USTL

LIFL Laboratoire d'Informatique Fondamentale de Lille

Numéro d'ordre : 1246

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Anne PARRAIN

TRANSFORMATIONS DE PROGRAMMES LOGIQUES ET SEMANTIQUE OPERATIONNELLE

Thèse à soutenir le 4 Février 1994 , devant la commission d'examen composée de

Président	Sophie Tison
Rapporteurs	Gérard Ferrand
	Laurent Fribourg
Examinateurs	Jean-Paul Delahaye
	Philippe Devienne
	Gilberto Filè
	Patrick Lebègue
	Baudouin Le Charlier

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.4[illegible].24 Fax. 20.43.65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

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

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

PROFESSEUR EMERITE

M. A. LEBRUN

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

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
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. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Merci à ...

Philippe Devienne et Patrick Lebègue qui m'ont encadrée en DEA puis en thèse. Leur disponibilité, leurs idées, leurs conseils, leur soutien et leur attention constante m'ont permis de faire aboutir cette thèse.

Gérard Ferrand et Laurent Fribourg qui ont accepté d'être rapporteurs de ma thèse. L'intérêt qu'ils ont montré pour mes travaux par leur lecture minutieuse et leurs remarques pertinentes sont pour moi un grand encouragement.

Jean-Paul Delahaye, qui anime l'équipe Metheol au sein du LIFL.

Sophie Tison, qui me fait l'honneur de présider ce jury.

Baudouin Le Charlier et Gilberto Filè qui ont accepté de faire partie de mon jury.

Je tiens à remercier également les copains et collègues du labo (qu'ils soient proches ou éloignés) pour toutes les discussions, pauses-café et autres pots qui ont rythmé mes bientôt quatre ans de présence ici. Mention spéciale pour Eric, Philippe, Stéphane et bien sûr Christophe qui partagent avec moi le bureau 308.

Je garde une pensée affectueuse pour ma famille, pour sa patience et son soutien.

Merci surtout à Marc.

Table des matières

Introduction

Les opérations classiques de transformations de programmes logiques (pliage, dépliage, ...) préservent la sémantique des programmes au sens du plus petit modèle de Herbrand ([TS84]) et au sens de l'ensemble des substitutions-réponses ([KK88]). Ces transformations ne peuvent pas être appliquées directement à des programmes écrits en Prolog complet (avec des effets de bord), car elles ne préservent pas l'équivalence opérationnelle des programmes.

Nous avons donc défini une équivalence opérationnelle associée au comportement "observable" des programmes. Cette équivalence nous permet de traiter les programmes réels. Nous avons ensuite étudié les transformations préservant une telle équivalence. Puis notre travail s'est divisé en deux parties:

- l'optimisation de programmes Prolog complet, en les manipulant par nos transformations et en utilisant des informations sur les programmes provenant d'un analyseur statique;

- l'utilisation des transformations de programmes comme outils de preuve pour la validation de méta-programmes.

Équivalence opérationnelle pour Prolog avec effets de bord L'idée principale est de garder une équivalence entre programmes prolog au sens des interpréteurs standards (non-équitables, en profondeur d'abord et de gauche à droite). Deux programmes sont dits équivalents s'ils ont les mêmes solutions, dans le même ordre, jusqu'à la première branche infinie. Cela étend les sémantiques logiques et dénotationnelles pour les interpréteurs standards.

Pour obtenir une telle équivalence, nous devons définir une sémantique opérationnelle pour les effets de bord. Nous considérons que les lectures et les écritures se font sur un ruban infini d'un côté, et qu'il y a un ruban pour chaque source (entrée/sortie standards ou sur fichiers). La tête de lecture est accessible en exclusion mutuelle pour l'utilisateur et le programme, et elle se déplace après chaque lecture ou écriture, toujours vers le côté infini. Deux programmes équivalents doivent avoir les mêmes rubans à la fin de leur exécution.

Nous étudions d'abord les transformations de programmes classiques et les modifications qui doivent leur être apportées pour qu'elles préservent l'équivalence opérationnelle forte. Puis nous élargissons ce travail, d'une part en

étudiant de nouvelles transformations proposées dans la littérature ([GB90]), d'autre part, en étudiant la manière d'inclure des informations de typage à l'intérieur d'un programme Prolog.

Vers une évaluation partielle guidée par interprétation abstraite L'idée est de créer un système qui allie transformations de programmes (évaluation partielle proprement dite) et analyse statique. Le système consiste en trois étapes : l'élimination des prédicats prédéfinis avant l'analyse statique; l'analyse statique (cette phase n'est pas traitée dans ma thèse); l'incorporation des informations provenant de l'analyseur dans le programme Prolog complet initial et les transformations sur ce programme :

- *Élimination des effets de bord*

  Si on considère Prolog complet, les propriétés de terminaison, déterminisme et l'ensemble des succès ne sont pas préservées par instanciation.

  Nous voulons donc en même temps éliminer les prédicats à effets de bord pour fournir à l'interprétation abstraite des programmes écrits en Prolog pur, et préserver certaines propriétés qui sont calculées par l'interprétation abstraite.

  Nous avons restreint notre étude aux `cut`, `read`, `write`, `assert` et `retract`, car :

  1. beaucoup de prédicats prédéfinis comme `bagof`, `if-then`, `if-then-else` peuvent être simulés à partir du `cut`, du `assert` et du `retract` (comme la négation par l'échec);
  2. les prédicats comme `var`, `functor`, ... sont déterministes et leur comportement est similaire, à notre sens, au comportement du prédicat `write`.

  Deux transformations sont appliquées. La première ajoute des faits au programme : `read(X)`, `write(X)`, `cut` et `retract(rule)` pour chaque règle du programme, ce qui annule les effets de bord de ces prédicats. La seconde ajoute au programme toutes les règles qui font l'objet d'un assert. Ces deux transformations suffisent à garder nos propriétés.

- *Transformation de programmes spécifiés*

  Notre système n'est pas conçu pour une interprétation abstraite particulière, mais est générique. Nous avons donc défini les propriétés minimales requises sur les informations fournies pour qu'elles puissent être intégrées à notre système. Ces propriétés (comportant la propriété de *solution compactness*) étant proches de celles décrites dans le cadre de CLP par Jaffar et Lassez ([JL86]), nous avons étendu notre cadre à celui de la programmation logique avec contraintes.

Les informations sont manipulées comme des contraintes. Elles ne correspondent pas à des conditions requises pour obtenir un succès, mais à des conditions qui sont vérifiées pendant l'exécution.

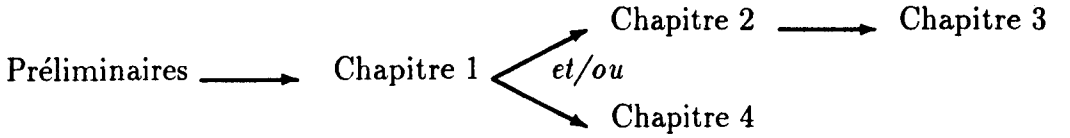
Dans un programme spécifié, un littéral a plusieurs schémas d'appels. Certains de ces schémas peuvent être déterministes. Pour appliquer certaines de nos transformations, comme le dépliage déterministe, une règle doit être partitionnée en deux : une règle avec les schémas d'appels déterministes, l'autre avec les non-déterministes. Ainsi, pendant l'étape des transformations de programmes, certaines règles sont dupliquées, et spécialisées pour certains schémas d'appels : c'est une sorte de typage de règle.

Une implémentation de notre système est en cours. Elle est réalisée par rapport à l'interprétation abstraite proposée par Christophe Lecoutre [LDL92b, Lec94].

Méta-interprétation Les transformations de programmes que nous avons étudiées nous ont permis de valider le vanilla méta-interpréteur et un méta-interpréteur quasi-itératif constitué de deux clauses binaires et d'un fait clos. Par des codages de ce méta-interpréteur, nous obtenons des résultats d'indécidabilité de l'arrêt et du vide pour des classes simples de programmes logiques.

Plan de la thèse

La thèse s'articulant autour de la notion d'équivalence opérationnelle forte (e.o.f.) et des transformations la préservant, j'ai opté pour une présentation séparée des deux axes qui en découlent. Le noyau central est développé dans le chapitre 1, son utilisation pour l'optimisation de programmes est contenue dans les chapitres 2 et 3, tandis que l'utilisation des transformations de programmes comme outils de preuve dans le cas des méta-interpréteurs est présentée dans le chapitre 4. J'ai donc essayé autant que possible de permettre une lecture distincte des deux axes :



Préliminaires Les préliminaires sont une présentation rapide et informelle des bases de la programmation logique. Cette partie nous sert surtout à fixer de manière précise les notations et le vocabulaire que nous emploierons par la suite.

Une équivalence pour les programmes logiques Le premier chapitre contient le "noyau central" autour duquel s'est développée la thèse. Nous présentons l'équivalence opérationnelle que nous considérons, et les transformations de programmes qui la préservent. L'équivalence est basée sur la conservation de la structure de l'arbre SLD d'un programme construit avec la règle de choix standard, et les transformations que nous étudions sont principalement des restrictions sur les techniques bien connues de pliage/dépliage. Nous étudions également une variante d'une technique proposée par Gallagher et Bruynooghe ([GB90]) et nous montrons qu'il s'agit en fait d'une utilisation séquentielle des opérations de pliage, dépliage et définition.

Transformations de programmes logiques Le second chapitre est une présentation du cadre dans lequel se situe notre approche des transformations de programmes logiques. Dans ce chapitre, nous essayons de dégager l'état actuel de la recherche dans ce domaine, ainsi que les nombreux problèmes qui restent posés. Nous nous intéressons d'abord aux travaux portant sur la correction des transformations, puis aux heuristiques d'optimisation de programmes.

Vers une évaluation partielle guidée par interprétation abstraite Le troisième chapitre présente notre approche de l'évaluation partielle de programmes Prolog complet, i.e. avec effets de bord. Les prédicats prédéfinis imposent beaucoup de restrictions sur les transformations applicables. Néanmoins ils sont inévitables si on veut traiter des programmes réels. Pour compenser ce problème, nous proposons d'intégrer aux programmes manipulés des informations sur leur comportement. Ces informations peuvent provenir du programmeur ou d'un logiciel d'analyse statique des programmes logiques. Nous présentons donc un système capable de manipuler de tels programmes "typés".

Méta-interprétation et transformations de programmes Le dernier chapitre propose un méta-interpréteur quasi-itératif pour Prolog pur. À l'aide de nos transformations de programmes, nous montrons que ce méta-interpréteur et le Vanilla-interpréteur sont équivalents (au sens de l'e.o.f.) au programme qu'ils interprétent. Par codage de ce méta-interpréteur, nous obtenons des résultats d'indécidabilité de l'arrêt et du vide pour certaines classes de programmes à une clause ternaire ou à deux clauses binaires.

Préliminaires

Dans ce chapitre, nous rappelons brièvement (et informellement) quelques notions, définitions et propriétés essentielles de la programmation logique et de la programmation logique avec contraintes. Ces rappels nous permettront de fixer les notations utilisées par la suite.

Programmation logique

Tous les résultats classiques qui sont énoncés ici sont présentés sans démonstration. Les principaux ouvrages qui nous ont aidés pour la rédaction de ce chapitre sont [Llo87], [Apt90] et [Del87]. Nous présentons surtout les programmes logiques définis: ils sont restreints syntaxiquement aux clauses de Horn, mais leur puissance de calcul est *complète* (ils ont la même puissance de calcul que les machines de Turing [Tär77]). Dans les chapitres concernant notre travail, nous ne nous intéressons qu'à cette catégorie de programmes. Les programmes logiques normaux et la SLDNF-résolution sont simplement évoqués.

Syntaxe

Un langage du premier ordre, donné par un alphabet, est défini comme l'ensemble de toutes les formules bien formées construites sur cet alphabet. Un tel alphabet consiste en sept catégories de symboles:

1. des variables que l'on notera par des caractères majuscules X, Y, W, Z, ... (représentées par l'ensemble $\mathcal{V}$);
2. des constantes qui sont des symboles de fonctions d'arité nulle, que l'on notera $a, b, \ldots$;
3. des symboles de fonctions munis d'une arité (non nulle), notés $f, g, \ldots$;
4. des symboles de prédicats munis d'une arité, notés $p, q, r, \ldots$ (représentés par l'ensemble $\mathcal{P}$);
5. les connecteurs logiques $\neg, \wedge, \vee, \rightarrow$ et $\leftrightarrow$;

6. les quantificateurs $\exists$ et $\forall$;

7. les symboles de ponctuation "(", ")", et ",".

Seules les quatre premières catégories varient d'un alphabet à l'autre. Tous les caractères utilisés pour noter une variable, une constante, un symbole de fonction ou de prédicat peuvent éventuellement être indicés ou avoir un ' ou un ". L'ensemble réunissant les symboles de fonctions et de constantes sera noté $\mathcal{F}$.

On définit maintenant la notion de **terme** et d'**atome**. Toute variable, toute constante est un terme, et, si f est un symbole de fonction d'arité n ($n > 0$), et si $t_1, t_2, \ldots t_n$ sont des termes, alors $f(t_1, t_2, \ldots t_n)$ est un terme. Si t est un terme sans variable, alors on dit que t est un *terme clos*. De la même façon, si p est un symbole de prédicat d'arité n, et si $t_1, t_2, \ldots t_n$ sont des termes, alors $p(t_1, t_2, \ldots t_n)$ est un atome. Si A est un atome sans variables, on dit que A est un *atome clos*.

Par la suite, les termes seront notés $t, l, \ldots$ et les atomes seront notés A, B, ... avec éventuellement indices et exposants. $\tilde{t}$ représentera un n-uplet de termes. $\mathcal{T}ermes(\mathcal{F}, \mathcal{V})$ représentera l'ensemble des termes et $\mathcal{A}tomes(\mathcal{P}, \mathcal{F}, \mathcal{V})$ l'ensemble des atomes (nous les noterons souvent seulement $\mathcal{T}ermes$ et $\mathcal{A}tomes$). Nous nous munissons de la fonction Var: $\mathcal{A}tomes \rightarrow 2^{\mathcal{V}}$ qui retourne l'ensemble des variables d'un atome. Cette fonction est également définie sur $\mathcal{T}ermes$.

Une formule bien formée est définie inductivement par :

- si A est un atome, alors A est une formule (on dira une formule atomique);

- si F et G sont des formules alors (F), $\neg F$ ("non F"), $F \wedge G$ ("F et G"), $F \vee G$ ("F ou G"), $F \rightarrow G$ ("G si F", souvent noté $G \leftarrow F$), $F \leftrightarrow G$ ("F ssi G") sont des formules;

- si F est une formule et X une variable, alors $\exists X F$ ("il existe X tel que F") et $\forall X F$ ("pour tout X, F") sont des formules.

Un **littéral** est un atome ou la négation d'un atome. Un littéral positif est un atome, un littéral négatif est la négation d'un atome. Nous pouvons maintenant définir (syntaxiquement) un programme logique:

Définition 1 *Une clause est une formule de la forme*

$$\forall X_1, X_2, \ldots X_n \ (L_1 \vee L_2 \vee \ldots L_m)$$

où chaque L_i est un littéral et $X_1, X_2, \ldots X_n$ sont toutes les variables apparaissant dans $L_1 \vee \ldots \vee L_m$.

Une **clause définie** est une clause qui contient exactement un littéral positif. Une **clause unité** est une clause définie qui ne contient aucun littéral négatif. Un **but défini** est une clause qui ne contient aucun littéral positif. Une **clause de Horn** est une clause définie ou un but défini. Un **programme défini** est un ensemble de clauses définies. Par souci de simplicité, nous appellerons programme logique un programme logique défini, sauf précision contraire.

Nous noterons une clause de Horn de la forme $A_0 \vee \neg A_1 \vee \ldots \neg A_n$ (avec $n \geq 0$) par $A_0 \leftarrow A_1, A_2, \ldots A_n$, et une clause de la forme $\neg A_1 \vee \ldots \neg A_n$ par $\leftarrow A_1, A_2, \ldots A_n$. Nous appelerons souvent règle une clause de Horn sous cette forme. La *tête* d'une clause désignera alors l'atome à gauche du symbole $\leftarrow$, et le *corps* désignera la liste des atomes à sa droite. Dans nos exemples écrits en Prolog, nous adopterons la syntaxe d'Edinburgh: `A0 :- A1,A2,...An.` ou `:- A1,...An.`

Modèles de Herbrand et opérateur de conséquence immédiate

La sémantique logique d'un programme P est donnée par l'ensemble des conséquences logiques de P. Lorsqu'on pose le but $\leftarrow B$ à P, ce qui nous intéresse est de savoir si on peut déduire B de P, autrement dit si B est une conséquence logique de P. Les systèmes de résolution travaillent par réfutation, en utilisant le principe de résolution de Robinson [Rob65]. Le but $\leftarrow B$ (autrement noté $\neg B$) est ajouté à P, et le système en déduit une contradiction en dérivant *la clause vide*: $P \cup \{\neg B\}$ est insatisfiable.

Définition 2 *La* **clause vide**, *notée* $\square$, *est la clause ayant une conséquence et un antécédant vides. Elle exprime une contradiction.*

On présente maintenant les notions d'interprétation et de modèle de Herbrand.

Pour donner une signification vraie ou fausse à une formule, il faut donner une signification aux symboles de cette formule. Les quantificateurs et les connecteurs ont une signification fixée, mais celle des variables, termes et atomes est libre. Pour certaines des *interprétations* possibles d'une formule, la formule représente une expression vraie. Ces interprétations particulières sont des *modèles* de la formule. Une catégorie importante en programmation logique est la catégorie des interprétations de Herbrand.

Soit P un programme (logique), l'univers de Herbrand de P, noté U_P, désigne l'ensemble des termes clos que l'on peut construire à partir des symboles de constantes et de fonctions qui apparaissent dans P (s'il n'y a pas de constantes, on rajoute un symbole spécial pour clore les termes). La base de Herbrand de P, notée B_P, désigne l'ensemble des atomes clos construits à partir des symboles de prédicats qui apparaissent dans P, et des éléments de

U_P. Une **interprétation de Herbrand** pour P est un sous-ensemble de B_P dont les éléments sont assignés à la valeur vrai. Les éléments qui ne sont pas dans l'interprétation sont assignés à la valeur faux.

Un **modèle de Herbrand** pour P est une interprétation de Herbrand pour P qui est un modèle de P. Il est intéressant de se ramener à la catégorie des modèles de Herbrand, car :

Proposition 1 *Soit P un ensemble de clauses. Si P a un modèle, alors P a un modèle de Herbrand, et P est insatisfiable ssi P n'a pas de modèle de Herbrand.*

On obtient encore des propriétés plus fortes dans le cas des programmes logiques (définis) :

Proposition 2 *Soient P un programme défini et $\{M_i\}_{i\in E}$ ($E \subseteq \mathbb{N}$) un ensemble non vide de modèles de P. Alors $\cap_{i\in E} M_i$ est un modèle de Herbrand de P.*

Tout programme défini P admet B_P comme modèle de Herbrand. Donc pour tout programme défini, l'ensemble de ses modèles de Herbrand est non vide, et l'intersection de ces modèles est un modèle de Herbrand de P. Ce modèle est le plus petit modèle de Herbrand de P, noté M_P. Ce modèle a la caractéristique importante de correspondre à l'ensemble des atomes clos qui sont conséquences logiques de P. Il fait le lien entre sémantique logique et déclarative.

Théorème 1 *[vEK76] Soit P un programme défini.*
Alors $M_P = \{A \in B_P |\ A$ est une conséquence logique de $P\}$.

On définit maintenant l'opérateur de conséquence immédiate qui relie sémantique déclarative et sémantique dénotationnelle des programmes logiques.

Soit P un programme défini, alors 2^{B_P} est l'ensemble de toutes les interprétations de Herbrand de P, c'est-à-dire l'ensemble de tous les sous-ensembles possibles de B_P. 2^{B_P} forme donc un treillis complet muni de l'ordre partiel $\subseteq$ (l'inclusion ensembliste), son plus grand élément est B_P, et son plus petit élément est $\emptyset$.

Définition 3 *[vEK76] Soit P un programme défini. L'application $T_P : 2^{B_P} \to 2^{B_P}$ est définie par : soit I une interprétation de Herbrand, alors*

$$T_P(I) = \left\{ A \in B_P \;\middle|\; \begin{array}{c} A \leftarrow A_1, \ldots A_n \text{ est une instance sans variable} \\ \text{d'une clause de } P \text{ et } \{A_1, \ldots A_n\} \subseteq I \end{array} \right\}$$

Cet opérateur est monotone, continu pour les programmes définis. Pour un programme défini P, un modèle de Herbrand de P est une interprétation de Herbrand de P notée I telle que $T_P(I) \subseteq I$. Enfin, cet opérateur donne une caractérisation en terme de point fixe du plus petit modèle de Herbrand d'un programme. On garde la notation anglaise lfp (*least fixpoint*) pour le plus petit point fixe.

Théorème 2 *[vEK76] Soit P un programme défini.*
Alors

$$M_P = \text{lfp}(T_P) = T_P \uparrow \omega$$

Substitutions, unification

Une **substitution** θ est un ensemble fini $\{X_1/t_1, \ldots X_n/t_n\}$, où les X_i sont des variables distinctes et les t_i des termes distincts. θ est une *substitution close* si tous les t_i sont des termes clos, et θ est un *renommage* si tous les t_i sont des variables distinctes entre elles. La substitution donnée par $\emptyset$ est la *substitution identité*. On notera les substitutions $\theta, \tau, \sigma, \ldots$

Une substitution peut s'appliquer à un terme, un atome ou une clause: l'application de θ à A consiste à remplacer dans A chaque occurrence d'une variable X_i par t_i, et on note le résultat θA. Une substitution θ est *idempotente* si $\theta\theta = \theta$. Soient θ une substitution, et V un ensemble de variables, on note $\theta \uparrow V$ la restriction de θ aux variables de V.

Soient t_1 et t_2 deux termes, t_2 est une **instance** de t_1, noté $t_2 \leq t_1$ s'il existe une substitution θ telle que $\theta t_1 = t_2$ (on dit aussi que t_1 *filtre* t_2). Si $t_1 \leq t_2$ et $t_2 \leq t_1$, on notera $t_1 \sim t_2$, c'est-à-dire t_1 et t_2 sont égaux à un renommage près. La relation d'ordre $\leq$ est également définie sur les atomes et les formules.

Soient A et B deux atomes, on dit que A **s'unifie** avec B s'il existe une substitution θ telle que $\theta A = \theta B$. θ est l'unificateur le plus général, noté **mgu** pour *most general unifier*, si pour tout autre unificateur σ de A et B, il existe une substitution τ telle que $\sigma = \theta\tau$. On note $A \vee B = \theta A = \theta B = C$ l'unifié de A et B. Enfin, l'**anti-unification** de A et B, notée $A \wedge B$ retourne un atome C tel que

$$A \wedge B = C \Leftrightarrow \begin{cases} \bullet\ A \leq C \text{ et } B \leq C \\ \bullet\ \forall D \text{ atome tel que } C \not\sim D, A \leq D \text{ et } B \leq D \\ \quad \text{alors } C \leq D \end{cases}$$

L'ordre $\leq$ forme un treillis complet sur les termes, modulo $\sim$ (il suffit de rajouter l'élément $\perp$ à $\mathcal{T}ermes$). On notera

- **glb** (pour *greatest lower bound*) le plus grand minorant d'un ensemble de termes, représenté par l'unifié de ces termes s'il existe, par $\perp$ sinon;
- **lub** (pour *least upper bound*) le plus petit majorant d'un ensemble de termes, représenté par l'anti-unifié de ces termes.

SLD-résolution

La sémantique opérationnelle des programmes logiques est donnée par la résolution SLD, basée sur le principe de la résolution de [Rob65]. La résolution SLD signifie résolution SL pour les programmes *Définis*, et SL signifie résolution *Linéaire* dirigée par une règle de *Sélection* (*Selection rule-driven Linear resolution for Definite clauses*). Cette procédure a été décrite pour la première fois dans [Kow74].

Définition 4 *Soient G le but $\leftarrow A_1, \ldots A_m, \ldots A_k$ et C la clause $A \leftarrow B_1, \ldots B_n$ du programme P. Alors G' dérive de G et C en utilisant le mgu θ et la règle de sélection S, si les conditions suivantes sont vérifiées :*

1. *A_m est l'atome choisi par S dans G;*
2. *θ est le mgu de A et A_m;*
3. *G' est le but $\leftarrow \theta(A_1, \ldots A_{m-1}, B_1, \ldots B_n, A_{m+1}, \ldots A_k)$.*

G' s'appelle une résolvante de G et C, avec le mgu θ.

En itérant cette procédure, on obtient une *dérivation SLD*, c'est-à-dire une séquence de résolvantes $G_0, G_1, \ldots G_n$ avec les mgu's $\theta_1, \theta_2, \ldots \theta_n$ (G_i est une résolvante de G_{i-1} avec le mgu θ_i). Soit θ la restriction de $\theta_1\theta_2\ldots\theta_n$ aux variables de G. On dit alors que G_0 a une dérivation de longueur n avec une substitution-réponse partielle θ, et une *résultante* $\theta G_0 \leftarrow G_n$ (si $n = 0$, la résultante est $G_0 \leftarrow G_0$). Si la dérivation SLD est finie, et que la dernière résolvante est la clause vide, alors c'est une *réfutation* pour G_0, et θ est la substitution-réponse pour G_0 et P.

La règle de sélection d'atome est la règle qui choisit l'atome à dériver à chaque pas de résolution. Étant donnée une telle règle S, on peut construire pour un programme P et un but G_0 l'*arbre SLD* de toutes les dérivations possibles. La racine est G_0, chaque nœud est étiqueté par une résolvante, et les arcs qui atteignent ces résolvantes sont étiquetés par les mgu's correspondants. Une feuille de l'arbre est soit la clause vide (la branche est une réfutation), soit une résolvante telle que l'atome choisi ne s'unifie avec aucune des têtes de règles de P (la branche est une dérivation échec).

Soit P un programme logique. On définit l'*ensemble des succès SS(P)* [AvE82] comme l'ensemble des atomes clos de B_P pour lesquels il existe une réfutation SLD dans P. *SS(P)* est souvent utilisé pour définir la sémantique opérationnelle de P.

Théorème 3 *[AvE82] Soit P un programme défini, $SS(P) = M_P$.*

Enfin, la résolution SLD est *monotone*: soient P_1, P_2 deux programmes logiques définis, et A un atome clos, si A est conséquence logique de P_1, alors A est conséquence logique de $P_1 \cup P_2$.

SLDNF-résolution

Un premier problème avec les programmes logiques définis est la difficulté à déduire des informations négatives.

Exemple 1

P : `couleur_primaire(bleu).`
`couleur_primaire(jaune).`
`couleur_primaire(rouge).`

La signification intuitive de ce programme est que ¬`couleur_primaire(vert)` est vrai. Pourtant, ¬`couleur_primaire(vert)` n'est pas une conséquence logique de P. On fait alors l'hypothèse du monde clos, notée CWA (*Closed World Assumption*) : soit P un programme, CWA(P) = $\{\neg A | A \in B_P$ et $A \notin M_P\}$. Avec cette règle d'inférence, on peut déduire tous les littéraux qui se trouvent dans M_P ou dans CWA(P). Mais cette règle n'est pas applicable pour la résolution : un littéral peut ne pas appartenir à M_P sans pour autant que sa résolution soit finie. On définit alors l'*ensemble des échecs finis FF(P)* [AvE82] : c'est l'ensemble des atomes clos A de B_P pour lesquels il existe un arbre SLD de racine A fini et ne contenant pas la clause vide. Si $A \in FF(P)$, alors par la règle de la *négation par l'échec* [Cla78], $\neg A$ peut être déduit. La résolution SLD étendue aux programmes logiques définis avec des buts *normaux* (i.e. avec des littéraux négatifs dans le but) est la résolution SLDNF$^-$ (résolution SLD avec négation par l'échec).

Néanmoins, la puissance d'expression des programmes logiques définis est limitée du fait de l'impossibilité d'avoir un littéral négatif dans le corps d'une clause.

Les programmes logiques normaux sont donc des ensembles de clauses de programmes, c'est-à-dire sans restriction sur le nombre de littéraux positifs dans les formules. Une clause de programme est de la forme $A_0 \leftarrow A_1, \ldots A_n$, où A_0 est un atome et où les A_i ($1 \leq i \leq n$) sont des littéraux (positifs ou négatifs). On ne peut toujours pas déduire d'informations négatives de ces programmes, puisque la flèche $\leftarrow$ est interprétée par *"si"*. Pour obtenir de l'information négative, on *complète* le programme, c'est-à-dire qu'on rajoute la partie "seulement si" à la définition des symboles de prédicats, et une théorie de l'égalité. Ainsi, le *complété de Clark* de notre exemple est

$$\forall x \; (\texttt{couleur\_primaire}(x) \Leftrightarrow (x = \texttt{bleu} \vee x = \texttt{rouge} \vee x = \texttt{jaune}))$$

La **résolution SLDNF**, est la résolution SLD avec *négation par l'échec* pour les programmes normaux. Lors d'une dérivation - notons G le but courant -, le comportement est le même si le littéral choisi dans G est un atome, mais s'il s'agit d'un littéral négatif $\neg A$, alors on construit un nouvel arbre SLDNF de racine la négation de ce littéral (A), et si ce nouvel arbre est un arbre échec fini, alors la résolvante de G est $G - \{\neg A\}$.

On notera que la règle d'inférence de la CWA et la résolution SLDNF sont non monotones (en reprenant l'exemple précédent, il suffit de rajouter le fait `couleur_primaire(vert)` pour que ¬`couleur_primaire(vert)` ne soit plus déductible).

Programmation logique avec contraintes

CLP(X) (*CLP* pour Constraint Logic Programming), proposé par J.Jaffar et J.-L.Lassez dans [JL86, JL87] est un cadre formel créé pour intégrer un mécanisme générique de calcul de contraintes dans la programmation logique classique. Nous rappelons ici les concepts de base présentés dans [JL86, JL87] et [Mah92].

Soient S un ensemble de sortes, et Σ un ensemble d'opérations (définies par des symboles de fonctions et de prédicats) munies d'arité sur S. Par souci de simplicité, on notera Σ la signature définie par Σ et S. Soit $\mathcal{V}$ un ensemble infini de variables. Soit A un S-ensemble (un S-ensemble est un ensemble muni d'une partition indexée par S). Alors le domaine de contraintes $(\mathcal{A}, \mathcal{L})$ est constitué d'une Σ-algèbre de termes (ou Σ-structure) $\mathcal{A}$ construite sur A et Σ, et d'un ensemble $\mathcal{L}$ des formules du premier ordre exprimables avec Σ et vérifiant les conditions de clôture par renommage et conjonction. $\mathcal{L}$ est le plus petit ensemble contenant les contraintes atomiques et satisfaisant ces conditions. Les contraintes atomiques sont les formules atomiques qu'on peut construire sur Σ et $\mathcal{V}$. On appelle $\mathcal{A}$-assignation une application de $\mathcal{V}$ dans A. Si θ est une $\mathcal{A}$-assignation du terme t, on note $t\theta$ l'élément correspondant dans A, et de manière similaire, si c est une contrainte atomique et θ une $\mathcal{A}$-assignation de c, alors on note $c\theta$ la proposition telle que

1. soit $\mathcal{A} \models c\theta$ ($c\theta$ est équivalent à *Vrai*)
2. soit $\mathcal{A} \models \neg c\theta$ ($c\theta$ est équivalent à *Faux*)

Si C est un ensemble (fini ou infini) de contraintes, on dira

1. $\mathcal{A} \models C\theta$ si $\forall c \in C,\ \mathcal{A} \models c\theta$
2. $\mathcal{A} \models \neg C\theta$ sinon

Lorsque 1) est vérifié, on dira que C est solvable dans $\mathcal{A}$ et θ est une $\mathcal{A}$-solution de C. On notera $Sol_C = \{\theta | \theta$ est une $\mathcal{A}$-solution de $C\}$.

Les domaines les plus courants sur lesquels les contraintes sont définies sont les arbres finis (Σ est un ensemble fini ou infini de symboles de fonctions, et A est l'ensemble des arbres finis (i.e. clos) construits sur Σ: $A = U_P$), les arbres rationnels ou les arbres infinis (A est l'ensemble des arbres infinis construits sur Σ), l'arithmétique réelle ($\mathcal{A} = \mathbb{R}$ et $\Sigma = \{0, 1, +, =, \times, \leq, <\}$), l'arithmétique

linéaire (sans symbole de multiplication $\times$ et $A = \mathbb{R}$ ou $A = \mathbb{Q}$), les domaines finis (par exemple $A = \mathbb{Z}$ et $\Sigma = \{[m..n]_{m \leq n}, +, \times, =, \neq\})$ ...

Différentes opérations effectuées sur les contraintes sont :

- le test de consistence;
- le test d'implication (étant donnés deux ensembles de contraintes C et C', est-ce que $\mathcal{A} \models C \rightarrow C'$);
- l'addition et l'effacement de contraintes (avec recalcul de la forme résolue ou canonique);
- la quantification existentielle des contraintes (pour la projection ou l'élimination de variables).

Les propriétés suivantes sont les propriétés les plus souvent vérifiées dans les domaines de contraintes.

Propriétés des domaines de contraintes

Solution compact La Σ-structure $\mathcal{A}$ d'un domaine de contraintes est *solution compact* si les deux hypothèses suivantes sont vérifiées :

1. chaque élément de $\mathcal{A}$ est la solution unique d'un ensemble fini ou infini de contraintes :
$$\forall d \in \mathcal{A},\ d = \cap c_i$$
2. chaque élément du complément de l'ensemble des solutions d'une contrainte c appartient à l'union de l'ensemble des solutions d'un ensemble fini ou infini de contraintes:
$$\forall c,\ \neg c = \cup c_i$$

Une *solution compact* structure est telle qu'un ensemble infini de contraintes est insatisfiable ssi un sous-ensemble fini de ces contraintes est insatisfiable.

Cette propriété est nécessaire pour retrouver toutes les bonnes propriétés sémantiques de la programmation logique (équivalence des sémantiques du plus petit point fixe, du plus petit modèle de Herbrand, de l'ensemble des succès et des échecs finis). [JL86] ne considèrent que les structures ayant cette propriété (la plupart des domaines de contraintes l'ont : tous ceux cités ci-dessus sont *solution compact*).

Indépendance des contraintes négatives Si $c_0 \wedge \neg c_1 \wedge \ldots \wedge \neg c_n$ n'est pas satisfiable dans $\mathcal{A}$, alors, $\exists j > 0$, $c_0 \wedge \neg c_j$ n'est pas satisfiable dans $\mathcal{A}$.

Forme résolue et forme canonique La forme résolue d'une contrainte c est

- *fail* si la contrainte n'est pas cohérente;
- *true* si c'est une tautologie;
- une contrainte sémantiquement équivalente dans une forme syntaxique simplifiée.

La forme résolue d'un ensemble de contraintes peut simplifier certains tests ou calculs, par exemple le calcul de projection sur un ensemble de variables (cf 3.3).

Deux ensembles de contraintes sémantiquement équivalents peuvent néanmoins avoir des formes résolues différentes. La forme canonique d'une contrainte c est une forme résolue de c telle que toutes les contraintes équivalentes à c aient la même forme canonique (cela suppose un algorithme de réduction des contraintes qui n'existe pas toujours).

Chapitre 1

Une équivalence pour les programmes logiques

"Une des relations les plus importantes entre programmes (quelque soit le langage de programmation) est la relation d'équivalence. Cette relation est à la base de la plupart des méthodologies de programmation" [Mah87]. En effet, lorsque l'on veut vérifier, comparer, prouver, ...un programme, on est obligé de tester l'équivalence entre ce programme et les spécifications, l'autre programme, ou la propriété qui nous intéresse. Mais chaque sémantique d'un programme induit une nouvelle relation d'équivalence. En programmation logique, on peut distinguer la sémantique logique (ou déclarative), la sémantique dénotationnelle (ou fonctionnelle), et la sémantique opérationnelle (ou procédurale). La sémantique logique peut par exemple, être caractérisée par les conséquences logiques du programme, ou les conséquences logiques du complété de Clark d'un programme, ou ces mêmes conséquences mais restreintes aux modèles de Herbrand; la sémantique dénotationnelle par l'opérateur T_P, ou par l'opérateur $T_P + Id$, ou par l'opérateur $[\![P]\!] = \bigcup_{i=0}^{\infty}(T_P+Id)$ [Mah87]; tandis que la sémantique opérationnelle peut être caractérisée par l'ensemble des succès *SS(P)* ou par l'ensemble des échecs finis *FF(P)*.

Comme on l'a vu au chapitre précédent, certaines de ces sémantiques sont équivalentes (cf. théorème 2). On pourrait alors se demander pourquoi définir différentes sémantiques lorsque finalement elles sont identiques? Les différents cadres de travail choisis fournissent différentes opérations et outils théoriques pour manipuler les programmes. Pour la sémantique logique, les outils sont les connecteurs logiques (conjonction, implication, ...), pour la sémantique dénotationnelle ce sont les opérations comme l'addition et la composition de fonctions. Ces outils permettent d'exprimer différentes propriétés : par exemple l'union de programmes par la sémantique logique (conjonction) ou la commutativité par la sémantique dénotationnelle ($[\![P_1]\!] \circ [\![P_2]\!] = [\![P_2]\!] \circ [\![P_1]\!]$). La sémantique dénotationnelle formalise le comportement d'un programme par des ensembles d'équations, la notion de comportement est vue comme un ob-

jet mathématique. La sémantique opérationnelle modélise un interpréteur, et peut contenir des détails d'implémentation techniques. La sémantique dénotationnelle semble donc un outil mieux adapté pour prouver la correction de transformations de programmes, tandis que la sémantique opérationnelle permet de mieux tenir compte des prédicats extra-logiques.

