G\}$.

Ce résultat établit la correction de la résolution SLD: toutes les réponses produites par la résolution SLD sont des réponses correctes. L'ensemble des succès d'un programme est donc inclus dans son plus petit modèle de Herbrand.

I.2.5 Complétude de la résolution SLD

Les résultats de complétude montrent que la résolution SLD permet de produire toutes les réponses correctes. Le premier résultat de complétude, qui établit la réciproque du résultat précédent, est dû à Apt et Van Emden [A, VE]:

Théorème:

L'ensemble des succès d'un programme défini est égal à son plus petit modèle de Herbrand.

Le résultat suivant a été établi par Hill [HIL] ainsi que par Apt et Van Emden [A, VE]:

Théorème:

Soit P *un programme défini et* G *un but défini. Si* $P \cup \{G\}$ *est insatisfiable, alors il existe une réfutation SLD de* $P \cup \{G\}$.

Ce théorème est renforcé par un résultat de Clark [CLAa] qui montre que toute réponse correcte est une instance d'une réponse calculée:

Théorème (Complétude de la résolution SLD):

Soient P *un programme défini et* G *un but défini. Pour toute réponse correcte* θ *pour* $P \cup \{G\}$, *il existe une réponse calculée* σ *pour* $P \cup \{G\}$ *et une substitution* γ *telles que* $\theta = \sigma\gamma$.

Ces résultats établissent la correction et la complétude de la résolution SLD: la résolution SLD ne produit que des réponses correctes et produit toutes les réponses correctes.

I.2.6 Indépendance de la règle d'évaluation

I.2.6.1 Règle d'évaluation

Dans la définition d'une dérivation SLD, nous avons parlé de l'atome choisi sans préciser la manière dont cet atome est choisi. Nous introduisons maintenant la notion de règle d'évaluation. La règle d'évaluation désigne la méthode employée pour choisir un atome dans une liste de buts.

Définition: Une *règle d'évaluation* (ou *stratégie de dérivation*) est une fonction qui à une dérivation SLD $D = L_0, L_1, \dots L_i$ associe un atome de L_i, appelé l'atome choisi.

Dans [LLO], Lloyd donne une définition plus restrictive d'une règle d'évaluation: Une règle d'évaluation est une fonction d'un ensemble de buts vers un ensemble d'atomes, telle que la valeur de la fonction sur un but soit un atome de ce but, appelé l'atome choisi. Lloyd précise que cette définition est restrictive, en ce sens que si le même but apparait à plusieurs reprises dans une dérivation SLD, le même atome sera choisi; il y a donc des dérivations SLD qui ne peuvent pas être obtenues en utilisant une règle d'évaluation.

La notion de règle d'évaluation que nous utilisons est plus générale. Nous considérons que toute dérivation SLD peut être obtenue en utilisant une certaine règle d'évaluation.

Définition: Soient P un programme défini, G un but défini et R une règle d'évaluation. Une dérivation SLD via R pour P∪{G} est une dérivation SLD de P∪{G} dans laquelle la règle d'évaluation R est utilisée pour choisir les atomes.

I.2.6.2 Indépendance de la règle d'évaluation

Le théorème suivant montre que l'obtention ou non de la clause vide ne dépend pas de la stratégie de dérivation utilisée ([LLO], [A, VE]).

Théorème:

Soient P *un programme défini et* G *un but défini. Supposons qu'il existe une réfutation SLD de* P∪{G} *avec* σ *pour réponse calculée. Alors pour n'importe quelle règle d'évaluation* R, *il existe une réfutation SLD de* P∪{G} *avec une réponse calculée* σ' *telle que* Gσ' *est une variante de* Gσ *(c'est à dire que* Gσ *et* Gσ' *sont identiques à un renommage de variables près).*

Ce résultat fondamental pour la mise en oeuvre de procédures de réfutation montre que, en théorie, on peut utiliser n'importe quelle stratégie de dérivation pour trouver une réfutation. Autrement dit, à chaque étape d'une dérivation, n'importe quel atome peut être choisi.

I.2.7 Arbres de dérivation SLD

Nous définissons maintenant les arbres de dérivation SLD. Un arbre de dérivation SLD correspond à un arbre de recherche envisagé par un système de programmation logique pour rechercher une réfutation.

Définition: Soient P un programme défini et G un but défini. L'arbre de dérivation SLD pour P∪{G} est l'arbre obtenu de la manière suivante:

- La racine de l'arbre est G

- Chaque noeud de l'arbre est un but défini
- Si $\leftarrow A_1, ..., A_m, ..., A_k$ est un noeud de l'arbre et A_m l'atome choisi, alors pour toute clause de P $A \leftarrow B_1, ... ,B_q$ telle que A et A_k soient unifiables avec θ comme plus grand unificateur, le noeud a un fils correspondant :

  $\leftarrow(A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ..., A_k)\theta.$

  Les différents noeuds sont ordonnés suivant l'ordre des clauses utilisées pour les obtenir. Si k=1 et q=0, le noeud fils obtenu est la clause vide, notée ❑.

Chaque branche de l'arbre est une dérivation SLD. Les branches correspondant aux dérivations réussies, c'est-à-dire qui se terminent par la clause vide sont appelées branches de succès. Les branches finies qui ne se terminent pas par la clause vide sont appelées branches d'échec.

Le théorème suivant [A, VE], qui découle de l'indépendance de la règle d'évaluation, montre la complétude des arbres de dérivation SLD:

Théorème:

S'il existe un arbre de dérivation SLD pour $P \cup \{G\}$ contenant la clause vide, alors tout arbre de dérivation SLD pour $P \cup \{G\}$ contient la clause vide.

Ce résultat montre que lorsqu'il existe une réfutation de G, n'importe quel arbre de dérivation SLD de racine G contient la clause vide. Pour trouver une réfutation de G, il suffit donc de parcourir l'un des arbres de dérivation SLD de racine G.

Le résultat suivant [LLO] précise ce résultat en montrant que les différents arbres de dérivation possibles ont les mêmes branches de succès:

Théorème:

Soient P *un programme défini et* G *un but défini. Alors soit tout arbre SLD pour* $P \cup \{G\}$ *a une infinité de branches de succès, soit tout arbre a le même nombre fini de branches de succès.*

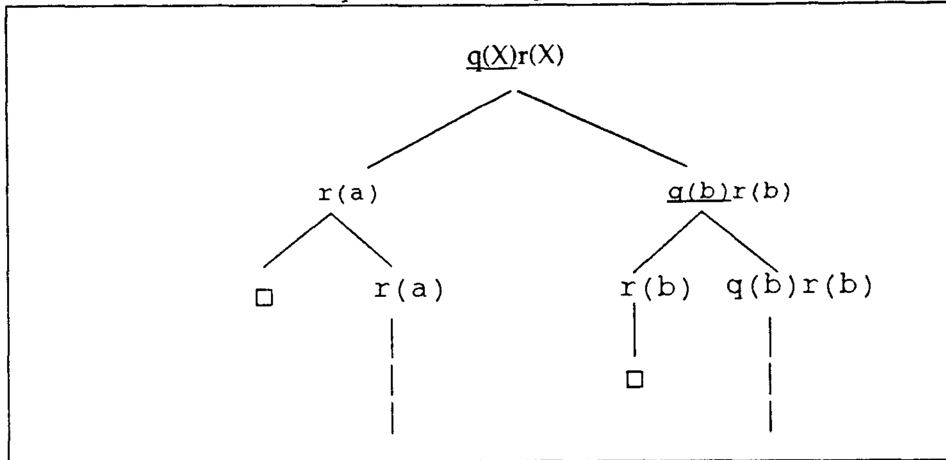
Les branches de succès sont équivalentes: les clauses utilisées sont les mêmes et les branches ont même longueur. Cependant, les arbres de dérivation SLD peuvent différer considérablement de par leur taille en raison du nombre de branches d'échecs et de l'existence ou non de branches infinies. Certains arbres peuvent être finis alors que d'autres arbres sont infinis.

Exemples:

On considère le programme défini P:

```
q(a).
q(b).
q(b):- q(b).
r(b).
r(a).
r(a) <- r(a).
```

Voici un arbre de dérivation SLD pour le but ←`q(X),r(X)`:

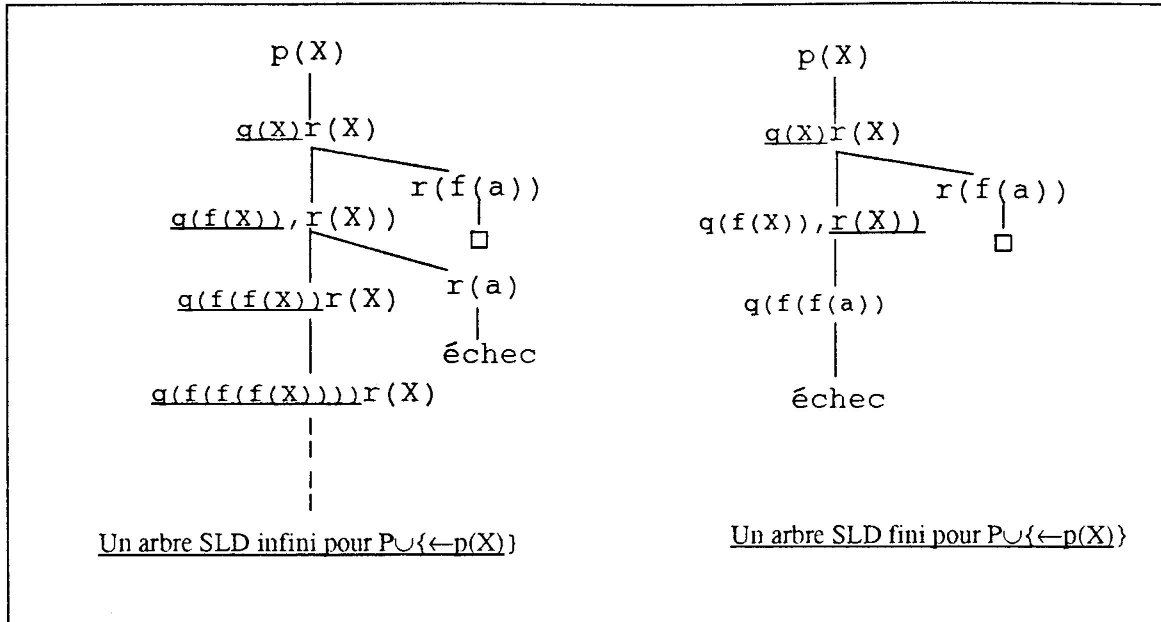


L'atome choisi dans chaque liste d'atomes est l'atome souligné. Dans ce cas tout arbre de dérivation SLD pour ←`q(X,r(X)` comprend deux branches de succès et deux branches infinies.

Soit le programme P:

```
p(X):- q(X), r(X).
q(X):- q(f(X)).
r(f(a)).
q(f(a)).
```

Pour ←`p(X)`, il existe des arbres de dérivation SLD finis et des arbres infinis:



Un arbre SLD infini pour P∪{←p(X)}

Un arbre SLD fini pour P∪{←p(X)}

I.2.8 Echecs finis d'un programme

Les résultats précédents caractérisent l'ensemble des succès obtenus pour la résolution SLD. Nous nous intéressons maintenant aux atomes pour lesquels la résolution SLD donne un échec. Les échecs finis d'un programme sont définis de la manière suivante:

Définition: L'ensemble des échecs finis d'un programme défini P est l'ensemble des atomes A de Bp pour lesquels il existe un arbre de dérivation SLD fini et sans clause vide (un tel arbre est appelé arbre d'échec fini).

L'ensemble des échecs finis d'un programme P correspond aux faits tels que la résolution SLD permet de prouver qu'ils ne sont pas conséquence logique de P.

La résolution SLD ne permet de déduire que de l'information positive. Pour déduire des faits négatifs, il faut introduire une nouvelle règle d'inférence. La règle de négation par l'échec proposée par Clark [CLA a], est basée sur les échecs finis:

Si un atome clos At est dans l'ensemble des échecs finis du programme P, alors on déduit ~A.

La complétion d'un programme P (notée comp(P), [CLA a]), qui consiste à compléter le programme en ajoutant une partie "seulement si" à chaque définition de prédicat, justifie l'utilisation de la négation par l'échec en caractérisant l'ensemble des échecs finis en terme de conséquence logique (la définition de la complétion d'un programme est donnée dans le §I.5.2):

Théorème:

Soit P *un programme défini et* A *un atome de* B(P). *A est dans l'ensemble des échecs finis de* P *si et seulement si* ~A *est conséquence logique de* comp(P).

Clark a prouvé la partie "seulement si" de ce théorème, qui montre la correction de la négation par l'échec [CLA a]. La partie "si" , qui montre la complétude, a été prouvée par Jaffar, Lassez et Lloyd [J, L, L].

La résolution SLD permet de déduire des conséquences logiques d'un programme. Ce résultat montre que les faits négatifs déduits par la règle de négation par l'échec sont également des conséquences logiques, mais du programme complété.

Pour qu'un atome soit dans l'ensemble des échecs finis, il faut qu'il existe un arbre de dérivation SLD fini et sans clause vide et non que tous les arbres soient finis et sans clause vide. Il se peut que certains arbres soient finis alors que d'autres ne le sont pas. Cela dépend de la règle d'évaluation utilisée. Lorsqu'on cherche à prouver qu'un atome appartient à l'ensemble des échecs finis, il est donc primordial de savoir quelles règles d'évaluation donnent des arbres finis (lorsque certains arbres sont finis). C'est dans ce but que Lassez et Maher [L, M] ont introduit la notion de règle d'évaluation équitable, que nous étudions dans le paragraphe suivant.

I.3 Résolution SLD équitable

I.3.1 Introduction

Nous nous intéressons maintenant aux dérivations SLD équitables. La notion d'équité a été introduite par Lassez et Maher [L, M]. L'équité concerne la règle d'évaluation: une règle d'évaluation est équitable si tous les atomes apparaissant dans une dérivation qui n'échoue pas sont sélectionnés.

Lassez et Maher ont montré que l'utilisation de règles d'évaluation équitables permet d'obtenir des arbres de dérivation SLD complets pour les échecs finis: S'il existe un arbre de dérivation SLD fini et sans clause vide pour un programme P et un but G, alors tout arbre de dérivation SLD obtenu via une règle d'évaluation équitable est fini et sans clause vide. Ce résultat montre l'intérêt qu'il y a à utiliser des dérivations équitables, en particulier lorsqu'on introduit la règle de négation par l'échec.

Dans ce paragraphe, nous démontrons un résultat qui généralise celui de Lassez et Maher: Si l'un des arbres de dérivation SLD pour $P \cup \{G\}$ est fini, alors tout arbre de dérivation SLD équitable pour $P \cup \{G\}$ est fini. Ce résultat a des conséquences pratiques intéressantes, qui sont étudiées dans le paragraphe suivant.

I.3.2 Dérivations équitables

I.3.2.1 Définitions

Définition: Une dérivation SLD est *équitable* si elle échoue ou bien si, pour chaque atome At apparaissant dans un but G_i de cette dérivation, il existe un but G_k dans lequel At (ou une version instanciée de At) est l'atome sélectionné.

Définition: Un *arbre de dérivation SLD* est *équitable* si chaque chemin de l'arbre est une dérivation équitable, c'est-à-dire si tout atome apparaissant dans une branche infinie est sélectionné.

Définition: Une *stratégie de dérivation S* est *équitable* si toute dérivation via S est une dérivation équitable.

Exemple: Dans le dernier exemple du §I.2.7, l'arbre infini n'est pas équitable car le deuxième sous-but n'est jamais sélectionné. La stratégie de dérivation qui consiste à sélectionner toujours le premier atome n'est pas équitable.

I.3.2.2 Propriétés des dérivations équitables

Nous montrons maintenant une propriété fondamentale des dérivations équitables: s'il existe une dérivation SLD équitable infinie pour $P \cup \{G\}$, alors pour toute stratégie de dérivation S, il existe une dérivation SLD via S pour $P \cup \{G\}$ qui est infinie.

La démonstration de ce résultat utilise le lemme d'interversion ("switching lemma") du à Lloyd ([LLO], lemme 9.1) . Ce lemme montre que l'on peut intervertir le choix de deux atomes dans une dérivation SLD sans que cela modifie le reste de la dérivation:

Lemme ("Switching lemma"):

Soient P *un programme défini et* G *un but défini.*

On considère une dérivation SLD pour $P \cup \{G\}$*:*

$D=G, G_1, \ldots, G_{q-1}, G_q, G_{q+1}, \ldots$

avec $C_1, \ldots, C_{q-1}, C_q, C_{q+1}$ *pour clauses d'entrée*

et $\theta_1, \ldots, \theta_{q-1}, \theta_q, \theta_{q+1}$ *pour mgu.*

Soit C_k- *le corps de la clause* C_k *et* C_k+ *la tête de* C_k.

On a: $G_{q-1}= A_1, \ldots, A_{i-1}, A_i, \ldots, A_{j-1}, A_j, \ldots, A_k, \ldots$

On suppose que A_i *est l'atome sélectionné dans* G_{q-1} *et* A_j *l'atome sélectionné dans* G_{q+1}, $G_q=(A_1, \ldots, A_{i-1}, C_{q-}, \ldots, A_{j-1}, A_j, \ldots, A_k)\theta_q$

$G_{q+1}=(A_1, \ldots, A_{i-1}, C_q^-, \ldots, A_{j-1}, C_{q+1}^-, \ldots Ak)\theta_q\theta_{q+1}$

Alors il existe une dérivation SLD de $P \cup \{G\}$ *dans laquelle* A_j *est choisi dans* G_{q-1} *au lieu de* A_i, A_i *est choisi dans* G_q *au lieu de* A_j *et le reste de la dérivation est identique modulo des variantes.*

La preuve de ce lemme ([LLO]) est un calcul technique sur les substitutions. Ce résultat est très utile. Il est utilisé par Lloyd pour montrer l'indépendance de la règle d'évaluation ([LLO], Théorème 9.2) ainsi que pour montrer l'équivalence entre les branches de succès de différents arbres SLD.

Dans [LLO], l'énoncé de ce lemme porte sur une réfutation SLD et non sur une dérivation quelconque. Cependant, cette restriction n'intervient pas dans la démonstration et donc le lemme est vrai également pour une dérivation quelconque, éventuellement infinie.

Nous pouvons maintenant énoncer le théorème sur les dérivations équitables:

Théorème:

Soient P *un programme défini et* G *un but défini.*

S'il existe une dérivation SLD infinie et équitable pour $P \cup \{G\}$, *alors pour toute stratégie de dérivation* S, *il existe une dérivation SLD via* S *pour* $P \cup \{G\}$ *qui est infinie.*

Preuve:

La démonstration de ce résultat est analogue à la démonstration du théorème montrant l'indépendance de la stratégie de dérivation qui est suggérée par Lloyd ([LLO], problème 15).

Soit $D=G_0, G_1, \ldots$ une dérivation infinie équitable de $P \cup \{G\}$.

Soit S une stratégie de dérivation quelconque.

La démonstration consiste à construire une dérivation via S de $P \cup \{G\}$ à partir de la dérivation D. Cette construction se fait en appliquant le lemme d'interversion à D.

On montre par récurrence que pour tout n, il existe une dérivation SLD de $P \cup \{G\}$ infinie et équitable qui soit une dérivation via S jusqu'à la nième étape au moins.

- Pour n=1: Soit A_1 l'atome de G sélectionné dans D et A_2 l'atome qui est sélectionné dans G en appliquant la stratégie S. A_2 est sélectionné dans D à l'étape k. En appliquant le lemme d'interversion k fois, on obtient une dérivation infinie équitable de $P \cup \{G\}$ dans laquelle A_2 est sélectionné au lieu de A_1 dans G.
- Soit $D_n=G_0, G_1, \ldots, G_n \ldots$ une dérivation infinie équitable de $P \cup \{G\}$ qui est une dérivation via S jusque G_n au moins.

Soit A_i l'atome sélectionné dans G_n.

Soit A_j l'atome sélectionné dans G_n en utilisant la stratégie S.

Si A_j est différent de A_i, alors il existe p tel que A_j est l'atome sélectionné dans G_{n+p}, car la dérivation D_n est équitable.

En appliquant le lemme d'interversion p fois à la dérivation D_n, on obtient une dérivation infinie équitable D_{n+1} dans laquelle A_j est l'atome sélectionné dans G_n. D_{n+1} est donc une dérivation via S jusque G_{n+1} au moins. D_{n+1} est identique (à un renommage des variables prés) à D_n à partir de G_{n+p}. D_{n+1} est donc une dérivation infinie équitable de $P \cup \{G\}$.

On peut donc construire une dérivation infinie via S,

$$Ds=G_0, G_1, \ldots, G_i, \ldots$$

de la manière suivante: Pour tout i, G_i est la ième clause commune aux dérivations SLD D_j telles que j>i. Il existe donc une dérivation SLD infinie via S de $P \cup \{G\}$.

I.3.2.3 Propriétés des arbres de dérivation équitables

Le théorème précédent appliqué aux arbres de dérivation permet d'obtenir les trois corollaires ci-dessous. Ces corollaires montrent l'intérêt des arbres de dérivation équitables.

Corollaire:

> *S'il existe un arbre de dérivation SLD de racine* At *fini alors tout arbre de dérivation équitable de racine* At *est fini.*

Ce premier corollaire s'obtient directement à partir de la définition des arbres SLD et du théorème: s'il existe un arbre de dérivation équitable infini, alors tout arbre de dérivation SLD est infini.

Corollaire:

> *S'il existe un arbre de dérivation SLD de racine* At *fini et contenant la clause vide, alors tout arbre de dérivation SLD équitable de racine* At *est fini et contient la clause vide.*

Ce deuxième corollaire, ainsi que le troisième, est obtenu en combinant le corollaire précédent et le résultat de complétude des arbres SLD.

Corollaire:

> *S'il existe un arbre de dérivation SLD de racine* At *fini et sans clause vide, alors tout arbre de dérivation SLD équitable de racine* At *est fini et sans clause vide.*

Ce dernier corollaire a été démontré précédemment par Lassez et Maher [L, M] et montre l'intérêt des arbres équitables: les arbres équitables permettent d'obtenir tous les échecs finis d'un programme, c'est-à-dire tous les atomes pour lesquels il existe un arbre de dérivation SLD fini et sans clause vide.

Notre deuxième corollaire montre que les arbres de dérivation SLD équitables sont complets également pour ce que nous appelons les succès finis: les arbres équitables permettent d'obtenir tous les atomes pour lesquels il existe un arbre de dérivation fini et contenant la clause vide. Ce résultat a des conséquences pratiques intéressantes, que nous développons dans le paragraphe suivant.

I.4 Interpréteurs PROLOG

I.4.1 Introduction

Un interpréteur Prolog standard est une implémentation particulière de la résolution SLD, qui est correcte et efficace mais malheureusement incomplète, car basée sur une gestion des sous-buts en pile et un parcours de l'arbre de dérivation en profondeur d'abord.

L'utilisation de stratégies de dérivation équitables permet d'obtenir des interpréteurs plus complets, car on obtient tous les succès finis et tous les échecs finis.

Nous étudions dans ce chapitre les différents ensembles de succès et d'échecs obtenus par un interpréteur Prolog, selon que la stratégie de dérivation employée est équitable ou non. Ceci met en évidence l'intérêt pratique des interpréteurs équitables. En particulier, avec une stratégie équitable, la terminaison d'un but ne dépend pas de l'ordre des sous-buts dans un corps de clause.

I.4.2 Interpréteurs Prolog

I.4.2.1 Définition d'un interpréteur

Un interpréteur Prolog (c'est-à-dire une procédure de réfutation SLD) effectue une recherche dans un arbre de dérivation SLD afin de trouver une réfutation, c'est à dire une clause vide. Un interpréteur est spécifié par deux paramètres:

- Le choix de l'arbre de dérivation SLD à explorer, c'est-à-dire le choix d'un atome à dériver dans chaque but. Ce choix est fixé par la stratégie de dérivation employée.
- Le type de parcours de l'arbre SLD obtenu.

Les résultats de correction et de complétude montrent que tous les arbres SLD donnent les mêmes réponses calculées. Cependant, la taille de l'arbre SLD envisagé dépend de la stratégie de dérivation et l'obtention ou non d'une réponse dépend de la manière dont l'arbre est parcouru.

I.4.2.2 Interpréteurs standards

Les interpréteurs Prolog standards sont basés sur des choix de la stratégie de dérivation et du type de parcours de l'arbre de résolution simples et efficaces.

Les listes d'atomes sont gérées comme une pile et les arbres de dérivation sont parcourus en profondeur d'abord avec retour en arrière.

Une gestion des buts en pile signifie que l'atome sélectionné est le premier sous-but de la liste et que, lorsqu'une règle s'applique, ce sous-but est remplacé par les éventuels nouveaux sous-buts obtenus (i.e une instanciation du corps de la règle). Ces choix ont des

conséquences sur les ensembles de succès (les atomes pour lesquels il existe une réfutation SLD) et d'échecs finis obtenus: les interpréteurs Prolog standards ne permettent généralement pas d'obtenir tous les succès et tous les échecs finis d'un programme. Ainsi, les résultats de complétude de la résolution SLD ne s'appliquent pas aux interpréteurs standards.

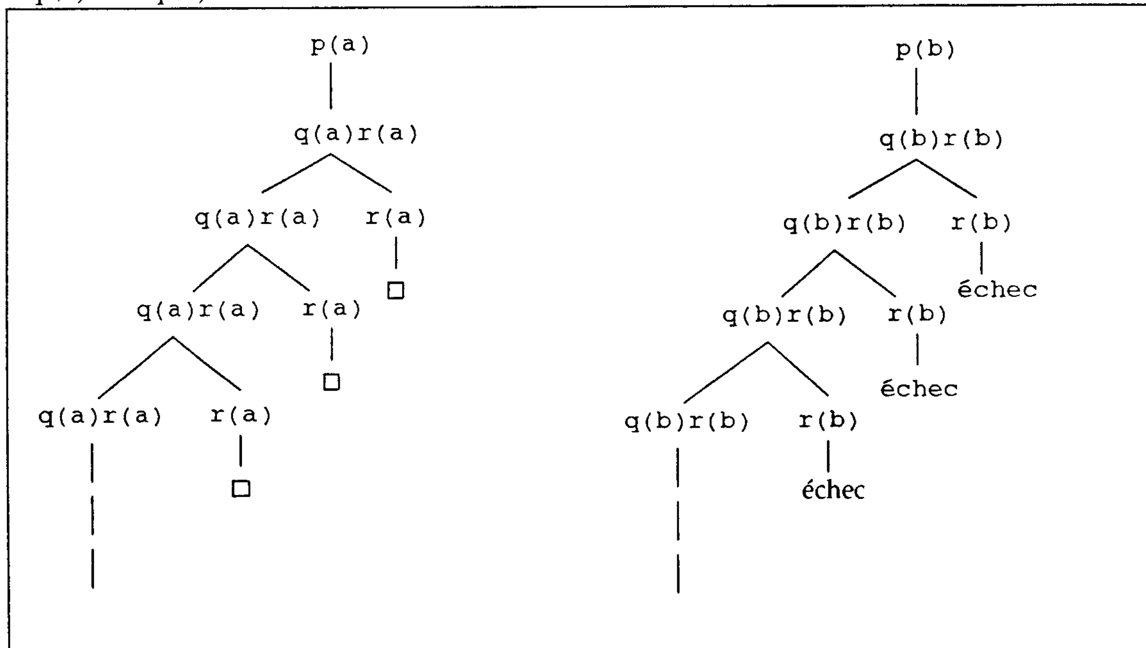
Ce que calcule effectivement un interpréteur Prolog standard n'est donc pas l'ensemble des conséquences logiques d'un programme, mais un ensemble plus petit. La sémantique des interpréteurs standards est étudiée dans [DEL] et [DEN].

L'exemple très simple ci-dessous illustre l'incomplétude des interpréteurs standards.

On considère le programme P:

```
r1: p(X):- q(X)r(X).
r2: q(X):- q(X).
r3: q(a).
r4: q(b).
r5: r(a).
```

Voici les arbres de dérivation SLD engendrés par un interpréteur standard pour les buts ←p(a) et ←p(b):



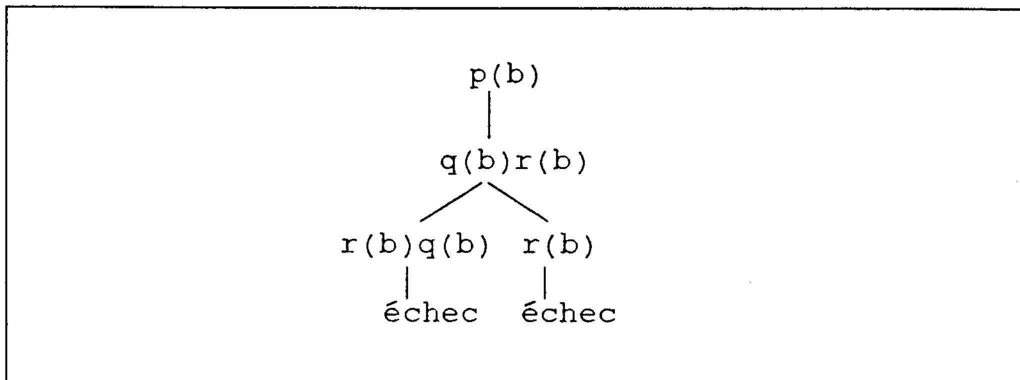
Pour le but ←p(a), l'interpréteur boucle sans jamais trouver la clause vide, car la branche gauche est infinie.

Pour le but ←p(b), l'interpréteur boucle au lieu de donner un échec car l'atome r(b) n'est jamais sélectionné dans la branche gauche de l'arbre.

I.4.2.3 Interpréteurs équitables

Définition: Un *interpréteur Prolog équitable* est un interpréteur qui utilise une stratégie équitable, c'est-à-dire un interpréteur qui ne manipule que des arbres de dérivation SLD équitables.