En effet, les sémantiques habituellement utilisées (plus petit modèle de Herbrand, ensemble des succès et plus petit point fixe de T_P) ne caractérisent ni les réponses d'un interpréteur Prolog (les substitutions, closes ou *non-closes*), ni les prédicats prédéfinis (ou les structures de contrôle comme le `cut`) des programmes Prolog réels. De nombreux travaux (dont nous ne donnons pas de liste exhaustive ici) ont donc contribué à une meilleure prise en compte de la réalité du langage Prolog, que ce soit par une sémantique dénotationnelle ou opérationnelle (voire même une approche déclarative dans le cas de [FLPM89] ou de [Del88]). Debray et Mishra [DM88] présentent une sémantique opérationnelle et dénotationnelle pour Prolog avec `cut`, en tenant compte de la règle standard de parcours de l'arbre SLD (en profondeur d'abord). Ses deux sémantiques sont équivalentes, et la sémantique dénotationnelle est un outil qui permet de prouver des transformations utilisées dans l'article de Debray et Warren [DW86] (*si le dernier littéral de la dernière clause d'un prédicat est un* `cut` *et si cette clause est déterministe indépendamment du* `cut`, *alors on peut supprimer le* `cut`, le deuxième théorème étant la transformation pratiquement inverse). Billaud [Bil91] propose aussi une sémantique dénotationnelle pour Prolog, mais il prend en compte la disjonction, le `cut`, et les prédicats d'entrées-sorties. La sémantique de Clark [Cla79] basée sur un univers de Herbrand étendu aux termes non-clos, ce qui permet de caractériser les substitutions-réponses d'un interpréteur Prolog, a été étudiée par Deransart, Małuszyński et Ferrand [Fer87, DM93], et par Falaschi, Palamidessi, Levi et Martelli [FLPM89]. Enfin Delahaye [Del88] s'attache à définir une sémantique dénotationnelle pour les programmes logiques définis avec un interpréteur standard (en profondeur d'abord, règle de sélection de l'atome à gauche). Par des opérateurs de conséquences, il définit l'ensemble des succès finis SSP et des échecs finis FFP obtenus par un tel interpréteur.

Notre approche de la programmation logique est liée à l'utilisation pratique de Prolog, dans le but de fournir au programmeur une boite à outils comprenant, par exemple, un évaluateur partiel, un débogueur, etc... Il faut pour cela définir une équivalence appropriée à la programmation de programmes Prolog complet (i.e. avec prédicats prédéfinis, donc effets de bord), ce qui n'est pas le cas des équivalences que nous venons de citer. Nous proposons donc une équivalence opérationnelle : deux programmes sont équivalents s'ils ont les mêmes solutions dans le même ordre, avant la première branche infinie. Nous considérons un interpréteur standard, et notre équivalence permet de prendre en compte les programmes Prolog complet, i.e. avec des prédicats prédéfinis et des structures de contrôle comme le `cut`. Cette équivalence est proche de

celle considérée par [PP91], bien que leur sémantique opérationnelle ne prenne pas du tout en compte les prédicats à effets de bord. C'est également une équivalence semblable qui est conservée par les systèmes d'évaluation partielle Mixtus [Sah91] et Paddy [Pre92a].

Nous présentons d'abord notre équivalence, puis nous étudions des transformations de programmes qui préservent cette équivalence. Ces transformations sont essentiellement des aménagements des techniques de pliage/dépliage proposées par H. Tamaki et T. Sato [TS84]. Ce sont des outils aussi bien pour l'évaluation partielle (chapitres 2 et 3), que pour la validation de méta-interpréteurs (chapitre 4), ou même pour faire un lien entre sémantique opérationnelle et dénotationnelle ([DD91]). Comme dans notre deuxième partie (chapitre 4) nous ne nous intéressons qu'au cas de Prolog pur, des extensions à nos transformations peuvent être appliquées. Elles sont présentées dans la dernière section de ce chapitre.

1.1 L'équivalence opérationnelle forte

L'idée principale est de garder une équivalence entre programmes Prolog au sens des interpréteurs standards (non-équitables, en profondeur d'abord et de gauche à droite). Deux programmes sont dits équivalents s'ils ont les mêmes solutions, (au sens des substitutions-réponses), dans le même ordre, jusqu'à la première branche infinie (la plus à gauche dans l'arbre de dérivation). Cela étend les sémantiques logiques et dénotationnelles pour les interpréteurs standards.

Pour obtenir une telle équivalence, nous devons définir une sémantique opérationnelle pour les effets de bord. Nous considérons que les lectures et les écritures se font sur un ruban infini d'un côté, et qu'il y a un ruban pour chaque source (entrée/sortie standards ou sur fichiers). La tête de lecture est accessible en exclusion mutuelle pour l'utilisateur et le programme, et elle se déplace après chaque lecture ou écriture, toujours vers le côté infini. Le comportement est analogue pour les `assert/retract`. Deux programmes équivalents doivent produire les mêmes rubans.

Les arbres de résolution que nous considérons sont les arbres SLD dont nous avons remplacé les substitutions par des systèmes d'équations. Toute branche d'un arbre de résolution est ainsi étiquetée de systèmes d'équations dont l'union est solvable. La fonction *Nœuds-solution*, définie sur un programme et un but, retourne la liste des substitutions-réponses associées aux nœuds étiquetés par $\square$ (ces substitutions sont obtenues en faisant l'union des systèmes d'équations qui se trouvent sur la branche du nœud succès). Enfin la relation d'ordre $\leq$ sur les branches infinies et sur les branches succès est donnée par le parcours de gauche à droite et en profondeur d'abord de l'arbre SLD : *"est à gauche de"*.

Définition 5 Equivalence Opérationnelle Forte

Soient deux programmes: (Π_1,$Goal_1$) et (Π_2,$Goal_2$), ils sont dits opérationnellement fortement équivalents si et seulement si:

1. *il existe une bijection θ :*

   θ : *Nœuds-Solution*(Π_1,$Goal_1$) $\longrightarrow$ *Nœuds-Solution*(Π_2,$Goal_2$)

   telle que θ est compatible avec la fonction mgu, où Mgu_1 (respectivement Mgu_2) est la fonction qui, pour un nœud-solution donné du programme Π_1 (respectivement Π_2), renvoie la substitution-réponse associée : $Mgu_1 = Mgu_2 \circ \theta$

2. *il existe une bijection θ' sur les branches infinies et les branches succès qui est monotone par rapport à la relation d'ordre $\leq$ ($\theta = \theta'$ sur les branches succès): soient br_1 et br_2 deux branches de dérivation (infinie ou succès)*

$$br_1 \leq br_2 \Leftrightarrow \theta'(br_1) \leq \theta'(br_2)$$

Remarques :

- l'équivalence opérationnelle forte peut être définie seulement sur un ensemble de variables. Le seul point modifié est le 1., car on ne considère que la projection de la fonction Mgu sur l'ensemble de variables;

- deux programmes opérationnellement fortement équivalents ont même ensemble de succès SS et même ensemble d'échecs finis FF. Pour conserver l'équivalence opérationnelle au sens des interpréteurs standards (les ensembles SSP et FFP de [Del88]), seule la première branche infinie aurait besoin d'être conservée dans le programme transformé. Notre équivalence, légèrement plus forte, préserve l'arbre SLD de façon plus générale.

1.2 Transformations de programmes préservant l'équivalence opérationnelle forte

Nous présentons ici les modifications à apporter aux opérations de pliage et dépliage ([TS84]) pour qu'elles préservent notre équivalence. Le chapitre suivant traite plus en détail des transformations de programmes logiques, mais pour respecter la forme de notre travail, nous présentons dans ce chapitre le noyau commun aux deux parties suivantes de la thèse : les résultats concernant les liens entre équivalence opérationnelle forte et transformations.

Nous nous restreignons à la classe des programmes Prolog contenant comme seul prédicats prédéfinis le `cut`, le `read` et le `write`. Un programme Prolog est pour nous une *liste* de clauses, et le corps d'une clause une *liste* d'atomes.

1.2.1 Définition

Un programme est divisé en deux ensembles non disjoints : l'ensemble des clauses du programme et l'ensemble des définitions du programme. Ce second ensemble est constitué au départ des clauses du programme qui définissent un symbole de prédicat qui n'apparaît jamais dans le corps d'une autre clause. La transformation appelée définition consiste à ajouter au programme et à l'ensemble des définitions une clause définissant un nouveau symbole de prédicat (une étape de définition préserve bien sûr l'équivalence opérationnelle forte puisqu'elle n'intervient pas sur l'exécution du programme). L'ensemble des définitions est généralement utilisé pour le pliage.

Au fur et à mesure des transformations effectuées sur un programme (pliage, dépliage), l'ensemble des définitions d'un programme n'est plus forcément inclus dans l'ensemble des clauses du programme.

1.2.2 Dépliage

La définition du dépliage proposée par Tamaki et Sato est très simple : c'est une étape de dérivation dans l'arbre SLD, sans restriction sur le littéral choisi.

Exemple 2

```
1. a(X,Y) :- b(X),c(Y).
2. b(0).
3. b(1).
4. c(0).
5. c(1).
```

On déplie l'atome `c(Y)` de la première clause, et on obtient :

```
1. a(X,0) :- b(X).
2. a(X,1) :- b(X).
3. b(0).
4. b(1).
5. c(0).
6. c(1).
```

Les deux arbres de résolutions pour le but `:-a(X,Y)` apparaissent sur la figure 1.1. L'ordre des solutions dans le cas du premier programme est `[(X=0,-Y=0),(X=0,Y=1),(X=1,Y=0),(X=1,Y=1)]` et `[(X=0,Y=0),(X=1,Y=0),(X=0,-Y=1),(X=1,Y=1)]` dans le cas du programme déplié.

Cette transformation ne préserve pas l'ordre des solutions. Plus de contraintes sont nécessaires :

- soit le littéral déplié s'unifie uniquement avec une tête de règle (dépliage déterministe);

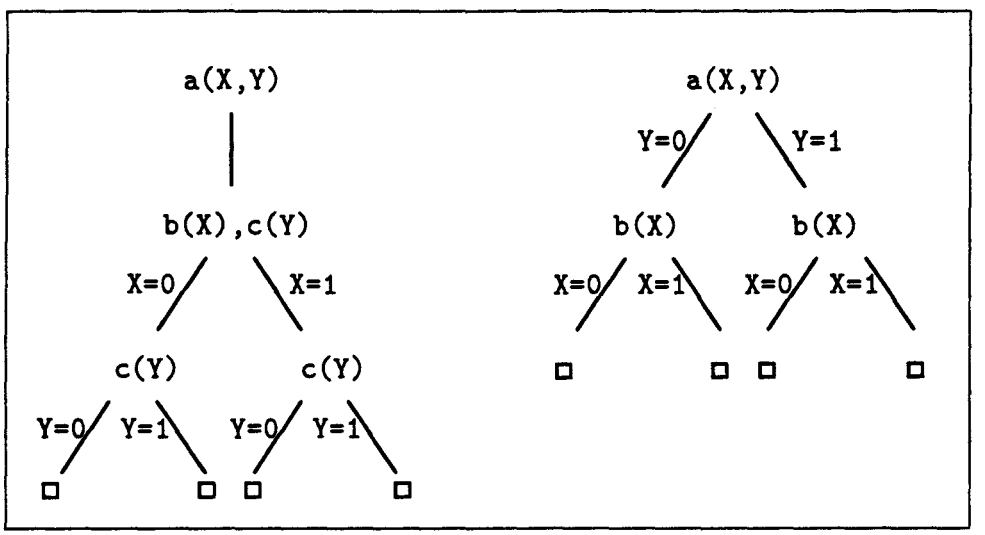


FIG. 1.1 - : exemple 2

- soit le littéral déplié est le premier littéral (en parcourant le corps de la clause de gauche à droite) qui n'est pas strictement déterministe.

D'autre part, pour tenir compte des prédicats à effets de bord, la clause dépliante ne doit pas contenir de `cut` [VD88]. Changer le niveau d'appel d'un `cut` dans un arbre SLD modifie entièrement son comportement : en particulier, si la clause dépliante contient un `cut`, le dépliage remonterait le `cut` dans l'arbre SLD, et risquerait d'élaguer des branches (et donc éventuellement des solutions).

Exemple 3
Soit le programme suivant :

```
1. a(X) :- b(X).
2. a(c).
3. b(a) :- !.
4. b(b).
```

Si on déplie le premier atome de la règle 1, on obtient :

```
1. a(a) :- !.
2. a(b).
3. a(c).
4. b(a) :- !.
5. b(b).
```

Pour le but `:-a(X)` le premier programme fournit deux solutions {`X=a;-X=c`}, tandis que le second programme ne donne plus qu'une solution {`X=a`} (cf. figure 1.2).

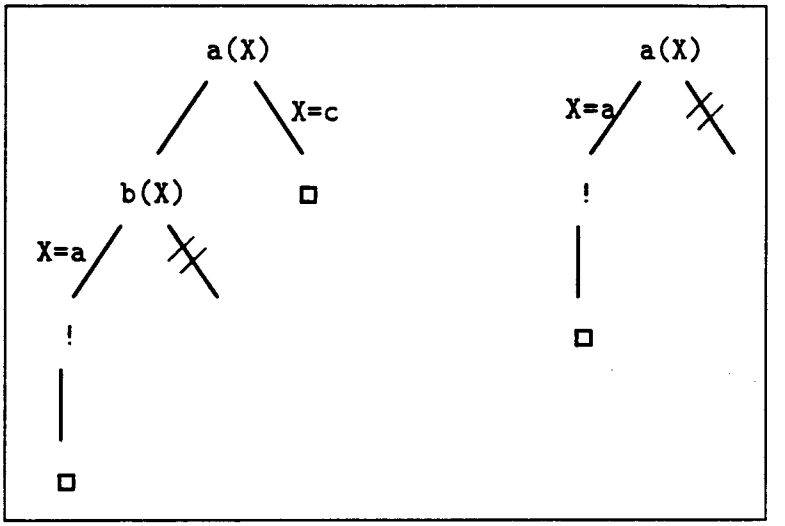


FIG. 1.2 - : exemple 3

Enfin, dernière restriction, l'instanciation de la règle résultant du dépliage ne doit pas être effectuée sur toute la règle:

Exemple 4
Soit le programme suivant:

```
1. a(X) :- b(X),!,c(X).
2. b(a).
3. b(b).
4. c(b).
```

Si on déplie `c(X)` dans la règle `1`, on obtient:

```
1. a(b) :- b(b),!.
2. b(a).
3. b(b).
4. c(b).
```

Pour le but `:-a(X)` le premier programme échoue, tandis que le second donne une solution {`X=b`}.

L'instanciation ne doit pas être propagée à gauche du littéral déplié. Cette restriction peut être levée dans le cas de Prolog pur (cf. section 1.3) tout en préservant l'équivalence opérationnelle forte.

Définition 6 Dépliage en tête
Soient P un programme Prolog, C une clause de P et B un atome du corps de C.

C peut être dépliée par rapport à B si chaque atome devant B dans le corps de C s'unifie avec au plus une clause, qui de plus est une clause unité. Les prédicats prédéfinis sont interdits à gauche de B.

Soient C_1, C_2, ..., C_k des clauses de P, et B_1, B_2, ... B_k leur tête, telles que

- *chaque B_j peut s'unifier avec B par le mgu θ_j;*
- *C_1, C_2, ... C_k ne contiennent pas de* `cut`.

Alors la clause C est remplacée par les clauses C'_1, C'_2, ... C'_k (dans cet ordre). Chaque C'_j est obtenu en insérant devant B dans C l'égalité $B = B_j$, puis en remplaçant B par le corps de C_j dans C.

C est appelée la clause dépliée, et C_j les clauses dépliantes.

Exemple 5

Le programme suivant est le méta-interpréteur Vanilla:

$\forall i \quad 1 \leq i \leq n$

1.i. `rule([`a_i`,`$b_{i,1}$ ... `]).`

2. `solve([]).`
3. `solve([`A_1`,`A_2`|B]) :- solve([`A_1`]), solve([`A_2`|B]).`
4. `solve([A]) :- rule([A|B]), solve(B).`

En dépliant le premier atome de la règle 4, on obtient:

$\forall i \quad 1 \leq i \leq n$

1.i. `rule([`a_i`,`$b_{i,1}$ ... `]).`

2. `solve([]).`
3. `solve([`A_1`,`A_2`|B]) :- solve([`A_1`]), solve([`A_2`|B]).`

$\forall i \quad 1 \leq i \leq n$

4.i. `solve([A]) :- A =` a_i`, solve([`$b_{i,1}$ ... `]).`

Proposition 3 *Le dépliage en tête préserve l'équivalence opérationnelle forte.*

Preuve

1. Bijection sur les nœuds-solution et conservation des substitutions-réponses.

   Le dépliage en tête est équivalent à la dérivation durant la résolution SLD du programme original de l'atome choisi (B_j dans la définition) en priorité par rapport aux autres littéraux de la règle, avant de reprendre la dérivation standard. Ce comportement est donc similaire à effectuer une permutation sur les atomes (en plaçant B_j en tête de la règle). Comme les atomes à gauche de B_j ne sont pas des prédicats prédéfinis, par le *switching lemma* [Llo87], on a la conservation des substitutions-réponses (et donc la bijection sur les nœuds-solution).

2. Ordre des solutions

Nous fixons d'abord les notations. (T, G) représentera un couple arbre de résolution-but (T_0 représente l'arbre correspondant au programme P_0, et T_1 celui correspondant au programme P_1). *Dom(T,G)* est l'ensemble des chemins de T (représentés par les règles qui les composent). $T(m)$ est le sous-arbre accessible à partir de la racine par le chemin m dans T, et $T[m]$ est l'étiquette du nœud accessible à partir de la racine par m dans T. Enfin, on note $Dom_{\Box}(T,G)$ l'ensemble des chemins m appartenant à *Dom(T,G)* tels que $T[m] = \Box$.

Soient P_0 un programme Prolog, et P_1 le programme résultat d'un dépliage en tête dans P_0. Soient N le numéro de la règle dépliée dans P_0 et $N_1, \ldots N_k$ les numéros des k règles dépliantes. $NN_1, \ldots NN_k$ sont les numéros des versions dépliées de N. Les clauses NN_j sont insérées dans P_1 exactement à la place de N. ω est le chemin de dérivation de tous les atomes à gauche de l'atome déplié ayant pour départ n'importe quelle instance de la tête de la règle de N (il n'y a qu'un seul chemin puisque par hypothèse les littéraux à gauche de l'atome déplié sont déterministes). Soit G un but.

(a) Fonction de transformation du dépliage en tête

La fonction δ, de *Dom(T₀,G)* vers *Dom(T₁,G)*, est la fonction qui modifie les chemins selon les règles du dépliage en tête :

i. $m \in Dom(T_0,G)$ et $N \notin m \Rightarrow \delta(m) = m$

ii. On considère le cas d'un dépliage de N par rapport à N_j, pour $1 \leq j \leq k$
$\delta(N.\omega.N_j) = NN_j.\omega$ par application de la définition du dépliage en tête

iii. pour ω_1 facteur gauche de ω, quel est le résultat de $\delta(N.\omega_1.fail)$? En appliquant la définition du dépliage en tête, on remplace la règle de N par k règles NN_j dont la tête et le corps *jusqu'au littéral* $\{B = B_j\}$ sont identiques.

- soit $\omega_1 \neq \omega$
Si la règle N a pu être utilisée dans P_0, alors toutes les règles NN_j ont pu l'être dans P_1, et l'échec survenu avant $\{B = B_j\}$ dans P_0 s'est également produit dans P_1. On obtient donc
$\delta(N.\omega_1.fail) = \{NN_j.\omega_1.fail | j \in [1,k]\}$
- soit $\omega_1 = \omega$
C'est le but qui est une instance de B, notée $\mathcal{B}$, qui ne s'est s'unifié avec aucune des têtes de règles de N_j. L'équation $\{\mathcal{B} = B_j\}$ n'est donc pas solvable, et ce littéral tombera

en échec quand on tentera de le résoudre dans une clauses NN_j. On obtient donc
$\delta(N.\omega_1.fail) = \{NN_j.\omega_1.fail | j \in [1,k]\}$

(b) Conservation de l'ordre des solutions

On s'intéresse uniquement maintenant aux transformations des éléments de $Dom_\Box(T_0,G)$. On sait par 1. qu'il y a bijection entre les feuilles de $Dom_\Box(T_0,G)$ et de $Dom_\Box(T_1,G)$, et par 2.(a) que ces chemins sont transformés par la fonction δ.

Il suffit de montrer que pour $m, m' \in Dom(T_0,G)$, et m, m' de la forme $N.\omega.N_j$ $(1 \leq j \leq k)$ ou tels que $N \notin m$ ou $N \notin m'$, on a bien : $m \leq m' \Rightarrow \delta(m) \leq \delta(m')$. La relation d'ordre $\leq$ est celle utilisée dans la définition de l'équivalence opérationnelle forte ($m \leq m'$ signifie *"m est à gauche de m'"*).

(Si δ conserve l'ordre des solutions, alors le dépliage en tête conserve l'ordre des solutions). On suppose maintenant que $m, m' \in Dom(T_0,G)$

i. $N \notin m$ et $N \notin m'$
Si $m \leq m'$, comme $\delta(m) = m$ et $\delta(m') = m'$, on a $m \leq m' \Leftrightarrow \delta(m) \leq \delta(m')$

ii. $N \notin m$ et $m' = N.\omega.N_j$, $j \in [1,k]$
$m \leq m' \Leftrightarrow m \leq N.\omega.N_j \Leftrightarrow m \leq N$
On sait que $\delta(m) = m$ et $\delta(N.\omega.N_j) = NN_j.\omega$ où NN_j fait partie des règles qui ont été insérées à la place de N dans P_1. Donc si m se trouve à gauche de N dans P_0, m se trouve toujours à gauche des règles qui sont à la même position ou plus loin que N (à cause du comportement d'un interpréteur standard): $m \leq m' \Leftrightarrow \delta(m) \leq \delta(m')$

iii. $m = N.\omega.N_j$, $j \in [1,k]$ et $N \notin m'$
$m \leq m' \Leftrightarrow N.\omega.N_j \leq m'$, avec $\delta(N.\omega.N_j) = NN_j.\omega$
m' commence par une règle M tel que $N \leq M$. Après un dépliage en tête, toutes les règles derrière N dans P_0 sont décalées de $k-1$ places dans P_1. On a donc, $N \leq M \Leftrightarrow NN_j \leq M$, et donc $m \leq m' \Leftrightarrow \delta(m) \leq \delta(m')$

iv. $m = N.\omega.N_i$ et $m' = N.\omega.N_j$ avec $i, j \in [1,k]$ et $i \neq j$

$\delta(m) \leq \delta(m')$
$\Leftrightarrow$ $\delta(N.\omega.N_i) \leq \delta(N.\omega.N_j)$
$\Leftrightarrow$ $NN_i.\omega \leq NN_j.\omega$
comme les règles NN_j respectent l'ordre des règles N_j
$\Leftrightarrow$ $N_i \leq N_j$
$\Leftrightarrow$ $N.\omega.N_i \leq N.\omega.N_j$
$\Leftrightarrow$ $m \leq m'$

3. Préservation de l'ordre des nœuds-solutions par rapport aux branches infinies

   Appelons *complexité* d'un nœud-solution n le nombre de nœuds de l'arbre SLD qu'on parcourt (par la stratégie en profondeur d'abord et de gauche à droite) avant d'atteindre n. On notera φ_i la fonction de complexité du programme i.

   Nous allons encadrer la complexité d'un nœud-solution dans P_1 par la complexité de son correspondant dans P_0. On notera n un nœud-solution de P_0 et $\theta(n)$ son correspondant dans P_1.

   - Dans le pire des cas, le dépliage en tête allonge les chemins pour accéder à certains nœuds (cas 2.(a).iii de la transformation δ). Notons *Maxl* le nombre maximal de littéraux dans une règle de P_0, et *Maxr* le nombre de règles de P_0. Alors dans le cas $\delta(N.\omega_1.fail) = \{NN_j.\omega_1.fail | j \in [1,k]\}$, au pire $k = Maxr$ et $|\omega_1| = Maxl$: $\varphi_1(\theta(n)) \leq \varphi_0(n) \times Maxl \times Maxr$
   - Dans le meilleur des cas, le dépliage en tête est également un dépliage déterministe, ce qui supprime un appel de règle, mais ajoute une égalité: $\varphi_0(n) = \varphi_1(\theta(n))$

   La complexité des nœuds-solutions du programme transformé étant linéairement dépendante de celle du programme original, la séquence des substitutions-réponses avant la première branche infinie n'est pas altérée par le dépliage en tête. Un raisonnement similaire que nous utilisons pour des complexités entières peut être utilisé pour des complexités ordinales, afin de montrer la conservation totale de la relation d'ordre $\leq$ (*"est à gauche de"*).

□

Il ne peut pas y avoir de **read** ou **write** à gauche du littéral déplié: en remplaçant une règle par k règles, on évalue k fois les littéraux à gauche du littéral déplié. Quand il y a des prédicats à effets de bord, la bande de sortie du programme est donc modifiée.

Définition 7 Dépliage déterministe *Soient P un programme Prolog, C une clause de P, et B un littéral du corps de C (B ne doit pas être un prédicat prédéfini).*

C peut être dépliée par rapport à B si B s'unifie avec une seule tête de règle, notée D, par le mgu θ, et si D ne contient pas de **cut**.

C' est la règle obtenue en remplaçant B dans le corps de C par l'égalité $\{B = D\}$ et par le corps de D, et le programme résultat P' est obtenu en remplaçant C par C' dans P.

Exemple 6
On continue avec le programme obtenu par un dépliage en tête du Vanilla méta-interpréteur :

$\forall i \quad 1 \leq i \leq n$

$1.i.$ `rule([`a_i`,`$b_{i,1}$ ... b_{i,n_i}`]).`

2. `solve([]).`
3. `solve([`A_1`,`A_2`|B]) :- solve([`A_1`]), solve([`A_2`|B]).`

$\forall i \quad 1 \leq i \leq n$

$4.i.$ `solve([A]) :- A =` a_i`,solve([`$b_{i,1}$ ... b_{i,n_i}`]).`

En effectuant n dépliages déterministes sur chacune des $4.i$ règles, on obtient :

$\forall i \quad 1 \leq i \leq n$

$1.i.$ `rule([`a_i`,`$b_{i,1}$ ... b_{i,n_i}`]).`

2. `solve([]).`
3. `solve([`A_1`,`A_2`|B]) :- solve([`A_1`]), solve([`A_2`|B]).`

Le résultat est n règles, quelques unes comme :

$4.i.$ `solve([A]) :- A =` a_i`.`

et d'autres comme :

$4.i.$ `solve([A]) :- A =` a_i`, solve([`$b_{i,1}$`]),solve([`$b_{i,2}$ ... b_{i,n_i}`]).`

Proposition 4 *Le dépliage déterministe préserve l'équivalence opérationnelle forte.*

Preuve Soit G un but.

On compare les arbres de résolutions de (P, G) et (P', G), qu'on notera pour abréger *Arbre(P,G)* et *Arbre(P',G)*. D est la clause dépliante dont nous noterons H le littéral de tête, et *Body(D)* le corps.

Tous les nœuds de *Arbre(P,G)* qui contiennent une instance de B (on les notera $\mathcal{B}$) ont pour correspondant dans *Arbre(P',G)* un nœud de même étiquette, à l'exception de $\mathcal{B}$ qui est remplacé par $(B = H, Body(D))$.

Supposons maintenant que N est un nœud de *Arbre(P,G)* de la forme : $\leftarrow \mathcal{B}, After$, avec *After* conjonction d'atomes, et qui de plus est le premier sur sa branche (de : $\leftarrow G$ à N) à remplir cette condition. On notera σ la substitution associée au chemin de la racine à N. N génère au plus une branche de dérivation (puisque par hypothèse B ne s'unifie qu'avec H), dont l'arc, si elle existe (si $\mathcal{B}$ et H s'unifient), est étiqueté par le numéro de D et par le système d'équation $\{\mathcal{B} = H\}$. Le nœud issu de cet arc a pour label $\{\mathcal{B} = H\} \cup \sigma$(:- *Body(D), After)*. Dans *Arbre(P',G)*, le nœud N' correspondant à N a pour label : $\leftarrow \mathcal{B} = H, Body(D), After$. σ est bien la substitution associée à N', puisque les deux branches sont identiques jusqu'à N et N'. N' génère une seule branche de

dérivation si $\mathcal{B}$ et H s'unifient, dont l'arc est étiqueté par le système d'équation $\{\mathcal{B} = H\}$. Le nœud issu de cet arc a pour label $\{\mathcal{B} = H\} \cup \sigma$*(:- Body(D), After)*. Ce raisonnement peut être appliqué par récurrence aux nœuds ayant pour premier atome $\mathcal{B}$ et qui se trouvent plus loin sur la branche partant de N.

Arbre(P,G) et *Arbre(P',G)* sont identiques à une modification d'étiquette près sur certains nœuds. L'équivalence opérationnelle forte est donc conservée. □

1.2.3 Pliage et normalisation de programme

Pliage

Notre pliage est une restriction du pliage de Tamaki et Sato: nous plions par rapport à une **séquence** d'atomes, et non par rapport à un ensemble d'atomes.

Définition 8 Pliage
Soient P un programme Prolog, θ une substitution, et C une clause de P de la forme
$A \leftarrow A_1, \ldots A_{j-1}, \theta(A_j, \ldots A_{j+k}), A_{j+k+1}, \ldots A_n.$
et D une clause de l'ensemble des définitions de P (qui ne contient pas de cut*) de la forme*
$G \leftarrow A_j, \ldots A_{j+k}.$
telles que

1. *θG s'unifie uniquement avec tête(D);*
2. *θ substitue à chaque variable locale de D une nouvelle variable n'apparaissant pas dans $\{A, A_1, \ldots A_{j-1}, A_{j+k+1}, \ldots A_n, \theta G\}$.*

Alors C est transformée en
$A \leftarrow A_1, \ldots A_{j-1}, \theta G, A_{j+k+1}, \ldots A_n.$

Proposition 5 *Le pliage préserve l'équivalence opérationnelle forte.*

Preuve Notre pliage est la transformation inverse du dépliage déterministe: on peut appliquer un dépliage déterministe sur la clause pliée par rapport à θG, puisqu'on vérifie bien toutes les conditions (unicité de la règle dépliante, pas de cut dans son corps). La différence vient du fait que D n'appartient pas forcément à P, mais fait partie de son ensemble de définitions:

- si $D \in P$: le résultat du dépliage déterministe est bien le programme original (avant le pliage);

- sinon, si $D \notin P$: lorsque D a été créée par une étape de définition, D a été ajoutée à P. Si D n'est plus dans le programme courant, c'est que sa définition a été modifiée par une transformation préservant l'équivalence opérationnelle forte. Notre pliage est donc la composée d'un dépliage déterministe et d'autres transformations préservant l'équivalence opérationnelle forte.

□

Normalisation de programmes

La condition du pliage portant sur les variables locales de la règle pliante peut être allégée par une transformation syntaxique des règles pour qu'elles vérifient toujours cette condition. Il suffit d'ajouter un argument *"fourre-tout"* pour que l'ensemble des variables du corps d'une règle soit inclu dans l'ensemble des variables de la tête.

Définition 9 Programme normalisé
Tout programme P dont toutes les clauses vérifient $Var(Tête) \supseteq Var(Corps)$ est appelé un programme normalisé.

Algorithme de normalisation d'une règle

Soit (P,G) un couple (programme-but), R la règle de P qu'on veut normaliser et r le prédicat défini par R.

Si $Var(Corps(R)) \subseteq Var(Tête(R))$ alors
  R est normalisé;
  retourner (P,G)
Sinon
  /* Maintenant, chaque fois que r apparaît dans (P,G), on lui ajoute */
  /* un argument */
  1. pour chaque littéral de prédicat r qui apparaît dans le corps d'une règle ou dans un atome du but G
      l'argument ajouté est une nouvelle variable;
  /* Notons (P', G') le programme obtenu */
  2. pour chaque règle définissant r dans P'
      si la règle est un fait alors
        l'argument est [];
      sinon
        l'argument ajouté aux têtes de règles définissant r est un terme $[X_1, X_2, \ldots, X_n]$ tel que $\{X_1, X_2, \ldots, X_n\} = Var(Corps) - Var(Tête)$
      fin si.
  3. Si G est une séquence de n atomes, notés G_i, on ajoute n règles $G_i \leftarrow G'_i$.
  4. Retourner (P',G)
Fin si.

Proposition 6 *Normaliser une règle d'un programme (P,G) préserve l'équivalence opérationnelle forte.*

Preuve Soient P un programme, R une règle de P et r le prédicat qu'elle définit. On construit (suivant l'algorithme précédent) R_{norm} la version normalisée de R (on notera de manière similaire r_{norm} et P_{norm}). On considère que tous les prédicats de P ont un seul argument (on peut toujours se ramener à ce cas, il suffit de prendre la liste de tous les paramètres habituels). Soit G un but.

La règle R est de la forme $r(\tilde{t}) \leftarrow c_1(\tilde{t_1}), \ldots c_n(\tilde{t_n})$. et la règle R_{norm} est de la forme $r(\tilde{t}, [X_1, \ldots X_m, L]) \leftarrow c_1(\tilde{t_1}), \ldots c_n(\tilde{t_n})$. avec $X_1, \ldots X_m$ variables locales de R et si R est récursive, L est la variable *fourre-tout* de l'atome $r(\tilde{t_i}, L)$ qui se trouve dans le corps de la règle R_{norm}.

Tout appel à la règle R dans *Arbre(P,G)* par un but $r(\tilde{t'})$ a un correspondant dans *Arbre(P_{norm},G)* par le but $r(\tilde{t'}, X)$ où X est une variable libre qui n'est jamais apparue auparavant dans l'arbre de résolution (étape 1 de l'algorithme de normalisation). Si σ est la substitution associée à $\{\tilde{t} = \tilde{t'}\}$, alors la substitution associée au but normalisé dans *Arbre(P_{norm},G)* est $\sigma \cup \{X = [X_1, \ldots X_m, L]\}$ où $X, X_1, \ldots X_m, L$ sont des variables libres qui n'apparaissent pas ailleurs dans σ. Le but dérivé est, dans *Arbre(P,G)*, $\sigma(c_1(\tilde{t_1}), \ldots c_n(\tilde{t_n}))$, et dans *Arbre($P_{norm}$,G)*, $\sigma(c_1(\tilde{t_1}), \ldots c_n(\tilde{t_n}))$, sauf si R est récursive, auquel cas un des atomes du but est de la forme $\sigma r(\tilde{t_i}, L)$, avec L variable libre. La raisonnement peut donc être appliqué itérativement pour montrer l'équivalence des dérivations. Si R est un fait, la variable X peut toujours s'unifier avec la liste vide ([]) sans modifier la résolution. □

Normaliser un programme c'est appliquer l'algorithme de normalisation à toutes les règles du programme. On obtient donc:

Proposition 7 *Tout programme (P_1,G) a un programme normalisé (P_2,G) qui lui est fortement opérationnellement équivalent.*

La normalisation nous permet de changer un peu la définition du pliage, afin d'élargir ses possibilités d'applications:

Définition 10 Pliage-normalisation

Soient P un programme Prolog, θ une substitution, et C une clause de P de la forme

$A \leftarrow A_1, \ldots A_{j-1}, \theta(A_j, \ldots A_{j+k}), A_{j+k+1}, \ldots A_n$.

et D une clause de l'ensemble des définitions de P (qui ne contient pas de cut*) de la forme*

$G \leftarrow A_j, \ldots A_{j+k}$.

telles que θG s'unifie uniquement avec tête(D).

Alors la clause D et le but G sont normalisés, et, si on note G_{norm} le normalisé de G, C est transformée en

$A \leftarrow A_1, \ldots A_{j-1}, \theta G_{norm}, A_{j+k+1}, \ldots A_n$.

Remarque: D peut être normalisée uniquement par rapport à l'ensemble des variables locales de D qui ne vérifient pas la condition 2 de la définition 8.

Exemple 7
Programme original:

```
1. a(X):-b,c(X),d(X).
2. e(X):-c(Y),d(X).
3. b.
4. c(0).
5. d(0).
6. d(1).
:- e(1).
```

Programme normalisé:

```
1. a(X) :- b,c(X),d(X).
2. e(X,[Y]) :- c(Y),d(X).
3. b.
4. c(0).
5. d(0).
6. d(1).
:- e(1,L).
```

Programme normalisé plié:

```
1. a(X) :- b,e(X,[X]).
2. e(X,[Y]) :- c(Y),d(X).
3. b.
4. c(0).
5. d(0).
6. d(1).
:- e(1,L).
```

1.2.4 Transformation de Gallagher-Bruynooghe

Nous montrons ici que la transformation dérivée de celle proposée par Gallagher et Bruynooghe dans [GB90] préserve notre équivalence opérationnelle forte. Cette transformation a pour but de retirer d'un programme les foncteurs inutiles, afin d'optimiser l'indexation des clauses à l'exécution du programme. Cette transformation est présentée plus en détail dans la section 2.2.2. L'idée principale est de calculer l'anti-unifié des têtes de clauses d'un prédicat (ou des corps des clauses d'un prédicat), et de ne plus considérer que les variables

distinctes occurant dans l'anti-unifié, en supprimant les foncteurs communs à toutes les têtes (corps) des clauses d'un prédicat.

Définition 11 Indexation sur les corps de règles

Soient P un programme Prolog, G un but et p un prédicat (non dynamique) défini par n clauses :

$$\begin{array}{lll} p(\tilde{t_1}) & \leftarrow & B_1 \\ p(\tilde{t_2}) & \leftarrow & B_2 \\ & \vdots & \\ p(\tilde{t_n}) & \leftarrow & B_n \end{array}$$

La définition de p est remplacée par la clause :

$$p(\tilde{T}) \leftarrow q(X_1, X_2, \ldots, X_k)$$

où

- *q est un nouveau symbole de prédicat, qui n'apparaît nulle part ailleurs dans P;*
- *$p(\tilde{T})$ est l'anti-unifié de tous les atomes de prédicat p qui apparaissent dans le corps d'une règle (ou dans G);*
- *les arguments de q sont les variables (distinctes entre elles) qui apparaissent dans $\tilde{T}$;*

La définition de q est alors :

$$\begin{array}{lll} \theta_1(q(X_1, X_2, \ldots X_k)) & \leftarrow & \theta_1(B_1) \\ \theta_2(q(X_1, X_2, \ldots X_k)) & \leftarrow & \theta_2(B_2) \\ & \vdots & \\ \theta_n(q(X_1, X_2, \ldots, X_k)) & \leftarrow & \theta_n(B_n) \end{array}$$

avec $\theta_i \tilde{T} = \tilde{t_i}$.

Et partout où apparaît un littéral $p(\tilde{t})$ dans P, il est remplacé par $\theta q(X_1, X_2, \ldots, X_k)$ tel que $\theta p(\tilde{T}) = p(\tilde{t})$.

Proposition 8 *L'indexation sur les corps de règles d'un programme et d'un but préserve l'équivalence opérationnelle forte.*

Preuve Nous allons montrer que cette transformation est une composition des transformations de définition, pliage et dépliage que nous avons déjà étudiées. Tous les symboles communs à la définition et à la preuve représentent les mêmes objets.

La première étape consiste à créer la définition :

$$q(X_1, X_2, \ldots, X_k) \leftarrow p(\tilde{T})$$

Cette règle est donc ajoutée à la fois au programme et à son ensemble de définition. On peut donc effectuer d'abord autant de pliages dans le programme et dans le but que possible (on vérifie toutes les conditions requises pour le pliage). Après cette étape, plus aucun appel à p n'est fait dans le programme (ni dans le but), sauf dans la définition de q. On déplie alors la définition de q.

Maintenant, p n'est plus du tout accessible, ni à partir du programme, ni à partir du but. On peut, dans un premier temps, supprimer les clauses définissant p. On peut donc ensuite créer la définition :

$$p(\tilde{T}) \leftarrow q(X_1, X_2, \ldots, X_k)$$

Il suffit maintenant d'effectuer un pliage sur le but (si le but fait appel à q) pour obtenir le programme indexé sur les corps de règles par rapport à p. □

La preuve de la seconde transformation (en calculant l'anti-unifié des têtes de règles) nécessite une transformation supplémentaire. Cette transformation est une généralisation de celle proposée en conclusion de l'article de Gallagher et Bruynooghe : *l'optimisation du premier appel.*

Définition 12 Optimisation des appels

Soient P un programme Prolog, G un but et p un prédicat défini par n clauses. Alors on peut placer devant chaque appel de p, noté $p(\tilde{t})$ (dans le corps d'une clause ou dans G) l'égalité $p(\tilde{T}) = p(\tilde{t})$, où $p(\tilde{T})$ est l'anti-unifié des têtes des n clauses définissant p.

Proposition 9 *L'optimisation des appels préserve l'équivalence opérationnelle forte.*

Preuve Par transformations (préservant l'e.o.f.) des deux programmes, on obtient le même programme. Dans chacun des deux programmes, pour chaque appel *optimisé* apparaissant dans une clause de la forme (*Avant* et *Après* représentent des conjonctions d'atomes quelconques) :

$$q(\tilde{t'}) \leftarrow Avant, p(\tilde{t}), Après.$$

ou

$$q(\tilde{t'}) \leftarrow Avant, p(\tilde{T}) = p(\tilde{t}), p(\tilde{t}), Après.$$

on crée une définition

$$def(\tilde{X}) \leftarrow p(\tilde{t}), Après.$$

ou

$$def(\tilde{X}) \leftarrow p(\tilde{T}) = p(\tilde{t}), p(\tilde{t}), Après.$$

et on effectue le pliage. Il suffit alors d'effectuer un (resp. deux) dépliage(s) sur l'appel de p (et sur l'égalité) dans les nouvelles définitions. □

Définition 13 Indexation sur les têtes de règles

Soient P un programme Prolog, G un but et p un prédicat (non dynamique) défini par n clauses :

$$\begin{aligned} p(\tilde{t_1}) &\leftarrow B_1 \\ p(\tilde{t_2}) &\leftarrow B_2 \\ &\vdots \\ p(\tilde{t_n}) &\leftarrow B_n \end{aligned}$$

La définition de p est remplacée par la clause :

$$p(\tilde{T}) \leftarrow q(X_1, X_2, \ldots, X_k)$$

où

- *q est un nouveau symbole de prédicat, qui n'apparaît nulle part ailleurs dans P;*
- $\tilde{T} = \bigwedge_{i \in [1,n]} \tilde{t_i}$;
- *les arguments de q sont les variables (distinctes entre elles) qui apparaissent dans $\tilde{T}$;*

La définition de q est alors :

$$\begin{aligned} q(\theta_1 X_1, \theta_1 X_2, \ldots, \theta_1 X_k) &\leftarrow B_1 \\ q(\theta_2 X_1, \theta_2 X_2, \ldots, \theta_2 X_k) &\leftarrow B_2 \\ &\vdots \\ q(\theta_n X_1, \theta_n X_2, \ldots, \theta_n X_k) &\leftarrow B_n \end{aligned}$$

avec $\theta_i \tilde{T} = \tilde{t_i}$.

Et partout où apparaît un littéral $p(\tilde{t})$ dans P, il est remplacé par $\theta q(X_1, X_2, \ldots, X_k)$ où θ est le mgu de $p(\tilde{t})$ et $p(\tilde{T})$.

Gallagher et Bruynooghe proposent une méthode plus sophistiquée qui utilise l'analyse statique du programme pour calculer une approximation de l'ensemble des appels de p qui soit au pire aussi précis que le calcul de l'anti-unifié.

Proposition 10 *L'indexation sur les têtes de règles préserve l'équivalence opérationnelle forte.*

Preuve La première étape consiste à calculer l'anti-unifié des têtes de règles définissant p, puis à appliquer la règle d'optimisation des appels. L'égalité engendrée par cette règle peut être supprimée ou seulement propagée comme spécifié dans la section suivante (1.3.1). La suite est similaire à l'indexation sur les corps de règles. Aucune information n'étant ajoutée aux têtes de règles, les substitutions θ_i ne sont pas répercutées sur les corps de règles. □

1.3 Extensions

1.3.1 Élimination des égalités

Dans une règle, la substitution associée à une égalité peut être propagée sur le corps de la règle à droite de l'égalité. Cela permet d'éliminer les égalités introduites par les opérations de dépliage. Néanmoins, on peut également la répercuter sur les atomes de gauche, tant que ceux-ci sont

- soit des prédicats prédéfinis dont le comportement n'est pas modifié par instanciation (cf. section 3.2.1);
- soit définis par l'utilisateur (sans faire d'appel à un prédicat prédéfini), si ceux-ci ont la propriété de terminer (cf. section 3.2.1).

Définition 14 Élimination des égalités
Soient P un programme Prolog, et C une clause de P de la forme :
$H \leftarrow B_1, \ldots B_{i-1}, B_i = B_{i+1}, \ldots B_n$
C peut être remplacée dans P par
$C' = H \leftarrow B_1, \ldots B_{i-1}, E_\theta, \theta(B_{i+2}, \ldots B_n)$
où θ est le mgu de B_i et B_{i+1}, et E_θ est le système d'équations associé à θ restreint aux variables de $Var(\theta) \cap Var(H)$ qui n'apparaissent pas dans $\{B_{i+1}, \ldots B_n\}$.

Proposition 11 *L'élimination des égalités par instanciation de la règle dans un programme Prolog conserve l'équivalence opérationnelle forte.*

Preuve Lévaluation de l'égalité correspond à effectuer un dépliage déterministe. Le *switching lemma* [Llo87] nous assure de la conservation de l'ensemble des solutions, et comme l'évaluation d'une équation est déterministe, l'ordre des solutions reste inchangé. □

1.3.2 Optimisation des appels

La transformation proposée par Gallagher et Bruynooghe peut encore être raffinée par rapport à la définition que nous en donnons (définition 12) : plutôt que de se restreindre à l'antiunifié, on peut essayer de distinguer les différents cas d'appels. Cette transformation peut servir à spécialiser finement une règle.

Définition 15 Spécialisation des appels
Soient P un programme Prolog, G un but et p un prédicat défini par n clauses défini par m clauses, dont on note les têtes de règles $p(\vec{r}_j)$, pour $1 \leq j \leq m$. Chaque corps de clause où p apparaît (Avant et Après représentent des conjonctions d'atomes quelconques) :

$$q(\vec{t}') \leftarrow Avant, p(\vec{t}), Après.$$

peut être remplacé par

$$q(\tilde{t'}) \leftarrow Avant, (p(\tilde{t}) = p(\tilde{T_1}); p(\tilde{t}) = p(\tilde{T_2}); \ldots p(\tilde{t}) = p(\tilde{T_n})), p(\tilde{t}), Après.$$

où

- "; " *représente la disjonction,*
- $n \leq m$,
- *chaque* $\tilde{T_i}$ *est l'antiunifié de quelques* $\tilde{r_j}$ *pour des* j *consécutifs,*
- *et* $\forall j \in [1,m]$, $\exists! i \in [1,n]$, *tel que* $\tilde{r_j} = \tilde{T_i}$ *soit solvable, et de plus* $\exists \theta$ *une substitution telle que* $\tilde{r_j} = \theta \tilde{T_i}$

Proposition 12 *La spécialisation des appels préserve l'équivalence opérationnelle forte.*

Preuve La preuve est tout à fait similaire à celle de l'optimisation des appels. Elle utilise en plus la sémantique opérationnelle de la disjonction, telle qu'elle est définie page 79. □

1.3.3 Variables anonymes

On peut alléger les conditions d'application du pliage lorsque la définition qui permet le pliage comporte des variables anonymes.

Définition 16 Variables anonymes
Une variable anonyme dans une clause est une variable qui apparaît une seule fois dans la clause, et uniquement dans la tête. Elle est notée par "_".

Lemme 1 *Si, pour un prédicat p, toutes les têtes de règles ont une variable anonyme comme* $n^{ième}$ *argument, alors on peut remplacer tous les appels à p dans le programme par un appel à p identique sauf pour le* $n^{ième}$ *argument qui devient une variable. Cette transformation préserve l'équivalence opérationnelle forte.*

Preuve Il suffit de faire un dépliage des appels à p dans les deux programmes et de comparer les systèmes d'équations dans les arbres de dérivation. Après un pas d'évaluation, les arbres sont égaux. □

Les variables anonymes qui ne correspondent pas à notre définition (qui se trouvent en corps de règle sans vérifier les conditions de la transformation) sont les variables spécifiées comme telles par le programmeur. Il faut faire la différence entre ces deux types de variables anonymes.

Proposition 13 *Si, pour un prédicat p d'arité m, toutes les têtes de règles ont une variable anonyme comme* $n^{ième}$ *argument* ($n \leq m$), *alors on peut remplacer tous les appels à p/m dans le programme par un appel à p/m-1, en supprimant le* $n^{ième}$ *argument (si p/m* $-$ 1 *n'est pas déjà défini dans le programme).*

Preuve La première étape consiste à remplacer le $n^{ième}$ argument de p/m par une variable anonyme comme expliqué dans le lemme 1. On fait ensuite une étape de définition en ajoutant la règle $p(X_1, \ldots\ X_{n-1}, X_{n+1}, \ldots\ X_m) \leftarrow p(X_1, \ldots\ X_m)$ au programme et à son ensemble de définitions. On effectue ensuite un dépliage de la nouvelle clause, puis un pliage de tous les appels à p/m par la nouvelle définition. Toutes les conditions pour effectuer le pliage sont vérifiées puisque la seule variable locale de la définition (X_n) est substituée par une variable anonyme. Après ces trois étapes, le prédicat p/m n'est plus accessible que directement à partir du but. □

1.4 Conclusion

L'équivalence opérationnelle que nous présentons semble à la fois

- correspondre à ce que peut désirer un utilisateur de Prolog: les transformations de programmes n'affectent pas le comportement "observable" du programme. Cette équivalence a donc un sens pour les programmes Prolog à effets de bord;

- être strict au niveau de la conservation de la structure de l'arbre SLD par rapport à la règle de sélection standard. Par les théorèmes d'indépendance de la règle d'évaluation et de forte complétude de la résolution SLD [Llo87], on sait donc que notre équivalence est plus forte qu'une équivalence en terme de plus petit modèle de Herbrand (ou ses équivalents: conséquences logiques et plus petit point fixe de T_P).

Les chapitres qui suivent illustrent donc deux utilisations différentes des techniques de manipulation syntaxique préservant l'e.o.f. Les chapitres 2 et 3 sont consacrés à l'utilisation des transformations de programmes comme techniques d'optimisation, tandis que le chapitre 4 porte sur les transformations de programmes comme outils de preuve dans le cadre de la méta-programmation.

Chapitre 2

Transformations de programmes logiques

"Étant donné un problème, trouver un programme (ou un ensemble de programmes) qui, d'une manière efficace, résout ce problème": voici, résumée par H.A. Partsch dans [Par90], la définition du développement de logiciels, de la conception à la réalisation. Cette tâche peut être décomposée en deux points principaux :

- à partir de l'énoncé d'un problème, informel et souvent incomplet, dégager une spécification formelle et complète (phase d'analyse);
- réaliser un programme efficace à partir de cette spécification (phase de programmation).

Le programme construit à partir de la spécification est d'abord un prototype, puis, par transformations successives (techniques d'optimisations, de compilation, ...), on doit obtenir un programme efficace. Utiliser des transformations formelles préservant la sémantique du programme permet de vérifier facilement pout tout programme, développé suivant cette méthodologie, sa correction par rapport à la spécification. C'est la technique *"règles + stratégie"*.

"L'une des idées principales de la programmation logique, due à R. Kowalski [Kow79] est qu'un algorithme est composé de deux composantes disjointes, la logique et le contrôle. La logique formule le problème à résoudre. Le contrôle formule comment le résoudre" [Llo87]. Les systèmes de programmation logique, tels que Prolog, sont déclaratifs, et sont donc un très bon outil pour le prototypage. Les transformations de programmes logiques permettent donc de passer du prototype à un programme Prolog plus efficace et gérant certaines contraintes d'implémentation comme l'indexation des têtes de clauses, ou bien la *Tail Recursive Optimization*, qui optimise les recursivités en fin de clause (la TRO est implémentée dans la Machine Abstraite de Warren (WAM)). En programmation logique, les transformations de programmes

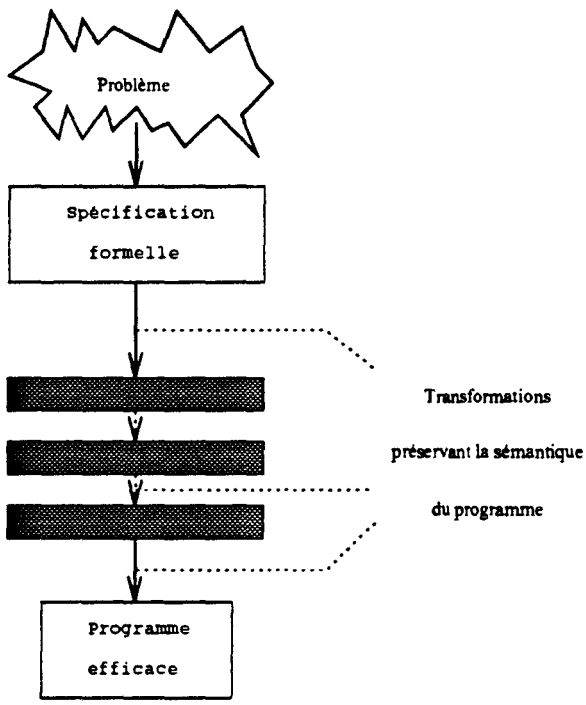


FIG. 2.1 - : [Par90] - Développement de logiciels et transformations de programmes

sont soit utilisées pour extraire un programme de spécifications écrites sous la forme de formules logiques du premier ordre (on parle alors de *synthèse de programmes*), soit utilisées pour améliorer l'efficacité d'un programme. On distingue alors plusieurs techniques de manipulations de programmes, comme la déduction partielle, l'évaluation partielle, la spécialisation de programmes, ... mais le nom générique le plus souvent employé est *évaluation partielle.* Ce chapitre est consacré à une présentation générale de différents travaux portant sur l'évaluation partielle. Ce sujet a suscité de nombreux travaux, aussi notre étude n'est pas exhaustive. Nous essayons simplement de dégager l'état actuel de la recherche dans ce domaine, et de faire ressortir les (nombreux) problèmes qui restent posés. Dans la première partie, nous nous attachons plutôt aux travaux sur la correction des diverses techniques de transformations de programmes logiques, tandis que la seconde partie porte sur différents systèmes et algorithmes d'évaluation partielle, qui utilisent les outils proposés dans la première partie. Parmi les nombreux documents qui nous ont aidés pour la rédaction de cette partie, nous devons mentionner le chapitre sur les transformations de programmes logiques de la thèse de A. Bouverot [Bou91].

2.1 Les règles

Les termes employés dans le cadre de l'optimisation des programmes logiques ne se recoupent pas toujours d'un article à l'autre: évaluation partielle, déduction partielle, spécialisation de programmes, transformations de programmes, pliage, dépliage, définition, reparamétrisation, ... Nous précisons donc les termes que nous utiliserons ici, en nous référant à ceux qui semblent être le plus communément adoptés:

- une *transformation de programme* est une opération de base dans la manipulation de programmes. C'est une technique formalisée, dont les propriétés sont connues, et qui correspond à une étape dans un algorithme d'optimisation de programmes, chaque étape n'apportant pas forcément un gain à l'exécution du programme. Les transformations sont utilisées dans bien d'autres cas que l'optimisation de programmes (cf. chapitres 1 et 4), mais seule cette application concerne ce chapitre;

- la *déduction partielle* est le résultat de l'application de l'évaluation partielle à la programmation logique pure: à partir d'un programme P et d'un but B, construire un programme P' qui donne les mêmes réponses que P pour B, et dont l'exécution est plus efficace;

- les programmes logiques réels contiennent beaucoup de prédicats prédéfinis non logiques et/ou à effets de bord. Comme le propose D. Sahlin dans sa thèse [Sah91], nous parlerons d'*évaluation partielle* à propos des systèmes manipulant des programmes Prolog avec effets de bord.

On peut distinguer (grossièrement) deux "écoles" dans les transformations de programmes. La première consiste à se doter de techniques diverses manipulant le programme directement à partir de son code ([BD77, TS84, GB90]). Ce sont les techniques de pliage, dépliage, factorisation de contraintes communes, ... La seconde consiste à manipuler le programme à partir de son arbre de dérivation SLD ([Kom81, LS87, BL89, MNL90]). Nous avons suivi ce plan pour présenter les règles de base de transformation de programmes.

2.1.1 Pliage/Dépliage

Les outils de base de la transformation de programmes sont les règles proposées par R.M. Burstall et J. Darlington ([BD77]) dans le cadre de la programmation fonctionnelle, et qui ont ensuite été étudiées et appliquées dans le cadre logique ([TS84]). Ce sont principalement les opérations de pliage et de dépliage.

Exemple 8
L'exemple que nous utilisons est proposé dans [BD77], mais nous le présentons sous forme de programme logique. Il s'agit d'un prédicat qui calcule le produit scalaire des n premiers éléments de deux vecteurs mis sous forme de listes :

```
produit_scalaire(_,_,0,0).
produit_scalaire([X|V1],[Y|V2],s(N),Res) :-
        produit_scalaire(V1,V2,N,ResPrime),
        Res is ResPrime + X * Y.
```

Avec ce prédicat, on en écrit maintenant un autre qui, à partir de quatre vecteurs, calcule le produit scalaire des deux premiers et des deux derniers, et renvoie comme résultat la somme des deux produits scalaires. Introduire une nouvelle règle, qui définit un nouveau nom de prédicat, est une étape de **définition**, et la règle ajoutée est un **eurêka** :

```
double_scalaire(V1,V2,V3,V4,N,Res) :-
        produit_scalaire(V1,V2,N,Res1),
        produit_scalaire(V3,V4,N,Res2),
        Res is Res1 + Res2.
```

On va essayer de modifier ce programme afin de ne pas séparer les deux calculs de produit scalaire. On effectue alors une première étape d'évaluation sur le littéral `produit_scalaire(V1,V2,N,Res1)` : on effectue un **dépliage**.

```
double_scalaire(_,_,V3,V4,0,Res) :-
        produit_scalaire(V3,V4,0,Res).
double_scalaire([X|V1],[Y|V2],V3,V4,s(N),Res) :-
        produit_scalaire(V1,V2,N,Res1),
        produit_scalaire(V3,V4,s(N),Res2),
        Res is Res1 + Res2 + X * Y.
```

Après deux nouvelles étapes de dépliage (sur la première et la seconde règle):

```
double_scalaire(_,_,_,_,0,0).
double_scalaire([X1|V1],[Y1|V2],[X2|V3],[Y2|V4],s(N),Res) :-
        produit_scalaire(V1,V2,N,Res1),
        produit_scalaire(V3,V4,N,Res2),
        Res is Res1 + Res2 + X1 * Y1 + X2 * Y2.
```

On remarque alors que

```
        produit_scalaire(V1,V2,N,Res1),
        produit_scalaire(V3,V4,N,Res2),
        Res is Res1 + Res2
```

est la définition de `double_scalaire(V1,V2,V3,V4,N,Res)`. On introduit alors une récursion (c'est l'étape de **pliage**), et grâce à l'associativité de l'addition, on obtient :

```
double_scalaire(_,_,_,_,0,0).
double_scalaire([X1|V1],[Y1|V2],[X2|V3],[Y2|V4],s(N),Res) :-
        double_scalaire(V1,V2,V3,V4,N,Res1),
        Res is Res1 + X1 * Y1 + X2 * Y2.
```

Ces trois opérations sont les opérations de base de tous les systèmes d'évaluation partielle. Le schéma général de l'évaluation partielle d'un programme est une séquence de dépliages, suivie de définitions ad hoc afin de pouvoir plier, et donc de récursiver, la définition d'un prédicat. La récursion (surtout terminale, i.e. à droite) est un facteur important de l'optimisation. Nous donnons maintenant les définitions de ces techniques. Un programme est un ensemble (et non pas une liste ordonnée) de clauses définies. Une clause définie est un couple (atome, multi-ensemble d'atomes), dont le premier élément est la tête, et le second le corps. On appelle séquence de transformations [KK88] P_0, P_1, ... P_N une séquence dans laquelle P_0 est le programme initial, et le programme P_{i+1} est obtenu en appliquant soit un pliage, soit un dépliage sur le programme P_i. Cette séquence de programmes évolue avec une séquence d'ensembles de définitions $D_0, D_1, \ldots, D_N$, qui contiennent les eurêka-prédicats. Au départ, $D_0 = \emptyset$.

Définition 17 *[TS84]* **Définition** *Soit C une clause de la forme $p(X_1, X_2, \ldots X_n) \leftarrow A_1, \ldots A_m$ où :*

1. *p est un nouveau nom de prédicat qui n'apparaît ni dans P_i ni dans D_i,*
2. *$X_1, X_2, \ldots X_n$ sont des variables distinctes,*

3. $A_1, A_2, \ldots A_m$ *sont des atomes dont les prédicats apparaissent dans* P_0.

Alors $P_{i+1} = P_i \cup \{C\}$, *et* $D_{i+1} = D_i \cup \{C\}$.

Définition 18 *[TS84]* **Dépliage** *Soient* P_i *un programme,* C *une clause de* P_i, A *un atome du corps de* C, *et* $C_1, C_2, \ldots C_k$ *les clauses de* P_i *dont la tête s'unifie avec* A *par les mgu* $\theta_1, \theta_2, \ldots \theta_k$. *Soit* C'_i *le résultat de l'application de* θ_i *sur* C *après avoir remplacé* A *par le corps de* C_i. *Alors* $P_{i+1} = (P_i - \{C\}) \cup \{C'_1, C'_2, \ldots C'_k\}$, *et* $D_{i+1} = D_i$. C *est appelé la clause dépliée, et* $C'_1, C'_2, \ldots C'_k$ *sont appelées les clauses dépliantes.*

Définition 19 *[TS84]* **Pliage** *Soit* P_i *un programme,* C *une clause de* P_i *de la forme*

$$A_0 \leftarrow A_1, A_2, \ldots A_n \quad (n > 0)$$

et D *une clause de* D_i *de la forme*

$$B_0 \leftarrow B_1, B_2, \ldots B_m \quad (m > 0)$$

Supposons qu'il existe une substitution θ *satisfaisant les conditions suivantes :*

1. $B_1\theta = A_{j_1}$, $B_2\theta = A_{j_2}$, $\ldots B_m\theta = A_{j_m}$, *où* $j_1, j_2, \ldots j_m$ *sont tous des indices distincts compris entre* 1 *et* n.

2. *à chaque variable locale de* D, θ *substitue une nouvelle variable n'apparaissant pas dans* $\{A_0, A_1, \ldots A_n\} - \{A_{j_1}, A_{j_2}, \ldots A_{j_m}\}$.

3. D *est la seule clause de* D_i *dont la tête s'unifie avec* $B_0\theta$.

4. *soit le prédicat défini par* C *appartient à* P_0, *soit* C *est le résultat d'au moins un dépliage dans la séquence* $P_0, P_1, \ldots P_i$.

Le résultat du pliage de C *par* D *est la clause* C' *de tête* A_0 *et de corps* $\{B_0\theta\} \cup (\{A_1, A_2, \ldots A_n\} - \{A_{j_1}, A_{j_2}, \ldots A_{j_m}\})$. *Alors* $P_{i+1} = (P_i - \{C\}) \cup \{C'\}$, $D_{i+1} = D_i$, C *est appelée la clause pliée, et* D *la clause pliante.*

Le pliage de [BD77] est beaucoup plus simple. Il est défini dans le cadre d'un langage d'équations récursives: *"Si* $E \Leftarrow E'$ *et* $F \Leftarrow F'$ *sont des équations et qu'il y a une occurrence d'une instance de* E' *dans* F', *remplacer cette occurrence par l'instance correspondante de* E. *Soit* F'' *le résultat, ajouter l'équation* $F \Leftarrow F''$ *au programme."* Cette définition correspond uniquement à la première condition du pliage selon [TS84], mais elle ne préserve pas la correction des solutions du programme transformé. Toutes les restrictions ont été rajoutées pour que le pliage soit correct par rapport à la sémantique du

plus petit modèle de Herbrand du programme transformé (ces conditions peuvent être résumées par *"si, juste après un pliage, on déplie $B\theta$, alors on doit retrouver C au nom des variables près"*):

- d'après la seconde condition, θ ne doit être qu'un renommage pour les variables locales de de D:

Exemple 9

P_1:
```
p(X,Y):-q1(X),q2(X,Y).
q(X,Y):-q1(X),q2(Z,Y).
q1(a).
q2(a,b).
q2(b,c).
```

ne peut pas être transformé en:

P_2:
```
p(X,Y):-q(X,Y).
q(X,Y):-q1(X),q2(Z,Y).
q1(a).
q2(a,b).
q2(b,c).
```

Pour le but `:- p(X,Y)` P_1 retourne la substitution-réponse {`X/a,Y/b`}, et P_2 retourne {`X/a,Y/b`} et {`X/b,Y/c`}.

Cette condition doit malgré tout être renforcée pour que la transformation soit correcte [She92]. Les nouvelles variables ne doivent pas non plus apparaître dans θG, sinon on pourrait obtenir l'erreur suivante:

Exemple 10

P_1:
```
h:-p(X,X).
q(X):-p(X,Y).
p(a,a).
p(a,b).
```

P_2:
```
h:-q(X).
q(X):-p(X,Y).
p(a,a).
p(a,b).
```

- l'exigence de l'unicité de la règle pliante s'illustre facilement:

Exemple 11

P_1:
```
p(a).
p(X) :- q(X).
q(b).
pp(X) :- q(X).
```

En suivant la définition de [BD77], on obtient :

P_2 :
```
p(a).
p(X) :- q(X).
q(b).
pp(X) :- p(X).
```

À la question `:- pp(X)`, P_1 renvoie la substitution-réponse `{X/b}`, tandis que P_2 renvoie deux solutions, `{X/b}` et `{X/a}`....

- la dernière restriction empêche une règle d'être pliée par rapport à elle-même :

Exemple 12

P_1 :
```
q(a)
p(X) :- q(X).
```
ne peut pas être transformé en P_2 :
```
q(a).
p(X) :- p(X).
```

P_2 n'est pas complet par rapport à P_1, et d'un point de vue opérationnel, le pliage a créé une branche infinie.

H. Tamaki et T. Sato complètent ces opérations avec quelques transformations auxiliaires :

- le remplacement de but : si deux ensembles d'atomes ont le même comportement (les mêmes conséquences logiques) et les mêmes variables locales à un renommage près, alors on peut substituer un ensemble par l'autre. C'est la règle qui est utilisée lorsqu'on se sert de propriétés données par l'utilisateur :

Exemple 13

`add(X,Y,Z),add(Z,T,ZZ)` (avec `Z` n'apparaissant pas dans le reste du corps de la clause) peut être remplacé par `add(X,T,Z),add(Y,Z,ZZ)`.

- la fusion de buts : deux atomes identiques dans le corps d'une clause peuvent être remplacés par un seul.

- la fusion de fonctions : soient $p(t_1,\ldots t_n,X), p(t_1,\ldots t_n,Y)$ deux atomes avec `p` fonctionnel, ils peuvent être remplacés par $p(t_1,\ldots\ t_n,X)$ en appliquant la substitution `{Y/X}` à toute la clause contenant ces deux atomes.

- les propriétés particulières : c'est l'utilisation de l'associativité, commutativité, distributivité, ...sur les prédicats qui vérifient ces propriétés. Cette règle est un cas particulier du remplacement de but.

H. Tamaki et T. Sato ont prouvé [TS84] que ces transformations préservent l'équivalence au sens du plus petit modèle de Herbrand entre les programmes transformés. Cette équivalence n'est pas assez fine pour caractériser les substitutions-réponses que renvoie un système Prolog. Par exemple, si on considère les deux programmes suivants ([KK88]) :

P_1 : `p(a).` $\qquad\qquad$ P_2 : `p(X).`

Si le but est `:- p(X)`, le premier programme retournera la substitution-réponse {`X/a`}, et le second la substitution-vide. Pourtant,ces deux programmes sont équivalents au sens du plus petit modèle de Herbrand. La sémantique de Clark [Cla79], basée sur l'univers de Herbrand étendu aux termes avec variables, permet de capturer cette différence. Cette sémantique a été étudiée par Deransart, Ferrand et Maluszyński [Fer87, DM93] (c'est le *modèle termal*) et par Falaschi, Levi, Palamidessi et Martelli [FLPM89] (c'est la *S-sémantique*). Le plus petit modèle de Clark d'un programme est également appelé *ensemble de succès d'un programme* par T. Kawamura et T. Kanamori. [KK88] et [BC93] ont montré que les transformations de définition, pliage et dépliage, telles qu'elles sont définies dans [TS84] préservent l'équivalence au sens du plus petit modèle de Clark des programmes transformés. C'est l'équivalence sémantique la plus forte qui est préservée par l'application sans restriction du dépliage. De plus, comme les transformations ne peuvent pas générer de nouvelles branches infinies, l'ensemble des échecs finis est conservé [Sek89]. Néanmoins, cette équivalence reste encore assez éloignée du comportement opérationnel des programmes logiques. Elle ne tient compte ni de la SLDNF, ni des stratégies de parcours des arbres SLD (et donc des différences de comportement dues aux branches infinies) ....

Exemple 14

P_1 :
```
p(X,Y):-q1(X),q2(Y).
q1(0).
q1(s(X)):-q1(s(X)).
q2(0).
q2(s(0)).
```

après un dépliage sur `q2(Y)`

P_2 :
```
p(X,0):-q1(X).
p(X,s(0)):-q1(X).
q1(0).
q1(s(X)):-q1(s(X)).
q2(0).
q2(s(0)).
```

Ces deux programmes ont même ensemble de succès et d'échecs finis, mais avec une stratégie Prolog standard (parcours de l'arbre SLD en profondeur d'abord, et sélection de l'atome le plus à gauche), les réponses calculées par le système sont différentes (cf. figures 2.2 et 2.3). Les branches infinies ont été déplacées de P_1 à P_2 par rapport à la deuxième solution. Avec une stratégie de parcours de l'arbre SLD en largeur d'abord, les réponses effectivement fournies par Prolog seraient les mêmes.

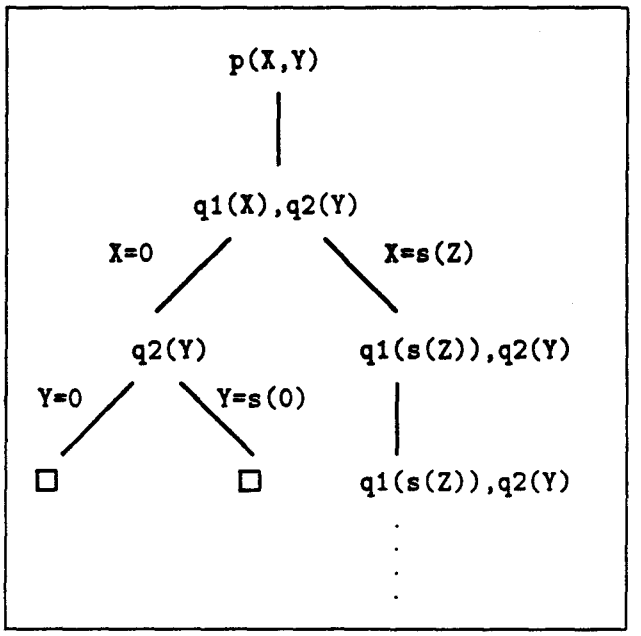


FIG. 2.2 - : Arbre de dérivation de P_1 pour :-p(X,Y)

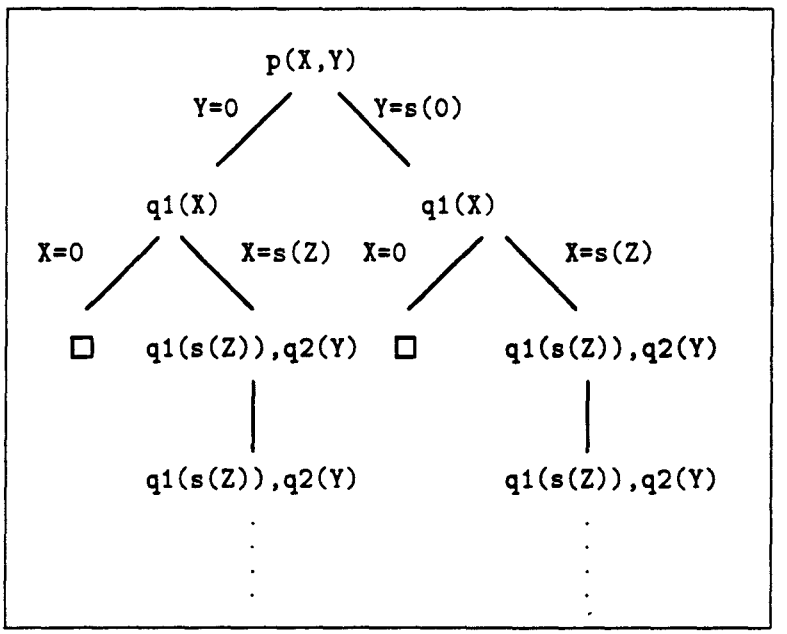


FIG. 2.3 - : Arbre de dérivation de P_2 pour :-p(X,Y)

2.1.2 Déduction partielle

La définition de la déduction partielle, au-delà des techniques de pliage/dépliage, est due à [Kom81], mais J.W. Lloyd et J.C. Shepherdson [LS87] ont étudié finement ses fondations théoriques. Avec la déduction partielle, on désire, à partir d'un programme P et d'un but B, obtenir un nouveau programme P' qui soit complet et correct par rapport à P et B. Soit P un programme logique normal. Informellement, une déduction partielle se calcule par rapport à un ensemble fini d'atomes $G =\leftarrow A_1, A_2, \ldots A_n$. Pour chaque A_i $(i \leq n)$, on construit un arbre SLDNF (à m branches), et on choisit exactement un but G_j $(0 \leq j \leq m)$ sur chaque branche non-échec. On remplace alors dans P les règles qui définissent le prédicat de A_i par les règles $\theta_j A_i \leftarrow G_j$ (les résultantes), avec θ_j substitution associée à la dérivation de G_j à partir de A_i. Le résultat est une déduction partielle de P par rapport à A_i. La déduction partielle de G dans P est l'union des déductions partielles de P par rapport à chaque A_i $(i \leq n)$ qui composent G.

Exemple 15
Soit P le programme suivant :

```
step(paris,londres).
step(londres,paris).
step(rome,bonn).
step(londres,bonn).
step(paris,rome).
flight(X,Y) :- step(X,Y).
flight(X,Y) :- step(X,I),flight(I,Y).
```

L'arbre SLD (partiel) pour le but `:- flight(X,Y)` est montré sur la figure 2.4.

Une déduction partielle possible est

```
step(paris,londres).
step(londres,paris).
step(rome,bonn).
step(londres,bonn).
step(paris,rome).
flight(paris,londres).
flight(londres,paris).
flight(rome,bonn).
flight(londres,bonn).
flight(paris,rome).
flight(X,Y) :- step(X,I),flight(I,Y).
```

Soit A un atome. Une déduction partielle du programme P par rapport à A en utilisant les dérivations SLD est une déduction partielle de P par rapport

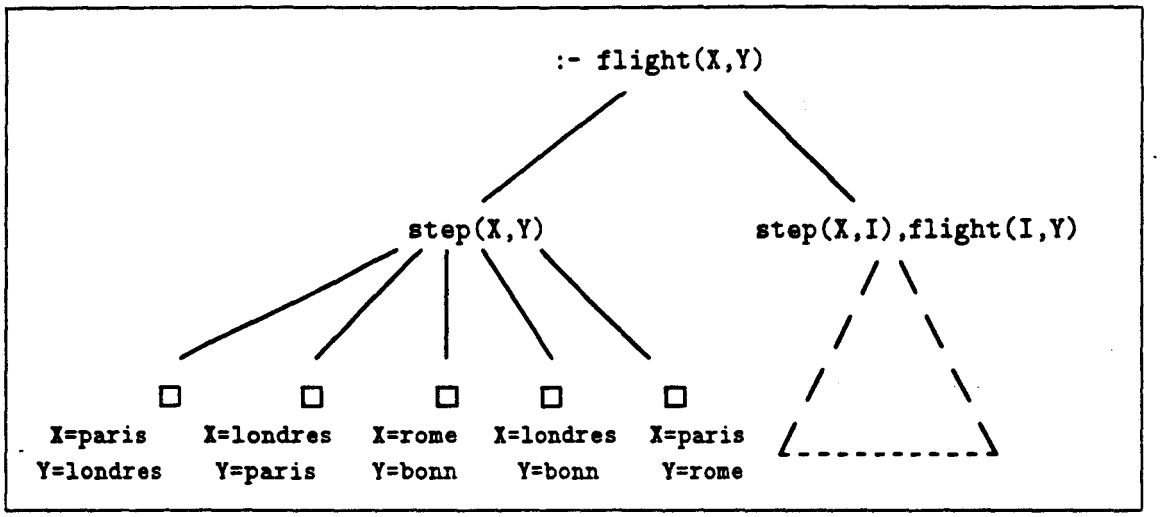


FIG. 2.4 - : Arbre SLD partiel pour :-`flight(X,Y)`

à A dans laquelle on a interdit toute étape de négation par l'échec (pas de sélection d'un littéral négatif).

Les buts de la forme :- `flight(X,rome),flight(paris,Y)` ne sont pas traités: comment effectuer l'union des déductions partielles? Lloyd et Shepherdson introduisent quelques restrictions pour préserver l'équivalence des succès et des échecs finis:

Définition 20 *[LS87] Un ensemble d'atomes A est singulier, si, pour chaque symbole de prédicat p, il y a au plus un atome dans A contenant p.*

Définition 21 *[LS87] Soit A un ensemble d'atomes et P un programme logique normal. P est A-clos si chaque atome de P de même symbole de prédicat qu'un élément de A est une instance d'un élément de A.*

Théorème 4 *[LS87] Soient P un programme normal, G un but normal, A un ensemble fini singulier d'atomes, et P' une déduction partielle de P par rapport à A en utilisant les dérivations SLD telle que $P' \cup \{G\}$ est A-clos. On a alors:*

1. *$P' \cup \{G\}$ a une réfutation SLD avec la substitution-réponse θ ssi $P \cup \{G\}$ l'a;*
2. *$P' \cup \{G\}$ a un arbre SLD finiment échec ssi $P \cup \{G\}$ l'a.*

Théorème 5 *[LS87] Soient P un programme normal, G un but normal, A un ensemble fini singulier d'atomes, et P' une déduction partielle de P par rapport à A tel que $P' \cup \{G\}$ est A-clos. On a alors:*

1. *$P' \cup \{G\}$ a une réfutation SLDNF avec la substitution-réponse θ ssi $P \cup \{G\}$ l'a;*

2. $P' \cup \{G\}$ a un arbre SLDNF finiment échec ssi $P \cup \{G\}$ l'a.