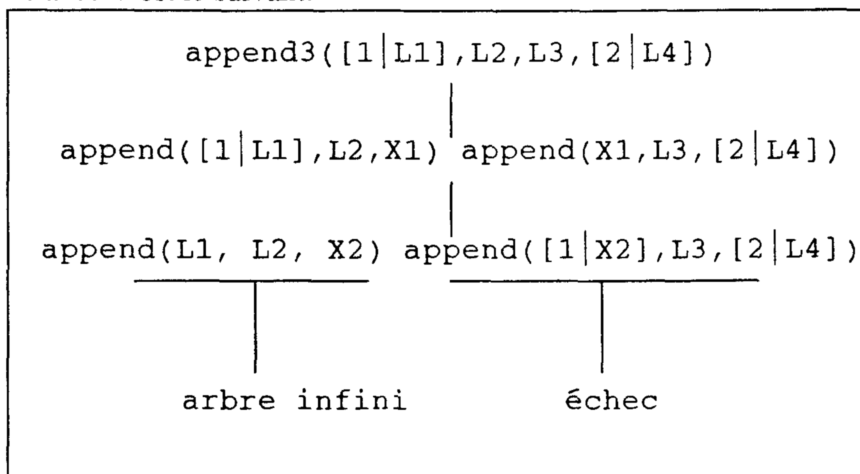
Un interpréteur équitable est donc complet pour les échecs finis et pour les succès finis. Dans l'exemple précédent, n'importe quel arbre équitable pour ←`p(b)` est fini car le sous-but ←`r(b)`, qui échoue, est sélectionné. Par exemple, en gérant les buts en file, on obtient l'arbre suivant:



Considérons le programme `append3` qui permet de concaténer trois listes:

```
append([],L,L).
append([X|L1],L2,[X|L3]):-
                  append(L1,L2,L3).
append3(L1,L2,L3,L4):-
                  append(L1,L2,LL),
                  append(LL,L3,L4).
```

Pour le but ←append3([1|L1,L2,L3,[2|L4]), l'arbre SLD considéré par un interpréteur standard est le suivant:



Le premier sous-but fait boucler un interpréteur standard. Par contre, un interpréteur équitable échoue finiment à cause du second sous-but.

Lorsque nous parlons d'interpréteurs équitables, cela ne concerne que la stratégie de dérivation, c'est à dire la règle de choix du sous-but à dériver dans un but, et non la stratégie de parcours de l'arbre de dérivation SLD.

Dans le paragraphe suivant, nous définissons de manière plus précise les ensembles de succès et d'échecs obtenus par un interpréteur Prolog en fonction de la stratégie de dérivation et du type de parcours de l'arbre de dérivation.

I.4.3 Ensembles d'échecs et de succès

I.4.3.1 Echecs finis

Définition: L'ensemble des échecs finis d'un interpréteur Prolog utilisant une stratégie de dérivation S pour un programme P, noté $FF_S(P)$, est l'ensemble des atomes de B(P) pour lesquels l'interpréteur ne donne pas de succès et ne boucle pas. C'est l'ensemble des atomes At tels que l'arbre de dérivation SLD via la stratégie S de racine At est fini et ne contient pas la clause vide.

L'ensemble des échecs finis obtenus dépend uniquement de la stratégie de dérivation utilisée.

Le résultat de Lassez et Maher [L, M] montre qu'un interpréteur équitable permet d'obtenir tous les échecs finis: l'ensemble des échecs finis d'un interpréteur équitable pour un programme P est égal à l'ensemble des échecs finis de P. Les interpréteurs Prolog équitables sont donc complets par rapport aux échecs finis, contrairement aux interpréteurs standards.

I.4.3.2 Ensembles de succès

Nous définissons ici les différents ensembles de succès obtenus par un interpréteur Prolog pour un programme P en fonction du type de parcours de l'arbre SLD et de la stratégie de dérivation. Nous rappelons la définition de l'ensemble de succès d'un programme puis nous nous intéressons aux ensembles de succès obtenus avec un interpréteur effectuant un parcours de l'arbre de recherche en profondeur d'abord.

I.4.3.2.1 Ensemble de succès d'un programme

Définition: L'ensemble des succès d'un programme P est l'ensemble des atomes At pour lesquels il existe un arbre de dérivation SLD qui contient la clause vide. C'est l'ensemble des atomes tels que n'importe quel arbre de dérivation SLD de racine At contient la clause vide.

L'ensemble des succès d'un programme correspond à l'ensemble des succès obtenus par un interpréteur effectuant un parcours équitable de l'arbre de résolution, par exemple en largeur d'abord. En général, les interpréteurs Prolog ne permettent pas d'obtenir tous les succès car ils effectuent un parcours de l'arbre de résolution en profondeur d'abord: la première branche infinie rencontrée empêche l'exploration du reste de l'arbre.

I.4.3.2.2 Ensemble de succès pour un interpréteur "en profondeur d'abord"

Définition: L'ensemble des succès d'un interpréteur Prolog en profondeur d'abord et utilisant une stratégie de dérivation S pour un programme P, noté $SSP_S(P)$ est l'ensemble des atomes pour lesquels cet interpréteur donne un succès. C'est l'ensemble des atomes pour lesquels l'arbre de dérivation SLD via la stratégie S contient la clause vide sans qu'il y ait de branche infinie à gauche de cette clause.

En général, $SSP_S(P)$ est strictement contenu dans SS(P).

L'obtention de tous les succès (SS) est liée à l'utilisation d'un parcours de l'arbre SLD équitable. Cependant, l'obtention ou non de certains succès avec un interpréteur en profondeur d'abord peut dépendre de la stratégie de dérivation utilisée: l'existence ou non d'une branche infinie ainsi que la position d'une branche de succès dans un arbre SLD peuvent être liées au choix de la stratégie de dérivation.

Dans l'exemple ci-dessous, l'ensemble des succès obtenus avec un interpréteur utilisant une stratégie de dérivation "en file" (gestion des buts en file) est strictement inclus dans l'ensemble des succès obtenus avec un interpréteur standard (gestion des buts en pile).

On considère le programme P:

```
but:- p(X).
p(X):- q(X),r(X).
q(X):- s(X).
s(a).
r(b):- r(b).
r(a).
s(b).
```

On obtient les ensembles de succès suivants:

SSP_{pile}={ s(a),r(a),s(b),q(a),q(b),p(a),but}

SSP_{file}={ s(a),r(a),s(b),q(a),q(b),p(a)}

Pour le but `but`, l'arbre de dérivation obtenu avec la stratégie standard contient une clause vide dans la première branche puis une branche infinie; un interpréteur en profondeur d'abord donne donc un succès avant de boucler. Dans l'arbre SLD obtenu en gérant les buts "en file", la branche qui donne un succès se trouve à droite d'une branche infinie; un interpréteur "en file" et en profondeur d'abord boucle sans donner de succès.

Nous nous intéressons maintenant à l'ensemble des atomes pour lesquels il existe un arbre de dérivation fini et contenant la clause vide.

I.4.3.2.3 Ensemble de succès finis d'un programme

Définition: L'ensemble des succès finis d'un interpréteur utilisant une stratégie de dérivation S pour un programme P, noté $SFS_S(P)$ (Set of Finite Success) est l'ensemble des atomes pour lesquels l'arbre de dérivation SLD via S est fini et contient la clause vide.

Cet ensemble correspond à l'ensemble des atomes pour lesquels l'interpréteur donne un succès et s'arrête. On a bien évidemment $SFS_S(P) \subseteq SSP_S(P) \subseteq SS(P)$.

Définition: L'ensemble des succès finis d'un programme P, noté SFS(P), est l'ensemble des atomes At pour lesquels il existe un arbre de dérivation SLD fini et contenant la clause vide.

C'est l'ensemble des atomes tels que tout arbre SLD équitable est fini et contient la clause vide: pour toute stratégie équitable SE, on a $SFS_{SE}(P)=SFS(P)$.

L'utilisation d'une stratégie de dérivation équitable permet donc d'obtenir tous les succès finis d'un programme.

Exemple: Pour le programme P ci-dessous, les différents ensembles de succès obtenus avec un interpréteur standard et avec un interpréteur équitable "en file" sont tous distincts.

Soit le programme P:

```
t:- r(X), échec.
t.
v:-r(X), échec.
v:-s(X), r(X).
p:-q(X), r(X).
q(X):- s(X).
s(a).
r(b):- r(b).
r(a).
s(b).
```

Les différents ensembles de succès obtenus sont les suivants:

• Ensemble des succès du programme P:

SS(P)={s(a), r(a), s(b), q(a), q(b), p, t, v}

• Ensemble des succès trouvés par un interpréteur standard:

SSP_{pile}(P)={s(a), r(a), s(b), q(a), q(b), p}

• Ensemble des succès trouvés par un interpréteur "en file" et en profondeur d'abord:

SSP_{file}(P)={s(a), r(a), s(b), q(a), q(b), t, v}

• Ensemble des succès finis du programme P:

SFS(P)={s(a), r(a), s(b), q(a), q(b), t}

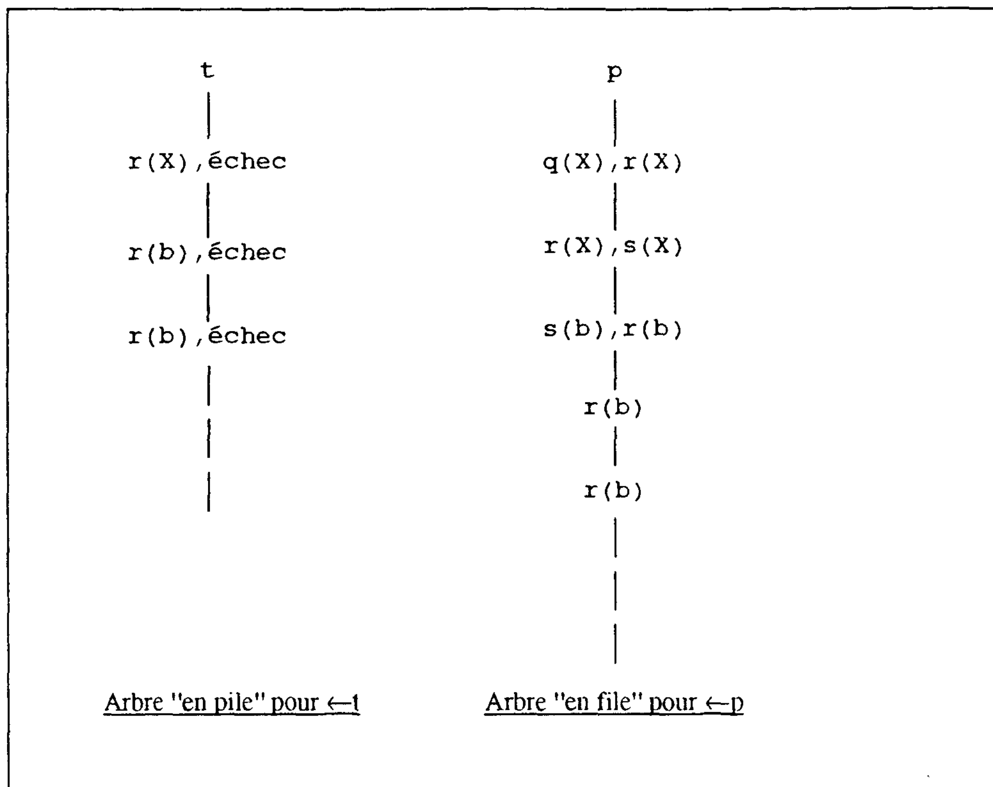
• Ensemble des succès finis pour la stratégie standard:

SFS_{pile}(P)={s(a), r(a), s(b), q(a), q(b)}

Pour cet exemple, toutes les inclusions entre les différents ensembles de succès vraies dans le cas général sont strictes:

- $SFS_{pile}(P) \subseteq SSP_{pile}(P) \subseteq SS(P)$
- $SFS(P) \subseteq SSP_{file}(P) \subseteq SS(P)$
- $SFS_{pile}(P) \subseteq SFS(P) \subseteq SS(P)$

Voici les deux arbres infinis obtenus respectivement pour le but ←t avec la stratégie "en pile" et pour ←t avec la stratégie "en file":



La stratégie "en pile" donne une branche infinie pour ←t, car le sous-but "échec" n'est jamais sélectionné. Un interpréteur en profondeur d'abord et "en file" ne donne pas de succès pour ←p car le parcours de l'arbre n'est pas équitable et il existe une branche infinie à gauche de la clause vide.

I.4.3.3 Boucles liées à la stratégie de dérivation standard

La notion de succès fini permet de caractériser les boucles dues à la stratégie de dérivation: lorsqu'un interpréteur Prolog standard boucle pour un but ←At avec At∈SFS (comme le but ←t dans l'exemple précédent), cette boucle est due à la stratégie de dérivation standard. Un interpréteur standard boucle alors que n'importe quel interpréteur équitable donne un succès sans boucler. C'est le cas en particulier lorsqu'une boucle est liée à l'ordre des sous-buts dans un corps de clause:

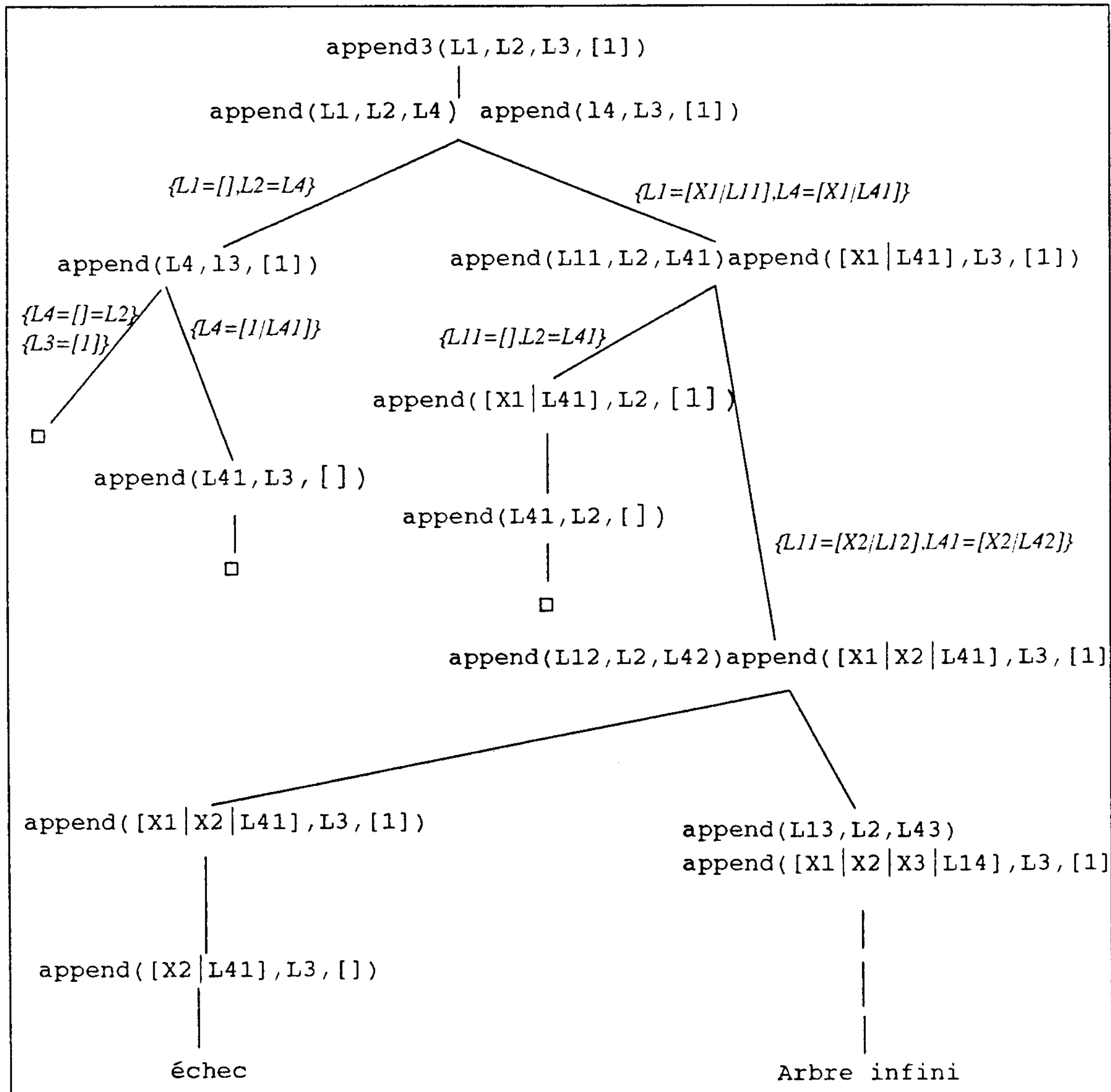
> *Toute boucle en Prolog standard qui peut être évitée par un réordonnancement des buts dans un corps de clause est "naturellement" évitée avec un interpréteur équitable.*

L'arbre de dérivation SLD obtenu avec la stratégie standard après réordonnancement des sous-buts dans un corps de clause est un arbre de dérivation SLD particulier pour le programme initial. Par conséquent, si cet arbre est fini, alors n'importe quel arbre de dérivation SLD équitable est aussi fini:

Avec un interpréteur équitable, la terminaison d'un programme ne dépend pas de l'ordre des sous-buts dans un corps de clause.

Certains prédicats Prolog usuels, par exemple `append3\4`, `permut\2` ou `naive-reverse\2` sont donc réversibles avec un interpréteur équitable:

• Le prédicat *append3\4* (§ I.4.2.3) fonctionne en mode(i,i,i,o) en Prolog standard, mais boucle en mode (o,o,o,i), comme le montre l'arbre de dérivation (Le comportement du *append3\4* est étudié dans [NAI a]). L'arbre de dérivation envisagé par Prolog standard pour append3 en mode (o,o,o,i) comporte une branche infinie dans laquelle un sous-but qui échoue n'est pas sélectionné. Avec une stratégie de dérivation équitable, le prédicat *append3\4* fonctionne aussi bien en mode (i,i,i,o) qu'en mode (o,o,o,i): Avec une stratégie équitable, l'arbre est fini, car le deuxième sous-but échoue dès que la taille de la liste générée est trop importante. La figure suivante montre l'arbre de dérivation obtenu avec la stratégie standard pour le but ←`append3(L1,L2,L3,[1])`:



• Considérons le prédicat `permut` défini comme suit:

```
permut([],[]).
permut(L,[X|LL]):-
     enlever(X,L,L2),
     permut(L2,LL).

enlever(X,[X|L],L).
enlever(X,[Y|L],[Y|LL]):-
     enlever(X,L,LL).
```

Cette version de `permut` fonctionne en mode(i,o) avec un interpréteur standard. En mode (o,i), on obtient une boucle après la première solution. Si l'on inverse les appels de `enlever` et `permut` dans la deuxième clause, il fonctionne alors en mode (o,i) uniquement (En mode (i,o), on obtient une boucle après la dernière solution). Le prédicat `permut` est donc réversible avec une stratégie de dérivation équitable.

Le cas du `permut` est étudié dans [ELC]. Elcock montre qu'il n'est pas possible d'écrire une version de `permut` qui fonctionne dans les deux sens, alors que la relation définie par le prédicat est symétrique. Lorsqu'on utilise une règle d'évaluation équitable, le `permut` est naturellement réversible.

I.5 Résolution SLD avec négation par l'échec

I.5.1 Introduction

Nous nous intéressons dans ce chapitre au traitement de la négation en Prolog. Le problème de la négation a été trés largement étudié et plusieurs approches ont été envisagées. Shepherdson propose une synthèse des différents travaux sur la négation en programmation logique [SHE c].

Nous nous intéressons uniquement à la négation par l'échec (Negation as Failure) pour les programmes avec négation dans les corps de clause (programmes normaux), qui est une forme de négation qui est couramment utilisée en Prolog.

L'introduction de littéraux négatifs dans les corps de clauses fait apparaitre des problèmes théoriques nouveaux: en particulier, la résolution SLDNF n'est pas complête en général et n'est correcte que sous certaines conditions. Ces problèmes, en particlulier la complétude, ont été très largement étudiés ([B, M], [C, L], [CLA a], [KUN], [LLO], [SHE a,b,d]). Nous présentons les principaux résultats concernant la correction et la complétude de la SLDNF. Les résultats de complétude nécessitent des hypothèses assez restrictives.

Nous discutons ensuite la possibilité d'implémenter des interpréteurs qui soient complets pour la SLDNF: Contrairement au cas de la résolution SLD pour les programmes définis, il n'est pas possible de concevoir des interpréteurs pour lesquels ces résultats s'appliquent.

I.5.2 Définitions

I.5.2.1 Programmes normaux

Nous suivons dans tout ce chapitre la terminologie de [LLO].

Les programmes normaux sont des programmes pouvant comporter des littéraux négatifs dans les corps de clauses.

Définition: Une clause normale est une clause de la forme:

$A \leftarrow L_1, L_2, \ldots L_m$, où A est un atome et $L_1, L_2, \ldots L_m$ sont des littéraux.

Définition: Un programme normal une suite finie de clauses normales.

Définition: Un but normal est une clause de la forme $\leftarrow L_1, L_2, \ldots , L_m$ où $L_1, L_2, \ldots ,L_m$ sont des littéraux.

I.5.2.2 Complétion d'un programme

La complétion d'un programme, introduite par Clark [CLAa], permet de donner un sens logique à la résolution SLD avec négation par l'échec. La complétion d'un programme consiste en quelque sorte à compléter la définition de chaque prédicat en traduisant la partie "seulement si". Des faits négatifs peuvent alors être déduits.

Définiton: Soit $p(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_m$ une clause de programme d'un programme normal P.

On réécrit cette clause sous la forme:

$p(x_1, \ldots, x_n) \leftarrow \exists y_1 \ldots \exists y_d((x_1=t_1) \wedge \ldots \wedge (x_n=t_n) \wedge L_1 \wedge \ldots \wedge L_m)$

où $x_1, \ldots, x_n$ sont des variables n'apparaissant pas dans la clause et $y_1, \ldots, y_d$ sont les variables de la clause originelle.

On effectue cette transformation pour chaque clause de la définition de p. On obtient k clauses de la forme: $p(x_1, \ldots, x_n) \leftarrow E_j$

La définition complétée du prédicat p est la formule:

$$\forall x_1 .. \forall x_n, p(x_1, \ldots, x_n) \leftrightarrow E_1 \vee \ldots. \vee E_k$$

Pour les symboles de prédicats n'apparaissant jamais en tête de clause, on ajoute explicitement une clause: $\forall x_1 \ldots \forall x_n; \sim q(x_1, \ldots, x_n)$.

Le symbole "=" est interprété comme l'identité.

La complétion d'un programme P, notée *comp(P)*, est l'ensemble des définitions complétées des symboles de prédicats plus les axiomes de la théorie de l'égalité. Nous définissons maintenant la notion de réponse correcte pour un programme normal par rapport au programme complété.

Définition: Soit P un programme normal, G un but normal $\leftarrow L_1, \ldots L_m$. Une réponse pour $P \cup \{G\}$ est une substitution pour les variables de G.

Définition: Soit P un programme normal, G un but normal et θ une réponse pour $P \cup \{G\}$. On dit que θ est une réponse correcte pour $comp(P) \cup \{G\}$ si $\forall((L_1 \wedge L_2 \wedge \ldots \wedge L_n)\theta)$ est conséquence logique de comp(P).

Cette définition généralise la définition d'une réponse correcte pour un programme et un but définis. Les résultats suivant montrent que dans le cadre des programmes définis, les réponses correctes pour un programme P et les réponses correctes pour la complétion du programme sont les mêmes.

Proposition:

Soit P *un programme normal. Alors* P *est conséquence logique de* comp(P).

Proposition:

Soit P *un programme défini et* $A_1, \ldots, A_m$ *des atomes. Si* $\forall(A_1 \wedge A_2 \wedge \ldots \wedge A_m)$ *est une conséquence logique de* comp(P), *alors elle est aussi conséquence logique de* P.

Ceci montre que les faits positifs qui peuvent être déduits de comp(P) et ceux qui peuvent être déduits de P sont les mêmes.

Théorème:

Soient P *un programme défini,* G *un but défini et* θ *une réponse pour* $P \cup \{G\}$. *Alors* θ *est une réponse correcte pour* $P \cup \{G\}$ *si et seulement si* θ *est une réponse correcte pour* $comp(P) \cup \{G\}$.

Lorsque l'on considère des programmes normaux, la complétion d'un programme peut ne pas être consistante.

Apt, Blair et Walker ont défini une classe de programmes, les programmes stratifiés, pour laquelle la complétion est toujours consistante [A,B, W].

Définition: Une application de niveau d'un programme est une application de l'ensemble de ses symboles de prédicats dans l'ensemble des entiers positifs.

Un programme normal est stratifié s'il possède une application de niveau telle que, dans toute clause du programme $p(t_1, \ldots t_n) \leftarrow L_1, \ldots, L_m$ le niveau des symboles de prédicats de tout littéral positif du corps soit inférieur ou égal au niveau de p et le niveau des symboles de prédicats de tout littéral négatif du corps de la clause soit strictement inférieur au niveau de p.

Proposition: [A,B, W]

Soit P *un programme normal stratifié. Alors* comp(P) *a un modèle de Herbrand minimal.*

I.5.3 Justesse de la Résolution SLDNF

I.5.3.1 Dérivations SLDNF, arbres de dérivation

Nous donnons maintenant les définitions d'une dérivation SLDNF et d'un arbre de dérivation SLDNF. Les définitions adoptées ici sont celles de Lloyd [LLO].

Les problèmes liés à l'implémentation effective de la négation dans un interpréteur Prolog sont discutés ensuite.

Définition: Soient G le but $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ et C la clause $A \leftarrow M_1, \ldots, M_q$.

Alors G dérive de G et C en utilisant un plus grand unificateur (pgu) θ si:

- L_m est un atome, appelé l'atome choisi de G

- θ est un plus grand unificateur de L_m et de A

- G' est le but normal $\leftarrow(L_1, \ldots, L_{m-1}, M_1, \ldots, M_q, L_{m+1}, \ldots L_p)\theta$

Définition: Soient P un programme normal et G un but normal. Une réfutation SLDNF de rang 0 de $P\cup\{G\}$ est une suite $G_0= G, G_1, \ldots, G_n$ de buts normaux, une suite $C_1, C_2, \ldots, C_n$ de variantes de clauses de programme de P et une suite $\theta_1, \theta_2, \ldots, \theta_n$ de pgu tels que G_{i+1} dérive de G_i et de C_{i+1} en utilisant le pgu θ_i.

Définition: Soient P un programme normal et G un but normal. Un arbre SLDNF de rang 0 de $P\cup\{G\}$ est un arbre de racine G tel que:

- L'arbre est fini et chaque nœud de l'arbre est un but normal non vide.

- Seuls des littéraux positifs sont choisis dans un noeud

- Soit $\leftarrow L_1, \ldots, L_m, \ldots, L_q$ un noeud qui n'est pas une feuille. Supposons que L_m soit un atome et soit choisi. Alors pour chaque clause de programme telle que L_m et A soient unifiables, le noeud a un fils correspondant.

- Soit $\leftarrow L_1, \ldots, L_m, \ldots, L_q$ une feuille de l'arbre et L_m un atome qui est choisi. Alors il n'y aucune règle de P dont la tête s'unifie avec L_m.

Définition: Soient P un programme normal et G un but normal. Une réfutation SLDNF de rang k+1 de $P\cup\{G\}$ est une suite $G_0=G, G_1, \ldots, G_n=\square$ de buts normaux, une suite $C_1, \ldots, C_n$ de variantes de clauses de programmes de P ou de littéraux négatifs clos et une suite $\theta_1, \ldots, \theta_n$ de substitutions telles que, pour chaque i, soit:

- G_{i+1} dérive de G_i et C_{i+1} en utilisant θ_{i+1}
- G_i est le but $\leftarrow L_1, \ldots, L_m, \ldots L_p$, le littéral choisi L_m est un littéral négatif clos et il y a un arbre SLDNF d'échecs finis de rang k pour $P\cup\{\leftarrow Am\}$. Dans ce cas G_{i+1} est le but $\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots L_p$, C_{i+1} est le littéral L_m et θ_{i+1} est la substitution identité.

Définition: Soient P un programme normal et G un but normal. Un arbre SLDNF d'échecs finis de rang k+1 pour $P\cup\{G\}$ est un arbre de racine G tel que:

- l'arbre est fini et chaque noeud est un but normal non vide

- la racine de l'arbre est G

- si $\leftarrow L_1, \dots, L_m, \dots L_p$ est un noeud de l'arbre qui n'est pas une feuille et L_m le littéral choisi, alors soit:
  - L_m est un atome et il n'y a pas de clause de P s'unifiant avec L_m
  - L_m est un littéral négatif clos $\sim A_m$ et il existe une réfutation SLDNF de rang k de $P \cup \{\leftarrow A_m\}$, auquel cas l'unique fils du noeud est $\leftarrow L_1, \dots, L_{m-1}, L_{m+1}, \dots, L_p$
- si $\leftarrow L_1, \dots, L_m$ est une feuille de l'arbre et L_m le littéral choisi, alors soit:
  - L_m est un atome et il n'y a pas de clauses de P s'unifiant avec L_m
  - L_m est un littéral négatif clos et il existe une réfutation SLDNF de rang k de $P \cup \{\leftarrow A_m\}$.

<u>Définition:</u> Soient P un programme normal et G un but normal. Une réfutation SLDNF de $P \cup \{G\}$ est une réfutation SLDNF de $P \cup \{G\}$ de rang k pour un certain k.

<u>Définition:</u> Soient P un programme normal et G un but normal. Un arbre SLDNF d'échecs finis pour $P \cup \{G\}$ est un arbre SLDNF d'échecs finis de rang k pour $P \cup \{G\}$, pour un certain k.

<u>Définition:</u> Soient P un programme normal et G un but normal. Une réponse calculée pour $P \cup \{G\}$ est la substitution obtenue en restreignant la composition $\theta_1, \theta_2, \dots, \theta_n$ aux variables de G, où $\theta_1, \theta_2, \dots, \theta_n$ est la suite des substitutions utilisées dans une réfutation SLDNF de $P \cup \{G\}$.

<u>Définition:</u> Soient P un programme normal et G un but normal.

Une dérivation SLDNF de $P \cup \{G\}$ consiste en une suite (finie ou infinie) $G_0 = G, G_1, \dots$ de buts normaux, une suite $C_1, C_2, \dots$ de variantes de clauses du programme P (appelées clauses d'entrée) ou de littéraux négatifs clos et une suite $\theta_1, \theta_2, \dots, \theta_n$ de substitutions telles que:

- pour chaque i, soit:
  - G_{i+1} dérive de G_i et d'une clause d'entrée C_{i+1} en utilisant θ_{i+1}
  - le littéral choisi dans G_i est un littéral négatif clos $\sim A_m$ et il y a un arbre SLDNF d'échecs finis pour $P \cup \{\leftarrow Am\}$. Dans ce cas, G_{i+1} est $\leftarrow L_1, \dots, L_{m-1}, L_{m+1}, \dots, L_p$, θ_{i+1} est la substitution identité et C_{i+1} est $\sim A_m$.
- si la dérivation est finie alors soit:

- le dernier but est vide
- le dernier but est $\leftarrow L_1, \ldots, L_m, \ldots L_p$, L_m est le littéral choisi et est un atome et il n'y a pas de règle de P s'unifiant avec L_m
- le dernier but est $L_1, \ldots, L_m, \ldots L_p$, L_m est le littéral choisi et est un litéral négatif clos et il n'y a pas de réfutation SLDNF de ce littéral

Définition: Soient P un programme normal et G un but normal.

Un arbre SLDNF pour $P \cup \{G\}$ est un arbre de racine G vérifiant les conditions suivantes:

- Chaque noeud de l'arbre est un but normal
- Soit $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ un noeud de l'arbre qui ne soit pas une feuille et supposons que L_m soit choisi. Alors soit:
  - L_m est un atome et pour chaque clause de programme dont la tête est unifiable avec L_m, le noeud a un fils correspondant
  - L_m est un littéral négatif clos et il existe un arbre de dérivation SLDNF d'échecs finis pour $P \cup \{A_m\}$, auquel cas le seul fils est $\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_p$
- Soit $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ une feuille de l'arbre et supposons que Lm soit choisi. Alors soit:
  - L_m est un atome et il n'y a pas de clause de P dont la tête s'unifie avec L_m
  - L_m est un littéral négatif clos et il existe une réfutation SLDNF de $P \cup \{\leftarrow A_m\}$

Ces définitions généralisent les définitions de dérivation SLD et d'arbres SLD. Une dérivation SLDNF peut être finie ou infinie. Une dérivation SLDNF finie échoue ou réussit. Une dérivation SLDNF qui réussit est une réfutation. Cependant, contrairement à ce qui se passe pour les programmes définis, il se peut qu'il n'existe pas de dérivation pour un but donné: si un littéral négatif clos $\sim A_m$ est tel que il n'existe pas d'arbre SLDNF d'échecs finis pour $P \cup \{\leftarrow A_m\}$ et il n'existe pas de réfutation de $P \cup \{\leftarrow A_m\}$, alors il ne peut exister de dérivations SLDNF dans laquelle $\sim A_m$ est sélectionné.

Exemple: On considère le programme suivant:

```
p :- ~q
q :- q
```

Il n'existe pas de dérivation SLDNF pour $P \cup \{\leftarrow q\}$. Par contre, il existe une dérivation SLDNF pour le but normal $\leftarrow q,r$, car r peut être sélectionné en premier et donne un échec. En fait, cette définition occulte la réalité: lorsqu'un littéral négatif clos ne peut être traité, c'est à dire lorsque le littéral positif correspondant ne donne ni un succès ni un échec fini, il n'existe pas de dérivation SLDNF. Aussi, l'existence ou non d'une dérivation SLDNF est

indécidable. C'est ce qu'a montré Shepherdson en prouvant qu'il n'existe pas de règles d'évaluation maximales récursives ([SHE d]).

La définition d'une dérivation SLDNF interdit qu'un littéral négatif non clos soit sélectionné. Cette condition, appelée la condition de sureté, est une condition suffisante pour éviter des réponses incorrectes. Considérons par exemple le programme normal suivant:

```
a(X) :- ~b(X), c(X).
b(a).
c(b).
```

Pour le but ←`a(X)`, si le sous-but `b(X)` est sélectionné en premier, on obtient un échec car `b(a)` donne un succès. Par contre, si la règle d'évaluation est sure, `c(X)` est sélectionné d'abord et on obtient un succès pour `X=b`. La condition de sureté peut cependant être affaiblie: un littéral négatif non clos peut être sélectionné à condition que sa dérivation ne provoque aucune instanciation.

Il est possible qu'un but ne contienne plus que des littéraux négatifs non clos. Dans ce cas, aucun littéral négatif ne peut être choisi. On dit alors que le calcul s'enlise. L'enlisement peut être évité en imposant certaines restrictions.

Définition: Soient P un programme normal et G un but normal. Un calcul de $P \cup \{G\}$ s'enlise si on atteint un but qui ne contient que des littéraux négatifs non clos.

La définition des programmes autorisés permet d'obtenir une condition suffisante de non enlisement:

Définition: Soient P un programme normal et G un but normal. Une clause normale est admissible si chaque variable qui apparait dans la clause apparait soit dans la tête A soit dans un littéral positif du corps. Une clause est autorisée si chaque variable apparaissant dans la clause apparait dans un littéral positif du corps. Un but normal G est autorisé si chaque variable apparaissant dans G apparait dans un littéral positif.

$P \cup \{G\}$ est autorisé si:

- G est autorisé
- toute clause de P est admissible
- toute clause de la définition d'un symbole de prédicat apparaissant dans un littéral positif du corps de G ou dans un littéral positif du corps d'une clause est autorisée.

Proposition ([LLO]):

Soient P *un programme normal et* G *un but normal tels que* P∪{G} *soit autorisé. Alors:*

- *aucun calcul de* P∪{G} *ne s'enlise*
- *toute réponse calculée pour* P∪{G} *est une substitution close pour toutes les variables de G.*

I.5.3.2 Justesse de la résolution SLDNF

Clark a montré la justesse de la règle de négation par l'échec pour les programmes normaux, ainsi que la justesse de la résolution SLDNF ([CLA a]).

Théorème:

Soient P *un programme normal et* G *un but normal. Si* P∪{G} *a un arbre SLDNF d'échecs finis, alors* G *est une conséquence logique de* comp(P).

Théorème:

Soient P un programme normal et G *un but normal. Alors toute réponse calculée pour* P∪{G} *est une réponse correcte pour* P∪{G}.

Ces résultats montrent que la résolution SLDNF est correcte par rapport à la complétion d'un programme normal. Par contre, elle n'est pas complète en général. Des résultats de complétude ont néanmoins été obtenus pour certaines classes de programmes.

I.5.4 Complétude de la SLDNF

I.5.4.1 Résultats de complétude

La résolution SLDNF n'est pas complète en général. Ceci vient en particulier du fait qu'il n'est pas toujours possible de construire une dérivation SLDNF. Considérons par exemple le programme suivant [LLO]:

```
r :- p.
r :- ~p.
p :- p.
```

Dans ce cas, r est conséquence logique de comp(P), mais il n'existe pas d'arbre de dérivation SLDNF pour P∪{←r}.

Il est donc nécessaire d'imposer des restrictions supplémentaires pour pouvoir obtenir des résultats de complétude. Plusieurs résultats de complétude ont été obtenus, sous des

hypothèses plus ou moins fortes ([B, M], [APT a],[C, L]). Kunnen [KUN] a montré un résultat de complétude dans le cadre de la logique trivaluée. Nous reprenons ici le résultat de Lloyd et Cavedon [C, L], qui montre la complétude de la SLDNF et de la résolution par l'échec pour des programmes normaux stratifiés et des buts stricts. Ce résultat prouve une conjecture de Apt, Blair et Walker [A,B,W].

Nous discutons ensuite de la possibilité de définir des procédures de résolution pour la SLDNF qui soient complètes.

Définition: Soient P un programme normal et G un but normal. On définit les ensembles POS et NEG de la façon suivante:

POS^0={p: p est le symbole de prédicat d'un littéral positif de G}

NEG^0={q: q est le symbole de prédicat d'un littéral négatif de G}

Pour i>0:

POS^i={p: il existe une clause $r(\tilde{t})$:-L_1, ... ,$p(\tilde{s})$, ... ,L_k avec $r \in POS^{i-1}$

ou il existe une clause $r(\tilde{t})$:-L_1, ... ,$\sim p(\tilde{s})$, ... ,L_k avec $r \in NEG^{i-1}$

NEG^i={q: il existe une clause $r(\tilde{t})$:-L_1, ... ,$q(\tilde{s})$, ... ,L_k avec $r \in NEG^{i-1}$

ou il existe une clause $r(\tilde{t})$:-L_1, ... ,$\sim p(\tilde{s})$, ... ,L_k avec $r \in POS^{i-1}$

$POS=\cup POS^i$, $NEG=\cup NEG^i$

On dit que $P\cup\{G\}$ est strict si $POS\cap NEG=\varnothing$

Théorème:

Soit P *un programme autorisé, stratifié, normal et* G *un but normal tel que* $P\cup\{G\}$ *soit strict. Si* θ *est une réponse correcte pour* $comp(P)\cup\{G\}$, *alors* θ *est une réponse calculée pour* $P\cup\{G\}$.

Le deuxième résultat montre la complétude de la négation par l'échec pour la SLDNF. Les arbres de dérivation SLDNF envisagés doivent être équitables. Nous donnons tout d'abord la définition des arbres de dérivation SLDNF équitables (au sens de Lloyd et Cavedon).

Définitions: Soit P un programme normal et G un but normal. Un littéral apparaissant dans un arbre SLDNF pour $P\cup\{G\}$ est d'échec potentiel si c'est un littéral positif ou bien un littéral négatif clos ~A tel que A est conséquence logique de comp(P).

Une branche d'un arbre SLDNF est équitable si c'est une branche d'échec ou bien si, pour tout littéral d'échec potentiel B, B (ou une version instanciée de B) est sélectionné au bout d'un nombre fini d'étapes.

Un arbre de dérivation SLDNF est équitable si toute branche de cet arbre est une branche équitable.

Théorème:

> *Soit* P *un programme autorisé, stratifié, normal et* G *un but normal tel que* P∪{G} *est strict. Si* G *est conséquence logique de* comp(P), *alors il existe un arbre SLDNF pour* P∪{G} *et tout arbre SLDNF équitable pour* P∪{G} *est d'échec fini.*

Comme dans le cas des programmes définis, ce résultat n'est vrai que pour des arbres SLDNF équitables. Malheureusement, pour la résolution SLDNF, une règle de sélection équitable des littéraux ne permet pas toujours d'obtenir un arbre de dérivation SLDNF équitable. Soit R une règle de sélection équitable. Il se peut qu'il n'existe pas de dérivation SLDNF via R pour P∪{G} alors qu'il existe un arbre SLDNF d'échecs finis pour P∪{G}.

Soit P le programme normal:

```
p(X) :- ~q(X), r(X).
r(b).
q(a ) :- q(a).
```

Pour le but ←p(a), le nouveau but obtenu après la première dérivation est ←~q(a), r(a). Si r(a) est le littéral sélectionné, on obtient un échec. Par contre, si ~q(a) est choisi, le calcul aboutit à une impasse. Cet exemple très simple montre qu'une règle d'évaluation équitable ne suffit pas pour obtenir tous les arbres d'échecs finis. En réalité il n'existe pas de règle d'évaluation permettant d'obtenir tous les échecs ou tous les succès ([SHE d]). C'est ce que prouve un résultat de Shepherdson, que nous détaillons ci-après.

I.5.4.2 Règles maximales récursives pour la SLDNF

Dans le cas des programmes définis, il est possible d'envisager un interpréteur "idéal" qui soit complet à la fois pour les succès et pour les échecs finis: une règle d'évaluation équitable donne la complétude pour les échecs finis et un parcours de l'arbre SLD équitable (par exemple en largeur d'abord) permet d'obtenir la complétude pour les succès. Nous étudions maintenant la possibilité de construire des interpréteurs "idéaux", c'est à dire tels que les résultats de complétude s'appliquent, pour la résolution SLDNF. L'exemple ci-dessus montre que l'utilisation d'une règle d'évaluation équitable n'est pas suffisante.

Shepherdson a montré qu'il n'existe pas de règles maximales récursives pour la SLDNF [SHE d] (une règle d'évaluation récursive est une règle d'évaluation au sens où nous

l'avons défini dans le paragraphe I.2.6.1). Une règle d'évaluation Rm est maximale pour les succès si, pour tout but pour lequel il existe une règle d'évaluation sure qui donne un succès, Rm donne un succès. De même, une règle Rm est maximale pour les échecs si pour tout but pour lequel il existe un arbre d'échecs finis, l'arbre SLDNF via Rm est aussi un arbre d'échecs finis.

Shepherdson ne considère que des règles d'évaluation donnant des arbres de dérivation SLDNF au sens de Lloyd, c'est-à-dire vérifiant les conditions suivantes:

- les littéraux négatifs sélectionnés sont clos (la règle est sure)
- lorsqu'un littéral négatif clos ~A est sélectionné, on essaye de terminer la construction d'un arbre d'échecs finis de racine ←A avant de traiter les autres littéraux

Shepherdson a démontré que, étant données ces conditions, il n'existe pas de règles d'évaluation maximales récursives pour la SLDNF, ce qui prouve qu'on ne peut construire un interpréteur complet pour la SLDNF en respectant ces conditions.

Cependant, comme le fait remarquer Shepherdson, la non existence de telles règles provient de la deuxième condition. Cette condition, qui est conforme à la définition d'une dérivation SLDNF de Lloyd [LLO], correspond à la manière dont la négation est traitée habituellement dans les interpréteurs Prolog. Cette condition peut être supprimée, et dans ce cas il existe des règles d'évaluation maximales [SHE d].

On peut envisager (comme le suggère Shepherdson) une règle d'évaluation développant alternativement la dérivation principale et les dérivations subsidiaires provenant des littéraux négatifs clos, et développant équitablement chaque dérivation. Une telle règle d'évaluation est proposée dans le §III.5.2.

Malheureusement, ceci n'est pas suffisant pour obtenir tous les échecs finis. En effet, lorsque la négation par l'échec est utilisée, les notions de succès et d'échec sont dépendantes.

Considérons le programme:

```
p :- p.
p.
```

Pour le but ← ~p, un interpréteur qui parcourt l'arbre de dérivation en profondeur d'abord boucle car il ne trouve pas de succès pour ←p. Pour obtenir tous les échecs, il faut utiliser une procédure de réfutation qui donne tous les succès lorsqu'un littéral négatif est dérivé.

Trois conditions sont donc nécessaires pour obtenir un interpréteur complet pour les échecs finis:

- Utiliser une règle d'évaluation équitable
- Développer alternativement la dérivation principale et les dérivations subsidiaires, et ce de manière équitable
- Utiliser un parcours de l'arbre de dérivation équitable pour les dérivations subsidiaires.

La combinaison de ces trois conditions fait qu'on ne peut implémenter de manière simple (par exemple à partir d'un interpréteur en profondeur d'abord) et complète la négation par l'échec pour les programmes normaux. Le traitement de la négation dans les interpréteurs standards ne respecte aucune des trois conditions, l'implémentation que nous avons réalisée avec la stratégie ABI ne vérifie que les deux premières.

Contrairement aux résultats obtenus pour la résolution SLD appliquée aux programmes définis, les résultats de complétude pour la SLDNF et les programmes normaux ne correspondent pas à quelque chose de facilement implémentable, ce qui limite leur intérêt pratique.

Chapitre II :

Définition de stratégies de dérivation équitables

II.1 Introduction

Nous avons montré dans le chapitre précédent l'intérêt des arbres de dérivation équitables. Nous présentons maintenant des exemples d'interpréteurs équitables. Nous décrivons tout d'abord la règle d'évaluation équitable obtenue en gérant les sous-buts "en file". Cette règle d'évaluation, qui est généralement citée comme exemple de stratégie équitable, est très inadaptée à la programmation en Prolog et n'est pas raisonnablement utilisable en pratique. Malgré l'importance de la notion d'équité dans les travaux sur la négation et la sémantique de Prolog, aucune autre règle équitable n'a été effectivement envisagée.

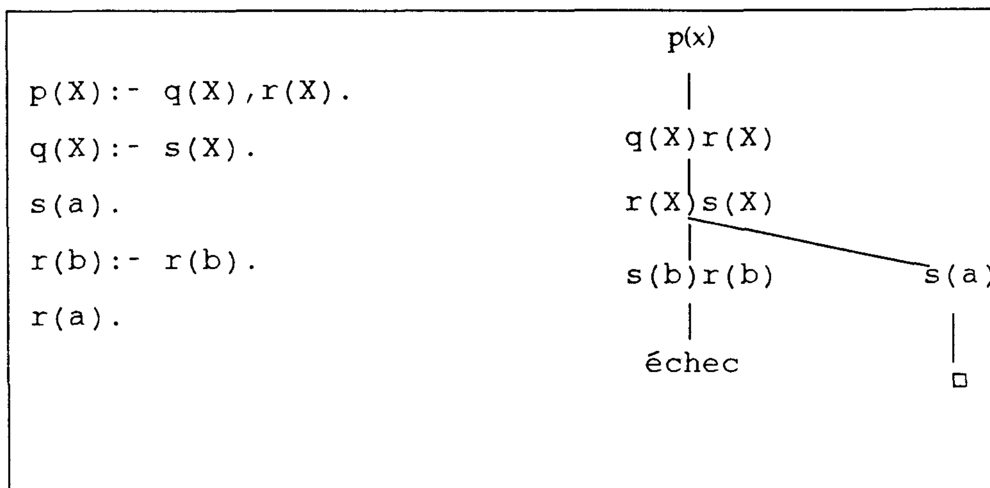
Nous proposons deux stratégies de dérivation équitables originales, basées sur l'utilisation d'indices de dérivation pour contrôler le choix des sous-buts sélectionnés. Nous rappelons que nous entendons par interpréteurs équitables des interpréteurs Prolog effectuant un parcours de l'arbre de recherche en profondeur d'abord (qui n'est donc pas équitable en ce sens que certains noeuds peuvent ne jamais être parcourus) et utilisant une règle de choix de l'atome à sélectionner équitable.

II.2 Stratégie de dérivation en file

La gestion des buts "en file" est le moyen le plus simple d'obtenir des dérivations équitables. L'atome sélectionné est le premier élément de la résolvante, les éventuels nouveaux buts obtenus sont ajoutés en queue de la liste et non en tête: la résolvante, c'est à dire la liste des sous-buts à prouver est gérée comme une file. Ceci garantit qu'aucun sous-but ne peut être laissé indéfiniment non sélectionné dans une dérivation infinie.

Cette règle d'évaluation est parfois appelée stratégie "en largeur" ([NAI a]): La liste de sous-buts est parcourue en largeur d'abord. Nous préférons parler de stratégie en file, ce qui évite toute confusion avec la stratégie de parcours de l'arbre de résolution.

<u>Exemple:</u>

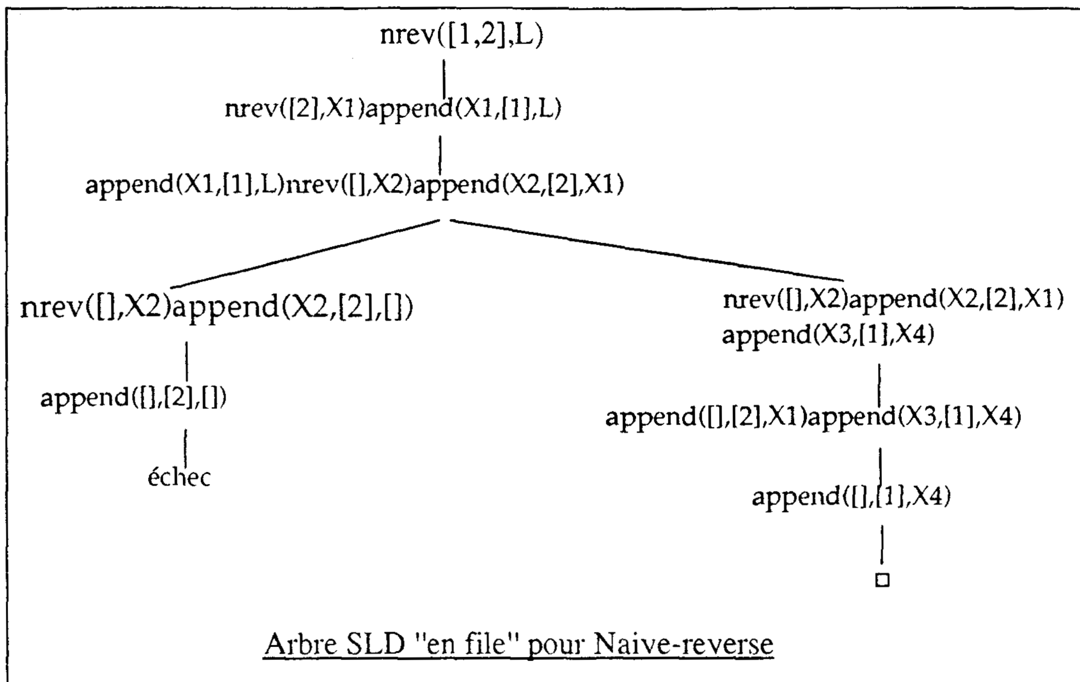


L'augmentation de complexité de l'arbre de résolution occasionnée par l'emploi d'une stratégie non standard peut être importante (cf chapitre IV). Les branches de succès sont les mêmes quel que soit l'arbre de dérivation envisagé [LLO], mais une règle d'évaluation inadaptée peut engendrer de nombreuses branches d'échec supplémentaires. Considérons par exemple le cas du naïve-reverse:

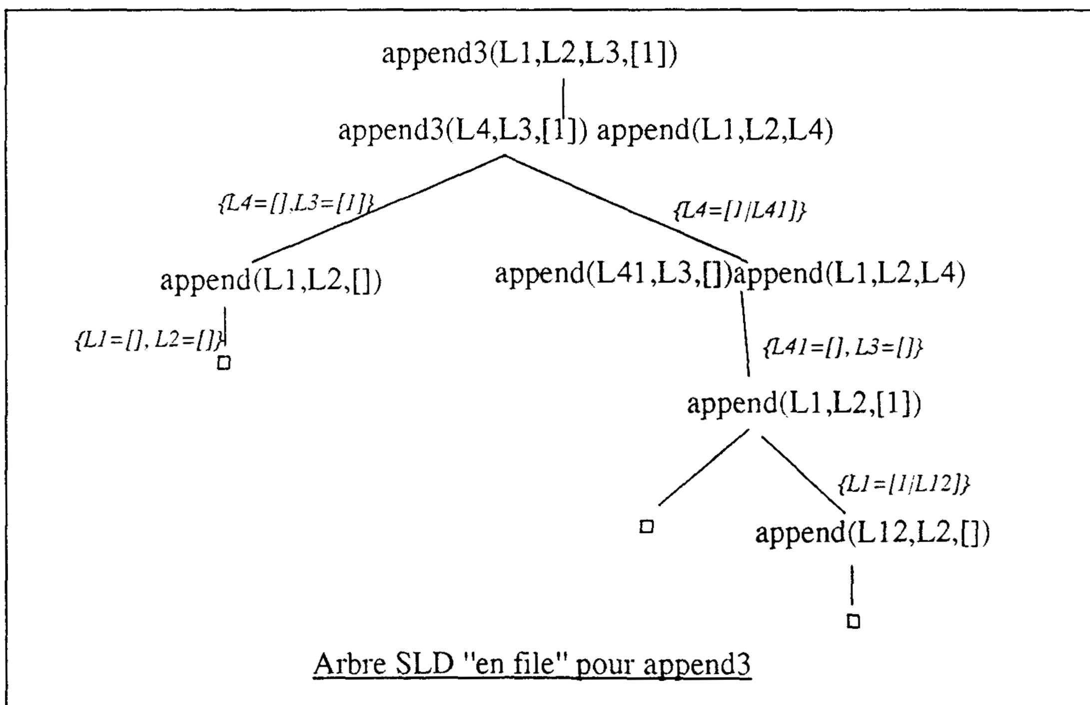
```
append([],L,L).
append([X|L1],L2,[X|L3):-
        append(L1,L2,L3).
nrev([],[]).
nrev([X|L1],L):-
        nrev(L1,L2),append(L2,[X],L).
```

Dans l'arbre de dérivation SLD obtenu pour `nrev([1,2],L)` avec la stratégie en file, le sous-but `append(X1,[1],L)` sélectionné à la troisième étape n'est pas suffisamment instancié et n'est pas déterministe, ce qui a pour effet de créer une branche d'échec supplémentaire. Pour ce programme, le nombre de dérivations nécessaires pour inverser une liste de taille n est en n^4 avec la stratégie "en file" au lieu de n^2 avec la stratégie standard. Par contre, avec un interpréteur équitable, le naïve-reverse est réversible. Pour inverser une liste à n éléments en mode (o,i) avec la stratégie en file, le nombre d'unifications est proportionnel à n^3. Dans certains cas, il se peut que la stratégie "en file" soit efficace. Pour le prédicat `append3/4` (défini §I.4.2.3) utilisé en mode (o,o,o,i), l'arbre de dérivation obtenu avec la stratégie en file ne comporte que des branches de succès, alors que l'arbre standard est infini.

Cette stratégie est très simple à mettre en oeuvre. C'est la raison pour laquelle elle est parfois citée comme exemple de stratégie équitable ([LLO], [NAI a]). Cependant, le rejet systématique des nouveaux buts en queue de la résolvante la rend inadaptée en pratique dans de nombreux cas, entre autres lorsque le programme comporte une récursivité à gauche. Cette méthode empêche tout contrôle de l'ordre d'exécution des buts. Les stratégies équitables que nous proposons maintenant permettent de contrôler la dérivation à l'aide d'indices.



Arbre SLD "en file" pour Naive-reverse



Arbre SLD "en file" pour append3

II.3 Stratégie de dérivation "Pile-Indicée"

II.3.1 Introduction

Nous présentons maintenant deux règles d'évaluation plus souples. Ces stratégies de dérivation sont définies à l'aide d'indices: des indices de dérivation sont introduits dans les programmes. Nous parlerons donc de programmes définis indicés.

La première de ces stratégies est la stratégie *Pile-Indicée*. Cette règle d'évaluation consiste à mixer la stratégie standard "en pile" (efficace) et la stratégie "en file" (équitable), ce mixage étant contrôlé par des indices de dérivation. L'équité est obtenue en ajoutant de temps en temps les nouveaux sous-buts en queue de la liste.

La résolvante est constituée d'une liste d'atomes indicés: un indice de dérivation est associé à chaque atome. Cet indice sert à limiter le nombre de dérivations pouvant être consacrées à ce sous-but avant d'examiner les autres sous-buts.

L'atome sélectionné est toujours le premier atome de la liste. Si son indice de dérivation est égal à 1, les nouveaux sous-buts sont placés en queue de la résolvante. Sinon, les nouveaux sous-buts sont ajoutés en tête de la résolvante, mais avec des indices strictement inférieurs à celui du but père.

II.3.2 Définitions

<u>Définition:</u> Une *clause définie indicée* est une clause définie dans laquelle chaque atome en partie droite est associé à un indice.

Cet indice est soit un entier strictement positif, soit une variable Prolog dont ce n'est pas la première occurrence.

<u>Définition:</u> Un *programme défini indicé* est une suite finie de clauses définies indicées.

<u>Exemple:</u>

```
pp(X):- qq(X)[5], rr(X)[5].
bb(R,N):- qq(X,N)[N], rr(X)[1].
```

Dans la deuxième règle, l'indice de `qq(X)` est une variable Prolog. La valeur de cet indice sera déterminée par la valeur d'instanciation de la variable.

II.3.3 La règle d'évaluation pile-indicée

Soient P un programme indicé et $L=b_1[n_1], b_2[n_2], \dots, b_p[n_p]$ une liste d'atomes indicés. Une étape de dérivation utilisant la règle d'évaluation "pile-indicée" est décrite comme suit:

- Sélectionner une clause de P dont la tête s'unifie avec b_1

Soit a:- $a_1[i_1]$, $a_2[i_2]$, ... ,$a_k[i_k]$ cette clause et θ un pgu de a et b_1. Soit L' la nouvelle résolvante.

Alors si $n_1>1$,

$$L'=(a_1[j_1], a_2[j_2], ... ,a_k[j_k],b_2[n_2], ... , b_p[n_p])\theta; \text{ avec } j_m=\min(n_1-1,i_m);$$

sinon $L'=(b_2[n_2], ... , b_p[n_p],a_1[j_1], a_2[j_2], ... ,a_k[j_k])\theta$

La liste de buts indicés L' dérive de L par la règle d'évaluation pile-indicée (PI).

Lorsque l'indice de l'atome sélectionné vaut 1, les nouveaux buts sont ajoutés en tête de la liste avec des indices strictement inférieurs, sinon ils sont ajoutés en queue de la liste. Ceci garantit qu'on ne peut ajouter un nombre infini de sous-buts devant un sous-but de la résolvante, et donc que la stratégie est équitable. Un indice de dérivation peut être une variable Prolog. La valeur de l'indice est alors fixée dynamiquement par l'instanciation de cette variable. Bien entendu, dans ce cas la variable doit être instanciée avant que l'atome soit sélectionné et sa valeur doit être un entier positif.