Une déduction partielle est une spécialisation du programme par rapport à un but par une succession de dépliages : on instancie les clauses définissant le prédicat contenu dans le but, puis on déplie celles-ci (aucun critère d'arrêt n'est donné). Cette approche peut donc être vue comme une généralisation du dépliage qui préserve la même équivalence opérationnelle. Ces résultats (sur la conservation de l'ensemble des substitutions-réponses calculées et de l'ensemble des échecs finis) sont obtenus sans restriction sur la règle de sélection choisie, ni pour construire l'arbre SLD (SLDNF) utilisé pour la déduction partielle, ni pour exécuter ensuite le programme. Les deux théorèmes sous-entendent *"$P' \cup \{G\}$ a une réfutation SLD (SLDNF) en utilisant une règle de sélection ssi $P \cup \{G\}$ a une réfutation SLD (SLDNF) en utilisant une (éventuellement différente) règle de sélection"*. Néanmoins, les théorèmes énoncés sont vrais pour le cas particulier où la même règle de sélection est utilisée pendant les trois dérivations (phase de déduction partielle, interprétation de $P \cup \{G\}$ et de $P' \cup \{G\}$), si :

1. la règle de sélection ne dépend que du but courant;

2. si le $i^{ème}$ littéral est sélectionné dans le but G, alors le $i^{ème}$ littéral est sélectionné dans le but θG.

La règle de sélection d'un interpréteur standard (sélection du premier atome à partir de la gauche) pour les programmes logiques définis vérifie ces deux conditions. Si en plus, les règles provenant d'une déduction partielle sont ajoutées au programme dans l'ordre (de la gauche vers la droite) où elles sont dans l'arbre SLD, alors la déduction partielle de [illegible] préserve notre équivalence opérationnelle forte (cf. [Kom93] pour une discussion sur la conservation de la structure SLD par la déduction partielle).

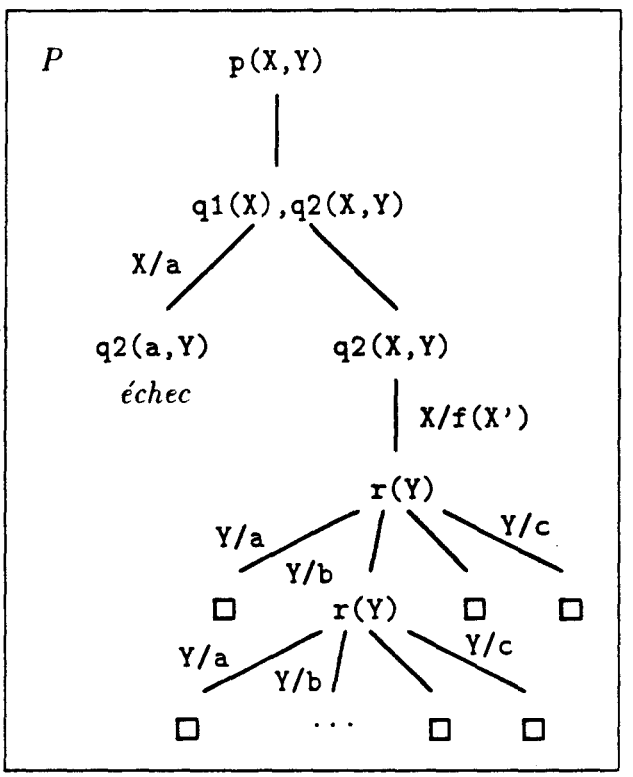
Exemple 16
Soit P le programme suivant :

```
p(X,Y) :- q1(X),q2(X,Y).
q1(a).
q1(X).
q2(f(X),Y) :- r(Y).
r(a).
r(Y) :- r(Y).
r(b).
r(c).
```

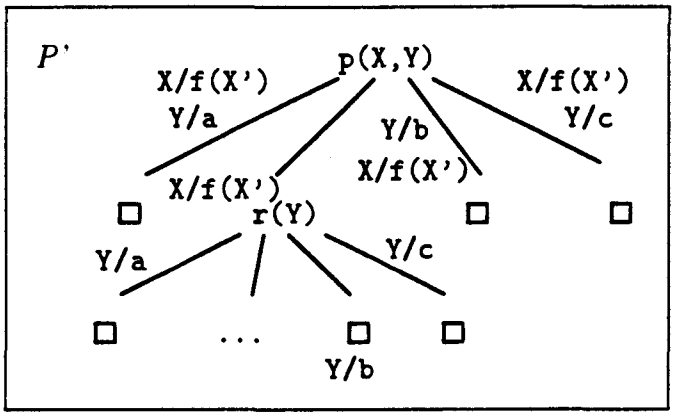
Le résultat d'une déduction partielle possible de P par rapport à `p(X,Y)` est le programme P' :

```
p(f(X),a).
p(f(X),Y) :- r(Y).
p(f(X),b).
p(f(X),c).
q1(a).
q1(X).
q2(f(X),Y) :- r(Y).
r(a).
r(Y) :- r(Y).
r(b).
r(c).
```

Les arbres de dérivation SLD de P et P' pour `:- p(X,Y)` sont :



et



Les branches succès et les branches infinies ne sont pas déplacées par la déduction partielle, seules certaines branches échecs sont supprimées. Cette technique peut être rapprochée de la spécialisation de programmes proposée par K. Marriott, L. Naish et J.-L. Lassez dans [MNL90]. Leur but est de spécialiser les programmes logiques par rapport à eux-mêmes, c'est-à-dire d'extraire les informations "cachées" d'un programme. Soit P un programme logique, une version plus spécifique de P, notée P', est un programme dont toutes les clauses sont des instances (des versions plus spécifiques) des clauses de P, et tel que $SS(P) = SS(P')$. Une version plus spécifique de P préservant les branches infinies, notée $P"$, est une version plus spécifique de P telle que $FF(P) = FF(P")$, et qui est donc cohérente avec la SLDNF. Quelque soit P un programme logique, la version la plus spécifique de P existe et est unique (au nom des variables près). K. Marriott, L. Naish et J.-L. Lassez donnent un algorithme qui s'arrête si et seulement si la version la plus spécifique d'un programme préserve les branches infinies. Soit B un littéral, on note $SD^k_P(B)$ l'ensemble des succès de B qui ont une dérivation succès dans P d'une profondeur $\leq k$, et $PD^k_P(B)$ l'ensemble des réponses partielles de B pour ses dérivations non-succès et non-échec dans P de profondeur k. On note *lub* $SD^k_P(B)$ l'anti-unifié de $SD^k_P(B)$ (*lub* est la notation anglaise pour le plus petit majorant, ici, par rapport à l'ordre $\leq$ sur les termes). La fonction $msg2_P(B)$ qui retourne la version la plus spécifique de B préservant les branches infinies est alors :

$k := 0$
répéter
  $k := k + 1$
jusqu'à lub $SD^k_P(B) \sim$ *lub* $SD^k_P(B) \cup PD^k_P$
return(lub $SD^k_P(B)$*)*

Exemple 17
La version la plus spécifique de l'exemple précédent préserve les branches infinies et est $P"$:

```
p(f(X),Y) :- q1(f(X)),q2(f(X),Y).
```

```
q1(a).
q1(X).
q2(f(X),Y) :- r(Y).
r(a).
r(Y) :- r(Y).
r(b).
r(c).
```

La branche échec qui se trouvait dans P pour le but `:-p(X,Y)` n'existe plus dans P".

Ils proposent également une fonction qui trouve (sous certaines conditions) la version la plus spécifique d'un programme (qui préserve *SS* mais pas *FF*). Cette fonction est basée sur l'opérateur de conséquence immédiate T_P, et calcule donc en mode bottom-up. Cette technique permet de calculer certaines informations qu'on ne peut pas obtenir par les transformations classiques, comme le dépliage. Si on reprend l'exemple précédent, on note P_1 le programme obtenu en changeant la règle `r(Y):-r(Y)` par `r(Y):-q3(Y)` et `q3(Y):-q3(Y)`. La version la plus spécifique de P_1 est identique à lui-même, sauf pour la définition du prédicat p, qui devient :

```
p(f(X),a).
p(f(X),b).
p(f(X),c).
```

FF n'est pas préservé par la version plus spécifique.

K. Benkerimi et J.W. Lloyd [BL89] ont proposé un algorithme qui génère une déduction partielle d'un programme à partir d'un but donné. Mais cet algorithme reste restrictif dans la mesure où dépliage et spécialisation de programmes ne permettent pas de récursiver la définition de certains programmes. Il est plus puissant que la technique proposée par [MNL90]- qui paraît plus adéquate pour le déboguage -, et possède des conditions naturelles pour garantir sa terminaison.

2.2 Les stratégies

La plupart des travaux sur l'évaluation partielle des programmes logiques utilisent les techniques de pliage et dépliage. Nous présentons différents types d'utilisation de ces techniques : les systèmes d'évaluation partielle automatique pour Prolog pur, d'abord, comme ceux de [PP93] et [Fuj87]. Puis nous étudions des stratégies de transformations qui ont pour cible une classe restreinte de programmes, mais qui utilisent des outils puissants d'optimisations : les différences de listes et l'utilisation de propriétés comme l'associativité, ainsi que des transformations qui visent à favoriser l'indexation des clauses dans la WAM.

Nous finissons le chapitre avec des évaluateurs partiels pour Prolog complet et les systèmes jumelant évaluation partielle et interprétation abstraite. Notre système se trouve à l'intersection de ces deux stratégies.

2.2.1 Deux algorithmes de déduction-évaluation partielle

Un algorithme générique de déduction partielle: (DFUR)* [PP93]

M. Proietti et A. Pettorossi proposent un algorithme générique de déduction partielle. Générique, car il permet de transformer un programme logique en fonction de ce que l'utilisateur veut (suppression des variables locales, récursion terminale ou linéaire, ...), et déduction partielle car cette procédure ne prend en compte que des programmes logiques purs. La stratégie (DFUR)* (definition-folding-unfolding-replacement) utilise les transformations définies par Tamaki et Sato: l'équivalence préservée est donc celle du plus petit modèle de Herbrand. L'algorithme a quatre paramètres:

- la propriété Φ que le programme résultat devra vérifier: elle doit être décidable et "locale", i.e. un programme vérifie Φ si chacune des clauses vérifie Φ. Plus formellement, si P_1 et P_2 sont deux programmes logiques, on doit avoir $\Phi(P_1)$ et $\Phi(P_2) \Leftrightarrow \Phi(P_1 \cup P_2)$.

- la fonction α, dite dp-fonction, ou fonction de définition-pliage: c'est elle qui gère la création des eurêka-prédicats et qui effectue les pliages correspondants. Elle dépend de la clause pour laquelle on veut créer un eurêka, des définitions déjà introduites, et du programme courant. Elle doit être cohérente avec la propriété Φ: soit C la clause qui vient d'être pliée, alors $\Phi(C)$ est vraie, sinon α est indéfinie pour C. Aucune variante ou instance d'un eurêka-prédicat déjà construit ne peut être créée[1];

- la fonction S de sélection de l'atome pour le dépliage;

- la fonction R de remplacement de but: cette fonction permet à l'utilisateur de spécifier les propriétés intéressantes de certains prédicats (fonctionnalité du prédicat `length`, associativité du prédicat **append** ...).

[1] Soient $H_1 \leftarrow Body_1$ et $H_2 \leftarrow Body_2$ deux définitions de deux eurêka-prédicats. Ces deux définitions sont des variantes l'une de l'autre ssi

- $Body_1 \sim Body_2$ et θ est la substitution de renommage telle que $\theta(Body_1) = Body_2$
- $\theta(Var(H_1)) = Var(H_2)$

Nous présentons maintenant l'algorithme. Les quatre fonctions sont définies, et la procédure est appliquée sur le programme logique P.

1. On divise P en deux ensembles de clauses *TransfP* et *RestofP*, tels que *TransfP* est le plus grand ensemble qui vérifie Φ.

2. Tant que *RestofP*$\neq \emptyset$

   (a) On choisit une clause C dans *RestofP*, et on effectue quelques étapes de définition et de pliage (par l'application de la fonction α), afin que $\Phi(\mathit{TransfP} \cup \{F\})$ soit vraie, si on appelle F la clause dérivée par des pliages de C.

   (b) Si les étapes de définition et de pliage n'ont pas été possibles, (parce que C est un eurêka, ou parce que α n'est pas définie pour C), alors

   - les ensembles *TransfP* et *RestofP* ne sont pas modifiés;

   sinon

   - F est ajoutée à *TransfP* et les définitions qui ont été créées pour plier C remplacent C dans *RestofP*.

   (c) On effectue quelques étapes de dépliage en fonction de S sur les clauses de *RestofP*, et si après ces étapes, pour un ensemble T de clauses de *RestofP*, on a $\Phi(\mathit{TransfP} \cup T)$, alors T est ajouté à *TransfP*.

M. Proietti et A. Pettorossi ont prouvé que si α est totale, et si l'ensemble des corps des définitions que α peut créer à partir de P est fini, alors l'algorithme (DFUR)* termine. Mais le problème général de l'arrêt de cette procédure reste indécidable.

Donnons d'abord un exemple de l'application de cette stratégie. L'exemple choisi est celui proposé par [TS84] et repris dans [PP89].

Exemple 18
Le programme suivant calcule les sous-listes communes à deux listes.

```
1 - csub(L1,L2,L3):-subseq(L1,L2),subseq(L1,L3).
2 - subseq([],_).
3 - subseq([X|L1],[X|L2]):-subseq(L1,L2).
4 - subseq([X|L1],[Y|L2]):-subseq([X|L1],L2).
```

- Propriété Φ : les corps de clause du programme résultat devront être linéaires par rapport à l'ensemble des variables de la tête de règle.

- Fonction S de sélection de l'atome pour le dépliage : l'atome choisi sera l'atome de plus petite profondeur[2], et si tous les atomes ont une profondeur égale, alors l'atome choisi sera l'atome de gauche.

- Fonction α de définition-pliage : soient deux atomes B_1 et B_2 d'un corps de clause, $link(B_1, B_2)$ est vrai si $Var(B_1) \cap Var(B_2) \neq \emptyset$. La clôture transitive de la fonction *link* partitionne un corps de clause en ensembles d'atomes. La fonction α créé une définition pour chacun de ces ensembles.

- Fonction R de remplacement de but : c'est la fonction *Identité*.

Seule la première clause ne vérifie pas la propriété Φ, on a donc au début *TransfP*=$\{2,3,4\}$ et *RestofP*=$\{1\}$. Par la fonction α on construit la définition suivante :

```
5 - new1(L1,L2,L3):-subseq(L1,L2),subseq(L1,L3).
```

et par pliage de la clause 1 avec la nouvelle définition, on obtient

```
6 - csub(L1,L2,L3):-new1(L1,L2,L3).
```

On procède alors à un dépliage de la clause 5 :

```
7 - new1([],_,L3):-subseq([],L3).
8 - new1([X|L1],[X|L2],L3):-subseq(L1,L2),subseq([X|L1],L3).
9 - new1(X|L1],[_|L2],L3):-subseq([X|L1],L2),subseq([X|L1],L3).
```

Les clauses 6 et 7 sont alors ajoutées à *TransfP*, et *RestofP*=$\{8,9\}$. On procède à nouveau à une étape de définition-pliage. Pour la clause 9, α ne crée pas de nouvelle définition, puisqu'on peut directement effectuer un pliage avec la définition 5 :

```
10 - new1([X|L1],[_|L2],L3):-new1([X|L1],L2,L3).
```

Pour la clause 8, on définit l'eurêka suivant :

```
11 - new2(X,L1,L2,L3):-subseq(L1,L2),subseq([X|L1],L3).
```

et le résultat du pliage est alors :

```
12 - new1([X|L1],[X|L2],L3):-new2(X,L1,L2,L3).
```

[2]La profondeur d'un atome est donnée par la profondeur de ses arguments. La profondeur d'un terme est 0 pour une constante, et pour un terme de la forme $f(t_1, t_2, \ldots t_n)$, la profondeur est $1 +$ max($profondeur(t_1)$,$profondeur(t_2, \ldots t_n)$).

La fonction de dépliage S sélectionne le deuxième atome de la clause 11 :

```
13 - new2(X,L1,L2,[X|L3]):-subseq(L1,L2),subseq(L1,L3).
14 - new2(X,L1,L2,[Y|L3]):-subseq(L1,L2),subseq([X|L1],L3).
```

Avant d'effectuer la dernière étape de définition-pliage, *TransfP*={2,3,4,6,-7,10,12}, et *RestofP*={13,14}. α ne construit pas de nouvelle définition pour 13 et 14, et le résultat du pliage est :

```
15 - new2(X,L1,L2,[X|L3]):-new1(L1,L2,L3).
16 - new2(X,L1,L2,[Y|L3]):-new2(X,L1,L2,L3).
```

Ces deux clauses vérifient la propriété Φ, *RestofP* est donc vide, la procédure s'arrête. Le programme résultat est donc constitué des clauses *TransfP* = {2,3,4,6,7, 10,12, 15,16}. En effectuant un dépliage sur la clause 7, on peut encore légèrement réduire le programme.

(DFUR)* est suivi d'une procédure de contraction du code qui effectue des dépliages sur les prédicats non récursifs afin d'éliminer certains eurêka-prédicats superflus (le problème de savoir si un prédicat peut être éliminé par dépliages est indécidable). Cette procédure de contraction ne réduit pas le programme dans cet exemple.

Cette stratégie est donc limitée par le fait qu'elle ne traite que les programmes logiques purs et définis et par le problème de l'arrêt. Un autre point noir est que la définition des fonctions de définition-pliage et de sélection soit laissée à la charge de l'utilisateur. Mais elle est très souple sur le choix de la fonction Φ, et peut être, par exemple, guidée par des informations sémantiques (M. Proietti et A. Pettorossi donnent un exemple écrit avec le langage Gödel [HL91]). Surtout, elle permet de faire des transformations non triviales de manière automatique.

Un algorithme d'évaluation partielle avec contraintes

H. Fujita [Fuj87] propose un algorithme plus particulièrement destiné à la spécialisation de programme (par rapport à un but assez fortement instancié). Cet algorithme se déroule en deux phases. Dans la première étape, appelée étape de spécialisation, le programme est déplié, et à chaque dépliage, une nouvelle définition est créée pour rendre compte des différents types d'appels possibles pour un prédicat. Lorsqu'on reconnaît un appel de procédure spécialisé (i.e. un appel déjà traité), on utilise la définition correspondante. Lorsqu'on rencontre une primitive, on l'évalue si possible, on la laisse inchangée sinon. À la fin de cette étape (pour lequel la terminaison n'est pas garantie, puisque l'algorithme ne détecte que des boucles simples), on a généré un grand nombre de règles superflues. Ces règles sont alors supprimées par la seconde phase

de l'algorithme qui calcule les *contraintes communes* à tous les appels d'un même atome. Les règles dont la tête est unifiable avec cet atome et qui ne sont pas cohérentes avec ces contraintes sont alors effacées. Cette seconde phase d'élimination peut être rapprochée de la fonction *msg2* proposée par [MNL90]. D. Smith et T. Hickey ont proposé diverses améliorations de cet algorithme [SH90]. Ils se situent dans le cadre d'un langage de programmation logique avec contraintes (égalités et diségalités) sur les arbres finis *CLP(FT)*. Ce cadre leur permet de rassembler les deux phases de l'algorithme en une seule, puisque les contraintes sur les appels sont directement liées aux atomes et peuvent donc être évaluées au moment du dépliage. D'autre part, leur détection de boucles plus perfectionnée (basée sur la même idée de J. Gallagher que dans [BL89]), garantit la terminaison de leur stratégie. Il s'agit d'une borne sur la profondeur des termes traités : au-delà de cette borne, les sous-termes sont remplacés par des variables (c'est une sorte d'abstraction). Le nombre de procédures spécialisées est donc limité. Ils proposent également un calcul des contraintes communes à tous les appels d'un même prédicat, afin de retirer cette information redondante du programme. Cette technique (*lifting constraint* suivi de *factorization*) correspond à ce qui est proposé dans l'article de J. Gallagher et M. Bruynooghe [GB90], mais étendu à des systèmes de contraintes comportant des égalités et des diségalités sur les arbres finis (i.e. sur l'univers de Herbrand). Cet algorithme ne semble néanmoins pas pouvoir se généraliser aux prédicats à effets de bord.

2.2.2 Quelques raffinements de transformations

Autour des transformations de base que nous avons présentées dans la section précédente, se sont développées une quantité d'autres transformations qui contribuent à améliorer les techniques offertes par les systèmes d'évaluation partielle.

Éliminer les informations redondantes

J. Gallagher et M. Bruynooghe [GB90] proposent une transformation qui vise à retirer d'un programme les symboles de fonction et les constantes superflus dans un programme (cf. 1.2.4).

Exemple 19
[GB90]Soit le programme d'inversion de liste écrit avec des listes différentielles :

```
rev(L1,L2) :- reverse(L1,L2-[]).
reverse([],L-L).
reverse([X|L1],L2-L3) :- reverse(L1,L2-[X|L3]).
```

Le symbole de fonction - des listes différentielles n'intervient pas dans le calcul de l'inversion de la liste. Il peut, sans modification du comportement du programme pour n'importe quel but portant sur le prédicat **reverse/2**, être éliminé.

```
rev(L1,L2) :- reverse(L1,L2,[]).
reverse([],L,L).
reverse([X|L1],L2,L3) :- reverse(L1,L2,[X|L3]).
```

Pour le premier programme, à chaque fois qu'un appel au prédicat **reverse/2** est créé, l'interpréteur doit stocker une structure de données complexe pour le deuxième argument. Dans cet exemple très simple, le gain en espace est déjà de 50%. Mais le bénéfice est gagné à plusieurs niveaux :

- en supprimant ce foncteur, l'unification est plus rapide,
- si le prédicat simplifié est défini par plusieurs règles, la transformation favorise l'indexation des clauses.

La technique proposée dans [GB90] spécialise une procédure pour un but (ici, pour le but `rev(L1,L2)`, qui est équivalent au but `reverse(L1,L2-[])`).

Définition 22 *[GB90] Soit P un programme contenant une procédure consistant en n clauses $p(t_1) \leftarrow B_1, \ldots, p(t_n) \leftarrow B_n$. Soit $p(s)$ un atome. On définit une procédure appelée la spécialisation de p pour $p(s)$. Soient $X_1, \ldots X_k$ les variables (distinctes entre elles) apparaissant dans $p(s)$ dans l'ordre de leur première occurrence (on suppose que $X_1, \ldots X_k$ n'apparaissent pas dans P). Soit q un nouveau symbole de prédicat (n'apparaissant pas dans P). Soit θ_i le mgu de $p(s)$ et $p(t_i)$, pour $1 \leq i \leq n$. La procédure pour q est l'ensemble des clauses*

$$q(X_1, \ldots X_k)\theta_i \leftarrow B_i\theta_i \text{ pour tout } i \text{ tel que } \theta_i \neq fail$$

Si, par analyse statique, on arrive à déterminer, pour chaque prédicat p de P, un atome $p(s)$ qui caractérise tous les appels de p (tout appel du prédicat p est une instance de $p(s)$), alors on peut remplacer la définition de p par une spécialisation de p pour $p(s)$. [GB90] propose une méthode pour calculer une approximation de l'ensemble des appels de p. Le principe du renommage d'un atome par un nouveau symbole de prédicat, avec comme arguments les différentes variables apparaissant dans l'atome, est appelé **reparamétrisation** (*reparameterization*) par T. Hickey ([SH90, HM89]). Une des optimisations de la WAM est la sélection rapide des règles qui peuvent s'unifier avec le but courant. Cette pré-selection avant unification se fait en comparant le premier argument du but courant et de la tête de règle. Si les deux termes sont compatibles à une profondeur 1 (deux variables, ou deux fois le même foncteur, ou un foncteur et une variable), alors la règle est sélectionnée. Cette technique est

appelée l'indexation de clauses. Le fait de renommer des procédures pendant une pré-compilation (ou phase d'analyse statique) permet d'éliminer les cas triviaux ($\theta_i = fail$) et d'amener au niveau supérieur les différences entre les clauses.

Utiliser l'associativité et les différences de listes

Nous présentons d'abord les structures de différences [CT77, SS86], ou différence de termes. Une différence de termes est un terme de la forme $T - T'$, où T et T' sont des termes classiques. Les termes de différences permettent de représenter les termes usuels. Le terme $T - T'$ représente le terme T dans lequel chaque occurrence de T' a été remplacé par une constante. T' est appelé le délimiteur de $T - T'$. Ainsi, on notera la liste `[a,b]` par la différence de listes `[a,b|L]-L`, et l'entier `s(0)` par `s(N)-N`. L'utilisation de structures de différences dans un programme nécessite l'emploi de l'unification avec occur-check.

Beaucoup de travaux ont été menés pour automatiser les transformations de programmes logiques dans le cas du traitement de listes, et même plus particulièrement dans le cas du prédicat **append/3**. Nous illustrons ces méthodes en commençant par un exemple classique.

Exemple 20
Le **reverse** naïf est le programme suivant :

```
1 - append([],L,L).
2 - append([X|L1],L2,[X|L3]) :-
    append(L1,L2,L3).
3 - reverse_naif([],[]).
4 - reverse_naif([X|L1],L2) :-
    reverse_naif(L1,L3), append(L3,[X],L2).
```

On ajoute la définition suivante :

```
5 - reverse(L1,L2,X) :-
    reverse_naif(L1,L3), append(L3,X,L2).
```

On effectue maintenant un dépliage de la définition :

```
6 - reverse([],X,X).
7 - reverse([Y|L1],L2,X) :-
    reverse_naif(L1,L3), append(L3,[Y],L4), append(L4,X,L2).
```

Par application de l'associativité du **append**, on obtient :

```
8 - reverse([Y|L1],L2,X) :-
    reverse_naif(L1,L3), append(L3,[Y|X],L2).
```

On peut maintenant effectuer les deux pliages (sur les clauses 4 et 8) par rapport à la définition (clause 5).

```
9 - reverse([Y|L1],L2,X) :-
    reverse(L1,L2,[Y|X]).
10- reverse_naif([X|L1],L2) :-
    reverse(L1,L2,[X]).
```

Le programme final est constitué des clauses {1,2,3,10,6,9}.

Par application des transformations proposées par Tamaki et Sato, on passe de la définition du **reverse_naif** à la définition du **reverse** avec accumulateur (qui est un cas particulier des différences de listes). J. Zhang et P.W. Grant [ZG88] proposent un algorithme déterministe et qui termine pour effectuer cette transformation. Pour chaque clause dont le corps contient des buts de la forme $p(X_1,\ldots,X_i,\ldots,X_n)$, ...append($X_i$, X,Y) (la variable X_i doit vérifier certaines hypothèses afin déviter que le processus ne se réitère à l'infini), on construit la définition

newp($X_1,\ldots,Y,\ldots,X_n$,X) :- $p(X_1,\ldots,X_i,\ldots,X_n)$, append($X'_i$,X,Y).

On effectue alors une opération de pliage (par rapport à ces définitions) sur toutes les clauses candidates, puis on essaie de transformer les définitions, par dépliage, application de l'associativité du **append** et enfin pliage. Si le résultat contient encore un appel au prédicat p, alors le programme original n'est pas modifié, sinon, on remplace le programme original par celui résultant de l'algorithme. Pour le programme **reverse_naif**, on obtient le **reverse** de l'exemple 19. Néanmoins, leur méthode reste très restrictive, puisqu'elle ne traite que les prédicats faisant appel à **append**.

D.R. Brough et C.J. Hogger [BH87] ont étudié d'une manière plus générale comment utiliser l'associativité des prédicats. Ils formalisent l'associativité en trois axiomes, et définissent un schéma général pour les prédicats récursifs. À partir de ce schéma et d'éventuelles propriétés supplémentaires des prédicats (comme la fonctionnalité), ils dérivent un algorithme de transformation qui préserve l'ensemble des succès du programme. Les programmes passent d'une récursion linéaire à une récursion terminale. Cette méthode, bien que ne les utilisant pas explicitement, est très proche des systèmes manipulant des structures de différences ([SS86, MS93]). Les axiomes utilisés par Brough et Hogger pour formaliser l'associativité peuvent être formalisés à l'aide des structures de différences. Ainsi, cette méthode donne le même résultat que celle de Zhang et Grant lorsqu'elle est appliquée au **reverse_naif**.

La dernière méthode que nous présenterons concernant les différences de listes (ou d-listes) est plus complexe, et mêle transformation de programmes et analyse statique (analyse de mode, inférence de types, ...). K. Marriott et H. Søndergaard [MS93] étudient la même classe de programmes que [ZG88], celle des programmes utilisant le **append**. Ils s'intéressent uniquement aux d-listes

fermées, i.e. les listes de la forme $[X_1, X_2, \ldots X_n \mid L] - L$, avec $n \geq 0$. Ces d-listes peuvent faire passer la concaténation de listes d'un temps linéaire en la longueur de la première liste (avec le prédicat **append**) à un temps constant. Si le premier argument du **append** est sous la forme d'une d-liste fermée, **append** devient `app*(L1-L2,L2,L1)`. Le délimiteur de la d-liste joue alors le rôle d'un pointeur pour effectuer le append. Si les trois arguments sont des d-listes fermées, alors la nouvelle définition de **append** est `app**(L1-L2,L2-L3,L1-L3)`. Lorsque le second et/ou le troisième argument sont des d-listes fermées, **append** ne peut pas être optimisé par les d-listes. L'introduction de d-listes n'est pas toujours correcte : même dans le cas du **reverse**, la transformation n'est correcte que si l'utilisateur lance un but avec comme premier argument une liste close (de la forme $[X_1, X_2, \ldots X_n]$, $n \geq 0$). Le problème se situe à deux niveaux différents : l'unification des d-listes n'est pas forcément cohérente avec l'unification des termes qu'elles représentent (`[]-[]` et `[a]-[a]` ne s'unifient pas alors que ces deux termes représentent `[]`); d'autre part, la transformation d'un **append** en un **app*** ou **app**** est liée à certaines conditions d'application (celles que nous avons décrites précédemment, plus d'autres contraintes sur le partage des variables entre les arguments du **append**). L'algorithme de Marriott et Søndergaard est décomposé en trois étapes. La première consiste, à partir d'un *marquage* du but fourni par l'utilisateur qui indique quels sont les arguments qui seront des d-listes fermées à l'exécution, à répercuter ce marquage sur tout le programme. À la fin de cette étape, les arguments marqués sont transformés en d-listes, et des appels au prédicat prédéfini **var** sont ajoutés afin de garantir la correction de l'unification sur les d-listes. Les deux phases suivantes nécessitent une analyse statique assez fine du programme (par des techniques d'interprétation abstraite par exemple). Les endroits où une concaténation rapide de listes peut être effectuée (où la transformation est correcte) sont détectés et modifiés (remplacement des appels à **append** par des appels à **app*** et **app****) dans la seconde étape. La troisième étape supprime les appels à **var** là où l'analyse statique le permet.

Cette méthode reste donc restreinte aux programmes de traitement de liste, mais elle étudie d'une manière précise les conditions sous lesquelles l'introduction de d-listes est valide (l'algorithme préserve l'ensemble des succès et la terminaison) et fait le lien entre transformations de programmes et interprétation abstraite. De plus, leur algorithme, comme celui de [BH87] et [ZG88] est déterministe et termine.

2.2.3 Deux systèmes automatiques d'évaluation partielle

De nombreux systèmes d'évaluation partielle existent (SPES [ABFQ92], ProMiX [Lak89], TREQUASI [Azi87], le système de Fujita [Fuj87], ...) mettant en pratique les algorithmes et les transformations que nous avons vus

précédemment. Nous avons restreint notre étude à deux systèmes

- qui sont **automatiques** : ils ne requièrent pas de directives de l'utilisateur, tout au plus à travers certains paramètres faisant varier le degré de précision de l'évaluateur (et son temps d'exécution);

- qui manipulent Prolog complet (i.e. avec prédicats prédéfinis), ce qui leur permet de traiter n'importe quel programme réel.

PADDY [Pre92a] et MIXTUS [Sah91] ont été développés le premier autour de Sépia Prolog, dans le projet Eclipse de l'ECRC, le second autour de Sicstus Prolog, chez SICS. Ces deux systèmes présentent de nombreux points communs. Ils préservent la même équivalence des programmes (même séquence de substitutions-réponses, avec éventuellement suppression de branches infinies apparaissant dans le programme original), très proche de la nôtre, et sont basés sur les opérations de pliage, dépliage et définition. Les différences se trouvent plus sur le traitement des prédicats prédéfinis et sur la prévention de boucles.

Le traitement des prédicats prédéfinis est différent dans ces deux applications. Ils passent tous les deux par une classification des prédicats, suivant leur comportement (nous nous inspirons de celle proposée dans PADDY, aussi nous la détaillerons plus loin). Il y a deux approches différentes des structures de contrôle :

- celle qui propose des structures de contrôle ayant une sémantique plus "propre", et en particulier compatible avec les techniques de pliage et dépliage (par exemple [Hil91]). PADDY appartient à cette famille.

- celle qui adapte le pliage/dépliage aux structures de contrôle, et les emploie, car la disjonction en particulier est implémentée d'une manière très efficace dans la plupart des Prolog. MIXTUS appartient à cette seconde famille.

PADDY simplifie le programme à optimiser, en supprimant les structure de contrôle `not/1`, `once`, `!` et $\leftarrow$ et `;`. Le résultat est un programme contenant uniquement les structure de conjonction (`,`) et des *cuts ancestraux*[3][Pre92b]. Le résultat est un programme pouvant contenir de tels cuts ancestraux. MIXTUS, par contre, traite directement ces structures de contrôle lors de son exécution, et même rajoute des **if-then-else** et surtout des disjonctions lors du dépliage.

La prévention de boucles est plus rapide dans PADDY que dans MIXTUS. Mixtus regarde la liste des ancêtres d'un but avant de le déplier, tandis que

[3]Le remplacement des cuts par les cuts ancestraux est une technique déjà proposée par R.A. O'Keefe [O'K90]. Elle consiste à décomposer le cut en deux littéraux, `mark/1` et `!/1`. L'argument sert à *lier* le cut à son *père* (la clause qui l'a introduit dans la dérivation), et permet donc d'appliquer le dépliage.

PADDY utilise une table des définitions déjà créées, et recherche les buts similaires déjà traités à l'aide d'une fonction de hachage. Cette modification lui permet d'obtenir des temps d'exécution de son évaluateur partiel rapides.

Enfin, la stratégie globale (pour Prolog pur) des deux systèmes est assez proche. Leurs règles de dépliage et définition sont semblables à celles de [PP91, PDL91], mais leur technique de pliage en est une restriction, puisqu'ils ne plient qu'un seul atome, jamais une séquence d'atomes. C'est ce qui leur permet d'automatiser facilement la création des eurêka-prédicats, mais c'est aussi ce qui les limite dans la modification des programmes : ils ne peuvent pas traiter, par exemple, le cas du **reverse** (passer du **reverse_naif** au **reverse** avec accumulateur), ni rendre récursif un prédicat qui ne l'était pas. Ils sont surtout efficaces lorsque le but est suffisamment instancié pour spécialiser le programme par rapport à cet appel.

2.2.4 Interprétation abstraite et évaluation partielle

L'évaluation partielle semble naturellement liée à l'interprétation abstraite. L'interprétation abstraite (ou toute autre forme d'analyse statique des programmes) calcule des informations sur les programmes (en général sur le comportement déterministe de certains prédicats, l'analyse des modes d'entrée-sortie, l'inférence de types , ...), et l'évaluation partielle peut utiliser ces informations afin soit d'augmenter le champ d'application des transformations de base, soit de guider sa stratégie. Il existe plusieurs articles traitant de ces liens ([GB91, GCS88, MS93]...). Nous avons choisi de présenter un système basé sur la résolution OLDT abstraite, comme celle que nous utilisons dans notre système.

D. Boulanger et M. Bruynooghe [BB92] proposent un système de transformations de programmes guidées par une interprétation abstraite basée sur la OLDT résolution (*Ordered Linear resolution for Definite clauses with Tabulation* [TS86]). La OLD résolution est une résolution SLD particulière, la règle de sélection de l'atome étant celle de l'atome le plus à gauche. La OLD résolution avec une table mémorise les atomes sélectionnés et leurs solutions. À chaque nœud de l'arbre OLD, on compare le but courant aux atomes de la table. S'il existe une variante (ou une instance) du premier atome du but dans la table, alors le but "pointe" sur les solutions de l'atome de la table (on est sur un nœud passif), on arrête sa résolution, et on sélectionne l'atome suivant du but. Sinon, on crée une entrée pour le but (on est sur un nœud actif) dans la table, et on continue la résolution de ce but. Cette méthode, si elle est combinée avec une abstraction (généralisation des atomes rencontrés), termine et permet de capturer l'essence du comportement opérationnel du programme. [KKH87] semble avoir été le premier articledans lequel est formalisée cette combinaison. Boulanger et Bruynooghe, étant donné un programme et un but, construisent une telle structure, et, à partir de là, effectuent des dépliages

et des pliages (on retrouve cette idée chez [LS87, BL89]). Dans une première phase, un ensemble de définitions est généré, qui est divisé en deux ensembles : les définitions à déplier, et les définitions pliantes. L'ensemble des clauses à déplier est fourni par les résultantes associées aux nœuds actifs, et l'ensemble des clauses pliantes est fourni par les solutions des nœuds actifs. Leur pliage ne respecte pas les conditions de correction de [TS84], mais leur système préserve quand même l'équivalence des programmes au sens de l'ensemble des succès. La deuxième phase consiste à effectuer les pliages et les dépliages en utilisant les définitions créées et les clauses du programme original. Cette deuxième phase est guidée par la structure OLDT abstraite du programme. Le programme obtenu est alors optimisé par des dépliages classiques afin d'éliminer les prédicats superflus générés aux étapes précédentes.

Cette méthode peut tenir compte de certains prédicats prédéfinis (qui ne sont pas des structures de contrôle) et de prédicats définis dans d'autres modules, par l'utilisation d'une table annexe qui caractérise ces prédicats et donne une approximation de leurs solutions. Suivant l'abstraction choisie pendant la construction de la structure OLDT, Boulanger et Bruynooghe retrouvent les mêmes résultats que [PP90, PP91], et leur méthode permet d'implémenter en plus les techniques proposées par [GB90] et [GCS88].