L'arbre de dérivation SLD-PI pour un programme Pi et un but indicé ←a[i] est défini de la manière suivante:

- la racine de l'arbre est ←a[i]
- chaque noeud de l'arbre est une liste de buts indicés L telle que:
  - Si L' est un fils de L, alors L' dérive de L par la règle d'évaluation PI
  - Pour toute règle de Pi permettant de dériver L, le noeud L a un fils correspondant. Les différents fils sont ordonnés suivant l'ordre des règles utilisées pour les obtenir. La liste vide est représentée par la clause vide et lorsqu'aucune règle ne permet de dériver la liste L, L a un fils "échec".

L' arbre de dérivation SLD associé à un arbre de dérivation SLD-PI Ti est l'arbre T obtenu en supprimant tous les indices dans Ti.

II.3.4 Equité de la règle d'évaluation pile-indicée

Nous démontrons maintenant l'équité de la stratégie pile-indicée. Nous montrons pour cela que tous les arbres de dérivation envisagés sont équitables.

Théorème:

Soit P *un programme défini indicé,* G *un but indicé. Tout arbre de dérivation SLD associé à un arbre de dérivation* SLD-PI *pour* Pi *et* G *est équitable.*

Preuve: On suppose qu'il existe une dérivation infinie de ←G.

Soit L une liste de buts indicés apparaissant dans cette dérivation.

Soit B un des sous-buts de L, d'indice i. On montre que B est sélectionné au bout d'un nombre fini d'étapes si la dérivation n'échoue pas.

On montre pour cela que le nombre de sous-buts pouvant être ajoutés devant B est fini.

Soit n le nombre maximum d'atomes en partie droite d'une clause de P.

On a $L=a_1[i_1]a_2[i_2] \ldots a_p[i_p]b[i]c_1[j_1]c_2[j_2] \ldots c_k[j_k]$

La dérivation de l'atome a_1 peut donner au maximum:

- n sous-buts d'indice inférieur ou égal à i_1-1

- n^2 sous-buts d'indice inférieurs ou égal à i_1-2

....

- n^{i1-1} sous-buts d'indice 1

Les éventuels sous-buts obtenus à partir des buts d'indice 1 ne sont pas ajoutés en tête, le nombre de sous-buts issus de a_1 susceptibles d'être placés devant B est donc inférieur à:

$n+n^2+ \ldots +n^{i1-1}$; le sous-but a_2 sera donc en tête après au plus $(1+n+ \ldots +n^{i1-1})$ dérivations.

Soit $m=\max(i_1, i_2, \ldots, i_p)$.

Le but b est en tête de la liste après au plus $p(1+n+ \ldots +n^{m-1})=p\left(\frac{1+n^m}{1+n}\right)$étapes de dérivation.

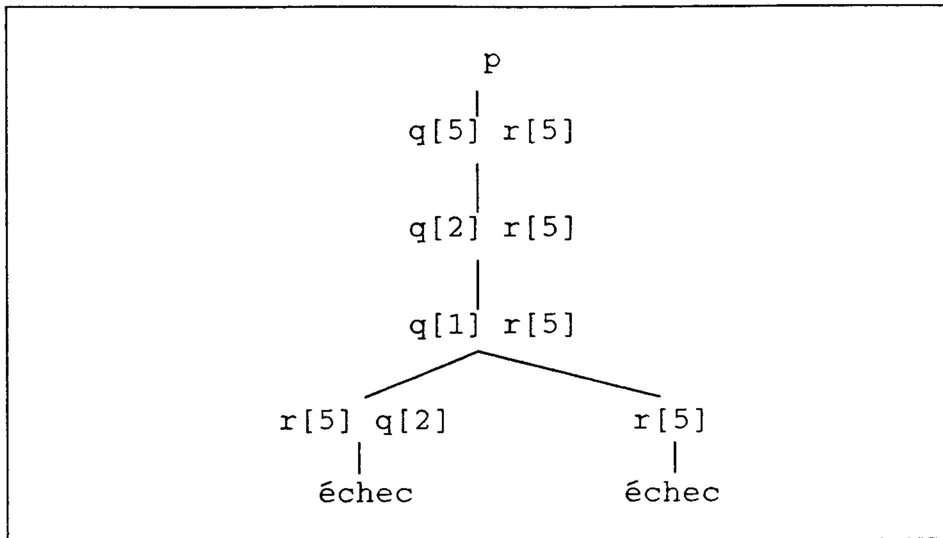
II.3.5 Exemples

Exemple 1:

On considère le programme indicé P_i :

```
p :- q[5], r[5].
q :- q[2].
q.
```

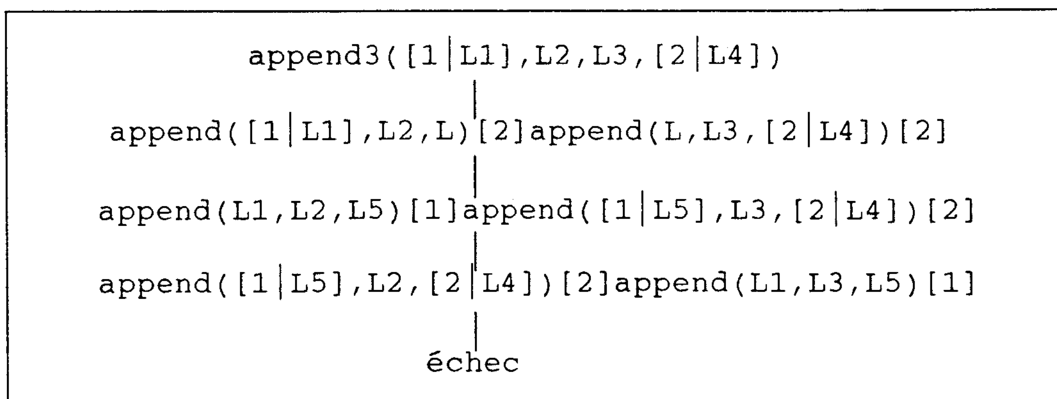
La figure suivante représente l'arbre de dérivation SLD-PI obtenu pour le but ←p. Après deux utilisations de la deuxième clause, l'indice du sous-but q est égal à 1. Le nouveau sous-but q est donc ajouté en queue de la résolvante. r est sélectionné et la dérivation échoue.



Exemple 2: On considère une version indicée du programme `append3`:

```
append([],L,L).
append([X|L1],L2,[X|L3]):- append(L1,L2,L3)[10].
append3(L1,L2,L3,L4):-
        append(L1,L2,L)[2],append(L,L3,L4)[2].
```

Pour le but ←`append3([1|L1],L2,L3,[2|L4])`, on obtient l'arbre de dérivation suivant:



II.3.6 Conclusion

Cette règle d'évaluation correspond à un mixage de la stratégie standard et de la stratégie "en file". Si tous les indices sont égaux à un certain *n*, la stratégie pile-indicée et la stratégie standard coïncident pour des dérivations de longueur inférieure à *n*. A l'opposé, si tous les indices sont égaux à 1, on obtient une gestion des buts "en file".

Une des utilisations possibles de cette stratégie consiste à affecter à chaque but un indice relativement élevé. Dans ce cas, les indices servent de "garde-fou" lorsqu'un des sous-buts engendre une branche infinie. Ainsi, l'équité est garantie sans pour autant affecter de manière systématique la sémantique procédurale habituelle de Prolog. Cette règle d'évaluation permet ainsi de rendre équitable la règle standard de manière simple et efficace.

Cependant, la stratégie Pile-Indicée ne permet pas de définir des mécanismes de contrôle facilement, car l'indice d'un sous-but ne correspond pas exactement au nombre de dérivations consécutives consacrées à ce sous-but. En particulier, il n'est pas possible d'effectuer un traitement alterné de deux sous-buts (cf chapitre V).

II.4 Stratégie de dérivation "Arbre de Buts Indicés"

II.4.1 Introduction

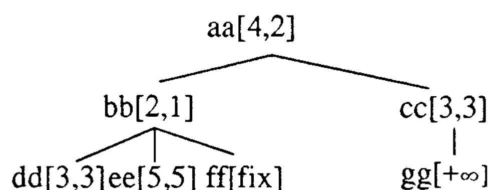
La stratégie de dérivation que nous présentons ici est une extension de celle introduite par Delahaye et Paradinas [D, P]. Cette stratégie est également basée sur l'utilisation d'indices de dérivation et utilise une structure d'arbre pour représenter la résolvante. A chaque sous-but est associé un indice qui représente le nombre maximum de dérivations auquel a droit ce sous-but avant de passer derrière ses but frères. Pour cela, les listes de buts sont remplacées par des arbres de buts indicés dont on fait "tourner" les sous-arbres. Certains sous-buts peuvent être suivis de sous-buts indicés par l'indice spécial "fix", ce qui signifie que ces sous-buts leur sont liés: Un sous-but lié est traité dès que le but auquel il est lié s'efface. Cet indice spécial permet en particulier d'assurer un traitement correct des prédicats de sortie. Un sous-but peut également être indicé par une variable Prolog. Dans ce cas, sa valeur est fixée dynamiquement par instanciation de la variable.

Cette stratégie nécessite l'utilisation d'une structure de données plus complexe. En contrepartie, elle est plus souple et plus simple à utiliser.

II.4.2 Définitions

Définition: Un arbre de buts indicés est un arbre fini dont chaque noeud est un atome associé soit à un couple (indice fixe, indice modifiable), soit à l'indice $+\infty$, soit à l'indice spécial "fix", soit à une variable Prolog.

Exemple:



Définition: Une *clause définie indicée* est une clause définie dont le corps est

- -soit vide
- -soit un seul atome indicé par $+\infty$
- -soit un atome indicé par un entier non nul ou une variable Prolog, suivi d'une liste d'atomes indicés par un entier non nul, une variable Prolog ou l'indice spécial "fix".

Définition: Un programme défini indicé est une suite finie de clauses définies indicées.

Exemple:

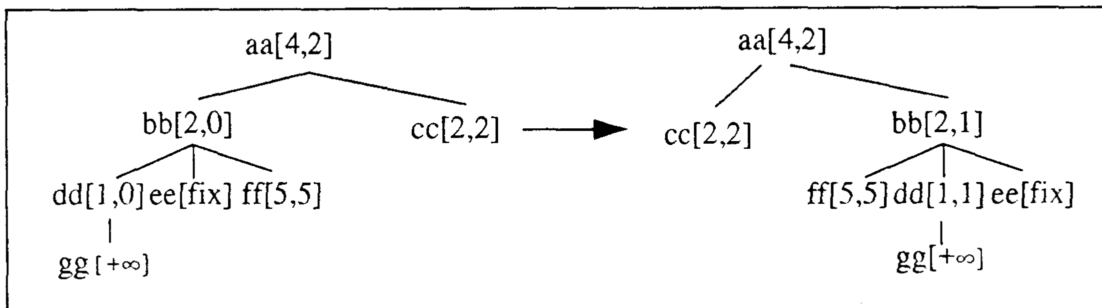
```
aa :- bb[1],cc[fix],ee[2].
bb :- dd[+∞].
```

L'indice spécial "fix" est utilisé pour assurer un traitement séquentiel de certains prédicats, en particulier les prédicats système. Un sous-but indicé par "fix" est traité dès que le sous-but qui le précède dans la clause est effacé. Un sous-but indicé par "fix" doit échouer ou donner un succès fini, sinon la règle d'évaluation n'est plus équitable. Lorsqu'un indice de dérivation est une variable Prolog, cette variable doit être instanciée au moment où le sous-but est effectivement sélectionné.

II.4.3 Mise à jour d'un arbre de buts indicés

L'opération de mise à jour d'un arbre de buts indicés est le mécanisme qui garantit l'équité. Chaque atome sur la branche gauche de l'arbre dont l'indice modifiable est égal à 0 est placé derrière ses buts frères et la valeur de son indice modifiable est réinitialisée à la valeur de son indice fixe. Lorsqu'un tel atome est suivi d'un atome indicé par le symbole "fix", cet atome est également permuté.

Exemple:



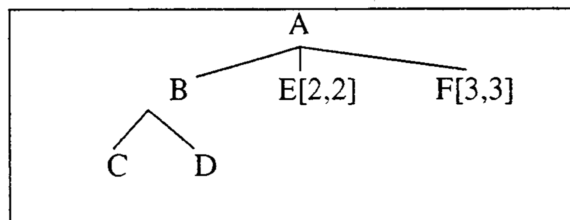
II.4.4 Programmes partiellement indicés

Selon la définition d'un programme indicé, chaque atome doit avoir un indice. L'indice d'un sous-but dans un corps de clause comportant plusieurs sous-but doit être un entier. En pratique il est possible de ne pas mettre d'indice à certains sous-buts sans remettre en cause l'équité.

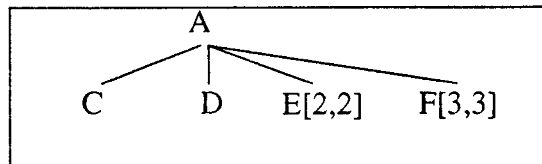
L'indice d'un sous-but sert à limiter le nombre de dérivations consécutives de ce sous-but par rapport à ses sous-buts frères. Un sous-but qui ne peut pas engendrer de dérivation infinie peut ne pas être indicé. On peut donc limiter l'usage des indices aux sous-buts récursifs.

Un sous-but qui n'a pas de frères n'a pas besoin d'indice. De même, un sous-but qui n'a plus de sous-buts frères au moment où il est sélectionné n'a pas besoin d'indice de dérivation. En particulier, tout dernier sous-but d'un corps de clause qui n'est précédé que de sous-buts qui ne sont pas indicés n'a pas besoin d'être indicé.

L'application systématique de ces règles permet de limiter le nombre d'indices de dérivation dans un programme tout en préservant l'équité. Ceci rend les programmes plus lisibles et, de plus, peut permettre une représentation des A.B.I simplifiée: lorsqu'un sous-but qui n'a pas d'indice est sélectionné, si les nouveaux sous-buts obtenus n'ont pas non plus d'indices, il n'est pas nécessaire de conserver le noeud correspondant à ce sous-but. Par exemple, l'arbre de buts indicés:



peut être représenté par l'ABI:



Ceci permet de limiter la profondeur de l'ABI et donc d'obtenir une représentation plus efficace de la résolvante.

II.4.5 La règle d'évaluation "Arbres de Buts Indicés"

Soient Pi un programme défini indicé, et T un arbre de buts indicés.

Une étape de dérivation indicée en utilisant la stratégie ABI est décrite comme suit:

-Soit $b[j_1,j_2]$ la feuille gauche de T. Sélectionner une règle de P dont la tête s'unifie avec $b[j_1,j_2]$.

Soit a :- $b_1[i_1], b_2[i_2], \ldots, b_n[i_n]$ cette règle et θ un pgu.

-Décrémenter de 1 les indices modifiables des sous-buts de la branche gauche de T. Si certains sous-buts sont indicés par l'indice "fix", alors seuls les sous-buts situés en dessous du plus profond de ces atomes doivent être traités: Lorsqu'un sous-but d'indice "fix" est traité la mise à jour de la partie supérieure de l'arbre de buts indicés est "gelée".

- Si la partie droite de la règle est vide, alors supprimer la feuille gauche de l'arbre ainsi que tous les ancêtres de $b[j_1,j_2]$ qui ont été complètement prouvés, c'est à dire tous les noeuds qui n'ont que $b[j_1,j_2]$ comme feuille parmi leurs descendants.

  Sinon, ajouter $b_1[i_1]$, $b_2[i_2]$, ... , $b_n[i_n]$ en dessous de $b[j_1,j_2]$.
- Appliquer la substitution θ à toutes les feuilles de T.
- Effectuer la mise à jour de l'arbre T, ce qui donne un arbre T'.

L'arbre T' dérive de l'arbre T par la règle d'évaluation Arbres de Buts Indicés. L'arbre de dérivation SLD-ABI pour un but $\leftarrow$A et un programme indicé Pi est défini de la manière suivante:

- La racine de l'arbre est l'arbre de buts indicés $A[+\infty]$.
- Chaque noeud de l'arbre est un arbre de buts indicés tel que:
  - Si un noeud A' est le fils d'un noeud A, alors A' dérive de A par la stratégie ABI.
  - Pour toute règle de Pi permettant de dériver A, le noeud A a un fils correspondant. Les différents fils sont ordonnés suivant l'ordre des règles utilisées pour les obtenir. La liste vide est représentée par la clause vide et lorsqu'aucune règle ne permet de dériver A, l'abi A a un unique fils "échec".

L'arbre de dérivation SLD associé à un arbre de dérivation SLD-ABI TI est l'arbre T obtenu en remplaçant chaque ABI dans TI par son feuillage, puis en supprimant tous les indices.

II.4.6 Equité de la règle d'évaluation "Arbre de Buts Indicés"

Théorème:

Soient Pi *un programme défini indicé ne comportant aucun indice "fix" et* P *le programme défini obtenu en supprimant les indices dans* P. *Alors tout arbre de dérivation SLD pour* P *correspondant à un arbre de dérivation indicée SLD-ABI pour* Pi *est un arbre équitable.*

Lemme:

Soit $A_1, A_2, ..., A_i$, .. *une suite infinie d'arbres de buts indicés tels que* A_{i+1} *dérive de* A_i. *Alors pour tout noeud* $N[i_1,i_2]$ *de* A_j, *il existe* k *tel que* $N[i_1,i_2]$ *se trouve sur la branche gauche de* A_{j+k}.

On montre par récurrence sur n que la propriété est vraie pour tout noeud de profondeur inférieure ou égale à n.

- n=1: La racine se trouve sur la branche gauche de l'arbre.
- soit $B[i_1,i_2]$ un noeud de profondeur n+1. Soit $C[j_1,j_2]$ le noeud père de B et $B_1, B_2, .. B_p$ les frères gauches de B.

C est un noeud de profondeur n, il existe donc k tel que C se trouve sur la branche gauche de A_{i+k}. B_1 se trouve également sur la branche gauche de A_{i+k}, soit v_1 l'indice modifiable de B_1 dans A_{i+k}. Dans A_{i+k+1} l'indice modifiable de B_1 est égal à v_1-1. En itérant ce raisonnement, on montre qu'il existe un entier f(v1) tel que l'indice modifiable de B_1 dans $A_{i+f(v1)}$ est égal à 1. Donc dans $A_{i+f(v1)+1}$, le fils gauche de C est B_2.

En itérant ce raisonnement p fois, on montre qu'il existe un entier q tel que le fils gauche de C dans A_{i+q} est B. C est un noeud de A_{i+q} de profondeur n. Soit r le plus petit entier tel que C se trouve sur la branche gauche de A_{i+q+r}. B est toujours le fils gauche de C dans A_{i+q+r} et donc B est sur la branche gauche de A_{i+q+r}.

Preuve du théorème:

Soit A_1,A_2, ... ,A_i, ... une suite infinie d'ABI tels que A_{i+1} dérive de A_i. Soit B un des atomes du feuillage d'un ABI A_i. On montre que B (ou une version instanciée de B) est sélectionné au bout d'un nombre fini de dérivations. Soit k le plus petit entier tel que B se trouve sur la branche gauche de A_{i+k}. B est toujours une feuille, donc B est la feuille gauche de A_{i+k} et B est sélectionné.

L'équité n'est garantie que lorsque les programmes ne comportent pas d'atomes indicés par "fix": Lorsqu'un sous-but d'indice "fix" est traité, il bénéficie d'un nombre de dérivations non borné. La dérivation peut donc ne pas être équitable si un tel but boucle. Par contre, si l'usage de cet indice est réservé à des prédicats dont on est sur du bon comportement (c'est à dire qui ne peuvent pas boucler), la propriété d'équité est conservée. Les indices représentés par des variables Prolog ne peuvent pas engendrer de dérivation non équitable, il permettent simplement de fixer la valeur d'un indice dynamiquement par instanciation d'une variable Prolog.

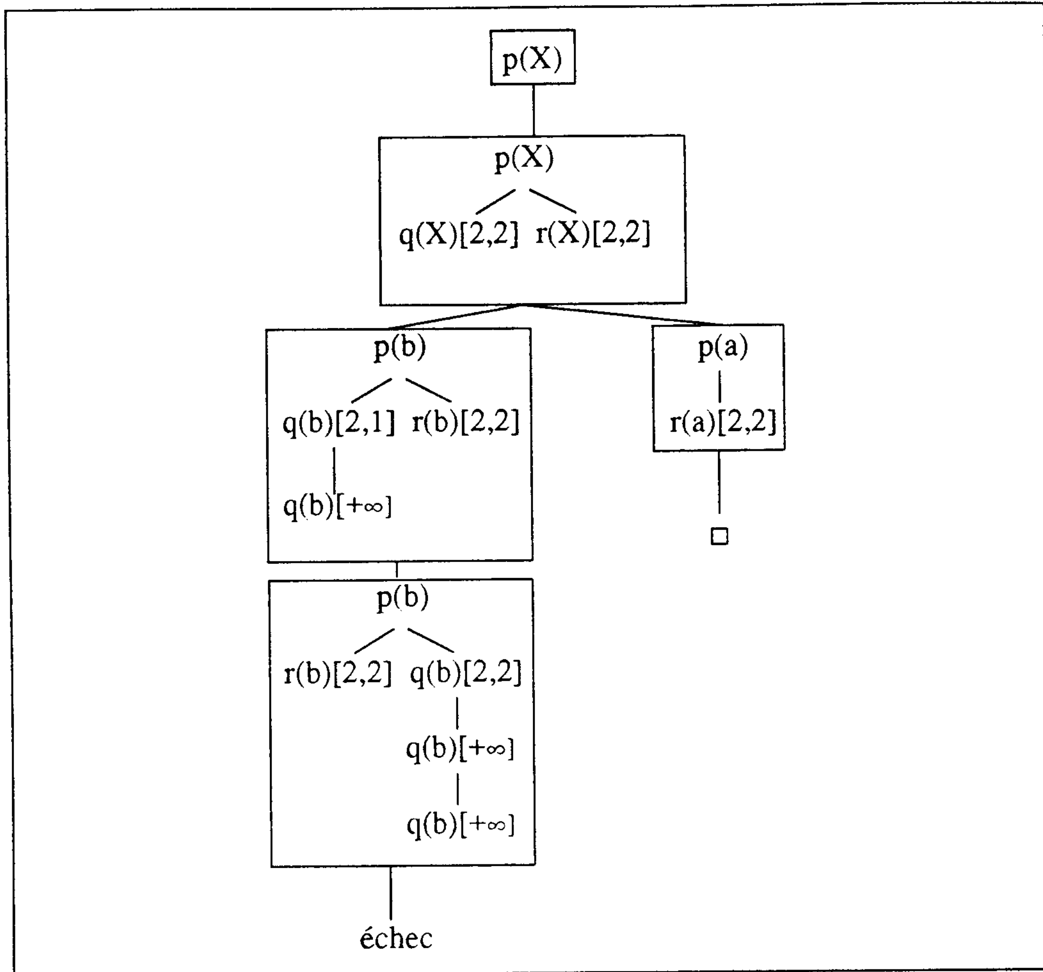
II.4.7 Exemples

Exemple 1:

On considère le programme indicé suivant:

```
p(X):- q(X)[2],r(X)[2].
q(b):- q(b)[+∞].
q(a).
r(a).
```

Pour le but ←p(X), l'arbre de dérivation indicée SLD-ABI obtenu pour P est le suivant:



Exemple 2:

Le programme indicé suivant est un programme très simple qui illustre l'utilisation d'indices représentés par des variables Prolog.

```
but(I1,I2):- but1 [i1], but2[i2].
but1 :- ecrire(1) [1], but1 [1].
but2 :- ecrire(2) [1], but2 [1].
```

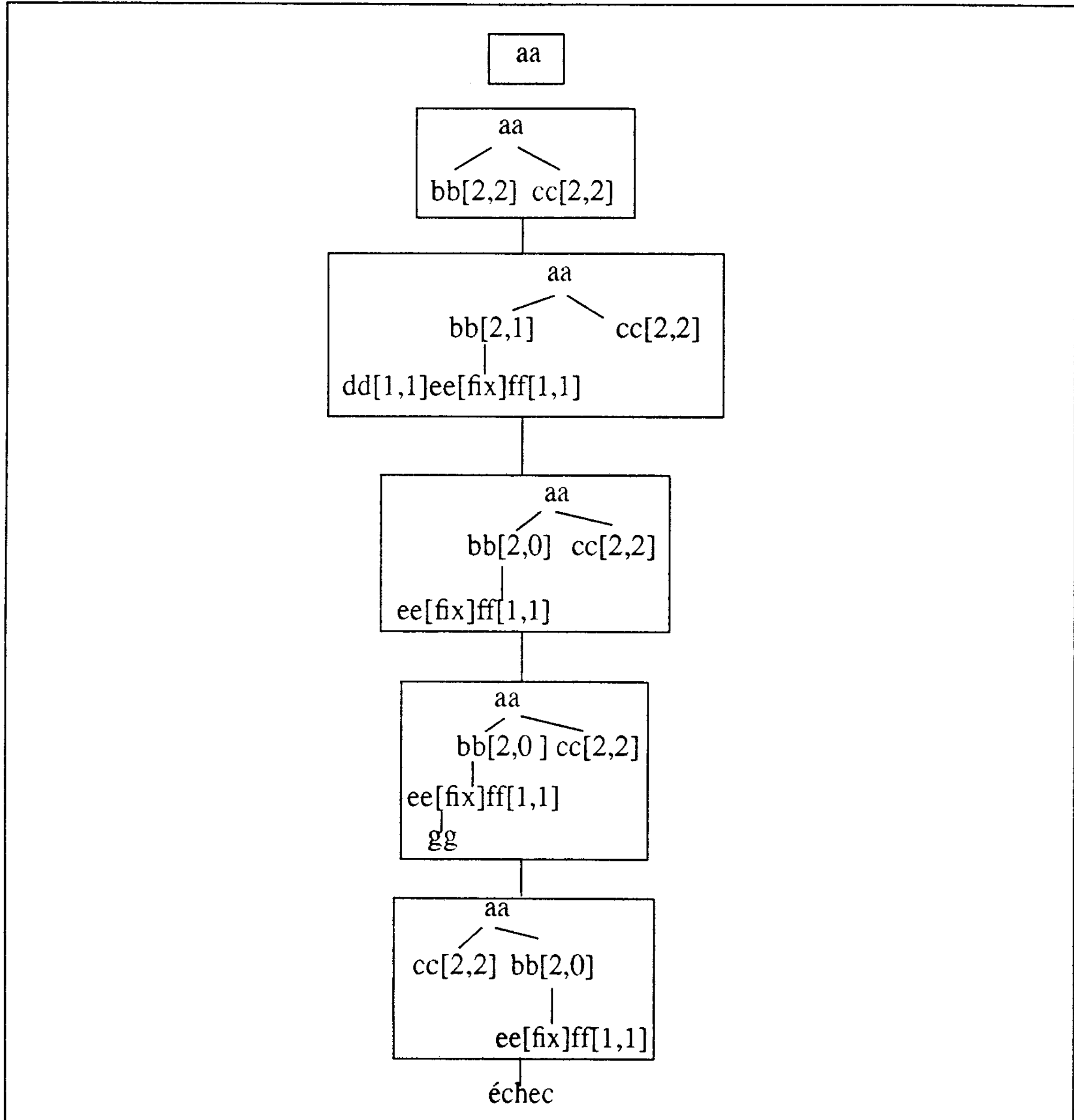
Les sous-buts `but1` et `but2` bouclent en affichant "1" et "2" (Le prédicat `ecrire` représente un prédicat prédéfini permettant d'afficher une valeur). Les indices dans la première clause permettent de dériver alternativement les sous-buts `but1` et `but2`. Avec la stratégie standard, on obtiendrait uniquement des "1" à l'affichage car le sous-but `but2` ne serait jamais sélectionné. Avec la stratégie ABI, pour le but `but(1,1)`, on obtient à l'affichage alternativement des "1" et des "2". Pour `but(1,2)`, on obtient alternativement un "1" et deux "2".

Exemple 3:

L'exemple qui suit illustre le fonctionnement de l'indice spécial "fix":

```
aa :- bb[2], cc[2].
bb :- dd[1], ee[fix], ff[1].
dd .
ee :- gg[+∞].
gg.
```

On obtient l'arbre de dérivation SLD-ABI suivant:



Un sous-but indicé par "fix" est traité dès que le sous-but auquel il est attaché (i.e son frère gauche) est effacé. Dans l'exemple ci-dessus, le sous-but bb devrait permuter avec son sous-but frère cc à la troisième étape, cc serait alors le sous-but sélectionné. Cependant, comme le but dd vient d'être effacé et que le but ee lui est lié, ee est traité immédiatement et tous les buts qui se trouvent au-dessus de ee sont "bloqués". Un sous-but indicé par "fix"

a droit à un nombre de dérivations non borné. Si un tel sous-but engendre une dérivation infinie, la dérivation n'est donc plus équitable. L'indice "fix" permet de traiter correctement les prédicats prédéfinis. Considérons par exemple la règle suivante:

```
pp(X,Y) :-
        calcul(X,R1)[1],ecrire(X)[fix],ecrire(R1)[fix],
        calcul(Y,R2)[1],ecrire(Y)[fix],ecrire(R2)[fix].
```

Le prédicat `calcul(X,R)` calcule un résultat X en fonction d'une donnée R. Les indices associés aux sous-buts `calcul(X,R1)` et `calcul(Y,R2)` permettent de développer alternativement les deux calculs. Dès qu'un calcul est terminé, les affichages correspondant sont effectués et les indices "fix" empêchent toute interférence entre les deux affichages.

Exemple 4:

Le programme `memesfeuilles` permet de comparer le feuillage de deux arbres binaires:

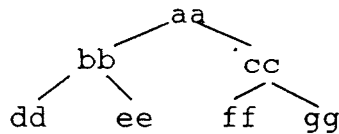
```
memesfeuilles(T1,T2):-
        feuilles(T1,L)[1],
        feuilles(T2,L)[1].
feuilles(ff(X),[X]).
feuilles(aa(ff(X),T),[X|L]):-
        feuilles(T,L)[+∞].
feuilles(aa(aa(T1,T2),T3),L):-
        feuilles(aa(T1,aa(T2,T3)),L) [+∞].
```

Les indices utilisés dans la première clause permettent de parcourir alternativement chacun des deux arbres. Lorsque les feuillages sont différents, on obtient donc un échec plus rapidement.

Exemple 5:

```
parcours(nil).
parcours(aa(X,G,D)):-
        ecrire(X),
        parcours(G)[1],
        parcours(D)[1].
but:- parcours(ff( aa,
              ff( bb, ff( dd,nil,nil),ff(ee,nil,nil)),
        ff(cc, ff(ff,nil,nil),ff(gg,nil,nil)))).
```

Ce programme permet de parcourir un arbre binaire en affichant la valeur de chaque noeud (le prédicat `ecrire` désigne un prédicat prédéfini permettant d'afficher une variable). Le but ←`but` correspond au parcours de l'arbre:



Avec les indices ci-dessus, on obtient comme affichage avec la stratégie ABI " `aa bb cc dd ff ee gg` ". Avec la stratégie standard, on obtient comme affichage " `aa bb dd ee cc ff gg` ", ce qui correspond à un parcours en profondeur d'abord de l'arbre. Avec la stratégie "en file", on obtient " `aa bb cc dd ee ff gg` ", ce qui correspond à un parcours en largeur d'abord. On peut noter que pour ce programme aucun jeu d'indices pour la stratégie ABI ne permet d'obtenir le même résultat qu'avec la stratégie en file. La stratégie en file n'est donc pas un cas particulier de la stratégie ABI.

II.4.8 Conclusions

La stratégie ABI permet de simuler la stratégie standard en employant des indices arbitrairement grands par rapport à la longueur d'une dérivation. Si tous les indices de dérivation sont supérieurs à n, il faut qu'une dérivation soit au moins de longueur n pour qu'un indice modifiable soit nul et qu'un sous-but soit placé derrière ses frères. Dans ce cas, la stratégie ABI coïncide avec la stratégie standard si aucune dérivation n'est de longueur supérieure à n.

Avec la stratégie ABI, l'indice d'un sous-but correspond exactement au nombre de dérivations consécutives consacrées à ce but avant de le permuter avec ses sous-buts frères. Contrairement à la stratégie Pile-indicée, la stratégie ABI permet de définir un mécanisme de coroutine simple (i.e. indépendant de l'état d'instanciation des variables) entre plusieurs sous-buts, comme par exemple pour le programme `memesfeuilles`.

L'utilisation de cette stratégie avec des programmes partiellement indicés permet dans certains cas de définir des mécanismes de contrôle simples et adaptés, tout en conservant les propriétés de complétude des interpréteurs équitables.

Des exemples d'utilisations particulières de la stratégie ABI sont décrits dans le chapitre V.

Chapitre III :

Réalisation d'interpréteurs équitables

III.1 Introduction

Nous décrivons dans ce chapitre l'implémentation des deux stratégies Pile-Indicée et Arbres-de-Buts Indicés. Dans un premier temps, des interpréteurs équitables ont été écrits en Prolog. Ces interpréteurs ont été réalisés sous la forme d'un méta-interpréteur Prolog modulaire dans lequel la stratégie de dérivation est définie explicitement et indépendamment des autres parties. Il est donc possible de définir ou de modifier une stratégie sans toucher au reste de l'interpréteur. Une commande permet de sélectionner l'une ou l'autre des stratégies disponibles. Ce méta-interpréteur permet donc de comparer aisément les différentes stratégies envisagées. Nous nous sommes ensuite intéressés aux techniques d'implémentation spécifiques développées afin de répondre aux besoins particulier du langage Prolog (représentation des termes Prolog, gestion de mémoire, gestion du non-déterminisme, compilation de Prolog), en vue de réaliser une implémentation efficace des stratégies de dérivation PI et ABI. Ceci nous a conduit à une collaboration avec l'équipe MALI de l'IRISA-RENNES. La machine MALI développée par cette équipe est une machine abstraite dédiée à l'implémentation des langages logiques ([BEK a], [RID]). Des interpréteurs équitables pour chacune des stratégies ont été réalisés, en collaboration avec Serge Le Huitouze, à partir de l'interpréteur PrologII/MALI [LEH a]. Nous disposons ainsi d'un interpréteur complet muni d'un récupérateur de mémoire efficace les stratégies PI et ABI. Dans les deux cas, les langages implémentés sont des sur-ensembles de Prolog: il est possible de mettre des indices seulement pour certains buts ou même de ne pas en mettre du tout, auquel cas la stratégie effectivement utilisée correspond à la stratégie standard.

III.2 Réalisations en Prolog

Nous décrivons tout d'abord l'implémentation en Prolog des stratégies équitables Pile-Indicée et Arbres de Buts Indicés. Ces stratégies ont été implémentées à partir d'un méta-interpréteur Prolog. Nous sommes partis d'un méta-interpréteur classique que nous avons modifié de manière à séparer la définition de la règle d'évaluation et la boucle de parcours de l'arbre de dérivation. On peut alors définir différentes règles d'évaluation de manière très souple. Pour ajouter une nouvelle stratégie, il suffit de définir un prédicat correspondant.

III.2.1 Méta-interpréteurs Prolog

Nous présentons ici les méta-interpréteurs qui ont servi de base à l'implémentation des interpréteurs équitables.

Considérons tout d'abord l'interpréteur *"Vanilla"* classique:

```
solve(true).
solve((A,B)):-
     solve(A),solve(B).
solve(A):-
     clause(A,B),solve(B).
```

Dans un tel méta-interpréteur, l'unification et le backtracking sont implicites. Seuls la sélection d'une clause et le choix de l'atome à dériver apparaissent explicitement. Dans cette version, la résolvante, c'est à dire la liste des sous-buts restant à prouver n'est pas accessible. Dans la troisième clause, lorsque de nouveaux buts sont obtenus, ils ne sont pas ajoutés à la liste des sous-buts restant mais traités par un appel récursif de `solve`.

Pour modifier la stratégie de dérivation, nous avons besoin de faire apparaître explicitement la résolvante. Dans le deuxième méta-interpréteur ci-dessous, la résolvante est représentée par une liste de buts. Les règles du programme objet sont représentées par des clauses Prolog `règle(Tête,Liste de buts)`.

```
résoudre([ ]).
resoudre([ A | L]) :-
     regle(A,B),append(B,L,LL),
     resoudre(LL).
```

On peut développer cet interpréteur de manière à séparer la dérivation d'un but et le parcours de l'arbre de dérivation:

```
resoudre([ ]).
resoudre(Liste_de_buts) :-
     deriver(Liste_de_buts,Nouvelle_liste),
     resoudre(Nouvelle_liste).
deriver([X|L],LL) :-
                         regle(X,Y),
                         append(Y,L,LL).
```

Dans cet interpréteur, le parcours de l'arbre de résolution et la stratégie de dérivation sont définis de manière indépendante: Le prédicat `resoudre` définit la stratégie de recherche et le prédicat `dériver` définit la stratégie de dérivation. On peut modifier l'interpréteur en récrivant l'un ou l'autre des prédicats. Par exemple, si on intervertit les deux premiers paramètres du `append`, on obtient un interpréteur avec une gestion des buts en file: les nouveaux buts obtenus sont ajoutés en queue de la résolvante au lieu d'être ajoutés en tête. De même, il est possible d'écrire un prédicat résoudre effectuant un parcours de l'arbre de

résolution en largeur d'abord. On obtient alors un interpréteur Prolog complet pour les succès.

III.2.2 Implémentation de la stratégie Pile-Indicée.

Pour la stratégie Pile-indicée, la résolvante est une liste de buts indicés, ce que l'on représente simplement par une liste Prolog dans laquelle chaque but est suivi de son indice de dérivation. Une clause définie indicée est représentée par une clause Prolog de la forme `règle(Tete, Liste de buts indicés).`

La règle d'évaluation Pile-indicée est simple à mettre en oeuvre: lorsque la résolvante est mise à jour, les éventuels nouveaux sous-buts obtenus sont ajoutés en tête ou en queue en fonction de la valeur de l'indice du sous-but sélectionné. Lorsque les nouveaux sous-buts sont ajoutés en tête, il faut parcourir la liste de ces sous-buts afin de calculer leur nouvel indice de dérivation. Ceci s'écrit de manière très simple en Prolog. Le prédicat dériver pour la stratégie pile-indicée est le suivant:

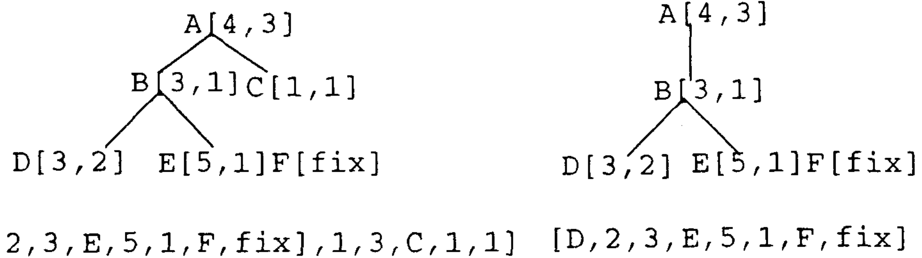
```
deriver([But,Indice|Queue_de_lbi],Nouvelle_liste):-
      regle(But,Nouv_buts),
      maj_lbi(Nouv_buts,Ind,Queue_de_lbi,Nouv_liste).
maj_lbi([ ],_,Queue_de_lbi,Queue_de_lbi):-  !.
maj_lbi(Nouv_buts,1,Queue_de_lbi,Nouv_liste):-!
             append(Queue_de_lbi,Nouv_buts,Nouv_liste).
maj_lbi([B,I| LB],IP, LBI , [ B,IPP| NLBI ]) :-
                    min(I,IP-1,IPP),
                    maj_lbi( LB ,IP, LBI , NLBI ).
```

L'ajout d'éléments en queue d'une liste par le append nécessite un parcours de cette liste, ce qui peut être évité en utilisant une liste en différence.

III.2.3 Implémentation de la stratégie Arbre de buts indicés

Cette stratégie est plus complexe à mettre en oeuvre, en raison de la structure d'arbre utilisée et des opérations de mise à jour à effectuer. Pour la stratégie ABI, deux indices de dérivation sont nécessaires lors de la résolution. Chaque indice apparaissant dans un corps de règle est dupliqué initialement. Les arbres de buts indicés sont représentés en utilisant les listes Prolog.

Exemples:



L'accès au sous-but à dériver nécessite de parcourir toute la branche gauche de l'abi. La mise à jour de l'arbre après la dérivation d'un sous-but nécessite un parcours ascendant de l'abi. Ces traitements sont implémentés à l'aide d'un seul prédicat récursif. Le traitement de l'indice spécial "fix" se fait sans trop de complications à l'aide d'une variable qui indique si un but indicé par "fix" est en cours de traitement. Le programme ci-dessous correspond à la définition de la stratégie ABI sans l'indice spécial "fix":

```
deriver([ [H|T] ,Iv,If|Q], Nouv_abi) :-!,
          % Parcours de l'arbre de buts jusqu'a la
          % feuille gauche: Traitement du sous_arbre gauche,
          % ce qui donne un nouveau sous_arbre
          deriver( [H|T] ,Nouv_sous_arbre),
          % Mis a jour de l'arbre de buts en fonction de la valeur
          % de l'indice variable du noeud et du nouveau sous_arbre
          maj_loc(Nouv_sous_arbre,Q,Iv,If,Nouv_abi).
deriver([ But,Iv,If|Abi_freres],Nouv_abi) :-
          % On se trouve sur la feuille gauche de l'abi, qui est le
          % but selectionne
          regle(But,Nouv_buts),
          % Mis a jour de l'arbre de buts en fonction de la valeur
          % l'indice variable du but selectionne et des nouveaux
          % buts obtenus.
          maj_loc(Nouv_buts,Abi_freres,Iv,If,Nouv_abi).
          % Le predicat maj_loc (mise a jour locale) effectue
          % la mise a jour du niveau courant de l'ABI
maj_loc([],AL,_,_,AL) :- !.
          % 1 er cas : Le nouveau sous_arbre gauche est vide.
maj_loc(NL,[],_,_,NL) :- !.
          % 2 ieme cas : Le but traite n'a pas de freres, il peut
          % donc etre supprime.
maj_loc(Nouv_buts,Abi_freres,1,I,Nabi) :- !,
          % 3 ieme cas : L' indice variable du but derive vaut 1,
```

```
                % dans ce cas le nouveau sous_arbre obtenu
                %permute avec ses freres.
                append(Abi_freres,[Nouv_buts,I,I],Nabi).
maj_loc(Nouv_buts,Abi_freres,Iv,If,[Nouv_buts,Iv2,If|Abi_freres) :-
                % 4 ieme cas : L'indice du but derive est
                % strictement superieur a 1,on le decremente d'une unite
                % et on 'accroche' le nouveau sous_arbre obtenu.
                Iv2 is Iv-1.
```

L'implémentation des indices variables pour la stratégie ABI ainsi bien que pour la stratégie PI ne nécessite aucun développement supplémentaire car les indices de dérivation des programmes indicés manipulés sont représentés par des variables Prolog. Un indice de dérivation variable est donc tout simplement une variable Prolog libre qui est instanciée au cours de la dérivation.

Une optimisation intéressante consiste à ne pas augmenter la profondeur de l'ABI lorsque cela n'est pas utile, c'est à dire lorsque l'indice du but sélectionné est égal à $+\infty$ et que les indices des nouveaux sous-buts sont tous égaux à $+\infty$. Cette optimisation est facilement réalisée en définissant une nouvelle clause pour le prédicat `maj-loc` et en déterminant au moment du chargement d'un programme indicé si un corps de règle contient ou non des indices différents de $+\infty$.

III.2.4 Interpréteurs avec "cut"

Nous avons souhaité implémenter un "cut" dans les interpréteurs équitables, afin de pouvoir utiliser des programmes Prolog standards avec ces interpréteurs. L'utilisation d'un "cut" avec une stratégie non standard pose certains problèmes, le fonctionnement du "cut" étant étroitement lié au comportement opérationnel de Prolog standard. Ces problèmes sont discutés dans le chapitre III.4.2. L'effet du "cut" sur le comportement de l'interpréteur est le suivant:

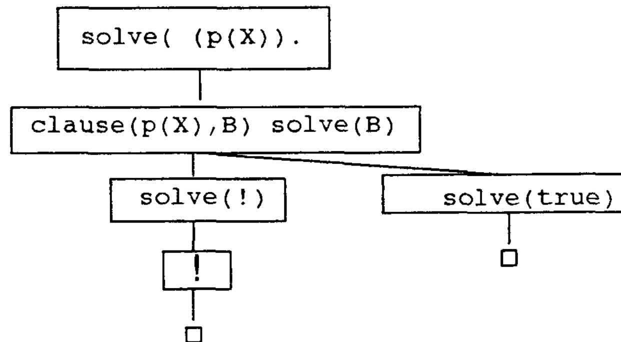
- Lorsqu'un sous-but "cut" est sélectionné, il est évalué à vrai.

- Pendant le parcours de l'arbre de résolution, si au cours d'un retour arrière on rencontre un noeud dont la liste de buts commence par "cut", alors on remonte directement jusqu'au noeud précédant le but qui a déclenché l'appel de la règle contenant le "cut", c'est à dire deux noeuds au-dessus de la première occurrence de ce "cut".

L'implémentation d'un "cut" dans un méta-interpréteur Prolog est délicate. La première idée qui vient à l'esprit pour définir un cut dans l'interpréteur *vanilla* est d'ajouter une clause `solve(!):-!`. Cette méthode ne fonctionne pas car la coupure intervient trop bas dans l'arbre de résolution. Considérons par exemple le programme:

```
p(a) :- !.
p(b).
```

Pour le but ←`solve(p(X))` avec l'interpréteur vanilla, on obtient l'arbre de dérivation suivant:



On peut remédier à ce problème si l'on dispose d'un prédicat prédéfini permettant de déclencher une coupure à un niveau supérieur de l'arbre de résolution. Une solution utilisant un tel prédicat est proposée dans [S, S]. Une autre solution, utilisant de manière subtile le "ou" disponible dans les Prolog type Edimburgh, est donnée dans [C,C]:

```
solve(true,_).
solve(!,_).
solve(!,cut).
solve((A,B), V ) :-
            solve( A,C ),
            (C==cut, V=cut ; solve(B,V) )).
solve(A, V ) :-
            clause(A,B) , solve(A, V ),
            (V==cut, ! , fail ; true).
```

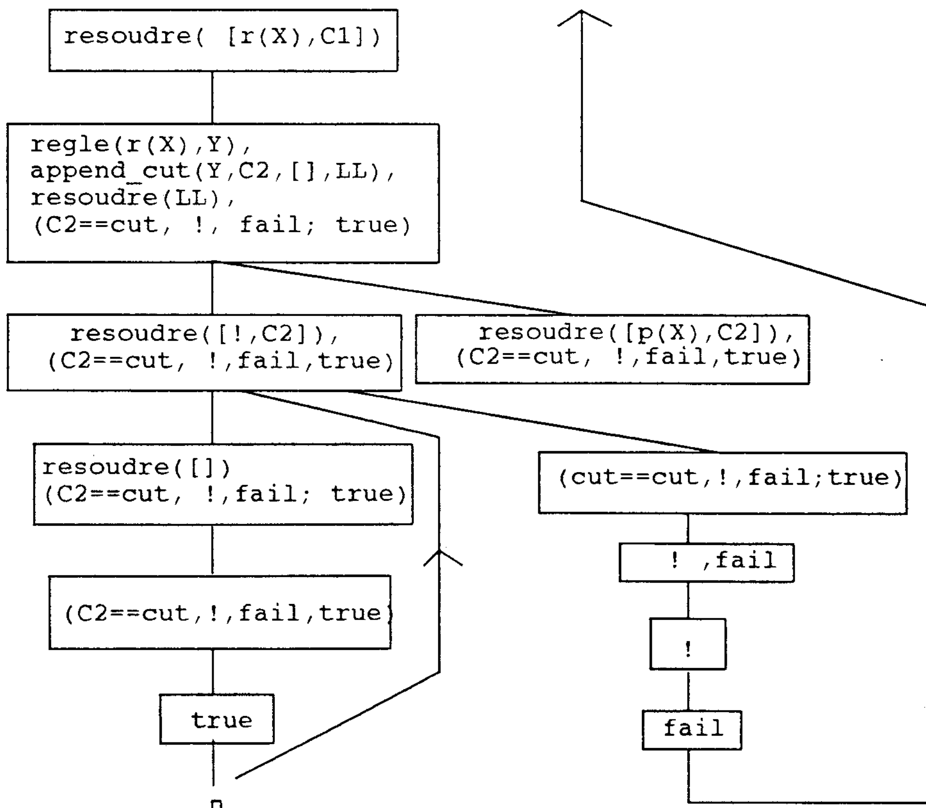
Cette solution consiste en quelque sorte à associer un "cut" potentiel à chaque sous-but. Lorsque ce sous-but est dérivé, si ce sous-but est effectivement un "cut", le "cut" potentiel qui lui est associé est déclenché et la coupure est effectuée à l'endroit voulu. En adaptant cette solution au deuxième méta-interpréteur, on obtient l'interpréteur suivant:

```
resoudre([ ]).
resoudre([ ! , _| L) :- resoudre(L).
resoudre([ !, cut | _ ]).
resoudre([ A,_  | L]) :-
                  regle(A,B),
                  append_cut(B, C ,L,LL),
                  resoudre(LL),
                  (C==cut , ! , fail ; true ).
append_cut([ ] , _ , L , L).
append_cut([ X | Q], C , L , [ X, C | LL]) :-
                  append_cut(Q, C , L ,LL).
```

La figure ci-dessous illustre le fonctionnement de cet interpréteur pour le programme:

```
r(X):-!.
r(X):-p(X).
```

et le but `r(X)`.



Lorsque le cut est sélectionné, il provoque une remontée dans l'arbre jusqu'au père du noeud où le cut est apparu pour la première fois, c'est-à-dire au-dessus du point de choix correspondant au but `regle(r(X),Y)`.

Une implémentation du cut pour les interpréteurs équitables a été réalisée en adaptant cette méthode. Cette adaptation ne se fait pas de manière simple, en particulier pour la stratégie ABI et il faut adapter le prédicat résoudre pour chaque stratégie. L'inconvénient est une perte de modularité: les prédicats définissant le type de parcours de l'arbre SLD et la stratégie de dérivation ne sont plus indépendants.

L'utilisation du "cut" peut sembler à priori incompatible avec une stratégie non standard. Néanmoins, le cut a été implémenté afin d'obtenir un langage Prolog indicé qui soit un sur-ensemble de Prolog standard. Ceci permet en particulier de tester les interpréteurs équitables avec des programmes standards. L'emploi du "cut" avec une stratégie non standard soulève de nouveaux problèmes: il ne faut pas qu'un "cut" soit sélectionné avant que le but qui le précède soit prouvé. Dans le paragraphe III.4.2, nous indiquons des possibilités d'utilisations du "cut" avec des indices "fix". De plus, le "cut" est utilisé pour implémenter la négation (cf paragraphe III.5.2).

III.3 Réalisation d'interpréteurs équitables avec MALI

III.3.1 Introduction

Nous avons utilisé la machine MALI ([BEK a]) pour réaliser une implémentation efficace des stratégies équitables Pile-Indicée et Arbre de Buts Indicés. L'implémentation d'interpréteurs équitables a été réalisée à partir d'un interpréteur déjà existant, PrologII/ MALI. Ceci nous a permis d'implémenter les stratégies équitables en ne modifiant que la stratégie de dérivation. Nous avons donc hérité de tout l'environnement de programmation ainsi que du prédicat `dif` particulier à PrologII et du cut. Nous présentons brièvement dans ce chapitre l'implémentation réalisée avec MALI.

La Machine Abstraite de Warren (WAM, [WAR]), utilisée dans la plupart des implémentations de Prolog actuelles, est étroitement liée à la sémantique procédurale de Prolog (d'où son efficacité) et par conséquent ne permet pas l'implémentation de stratégies non standards qui ne respectent pas cette sémantique procédurale. La machine MALI est plus générale que la WAM et n'est pas liée à une règle d'évaluation particulière. En MALI, la résolvante est représentée par un terme MALI. Sa structure peut être aisément modifiée, ce qui permet de définir des stratégies plus complexes. Dans MALI, l'interpréteur et le récupérateur de mémoire sont indépendants. On peut donc modifier la stratégie de dérivation sans avoir à se préoccuper de la gestion de mémoire.

III.3.2 La machine MALI

MALI (Machine Adaptée aux Langages Indéterministes) est une machine abstraite dédiée à l'implémentation des langages logiques, développée à l'IRISA-RENNES. MALI est décrite dans [RID], [BEK a]. Nous présentons brièvement la machine MALI. MALI comprend d'une part un ensemble de commandes permettant de créer et de parcourir des termes et d'autre part des commandes permettant de gérer une pile de termes et un récupérateur de mémoire adapté à Prolog.

III.3.2.1 Les termes MALI

Les structures de données manipulées par MALI sont des termes analogues aux termes Prolog.

Les principales commandes pour les termes MALI sont:

- la création et l'accès de termes construits binaires
- la création et l'accès de termes construits n-aires
- la création et la substitution de variables MALI
- la création et la substitution de variables à attribut.

Les variables MALI sont des variables logiques qui correspondent aux variables de Prolog. Les variables à attribut sont des variables logiques étendues ([LEH b]): ces variables comportent en plus un champ "attribut" (qui est un terme) qui n'est visible que lorsque la variable est dans l'état libre. Ces variables permettent l'implémentation de termes modifiables.

III.3.2.2 Gestion de l'indéterminisme

La mise en oeuvre de l'indéterminisme est réalisée à l'aide d'une pile de recherche. Les commandes `sauvegarder` et `reprendre` permettent d'accéder à cette pile: `sauvegarder` crée un point de choix et `reprendre` permet de dépiler un point de choix lors du retour arrière.

Une autre commande, utilisée pour implémenter le "cut", a pour effet de couper la pile de recherche à un niveau donné.

III.3.2.3 Le récupérateur de mémoire de MALI

Le récupérateur de mémoire de MALI est un récupérateur de mémoire spécifique respectant la logique d'utilité de Prolog. Son fonctionnement est décrit dans [RID].

Le récupérateur de MALI est déclenché par la commande `réduire`. Cette commande a pour paramètres des termes représentant l'ensemble des termes utiles. Le récupérateur n'est déclenché effectivement que lorsqu'un certain seuil d'occupation de la mémoire est atteint.

III.3.3 PrologII/MALI

Les interpréteurs équitables ont été développés à partir d'un interpréteur Prolog déjà existant, PrologII/MALI. PrologII/MALI est un interpréteur Prolog réalisé avec la machine MALI (version v04) [LEH a] qui est complètement compatible avec le langage PrologII [COL b]. Cet interpréteur comprend en particulier les prédicats `dif` et `geler` de PrologII. L'utilisation de MALI a permis d'obtenir une gestion de mémoire efficace.

Cet interpréteur a servi de base au développement de nos interpréteurs équitables pour les stratégies Pile-Indicée et Arbres de Buts Indicés.

III.3.4 Implémentations

III.3.4.1 Modifications apportées à PrologII/MALI

Les modifications apportées pour implémenter nos stratégies équitables concernent la représentation des clauses, afin de prendre en compte les indices de dérivation, et le "moteur d'inférences" de l'interpréteur, qui décrit les opérations à effectuer au cours d'une

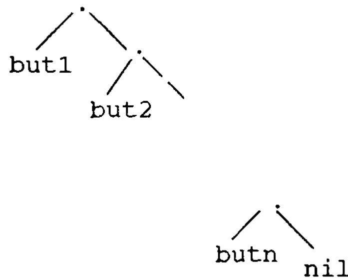
étape de dérivation. Les autres parties de l'interpréteur ont pu être conservées telles qu'elles.

Ceci nous a permis d'hériter de l'algorithme d'unification de PrologII/MALI ainsi que de tout l'environnement de programmation, en particulier tous les prédicats système. Grâce à cela, nous avons pu obtenir assez rapidement des interpréteurs complets utilisant des règles d'évaluation équitables.

La gestion de mémoire est toujours effectuée par le récupérateur de mémoire de MALI, qui est indépendant de l'interpréteur.

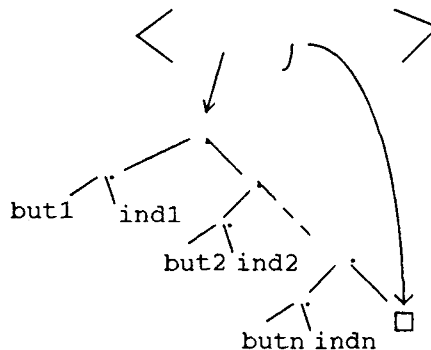
III.3.4.2 Implémentation de la stratégie Pile-Indicée

Dans PrologII/MALI, la résolvante est représentée explicitement par une liste de sous-buts, qui est modifiée à chaque dérivation: Le premier sous-but de la liste est supprimé et est remplacé par une liste de sous-buts issue de l'exemplarisation d'un corps de clause. La résolvante a la forme suivante:



Pour implémenter la règle d'évaluation pile-indicée, les besoins nouveaux sont l'introduction d'indices de dérivation dans les corps de clauses ainsi que dans la résolvante et la possibilité d'ajouter de nouveaux sous-buts en queue de la résolvante au lieu de les ajouter en tête.

Pour permettre l'ajout en queue sans parcours de la liste de buts, la résolvante est représentée par une liste en différence:



Le dernier élément de la liste est une variable, à laquelle on substitue la nouvelle queue de liste.

Lors de l'ajout en tête, la liste des nouveaux sous-buts est parcourue afin de déterminer la valeur de l'indice de chaque sous-but.

Les autres parties de l'interpréteur ne nécessitent pas de modifications. En particulier, l'unification, la gestion de mémoire et la gestion de l'indéterminisme (i.e le parcours de l'arbre de recherche) sont inchangés.

III.3.4.3 Implémentation de la stratégie Arbres de buts indicés

Cette stratégie de dérivation est nettement plus complexe à implémenter. La résolvante habituelle, qui est la liste des sous-buts restant à prouver, est remplacée par un arbre de buts indicés. La construction d'une nouvelle résolvante nécessite beaucoup plus de mises à jour que dans le cas de la stratégie pile-indicée.

Les opérations à effectuer à chaque étape sont: développer la feuille gauche, décrémenter les indices de la branche gauche et éventuellement permuter des sous-arbres.

Les arbres de buts indicés sont représentés par des n-uplets de la forme suivante:

```
Noeud :  < indice fixe , indice modifiable , sous-arbre >
Feuille :  < indice fixe , indice modifiable , littéral >
```

La modification des indices ainsi que les permutations de sous-arbres doivent être effectuées en évitant de faire une recopie de tout l'arbre de buts à chaque fois. La solution adoptée consiste à utiliser les variables à attribut pour représenter des termes modifiables: Un terme modifiable est représenté par une variable à attribut (vara) dont l'attribut est égal à la valeur courante du terme, la modification du terme est réalisée en substituant à la vara la nouvelle valeur, qui peut être également un terme modifiable représenté par une vara.

Pour la stratégie ABI, l'accès au sous-but sélectionné, qui est la feuille gauche de l'arbre, nécessite un parcours de la branche gauche de l'arbre de buts. Ceci entraîne un ralentissement important du temps d'accès au sous-but lorsque la profondeur de l'arbre augmente.