2.3 Conclusion

Il existe un grand nombre de travaux portant sur les transformations de programmes logiques purs, basés essentiellement sur l'équivalence au sens de l'*ensemble des succès*. Cependant, les articles portant sur l'évaluation partielle de programmes Prolog complets considèrent les *séquences de substitution-réponses*. Ces systèmes sont alors très limités quant aux optimisations qu'ils peuvent effectuer, ceci étant dû aux restrictions d'application des opérations de pliage et dépliage pour préserver cette équivalence. Il semble donc nécessaire de jumeler l'évaluation partielle de Prolog complet à une analyse statique des programmes, afin de compenser ces restrictions. Mais les problèmes fondamentaux de l'évaluation partielle ne semblent pas prêt d'être résolus.

Dans [BD77], Burstall et Darlington posent les questions suivantes :

1. quelle est la taille de la classe des programmes qui peuvent être améliorés par les techniques de pliage/dépliage?

2. quelles sont les conditions (nécessaires et suffisantes) qui garantissent que de telles transformations apportent réellement des améliorations?

Des embryons de réponses ont été apportées à ces questions, mais pas de réponse générale. Certains travaux, comme [ZG88, BH87, MS93] définissent la classe des programmes que leurs transformations améliorent. D'autres justifient l'optimisation qu'ils apportent : suppression de parcours de listes ([ZG88,

MS93]), déterminisation du processus ([BH87]), simplification de l'unification et favorisation de l'indexation des clauses ([GB90]), suppression de variables locales inutiles ([PP90]), ... Pour la plupart, la classe des programmes améliorés n'est pas connue ([Fuj87] ...), et l'amélioration peut être très dépendante de l'implémentation du langage (introduction de la disjonction dans MIXTUS). Par exemple, aucun critère général d'arrêt du dépliage satisfaisant n'est donné : trop de dépliages risquent de faire exploser le nombre de règles du programme, tandis que des pliages inutiles augmentent le nombre de pas de calcul nécessaires. Seize ans après leur formulation, les problèmes soulevés par [BD77] sont toujours d'actualité.

Nous allons présenter maintenant notre approche de l'évaluation partielle, qui se base sur la transformation de programmes Prolog complet et l'utilisation de l'interprétation abstraite pour guider les opérations de transformations.

Chapitre 3

Vers une évaluation partielle guidée par interprétation abstraite

Les opérations classiques de transformations (pliage, dépliage, ...) conservent la sémantique au sens de l'ensemble des succès des programmes logiques. Elles ne peuvent donc pas être appliquées directement aux programmes Prolog comportant des prédicats prédéfinis, et surtout avec des structures de contrôle comme le !, car elles ne préservent pas le comportement opérationnel (avec un interpréteur standard) du programme original. D'autre part, les systèmes déjà existant manipulant Prolog complet ont des difficultés à appliquer certaines transformations par manque d'informations sur le programme.

Les transformations que nous proposons préservent donc l'équivalence opérationnelle forte (décrite dans le chapitre 1), et prennent en compte Prolog complet. Certains travaux ont déjà étudié les transformations préservant une équivalence opérationnelle similaire ([PP91]), en considérant Prolog complet ([Sah91, PDL91]), sans information sur les programmes. D'autres systèmes proposent des transformations pour les programmes logiques récursifs ([Azi87]), ou ne transformant que certains prédicats récursifs pour obtenir des résultats plus puissants mais moins généraux ([BH87]). Dans ces deux cas, ils utilisent la connaissance de propriétés comme l'associativité. [BR89] considère les programmes Prolog avec cut et utilise des informations de mode sur les prédicats. Enfin, l'algorithme de spécialisation de programmes logiques de [GB91] est réellement guidé par une interprétation abstraite, mais n'utilise pas les mêmes informations que nous et ne considère que Prolog pur.

Dans ce chapitre, nous utilisons une interprétation abstraite basée sur la résolution OLDT, proposée par [Lec94], qui donne à notre système l'ensemble des formes d'appels et de solutions qui apparaissent durant la résolution, ainsi que des informations concernant leur terminaison et leur comportement déterministe. Contrairement à [BB92], nous ne guidons pas nos transformations

par une structure "externe", mais nous "incorporons" les informations fournies par l'interprétation abstraite à l'intérieur du source du programme, manipulant ainsi des programmes "typés". Ces informations sont ensuite effacées du programme final. Notre système ne propose pas de stratégie pour transformer les programmes, mais nous essayons de définir les propriétés minimales à satisfaire par les renseignements donnés par les analyses statiques (ou l'utilisateur), afin de les incorporer à nos programmes. Ces conditions étant très proches des contraintes définies pour le langage de programmation logique avec contraintes *CLP(X)* [JL86, JL87], nous avons étendu nos transformations à *CLP*. Nous espérons ainsi fournir un cadre générique de transformations de programmes logiques avec contraintes guidées par interprétation abstraite.

Les prédicats prédéfinis posent deux types de problèmes: par rapport à l'interprétation abstraite et par rapport aux transformations de programmes. Lorsqu'une analyse statique donne des renseignements sur le comportement opérationnel d'un programme pour un but donné, on s'attend à ce que ces renseignements soient vrais pour toute instance de ce but. Cela est toujours vrai dans le cadre de la programmation logique pure, mais devient faux dès qu'on intègre des prédicats prédéfinis: `var(X)` donne un succès, `var(a)` un échec. D'autre part, les transformations de programmes définies au chapitre 1 ont des conditions d'application encore plus restrictives avec de tels prédicats. Il faut donc éliminer ces prédicats avant la phase d'analyse statique, afin d'obtenir des informations correctes pour le programme complet. Intégrer les informations fournies par l'interprétation abstraite dans le programme Prolog complet initial nous permet:

- d'une part, de compenser la prise en compte des prédicats à effets de bord: les prédicats prédéfinis restreignent les possibilités d'applications des transformations, mais des informations supplémentaires sur le comportement particulier (déterminisme, ...) de certains littéraux peuvent amoindrir ces restrictions;

- d'autre part, de mieux manipuler les prédicats prédéfinis par la connaissance de leurs contextes d'appel.

Nous présentons ensuite les conditions sur l'interprétation abstraite nécessaires à notre système, et le formalisme choisi. Puis nous étendons la définition de nos transformations à ces programmes *spécifiés*, en prenant en compte notre classification des prédicats prédéfinis.

3.1 Schéma général

Notre but est donc de proposer un système de transformations de programmes Prolog complet en intégrant des informations (de nature non définie) sur le programme. Le dessin 3.1 montre les différentes étapes du système.

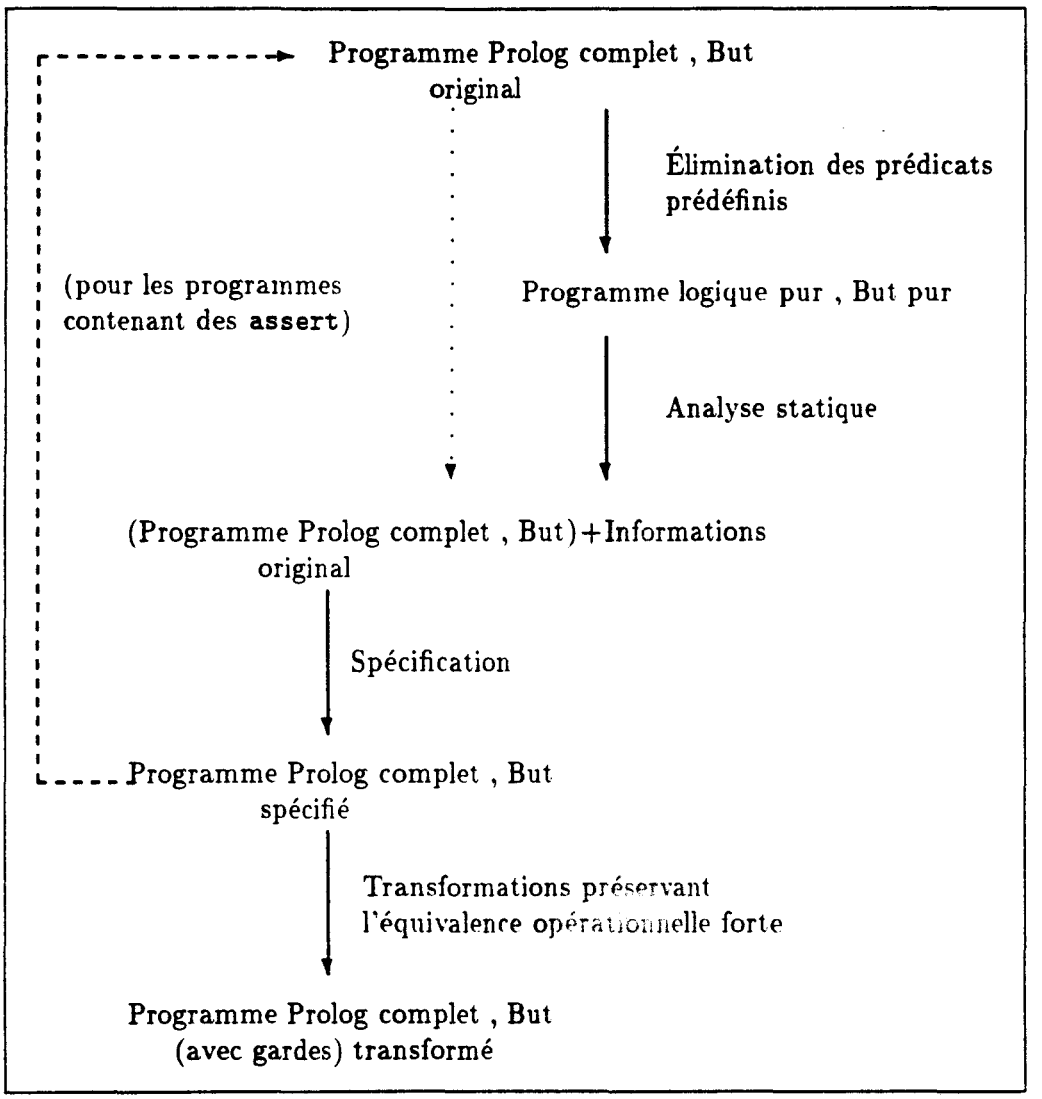


FIG. 3.1 - : Schéma général

Le plan de ce chapitre suit à peu près le programme du début à la fin :

- l'étape d'analyse statique n'est pas de notre ressort, nous présenterons au moment de la spécification les conditions que ses résultats doivent respecter pour pouvoir être incorporés à notre système;

- la flèche en tirets indique une opération qui n'est effectuée que dans certains cas : le programme contient des **assert** et l'analyse statique permet de connaître la forme des arguments du **assert**. On appelle *Specif* l'opérateur qui boucle sur le schéma (combinaison d'élimination des prédicats prédéfinis, de spécification et de traitement des **assert**). Un exemple sera développé avant de présenter les transformations de programmes;

- après les transformations de programmes, on donnera quelques algorithmes qui permettent de spécialiser les clauses en fonction de leurs schémas d'appel.

3.2 Les prédicats prédéfinis

Notre but est de proposer des transformations de programmes qui soient respectueuses du comportement opérationnel standard de Prolog. Pour obtenir une telle équivalence, nous devons définir une sémantique opérationnelle pour les effets de bord ([AMP92] étudie également une nouvelle sémantique pour les programmes Prolog comportant des prédicats prédéfinis). Nous considérons que les lectures et les écritures se font sur un ruban infini d'un côté, et qu'il y a un ruban pour chaque source (entrée/sortie standards ou sur fichiers). La tête de lecture est accessible en exclusion mutuelle pour l'utilisateur et le programme, et elle se déplace après chaque lecture ou écriture, toujours vers le côté infini. Deux programmes équivalents devront produire les mêmes rubans. On dira qu'un atome est à effet de bord s'il modifie un des rubans d'entrée-sortie, ou si son exécution modifie le mécanisme de résolution SLD. Ainsi, nous nous plaçons dans le cadre des clauses définies, car le méta-prédicat `not/1` de négation par l'échec, tel qu'il est défini dans Prolog, peut être simulé par un ! :

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

Dans la première partie, nous proposons une classification des prédicats prédéfinis en fonction des propriétés qui nous intéressent, tandis que dans la deuxième partie, nous nous intéresserons plus particulièrement aux prédicats prédéfinis qui posent des problèmes pour la correction des informations fournies par l'interprétation abstraite.

3.2.1 Classification des prédicats à effets de bord

Propriétés intéressantes

Les principales transformations que nous voulons effectuer sont celles présentées dans la section 1.2, c'est-à-dire le pliage, le dépliage déterministe, et le dépliage en tête, qui est la transformation la plus restrictive: les atomes à gauche de l'atome déplié doivent être déterministes et sans effets de bord, et la substitution qui résulte du dépliage est insérée juste avant cet atome. On voit donc que les propriétés de déterminisme et d'effets de bord sont primordiales pour appliquer les opérations de base. Nous donnons ici la liste des propriétés intéressantes pour nos transformations. Soit A un atome. Il peut être:

- **instanciable** [Pre92a]: si $(A, \texttt{X=t}) \equiv (\texttt{X=t}, A)$ $\forall$ `X` variable, `t` terme, et où $\equiv$ dénote l'équivalence opérationnelle.
- **terminant**: si sa dérivation ne contient pas de branche infinie. Par exemple, tous les prédicats prédéfinis (sauf les méta-prédicats) terminent;
- **sans effets de bord**;
- **effaçable par échec** [Pre92a]: si $(A, \texttt{fail}) \equiv (\texttt{fail}, A) \equiv \texttt{fail}$. Le ! et tous les prédicats à effets de bord ne sont pas effaçables par échec. C'est la conjugaison des deux propriétés précédentes.

  Remarque: un atome effaçable par échec n'est pas forcément déterministe (ex: le prédicat `clause/2,3` qui teste (`clause/2`) si une clause de tête le premier argument et de corps le second existe dans le programme courant, et renvoie (`clause/3`) le numéro de référence unique assigné à chaque clause. Le prédicat défini par la clause argument doit être dynamique).
- **déterministe**: si l'arbre-SLD de racine A est constitué d'une seule branche.
- **prédicat-test**: On appelle prédicat-test un prédicat `p` tel que, pour tout terme `t`:
  - `p(t)` termine, est déterministe et sans effet de bord;
  - la substitution-réponse de `:- p(t)` est vide.

  (Deux prédicats-tests peuvent permuter entre eux.)

Par construction du graphe de dépendances des prédicats, on peut ensuite classer les prédicats statiques définis par l'utilisateur. Soit p un prédicat défini par l'utilisateur. Il faut que les prédicats dont dépend p aient tous la même propriété pour que p en hérite. Ceci est vrai pour toutes les propriétés énoncées

ci-dessus, sauf pour le déterminisme, qui dépend également des clauses définissant p (soit toutes ses clauses sont en exclusion mutuelle, soit il n'a qu'une clause).

La *prédiction* des propriétés des prédicats dynamiques (et donc des prédicats qui en dépendent) est plus difficile. Elle peut provenir soit d'une analyse statique (interprétation abstraite par exemple) fine, soit par le graphe de dépendances du programme généralisé obtenu dans la section 3.2.2 (les propriétés des prédicats dynamiques sont alors déduites de la même façon que pour les prédicats statiques). Le déterminisme ne peut pas être déduit par cette seconde méthode.

Classement des prédicats prédéfinis

Nous donnons ici un classement de quelques prédicats prédéfinis choisis (arbitrairement) parmi les plus courants. Nous nous basons sur le langage Sicstus Prolog, sauf pour les cuts ancestraux qui sont définis dans Sépia Prolog. Le `mark/1` et le `!/1` sont les primitives utilisées dans [Pre92b] pour pouvoir déplier le `cut` classique. `mark(v)` donne un succès à l'appel, lie `v` à une valeur unique, et échoue au moment du backtracking. `!(v)` donne un succès à l'appel et enlève tous les points de choix jusqu'au littéral `mark(v)`.

	1 Inst.	2 Term.	3 Sans edb	4 Det	5 σ vide	2+3 Eff. échec	2+3 + 4+5 Pred.-test
<,>,≤,≥ ...		×	×	×	×	×	×
`var`		×	×	×	×	×	×
`nonvar`		×	×	×	×	×	×
`atomic,float`		×	×	×	×	×	×
`==` (syntax.)		×	×	×	×	×	×
`arg`		×	×	×		×	
`functor`		×	×	×		×	
`clause/2,3`		×	×			×	
`dif`	×	×	×	×	×	×	×
`copyterm`		×	×	×		×	
`assert`		×		×	×		
`read`		×		×			
`write`		×		×	×		
`!/0`		×		×	×		
`retract`		×					
`=` (unif.)		×	×	×		×	
`mark/1`	×	×	×	×		×	
`!/1`		×		×	×		
`is` (affectation)		×	×	×		×	

Méta-prédicats

	1 Inst.	2 Term.	3 Sans edb	4 Det	5 σ vide	2+3 Eff. échec	2+3 + 4+5 Pred.-test
`freeze`	×	?	?	?	×	?	?
`setof,bagof`	?	?	?	?		?	?
`findall`	?	?	?	×		?	?
`call`	?	?	?	?	?	?	?
`once`	?	?	?	×	?	?	?

Les '?' signifient que la propriété dépend de l'argument du méta-prédicat.

Ce tableau permet d'appliquer les transformations correctement pendant la phase d'évaluation partielle, mais pourrait être utilisé comme une pré-table des solutions par l'interprétation abstraite (c'est le système employé par [BB92]).

3.2.2 Élimination des prédicats à effets de bord par généralisation de programme

Les prédicats à effets de bord ne sont pas monotones

Si on considère un programme P écrit en Prolog complet, et G un but pour P, alors on s'aperçoit que les propriétés suivantes :

- Si la résolution de G termine, alors $\forall\sigma$ substitution, $\sigma(G)$ termine;
- $\forall\sigma$ substitution, $Solutions(\sigma(G)) \subseteq \sigma(Solutions(G))$

ne sont pas vérifiées.

Exemple 21

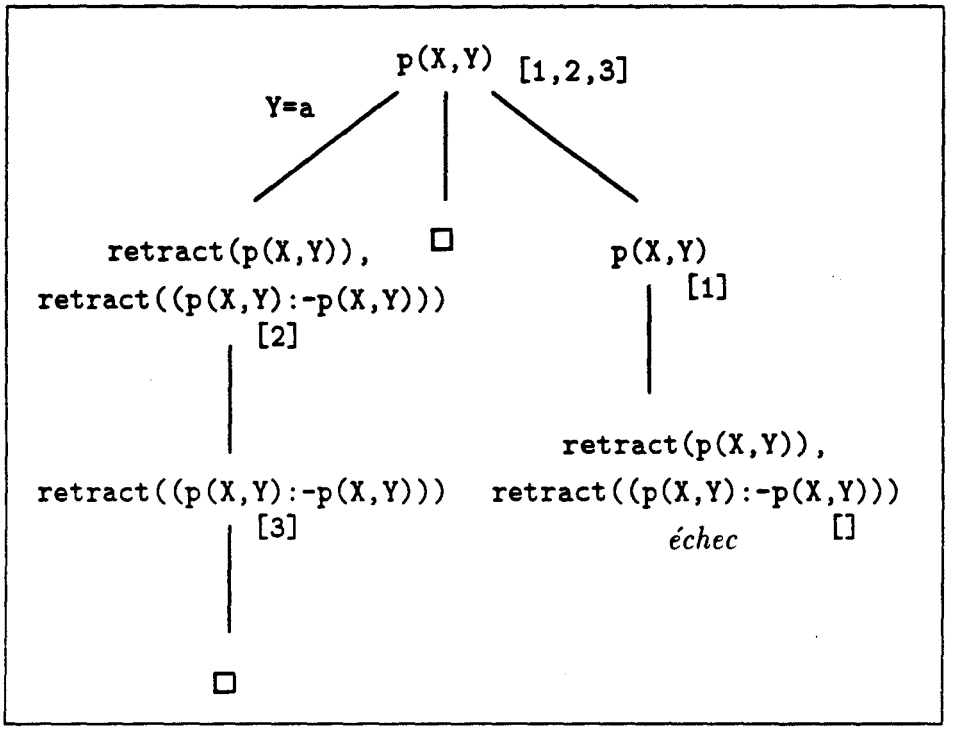
```
p(X,b) :- !.
p(0,a).
p(s(X),a) :- p(X,a).
```

Pour le but `p(X,Y)` : 1 solution.
Pour le but `p(X,a)` : une infinité de solutions.

Exemple 22

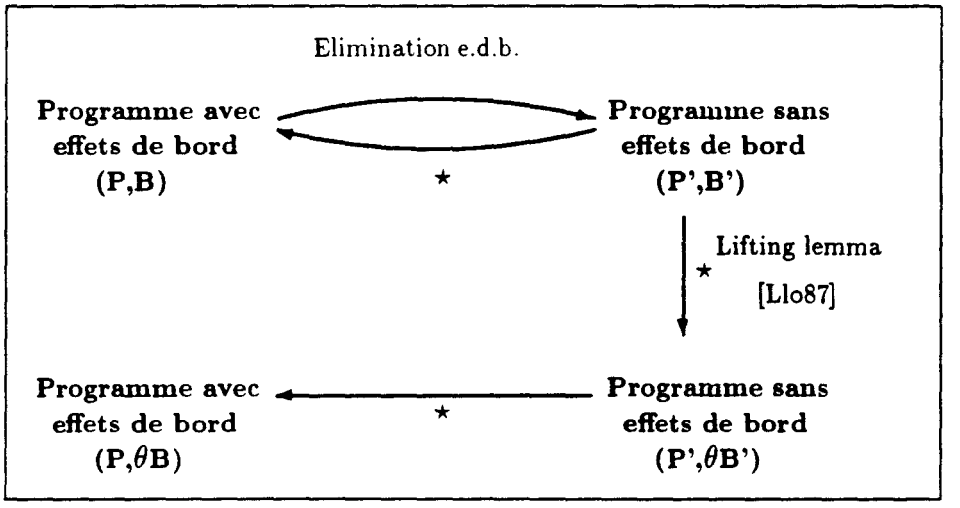
```
p(X,a) :- retract(p(X,Y)),retract(p(X,Y):-p(X,Y)).
p(X,Y).
p(X,Y) :- p(X,Y).
```

Pour le but `p(X,Y)` : 2 solutions.
Pour le but `p(X,b)` : une infinité de solutions.



Les listes de numéros représentent les listes de points de choix (par numéro de règles) de chaque nœud.

Dans cette section, nous proposons une méthode pour éliminer les prédicats à effets de bord d'un programme, afin que les propriétés calculées par interprétation abstraite sur ce programme pour un but donné soient vraies pour toute instance de ce but.



(Programme,But) $\xrightarrow{*}$ (Programme',But') : les propriétés calculées pour le programme de gauche sont correctes pour le programme de droite.

Propriétés calculées par l'interprétation abstraite

Pour un programme Prolog pur et un but donnés, les informations fournies par interprétation abstraite que nous pouvons traiter portent sur la forme des littéraux appelés pendant l'exécution (schéma d'appel, ou encore pré-schéma), la forme de leurs solutions (schéma de solutions ou post-schéma), leur terminaison et leur déterminisme, telles que les schémas d'appels et de solutions soient exprimés sous la forme de systèmes d'équations et de diséquations, qui peuvent être vus comme des contraintes sur les termes des atomes appelés.

Principe général

Cadre choisi Nous avons choisi le cadre de Sicstus Prolog, qui est compatible avec un grand nombre d'autres Prolog.

Les prédicats qui ne sont pas monotones sont les structures de contrôle, qui modifient le mécanisme de résolution, les prédicats dynamiques, qui modifient le source de programme. Nous étudions également les deux autres prédicats à effets de bord, i.e. les prédicats d'entrée et de sortie : le cas de tous les prédicats prédéfinis est équivalent.

Description de la méthode d'élimination des prédicats à effets de bord par généralisation de programmes La plupart des prédicats que nous traitons ont une action restrictive sur le comportement du programme : le ! coupe des branches, le `read` en instanciant ses arguments peut faire échouer des unifications. Notre méthode consiste donc à trouver une version plus générale du programme original (cf. définitions 23 et 24, proposition 14), qui soit sans effets de bord. Nous modifions les programmes afin d'obtenir un programme dont l'arbre SLD *contient* l'arbre SLD du programme original : l'arbre SLD original doit être un calque de l'arbre SLD du programme logique pur, auquel on a retiré certaines branches. Cela nous permet de conserver les propriétés de déterminisme (si le plus général l'est, le plus précis l'est), d'ensemble de substitutions-réponses et d'arrêt.

La généralisation du programme ne peut pas être exactement obtenue pour un programme contenant des `assert` : on obtient un programme plus général pour les propriétés d'arrêt et de substitutions-réponses, mais pas pour la propriété de déterminisme.

a) read/write

Ces prédicats sont tous les deux déterministes, et leur action porte sur le ruban d'entrée-sortie et sur la tête de lecture. L'effet du `write` s'arrête là, il n'influe pas sur les propriétés calculées par l'interprétation abstraite. Par-contre, le `read` unifie son argument avec le terme lu sur le ruban de lecture. Il peut donc soit échouer, soit instancier son paramètre. C'est une action "restrictive" sur le programme, qui n'influe pas sur la terminaison ou le déterminisme d'un

programme, mais qui détermine de manière plus précise ses schémas d'appels ou ses schémas de solutions. Si on élimine les **read** d'un programme, les pré-schémas et les post-schémas calculés pour ce programme incluront de manière large les pré-schémas et les solutions du programme au moment de l'exécution. On peut éliminer pour les mêmes raisons les autres prédicats prédéfinis qui ne sont pas particulièrement étudiés ici.

b) Coupe-Choix

Le **cut** est déterministe, et son action revient à couper des branches dans l'arbre de résolution SLD. Il influe donc sur toutes les propriétés calculées par l'interprétation abstraite, dans le sens où, si un programme P avec coupe-choix termine pour un but B, le même programme P sans coupe-choix (par une manipulation syntaxique) ne termine pas forcément pour B, puisque un **cut** peut empêcher de "tomber" dans une branche infinie. Par-contre, si P sans coupe-choix termine pour B, alors P avec **cut** termine pour B. Il en est de même pour toutes les autres propriétés calculées par l'interprétation abstraite.

c) retract

Le prédicat **retract** n'est pas un prédicat déterministe, mais son action, en ce qui concerne la conservation des propriétés de l'interprétation abstraite, est semblable à celle du **cut**, car elle ne fait que supprimer des branches dans l'arbre. Pour simuler le déterminisme du **retract**, il faut donc ajouter au programme des faits de la forme **retract(règle)** soit pour chaque règle du programme, soit pour un sous-ensemble des règles du programme, si on peut faire un caractérisation assez fine des règles dynamiques.

d) assert

Le prédicat **assert** est déterministe, mais il ajoute des branches de dérivation à l'arbre de résolution SLD en cours d'exécution. Nous montrons dans la section 25 qu'en fait il suffit d'effectuer une fois chaque **assert** dans le programme original avant de le donner à l'interprétation abstraite, et d'annuler l'action des **assert**, pour que les propriétés calculées par l'interprétation abstraite soient valides (seul le déterminisme est perdu). Nous ne pouvons traiter que les **assert** dont au moins le symbole du prédicat de l'argument est connu (sinon, la transformation est triviale et l'interprétation abstraite ne pourra plus calculer d'informations intéressantes).

Exemple 23

Soit le programme composé d'une seule clause :

```
p(s(X)) :- assert(p(0)), p(X).
```

Si on applique la transformation sur le **assert** pour fournir le programme à l'interprétation abstraite, on obtient :

```
p(0).
p(s(X)) :- p(X).
```

L'interprétation abstraite va alors déduire que pour tout but de la forme

`:-p(s`^{n}`(0))`,
le programme termine et est déterministe. La propriété de terminaison est vraie pour le programme original, mais pas la propriété de déterminisme.

Néanmoins, si l'interprétation abstraite peut détecter les atomes (et donc certains **assert**) qui ne seront appelés qu'**une seule fois** pour un certain but, alors la propriété de déterminisme peut être calculée correctement.

e) disjonction et if-then-else

La disjonction est notée par l'opérateur ;. Soient P_1 et P_2 les deux sous-programmes suivants:

P_1 : `A :- Avant, (B1;B2), Apres.`
et

P_2 : `A :- Avant, B.`
`B :- B1, Apres.`
`B :- B2, Apres.`

où

- `Avant`, `Apres`, `B1` et `B2` sont des conjonctions d'atomes;
- `B` est un atome avec un nouveau symbole de prédicat, et dont les arguments sont les variables qui apparaissent à la fois dans {`A,Avant`} et dans {`B1,B2,Apres`}.

On a $P_1 \equiv P_2$ si ni B_1 ni B_2 ne contiennent de !, puisqu'on peut assimiler la différence entre les deux sous-programmes à une séquence de définition-pliage-dépliage. Il faut donc respecter les conditions d'application du dépliage.

Exemple 24

```
p(X) :- ok, (X is 1,!;X is 2).
p(0).
ok.
```

Il y a une solution {`X=1`} pour le but `:-p(X)`. Si on transforme le programme:

```
p(X) :- ok, pnew(X).
p(0).
ok.
pnew(X) :- X is 1, !.
pnew(X) :- X is 2.
```

On obtient deux solutions, {`X=1`} et {`X=0`}, pour le but `:- p(X)`. Mais si on ignore le !, on obtient alors les trois solutions {`X=1`}, {`X=2`} et {`X=0`}.

La disjonction ne peut donc pas être toujours facilement remplacée (dans un programme réel) par deux clauses. Mais cette transformation peut être effectuée en vue de la phase d'interprétation abstraite, puisqu'on élimine les !.

La sémantique de l'opérateur '->' est if-then [Swe93]:

```
   :-A->B
≡  if A then B
≡  :-A,!,B.
```

Combiné avec la disjonction, on obtient l'opérateur 'if-then-else' :

```
   :-A->B;C
≡  if A then B else C
≡  :- A,!,B.
   :- C.
```

Les ! n'étant pas acceptés dans la partie condition (`A`), cette transformation est toujours correcte.

Voici un résumé des propriétés calculées par l'interprétation abstraite sur des programmes "épurés" et qui sont conservées pour les programmes originaux :

Atomes dépendant de → Propriétés ↓	`read/write, !, retract` Disjonction, If-then-else	`assert`
Terminaison	oui	oui
Déterminisme	oui	
Schémas d'appels	oui	oui
Schémas de solutions	oui	oui

La méthode consiste donc à transformer les disjonctions et les alternatives, puis à éliminer les `read`, `write`, `!`, et `retract`; et enfin à effectuer les `assert` (avant d'éliminer les appels à `assert`). Dans la suite, nous nous attachons à prouver la correction de notre méthode, sans tenir compte de la disjonction et du if-then-else. La preuve consiste à comparer les arbres de résolution SLD d'un programme avant et après transformation. Pour cela, nous définissons les notions d'arbre de résolution et d'arbres partiels associés, et nous montrons qu'ils ont exactement le même rapport qu'entre un programme à effets de bord et le même programme "épuré". La preuve pour le `assert` est différente et séparée.

Notion d'Arbre Maximal et d'Arbres Partiels associés

Notre but est donc de passer d'un programme avec effets de bord à un programme écrit en Prolog pur. Cette transformation s'effectuant en deux étapes,

nous présentons d'abord les différents domaines de définition des programmes Prolog que nous considérons :

- Π_{pur} = {programmes Prolog pur}

- Π_{ass} = {programmes Prolog pur + prédicats **assert**}

- Π_{edb} = {programmes Prolog contenant les prédicats à effets de bord **cut**, **read/write**,**assert/retract**}

On induit d'abord la définition de l'ensemble des succès d'un programme par rapport à un but, qu'on notera *SS(P,A)* pour P un programme et un A un but, de la définition de l'ensemble des succès d'un programme (*SS(P)*). *SS(P,A)* est donc la restriction de *SS(P)* à tous les atomes qui sont instances de A. Pour caractériser le comportement d'un programme, nous définissons maintenant la notion d'arbre de dérivation maximal, puis des arbres de dérivation partielles qui peuvent lui être associés.

Définition 23 *L'arbre de résolution d'un programme P, P∈ Π_{edb}, et d'un but B, noté Arbre(P,B), est l'arbre de dérivation SLD avec la règle de sélection standard, tel que :*

- *ses nœuds sont étiquetés par une liste d'atomes, auxquels on associe la liste des règles candidates à l'unification avec le premier atome de la liste, dans l'ordre où ces règles apparaissent dans le programme (liste vide pour les prédicats prédéfinis qui font appel à la librairie Sicstus);*

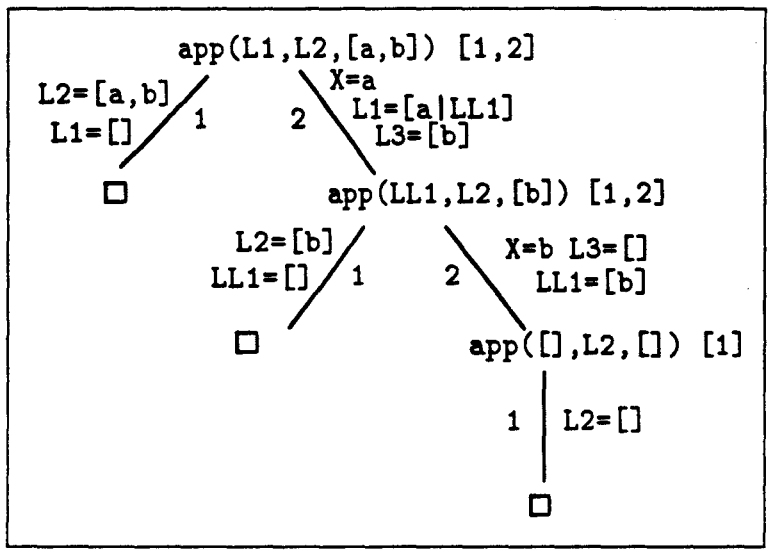
- *ses arcs sont étiquetés par le numéro de la règle utilisée et par le système d'équations {premier atome = tête(règle)}*

- *ses noeuds solutions sont ceux étiquetés par la clause vide, notée* □.

Exemple 25

1. `app([],L,L).`
2. `app([X|L1],L2,[X|L3]) :- app(L1,L2,L3).`

```
:- app(L1,L2,[a,b]).
```



Définition 24 *Un arbre partiel de résolution d'un programme P, P∈ Π_{edb}, et d'un but B, noté Arbre-partiel(P,B), est un sous-arbre haut de Arbre(P,B), tel que :*

- *ils ont même racine;*
- *tout nœud-solution de Arbre-partiel(P,B) est un nœud-solution de Arbre-(P,B));*
- *il existe une application injective monotone pour l'ordre de Dewey de l'arbre partiel vers un sous-ensemble de l'arbre :*

  θ : *Nœud(Arbre-partiel(P,B))* $\longrightarrow$ {*Nœud(Arbre(P,B))*}*

  $\forall$ $n \in$ {*Nœud(Arbre-Partiel(P,B))*}*, *l'étiquette de n est identique ou plus générale que celle de* $\theta(n)$, *et la liste associée à n est identique ou plus grande que celle associée à* $\theta(n)$.

Si on superpose un arbre partiel sur l'arbre de dérivation auquel il se rapporte, l'arbre partiel doit être filtré par l'arbre de dérivation.

Soient $\theta_1, \theta_2, \ldots \theta_n$ les substitutions-réponses associées aux branches succès de l'arbre (partiel) de résolution du programme P et du but B. On notera *Solutions(Arbre(P,B))* (*Solutions(Arbre-partiel(P,B))*) l'ensemble des instances closes de $\{\theta_1 B, \theta_2 B, \ldots \theta_n B\}$. De la même façon, on notera *Schémas-d'appels(Arbre(P,B))* (ou *Schémas-d'appels(Arbre-partiel(P,B))*) l'ensemble des instances closes des atomes qui apparaissent comme étiquette d'un nœud de l'arbre (partiel) de résolution de P et de B. Pour éviter les surcharges de notations, *Solutions(P,B))* et *Schémas-d'appels(P,B))* dénoteront respectivement *Solutions(Arbre(P,B))* et *Schémas-d'appels(Arbre(P,B))*.

Proposition 14 *Soient P un programme et B un but :*

1. *Si Arbre(P,B) est fini, alors tout arbre partiel pour P et B est fini;*
2. *Si Arbre(P,B) est déterministe, alors tout arbre partiel pour P et B est déterministe;*
3. *Solutions(Arbre-partiel(P,B)) $\subseteq$ Solutions(Arbre(P,B));*
4. *Schémas-d'appels(Arbre-partiel(P,B)) $\subseteq$ Schémas-d'appels(Arbre(P,B)).*

Preuve

Cette proposition est immédiate à partir de la définition d'un arbre partiel.

1. Chaque nœud de *Arbre-partiel(P,B)* appartient également à *Arbre(P,B)*, donc tout arbre partiel de *Arbre(P,B)* ne peut contenir une branche infinie que si *Arbre(P,B)* en contient une.
2. Si *Arbre(P,B)* a une seule solution, alors tout arbre partiel pour P et B a au plus une solution. De plus, si *Arbre(P,B)* est complètement déterministe (constitué d'une seule branche), alors tout arbre partiel pour P et B est complètement déterministe.
3. immédiat.
4. D'après la définition, les étiquettes de *Arbre-partiel(P,B)* sont plus instanciées que celles de *Arbre(P,B)*.

□

Elimination des !, read/write et retract

Dans une première partie, nous nous limitons aux programmes contenant les prédicats à effets de bord !, `read`, `write`, et `retract`.

Définition 25 Transformation

$\varphi_1 : \Pi_{edb} \longrightarrow \Pi_{ass}$

avec φ_1 homomorphisme qui ajoute des faits :

1. `!` $\leftarrow$.
2. `read(_)` $\leftarrow$.
3. `write(_)` $\leftarrow$.
4. *on recopie chacune des règles du programme dans un fait :*
   `retract(règle)` $\leftarrow$.

Soient $P \in \Pi_{edb}$ un programme et B un but. Pendant la résolution de $\varphi_1(P)$ par rapport à B, les effets de bord sont inhibés : le !, le **read** et le **write** s'effacent lors de la résolution, sans la modifier. Le **retract** reste indéterministe, mais il ne modifie pas le code du programme.

Définition 26 Comportement du Coupe-choix

Le ! efface la fin de la liste des points de choix sur tous ses nœuds ancêtres qui se situent sur le chemin à partir du nœud qui le génère jusqu'au nœud où il est le premier atome de la conjonction.