III.3.5 Evaluation des interpréteurs équitables PrologII/PI et PrologII/ABI

Nous discutons dans ce chapitre des performances obtenues avec les deux interpréteurs, PrologII/PI et PrologII/ABI. Nous comparons les temps d'exécution des interpréteurs équitables et de PrologII/MALI pour des programmes standards. Ceci permet d'évaluer le coût technique des modifications réalisées: les écarts de temps représentent la perte d'efficacité lorsque l'équité n'est pas utilisée.

III.3.5.1 PrologII/PI et PrologII/ABI

Les deux langages implémentés, appelés PrologII/PI et PrologII/ABI sont en fait des sur-ensembles de PrologII: les indices de dérivation sont facultatifs, chaque atome d'un corps de clause peut être affecté ou non d'un indice de dérivation. Un indice $+\infty$ est affecté à chaque atome qui n'a pas d'indice.

L'équité est garantie lorsque le programme est indicé conformément aux définitions du chapitre II. Lorsque certains buts sont indicés par $+\infty$, la dérivation peut ne pas être équitable: les sous-buts indicés par $+\infty$ sont traités de manière non équitable, ils peuvent être dérivés indéfiniment sans que les autres sous-buts soient sélectionnés.

Les interpréteurs PrologII/PI et PrologII/ABI peuvent donc être utilisés avec des programmes standards, ce qui permet de comparer leurs performances avec celles de PrologII/MALI: lorsque tous les indices sont égaux à $+\infty$, ainsi que lorsque tous les indices sont supérieurs à la profondeur de l'arbre standard, les arbres de dérivation obtenus avec les deux règles d'évaluation PI et ABI sont les mêmes qu'avec la stratégie standard. Nous avons comparé les performances des interpréteurs équitables avec celles de l'interpréteur PrologII/MALI dans ces deux cas de figure. Cette comparaison permet d'évaluer le coût des modifications réalisées pour implémenter les stratégies équitables.

Lorsque la règle d'évaluation effectivement utilisée est différente, la comparaison des temps d'exécution n'a plus de sens, dans la mesure où les arbres de dérivation obtenus peuvent varier considérablement

III.3.5.2 Performances de l'interpréteur PrologII/PI

La mise en oeuvre de la stratégie Pile-Indicée ne nécessite pas de structures de données beaucoup plus complexes. La perte d'efficacité est donc relativement faible, comme le montre le tableau ci-dessous:

append3	nrev(400)	huit-reines	qsort(400)	hanoi(10)	fib(20)	ack(3,3)
1,16	1,13	1,39	1,71	1,73	1,86	1,88

Rapport entre les temps d'exécution de PrologII/PI et PrologII/MALI

Ces temps sont obtenus avec des programmes standards: dans ce cas, un indice infini est affecté à chaque atome et donc la règle d'évaluation utilisée par l'interpréteur pile-indicée est la règle standard.

L'augmentation des temps d'exécution est liée à l'utilisation de structures de données plus importantes ainsi qu'aux traitements supplémentaires effectués à chaque étape de dérivation. La perte d'efficacité est plus importante pour les programmes utilisant des prédicats arithmétiques. Ceci provient d'une optimisation de PrologII/MALI, la compilation des prédicats arithmétiques en tête de clause, qui n'existe pas dans les interpréteurs équitables.

Ces différences de performances nous paraissent tout à fait raisonnables compte tenu des possibilités supplémentaires introduites grâce aux indices de dérivation.

Les temps mis pour l'inversion d'une liste avec naïve-reverse ainsi que pour le tri d'une liste avec quicksort montrent que l'écart entre PrologII/MALI et PrologII/PI ne dépend pas de la taille de la liste.

n	50	100	150	200	250	300	400	500
t1/t2	1,15	1,15	1,22	1,14	1,13	1,18	1,13	1,11

Rapport entre les temps d'exécution de PrologII/PI et PrologII/MALI
pour l'inversion d'une liste à n éléments avec naive-reverse

n	50	100	150	200	250	300	400	500
t1/t2	1,59	1,72	1,75	1,71	1,70	1,68	1,71	1,68

Rapport entre les temps d'exécution de PrologII/PI et PrologII/MALI
pour trier une liste déjà triée avec quicksort

Ceci montre que le récupérateur de mémoire de MALI a un comportement tout à fait régulier avec PrologII/PI: les modifications effectuées pour implémenter la stratégie équitable ne remettent pas en cause le bon fonctionnement de l'interpréteur. L'écart plus important obtenu pour le quicksort est du à l'optimisation de PrologII/MALI mentionné ci-dessus. En ajoutant un prédicat nil en tête de la deuxième clause, le prédicat arithmétique n'est plus en tête et l'optimisation est sans effet. On obtient alors des écarts moindres:

n	50	100	150	200	250	300	400	500
t1/t2	1,34	1,36	1,36	1,42	1,35	1,30	1,38	1,36

Rapport entre les temps d'exécution de PrologII/PI et PrologII/MALI
pour trier une liste déjà triée avec quicksort modifié

Le tableau suivant montre les écarts obtenus lorsqu'on utilise des programmes indicés, mais en choisissant des indices de dérivation arbitrairement grands, de sorte que la stratégie effectivement utilisée est la stratégie standard. Pour le naïve-reverse avec un indice 100000 affecté à chaque atome, on obtient les rapports suivant:

n	50	100	150	200	250	300	400	500
t1/t2	1,02	1,00	1,02	1,01	1,01	1,03	1,01	1,01

Rapport entre les temps d'exécution de PrologII/PI
pour naive-reverse avec indices et naive-reverse sans indices

Ces écarts montrent que le temps de traitement des indices est négligeable pour la stratégie Pile-Indicée.

III.3.5.3 Performances de l'interpréteur PrologII/ABI

Pour la stratégie Arbres de Buts Indicés, la structure de données utilisée et les algorithmes de gestion de la résolvante sont beaucoup plus complexes. Les performances de l'interpréteur en sont par conséquent considérablement diminuées. De plus, lorsqu'on utilise cet interpréteur avec des programmes standards, aucune optimisation n'est réalisée pour prendre en compte le fait que les sous-buts ne permutent jamais: la résolvante est toujours représentée par un arbre.

Le tableau ci-dessous donne quelques exemples de temps d'exécution de PrologII/ABI:

append3	nrev(400)	huit-reines	qsort(400)	hanoi(10)	fib(20)	ack(3,3)
12	19	19	35	14	16	50

Rapport entre les temps d'exécution de PrologII/ABI et PrologII/MALI

Avec la stratégie ABI, l'accès au sous-but sélectionné nécessite un parcours de la branche gauche de l'arbre de buts. Le temps nécessaire pour effectuer une étape de dérivation est donc dépendant de la longueur de la branche gauche de l'arbre. La longueur de cette branche augmente a chaque fois que le sous-but sélectionné produit au moins deux nouveaux sous-buts. Pour le naïve-reverse, la profondeur de l'ABI augmente à chaque dérivation d'un sous-but `reverse` avec une liste non vide comme premier argument. L'inversion d'une liste de taille n engendre donc la construction d'un ABI de profondeur n.

n	50	100	150	200	250	300	350	400
t1/t2	19	35	53	63	79	95	105	117

Rapport entre les temps d'exécution de PrologII/ABI et PrologII/MALI
pour l'inversion d'une liste à n éléments avec naive-reverse

n	50	100	150	200	250	300	400	450
t1/t2	21	36	50	64	78	92	106	120

Rapport entre les temps d'exécution de PrologII/ABI et PrologII/MALI
pour trier une liste déjà triée avec quicksort

Ces résultats montrent que les temps d'exécution deviennent beaucoup trop importants dès que la taille de l'arbre augmente.

L'utilisation d'indices avec cette stratégie est également coûteuse: le temps d'inversion d'une liste de taille 100 en utilisant naïve-reverse est multiplié par 2 lorsque chaque sous but est affecté d'un indice 100000. Ceci vient du fait que l'indice modifiable de chaque but se trouvant sur la branche gauche est mis à jour à chaque étape de dérivation.

Le tableau suivant montre les résultats obtenus pour la concaténation de trois listes avec le programme append3. Pour ce programme, la résolvante est un arbre de buts indicés ne comportant que trois noeuds: chaque dérivation d'un sous-but append donne au plus un

nouveau sous-but. Dans ce cas, il n'est pas nécessaire de conserver le noeud correspondant dans l'ABI. La profondeur de l'abi est donc limitée à deux dans ce cas.

n	500	1000	1500	2000
t1/t2	12	12	9	7

Rapport entre les temps d'exécution de PrologII/ABI et PrologII/MALI
pour concaténer trois listes à n éléments

III.3.6 Conclusions

L'utilisation de la machine MALI pour implémenter nos interpréteurs équitables s'est avérée concluante: nous avons pu, à partir de PrologII/MALI, implémenter des règles d'évaluation nécessitant des structures de données plus complexes. L'utilisation de PrologII/MALI nous a permis d'obtenir rapidement des interpréteurs équitables disposant d'un environnement de programmation et d'une gestion de mémoire efficace.

Les réalisations effectuées à partir de PrologII/MALI montrent qu'il est possible d'étendre un interpréteur pour le rendre équitable avec un surcoût raisonnable (pour la stratégie Pile-Indicée).

Cependant, les interpréteurs équitables réalisés souffrent des mêmes limitations que PrologII/MALI: PrologII/MALI est un interpréteur et non un compilateur, aussi ses performances sont assez éloignées de celle des systèmes Prolog actuels basés sur la WAM [WAR], comme par exemple Quintus Prolog [QUI] ou SICStus Prolog [SIC].

Comme nous l'avons mentionné au début de ce chapitre, une implémentation des stratégies équitables à partir d'un système basé sur la WAM n'est pas possible directement: Dans la WAM, les listes de sous-buts sont représentés à travers une pile, aussi il n'est pas possible de modifier la gestion des sous-buts pour implémenter une règle d'évaluation non standard. Cependant, une implémentation de stratégies équitables utilisant certaines des instructions de la WAM (par exemple pour l'unification, la représentation des termes, la compilation des clauses) est tout à fait envisageable.

III.4 Stratégies non standards et prédicats prédéfinis

En prolog "pur", on peut modifier la règle d'évaluation sans remettre en cause la correction et la complétude. En pratique, l'utilisation d'une stratégie non standard, qui conduit à modifier l'ordre d'évaluation des sous-buts, peut sembler à priori incompatible avec l'utilisation de prédicats prédéfinis. Nous examinons les problèmes liés à l'usage des prédicats prédéfinis avec une stratégie non standard et décrivons les solutions adoptées pour rendre nos interpréteurs équitables pratiquement utilisables.

III.4.1 Traitements des prédicats système avec les stratégies équitables

Dans un interpréteur "réel", l'utilisation de prédicats prédéfinis avec une stratégie non standard peut poser des problèmes: l'ordre d'évaluation des sous-buts est modifié, un sous-but peut être sélectionné avant que les sous-buts qui le précèdent soient effacés. Considérons par exemple une clause `p(X):- q(X), write(X)`. Si la stratégie utilisée n'est pas la stratégie standard (de gauche à droite et en profondeur d'abord), le prédicat `write(X)` peut être sélectionné avant que `q(X)` soit effacé, et donc potentiellement avant que la variable `X` soit instanciée. Ceci est donc une source de problèmes avec nos stratégies équitables qui n'évaluent pas toujours les sous-buts séquentiellement. Nous proposons deux solutions pour résoudre ces problèmes:

• L'indice spécial "fix": Dans nos interpréteurs équitables, un indice spécial, noté "fix", permet de résoudre ces problèmes. Un sous-but indicé par "fix" dans un corps de clause est lié au sous-but qui le précède. Dans ce cas, il est sélectionné dès que le sous-but qui le précède est effacé. Dans l'exemple, si on écrit: `p(X):- q(X)[10], write(X)[fix]`, le sous-but `write(X)` est sélectionné uniquement si `q(X)` est effacé et dans ce cas dès que `q(X)` est effacé. D'autre part, un sous-but indicé par "fix" bénéficie d'un nombre de dérivations non borné, de telle sorte que son traitement ne peut pas être interrompu. Ceci permet de traiter correctement les prédicats système, en particulier les prédicats d'entrées-sorties. Cependant, l'indice "fix" n'est pas utilisable pour un sous-but qui est en tête d'un corps de clause, ce qui peut être le cas en particulier pour des prédicats arithmétiques.

• Suspension des prédicats système: Une autre solution est de suspendre les prédicats système jusqu'à ce que les variables requises soient instanciées: lorsqu'un prédicat prédéfini est sélectionné, il n'est exécuté que si les variables requises sont instanciées, dans le cas contraire le prédicat est suspendu jusqu'à ce qu'il soit suffisamment instancié. Un tel mécanisme est utilisé dans MU-Prolog [NAI a], ainsi que dans PrologII pour le prédicat `dif`. Ceci peut être réalisé en utilisant une primitive de retardement analogue au `Geler` de PrologII [COL b]. Dans les versions Prolog des interpréteurs équitables, une primitive

suspendre(L,Pred) a été développée à cette fin. Cette primitive a pour effet de suspendre le sous-but Pred jusqu'à ce que toutes les variables de L soient closes.

L'utilisation systématique d'une primitive de retardement pour les prédicats prédéfinis permet de palier à tous les problèmes éventuels provenant de la modification de l'ordre d'évaluation des sous-buts, en protégeant ces prédicats contre toute sélection prématurée.

III.4.2 Utilisation du cut avec les interpréteurs équitables

Le cut, bien que très critiqué, est couramment utilisé dans les systèmes Prolog actuels. Pour cette raison, nous avons souhaité en disposer dans nos interpréteurs. De plus, le cut est utilisé pour implémenter la négation par l'échec. L'implémentation du "cut" est liée uniquement au mécanisme de backtraking, par conséquent le cut de PrologII/MALI est hérité dans les interpréteurs PrologII/ABI et PrologII/PI. Le cut a été également implémenté dans les versions Prolog des interpréteurs équitables.

L'effet du cut sur la résolution est le suivant: lors d'un retour arrière, si on rencontre un noeud qui commence par un cut, alors on remonte directement jusqu'au noeud précédant le but qui a fait apparaître le cut.

Cependant, l'utilisation du cut est étroitement liée à la règle d'évaluation de Prolog standard. Le cut est utilisé pour supprimer des points de choix sous certaines conditions. Si l'exécution des sous-buts n'est plus séquentielle, un cut peut alors être sélectionné prématurément et provoquer des coupures autres que celles attendues par le programmeur.

Considérons le programme suivant:

```
p(X):- r(X),!.
p(X):- q(X).
r(X):- s(X).
q(a).
s(b).
```

Le cut dans la première règle indique qu'il faut couper les choix sur le but p lorsque le sous-but r réussit. En Prolog standard, le but ←p(b) donne un succès car la deuxième règle est appliquée après l'échec du sous-but r(b). Par contre, le but ←p(X) donne un seul succès avec la substitution X=a: le cut de la première règle supprime les choix restant pour p(X) et empêche l'application de la deuxième règle. Avec une stratégie non standard, le cut peut être sélectionné avant que r(X) soit effacé. Considérons par exemple une version indicée du programme obtenue en indiceant la première règle de la manière suivante:

```
p(X) :- r(X)[1], ! [1].
```

Avec la stratégie ABI, les sous-buts r(X) et ! permutent après la première dérivation, et ! est sélectionné avant que r(X) soit effacé ou échoue. On obtient alors un échec pour le but ←p(a).

L'utilisation de l'indice spécial fix pour les cut permet d'éviter ce problème: si le sous-but ! est indicé par "fix", il ne sera sélectionné que si le sous-but `r(X)` est prouvé et dans ce cas dès que `r(X)` est effacé. Le cut ne pourra alors pas provoquer de coupures supplémentaires. L'usage systématique de l'indice "fix" pour indicer les cut qui ne se trouvent pas en position de premier sous-but d'un corps de clause permet donc d'éviter les coupures prématurées (lorsqu'un cut se trouve en tête d'un corps de clause, il n'y a pas de risque de sélection prématurée de ce cut).

Cependant, lorsqu'une règle d'évaluation différente est utilisée, si un but donne plusieurs solutions, ces solutions sont susceptibles d'être obtenues dans un ordre différent. Par conséquent, si un programme initialement non déterministe comporte des "cut", l'ensemble des solutions obtenues avec une stratégie non standard peut être différent.

Considérons par exemple le programme indicé suivant:

```
a(X) :- p(X) [2], b(X)[2].
b(a).
p(X):- r(X)[5],! [fix].
p(X):- q(X).
r(X):- s(X).
q(a).
s(b).
```

Dans ce cas, Prolog standard (avec le programme obtenu en supprimant les indices) échoue pour le but ←`a(X)`, car le sous-but `p(X)` provoque l'instanciation de `X` à `b` et `b(b)` échoue. Avec la stratégie ABI, on obtient un succès avec `X=a` car les sous-buts `s(X)` et `b(X)` permutent et c'est le sous-but `b(X)` qui instancie `X`.

En prolog standard, le cut a pour effet de supprimer la solution unique `X=a`, alors qu'avec la stratégie ABI, cette solution n'est pas supprimée. Cependant, on peut considérer que cette différence n'est pas gênante dans la mesure où il s'agit là d'un usage du cut incorrect: le but ←`a(X)` échoue alors qu'un but moins général, ←`a(a)`, donne un succès.

III.5 Implémentation de la négation

III.5.1 Négation en Prolog standard

En prolog standard, les sous-buts négatifs sont simplement traités par un appel récursif qui effectue le test de la négation par l'échec. Ceci est une source supplémentaire d'incomplétude: si l'appel récursif boucle, les autres sous-buts ne sont jamais pris en compte. De plus, il se peut qu'un atome clos A soit dans l'ensemble des échecs finis alors que l'arbre de dérivation standard est infini. Dans ce cas, un interpréteur standard boucle pour la question $\leftarrow \sim A$ alors que la règle de négation par l'échec permet de déduire $\sim A$.

D'autres problèmes viennent du fait que la règle de sélection des littéraux utilisée n'est pas sure: lorsqu'un littéral négatif est sélectionné, on ne vérifie pas qu'il est clos. Si un littéral négatif non clos est dérivé, cela peut donner des réponses incorrectes si des variables sont instanciées lors de la dérivation de ce littéral. La règle de sûreté peut être implémentée à l'aide d'un mécanisme de suspension des buts: si le littéral sélectionné est un littéral négatif comportant des variables libres, il est suspendu jusqu'à ce que ces variables soient instanciées. Les exemples qui suivent illustrent les différents problèmes liés au traitement de la négation.

Exemple 1:

```
p(X):- ~q(X).
q(a):- r(a), s(a).
r(a):- r(a).
s(b).
```

Pour le but $\leftarrow$p(a), un interpréteur standard boucle car il boucle pour q(a): l'arbre de dérivation SLD standard pour q(a) est infini, alors que tout arbre équitable est fini.

Exemple 2:

```
a(X):- ~b(X), c(X).
b(a).
c(b).
```

Un interpréteur Prolog standard donne un échec pour le but $\leftarrow$a(X): ~b(X) est sélectionné et échoue car il existe une réfutation de b(X). Cette réponse est incorrecte: si une règle d'évaluation sure est utilisée, c(X) est sélectionné avant b(X) et on obtient une solution avec la substitution réponse X=b.

Exemple 3:

```
p(X) :- ~q(X), r(X).
```

```
q(a) :- q(a).
r(b).
```

a) Pour le but ←p(a), un interpréteur standard boucle en essayant de prouver q(a), alors que le deuxième sous-but, r(a) échoue. Avec une règle d'évaluation équitable développant alternativement la dérivation subsidiaire de q(a) et la dérivation principale, r(a) est sélectionné et la dérivation échoue.

b) Pour le but ←p(X), Prolog standard boucle également. Avec une règle sure, on obtient un succès pour ←p(X), avec la solution X=b, car r(X) est sélectionné avant ~q(X).

Des implémentations correctes de la négation ont été réalisées dans IC-PROLOG [CLA b], MU-PROLOG [NAI b] et NU-PROLOG [NAI c] ainsi que dans PROLOGII/MALI [BEK b]. Dans ces implémentations, une condition de sûreté affaiblie est utilisée, afin d'éviter certains cas d'enlisement: un littéral négatif non clos peut être sélectionné, mais aucune instanciation ne doit intervenir lors du traitement de ce littéral. Ces interpréteurs évitent donc les réponses incorrectes, mais pas les boucles liées à la règle d'évaluation. Dans l'exemple 1, on obtient toujours une boucle pour le but ←p(a). Dans l'exemple 3, on obtient une boucle pour ←p(a), mais pas pour ←p(X). Nous décrivons maintenant une implémentation équitable de la négation avec la stratégie ABI.

III.5.2 Négation avec la stratégie ABI

La stratégie ABI permet de définir un traitement de la négation complètement équitable: tous les sous-buts d'une dérivation qui n'échoue pas sont sélectionnés, même si le traitement d'un littéral négatif donne une dérivation subsidiaire qui boucle. Pour les exemples 1 et 3 a), les boucles sont donc évitées et un échec fini est obtenu.

III.5.2.1 Négation équitable

Pour cela, nous commençons par définir la négation de manière classique à partir du cut:

```
not(P) :- P, !, fail.
not(P).
```

Cette définition correspond au fonctionnement de la négation dans les interpréteurs standards.

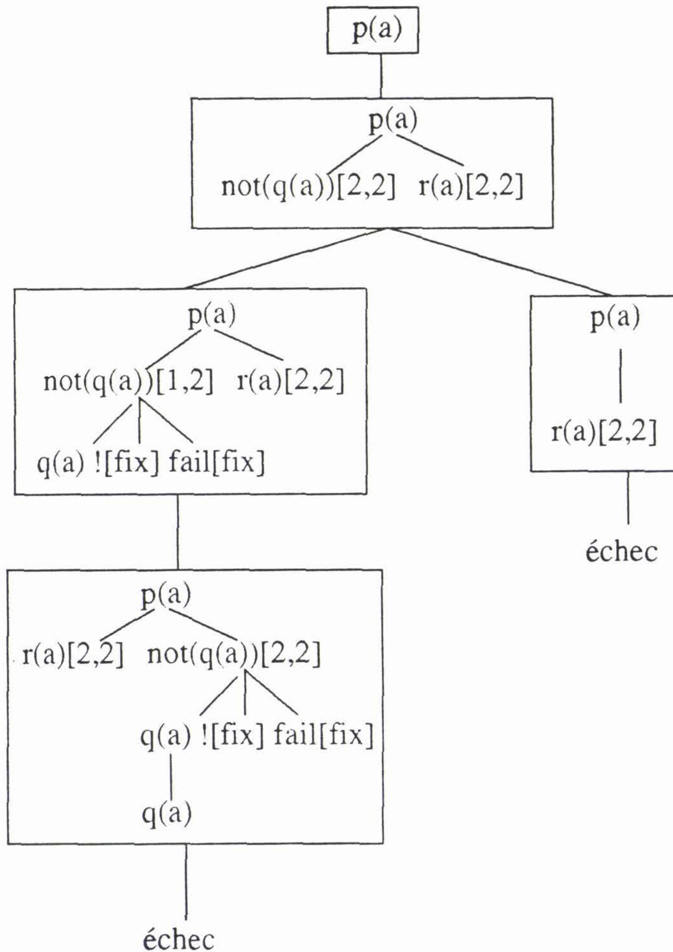
Lorsqu'un sous-but `not(q)` apparait dans un corps de règle, il doit être associé à un indice de dérivation, `not(q)` est donc traité de manière équitable: le nombre de dérivations consécutives que l'on peut lui consacrer avant d'examiner les autres sous-buts est borné par son indice de dérivation. Pour que les sous-buts `!` et `fail` soient sélectionnés immédiatement après p, il est nécessaire de leur associer un indice "fix":

```
not(P) :- P, ! [fix], fail [fix].
not(P).
```

La définition de la négation de cette façon permet d'obtenir des dérivations complètement équitables. Considérons une version indicée du programme de l'exemple 3:

```
p(X) : not(q(X))[2], r(X)[2].
q(a) :- q(a).
r(b).
```

Pour le but ←`p(a)`, on obtient un échec. L'arbre SLD-ABI envisagé est le suivant:

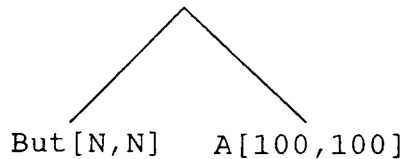


Cette figure fait apparaître un inconvénient de cette méthode: comme le sous-but `q(a)` ne s'efface pas, le cut n'est pas sélectionné et la deuxième règle définissant le not est appliquée lorsque r(a) provoque un échec, ce qui conduit à explorer une branche inutile: on obtient un échec pour les mêmes raisons dans la deuxième branche (dans la mesure où le sous-but négatif est clos).

III.5.2.2 Négation sure

Bien entendu, l'implémentation décrite ci-dessus n'est pas sure: un littéral négatif non clos peut être sélectionné et provoquer des réponses incorrectes. Nous décrivons maintenant une

implémentation de la négation sure. Le moyen le plus simple pour implémenter une négation par l'échec sure est de suspendre les littéraux négatifs non clos. Nous avons défini pour cela une primitive de retardement, `suspendre(Listvar, But)`, permettant de suspendre un prédicat en fonction de plusieurs variables. Le prédicat `suspendre(Litstvar, But)` a pour effet de suspendre le sous-but `But` jusqu'à ce que toutes les variables de la liste `Listvar` soient closes. Dans un interpréteur standard, lorsqu'un but suspendu est réveillé, il peut être traité directement. Dans le cadre d'un interpréteur équitable, il est souhaitable que le traitement des buts suspendus ne mettent pas en cause l'équité. Plus précisément, un but réveillé ne doit pas bénéficier d'un nombre de dérivations consécutives illimitées. Pour cela, dans notre interpréteur, lorsqu'un but est réveillé, il est replacé dans la résolvante de manière équitable. Ceci est réalisé en greffant une branche à l'arbre de buts indicés courant. Lorsqu'un but `But` est réveillé, si `A` est l'arbre représentant la résolvante courante, la nouvelle résolvante construite lors du réveil de `But` est l'arbre suivant:



N est l'indice de dérivation qui était associé au sous-but `suspendre(L, But)`. L'indice de `A` (égal à 100) est fixé arbitrairement: il est nécessaire d'associer un indice entier à ce sous-arbre pour garantir l'équité (cet indice n'intervient que si `But` et `A` permutent au moins deux fois, c'est à dire si la dérivation de `But` est de profondeur supérieure à N et la dérivation de `A` est de profondeur supérieure à 100).

III.5.2.3 Négation sure et équitable

En combinant les deux aspects décrits précédemment, on peut réaliser une implémentation de la négation à la fois sure et équitable (sous la forme d'un prédicat prédéfini `not`). Nous appelons maintenant `not_clos` le prédicat défini à l'aide du cut:

```
not_clos(P):- P,! [fix], fail [fix].
not_clos(P).
```

Le nouveau prédicat `not` fonctionne de la façon suivante: lorsqu'un sous-but `not(P)` est sélectionné, alors:

- Si `P` est clos, alors on remplace `not(P)` par `not_clos(P)`.
- Sinon, on remplace `not(P)` par le sous-but `suspendre(L,not_clos(P))`, où `L` est la liste des variables libres de `P`.

Le prédicat not décrit ci-dessus est implémenté uniquement dans la version écrite en Prolog de la stratégie ABI. Cette implémentation de la négation est correcte (car la règle d'évaluation est sure) et équitable: les littéraux négatifs sont suspendus s'ils ne sont pas clos et sont traités équitablement par le biais du prédicat `not_clos`.

Exemples:

• Considérons une version indicée de l'exemple 3:

```
p(X):- not(q(X))[10], r(X)[10].
q(a):- q(a).
r(b).
```

On obtient un succès pour le but ←`p(X)`, avec la réponse `X=b`, car la règle de sélection est sure: `not(q(X))` est suspendu, puis `r(X)` est sélectionné et provoque l'instanciation de `X` à `b`, `not(q(b))` est alors réveillé et réussit car `q(b)` échoue.