Définition 27 Comportement du retract

Un **retract** *supprime les règles qui s'unifient avec son argument, et efface aussi toutes les occurrences de ces règles dans les listes de points de choix des nœuds de l'arbre de dérivation qui sont interprétés après lui par la règle de parcours standard (le* **retract** *ne remet pas en cause les listes de points de choix qui ont déjà été construites).*

Exemple 26

```
fait(b).
fait(c).

p(X) :- retract(fait(X)).
```

Si on pose la question `:- p(X)`, la première fois, les solutions sont {`X=b`; `X=c`}, puis, à chaque fois qu'on repose cette même question, il n'y a pas de solution.

Exemple 27

```
fait(b).
fait(c).

p(X,Y) :- retract(fait(X)), q(Y).
q(X) :- retract(fait(X)).
```

`:- p(X,Y)` est déterministe, est la solution est {`X=b,Y=c`}.

Proposition 15 *L'arbre de résolution d'un programme prolog P, pouvant comporter les prédicats à effets de bord* !, **read**/**write** *et* **retract**, *et d'un but B est un arbre partiel de l'arbre de résolution Arbre($\varphi_1(P)$,B).*

Preuve

a) Si on ajoute des **write**(_) à $\varphi_1(P)$: le prédicat **write** est un prédicat déterministe, dont l'effet de bord s'effectue à l'écran, et n'affecte pas le déroulement du programme;

b) Si on ajoute des **read**(_) à $\varphi_1(P)$: le prédicat **read** est un prédicat déterministe, qui instancie le terme qu'il a en paramètre.

- Si l'unification échoue, la branche se termine en échec, on a enlevé un sous-arbre à *Arbre(P,B)* : on a toujours

  $$Solutions(Arbre(P,B)) \subseteq Solutions(Arbre(\varphi_1(P),B))$$

- Si l'unification réussit, le sous-arbre fils sera plus instancié: soit θ la substitution-réponse du **read**. La résolution va se faire comme pour *Arbre(P,B)*, modulo θ puisque tous les nœuds seront plus instanciés. Tant que θ ne sera pas inconsistante avec le système d'équations du sous-arbre de *Arbre(P,B)*, *Arbre(φ_1(P),B)* sera identique; la où θ sera inconsistente, la branche s'arrêtera en échec dans *Arbre(P,B)*.

c) Si on ajoute des ! à φ_1(P): le ! efface des éléments dans la liste des points de choix, ce qui revient à couper des sous-arbres dont la résolution n'influe pas sur les autres sous-arbres frères. On a donc bien: *Arbre(P,B)*$\in$ {*Arbre-partiel(φ_1(P),B)*} (*Arbre(P,B)* appartient bien à l'ensemble des arbres partiels qu'on peut construire pour P et B).

d) Un **retract** coupe des sous-arbres par rapport à *Arbre(P,B)*, mais ne remet pas en cause les substitutions partielles qui ont été calculées avant son exécution (le problème est donc semblable à celui du !). D'autre part, l'indéterminisme du **retract** est simulé par la recopie et l'encapsulation de toutes les règles de P dans des faits de la forme `retract(p(X) :- q(Y))`. Donc toutes les branches de l'arbre de dérivation de (P, B) sont également dans *Arbre(P,B)*.

En supprimant un **retract**, les substitutions calculées par $\varphi_1(P,B)$ sont plus générales que celles de (P, B).

□

Lemme 2 (Lifting lemma [Llo87]) *Soient P un programme défini, G un but défini et θ une substitution. S'il existe une réfutation pour $P \cup \{\theta G\}$, alors il en existe une pour $P \cup \{G\}$ de la même longueur, telle que, si $\theta_1, \theta_2, \ldots \theta_n$ sont les mgus de la réfutation SLD de $P \cup \{\theta G\}$, et $\theta'_1, \theta'_2, \ldots \theta'_n$ les mgus de la réfutation SLD de $P \cup \{\theta G\}$, alors il existe une substitution γ telle que $\theta_1\theta_2 \ldots \theta_n = \gamma\theta'_1\theta'_2 \ldots \theta'_n$.*

On dira qu'une propriété est monotone si une propriété qui est vraie pour $(\varphi_1(P), B)$ est vraie pour $(P, \theta B)$.

Proposition 16 *Soient P un programme Prolog pouvant contenir les prédicats à effets de bord cut,* **read**, **write** *et* **retract**, *et B un but.*

Les propriétés de terminaison, déterminisme, schémas d'appel des atomes, et formes des solutions entre $(\varphi_1(P), B)$ et $(P, \theta B)$ sont monotones.

Preuve

- Par les propositions 14 et 15 les propriétés de terminaison, déterminisme, Schéma d'Appel et forme des solutions sont monotones entre $(\varphi_1(P), B)$ et (P, B).

- On déduit la propriété suivante du Lifting lemma ($\varphi_1(P)$ est un programme Prolog pur): *Solutions(φ_1(P),θB)* $\subseteq$ *Solutions(φ_1(P),B)*

Il en découle que les propriétés de terminaison, déterminisme, schéma d'appel des littéraux et forme des solutions sont monotones pour $(\varphi_1(P), B)$. (Toute propriété vraie pour $(\varphi_1(P), B)$, est vraie pour $(\varphi_1(P), \theta B)$.
$\Rightarrow$ Toute propriété vraie pour $(\varphi_1(P), B)$ est vraie pour $(P, \theta B)$.

□

Elimination des assert

Nous considérons maintenant les programmes Prolog ne contenant plus comme prédicat prédéfini que des **assert**.

Définition 28 Transformation
$\varphi_2 : \Pi_{ass} \times But \longrightarrow \Pi_{pur} \times But$
avec φ_2 homomorphisme qui ajoute des règles :

1. `assert(_).`
2. *pour chaque atome* `assert(X)`, *on ajoute la règle* `X.` *au programme (si le But contient un* **assert**, *la règle est ajoutée au programme).*

Soient $P \in \Pi_{ass}$ et B un but. *Arbre(P,B)* n'est pas un arbre partiel de *Arbre(φ_2(P,B))*, comme le montre l'exemple 28. Néanmoins, nous montrons que la propriété de terminaison, la forme des solutions et le schéma d'appel des atomes sont conservés entre $\varphi_2(\varphi_1(P), B)$ et (P, B). Il paraît évident, que si le programme contient des **assert** dont l'argument est une variable libre, ajouter la clause `X :- Y` au programme est la condition de correction des propriétés que nous voulons conserver, mais empêchera l'interprétation abstraite de calculer des informations intéressantes (par exemple sur la terminaison). L'opérateur *Specif* (cf. section 3.4) pallie à ce problème dans certains cas.

Remarque Pour les règles (ou but) dont le corps contient un atome de la forme `assert(a:-assert(b))`, le programme épuré est $\varphi_2^n(P, B)$, n étant la profondeur maximal d'un **assert** dans le programme (2 dans l'exemple). Pour simplifier les notations, nous écrirons toujours $\varphi_2(P, B)$, même s'il s'agit de $\varphi_2^n(P, B)$.

Définition 29 Comportement du assert
Un **assert** *ajoute la règle qu'il a en paramètre au programme à la fin de la définition du prédicat de la règle (si ce prédicat existait déjà dans le programme, à la fin du programme sinon). Un* **assert** *ne modifie pas les listes de points de choix, donc ne remet pas en cause ce qui a été calculé avant son exécution.*

Exemple 28
```
fait(b).
```

```
fait(c).

p(X) :- fait(X), assert(fait(a)).
```

Si on pose la question `:- p(X)`, le programme ne boucle pas. La première fois, les solutions sont {`X=b;X=c`}, puis, (lors d'une deuxième session) les solutions sont {`X=b;X=c;X=a;X=a`}, puis {`X=b;X=c;X=a;X=a;X=a;X=a`}, ...

Lemme 3 *Soient P un programme Prolog pur et B un but. Soit P' un programme contenant toutes les règles de P dans le même ordre, mais dont certaines des règles ont été recopiées un nombre fini de fois et instanciées. (P, B) termine si et seulement si (P', B) termine aussi.*

Preuve Soient P un programme Prolog pur et B un but.

Soit P' un programme contenant toutes les règles de P dans le même ordre, mais dont certaines des règles ont été recopiées plusieurs fois.

Considérons qu'une seule clause, R, a été répétée, et que B est le but le plus général qui puisse s'unifier avec la tête de R. Soient n le nombre de règles R et p la profondeur de l'arbre de résolution de (P, B). L'arbre de résolution de (P', B) a au plus $n \times (p!)$ nœuds de plus que l'arbre de dérivation de (P, B), soit un nombre fini.

D'après le lifting lemma, le passage à des règles plus instanciées conserve la terminaison (nous considérons des programmes Prolog purs). Donc le fait que R puisse avoir été répétée et instanciée ne modifie pas la terminaison. □

On appellera profondeur d'un **assert** son niveau d'imbrication. Par exemple, la profondeur du littéral `assert(p(X) :- assert(q(Y)))` est de 2. On appelera *règles dynamiques* les règles générées par des assert, et *règles dynamiques de profondeur n* les règles générées par des **assert** de profondeur n.

Lemme 4 *Soient P un programme Prolog, $P \in \Pi_{ass}$, et B un but. Si (P, B) génère un nombre infini de règles, alors $\varphi_2(P, B)$ ne termine pas.*

Preuve

On le démontre par récurrence sur la profondeur des **assert**.

1. P ne contient pas de **assert**: $\varphi_2(P) = P$, donc, en particulier, l'arrêt est le même pour les deux programmes.

2. On pose l'hypothèse de récurrence suivante: si la profondeur des **assert** de P ne dépasse pas n, alors la proposition 4 est vérifiée.

   Maintenant, on suppose que P contient des **assert** imbriqués jusqu'à une profondeur $n + 1$.

   D'après l'hypothèse de récurrence, on sait que jusqu'à la profondeur n, les clauses générées ne posent pas de problème. Donc, si la proposition

n'est pas vraie pour une profondeur $n+1$, c'est à cause de la clause générée par le $n+1^{ième}$ **assert**.

Il suffit d'un nombre fini de pas de résolution pour générer un nombre fini de nouvelles règles (chacune de ces règles étant une instance d'argument d'un **assert**, à quelque profondeur que ce soit), puisque les **assert** sont déterministes.

Le comportement du **assert** interdit les "backtracks infinis" (i.e. la construction d'un arbre de résolution infini en largeur): la liste des points de choix qui est calculée avant l'exécution d'un **assert** ne peut être remis en cause, un programme comme celui de l'exemple 28 ne boucle pas. Donc, pour générer un nombre infini de règles, il faut exécuter un nombre infini de fois au moins une règle de Π contenant un **assert**, c'est-à-dire qu'il faut qu'au moins une règle de P contenant un **assert** apparaisse dans une branche infinie. Deux cas se présentent :

- la branche infinie n'est constituée que d'appels à des anciennes règles (règles qui ne sont pas générées par le programme ou règles dynamiques de profondeur $\leq n$) : alors cette branche infinie existe aussi dans $\varphi_2(P,B)$ puisque soit elle ne dépend pas d'un **assert**, soit elle vérifie l'hypothèse de récurrence.
- la branche infinie contient des appels à des nouvelles règles (règles dynamiques de profondeur $n+1$) : toutes ces règles sont des instances ou des variantes de celles qui ont été ajoutées dans $\varphi_2(P,B)$ (définition de φ_2), donc la branche infinie existe aussi dans $\varphi_2(P,B)$ (lemme 3).

Dans les deux cas, s'il y a une branche infinie dans l'arbre de dérivation de (P,B), elle apparaît aussi dans l'arbre de dérivation de $\varphi_2(P,B)$.

□

Proposition 17 *Soient P un programme Prolog, $P \in \Pi_{edb}$, et B un but. Si $\varphi_2(\varphi_1(P),B)$ termine, (P,B) termine.*

Preuve Si $(\varphi_1(P),B)$ termine, $(P,B$ termine (proposition 16). Enfin $\varphi_2(\varphi_1(P),B)$ termine $\Rightarrow$ (P,B) termine, est la contraposée du lemme 4. □

Remarque : la réciproque est fausse.

Exemple 29

```
P     p(X).
      q(X) :- assert(p(X) :- p(X)).
But   :- p(X)
```

(P, But) termine, $\varphi_2(P, But)$ boucle.

Proposition 18 *Soient $P \in \Pi_{ass}$, et B un but.*

1. *Solutions(Arbre(P,B)) $\subseteq$ Solutions(Arbre(φ_2(P,B)))*
2. *Schémas-d'appels(Arbre(P,B)) $\subseteq$ Schémas-d'appels(Arbre(φ_2(P,B)))*

Preuve Soient C_{P_0} l'ensemble des clauses de P avant l'exécution de (P, B), $C_{dyn(P)}$ l'ensemble des clauses contenues dans un **assert** de P, et $C_{\varphi_2(P)}$ l'ensemble des clauses de $\varphi_2(P)$:

$$C_{\varphi_2(P)} = C_{P_0} + C_{dyn(P)}$$

C_{P_n} est l'ensemble des clauses de P à une étape n de son exécution.
$\forall R \in C_{P_n}, \exists R' \in C_{\varphi_2(P)}$ et θ une substitution, telles que $R = \theta R'$
$\Rightarrow \forall R \in C_{dyn(P)}, R \geq \bigwedge\{R' \in C_{P_n} / \exists \theta$ une substitution telle que $R = \theta R'\}$
$\Rightarrow$ les étiquettes de l'arbre de résolution de $\varphi_2(P, B)$ sont moins instanciées que les étiquettes correspondantes dans l'arbre de résolution de (P, B)
$\Rightarrow$ sur chaque branche de l'arbre de résolution de $\varphi_2(P, B)$, le système d'équations est plus général que l'anti-unifié des systèmes d'équations des sous-arbres correspondants dans l'arbre de résolution de (P, B)
$\Rightarrow$ les solutions de $\varphi_2(P, B)$ seront plus générales que les solutions de (P, B) et le schéma d'appel des littéraux pour $\varphi_2(P, B)$ est plus général que le schéma d'appel des littéraux pour (P, B). □

3.3 Programmes logiques définis spécifiés

Nous nous proposons de définir un cadre de travail générique pour effectuer une évaluation partielle guidée par interprétation abstraite sur des programmes logiques (avec contraintes). Les informations sur le comportement du programme sont vues comme des annotations sur la forme des termes dans le programme qui permettront à l'évaluateur partiel de spécialiser certaines clauses en fonction de la forme de leurs arguments.

Ces annotations n'apparaissent que dans la phase de précompilation. Ne resteront dans le programme que les informations jouant le rôle de "garde" pour les clauses spécialisées. Elles seront alors exécutées comme des contraintes. Le domaine d'informations (appelées dans la suite contraintes) que nous utilisons doit donc vérifier les conditions imposées aux domaines de contraintes dans le cadre de *CLP*.

Nous présentons dans cette section les propriétés minimales dont nous avons besoin, ainsi que les opérations que nous effectuons sur ces contraintes. Un programme Prolog sera pour nous un programme écrit en $CLP(\mathcal{X})$, avec $\mathcal{X}$ domaine de contraintes *solution compact*.

3.3.1 Une algèbre pour les atomes spécifiés

Le domaine des contraintes que nous allons manipuler est donc un couple noté $(\mathcal{A}, Info)$, où $\mathcal{A}$ est une Σ-algèbre de termes sur un S-ensemble de sortes A et une signature Σ, et *Info* un sous-ensemble des formules du premier ordre exprimables avec Σ. Σ contient le symbole = qui est interprété (dans $\mathcal{A}$) comme l'identité sur A, et la constante *fail*. Les noms des opérations et des fonctions définies par Σ sont tous distincts de ceux définis par la signature $\Sigma_{\mathcal{X}}$ associée à $\mathcal{X}$. *Info* est clos par renommage et conjonction, et contient des formules sous forme normale conjonctive solvables, et les constantes *fail* et *true*. Une contrainte atomique est construite sur Σ et $\mathcal{V}$ (ensemble infini de variables). Une $\mathcal{A}$-assignation de $\mathcal{V}$ dans A est une substitution (puisque nous travaillons sur une algèbre de termes libres). Enfin, la fonction Var : $Info$,A,... $\rightarrow \mathcal{V}$ retourne l'ensemble des variables qui apparaissent dans n'importe quel objet syntaxique.

Dans la suite, on appellera système de contraintes un élément de *Info*, et contrainte une contrainte atomique. Un programme $CLP(\mathcal{X})$ est un ensemble ordonné de règles de la forme $H \leftarrow c_{\mathcal{X}}, B_1, \ldots, B_n$ où $H, B_1, \ldots B_n$ sont des atomes et $c_{\mathcal{X}}$ est un système de contraintes défini sur $\mathcal{X}$. Les contraintes fournies par l'analyseur ou l'interprétation abstraite seront notées $c, s, \ldots$ pour un système de contraintes, et $C, S, \ldots$ pour un ensemble de systèmes de contraintes. En général, les règles sont notées avec les contraintes rassemblées en début de corps, bien qu'en $CLP(\mathcal{X})$ les contraintes puissent se trouver à n'importe quel endroit du corps de la règle. Elles sont ajoutées au système de contraintes courant lorsqu'elles se trouvent en tête de but. Le modèle d'exécution de Prolog III est un peu différent : les contraintes sont rassemblées (syntaxiquement en fin de règle, derrière l'opérateur |) et sont ajoutées au système de contraintes au moment de l'unification du but avec la tête de la règle dans laquelle elles sont définies. Cette petite différence opérationnelle sera précisée lorsque nous parlerons des transformations de programmes spécifiés (cf. 3.5).

Définition 30 *Un atome spécifié est un triplet* (c, a, s) *où:*

- *a est un atome;*
- $c, s \in Info$.

Ces systèmes de contraintes représentent une partie de ce qui est connu pendant l'exécution du programme, à l'appel de l'atome spécifié. Ces systèmes définissent donc un ensemble de termes : un atome spécifié est un triplet dans lequel les deux systèmes de contraintes ont un ensemble de solutions non vides, ou alors, le fait que la résolution de l'atome est un échec fini est exprimé par la constante *fail*. Nous appellerons souvent dans la suite schéma d'appel le premier élément d'un triplet, et schéma de solution le dernier élément. Souvent, les conjonctions d'atomes spécifiés $(c_1, a_1, s_1), (c_2, a_2, s_2)$ seront telles que

$s_1 = c_2$. Mais cette notation nous permet une définition plus simple des transformations de programmes. Si on supprime les parenthèses autour des atomes spécifiés, un programme spécifié est un programme écrit en $CLP(\mathcal{X}+Info)$.

3.3.2 Propriétés requises pour la structure *Info*

Les propriétés suivantes sont présentées par analogie avec [Smi91] et [LMM87].

Soient c un système de contraintes et a un atome. On note Sol_c l'ensemble des substitutions closes θ sur $Var(c)$ telles que $\theta c \overset{sem}{\equiv} true$. Par extension, on note $Sol_c(a)$ l'ensemble des instances closes de a qui vérifient c. Pour c un système de contraintes, si c n'est pas solvable (i.e. $c \overset{sem}{\equiv} fail$), $Sol_c = \emptyset$.

Pour c un système de contraintes, $\forall \theta \in Sol_c, dom(\theta) \subseteq Var(c)$. Une forme résolue et une forme canonique doivent être définies (calculables) sur *Info*. On définit la projection d'une substitution sur un ensemble de variables, notée $\uparrow$, comme la restriction de la substitution sur cet ensemble de variables. On peut donc généraliser à un ensemble de substitutions, et ainsi $Sol_c \uparrow V$ désigne l'ensemble des substitutions closes appartenant à Sol_c et restreintes à V. De la même façon, on définit la projection sur un ensemble de variables d'un système de contraintes c par $c' = c \uparrow V$ ssi $Var(c') \subseteq V$ et $Sol_c \uparrow V = Sol_{c'} \uparrow V$. On définit ensuite la relation d'ordre suivante :

Définition 31 *Soient c et c' deux systèmes de contraintes. La relation $\rightarrow$ "implique" sur les systèmes de contraintes est définie par :*

$$c \rightarrow c' \Leftrightarrow Sol_c \uparrow Var(c') \subseteq Sol_{c'}$$

On définit la relation $\rightarrow_a$ par restriction sur la relation $\rightarrow$. Soient a un atome, c et c' deux systèmes de contraintes.

$$c \rightarrow_a c' \Leftrightarrow Sol_c \uparrow Var(a) \subseteq Sol_{c'} \uparrow Var(a)$$

La relation $\preceq$ ($\preceq_a$) "est moins précis que" est définie comme l'inverse de $\rightarrow$ ($\rightarrow_a$). Soient c et c' deux systèmes de contraintes, $c \preceq c' \Leftrightarrow c' \rightarrow c$ (et de manière similaire $c \preceq_a c' \Leftrightarrow c' \rightarrow_a c$).

Info possède une borne supérieure, notée $\top$, et une borne inférieure (*fail*), dans *Info* pour $\subseteq$: $Sol_\top = Atom$ et $Sol_{fail} = \emptyset$.

Info est *solution compact* et vérifie la seconde hypothèse de cette propriété de manière un peu plus forte :

$$\forall c \in Info \ \neg c = \bigcup_{i \in I} c_i \text{ avec } I \text{ ensemble fini.}$$

Cette condition est nécessaire pour la spécialisation des clauses.

3.3.3 Spécifications complètes et valides

Une règle spécifiée est une règle dans laquelle on a associé à chaque atome une disjonction de schémas d'appel et une disjonction de schémas de solution. Ces schémas d'appel et de solution sont des systèmes de contraintes, et chaque schéma d'appel est le "père" d'au moins un schéma de solution (à chaque schéma d'appel correspond au moins un schéma de solutions). La spécification peut ainsi être vue comme un arbre de spécification. Les racines sont les schémas d'appel du premier atome du corps de la règle, et les feuilles sont les schémas de solution du dernier atome du corps de la règle. La tête de règle possède aussi des schémas d'appels et de solutions qui forment un second arbre de spécification. Les deux arbres sont reliés car les schémas d'appels de la tête sont les pères des schémas d'appels du premier atome du corps.

Définition 32 *On appellera arbre de spécification l'arbre associé à une règle, constitué par les spécifications du corps de cette règle. Chaque nœud de l'arbre est étiqueté par un système de contraintes.*

Soient c un système de contraintes et T un arbre de spécification. On désignera par *père(c)* (respectivement *fils(c)*) le système de contraintes qui est le père de c dans T (l'ensemble des systèmes de contraintes qui ont pour père c).

Définition 33 *On définit la relation $\vdash\!\sim$ "dérive" : pour c et s deux systèmes de contraintes, $c \vdash\!\sim s$ si et seulement si père(s) = c dans un arbre de spécification.*

Fonction génération(c,n) La fonction génération retourne tous les descendants de c à la $n^{\text{ième}}$ génération : $generation(c,n) = \{c' | c \overset{n}{\vdash\!\sim} c'\}$

On généralise maintenant les relations $\vdash\!\sim$, $\rightarrow$ et $\rightarrow_a$ aux disjonctions de systèmes de contraintes d'un même arbre de spécification :
soient C_1 et C_2 deux disjonctions de systèmes de contraintes,

- $C_1 \vdash\!\sim C_2 \Leftrightarrow \begin{cases} \forall c \in C_1 \ fils(c) \subseteq C_2 \\ \text{et réciproquement} \\ \forall c \in C_2 \ pere(c) \in C_1 \end{cases}$
- $C_1 \rightarrow C_2$ si et seulement si
  - $\forall c_1 \in C_1, \forall c_2 \in C_2$ tel que $c_1 \vdash\!\sim c_2 \Rightarrow c_1 \rightarrow c_2$
  - si pour un $c_1 \in C_1$, $\nexists c_2 \in C_2$ tel que $c_1 \vdash\!\sim c_2$, alors $c_1 \vdash\!\sim$ *fail*, et *fail* $\in C_2$

On généralise de la même façon la relation $\rightarrow_a$.

Pour plus de clarté, on confondra par la suite les notions de disjonctions et d'ensembles de systèmes de contraintes. On appellera ensemble de systèmes de contraintes tout ensemble construit sur *Info*.

Pour un atome spécifié par des ensembles de systèmes de contraintes (C,-a,S), on appellera C le schéma d'appels ou pré-schéma de a, S le schéma de solutions ou post-schéma de a.

Définition 34 *Soit P un programme spécifié. Sa spécification est complète si et seulement si, pour tout (C, a, S) apparaissant dans P, avec C et S ensembles de systèmes de contraintes, si on note Solutions(P,a) l'ensemble des solutions calculées par P pour le but a, alors*

$$\forall c \in C,\ \forall a' \in Sol_c(a)\ Solutions(P,a') \subseteq \bigcup_{s \in fils(c)} Sol_s(a')$$

Définition 35 *Soit P un programme spécifié. Sa spécification est valide si et seulement si elle est complète, et, pour chaque règle R de P de la forme*

$$(C, H, S) \leftarrow c_{\mathcal{X}}, (C_1, B_1, S_1), (C_2, B_2, S_2), \ldots, (C_n, B_n, S_n)$$

où $c_{\mathcal{X}}$ est un système de contraintes défini sur $\mathcal{X}$ et chaque C_i et S_i sont des ensembles de systèmes de contraintes,

1. $C \preceq C_1 \preceq S_1 \preceq \ldots \preceq C_i \preceq S_i \preceq C_{i+1} \preceq \ldots \preceq C_n \preceq S_n$
2. $S \preceq_H S_n$
3. $\forall (c, B_i, s)$ *un atome spécifié du corps de la règle, avec* $c \vdash\!\sim s$, *il existe au moins une règle de P de même symbole de prédicat que B_i*

   $$(C', H', S') \leftarrow (C'_1, D_1, S'_1), \ldots, (C'_n, D_n, S'_n)$$

   telle que $\exists c' \in C', s' \in S', c' \vdash\!\sim s' \Rightarrow c \rightarrow c'$ *et* $s \rightarrow s'$

Un ensemble de systèmes de contraintes qui apparaît dans une règle spécifiée correspond à l'ensemble des systèmes de contraintes qui apparaissent à une même hauteur dans l'arbre de spécification.

Un programme non spécifié peut-être considéré comme un programme dont l'arbre de spécification est l'arbre déterministe dont tous les nœuds sont l'ensemble $\{\top\}$. Cette spécification est correcte. Toute spécification peut donc être rendue valide par ajout de cette branche. Un algorithme de validation d'une spécification est :

Soit P un programme (partiellement) spécifié. Pour chaque règle de P, faire :

- ajouter $\top$ au pré-schéma de la tête de règle;
- puis, on parcourt l'arbre de spécification en largeur d'abord et de haut en bas, et, pour chaque système de contraintes c :
  - si *père(c)* n'existe pas, alors $\top \vdash\!\sim c$

- soit s un autre système de contraintes. Si $c \vdash\!\sim s$ mais
  - $s \supseteq c$, alors $c \vdash\!\sim c$, et s est supprimé
  - c et s ne sont pas comparables, alors $c \vdash\!\sim c$ et $c \vdash\!\sim cs$
- si $\forall a' \in Sol_c(a)\ \mathit{Solutions}(P,a) \not\subseteq \bigcup_{s \in fils(c)} Sol_s(a')$, alors $c \vdash\!\sim \mathit{fils}(c) \cup \{c\}$

- le post-schéma de la tête de la règle est égal à l'union des post-schémas du dernier atome du corps.

Cet algorithme n'est absolument pas efficace dans le cas général, et on suppose que chaque analyseur de programmes (interprétation abstraite, utilisateur, ...) dispose de son propre algorithme de validation de spécification. Néanmoins, cet algorithme peut être utile dans le cas de systèmes qui ne fourniraient que des contraintes pertinentes (pour obtenir telle ou telle propriété).

L'interprétation abstraite proposée par [LDL92b] est basée sur la résolution OLDT. Elle énumère tous les schémas d'appels (et de solutions associés) qui sont générés pendant la résolution. Notons $\times$ le produit cartésien sur des ensembles de systèmes de contraintes. Notons C_A (S_A) le schéma d'appels (de solutions) exhibé par l'interprétation abstraite pour l'atome A. L'algorithme de spécification pour cette interprétation abstraite est :

- pour toute règle $H \leftarrow c_\chi, B_1, B_2, \ldots B_n$, on opère la transformation suivante :
  - sur le début de la règle :

    $$(C_H, H, S_H) \leftarrow c_\chi, (C_H \times C_{B_1} \times c_\chi, B_1, ((C_H \times C_{B_1} \times c_\chi) \times_s S_{B_1}), B_2, ..$$

  - puis, en parcourant la règle de gauche à droite, sur chaque littéral :

    $$(C_{i-1}, B_{i-1}, S_{i-1}), B_i$$

    est remplacé par

    $$(C_{i-1}, B_{i-1}, S_{i-1}), (S_{i-1} \times C_{B_i}, B_i, S_{i-1} \times S_{B_i})$$

3.3.4 Opérations sur des systèmes de contraintes

Définition 36 *Soient c un système de contraintes et C un ensemble de systèmes de contraintes. On définit l'ensemble des plus grands minorants de c dans C par:*

$$pgm(c, C) = \{c_1 \in C \mid c_1 \rightarrow c \text{ et } \nexists\, c_2 \in C, c_2 \neq c_1 \text{ tel que } c_1 \rightarrow c_2 \text{ et } c_2 \rightarrow c\}$$

On définit l'union (ou unification) de deux systèmes de contraintes c_1 et c_2 par :

- $c_1.c_2 = c_1c_2 = c_1 \cup c_2$ si $c_1 \cup c_2$ est solvable;
- $c_1c_2 = \mathit{fail}$ sinon;

et le calcul des contraintes communes à (ou l'antiunifié de) c_1 et c_2 par $c_1 \cap c_2$.

Par extension, on définit l'instanciation d'un ensemble de systèmes de contraintes C par un système de contraintes σ par : $\sigma C = \bigcup_{c \in C} \sigma c$

Enfin, on définit le filtrage d'un atome spécifié par un autre atome spécifié.

Définition 37 *Soient (C, a, S) et (C', b, S') deux atomes spécifiés (par des ensembles de systèmes de contraintes), on dit que (C, a, S) filtre (C', b, S') si :*

- *a et b s'unifient;*
- *$\exists c' \in C'$, $c \in C$ tels que $c' \rightarrow_{\{a,b\}} c$*

3.3.5 Opérations sur des arbres de spécifications

Dans la suite, on appellera arbre de spécification n'importe quel arbre ou sous-arbre d'une spécification **valide**, et arbre déterministe un arbre composé d'une seule branche.

Propagation de contraintes sur un arbre de spécification Soient σ un système de contraintes, et T un arbre de spécification. σT est l'arbre de spécification dont les nœuds sont les nœuds de T instanciés par σ. La propagation se fait de la racine vers les feuilles.

Fusion de branches On peut toujours généraliser un arbre de spécification. Soit c un système de contraintes :

- si $\exists c$, $c' \in \mathit{fils}(c)$ tels que $c \rightarrow c'$, alors on peut supprimer le sous-arbre de racine c;
- on peut toujours remplacer l'arbre de spécification de racine c (ou une partie des arbres issus de c) par une branche déterministe dont chaque nœud est l'antiunifié de l'ensemble des systèmes de contraintes auquel il correspond.

Ajout d'une branche Soient c un nœud d'un arbre de spécification, T l'arbre de spécification de racine c, et σ une substitution spécifiée. On peut toujours créer un nouvel arbre de spécification σT (par propagation de la contrainte σ sur T), tel que si $c' \vdash c$, alors $c' \vdash \sigma c$.

Suppression d'une branche Tout sous-arbre déterministe d'un arbre de spécification terminée par un *fail*, et qui ne porte pas sur un prédicat *effaçable par échec*, est remplacée par *fail* à sa racine.

Extension aux programmes logiques avec prédicats prédéfinis Les opérations sur les arbres de spécification que nous proposons supposent que les littéraux du programme soient monotones (si σG a une réfutation, alors G a une réfutation), ce qui est faux pour la plupart des prédicats prédéfinis. Nous utilisons donc la table des propriétés conservées par de tels prédicats (cf. 3.2.1). Cette table est nécessaire pour contrôler la propagation d'informations sur les arbres de spécification, et pour effectuer les transformations de programmes.

Exemple 30
Considérons l'atome spécifié $(c, \texttt{var(X)}, s)$. Si on ajoute la contrainte $\{\texttt{X=a}\}$, on obtient $(c \cup \{\texttt{X=a}\}, \texttt{var(X)}, \mathit{fail})$. Comme var est un prédicat déterministe, $(c \cup \{\texttt{X=a}\}, \texttt{var(X)}, \mathit{fail}) \equiv (\mathit{fail}, \texttt{var(X)}, \mathit{fail})$

3.4 L'opérateur Specif

Après la phase de spécification, une première étape est d'instancier le programme par les informations de l'analyseur statique :

Instanciation Soit P un programme spécifié. Pour chaque règle de P, faire

- pour chaque atome spécifié (C, A, S) de la règle (tête et corps), et de gauche à droite, faire
  - appliquer $\mathit{lub}(\mathit{Sol}_{\cup C})$ sur A et tous les atomes à droite de A, où $\cup C$ est l'antiunifié des systèmes de contraintes appartenant à C

On appelera **nettoyage** l'étape qui consiste à supprimer tout arbre de spécification d'un programme Prolog spécifié : cette opération retourne un programme Prolog complet non spécifié.

On définit l'opérateur *Specif* par :

$$\mathit{Specif} = \mathit{nettoyage} \circ \mathit{instanciation} \circ \mathit{spécification} \circ \varphi_2 \circ \varphi_1$$

Intuitivement, cet opérateur permet de souligner les informations concrètes données par la spécification sur les atomes prédéfinis. Si P est un programme logique pur, le plus souvent $\mathit{Specif}(P) = \mathit{Specif}(\mathit{Specif}(P))$: l'analyseur statique ne déduit pas plus de choses au deuxième passage qu'au premier (c'est le cas de l'interprétation abstraite de [Lec94]). On se restreindra d'ailleurs au cas où l'analyseur statique est idempotent. Un programme Prolog comportant des prédicats prédéfinis comme le `read`, `write`, `!`, `retract` ... peut être considéré comme un programme Prolog pur, dû à la manière dont ils sont éliminés (et

simulé, pour le **retract**). Par contre, la transformation φ_2 pour le prédicat **assert** peut être améliorée par une itération de l'opérateur *Specif*.

Exemple 31
Soit P le programme

```
main(Y,W) :- p(W,X), q3(X,Y).
p(Z,X) :- q1(Z,Y), assert(r(Y)), q2(X).
q1(0,a).
q1(s(X),Y) :- q1(X,a).
q2(X) :- r(X).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```

Le programme *épuré* $(\varphi_2 \circ \varphi_1)$ est :

```
main(Y,W) :- p(W,X), q3(X,Y).
p(Z,X) :- q1(Z,Y), assert(r(Y)), q2(X).
q1(0,a).
q1(s(X),Y) :- q1(X,Y).
q2(X) :- r(X).
r(_).
assert(_).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```

L'interprétation abstraite, sur ce programme, donne le schéma d'appel {Y=a} pour q1(Y). Après la spécification, on obtient :

```
main(Y,W) :- p(W,X), q3(X,Y).
p(Z,X) :- q1(Z,Y), {Y=a}, assert(r(Y)), q2(X).
q1(0,a).
q1(s(X),Y) :- q1(X,Y).
q2(X) :- r(X).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```

L'instanciation retourne le programme suivant :

```
main(Y,W) :- p(W,X), q3(X,Y).
p(Z,X) :- q1(Z,Y), assert(r(a)), q2(X).
q1(0,a).
q1(s(X),Y) :- q1(X,Y).
```

```
q2(X) :- r(X).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```

Dans cet exemple, une itération de l'opérateur *Specif* permettrait à l'interprétation abstraite de fournir plus de renseignements sur le comportement de ce programme: les liaisons entre `r(_)` et `q1(Y)` sont cachées par l'élimination des prédicats prédéfinis. La spécification et l'instanciation permettent de les révéler. On continue l'exemple:

Le programme *épuré* $(\varphi_2 \circ \varphi_1)$ est maintenant:

```
main(Y,W) :- p(W,X), q3(X,Y).
p(Z,X) :- q1(Z,a), assert(r(a)), q2(X).
q1(0,a).
q1(s(X),Y) :- q1(X,Y).
q2(X) :- r(X).
r(a).
assert(_).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```

L'interprétation abstraite, sur ce programme, donne le schéma de solution `{X=a}` pour `q2(X)`. Après la spécification, on obtient:

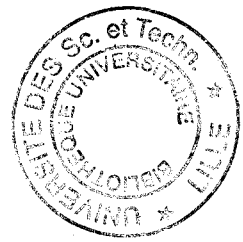
```
main(Y,W) :- p(W,X), {X=a}, q3(X,Y).
p(Z,X) :- q1(Z,a), assert(r(a)), q2(X), {X=a}.
q1(0,a).
q1(s(X),Y) :- q1(X,Y).
q2(X) :- {X=a}, r(X).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```

L'instanciation retourne le programme suivant:

```
main(Y,W) :- p(W,a), q3(a,Y).
p(Z,a) :- q1(Z,a), assert(r(a)), q2(a).
q1(0,a).
q1(s(X),Y) :- q1(X,Y).
q2(a) :- r(a).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```

On peut maintenant effectuer un dépliage déterministe sur **q3** et **p**. Le programme final est alors :

```
main(b,W) :- q1(W,a), assert(r(a)),q2(a).
p(Z,a) :- q1(Z,a), assert(r(a)), q2(a).
q1(0,a).
q1(s(X),Y) :- q1(X,Y).
q2(a) :- r(a).
q3(a,b).
q3(b,Y) :- q3(X,Z).
:- main(X,Y).
```



Dans le cas d'un littéral comme **assert(r(X) :- assert(...))**, trois itérations de *Specif* auraient été nécessaires. Si l'analyseur statique est idempotent pour les programmes logiques purs, il existe un point fixe pour l'opérateur *Specif*, qui est atteint après un nombre fini d'étapes, borné par le nombre de **assert** dans le programme et leur profondeur.

Cet opérateur permet d'apporter une réponse (partielle) à la défaillance de notre fonction φ_2 dans le cas de littéraux **assert(X)** avec **X** variable libre.

3.5 Transformations de programmes spécifiés

Les techniques classiques de définition, dépliage et pliage proposées par [TS84] sont aménagées pour d'une part conserver notre équivalence opérationnelle, et, d'autre part, pour prendre en compte les contraintes du programme. Les arbres de spécifications permettent d'affiner un peu les conditions d'application de ces transformations. Nous étendons d'abord la définition de l'équivalence opérationnelle forte que nos transformations doivent préserver aux contraintes.

Définition 38 *Soient (P_1, B_1) et (P_2, B_2) deux couples programmes-buts. Ils sont fortement opérationnellement équivalents si et seulement si, en considérant leur arbre de résolution :*

- *il existe une bijection*
  $$\theta : \textit{Nœuds-solution}(P_1, B_1) \rightarrow \textit{Nœuds-solution}(P_2, B_2)$$
  telle que, si on note c_s le système de contraintes étiquetant le nœud solution s de Arbre(P_1, B_1), *alors* $\forall s \in$ Nœuds-solution(P_1, B_1), $Sol_{c_s}(B_1) = Sol_{c_{\theta(s)}}(B_2)$

- *il existe une bijection θ' sur les branches succès et les branches infinies qui est monotone par rapport à la relation d'ordre $\leq$ "est à gauche de ".*

Un programme est composé d'un ensemble de clauses, et d'un ensemble de définitions, qui sont mises-à-jour par l'opération de définition, et utilisées par le pliage. Les définitions suivantes sont les extensions des définitions données dans la section 1.2 aux programmes Prolog complet spécifiés. Tous les exemples sont donnés par rapport à l'interprétation abstraite de [Lec94].

3.5.1 Définition

Définition
Soient P un programme spécifié, D son ensemble de définitions et R une clause de la forme

$$(C_1, H, S_n) \leftarrow c_\chi, (C_1, B_1, S_1), \ldots (C_n, B_n, S_n)$$

avec

- le prédicat de H est un nouveau symbole de prédicat qui n'apparaît nulle part ailleurs dans P;
- les arguments de H sont les variables (distinctes entre elles) qui apparaissent dans $B_1, \ldots, B_n$;
- $B_1, \ldots, B_n$ sont des atomes dont les symboles de prédicats apparaissent déjà dans P et
  - soit la contrainte et la séquence de littéraux $c_\chi, (C_{i_1}, B_1, S_{i_1}), \ldots (C_{i_n}, B_n, S_{i_n})$ apparaît dans la $i^{\text{ième}}$ clause de P, alors $\forall j \in [1, n]$ $C_j = \bigcup C_{i_j}$ et $S_j = \bigcup S_{i_j}$
  - soit cette séquence de littéraux n'apparaît pas dans P, et alors une phase de spécification est nécessaire pour calculer l'arbre de spécification de R.

R est ajoutée à P et à D.

Les règles de D ne peuvent pas définir un prédicat dynamique.

3.5.2 Dépliage

Dépliage en tête
Soient P un programme spécifié et R une règle de P. R est de la forme:

$$(C, H, S) \leftarrow c_\chi, (C_1, B_1, S_1), (C_2, B_2, S_2), \ldots, (C_n, B_n, S_n)$$

R peut être dépliée par rapport à (C_i, B_i, S_i) si tous les littéraux à gauche de B_i sont déterministes, sans effets de bord, *instanciables** et *effaçables par échec**. R ne doit pas définir un prédicat dynamique.

Soient R_1, R_2, ..., R_k les clauses de P, (C'_1, H_1, S'_1), (C'_2, H_2, S'_2), ..., (C'_k, H_k, S'_k) leur littéral de tête et $c_{\chi_1}, \ldots, c_{\chi_k}$ leur contrainte telles que:

- $\forall j \in \{1, \ldots, k\}$, chaque (C'_j, H_j, S'_j) filtre (C_i, B_i, S_i), et $c_{\chi_j} \cup c_{\chi'} \cup H_j = B_i$ est consistant;
- elles ne contiennent pas de !;

Alors, la clause R est remplacée par la suite de clauses R'_1, R'_2, ..., R'_k, et chaque R'_j est obtenue par les étapes suivantes:

- $\forall c \in C_i$
  - calculer $\text{pgm}(c, C'_j)$
  - si $\text{pgm}(c, C_i) \neq \emptyset$, $\forall c'_k \in \text{pgm}(c, C_i)$
    - calculer le système de contraintes σ_k tel que $\sigma_k c'_k = c$
    - propager σ_k sur l'arbre de racine c'_k.
- (C_i, B_i, S_i) est remplacé dans R par $(C_i, B_i = H_j, \cup\{\mathit{fils}(\sigma_k c'_k)\})$ (avec les $\sigma_k c'_k$ calculés précédemment), suivi du corps de R_j, avec l'arbre de spécification de racine $\cup\{\mathit{fils}(\sigma_k c'_k)\}$
- $\forall c \in C_i$
  - calculer $S_1 = \mathit{fils}(c)$ dans R et $S_2 = \mathit{generation}(c, 2m)$ dans R'_j (m étant le nombre d'atomes du corps de R_j)
  - pour chaque élément s_1 de S_1 et pour chaque élément s_2 de S_2
    - calculer le système de contraintes θ tel que $s_2 = \theta s_1$
    - propager θ sur l'arbre de racine s_1
    - $s_2 \vdash \theta s_1$ dans R'_j

Dépliage déterministe

Soient P un programme spécifié et R une règle de P. R est de la forme:

$$(C, H, S) \leftarrow c_{\chi'}, (C_1, B_1, S_1), (C_2, B_2, S_2), \ldots, (C_n, B_n, S_n)$$

R peut être dépliée par rapport à (C_i, B_i, S_i) si les littéraux à gauche de B_i sont *instanciables** et *effaçable par échec** et (C_i, B_i, S_i) n'est filtré que par une seule tête de règle, et que cette règle ne contient pas de !. R ne doit pas définir un prédicat dynamique.

La procédure de dépliage est la même que pour le dépliage en tête.

Les conditions marquées par un * dans les définitions du dépliage déterministe et en tête ne sont nécessaires que si on se place dans un cadre opérationnel à la Prolog III. Elles peuvent être supprimées, en restant dans ce

cadre, si on ne déplie pas les contraintes des règles dépliantes: on définit des règles avec un nouveau nom de prédicat, dont le corps est uniquement constitué des contraintes d'une règle dépliante. Ces règles sont appelées juste devant les littéraux dépliés, et les contraintes des règles dépliées ne sont alors pas modifiées. Si on se place dans le cadre opérationnel de $CLP(\mathcal{X})$, alors les conditions marquées d'un $*$ sont supprimées, les contraintes $c_\mathcal{X}$ ne représentent que les contraintes sur $\mathcal{X}$ qui sont effectivement placées en début de règle, et les contraintes $c_{\mathcal{X}_j}$ des clauses dépliantes sont placées dans le corps de la clause dépliée au même endroit que dans le corps de la clause dépliante (la condition $c_{\mathcal{X}_j} \cup c_\mathcal{X} \cup H_j = B_i$ est consistant est supprimée sauf si B_i est le premier atome du corps de la règle dépliée et si les $c_{\mathcal{X}_j}$ sont toutes en début de la clause dépliante).

Exemple 32
Dans cet exemple de [HJ92], on utilise l'ordre de Dewey pour étiqueter les nœuds de l'arbre de spécification.

```
main(Y) :- p(X),q(X,Y).

p(f(Z1)) :- r1(Z1).
p(h(Z2)) :- r2(Z2).

q(f(g(W1,W2),W1).
q(h(Z3),Z3).

r1(g(a,W3)).

r2(b).
r2(f(V)) :- r2(V).
```

Prédicats	Schémas d'appels	Schémas de solutions
main	main(X)	X/a
		Id
p	p(X)	X/f(g(a,W))
		X/h(Y)
r1	r1(X)	X/g(a,W)
r2	r2(X)	X/b
		X/f(Y)
q	q(f(g(a,X)),X)	X/a
	q(h(b),X)	X/b
	q(h(f(b)),X)	X/f(b)
	q(h(f(f(Y))),X)	X/f(f(Y))

Spécification des trois premières règles :

```
({1},main(Y),{11,Y/a}+{12,Id}) :-
```

```
    ({11},p(X),{111,X/f(g(a,W))}+{112,X/h(Z)}),

    ({1111,X/f(g(a,W)),q(X,W)}          {11111,W/a,Y/W}
    +{1121,X/h(Z),Z/b}          ,q(X,Y),+{11211,Y/Z}
    +{1122,X/h(Z),Z/f(b)}               +{11221,Y/Z}
    +{1123,X/h(Z),Z/f(f(Z1))}           +{11231,Y/Z}).

({1},p(f(X)),{11,f(g(a,W))}) :-
    ({11},r1(X),{111,X/g(a,W)}).

({1},p(h(X)),{11,X/h(Y)}) :-
    ({11},r2(X),{111,X/b}+{112,X/f(Y)}).
```

Dépliage en tête de la première règle :

```
({1},main(Y),{11,Y/a}) :-
    ({11},p(X)=p(f(X')),{111,X/f(X')}),

    ({1111,X/f(X')},r1(X'),{11111,X'/g(a,W)}),

    ({111111,X/f(X'),
      X'/g(a,W),q(X,W)}, q(X,Y), {1111111,W/a,Y/W}).

({1},main(Y),{11,Id}) :-
    ({11},p(X)=p(h(X')),{111,X/h(X')}),

    ({112,X/h(X')},r2(X),{1121,X/h(X'),X'/b}
                         +{1122,X/h(X'),X'/f(Z)}),

    ({11211,X/h(X'),X'/b}                          {112111,Y/X'}
    +{11221,X/h(X'),X'/f(Z),Z'/b}     , q(X,Y), +{112211,Y/Z'}
    +{11222,X/h(X'),X'/f(Z),Z'/f(Z1)}           +{112221,Y/Z'})
```

Élimination des égalités :

```
({1},main(Y),{11,Y/a}) :-
    ({11,X/f(X')},r1(X'),{111,X'/g(a,W)}),

    ({111,X'/g(a,W),q(f(X'),W)}, q(f(X'),Y), {11111,W/a,Y/W}).

({1},main(Y),{11,Id}) :-
    ({11,X/h(X')},r2(h(X')), {111,X'/b}
                             +{112,X'/f(Z)}),

    ({1111,X'/b}                              {11111,X'/b,Y/X'}
```

```
+{1121,X'/f(Z),Z/b}     , q(h(X'),Y),+{11211,X'/f(Z),Z/b,Y/Z}
+{1122,X'/f(Z),Z/f(Z1)}                +{11221,X'/f(Z),Z/f(Z1),
                                              Y/Z})
```

3.5.3 Pliage

Nous distinguons le "pliage classique", où la clause pliante a un arbre de spécification compatible avec celui de la clause pliée, et le "pliage-spécification", où la clause pliante n'a pas un tel arbre de spécification : on lui rajoute alors cet arbre.

Pliage classique
Soient P un programme spécifié, D son ensemble de définitions et R_1 une règle de P, de la forme :

$$\begin{aligned}(C,H,S) \quad \leftarrow \quad & c_{\chi_1} \cup c_{\chi_2}, (C_1,B_1,S_1),\ldots,\\ & \theta((C_i,B_i,S_i),\ldots(C_{i+k},B_{i+k},S_{i+k})),\\ & \ldots,(C_n,B_n,S_n)\end{aligned}$$

où θ est une substitution classique, et R_2 une règle de D de la forme :

$$(C',H',S') \leftarrow c_{\chi_2}, (C'_1,B_i,S'_1),\ldots,(C'_{k+1},B_{i+k},S'_{k+1})$$

telle que

- $\theta C_i \rightarrow_{B_i} C'_1$,
- $\theta H'$ s'unifie uniquement avec la tête de R_2,
- les littéraux à gauche de B_i sont sans effet de bord et ne sont pas des prédicats-test,
- θ appliquée sur R_2 n'instancie pas les variables locales de R_2,
- et R_2 ne contient pas de !.

R_1 est alors transformée en :

$$(C,H,S) \leftarrow c_{\chi_1}, (C_1,B_1,S_1),\ldots,(\theta C_i,\theta H',\theta S_{i+k}),\ldots,(C_n,B_n,S_n)$$

Pliage-Spécification
Soient P un programme spécifié, D son ensemble de définitions et R_1 une règle de P, de la forme :

$$\begin{aligned}(C,H,S) \quad \leftarrow \quad & c_{\chi_1} \cup c_{\chi_2}, (C_1,B_1,S_1),\ldots,\\ & \theta((C_i,B_i,S_i),\ldots(C_{i+k},B_{i+k},S_{i+k})),\\ & \ldots,(C_n,B_n,S_n)\end{aligned}$$

où θ est une substitution classique, et R_2 une règle de D de la forme :

$$(C',H',S') \leftarrow c_{\chi_2}, (C'_1,B_i,S'_1),\ldots,(C'_{k+1},B_{i+k},S'_{k+1})$$

telle que

- $\neg(C_i \rightarrow_{B_i} C'_1)$,
- $\theta H'$ s'unifie uniquement avec la tête de R_2,
- les littéraux à gauche de B_i sont sans effet de bord et différents du prédicat **var**,
- θ appliquée sur R_2 n'instancie pas les variables locales de R_2,
- et R_2 ne contient pas de !.

Alors R_2 est transformée en :

$$(C' \cup C_i, H', S' \cup S_{i+k}) \leftarrow c_{\chi_2}, (C'_1 \cup C_i, B_i, S'_1 \cup S_i), \ldots, (C'_{k+1} \cup C_{i+k}, B_{i+k}, S'_{k+1} \cup S_{i+k})$$

et R_1 en :

$$(C, H, S) \leftarrow c_{\chi_1}, (C_1, B_1, S_1), \ldots, (C_i, \theta H', S_{i+k}), \ldots, (C_n, B_n, S_n)$$

3.5.4 D'autres transformations classiques

Méthode de Gallagher-Bruynooghe [GB90]

Nous reprenons ici la transformation proposée par Gallagher et Bruynooghe dans [GB90], et présentée dans la section 2.2.2.

Soient P un programme Prolog et p un prédicat (non dynamique) défini par n clauses :

$$\begin{aligned}
(C_1, p(\tilde{t}_1), S_1) &\leftarrow c_{\chi_1}, (C_{b_1}, B_1, S_{b_1}) \\
(C_2, p(\tilde{t}_2), S_2) &\leftarrow c_{\chi_2}, (C_{b_2}, B_2, S_{b_2}) \\
&\vdots \\
(C_n, p(\tilde{t}_n), S_n) &\leftarrow c_{\chi_n}, (C_{b_n}, B_n, S_{b_n})
\end{aligned}$$

La définition de p est remplacée par la clause :

$$(C, p(\tilde{T}), S) \leftarrow (C, q(X_1, X_2, \ldots, X_k), S)$$

où

- q est un nouveau symbole de prédicat, qui n'apparaît nulle part ailleurs dans P;
- $\tilde{T} = \bigwedge_{i \in [1,n]} \tilde{t}_i$;
- les arguments de q sont les variables (distinctes entre elles) qui apparaissent dans $\tilde{T}$;
- $C = \bigcup_{i \in [1,n]} C_i$ et $S = \bigcup_{i \in [1,n]} S_i$

La définition de q est alors :

$$\begin{array}{rcl}(C_1, q(\theta_1 X_1, \theta_1 X_2, \ldots, \theta_1 X_k), S_1) & \leftarrow & c_{\chi_1}, (C_{b_1}, B_1, S_{b_1}) \\ (C_2, q(\theta_2 X_1, \theta_2 X_2, \ldots, \theta_2 X_k), S_2) & \leftarrow & c_{\chi_2}, (C_{b_2}, B_2, S_{b_2}) \\ & \vdots & \\ (C_n, q(\theta_n X_1, \theta_n X_2, \ldots, \theta_n X_k), S_n) & \leftarrow & c_{\chi_n}, (C_{b_n}, B_n, S_{b_n})\end{array}$$

avec $\theta_i \tilde{T} = \tilde{t_i}$.

Puis, partout où apparaît un littéral $(C', p(\tilde{t}), S')$ dans P, il est remplacé par

$$(E_\theta \cup C, \theta q(X_1, X_2, \ldots, X_k), E_\theta \cup S),$$

où θ est le mgu de $p(\tilde{t})$ et $p(\tilde{T})$, et E_θ est le système d'équations associé à θ.

Permutation de deux atomes dans le corps d'une règle
Soit R une règle spécifiée de la forme :

$$(C, H, S) \leftarrow c_\chi, B_{efore}, (C_1, B_1, S_1), (C_2, B_2, S_2), A_{fter}$$

où

- B_{efore} et A_{fter} sont des séquences d'atomes spécifiés;
- B_2 est un atome déterministe et sans effet de bord;
- la résolution de B_1 ne comporte pas d'appel à un prédicat à effet de bord ou non monotone sous les conditions C_1;

alors on peut permuter B_1 et B_2 dans R, et on obtient :

$$(C, H, S) \leftarrow c_\chi, B_{efore}, (C_1, B_2, S'_1), (S'_1, B_1, S_2), A_{fter}$$

Comme on ne manipule que des prédicats sans effets de bord (B_1 et B_2), le *switching lemma* [Llo87] nous assure la conservation de l'ensemble des substitutions-réponses, et comme l'un des atomes est déterministe, l'ordre des solutions ne peut pas être changé.

3.6 Spécialisation de règles spécifiées

3.6.1 Programmes gardés

Dans cette partie, nous introduisons des "gardes" dans les clauses pour les spécifier, c'est-à-dire mettre en exclusion mutuelle la clause spécialisée pour un appel et la clause pour le cas général. Les "gardes" sont des contraintes définies sur *Info* et résolues par un solveur de contraintes spécialisé pour ces spécifications. Si la garde s'efface, alors le reste de la clause est évalué (contraintes et littéraux), sinon c'est un échec (la garde est évaluée à $\bot$). Pour une garde négative, on utilise le principe de la négation par l'échec.

Les gardes sont les seules spécifications qui restent dans le programme à l'exécution.

Par analogie avec les programmes gardés (GHC), on séparera par l'opérateur | (*commit choice*) les gardes des contraintes.

3.6.2 Algorithmes de transformations

Nous voulons passer d'un programme spécifié à un programme typé. Dans un programme spécifié, les schémas d'appels des têtes de clauses expriment toutes les conditions sous lesquelles la clause va être appelée. Dans un programme typé, la garde de la clause exprime les conditions *à vérifier* pour que la clause soit évaluée. La partition des clauses revient à diviser une clause spécifiée par un ensemble de clauses de systèmes d'équations en plusieurs clauses, chacune typée par un système d'équations de l'ensemble de départ.

L'algorithme suivant (spécialisation d'un littéral dans une clause) permet d'effectuer cette partition pour que dans l'une des clauses résultant de la partition un littéral ait, quelque soit son schéma d'appel dans cette clause, une certaine propriété.

Cet algorithme ne suffit pas toujours. Si on veut appliquer un dépliage déterministe sur un littéral, il faut d'abord effectuer une spécialisation du littéral dans la clause par rapport au déterminisme. Mais ensuite, rien ne garantit qu'on saura quelle clause utiliser pour le dépliage.

Exemple 33

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Si le schéma d'appel de `add(X,Y,Z)` est {`X est clos`}, alors `add(X,Y,Z)` est déterministe. Mais ça ne suffit pas pour appliquer le dépliage déterministe.

Le dernier algorithme permet cette partition-spécialisation.

Partition de clauses

Soit R une règle spécifiée de tête (C, H, S). Le résultat de la partition de R par rapport au système de contraintes E est deux clauses R_1 et R_2 de littéraux identiques mais d'arbre de spécifications disjoints. Si on note T_c l'arbre de spécification de racine c, alors $T = \bigcup_{c \in C} T_c$ est l'arbre de spécification de R et $T_1 = \{T_c \in T \mid c \rightarrow E\}$ et $T_2 = T - T_1$ sont les arbres de spécification de R_1 et R_2.

R_1 et R_2 sont en exclusion mutuelle (on ajoute à R_1 la garde E et à R_2 la garde $\neg E$).

Spécialisation d'un littéral dans une clause (par rapport à une propriété)
Soit R une règle spécifiée de la forme:

$$(C, H, S) \leftarrow c_\chi, B_{efore}, (C', L_{itteral}, S'), A_{fter}$$

où

- B_{efore} et A_{fter} sont des séquences d'atomes spécifiés;
- $C' = C'_P \cup C'_{?P}$, C'_P est l'ensemble des systèmes de contraintes d'appels de $L_{itteral}$ qui vérifient la propriété P, et $C'_{?P} = C' - C'_P$
- $C'_P \upharpoonright\sim S'_P$ et $C'_{?P} \upharpoonright\sim S'_{?P}$

1er cas: C peut se diviser en deux sous-ensembles $C = C_P \cup C_{?P}$ tels que:

$$\left.\begin{array}{l}(1)\ \forall c' \in C'_P,\ \exists c \in C_P \\ (2)\ \forall c \in C_P,\ \exists c' \in C'_P\end{array}\right\} \text{ tels que } c \overset{*}{\upharpoonright\sim} c'$$

alors on peut partitionner R par rapport à C_P (on obtient deux clauses R_P et $R_{?P}$). Si les deux conditions sont vérifiées, alors on dira que R_P est complètement spécialisée pour $L_{itteral}$ par rapport à P. Si seule la condition (2) est vérifiée, alors on dira que R_P est partiellement spécialisée.
2^{ième} cas: $\forall c_1 \in C'_{?P}$, $\exists c_2 \in C_P$, $\exists \sigma \in Const$ tels que $c_2 = \sigma c_1$ et $Var(\sigma) \subseteq Var(H)$ alors on ajoute au programme une règle R' construite à partir de R:

$$(C, H, S) \leftarrow \sigma \mid c_{\chi'}, B_{efore}, (C'_P \cup \sigma C'_{?P}, L_{itteral}, S''), A'_{fter}$$

et on ajoute la garde $\neg\sigma$ à R.

Déterminisation d'une clause
Soit R une règle spécifiée de la forme:

$$(C, H, S) \leftarrow c_\chi, B_{efore}, (C', L_{itteral}, S'), A_{fter}$$

où

- B_{efore} et A_{fter} sont des séquences d'atomes spécifiés;
- C' l'ensemble des systèmes d'équations d'appels de $L_{itteral}$. $L_{itteral}$ est déterministe pour chacun des systèmes d'équations de C'. On notera par σ_i les systèmes d'équations de C': $C' = \bigcup_{i\in[1,n]} \sigma_i$.

On note C'_H la restriction des éléments de C' aux variables de H: $C'_H = \{c \upharpoonright Var(H) | c \in C'\}$.

Soient $D_1, \ldots D_m$ les règles qui s'unifient avec $L_{itteral}$. On note Ψ l'ensemble des mgus de $L_{itteral}$ avec les têtes des règles D_i : $\Psi = \{\psi_i | \psi_i = mgu(L_{itteral}, tête(D_i), i \in [1,m]\}$.

Enfin on calcule le surplus d'informations contenu dans les mgus par rapport aux schémas d'appels : $\delta_{i,j} = \psi_i \uparrow Var(\sigma_j)$ avec $i \in [1,m]$, $j \in [1,n]$.

- 1^{er} cas : $\forall i, i' \in [1,m]$ et $\forall j, j' \in [1,n]$ tels que $(i,j) \neq (i',j')$, alors $\delta_{i,j} \neq \delta_{i',j'}$

  Dans ce cas, la clause peut être rendue entièrement déterministe. Le résultat est $m+1$ clauses C_i construites à partir de C, de la forme :

$$\forall 1 \leq i \leq m \quad C_i \ : \ (C,H,S) \leftarrow \bigcup_{j \in [1,n]} \delta_{i,j} | c_\chi, B_{efore}, (C', L_{itteral}, S'), A_{fter}$$

$$c_{m+1} \ : \ (C,H,S) \leftarrow \neg(\bigvee_{i \in [1,m]} \bigcup_{j \in [1,n]} \delta_{i,j}) | c_\chi, B_{efore}, (C', L_{itteral}, S'), A_{fter}$$

  où $\bigvee$ dénote la disjonction;

- $2^{ème}$ cas : le premier cas n'est pas vérifié, mais $\exists j \in [1,n]$ tel(s) que $\exists i, \forall i' \in [1,m], i \neq i'$ $\delta_{i,j} \neq \delta_{i',j}$ (1), alors on peut rendre partiellement déterministe la clause C. Soit Δ l'ensemble de tous les $\delta_{i,j}$ qui vérifient la condition (1). Soit k le nombre d'indices i différents dans Δ. Le résultat de la partition sera $k+1$ clauses C_i de la forme :

  $\forall 1 \leq i \leq k$

$$C_i : \ (C,H,S) \leftarrow \bigcup_{\forall j \in [1,n], \delta_{i,j} \in \Delta} \delta_{i,j} | c_\chi, B_{efore}, (C', L_{itteral}, S'), A_{fter}$$

$$c_{k+1} \ : \ (C,H,S) \leftarrow \neg(\bigvee_{\delta \in \Delta} \delta) | c_\chi, B_{efore}, (C', L_{itteral}, S'), A_{fter}$$

Cet algorithme est d'une utilisation beaucoup moins large que le premier (il n'est utile que dans certains cas pour le dépliage déterministe), mais il illustre les possibilités de typage des règles à partir d'une spécification. On peut ainsi réellement *isoler* certains cas d'applications des règles pour lesquelles on va pouvoir écrire un algorithme beaucoup plus performant que l'algorithme général.

3.6.3 Un exemple

L'exemple suivant est développé à partir du modèle d'interprétation abstraite proposé par C. Lecoutre [Lec94]. Son analyse des schémas d'appels et de solutions des prédicats est basée sur une résolution OLDT abstraite. Il calcule également les schémas pertinents pour obtenir certaines propriétés comme le déterminisme (cf. [LDL92a]). Ainsi, dans l'exemple suivant, nous avons deux

tableaux : celui des schémas d'appels et de solutions associés qui donne des informations de typage sur les atomes pendant l'exécution, et celui des conditions nécessaires sur les atomes pour qu'ils aient un comportement déterministe. La figure 3.2 représente la structure OLDT créée pour l'analyse du programme `mul`. Les nœuds encadrés sont les nœuds actifs, pour lesquels on effectue une étape de dérivation "classique" (flèches en trait continu), les autres nœuds sont les nœuds passifs, instances du nœud actif de même numéro, et qui se réfèrent à ce nœud actif pour leurs dérivations (flèches en pointillés).

C. Lecoutre utilise les contraintes ensemblistes comme moyen de typage sur l'univers de Herbrand. Pour simplifier les notations, l'expression X : `ent` représentera l'égalité ensembliste $(X \in \mathcal{N}) \wedge (\mathcal{N} = s(\mathcal{N}) \cup 0) \wedge ground(X)$[1] (où $\mathcal{N}$ est une variable ensembliste) qu'on peut interpréter par *"$\mathcal{N}$ est l'ensemble des termes représentants les entiers, et* `X` *est un terme clos de cet ensemble"*.

On utilise l'ordre de Dewey pour étiqueter les nœuds de l'arbre de spécification.

```
1 - add(0,X,X).
2 - add(s(X),Y,s(Z)) :- add(X,Y,Z).
3 - mul(0,X,0).
4 - mul(s(X),Y,Z) :- mul(X,Y,ZZ), add(Y,ZZ,Z).

:- mul(X,Y,Z).
```

Prédicats	Schémas d'appels	Schémas de solutions
add	`add(Y,0,Z)`	`add(0,0,0)`
		`add(s(Y'),0,s(Z'))` `Y':ent,Z':ent`
	`add(Y,I,Z) Y:ent,I:ent`	`add(0,I,I) I:ent`
		`add(Y,I,Z) Y:ent,` `I:ent,Z:ent`
mul	`mul(X,Y,Z)`	`mul(0,Y,0)`
		`mul(X,Y,Z) X:ent,` `Y:ent,Z:ent`

Schémas d'appels déterministes	
add	`add(Y,0,Z) Y:ent`
	`add(Y,0,Z) Z:ent`
	`add(X,I,Z) X:ent,I:ent`
mul	`mul(X,Y,Z) X:ent,Y:ent`
	`mul(X,Y,Z) X:ent,Z:ent`

[1] l'information de clôture est une extension de l'information de type proposée dans [Lec94]

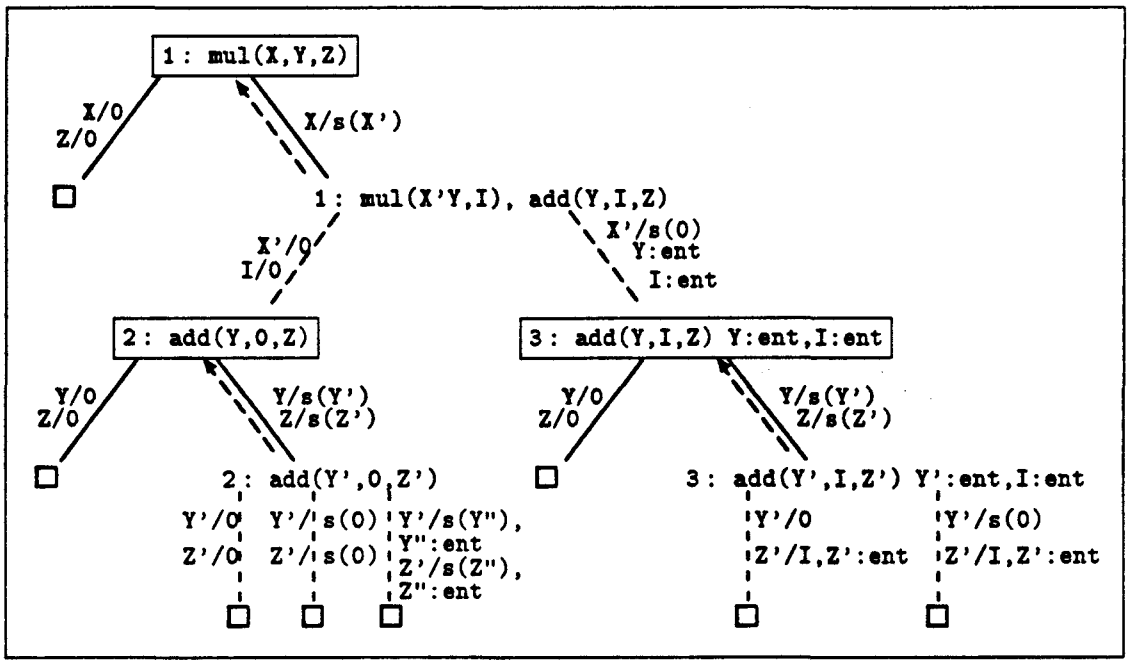


FIG. 3.2 - : Arbre de résolution OLDT pour le but `mul(X,Y,Z)`

On procède maintenant à l'étape de spécification :

1. `({1,X=0}` `,add(0,X,X),` `{11,X=0}`
   `+{2,X:ent}` `+{21,X:ent}).`
2. `({1,Y=0}` `,add(s(X),Y,s(Z)),` `{111,Y=0,`
   `+{2,X:ent,Y:ent}` `X=0,Z=0}`
   `+{112,Y=0,`
   `X/s(X'),X':ent,`
   `Z/s(Z'),Z':ent}`
   `+{211,X=0,`
   `Y:ent,Y=Z}`
   `+{212,X:ent,`
   `Y:ent,Z:ent})`

   `:-({11,Y=0}` `,add(X,Y,Z),` `{111,Y=0,`
   `+{21,X:ent,Y:ent}` `X=0,Z=0}`
   `+{112,Y=0,`
   `X/s(X'),X':ent,`
   `Z/s(Z'),Z':ent}`
   `+{211,X=0,`
   `Y:ent,Y=Z}`
   `+{212,X:ent,`
   `Y:ent,Z:ent}).`

3. `({1,⊤}` `,mul(0,X,0),` `{11,⊤}`
   `+{12,X:ent}).`

```
4.  ({1,⊤}                ,mul(s(X),Y,Z),  {...})
    :-({1,⊤}              ,mul(X,Y,ZZ),    {111,X=0,ZZ=0}   (1)
                                           +{112,X:ent,     (2)
                                           Y:ent,ZZ:ent}),
    ({1111,(1)}
    +{1112,(1),Y:ent}
    +{1121,(2)}
    +{1122,(2),ZZ=0}      ,add(Y,ZZ,Z),    {...}).
```

On va maintenant s'intéresser plus précisément à la règle 4. Les schémas 111 et 112 de cette règle et leurs extensions sont donc repérés respectivement par (1) et (2). En regardant la table des schémas d'appels déterministes (s.a.d.), on s'aperçoit que trois sur quatre des schémas d'appels possibles pour add dans la clause 4 sont déterministes. Pourtant, une simple partition de la clause ne suffirait pas, puisqu'il n'y a pas de schémas d'appels sur la tête de la règle. Il faut donc dégager une contrainte commune aux s.a.d., portant sur les variables de la tête de règle, et la remonter dans la clause. On applique l'algorithme de spécialisation d'un littéral dans une clause par rapport au déterminisme.

La contrainte nécessaire aux s.a.d. et au pré-schéma (1) est: {Y:ent}. Donc, le résultat de la première étape est (pour la clause 4, les autres restant inchangées):

```
4.1 ({1,⊤}                ,mul(s(X),Y,Z),  {...})
    :-                    ¬(Y:ent) |
    ({1,⊤}                ,mul(X,Y,ZZ),    {111,(1)}
                                           +{112,(2)}),
    ({1111,(1)}
    +{1112,(1),Y:ent}
    +{1121,(2)}
    +{1122,(2),ZZ=0}      ,add(Y,ZZ,Z),    {...}).
4.2 ({1,Y:ent}            ,mul(s(X),Y,Z),  {...})
    :-                    Y:ent |
    ({1,Y:ent}            ,mul(X,Y,ZZ),    {111,(1),Y:ent}
                                           +{112,(2)}),
    ({1112,(1),Y:ent}
    +{1121,(2)}
    +{1122,(2),ZZ=0}      ,add(Y,ZZ,Z),    {...}).
```

Tous les schémas d'appels de add dans la clause 4.2. sont déterministes, mais il y a plusieurs clauses candidates pour permettre la dépliage déterministe: il faut affiner les conditions dans le pré-schéma de C pour pouvoir effectuer un dépliage déterministe.

Deuxième étape avant le dépliage: la déterminisation d'une clause. Le $L_{litteral}$ considéré est toujours add(Y,ZZ,Z).

Schémas d'Appels → mgus ↓	Y:ent
Y=0,ZZ=Z	Y=0
Y=s(X), Z=s(Z')	Y=s(X)

On peut déterminiser complètement la clause 4.2.