• On considère le programme indicé suivant:

```
p(X):- not(q(X))[2],r(X),s(X).
r(b).
r(a).
q(b):- q(b).
s(a).
```

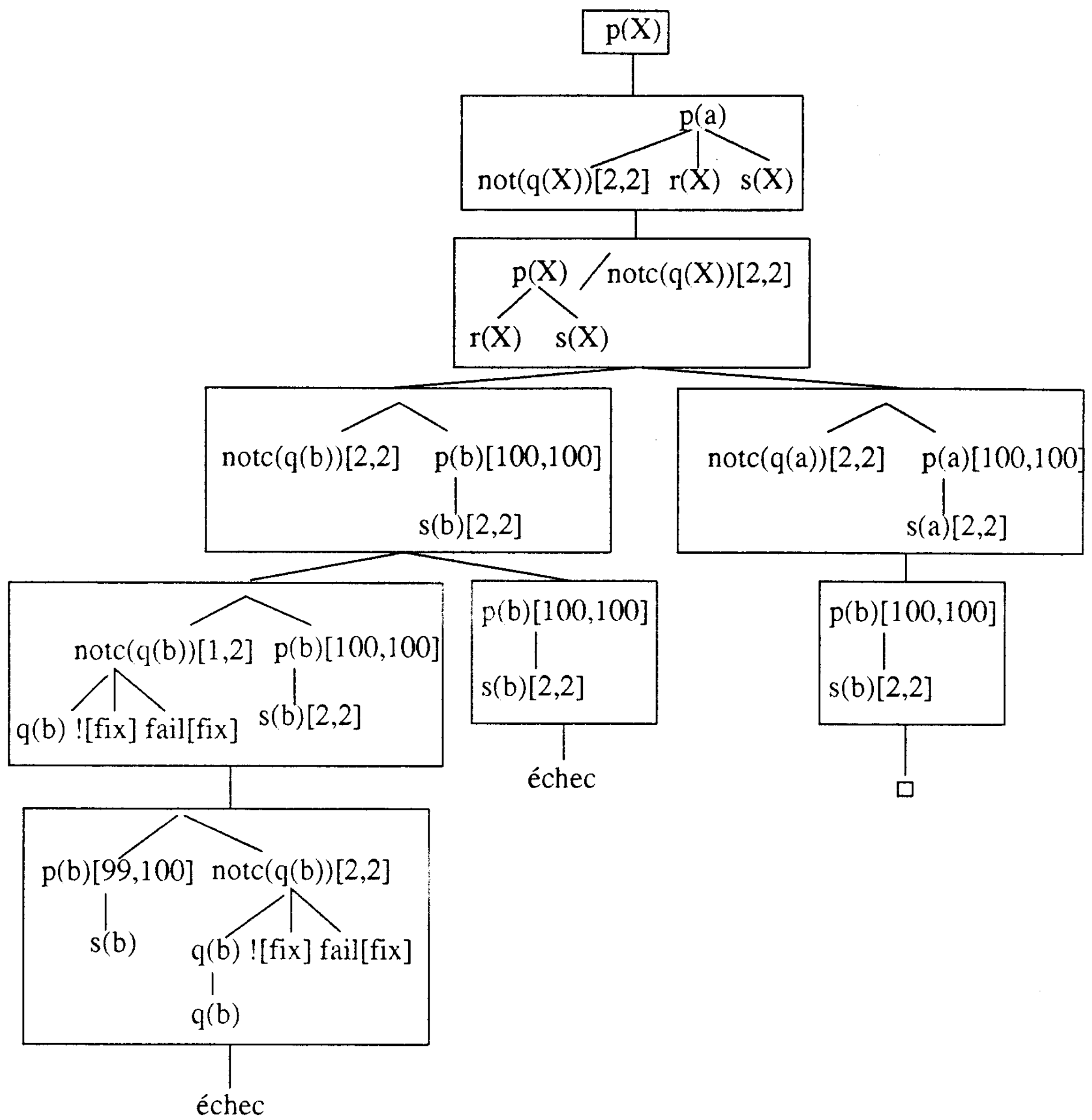
Pour le but ←`p(X)`, notre interpréteur donne un succès (avec `X=a`) alors qu'une implémentation sure mais non équitable boucle lorsque le but `not(q(b))` est réveillé. L'arbre de dérivation obtenu pour ←`p(X)` illustre le fonctionnement du `not`.

Malheureusement, cette implémentation de donne pas un interpréteur complet pour les échecs pour les programmes avec négation (même lorsqu'il n'y a pas d'enlisement). Ceci est lié au fait que lorsqu'un littéral négatif ~A est sélectionné, pour essayer de trouver une réfutation ou un échec fini pour le but ←A, on construit un arbre de dérivation en profondeur d'abord. Ceci permet de trouver tous les échecs finis (lorsque la règle d'évaluation est équitable), mais pas toutes les réfutations: lorsque l'arbre de dérivation comporte une branche infinie à gauche de la première branche de succès, un parcours en profondeur d'abord boucle sans donner de succès. Considérons par exemple le programme P:

```
p :- p
p.
q :- not(p).
```

Il existe un arbre SLDNF d'échecs finis pour le but ←`q` (car il existe une réfutation de p), et donc ~q est conséquence logique de comp(P). Cependant, un interpréteur équitable mais effectuant un parcours en profondeur d'abord pour le test de la négation par l'échec boucle. Contrairement au cas des programmes sans négation, un interpréteur équitable n'est pas complet pour les échecs finis.

Néanmoins, il est possible de réordonner les clauses de telle sorte que la première clause vide se trouve avant la première branche infinie.


p(X)
p(a)
not(q(X))[2,2] r(X) s(X)
p(X) / notc(q(X))[2,2]
r(X) s(X)
notc(q(b))[2,2] p(b)[100,100]
s(b)[2,2]
notc(q(a))[2,2] p(a)[100,100]
s(a)[2,2]
notc(q(b))[1,2] p(b)[100,100]
q(b) ![fix] fail[fix] s(b)[2,2]
p(b)[100,100]
s(b)[2,2]
échec
p(b)[100,100]
s(b)[2,2]
p(b)[99,100] notc(q(b))[2,2]
s(b) q(b) ![fix] fail[fix]
q(b)
échec

Chapitre IV :

Techniques de contrôle et règle d'évaluation

IV.1 Introduction

Nous étudions dans ce chapitre les liens entre la règle d'évaluation et la taille de l'arbre de dérivation SLD obtenu.

Nous nous intéressons tout d'abord aux techniques de contrôle ayant pour effet de modifier la règle d'évaluation standard. Ces méthodes consistent à modifier dynamiquement l'ordre d'évaluation des sous-buts afin d'éviter certaines boucles et/ou d'optimiser l'exécution des programmes.

Nous étudions ensuite les variations de complexité des arbres de dérivation SLD lorsque la règle d'évaluation standard est modifiée. L'exemple du naïve-reverse montre que ces variations peuvent être considérables. Afin d'évaluer l'apport potentiel d'une règle d'évaluation équitable, nous distinguons différents cas, selon que l'arbre standard est fini ou non et que la taille d'un arbre équitable est bornée ou non.

IV.2 Contrôle dynamique en Prolog

IV.2.1 Introduction

Par contrôle dynamique, nous désignons les techniques visant à améliorer l'efficacité au moment de l'exécution d'un programme. Ceci comprend entre autres les techniques de détection de boucles (voir par exemple [APT b]) ou de backtracking intelligent [COD], qui ont pour effet d'élaguer l'arbre de recherche. Nous nous intéressons uniquement aux méthodes qui consistent à modifier la règle d'évaluation standard afin d'éviter des boucles et d'obtenir des exécutions plus efficaces. Typiquement, l'objectif de ces méthodes est d'obtenir une exécution efficace de programmes faciles à lire et à écrire, mais inefficaces (ou qui bouclent) en Prolog standard. Ces méthodes ont des points communs avec les stratégies équitables dans la mesure où elles permettent d'éviter des boucles liées à la règle d'évaluation standard de Prolog. Nous étudions tout d'abord quelques unes des extensions proposées dans des systèmes Prolog afin de modifier dynamiquement l'ordre d'évaluation des sous-buts. Nous présentons ensuite les techniques de contrôle de MU-PROLOG [NAI b], qui permettent de déterminer automatiquement la règle d'évaluation utilisée.

IV.2.2 Primitives de contrôle

Nous étudions ici les méthodes visant à améliorer l'efficacité d'un programme en utilisant une règle d'évaluation plus adaptée. Ces méthodes s'appliquent essentiellement à des dérivations comportant un schéma producteur/consommateur. L'exemple type envisagé est le programme des huit-reines (voir § V.3.2), version "naïve". La première règle a la forme suivante:

```
solution(Sol) :-
        permut([1,2,3,4,5,6,7,8],Sol),
        correcte(Sol) .
```

Un sous-but (`permut`) non-déterministe génère une liste de positions qui est ensuite testée par un autre sous-but (`correcte`). Avec la règle d'évaluation standard, ceci engendre de nombreux calculs inutiles car chaque liste doit être complètement calculée avant d'être testée. Dans une telle situation, il serait intéressant de pouvoir spécifier un ordre d'évaluation plus adapté, permettant de tester au fur et à mesure chaque nouvelle position calculée. De nombreux outils de contrôle ont été développés à cet effet. Gallaire et Lasserre ont proposé un langage de contrôle à l'aide de métarègles [G, L]. Plusieurs extensions de Prolog proposent des connecteurs "et" spéciaux permettant de faire des évaluations pseudo parallèles, par exemple IC-PROLOG [CLA b], EPILOG [POR a] ou TWO-LEVEL PROLOG [POR b]. D'autres mécanismes permettent de modifier l'ordre d'évaluation des buts en fonction de l'état d'instanciation des variables. Nous décrivons ci-dessous les annotations de variables de IC-PROLOG [CLA b], le geler de PrologII [COL b] et les déclarations WAIT de MU-PROLOG [NAI b].

IV.2.2.1 Les annotations de variables

IC-PROLOG propose un mécanisme d'annotation des variables, qui détermine des priorités pour les sous-buts. Un sous-but B ayant une variable X comme argument peut être soit un consommateur avide (eager consumer) de X (la variable est notée X?), soit un producteur paresseux (lazy producer) de X (X est notée ^X). Lorsque B est un producteur paresseux de X, cela donne la plus faible priorité à B tout en empêchant tout autre but de lier X: si un autre but essaie de lier X, il est mis en attente et B est sélectionné. Si B est un consommateur avide de X, B a la plus forte priorité: le sous-but B doit être le premier consommateur de X et est sélectionné dès que X est instanciée. Les annotations de variables définissent un contrôle dirigé par les données: l'instanciation des variables détermine l'ordre d'évaluation des sous-buts.

Les annotations de variables sont données par l'utilisateur, qui doit donc être capable de déterminer correctement quels sont les sous-buts producteurs et les sous-buts consommateurs pour que le contrôle obtenu soit efficace.

IV.2.2.2 Geler

Le geler est un prédicat prédéfini proposé dans PrologII [COL b]. Le geler permet de suspendre un but jusqu'à ce qu'une variable soient instanciée: lorsqu'un sous-but `geler(X,B)` est sélectionné, si X est une variable libre, alors le sous-but B est mis en attente, sinon B est sélectionné normalement.

L'exemple ci-dessous, tiré de [GIA], illustre le fonctionnement de geler:

```
plus(x, y, z) -> geler(x, geler(y, somme(x,y,z)));
somme(x,y,z) > val(add(x,y));
```

Le prédicat `plus(x,y,z)` calcule la somme de x et y uniquement si ces deux variables sont instanciées, sinon il est mis en attente. L'utilisation du geler avec des listes permet d'exprimer des contraintes sur la construction des listes et de mettre en place des mécanismes de coroutines entre producteur et consommateur. Des exemples de programme PrologII utilisant le geler sont présentés dans [GIA].

Le prédicat geler est un outil de contrôle simple et utile qui permet d'exprimer très facilement des contraintes d'instanciation pour certains appels de sous-buts. Le geler, introduit initialement dans PrologII, a été repris dans d'autres systèmes Prolog (SICStus Prolog [SIC] par exemple).

IV.2.2.3 Les déclarations WAIT

Les déclarations WAIT de MU-PROLOG [NAI b], comme le prédicat Geler, ont pour effet de suspendre un sous-but lorsque certaines conditions d'instanciation ne sont pas vérifiées. Cependant, les déclarations WAIT s'appliquent à la définition d'une procédure et non à un appel particulier d'un sous-but. Les déclarations WAIT portent sur plusieurs arguments et on peut avoir plusieurs déclarations WAIT pour le même prédicat.

Nous reprenons la définition du append avec les déclarations WAIT donnée par Naish [NAIa]:

```
append([],A,A).
append([A|B],C,[A|D]):-
            append(B,C,D).
?- wait append(1,1,0).
?- wait append(0,1,1).
```

Une déclaration WAIT pour un prédicat associe un 0 ou un 1 pour chaque argument. Un "1" indique qu'un argument peut être construit lors de l'unification. Un argument est construit s'il contient une variable qui est unifiée avec un terme qui n'est pas une variable. Lors d'une unification réussie, l'unification fournit la liste des arguments construits, qui est comparée avec les déclarations WAIT. Le sous-but réussit s'il n'a pas de déclarations WAIT ou bien si la liste des arguments construits est compatible avec l'une des déclarations WAIT, sinon le sous-but est suspendu jusqu'à ce que l'une des variables en cause soit instanciée. Par exemple, le seul argument construit lors de la dérivation du but ←`append(X,[3],[1,2,3])` est X, ce qui est autorisé par la déclaration `?-wait append(1,1,0)`. Par contre, l'unification du but ←`append(X,[1,2],Y)` avec la tête de la deuxième règle construit à la fois le premier et le troisième argument, ce qui est interdit par les deux déclarations WAIT. Les déclarations WAIT définissent les différents

modes possibles pour chaque prédicat, ce qui permet d'éviter des boucles en interdisant certains modes. Complétons le programme ci-dessus pour définir le prédicat append3:

```
append3(A,B,C,D):- append(A,B,L), append(L,C,D).
```

Les déclarations WAIT (pour `append`) permettent d'éviter la branche infinie obtenue en Prolog standard lorsque le premier argument est une variable libre.

Pour le but ←`append3(X,[3],[4],[1,2,3,4])`, on obtient la dérivation suivante ([NAIa]) (les sous-buts qui se trouvent après "&" sont les sous-buts suspendus):

```
←append3(X,[3],[4], [1,2,3,4])
                   |
←append(X,[3],C) append(C,[4],[1,2,3,4])
                   |
←append(C,[4],[1,2,3,4]) & append(X,[3],C)
                   |
←append(X,[3],[1|C1]), append(C1,[4],[2,3,4])
                   |
←append(X1,[3],C1), append(C1,[4],[2,3,4])
                   |
←append(C1,[4],[2,3,4]) & append(X1,[3],C1)
                   |
                   |
```

Chaque sous-but append avec le premier et le troisième argument libres est suspendu, ce qui évite une branche infinie.

IV.2.3 MU-PROLOG

MU-PROLOG, contrairement aux autres systèmes Prolog évoqués précédemment, propose un algorithme de génération automatique de contrôle, basé sur l'utilisation des déclarations WAIT. Dans MU-PROLOG, des déclarations WAIT sont générées automatiquement, ce qui permet d'éviter certaines des boucles liées à la règle d'évaluation standard et d'obtenir des dérivations plus efficaces: lors de l'exécution d'un programme, la règle d'évaluation standard est modifiée dynamiquement par la suspension des sous-buts qui ne satisfont pas les déclarations WAIT.

IV.2.3.1 Génération des déclarations WAIT

Les déclarations WAIT sont calculées automatiquement par un pré-compilateur. La méthode consiste à détecter des boucles potentielles, en comparant les arguments des appels récursifs avec ceux de la tête de la règle, puis à ajouter suffisamment de déclarations WAIT pour éviter ces boucles. L'algorithme produit généralement les déclarations adaptées. Cependant, l'algorithme échoue dans certains cas, par exemple lorsque les

arguments d'un sous-but récursif sont aussi généraux que ceux de la tête. Dans ce cas, le sous-but ne peut être suspendu par les déclarations WAIT. Lorsque trop peu de déclarations WAIT sont générées, cela peut conduire à suspendre des buts qui n'engendrent pas de branches infinies.

IV.2.3.2 Réordonnancement des sous-buts

L'utilisation des déclarations WAIT est complétée par un réordonnancement des sous-buts dans les corps de clause, qui est fait automatiquement par le pré-compilateur. La principale heuristique appliquée consiste à placer les tests avant les générateurs. Par exemple, pour le programme des huit-reines, l'ordre des sous-buts dans la première clause est inversé:

```
solution(Sol) :-
        correcte(Sol),
        permut([1,2,3,4,5,6,7,8],Sol).
```

Une déclaration `?-wait correcte(0)` est générée, ce qui indique que l'argument de `correcte` ne doit pas être une variable libre. Lorsque que le sous-but `correcte` est sélectionné initialement, il est donc suspendu, puis réveillé lorsque le premier élément de la liste est produit. Cette méthode permet d'obtenir un ordre d'évaluation optimal: chaque nouvelle valeur produite provoque le réveil d'un appel récursif du test et est donc testée immédiatement.

IV.2.4 Conclusions

Nous avons présenté quelques méthodes de contrôle basées sur la modification de la règle d'évaluation afin d'obtenir un contrôle dirigé par les données. Ces méthodes sont efficaces dans certains cas, notamment pour des exemples types (tels que les huit-reines), en ce sens qu'elles permettent de minimiser la taille de l'arbre de dérivation. Cependant, les calculs effectués pour trouver un ordre d'évaluation adapté peuvent être très coûteux. A l'exception de MU-PROLOG, les différents systèmes évoqués fournissent des primitives de contrôle supplémentaires mais le contrôle n'est pas automatique. L'usage de ces primitives est parfois délicat et requiert une analyse approfondie du programme.

MU-PROLOG permet d'effectuer un contrôle automatique grâce à la génération automatique des déclarations WAIT. Le contrôle obtenu est efficace et permet d'éviter certaines des boucles liées à la stratégie de dérivation standard (avec les déclarations WAIT générées, `append3` en mode (o,o,o,i) et `permut` en mode (o,i) ne bouclent pas). Cependant, une telle méthode, qui consiste à améliorer la stratégie standard grâce à un mécanisme de suspension des sous-buts, ne peut être équitable et garantir la complétude. Lorsqu'un sous-but est sélectionné, il n'est effectivement dérivé que si les conditions exprimées par les déclarations WAIT sont vérifiées, mais rien ne permet d'assurer qu'un

sous-but va être effectivement sélectionné. Naish suggère de combiner les déclarations WAIT avec une règle d'évaluation équitable pour obtenir la complétude ([NAI a]).

Nous n'avons évoqué dans ce chapitre que des techniques ayant pour effet de modifier la règle d'évaluation. Ces techniques sont coûteuses car elles effectuent un contrôle dynamique, qui nécessitent des calculs supplémentaires au moment de l'exécution du programme.

Pour éviter l'overhead lié au contrôle dynamique, Bryunooghe [BRU] propose une méthode de transformation de programme qui permet d'obtenir le même effet: étant donné un programme et une règle d'évaluation efficace, qui peut être déterminée automatiquement [VER], ce programme est transformé, à partir de la trace, en un programme qui est efficace avec la règle d'évaluation standard. Cependant, cette méthode s'applique uniquement à des buts qui ne bouclent pas avec la règle d'évaluation standard.

IV.3 Complexité de l'arbre SLD et règle d'évaluation

IV.3.1 Introduction

Les résultats de complétude de la résolution SLD montrent que l'existence ou non d'une clause vide ne dépend pas de la stratégie de dérivation employée: s'il existe un arbre de dérivation SLD pour un programme et un but donné qui contient la clause vide, alors tous la contiennent. De plus, les branches de succès des différents arbres envisageables sont équivalentes. Cependant, la taille de l'arbre envisagé peut varier considérablement en fonction de la règle d'évaluation utilisée. Nous étudions dans ce chapitre les différents cas de figure.

IV.3.2 Exemple

Nous montrons sur un exemple classique, le programme `naïve-reverse`, les variations de complexité des arbres de dérivation SLD obtenus avec différentes règles d'évaluation.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
nrev([],[]).
nrev([X|L1],L):- nrev(L1,L2),append(L2,[X],L).
```

En Prolog standard, le naïve-reverse fonctionne en mode(i,o), c'est-à-dire avec le premier argument clos, et est déterministe dans ce cas: Si le premier argument est une liste instanciée et le deuxième une variable libre, l'arbre SLD contient une branche unique qui donne un succès. Le nombre d'unifications nécessaires pour inverser une liste de taille n, c'est-à-dire la longueur de cette branche, est fonction de n^2. Par contre, avec une règle d'évaluation "en file", l'arbre contient des branches d'échec supplémentaires et le nombre d'unifications pour inverser une liste de taille n est fonction de n^4 (cf III.2). L'arbre de dérivation peut également être infini: c'est le cas par exemple en employant la règle d'évaluation qui sélectionne toujours l'atome le plus à droite. Par conséquent, tout arbre de dérivation SLD équitable est fini (car l'arbre standard est fini), mais de taille non bornée car il existe des arbres de dérivation infinis. On peut donc construire un arbre de dérivation SLD arbitrairement grand. Dans ce cas, l'arbre de dérivation SLD envisagé par Prolog standard est optimal et l'utilisation d'une autre règle d'évaluation donne un arbre dont la taille n'est pas bornée.

Le tableau ci-dessous donne le nombre de dérivations obtenues pour nrev avec différentes stratégies: gestion des buts "en pile" (stratégie standard), "en file" et stratégie ABI (la stratégie abi[I,J] est la stratégie abi avec les indices I et J dans la dernière clause:

```
nrev([X|L1],L):- nrev(L1,L2)[I],append(L2,[X],L)[J].
```

Taille de la liste	Nombre d'unifications pour nrev en mode (i,o)				en mode(o,i)			
	pile	file	abi[1,1]	abi[10,10]	pile	file	abi[1,1]	abi[10,10]
5	21	66	258	70		139	52	242
10	66	606	198901	36550		649	140	686
15	136	2621	-			1784	262	1231
20	231	7736	-	-		3794	432	2025
30	496	36891	-	-		11439	879	4060
50	1326	273276	-	-		48229	2285	10234
n	$(n^2+3n+2)/2$	$(n^4+2n^3+23n^2+22n+24)/24$	-	-	infini	$(2n^3+15n^2+37n+24)/6$	-	$(7n^2+43n)/2 + k^*$

* dépend de n/10

Par contre, pour un but ←`nrev(L1,L2)` en mode (o,i), la stratégie standard donne un arbre de dérivation infini, alors que tout arbre équitable est fini. Avec la stratégie "en file", la taille de l'arbre de dérivation obtenu est alors proportionnelle à n^3. Lorsque le premier argument est une variable libre, il n'est plus possible d'obtenir un arbre de dérivation ne comportant qu'une seule branche. Par exemple, pour le but ←`nrev(L,[1])`, on obtient la nouvelle liste de buts ←`nrev(L1,L2),append(L2,[X1],[1])`, qui ne comporte que des buts non déterministes. Les résultats donnés dans ce tableau montrent que le choix des indices est primordial, car certains indices donnent des résultats catastrophiques: pour la stratégie abi[1,1], il est impossible d'inverser une liste de taille 20 (en mode (i,o)), car l'arbre de dérivation obtenu est beaucoup trop volumineux.

IV.3.3 Une bonne règle d'évaluation

Le tableau ci-dessus montrent que les variations de complexité des différents arbres de dérivation SLD possibles pour un but donné peuvent être très importante. Pour la stratégie ABI avec indices [1,1], l'arbre obtenu pour inverser une liste de taille 100 (en mode (i,o)) est certes fini, mais beaucoup trop volumineux pour être parcouru par un interpréteur (du moins en temps raisonnable). Il est donc important d'utiliser une règle d'évaluation adaptée. Idéalement, une bonne règle d'évaluation est une règle qui donnerait un arbre de dérivation de taille minimale. Trouver une telle règle est en général impossible. Par la suite, nous distinguons différents cas possibles selon que l'arbre de dérivation standard est fini ou non et selon que la taille d'un arbre de dérivation équitable est bornée ou non.

IV.3.3.1 Minimiser l'arbre de dérivation SLD

Dans l'absolu, une bonne règle d'évaluation devrait permettre d'obtenir l'arbre de dérivation SLD le plus petit possible. Ceci n'est généralement pas implémentable. De plus,

il n'est pas toujours possible de trouver des heuristiques ayant un rapport direct avec la taille de l'arbre SLD obtenu [NAI a]. Cependant, Naish propose deux règles de bas niveau pour minimiser la taille de l'arbre de dérivation: détecter l'échec et éviter l'échec.

• Détecter l'échec: Lorsqu'un but donne un échec, il faut sélectionner en priorité les atomes qui échouent afin d'obtenir l'échec le plus vite possible. En dehors du cas où l'un des atomes ne peut s'unifier avec aucune tête de règle, il n'est pas possible de déterminer directement quels atomes échouent. De plus il se peut qu'un but non clos échoue alors que chaque atome donne au moins un succès. Une heuristique utilisée dans MU-PROLOG pour détecter l'échec consiste à sélectionner les tests dès que possible (c'est à dire dès qu'ils sont suffisamment instanciés), les sous-buts tests étant plus susceptibles d'échouer que les autres.

• Eviter l'échec: Cette règle consiste à essayer d'éviter de créer des branches d'échec lorsqu'un but donne un succès. Dans ce cas, il est optimal de sélectionner d'abord les sous-buts localement déterministes [NAI a], c'est à dire les sous-buts qui s'unifient avec une seule tête de clause.

Dans le paragraphe suivant, nous nous intéressons à la taille maximale des arbres de dérivation équitable, afin d'évaluer la différence de complexité entre l'arbre de dérivation standard et un arbre de dérivation équitable.

IV.3.3.2 Les différents cas possibles

On considère un programme P et un but At. Nous envisageons les différents cas de figure selon que les arbres de dérivation SLD pour $P \cup \{At\}$ contiennent ou non la clause vide et qu'il existe ou non des arbres de dérivation finis. Les différents cas envisageables sont les suivants:

- il existe un arbre de succès fini
- tous les arbres sont infinis et contiennent la clause vide
- il existe un arbre d'échec fini
- tous les arbres sont infinis et ne contiennent pas la clause vide

Pour chaque cas, nous comparons l'arbre de dérivation standard et les arbres équitables.

<u>• Premier cas: Il existe un arbre de succès fini</u>

Dans ce cas tout arbre de dérivation contient la clause vide et tout arbre équitable est de plus fini (cf chapitre I). Nous distinguons deux sous-cas selon qu'il existe ou non des arbres infinis:

• Tous les arbres sont finis

Si tous les arbres de dérivation sont finis, alors il existe un nombre fini d'arbres de dérivation (car le nombre de règles est fini et le nombre d'atomes dans une règle est fini):

Proposition:

Soit P *un programme défini et* G *un but défini tel que tout arbre de dérivation SLD pour* P∪{G} *soit fini. Alors il existe un nombre fini d'arbres de dérivation SLD possibles pour P∪{G}.*

On peut construire l'arbre de toutes les dérivations possibles pour P∪{G}: les descendants d'un but L sont tous les nouveaux buts pouvant être obtenus en dérivant l'un des atomes du but avec l'une des règles de P. Chaque noeud a un nombre fini de fils car le nombre de règles de P est fini et le nombre d'atomes dans un corps de règle est également fini. La profondeur de l'arbre est finie car il n'existe pas de dérivations infinies de P∪{G}. Cet arbre est donc fini. Toute arbre de dérivation SLD de P∪{G} est contenu dans cet arbre, il existe donc un nombre fini d'arbres de dérivation SLD pour P∪{G}.

Par conséquent, il existe un arbre de taille maximale. Le choix de la règle de dérivation est donc relativement peu conséquent dans ce cas puisque le nombre de noeuds de l'arbre de dérivation obtenu est borné.

• Il existe un arbre infini

Cette situation est fréquente pour des programmes récursifs. Elle inclue les cas où la stratégie standard peut donner ou non un arbre infini selon l'ordre des sous-buts dans certaines clauses, comme par exemple pour `naïve-reverse` ou `permut` (en mode (i,o) et en mode (o,i)). Nous distinguons deux sous-cas selon que l'arbre standard est ou non fini.

- l'arbre standard est fini: tout arbre équitable est également fini, mais peut être arbitrairement grand: le choix d'une bonne règle d'évaluation est important dans ce cas. Comme le montrent les résultats obtenus avec `naive-reverse` (dans ce cas, l'arbre de dérivation SLD standard est minimal car il ne comporte aucune branche d'échec), une stratégie inadaptée peut provoquer une augmentation de complexité considérable.

- l'arbre standard est infini: l'utilisation d'une stratégie équitable permet d'éviter une boucle, mais l'arbre obtenu peut être très grand. C'est le cas par exemple pour `permut` en mode (o,i) ou `nrev` en mode (o,i).

• Deuxième cas: tous les arbres sont infinis et contiennent la clause vide

Dans ce cas, tous les arbres contiennent au moins une branche infinie, qui ne dépend pas de la stratégie de dérivation, et au moins une clause vide. Avec un interpréteur Prolog en profondeur d'abord, l'obtention ou non d'un succès dépend alors de la position relative de la première clause vide (i.e la plus à gauche) et de la première branche infinie, ce qui est directement lié à l'ordre des clauses. Considérons par exemple le programme suivant:

```
p(X):- p(X).
p(a).
```

L'arbre de dérivation SLD pour ←`p(a)` comporte une branche gauche infinie et une infinité de clauses vides à droite de cette branche infinie. Si les clauses sont inversées,

l'arbre contient alors une infinité de clauses vides à gauche de la branche infinie et un interpréteur Prolog donne un succès.

Cependant, la règle d'évaluation utilisée peut parfois avoir pour effet de modifier la position relative de la première clause vide et de la première branche infinie. L'utilisation d'une règle d'évaluation équitable peut alors faire gagner ou perdre un succès. Considérons par exemple le programme suivant:

```
p :- q(X).
q(X) :- r(X),s(X).
r(X):-t(X).
t(b):-t(b).
t(a).
s(a).
s(b).
```

Un interpréteur standard boucle pour le but ←p, car l'arbre SLD contient une branche infinie à gauche de la clause vide, alors qu'un interpréteur équitable "en file" donnerait un succès car la clause vide se trouve à gauche de la branche infinie. Bien entendu, il est également possible d'envisager un exemple similaire où un interpréteur équitable ne donnerait pas de succès alors qu'un interpréteur standard donnerait un succès.

• Troisième cas: il existe un arbre d'échec fini

Dans ce cas, tout arbre équitable est un arbre d'échec fini.

Comme dans le cas 1, la complexité d'un arbre de dérivation est bornée si tous les arbres sont finis. Par contre, s'il existe un arbre de dérivation qui est infini, alors la taille d'un arbre de dérivation équitable n'est pas bornée. On peut faire dans ce cas les mêmes remarques que dans le premier cas: le choix de la règle d'évaluation utilisée est important lorsque la taille de l'arbre de dérivation n'est pas bornée.

• Quatrième cas: tout arbre de dérivation est infini et ne contient pas la clause vide

La résolution SLD ne donne aucun résultat pour de tels buts (dans le cas de buts clos, il s'agit d'atomes At tels que At $\notin SS \cup FF$), et donc la règle d'évaluation utilisée n'a aucune importance.

Chapitre V:

Utilisation des interpréteurs équitables

V.1 Introduction

Dans le chapitre I, nous avons montré l'intérêt théorique des stratégies de dérivation équitables. Les résultats de complétude ont motivé la définition et la réalisation d'interpréteurs Prolog équitables. Nos interpréteurs permettent de contrôler la règle d'évaluation et sont effectivement utilisables. Les implémentations réalisées montrent que le coût de l'équité n'est pas rédhibitoire.

Ce chapitre est consacré à l'utilisation pratique des interpréteurs équitables Pile-indicée et Arbres de Buts Indicés. Les stratégies ABI et PI ont été définies dans le but d'obtenir des interpréteurs équitables. Notre objectif dans ce chapitre est d'étudier comment utiliser au mieux les mécanismes introduits pour apporter l'équité.

La stratégie Pile-Indicée permet de garantir l'équité à faible coût, par exemple en choisissant les indices de dérivation de telle sorte que la règle d'évaluation standard ne soit modifiée qu'au delà d'une certaine profondeur de l'arbre de dérivation.

La stratégie ABI offre un mécanisme d'évaluation plus souple: l'indice de dérivation d'un sous-but correspond au nombre de dérivations consécutives de ce sous-but par rapport à ses sous-buts frères. Des exemples, illustrant différents types d'utilisations possibles de la stratégie ABI, sont décrits.

V.2 Utilisation de la stratégie Pile-Indicée

V.2.1 Contrôle avec la stratégie Pile-indicée

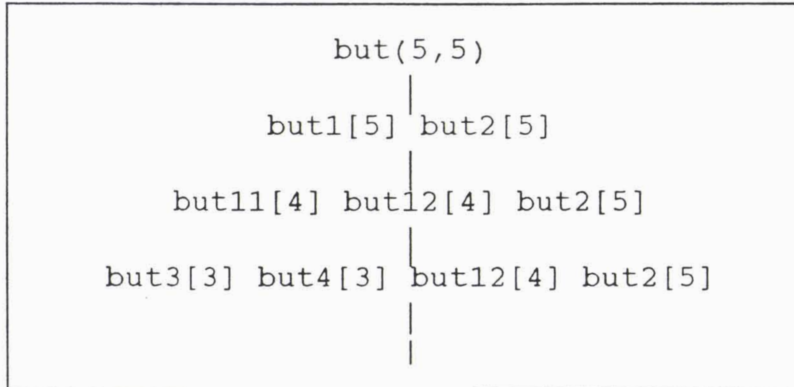
La stratégie Pile-Indicée permet de modifier l'ordre d'évaluation en indiceant des sous-buts. Cependant, lorsqu'on utilise des indices de dérivation provoquant effectivement un ordre d'évaluation différent, certains problèmes apparaissent. L'indice affecté à un sous-but permet de borner le nombre de dérivations consécutives consacrées à ce sous-but avant d'ajouter les nouveaux sous-buts en queue. Néanmoins, cet indice n'indique pas exactement le nombre de dérivations consécutives consacrées à ce sous-but, car ce nombre dépend également des indices des sous-buts descendants. Dans certains cas de figure, ceci est susceptible d'engendrer un comportement opérationnel non satisfaisant, ce que nous illustrons par l'exemple ci-dessous:

Considérons le programme indicé suivant:

```
but(i1,i2):- but1[i1], but2[i2].
but1 :- but11[5], but12[5].
but2 :- but21[10], but22[10].
but11 :- but3[10], but4[10].
...
```

Ce petit programme met en évidence les problèmes rencontrés avec la stratégie "pile-indicée" lorsqu'on utilise des indices de dérivation pour obtenir un contrôle particulier. Les indices `i1` et `i2` assurent un traitement équitable de `but1` et `but2`.

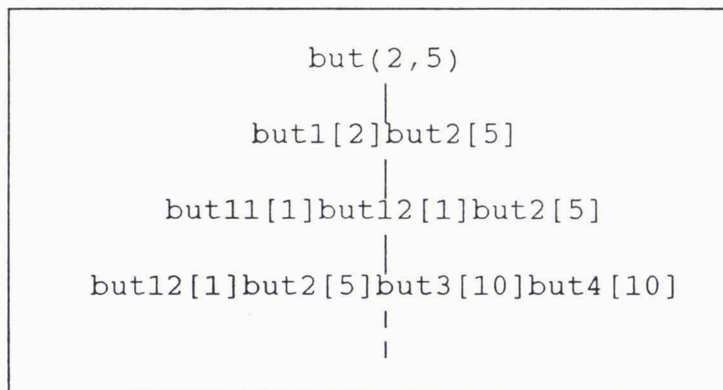
Considérons la dérivation obtenue pour `but(5,5)`:



Dans cette dérivation, l'atome `but3` (ainsi que `but4`) est ajouté en tête de la résolvante avec un indice de dérivation plus faible que son indice de dérivation dans le programme.

Le nombre de dérivations qui vont être consacrées à `but3` avant de sélectionner `but4` est borné par l'indice 3 et non par l'indice de `but3` dans la clause, qui est égal à 10. Ceci peut engendrer des calculs inefficaces, par exemple si `but4` consomme une variable produite par `but3`, et dont la production nécessite dix dérivations.

Considérons maintenant la dérivation obtenue pour `but(2,5)`:



Dans ce cas, lorsque l'atome but3 sera sélectionné, il bénéficiera d'un nombre de dérivations consécutives borné par l'indice 10, alors que l'indice de son but père, `but11` dans le programme est égal à 5: l'indice du sous-but `but11` ne permet pas de contrôler la dérivation de `but3`.

Pour ce programme, il n'est pas possible de choisir les indices de manière à effectuer une dérivation alternée de `but1` et `but2`, par exemple effectuer alternativement n dérivations des atomes descendant de `but1` et n dérivations des atomes descendant de `but2`. Le type de contrôle obtenu n'est pas satisfaisant dans la mesure où le nombre de dérivations

relatives d'un sous-but par rapport aux sous-buts issus de la même clause ne correspond pas toujours aux indices fixés dans le programme.

L'utilisation de la stratégie Pile-Indicée pour mettre en place des mécanismes de contrôle différents n'est pas très concluante: une telle utilisation est susceptible d'engendrer des dérivations très inefficaces et surtout un comportement procédural difficilement compréhensible pour le programmeur.

V.2.2 Utilisation "standard" de Pile-Indicée

Le mécanisme de la stratégie Pile-Indicée n'est pas assez souple pour permettre un contrôle satisfaisant. Par contre, la stratégie pile-indicée permet d'apporter l'équité en modifiant le moins possible la règle d'évaluation de Prolog standard: Si chaque sous-but a un indice égal à N, l'arbre de dérivation effectivement envisagé coïncide avec l'arbre standard au moins jusqu'à la profondeur N.

En affectant automatiquement à chaque sous-but un indice égal à N, on obtient une stratégie équitable, mais dont le comportement procédural sera identique à la stratégie standard pour toutes les dérivations de longueur inférieure à N.

Le choix de N, qui fixe la profondeur limite, doit résulter d'un compromis:

- Plus N est élevé, plus la stratégie est proche de la stratégie standard

- Plus l'indice est faible, plus les boucles liées à la règle d'évaluation sont susceptibles d'être détectées rapidement.

V.3 Utilisation de la stratégie ABI

V.3.1 Introduction

Contrairement à la stratégie Pile-Indicée, la stratégie A.B.I, basée sur une structure plus complexe, permet, de manière simple, de mettre en place des mécanismes de contrôle. Les exemples décrits ci-après illustrent les possibilités d'utilisation de la stratégie ABI.

Nous étudions trois exemples différents d'utilisation de la stratégie ABI. Le premier exemple est le programme des N-reines, qui est un exemple type de programme fonctionnant avec la règle d'évaluation standard, mais de manière inefficace. Bien que les stratégies équitables ne soient pas conçues pour optimiser des programmes de ce type, l'utilisation d'indices permet de réduire le nombre d'unifications réalisées pour résoudre le problème des huit-reines.

Le deuxième exemple est un exemple de programme de recherche de chemin dans un graphe, écrit de manière naïve, qui peut donner des dérivations infinies pour certains buts en Prolog standard alors qu'une stratégie équitable donne un arbre fini pour ces buts.

Le troisième exemple est un exemple d'utilisation "exotique" de la stratégie ABI qui consiste à utiliser un indice de dérivation pour limiter la profondeur de l'arbre de dérivation. Cet exemple est un programme qui exploite directement le mécanisme de la stratégie ABI et ne peut pas fonctionner avec une autre règle d'évaluation.

V.3.2 Les 8-reines avec la stratégie ABI

Nous envisageons tout d'abord le programme des huit-reines, fréquemment cité pour illustrer les mécanismes de contrôle:

```
solution(Ln,Sol) :-
        permut(Ln,Sol),
        correcte(Sol) .
permut([],[]).
permut([X|L],[Y|LL]):-
        effacer([X|L],Y,L2),permut(L2,LL).
effacer([X|L],X,L).
effacer([X|L],Y,[X|LL]) :-
        effacer(L,Y,LL).
correcte([]).
correcte([X|L]):-
        n_attaque_pas(X,L,1),
        correcte(L).
n_attaque_pas(X,[],_).
```

```
n_attaque_pas(X,[Y|L],D):-
     =\=(D,X-Y), =\=(D,Y-X), D1 is D+1,
      n_attaque_pas(X,L,D1).
```

Ce programme est un exemple type de programme de la forme générer/tester: un prédicat non déterministe calcule successivement toutes les solutions possibles, qui sont testées par un autre prédicat. Il s'agit d'un type de programme qui peut être très inefficace lorsque de nombreuses solutions incorrectes sont générées avant de trouver la bonne.

L'exécution de programmes de ce type peut être considérablement améliorée en utilisant un contrôle dirigé par les données, c'est à dire permettant de tester chaque solution potentielle au fur et à mesure. Nous étudions comment améliorer l'exécution du programme des N-reines naïf avec la stratégie ABI. La stratégie ABI ne permet pas d'effectuer un contrôle en fonction de l'état d'instanciation des variables, mais simplement de contrôler le nombre de dérivations consécutives d'un sous-but.

On peut obtenir un développement alterné du générateur et du test très simplement en mettant des indices dans la première clause :

```
solution(Ln,Sol) :-
        permut(Ln,Sol)[I],
        correcte(Sol) [J].
```

Pour minimiser le nombre de tests effectués, la valeur de I doit correspondre au nombre de dérivations nécessaires pour produire un nouvel élément de Sol (*n1*) et J doit être égal au nombre de dérivations nécessaires pour tester la partie de la liste déjà instanciée (*n2*). Cependant, *n1* et *n2* varient: *n1* est égal à 2 lors du calcul de la première permutation, puis dépend ensuite du "déplacement" de chaque élément, *n2* dépend de la longueur de la liste. Il n'est donc pas possible de déterminer un jeu d'indice donnant un contrôle optimal.

Pour certaines valeurs de *I* et *J*, les sous-buts système =\=(D,X-Y), =\=(D,Y-X) peuvent être sélectionnés trop tôt, c'est à dire avant que les variables soient instanciées. Pour éviter ce problème, il est nécessaire de suspendre ces prédicats jusqu'à instanciation de la variable Y. Ceci est réalisé en utilisant la primitive de suspension `suspendre/2` (disponible dans la version Prolog):

```
n_attaque_pas(X,[Y|L],D):-
     suspendre([Y],=\=(D,X-Y)),
     suspendre([Y],=\=(D,Y-X),
     is(D1,D+1),n_attaque_pas(X,L,D1).
```

Le tableau suivant donne le nombre d'unifications obtenues pour différentes valeurs de I et J. En choisissant la même valeur pour I et J, le nombre d'unifications nécessaires pour résoudre les huit-reines est réduit approximativement de moitié par rapport à la stratégie standard. Le gain n'est pas énorme, mais est obtenu de manière très simple, en mettant des indices uniquement dans la première règle. Aucun calcul supplémentaire n'est nécessaire

au moment de l'exécution pour déterminer l'ordre d'évaluation, l'amélioration du contrôle est obtenue uniquement en utilisant le mécanisme de la règle d'évaluation.

N	stratégie standard	abi[1,1]	abi[2,2]	abi[2,1]	abi[4,4]
4	213	164	164	170	149
6	4269	2321	2341	2870	2282
8	87592	43958	44135	58481	43089

Nombre d'unifications pour le problème des N-reines (1ère solution)

Bien entendu, la façon la plus efficace de résoudre le problème des 8-reines consiste à transformer le programme "à la main" pour pouvoir effectuer les test au fur et à mesure, ce qui donne le programme suivant, habituellement appelé version "intelligente":

```
sol(Ln,S) :-
        reines(Ln,[],S).

reines([],S,S):- !.
reines(Reste,Solpar,S):-
        effacer(Reste,X,Nreste),
        n_attaque_pas(X,Solpar,1),
        reines(Nreste,[X|Solpar],S).
```

Cependant, la version précédente, qualifiée de "naive", est beaucoup plus simple et constitue une formulation du problème logique et déclarative.

V.3.3 Exemple 2

Ce programme (d'après un exemple de [VER]), recherche un chemin dans un graphe, représenté par une suite de faits de la forme arc(S1,S2). Il s'agit également d'un programme de la forme générer/tester, mais dans ce cas le générateur peut boucler: avec la stratégie standard, le sous-but `chemin` boucle dès qu'il y a un cycle, alors que le sous-but `sanscycle` permet de détecter les cycles: Avec une stratégie équitable, ce programme ne boucle donc pas.

```
bonchemin(B,E,Chemin) :-
     chemin(B,E,Chemin) [1],
     sanscycle(Chemin) [1].
chemin(B,E,[B,E]) :-
     arc(B,E).
```

```
    chemin(B,E,[B|P]):-
          arc(B,I),
          chemin(I,E,P).
    sanscycle([N]).
    sanscycle([N1,N2|Ns]):-
          nappartientpas(N1,[N2|Ns]),
          sanscycle([N2|Ns]).
    nappartientpas(_,[]).
    nappartientpas(N,[N1|Ns]):-
          \==(N,N1),
          nappartientpas(N,Ns).
arc(a,b).
arc(b,a).
arc(a,c).
```

Avec une stratégie équitable, ce programme ne boucle donc pas. Avec la stratégie ABI, les indices dans la première clause permettent d'alterner le développement des deux sous-buts et d'obtenir un comportement correct du programme. Par exemple, pour le but ←`bonchemin(a,b,C)`, on obtient un succès avec `C=[a,b,c]`. Bien entendu, n'importe quelle règle d'évaluation équitable et donc n'importe quel choix d'indices permet d'éviter les boucles (à condition que le sous-but \==(X,Y) soit suspendu lorsque l'une des variables n'est pas instanciée).

V.3.4 Une utilisation particulière de la stratégie ABI

L'exemple suivant est un exemple d'utilisation particulière du mécanisme de la stratégie ABI: l'indice de dérivation d'un sous-but limite le nombre de dérivation consécutives de ce sous-but, on peut donc utiliser un indice pour limiter la longueur de la dérivation d'un sous-but. Le programme ci-dessous est un interpréteur à profondeur limitée: `prof(X,N)` effectue la dérivation du but X en limitant la profondeur de l'arbre de dérivation à N.

```
prof(N,X):-
      is(M,N+1),
      resoudre(X,Q)[M],
      succes_ou_echec(Q).
resoudre(X,Q):-
      X ,
      lier(Q) [fix].
```

```
lier(aa).

succes_ou_echec(X) :-
     var(X),write('échec '),fail.
succes_ou_echec(X) :-
     atomic(X).
```

Interpréteur à profondeur limitée

Le sous-but `resoudre(X,Q)` est dérivé pendant au plus N+1 dérivations, X est donc dérivé pendant au plus N dérivations. Le second argument de `resoudre` est une variable libre (Q) qui est utilisée pour indiquer si le sous-but X donne un succès ou non: lorsque X donne un succès, la variable Q est liée. Le prédicat `succès_ou_echec` échoue en affichant "`echec`" lorsque X a été développé jusqu'à la profondeur N sans s'effacer et réussit si X a donné un succès, c'est à dire si la variable X a été liée. Voici un exemple d'utilisation de ce programme avec `naïve-reverse` (sans indices):

```
?-prof(65,nrev([1,2,3,4,5,6,7,8,9,10],L)).
echec
no
 ?-prof(66,nrev([1,2,3,4,5,6,7,8,9,10],L)).
L = [10,9,8,7,6,5,4,3,2,1]
?-prof(10,nrev(L,[1,2,3])).
L = [3,2,1] ;
echec echec echec echec echec echec echec
no
```

Pour inverser une liste de taille 10, le nombre d'unifications nécessaires est égal à 66 et l'arbre de dérivation ne comporte qu'une seule branche. On obtient un échec provoqué par la limitation de la profondeur pour N inférieur à 66.

Dans le dernier cas, l'arbre de dérivation est infini. L'affichage indique que sept branches de l'arbre de longueur supérieure à 10 n'ont pas été explorées complètement.

CONCLUSION

Les résultats de complétude sur les arbres de dérivation équitables montrent l'intérêt des stratégies de dérivation équitables: l'usage d'une règle d'évaluation équitable permet d'éviter toutes les boucles liées à la définition de l'interpréteur. De plus, un interpréteur équitable est une implémentation correcte et complète de la règle de négation par l'échec (pour les programmes définis,[LL0]).

Cependant, l'utilisation en pratique de règles d'évaluation équitables est généralement considérée comme trop coûteuse et inadaptée aux programmes Prolog usuels, ce qui est effectivement le cas pour la règle d'évaluation "en file".

Nos travaux montrent que l'on peut envisager et implémenter des modèles de stratégies équitables effectivement utilisables. Nos interpréteurs équitables permettent d'utiliser sans problèmes les prédicats prédéfinis usuels. Par ailleurs, il est possible de ne modifier la règle d'évaluation que pour certaines parties du programme.

L'utilisation de la machine MALI nous a permis de réaliser une implémentation efficace de nos stratégies, en particulier pour le modèle Pile-indicée. La stratégie ABI, plus complexe, est de toutes façons plus coûteuse à implémenter. Cependant, l'implémentation réalisée à partir de PrologII/MALI n'est pas optimisée (en particulier elle ne prend pas en compte le fait que certains sous-buts peuvent ne pas être indicés), aussi nous pensons qu'une implémentation plus efficace est possible. Une implémentation utilisant en partie les instructions de la WAM est en projet (dans le cadre du projet ORGANON*).

L'intérêt d'une stratégie équitable est indiscutable dans certains cas, cependant l'usage systématique de l'équité n'est pas nécessaire, et modifier la règle d'évaluation peut être parfois une source d'inefficacité, en particulier lorsque la règle d'évaluation standard donne une dérivation optimale (pas de branches d'échec). Il pourrait être intéressant d'essayer de caractériser les buts pour lesquels l'équité est effectivement nécessaire, c'est-à-dire les buts pour lesquels l'arbre de dérivation SLD standard est infini alors qu'il existe un arbre de dérivation fini.

Nos stratégies équitables sont paramétrées par des indices de dérivation, ce qui introduit un moyen de contrôle nouveau. En particulier, la stratégie ABI permet de définir des mécanismes de coroutines simples entre des sous-buts. Ceci permet parfois d'améliorer l'exécution de programmes inefficaces avec la règle d'évaluation standard, mais l'ordre d'évaluation obtenu est moins efficace - en terme de taille de l'arbre de dérivation SLD envisagé - qu'avec des méthodes prenant en compte l'état d'instanciation des variables. Cependant, si le contrôle obtenu n'est pas optimal, il n'est pas coûteux dans la mesure où il consiste uniquement à exploiter le mécanisme de l'interpréteur. Pour obtenir un contrôle "idéal", il serait intéressant de combiner l'utilisation d'une règle d'évaluation équitable

*. Opération financée par le GRECO de Programmation

avec des techniques de contrôle dynamique, telles que celles proposées dans MU-PROLOG ([NAI b]).

Il serait également intéressant d'étudier dans quelle mesure certaines techniques d'optimisation statique pourraient être utilisées sous l'hypothèse d'une règle d'évaluation non standard.

Quoiqu'il en soit, l'usage d'une règle équitable autorise une programmation plus déclarative: des programmes écrits de manière simple peuvent fonctionner avec une règle équitable, alors qu'ils ne fonctionnent pas tels quels en Prolog standard. Notre travail montre que l'on peut raisonnablement étendre Prolog standard pour y intégrer des mécanismes d'équité.

BIBLIOGRAPHIE

[APT a] Apt K.R., *Introduction to Logic Programming*, Technical report TR-87-35, Dept of Computer Science, University of Texas at Austin, 1987.

[APT b] Apt K.R, *On the safe termination of Prolog programs*, Proc of the sixth Internatinal Conference on Logic Programming, MIT Press, Lisboa, 1989.

[A, B, W] Apt K. R., Blair H., and Walker A., *Toward a theory of declarative knowledge*, in Foundations of Deductive Databases and Logic Programming, J. Minker (ed), Morgan Kaufman, Los Altos, 1988, pp 89-148.

[A, VE] Apt K.R. and Van Emden M.H., *Contribution to the theory of logic programming*, Journal of the Association for Computing Machinery, vol 29-3, 1982, pp 841-862.

[B, M] Barbuti R. and Martelli M., *Completeness of the SLDNF-Resolution for a Class of Logic Programs*, Proceedings of the third International Conference on Logic Programming, London, 1986, pp. 600-614.

[BEK a] Bekkers Y., Canet B., Ridoux O., Ungaro O.: MALI : *A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages*, Proc. of the 3rd Symposium on Logic Programming, IEEE, Salt-Lake City, USA, 1986.

[BEK b] Bekkers Y, *Une contrainte de négation par l'échec*, Séminaire de Programmation en Logique de Trégastel, CNET, 1990.

[BRU] Bruynooghe M, De Schreye D, Krekels B, *Compiling Control*, Journal of Logic Programming, vol 6-1, 1989, pp 135-162.

[C, C] Coelho H. and Cotta J. C., *Prolog by Example*, Spinger Verlag, New-York, 1988.

[C, L] Cavedon L. and Lloyd J. W., *A completeness theorem for SLDNF resolution*, Journal of Logic Progamming, vol , 1990.

[CLA a] Clark K. L., *Negation as failure*, in Logic and Data Bases, Gallaire H. and Minker J. (eds), Plenum Press, New York, 1973.

[CLA b] Clark K. L., *IC-PROLOG languages features*, in Logic Programming, Clark and Tarlund (eds), Academic Press (1982).

[COD] Codognet P., *Backtraking intelligent en Programmation Logique: Un cadre général*, Thèse nouveau régime, Université de Bordeaux, 1989.

[COL a] Colmerauer A., Kanoui H., Roussel P., Pasero R., *Un Système de Communication Homme-Machine en Français*, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[COL b] Colmerauer A., PrologII: *Manuel de référence et modèle théorique*, Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 1982.

[DEL] Delahaye J. P., *Sémantique logique et dénotationnelle des interpréteurs Prolog*, Informatique théorique et applications, vol 22,1 , 1988, pp 3-42.

[DEN] Denis F., *Contribution à l'étude des sémantiques axiomatiques de Prolog*, Thèse, Université de Lille I, 1990.

[D, P] Delahaye J.P et Paradinas P., *Stratégies équitables en programmation logique*, Actes du Séminaire de Programmation en Logique, Trégastel, 1986.

[ELC] Elcock E. W., *The pragmatics of Prolog: some comments*, Proceedings of Workshop on Logic Programming, Algarve, Portugal, 1983.

[GIA] Giannesini F., Kanoui H., Pasero R., Van Caneghem M., *Prolog*, Interéditions, 1985.

[G, L] Gallaire H. et Lasserre C., *Metalevel Control for Logic Programs*, in "Logic Programming", Clark and Tarlund (eds), 1982.

[HIL] Hill R., *LUSH resolution and its completeness*, DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[J, L, L] Jaffar J., Lassez J.L. and Lloyd J.W., *Completeness of the negation as failure rule*, 8th IJCAI, Karlsruhe, 1983, pp 500-506.

[J, L, M] Jaffar J., Lassez J.L. and Maher M.J., *Some issues and trends in the semantics of logic programming*, Proceedings of the third International Conference on Logic Programming, London, LNCS 225, 1986, pp 223-240.

[J, L] Janot S. and Le Huitouze S., *Definition of Fair Prolog interpreters and implementation with MALI*, research report IT-195,LIFL, Université de Lille I, 1990.

[KOW] Kowalski R. A., *Predicate as a Logic Programming language*, Information Processing 74, Stockholm, North Holland, 1974.

[KUN] Kunen K. , *Signed data dependencies in Logic Programs*, Journal of Logic Programming , 7, 1989, pp 231-245.

[L, M] Lassez J.L. and Maher M.J., *Closure and fairness in the semantics of logic programming*, Theoretical Computer Science 29, 1984.

[LEH a] Le Huitouze S, *Mise en oeuvre de PrologII/MALI*,Thèse, Université de Rennes I, 1988.

[LEH b] Le Huitouze S, *A new data structure for implementing extensions to Prolog*, Proc. of 2nd Workshop on Programming Language Implementation and Logic Programming, Linkoping, Sweden, 1990.

[LLO] Lloyd J.W., *Foundations of logic programming*, 2nd edition, Springer Verlag, 1987.

[NAI a] Naish L., *Automating control for logic programs*, Journal of Logic Programming, vol 2-3 ,1985, pp 167-183.

[NAI b] Naish L., *Negation and control in Prolog*, Springer Verlag, 1986.

[NAI c] Naish L., *Negation and quantifiers in NU-PROLOG*, Proceedings of the third International Conference on Logic Programming, London, LNCS 225, 1986.

[POR a] Porto A., EPILOG: *A language for Extended Programming in Logic*, Proc. of the first International Logic Programming Conference, Marseille, 1982.

[POR b] Porto A., *Two-level PROLOG*, Proc. of International Conference on Fith Generation Computer Systems, 1984.

[QUI] *Quintus Prolog Reference manual*. Quintus Computer Systems.

[RID] Ridoux O.,*Gestion de mémoire temps-réel dans les langages de programmation relationnelle*. Thèse, Université de Rennes I, 1986.

[ROB] Robinson J. A., *A Machine-oriented Logic Based on the Resolution Principle*, Journal of the Association for Computing Machinery 12-1, 1965.

[ROU] Roussel P., *PROLOG: Manuel de référence et d'utilisation*, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, 1975.

[SHE a] Shepherdson J. C. *Negation as failure: A comparison of Clark's Completed Data Base and Reiter's Closed World Assumption*, Journal of Logic Programming, vol 1,1 1984, pp 51-79.

[SHE b] Shepherdson, J. C., *Negation as failure II*, Journal of Logic Programming, vol 2,3, 1985, pp 185-202.

[SHE c] Shepherdson J.C., *Negation in Logic Programming*, in Foundations of Deductive Databases and Logic Programming, J. Minker (ed), Morgan Kaufmann, Los ALtos, 1988, pp 19-88.

[SHE d] Shepherdson J.C., *Unsolvable problems for SLDNF resolution*, Journal of Logic Programming, vol 10, 1991, pp 19-22.

[S, S] Sterling L., Shapiro E., *The art of PROLOG*, The MIT Press, 1986

[SIC] Carlsson M. and Wildén J., *SICStus PROLOG user's manual,* Swedish Institute of Computer Science, 1990.

[VER] Verschaetse K, De Schreye D, Bruynooghe M, *Generation And Compilation Of Efficient Computation Rules*, Proc of the Seventh International Conference on Logic Programming, Jerusalem, 1990.

[WAR] Warren D.H.D. *An abstract Prolog Instruction Set* , Technical note 309, SRI International, 1983.

BIBLIOTHEQUE INTERUNIVERSITAIRE
SECTION
DE
SCIENCES
LILLE

FAIR COMPUTATION RULES IN LOGIC PROGRAMMING

Abstract

Prolog interpreters are not complete in general for both success and finite failure, because they use a depth-first left-to-right computation rule and a depth-first search of the SLD-tree. An interpreter using a fair computation rule (each subgoal which appears in an infinite derivation is actually selected) is complete for finite failure and for finite success: If there exists a finite SLD-tree for a program P and a goal G, then every fair SLD-tree for P and G is finite.

We propose two fair computation rules, based on the use of derivation indices. An efficient implementation of these computation rules has been realized. We took into account the problems raised by the use of a non standard computation rule, and our fair interpreters can actually be used.

Keywords

PROLOG
LOGIC PROGRAMMING
DEFINITE PROGRAM
FINITE FAILURE
STANDARD PROLOG INTERPRETER
FAIR INTERPRETER
COMPUTATION RULE
FAIR DERIVATION

PPN 036117110

Résumé

Les interpréteurs Prolog ne sont pas complets en général pour les échecs et pour les succès car ils sont basés sur un parcours de l'arbre de résolution en profondeur d'abord associé à une gestion des buts en pile. Cependant, la résolution SLD sur laquelle Prolog est basé est correcte et complète. L'utilisation d'une règle d'évaluation équitable - tout sous-but qui apparait dans une dérivation infinie est effectivement sélectionné- permet d'obtenir tous les échecs finis et tous les succès finis d'un programme: s'il existe un arbre de dérivation SLD fini pour un programme P et un but B, alors tout arbre de dérivation équitable pour P et B est fini. Un interpréteur Prolog équitable permet donc d'éviter toutes les boucles liées à l'ordre d'évaluation des sous-buts. Nous étudions l'utilisation de règles d'évaluation équitables en Prolog, des aspects théoriques aux aspects pratiques.

Deux modèles de règles d'évaluation équitables, Pile-indicée et Arbres-de-buts-indicés sont proposés. Ces stratégies de dérivation sont basées sur l'utilisation d'indices de dérivation pour controler l'ordre d'évaluation des sous-buts. Des implémentations efficaces de ces règles ont été réalisées. Les problèmes pratiques liés à l'utilisation d'un ordre d'évaluation différent sont pris en compte et nos interpréteurs équitables sont effectivement utilisables.

Mots-Clés

PROLOG
PROGRAMMATION LOGIQUE
PROGRAMME DEFINI
ECHECS FINIS
INTERPRETEUR PROLOG STANDARD
INTERPRETEUR EQUITABLE
REGLE D'EVALUATION
DERIVATION EQUITABLE