```
4.1    ({1,⊤}                      ,mul(s(X),Y,Z),     {...})
       :-                          ¬(Y:ent) |
       ({1,⊤}                      ,mul(X,Y,ZZ),       {111,(1)}
                                                       +{112,(2)}),

       ({1111,(1)}
       +{1112,(1),Y:ent}
       +{1121,(2)}
       +{1122,(2),ZZ=0}            ,add(Y,ZZ,Z),       {...}).
4.2.1  ({1,Y=0}                    ,mul(s(X),Y,Z),     {...})
       :-                          Y:ent |
       ({1,Y=0}                    ,mul(X,0,ZZ),       {111,(1),Y=0}
                                                       +{112,(2),Y=0}),

       ({1112,(1),Y=0}
       +{1121,(2),Y=0}
       +{1122,(2),ZZ=0,Y=0}        ,add(0,ZZ,Z),       {...}).
4.2.2  ({1,Y/s(Y'),Y':ent}         ,mul(s(X),Y,Z),     {...})
       :-                          Y:ent |
       ({1,Y/s(Y'),Y':ent}         ,mul(X,s(Y'),ZZ),   {111,(1),
                                                       Y/s(Y'),Y':ent},
                                                       +{112,(2)}),

       ({1112,(1),
       Y/s(Y'),Y':ent}
       +{1121,(2),
       Y/s(Y'),Y':ent}
       +{1122,(2),ZZ=0,
       Y/s(Y'),Y':ent}             ,add(s(Y'),ZZ,Z),   {...}).
```

On peut maintenant effectuer le dépliage déterministe:

```
4.1    ({1,⊤}                  ,mul(s(X),Y,Z),      {...})
       :-                      ¬(Y:ent) |
       ({1,⊤}                  ,mul(X,Y,ZZ),        {111,(1)}
                                                    +{112,(2)}),
       ({1111,(1)}
       +{1112,(1),Y:ent}
       +{1121,(2)}
       +{1122,(2),ZZ=0}        ,add(Y,ZZ,Z),        {...}).
4.2.1  ({1,Y=0}                ,mul(s(X),Y,Z),      {...})
       :-                      Y=0 |
       ({1,Y=0}                ,mul(X,0,ZZ),        {...}).
4.2.2  ({1,Y/s(Y'),Y':ent}     ,mul(s(X),Y,s(Z')),  {...})
       :-                      Y=s(Y'),Y':ent |
       ({1,Y/s(Y'),Y':ent}     ,mul(X,s(Y'),ZZ),    {111,(1),
                                                    Y/s(Y'),Y':ent},
                                                    +{112,(2)}),
       ({1112,(1),
       Y/s(Y'),Y':ent}
       +{1121,(2),
       Y/s(Y'),Y':ent}
       +{1122,(2),ZZ=0,
       Y/s(Y'),Y':ent}         ,add(Y',ZZ,Z'),      {...}).
```

Le programme nettoyé est le programme écrit en *CLP($\mathcal{FT}$ + $\mathcal{SC}$), Finite Trees and Set Constraints* en suivant les notations de [Lec94], que nous présentons maintenant :

```
1      add(0,X,X).
2      add(s(X),Y,s(Z))    :- add(X,Y,Z).
3      mul(0,X,0).
4.1    mul(s(X),Y,Z)       :-¬(Y:ent) |
                           mul(X,Y,ZZ),add(Y,ZZ,Z).
4.2.1  mul(s(X),Y,Z)       :- Y=0 |
                           mul(X,0,ZZ).
4.2.2  mul(s(X),Y,s(Z'))   :- Y=s(Y'),Y':ent |
                           mul(X,s(Y'),ZZ),add(Y',ZZ,Z').
```

3.7 Conclusion

Nous avons proposé un système d'évaluation partielle pour Prolog complet intégrant des informations extérieures de façon générique. Ce système est basé sur la conservation de l'équivalence opérationnelle forte et utilise les outils présentés au chapitre 1. Nous n'avons décrit aucune heuristique pour transformer les programmes logiques, mais nous pensons que notre contribution a sa place

non pas pour des cas d'écoles, mais pour des programmes Prolog réels.

Un des (nombreux) aspects à développer sur notre "plate-forme" est l'indexation des clauses : comment sélectionner les clauses candidates à l'unification avec un but, rapidement et en utilisant les gardes?

Chapitre 4

Méta-interprétation et transformations de programmes

Les méta-programmes sont les programmes qui ont pour donnée un autre programme. Cela comprend les compilateurs, les débogueurs, les éditeurs, les évaluateurs partiels, les interpréteurs, ...La méta-programmation, par son large champ d'applications, est donc le sujet de beaucoup de travaux, aussi bien pratiques que théoriques.

En programmation logique (cf. [AR89, Bru90]), la méta-programmation est naturelle, puisque les données et les programmes sont de même nature : ce sont des termes Prolog. Néanmoins, la gestion des deux langages, le langage *objet* et le langage *méta*, i.e. le langage du programme objet (ou cible) et le langage du méta-programme, ainsi que la manipulation des prédicats méta-logiques de Prolog entraînent des problèmes de sémantique. Ceux-ci ont d'abord été étudiés dans [BK82], puis approfondis parallèlement dans [HL89] et dans [Sub89].

La classe des méta-programmes qui nous intéressent dans ce chapitre est la classe des méta-interpréteurs. Nous avons dit que les méta-programmes possédaient souvent des prédicats extra-logiques de manipulation des variables du langage objet. Pourtant, nous considérons les programmes logiques purs, sans prédicats prédéfinis ni prédicats méta-logiques : le langage du programme objet et du méta-programme sont identiques, leurs variables appartiennent à un même domaine. Notre but est de montrer que ces méta-interpréteurs préservent la sémantique du programme objet au sens de l'équivalence opérationnelle forte.

Nous ferons une distinction entre ce que nous appelerons par convention les *méta-programmes* et les *méta-interpréteurs* :

- un méta-programme est un programme qui code à l'intérieur d'une certaine structure un autre programme (le programme objet), qui a pour donnée un but, et qui interprète le programme objet pour ce but;

- un méta-interpréteur est un programme qui prend comme donnée un

programme et un but, et qui interprète ce programme pour ce but.

Donc, pour nous, le Vanilla interpréteur est un méta-programme, et un méta-interpréteur est un programme universel. Pour le Vanilla, nous garderons la dénomination "interpréteur", pour conserver les notations usuelles.

Dans ce chapitre, nous étudions principalement le Vanilla interpréteur et un méta-programme quasi-itératif constitué d'une clause binaire et de deux faits. En utilisant les transformations définies au chapitre 1, nous montrerons que ces deux méta-programmes sont équivalents (au sens de l'e.o.f.) au programme qu'ils interprètent. Puis nous étendrons ces résultats à un autre méta-programme, à une clause ternaire et deux faits dont un clos, et nous ferons le lien avec le problème de l'implication de clauses.

4.1 Le Vanilla méta-interpréteur

4.1.1 Le Vanilla-interpréteur et la SLD-résolution

Soit P le programme logique suivant, constitué de n règles :

$\forall i \quad 1 \leq i \leq n$

$1.i.$ $\mathtt{a}_i$ `:-` $\mathtt{b}_{i,1} \ldots \mathtt{b}_{i,n_i}$ `.`

`:-` $\mathtt{But}_1, \ldots \mathtt{But}_n$ `.`

Tous les prédicats définis dans P sont d'arité quelconques, les arguments sont omis ici pour simplifier les notations. Chaque règle a un corps constitué de 0 à n_n atomes, notés $\mathtt{b}_{i,j}$ et une tête de règle notée $\mathtt{a}_i$. Le Vanilla méta-interpréteur pour P est le méta-programme suivant :

$\forall i \quad 1 \leq i \leq n$

$1.i.$ `rule([`$\mathtt{a}_i$`,`$\mathtt{b}_{i,1} \ldots \mathtt{b}_{i,n_i}$`]).`

2. `solve([]).`
3. `solve([`$\mathtt{A}_1$`,`$\mathtt{A}_2$`|B]) :- solve([`$\mathtt{A}_1$`]), solve([`$\mathtt{A}_2$`|B]).`
4. `solve([A]) :- rule([A|B]), solve(B).`

`:- solve([`$\mathtt{But}_1, \ldots \mathtt{But}_n$`]).`

Par les transformations de programmes que nous avons présentées dans la section 1.2, nous obtenons la proposition suivante :

Proposition 19 *Tout programme logique défini (P,But) admet un programme opérationnellement fortement équivalent de la forme du Vanilla-interpréteur, de but* `:- solve([`*But*`])`*, et ce méta-programme est calculable.*

Preuve On effectue un dépliage en tête sur la clause 4 du Vanilla interpréteur :

$\forall i \quad 1 \leq i \leq n$
$1.i.$ `rule([`$\mathtt{a}_i$`,`$\mathtt{b}_{i,1}$ ... $\mathtt{b}_{i,n_i}$`]).`

2. `solve([]).`
3. `solve([A1,A2|B]) :- solve([A1]), solve([A2|B]).`

$\forall i \quad 1 \leq i \leq n$
$4.i.$ `solve([`$\mathtt{a}_i$`]) :- solve([`$\mathtt{b}_{i,1}$ ... $\mathtt{b}_{i,n_i}$`]).`
`:- solve([But1,...Butn]).`

Les règles $1.i$ ne peuvent plus être appelées pendant la résolution du but, elles peuvent donc être supprimées. On déplie maintenant les clauses $4.i$ (renumérotées $3.i$) et le but :

1. `solve([]).`
2. `solve([A1,A2|B]) :- solve([A1]), solve([A2|B]).`

Le résultat est n règles :
$\forall i \quad 1 \leq i \leq n$
$3.i.$ `solve([`$\mathtt{a}_i$`]) :- solve([`$\mathtt{b}_{i,1}$`]),solve([`$\mathtt{b}_{i,2}$ ... $\mathtt{b}_{i,n_i}$`]).`
`:- solve([But1]),...solve([Butn]).`

On effectue alors autant de dépliages déterministes par rapport à la clause 2 sur les règles $3.i$ que nécessaires. On obtient alors :

1. `solve([]).`
2. `solve([A1,A2|B]) :- solve([A1]), solve([A2|B]).`

$\forall i \quad 1 \leq i \leq n$
$3.i.$ `solve([`$\mathtt{a}_i$`]) :- solve([`$\mathtt{b}_{i,1}$`]),solve([`$\mathtt{b}_{i,2}$`]), ...solve([`$\mathtt{b}_{i,n_i}$`]).`
`:- solve([But1]),...solve([Butn]).`

Les règles 2 et 3 n'étant plus accessibles à partir du but, on peut les supprimer. Il ne reste maintenant plus qu'à supprimer le symbole de prédicat `solve` et le symbole de liste autour des $\mathtt{a}_i$ et des $\mathtt{b}_{i,j}$. Soit m le nombre de prédicats définis dans P. On effectue alors m définitions. À chaque symbole de prédicat `p` d'arité k on associe un nouveau symbole de prédicat `p'` d'arité k par la définition :

$$\mathtt{p'}(\tilde{X}) \mathtt{ :- solve([p}(\tilde{X})\mathtt{]).}$$

où $\tilde{X}$ est un k-uplet de variables toutes distinctes. Les m définitions sont alors dépliées et on obtient n nouvelles règles (les atomes $\mathtt{a}'_i$ ou $\mathtt{b}'_{i,j}$ dénotent les correspondants des atomes $\mathtt{a}_i$ ou $\mathtt{b}_{i,j}$ avec les nouveaux symboles de prédicats) :

$\forall i \quad 1 \leq i \leq n$
$1.i.$ `solve([`$\mathtt{a}_i$`]) :- solve([`$\mathtt{b}_{i,1}$`]),solve([`$\mathtt{b}_{i,2}$`]), ...solve([`$\mathtt{b}_{i,n_i}$`]).`
$2.i.$ $\mathtt{a}'_i$ `:- solve([`$\mathtt{b}_{i,1}$`]),solve([`$\mathtt{b}_{i,2}$`]), ...solve([`$\mathtt{b}_{i,n_i}$`].`
`:- solve([But1]),...solve([Butn]).`

Chaque atome `solve([`$b_{i,j}$`])` s'unifie avec le corps d'une définition (et une seule), et de plus `solve([`$b_{i,j}$`])` est une instance de ce corps. On effectue alors autant de pliage que d'atomes **solve** dans les règles $2.i$ et sur le but. Les règles définissant **solve** ne peuvent plus être appelées à partir du but, on peut les supprimer.

$\forall i \quad 1 \leq i \leq n$

$1.i.$ a'_i `:-` $b'_{i,1}$`,`$b'_{i,2}$`, ...`b'_{i,n_i}`.`

`:-` But'_1`,...`But'_n`.`

Le programme obtenu est donc bien le programme P initial à une notation près des symboles de prédicats (lesquels peuvent être retrouvés par une nouvelle étape de m définitions-pliages-dépliages). □

4.1.2 Le Vanilla-interpréteur et la SLDNF-résolution

On considère un programme logique P comme dans la section précédente, mais P est maintenant un programme logique *normal*, i.e. les $b_{i,j}$ sont des littéraux positifs ou négatifs. Le Vanilla méta-interpréteur pour P est alors le méta-programme suivant :

$\forall i \quad 1 \leq i \leq n$

$1.i.$ `rule([`a_i`,`$b_{i,1}$ `...`b_{i,n_i}`]).`

2. `solve([]).`
3. `solve([`A_1`,`A_2`|B]) :- solve([`A_1`]), solve([`A_2`|B]).`
4. `solve([A]) :- rule([A|B]), solve(B).`
5. `solve([not(A)]) :- not solve([A]).`

`:- solve([`But_1`,...`But_n`]).`

En considérant maintenant une extension de notre équivalence opérationnelle forte à une prise en compte des arbres de dérivation SLDNF (à la Prolog), nous obtenons la proposition suivante :

Proposition 20 *Tout programme logique normal (P,But) admet un programme opérationnellement fortement équivalent de la forme du Vanilla-interpréteur, de but* `:- solve([`*But*`])`*, et ce méta-programme est calculable.*

Preuve La preuve est très proche de celle de la proposition précédente. Les deux premières étapes de transformations (dépliage en tête par rapport à **rule** et dépliages déterministes des clauses $3.i$ par rapport à la clause 2) sont les mêmes. On obtient alors le méta-programme suivant :

1. `solve([]).`
2. `solve([A1,A2|B]) :- solve([A1]), solve([A2|B]).`

$\forall i \quad 1 \leq i \leq n$

3.i. `solve([ai]) :- solve([bi,1]),solve([bi,2]), ...solve([bi,ni]).`

4. `solve([not(A)]) :- not solve([A]).`
   `:- solve([But1]),...solve([Butn]).`

Avant de supprimer les symboles de prédicat `solve`, on effectue tous les dépliages déterministes possibles par rapport à la clause 4. Les règles 3.i contiennent alors des littéraux de la forme `not solve([`$b_{i,j}$`])`. La fin de la preuve est alors tout à fait similaire à la preuve précédente. L'étape de pliage ne pose pas de problème particulier pour les littéraux négatifs puisque le pliage est l'inverse d'un dépliage déterministe: le comportement du programme n'est pas changé par rapport à la résolution SLDNF (même si on considère la négation comme un prédicat à effet de bord, cf. page 72). On obtient donc le programme suivant :

$\forall i \quad 1 \leq i \leq n$

1.i. a'_i `:- b'`$_{i,1}$`,b'`$_{i,2}$`, ...b'`$_{i,n_i}$`.`
   `:- But'`$_1$`,...But'`$_n$`.`

où les `b'`$_{i,j}$ négatifs (de la forme `not p(`$\tilde{t}$`)`) correspondent aux littéraux $b_{i,j}$ négatifs. □

P.M. Hill et J.W. Lloyd ont étudié le Vanilla-interpréteur ([HL89]) dans le cadre de la SLDNF résolution et ont montré que le complété de Clark du Vanilla-interpréteur et le complété de Clark du programme *objet* (ici, du programme P) ont les mêmes substitutions-réponses, que les conséquences logiques de l'un sont les conséquences logiques de l'autre, et enfin que les deux programmes ont même ensemble d'échecs finis. Leur preuve fait appel à un typage des variables : appartenances des variables de P à la sorte o (pour *objet*), et des variables du Vanilla-interpréteur proprement dit à la sorte μ (pour *méta*); et à des calculs de point fixe de l'opérateur T_P. Ils n'utilisent l'évaluation partielle que pour une étape de la preuve de leur dernier résultat (cela correspond à notre premier dépliage en tête par rapport à `rule`).

Notre preuve par transformations syntaxiques semble plus simple. Mais les résultats sont un peu différents. Nous montrons en fait que pour n'importe quelle règle de choix fixée pour le programme objet, il existe une règle de choix pour le Vanilla (calculable par rapport à la première) qui conserve la structure de l'arbre SLD, et donc garantit un même comportement opérationnel : lorsqu'un atome A est choisi dans le programme objet, la clause 3 du Vanilla est utilisée jusqu'à ce que l'atome `solve([`A`])` apparaisse dans le but, et alors il est sélectionné. Mais quelle que soit cette règle, la règle de choix du Vanilla favorise toujours un atome de prédicat `rule` par rapport aux autres. Notre résultat est donc valide par rapport à n'importe quelle règle de choix fixée sur

le programme objet, mais la réciproque est fausse: on peut trouver une règle de choix pour le Vanilla qui le fait toujours boucler (dès que la règle 4 peut être appliquée, sélectionner le nouvel atome de prédicat **solve**). Le résultat de Hill et Lloyd, plus général, prouve l'équivalence des conséquences logiques du complété du programme objet et du complété du Vanilla pour ce programme, pour toute règle de choix.

4.2 Un méta-interpréteur quasi-itératif

Soit le méta-programme suivant:

```
1. choix([But|ListeDeButs],[[But|CorpsDeRegle]-ListeDeButs|_],
     CorpsDeRegle,Programme).
2. choix(ListeDeButs,[_|FinDuProgramme],
     ListeDeButs,FinDuProgramme).
3. execute([],[]).
4. execute(But1,Programme1):-
          choix(But1,Programme1,But2,Programme2),
          execute(But2,Programme2).

:- execute([But1,...Butn],Programme).
```

où Programme est une liste composée de n règles: Programme = [$Règle_1$, $Règle_2$, ... $Règle_n$], et, $\forall i$, $1 \leq i \leq n$, $Règle_i$ est une différence de listes: $Règle_i$= $[a_i, b_{i,1}, b_{i,2}, \ldots b_{i,n_i}|L]$-L. Programme ne partage jamais de variable avec le reste de la clause (ou du but) où il apparaît.

Le prédicat **choix** est le prédicat qui détermine la partie contrôle sur la résolution. On peut imaginer qu'on a d'une part la liste de buts à prouver, le premier élément de la liste étant le but sélectionné, et d'autre part le programme *objet* (ou *cible*) avec un pointeur balayant le programme. La règle 1 nous dit: *"si le but courant s'unifie avec la règle pointée alors remplacer ce but par le corps de la règle et remettre le pointeur au début du programme"*. La règle 2 nous dit: *"descendre le pointeur sur le programme d'une règle"*. L'utilisation de différences de listes dans le codage du programme permet la concaténation de la liste de buts restant à traiter avec le corps de la règle sans utiliser le prédicat append. Si le but courant est de la forme $[a_i|$ListeDeButs$]$, et le pointeur se trouve sur la règle $[a_i, b_{i,1}, b_{i,2}, \ldots b_{i,n_i}|L]$-L alors le résultat de l'unification avec la première règle donne

$$\begin{cases} \text{L = ListeDeButs} \\ \text{CorpsDeRegle = } [b_{i,1}, b_{i,2}, \ldots b_{i,n_i}|\text{ListeDeButs}] \end{cases}$$

Nous allons maintenant montrer que ce méta-programme, avec une règle ternaire et trois faits, est équivalent, au sens de l'équivalence opérationnelle

forte, au programme original qu'il interprète (programme codé dans Programme).

Exemple 34
Pour le programme *append*:

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

   :- append(L1,L2,[1,2,3]).
```

le méta-programme complet est

```
1. choix([But|ListeDeButs],[[But|CorpsDeRegle]-ListeDeButs|_],
     CorpsDeRegle,
     [[append([],Lapp1,Lapp1)|L]-L,
     [append([X|Lapp2],Lapp3,[X|Lapp4]),
      append(Lapp2,Lapp3,Lapp4)|LL]-LL]).
2. choix(ListeDeButs,[_|FinDuProgramme],
     ListeDeButs,FinDuProgramme).
3. execute([],[]).
4. execute(But1, Programme1) :-
          choice(But1,Programme1,But2,Programme2),
          execute(List_of_Goals_2, List_of_Rules_2).

   :- execute([append([1],[2,3],L1)]-[],
          [[append([],Lapp1,Lapp1)|L]-L,
          [append([X|Lapp2],Lapp3,[X|Lapp4]),
           append(Lapp2,Lapp3,Lapp4)|LL]-LL]).
```

Proposition 21 *Tout programme logique défini (P,But) admet un méta-programme opérationnellement fortement équivalent constitué d'une règle ternaire et de trois faits.*

Preuve On effectue un dépliage en tête sur la clause 4 du méta-interpréteur.

```
1. choix([But|ListeDeButs],[[But|CorpsDeRegle]-ListeDeButs|_],
     CorpsDeRegle,Programme).
2. choix(ListeDeButs,[[_|FinDuProgramme],
     ListeDeButs,FinDuProgramme).
3. execute([],[]).
4. execute([But|ListeDeButs],
     [[But|CorpsDeRegle]-ListeDeButs|_]):-
           execute(CorpsDeRegle,Programme).
5. execute(ListeDeButs,[_|FinDuProgramme]):-
           execute(ListeDeButs,FinDuProgramme).

   :- execute([But_1,...But_n],Programme).
```

Prog$_1$

Les clauses de prédicat **choix** n'appartiennent plus au graphe de dépendance du prédicat **execute** : elles peuvent donc être éliminées du programme par rapport au but `:- execute([But`$_1$`,...But`$_n$`],Programme)`. Les clauses de **execute** sont alors renumérotées (3, 4 et 5 vers 1, 2 et 3) et on effectue un dépliage en tête sur la dernière clause :

```
1. execute([],[]).
2. execute([But|ListeDeButs],
     [[But|CorpsDeRegle]-ListeDeButs|_]):-
           execute(CorpsDeRegle,Programme).
3. execute([],[_]).
4. execute([But|ListeDeButs],
     [_,[But|CorpsDeRegle]-ListeDeButs|_]):-
           execute(CorpsDeRegle,Programme).
5. execute(ListeDeButs,[_,_|FinDuProgramme]):-
           execute(ListeDeButs,FinDuProgramme).

   :- execute([But_1,...But_n],Programme).
```

Prog$_2$

Les arguments de l'atome du corps de la dernière règle du programme étant des variables libres, un dépliage en tête de cet atome se fait par rapport à toutes les règles du programme courant. On veut déplier le programme jusqu'à obtenir dans une règle binaire et dans un fait (`execute([],[_,...])`) un second argument qui soit une liste d'au moins n éléments. On doit donc arriver à un programme contenant $2(n+1)$ règles.

À chaque dépliage en tête sur l'atome du corps de la dernière règle, le programme passe de k règles à $2k-1$ règles. Soit $u_{i,i\geq 0}$ la suite qui représente la taille (en nombre de règles) des programmes obtenus après chaque dépliage. On a :

$u_0 = 3$
$u_i = 2u_{i-1} - 1$

c'est-à-dire $u_i = 3.2^i + 1 - 2^i = 2^{i+1} + 1$, où i représentant le numéro du programme représente aussi le nombre de dépliages en tête à effectuer. Pour obtenir notre programme, il faut donc effectuer i dépliages avec i le plus petit entier qui vérifie $i \geq \log_2(2(n+1) - 1) - 1$ (n est le nombre de règles dans `Programme` et $2(n+1)$ est le nombre de règles qu'on veut pour le méta-programme). On obtient alors :

```
1.      execute([],[]).
2.      execute([But|ListeDeButs],
          [[But|CorpsDeRegle]-ListeDeButs|_]):-
                execute(CorpsDeRegle,Programme).
3.      execute([],[_]).

4.      execute([But|ListeDeButs],
          [_,[But|CorpsDeRegle]-ListeDeButs|_]):-
                execute(CorpsDeRegle,Programme).
5.      execute([],[_,_]).
⋮       ⋮
2n.     execute([But|ListeDeButs],
          [_,...,_,[But|CorpsDeRegle]-ListeDeButs|_]):-
                execute(CorpsDeRegle,Programme).        Prog3
        /*   la règle 2n contient n-1 variables anonymes  */
        /*          avant [But|CorpsDeRegle]              */

2n+1.   execute([],[_,...,_]).
        /*   la règle 2n+1 contient n variables anonymes  */
        /*          dans le deuxième argument             */
⋮       ⋮
2^i+1+1. execute(ListeDeButs,[_,...,_|FinDuProgramme]):-
           execute(ListeDeButs,FinDuProgramme).
        /* la règle 2^i+1+1 contient 2n variables anonymes */
        /*          dans le deuxième argument             */

        :- execute([But_1,...But_n],Programme).
```

On introduit maintenant dans le programme une nouvelle définition :
`exe(CorpsDeRegle) :- execute(CorpsDeRegle,Programme).`
On effectue alors $n+1$ pliages dans le programme :

```
1.      execute([],[]).
2.      execute([But|ListeDeButs],
          [[But|CorpsDeRegle]-ListeDeButs|_]):-
                exe(CorpsDeRegle).
3.      execute([],[_]).

4.      execute([But|ListeDeButs],
          [_,[But|CorpsDeRegle]-ListeDeButs|_]):-
                exe(CorpsDeRegle).
5.      execute([],[_,_]).
⋮       ⋮
2n.     execute([But|ListeDeButs],
          [_,...,_,[But|CorpsDeRegle]-ListeDeButs|_]):-
                exe(CorpsDeRegle).
        /*  la règle 2n contient n-1 variables anonymes  */
        /*          avant [But|CorpsDeRegle]             */

2n+1.   execute([],[_,...]).
        /*  la règle 2n+1 contient n variables anonymes  */
        /*          dans le deuxième argument            */
⋮       ⋮
2^i+1+1. execute(ListeDeButs,[_,...|FinDuProgramme]):-
           execute(ListeDeButs,FinDuProgramme).
        /* la règle 2^i+1+1 contient 2n variables anonymes */
        /*          dans le deuxième argument            */

2^i+1+2. exe(CorpsDeRegle):-
                execute(CorpsDeRegle,Programme).

      :- exe([But1,...Butn]).
```

$Prog_4$

Puis on effectue un dépliage sur la définition de **exe** (il n'y a que $n+1$ règles dépliantes). Les définitions de **execute** ne sont alors plus accessibles à partir du but et on peut les enlever. Notons M_1 le résultat de toutes nos transformations :

```
∀i, 1 ≤ i ≤ n.  exe([a_i|ListeDeButs]):-
                      exe([b_i,1,...b_i,n_i|ListeDeButs]).
n+1.            exe([]).

                :- exe([But1,...Butn]).
```

$Prog_5$

On revient maintenant au Vanilla-interpréteur :

$\forall i \quad 1 \leq i \leq n$
$1.i.$ `rule([`a_i`,`$b_{i,1}$ ... `]).`

2. `solve([]).`
3. `solve([`A_1`,`A_2`|B]) :- solve([`A_1`]),solve([`A_2`|B]).`
4. `solve([A]) :- rule([A|B]),solve(B).`
   `:- solve([`But_1`,...`But_n`]).`

Un dépliage en tête sur la règle 3 :

$\forall i \quad 1 \leq i \leq n$
$1.i.$ `rule([`a_i`,`$b_{i,1}$ ... `]).`

2. `solve([]).`
3. `solve([`A_1`,`A_2`|B]) :- rule([`A_1`|C]),solve(C),solve([`A_2`|B]).`
4. `solve([A]) :- rule([A|B]),solve(B).`
   `:- solve([`But_1`,...`But_n`]).`

Un autre dépliage en tête sur cette même clause :

$\forall i \quad 1 \leq i \leq n$
$1.i.$ `rule([`a_i`,`$b_{i,1}$ ... `]).`

2. `solve([]).`
$\forall i \quad 1 \leq i \leq n$
$3.i.$ `solve([`a_i`,`A_2`|B]) :- solve([`$b_{i,1}$`,...`b_{i,n_i}`]),solve([`A_2`|B]).`
4. `solve([A]) :- rule([A|B]), solve(B).`
   `:- solve([`But_1`,...`But_n`]).`

On effectue maintenant un dépliage en tête sur la clause 4. Les règles $1.i.$ peuvent être supprimées : le prédicat `rule` n'appartient plus au graphe de dépendance de `solve`. Le méta-programme simplifié (noté M_2) devient donc :

1. `solve([]).`
$\forall i \quad 1 \leq i \leq n$
$2.i.$ `solve([`a_i`,`A_2`|B]) :- solve([`$b_{i,1}$`,...`b_{i,n_i}`]),solve([`A_2`|B]).`
$\forall i \quad 1 \leq i \leq n$
$3.i.$ `solve([`a_i`]) :- solve([`$b_{i,1}$`,...`b_{i,n_i}`]).`

`:- solve([`But_1`,...`But_n`]).`

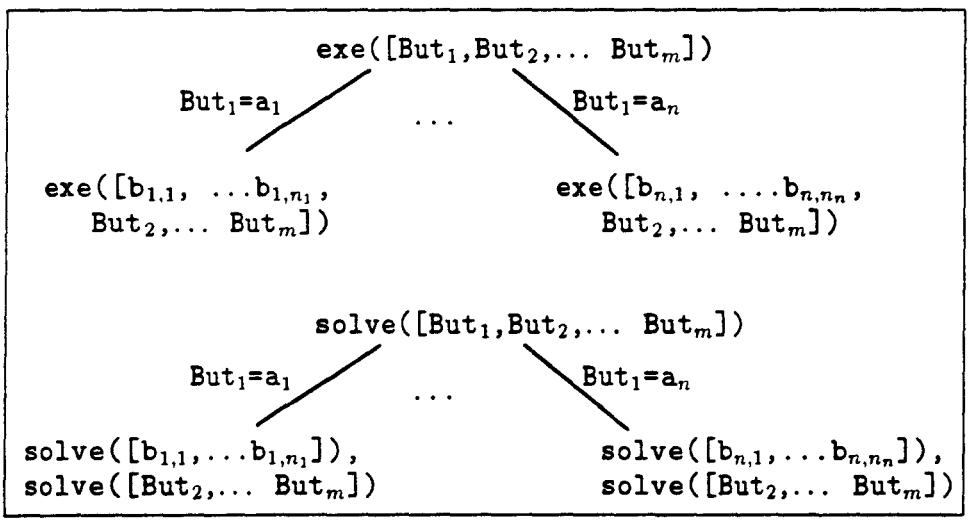
Nous allons maintenant montrer que M_1 et M_2 sont équivalents (au sens de l'e.o.f.) en comparant leurs arbres de résolution. Nous ne nous intéresserons qu'à la forme de l'arbre et aux systèmes d'équations étiquetant les branches (nous les appellerons les arbres des systèmes). Si les substitutions-réponses sont les mêmes dans les deux arbres et données dans le même ordre avant la

première branche infinie, alors on peut conclure que le méta-programme quasi-itératif préserve l'équivalence opérationnelle forte par rapport au programme qu'il interprète.

Remarque: les buts pour M_1 et M_2 sont :- `exe([But`$_1$`,...But`$_n$`])` et :- `solve([But`$_1$`,...But`$_n$`])`. Le fait que la taille de leur liste soit fixée est impératif pour avoir l'équivalence opérationnelle forte entre les deux méta-programmes, mais ce n'est pas une contrainte forte du point de vue de leur utilisation pour la méta-interprétation.

- Hypothèse de départ: jusqu'à une profondeur 1, les arbres des systèmes des deux méta-programmes M_1 et M_2 sont égaux, et les buts qui étiquettent les nœuds ont la propriété suivante: la concaténation des listes arguments d'un nœud de M_2 est égale à la liste argument du nœud correspondant de M_1.

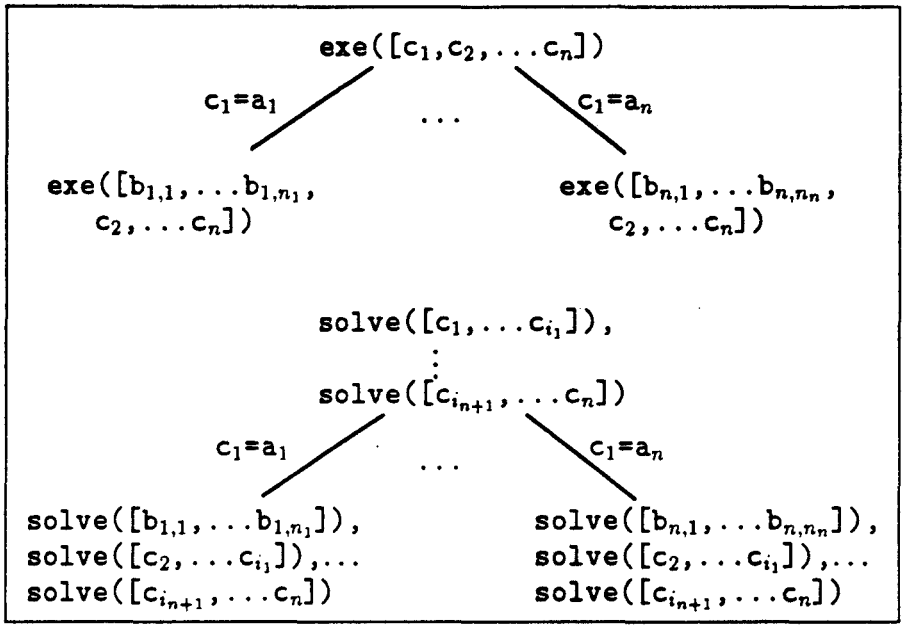
  On compare les dérivations à partir du but :-`exe([But`$_1$`,But`$_2$`,...But`$_m$`])` pour M_1 et du but :-`solve([But`$_1$`,But`$_2$`,...But`$_m$`])` pour M_2.



- Hypothèse de récurrence: les deux arbres des systèmes sont égaux jusqu'à une profondeur k, et leurs nœuds vérifient l'hypothèse de départ. On énumère les quatre cas possibles pour des branches non finies à la profondeur k:

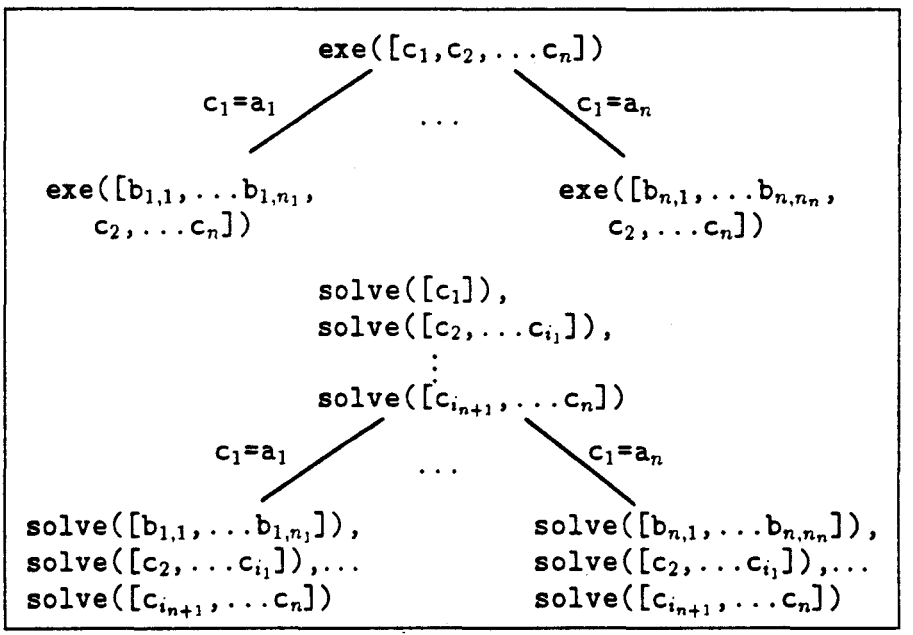
  1. certains nœuds du programme M_1 sont de la forme `exe([c`$_1$`,...c`$_n$`])` et les nœuds correspondants dans M_2 sont de la forme `solve([c`$_1$`, ...c`$_{i_1}$`])`, `solve([c`$_{i_1+1}$`, ...c`$_{i_2}$`])`,...`solve([c`$_{i_n+1}$`, ...c`$_n$`])` tels

que $concat([c_1, \ldots c_{i_1}], [c_{i_1+1}, \ldots c_{i_2}], \ldots [c_{i_{n+1}}, \ldots c_n]) = [c_1, \ldots c_n]$.



Les deux hypothèses sont toujours vérifiées à l'étape $k+1$.

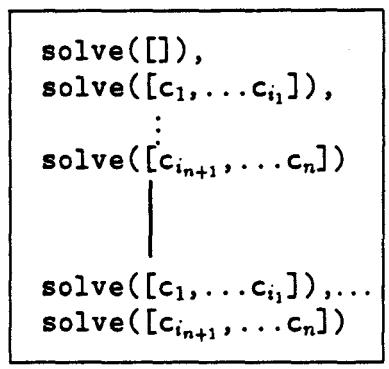
2. certains nœuds du programme M_1 sont de la forme `exe([c1,... cn])` et les nœuds correspondants dans M_2 sont de la forme `solve([c1])`, `solve([c2, ... cn])`.



Les deux hypothèses sont toujours vérifiées à l'étape $k+1$.

3. certains nœuds du programme M_1 sont de la forme `exe([c1,... cn])` et cette fois-ci les nœuds correspondants dans M_2 sont de la forme

`solve([])`, `solve([`c_1, ... c_n`])`.



Il suffit d'un pas de dérivation sur M_2 pour se retrouver dans le cas 1 sans que les systèmes d'équations soient modifiés.

4. enfin les autres nœuds de M_1 sont `execute([])`, et les nœuds correspondants dans M_2 sont `solve([])`. Dans les deux cas, on arrive en une étape de dérivation à un nœud-solution.

□

Dans le méta-programme que nous proposons, lorsque la clause vide est générée, (premier argument de `execute` vaut `[]`), il faut "dérouler" tout le programme par la deuxième clause avant d'obtenir un succès par unification avec le fait. Il peut être légèrement amélioré, en remplacant le fait `execute([],[])` par `execute([],ProgrammeClos)`, où `ProgrammeClos` est n'importe quelle instance close de `Programme`. La preuve de l'équivalence avec cette variante du méta-programme peut se faire en réutilisant la preuve de la proposition 21. En effectuant les mêmes étapes de transformations, on obtient des programmes similaires à $Prog_1$, $Prog_2$, $Prog_3$ et $Prog_4$, sauf pour les faits, qui sont de la forme `execute([],ProgrammeClos)`, `execute([],[_|ProgrammeClos])`, ... `execute([],[_,...,_|ProgrammeClos])`. L'égalité syntaxique des deux méta-programmes est obtenue au $Prog_5$.

Comparons maintenant la complexité du programme original et la complexité de ce programme codé dans notre méta-programme quasi-itératif. On reprend les notations employées dans la preuve de la proposition 3 page 27. Appelons *complexité* d'un nœud-solution sol_n le nombre de nœuds de l'arbre SLD qu'on parcourt (par la stratégie en profondeur d'abord et de gauche à droite) avant d'atteindre sol_n. On notera φ_i la fonction de complexité du programme i, et N_i le nombre de règles qui le définissent. i prendra les valeurs o pour le programme objet, et m pour le méta-programme.

Au mieux, le but s'unifie avec la première règle du programme original, et cette règle est un fait. Au pire, il s'unifie avec la dernière règle du programme. Dans le cas où le fait du méta programme est `execute([],[])`, on obtient donc :

$$\varphi_o(sol_n) + N_o \leq \varphi_m(sol_n) \leq (\varphi_o(sol_n) \times N_o) + N_o$$

et dans le cas où le fait du méta-programme est `execute([],ProgrammeClos)`, on obtient :

$$\varphi_o(sol_n) \leq \varphi_m(sol_n) \leq \varphi_o(sol_n) \times N_o$$

4.3 Le résultat de [DLRW94] sur la puissance de calcul d'une clause binaire

Ph. Devienne, P. Lebègue et J.-C. Routier se sont intéressés à l'étude de la classe des programmes constitués d'une clause de Horn binaire, d'un but et d'un fait. Ils ont montré l'indécidabilité de l'arrêt et de l'existence de solutions pour un programme particulier de cette classe ([DLR93b, DLR93a, Rou94]). Leur preuve utilise la correspondance entre les machines de Minsky (un formalisme équivalent aux machines de Turing) et les fonctions de Conway, et leur permet de coder une fonction de Conway dans un programme de cette classe.

Ph. Hanschke et J. Würtz ont prouvé, indépendamment et à la même période, le même résultat d'indécidabilité du vide. Leur preuve consiste en un codage du problème de correspondance de Post, en utilisant les différences de listes. Pour Σ un alphabet, un système de correspondance de Post est un ensemble non vide $\mathcal{S} = \{(g_i, d_i) \mid i \in [1, \ldots m]\}$ où les g_i et les d_i sont des mots sur Σ. Il existe une solution à ce système s'il existe une séquence non vide de n indices i_j telle que $\forall j,\ j \in [1..n], 1 \leq i_j \leq m$ et $g_{i_1} \ldots g_{i_n} = d_{i_1} \ldots d_{i_n}$. Ce problème est en général indécidable à partir du moment où Σ contient au moins deux symboles. Leur codage permet de générer toutes les combinaisons de séquences de g_i et de d_i : toutes les combinaisons de taille 1, puis toutes les combinaisons de taille 2, de taille 3, ... et ainsi de suite. Les séquences de g_i sont stockées dans une liste, les séquences de d_i dans une autre. Les différences de listes permettent de faire grossir la *même* liste sans utiliser de *append*:

$$\begin{cases} p([X|\_] - \_, [X|\_] - \_) \leftarrow \ . \\ p([S_1|G] - [[g_1|S_1], \ldots [g_m|S_1]|L_1], [S_2|D] - [[d_1|S_2], \ldots [d_m|S_2]|L_2]) \\ \qquad \leftarrow p(G - L_1, D - L_2) \ . \\ \leftarrow p([[g_1] \ldots [g_m]|L_1] - L_1, [[d_1] \ldots [d_m]|L_2] - L_2) \ . \end{cases}$$

En combinant les deux techniques de preuves et notre méta-interpréteur, Ph. Devienne, P. Lebègue, J.-C. Routier et J. Würtz ont prouvé qu'*il existe un programme à une clause de Horn binaire et deux clauses unaires qui a la même puissance de calcul que les machines de Turing* [DLRW94, Rou94].

En généralisant, le codage de Würtz et Hanschke permet de générer tous les mots sur un alphabet donné dans le but. En particulier, si l'alphabet est fixé à $\mathcal{A} = (droite_1, gauche_1)$ et $\mathcal{B} = (droite_2, gauche_2)$, le programme va générer toutes les combinaisons possibles de $\mathcal{A}$ et de $\mathcal{B}$. Si maintenant on complique un peu le programme (notons le $\mathcal{M}_1$) :

$$\begin{cases} p([X|\_] - \neg X) \leftarrow \ . \\ p([UneDeriv|LesAutresDerivs] - [[\mathcal{A}|UneDeriv], [\mathcal{B}|UneDeriv]|L_1], [H|L_2]) \\ \qquad\qquad \leftarrow p(LesAutresDerivs - L_1, [H, X, X|L_2]) \ . \\ \leftarrow p([[but|L]|L_1] - L_1, [fait|L_2]) \ . \end{cases}$$

On a obtenu une simulation d'un parcours en largeur d'abord de l'arbre de résolution SLD du programme suivant (notons le Π):

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche_1) \leftarrow p(droite_1) \ . & (= \mathcal{A}) \\ p(gauche_2) \leftarrow p(droite_2) \ . & (= \mathcal{B}) \\ \leftarrow p(but) \ . \end{cases}$$

$\mathcal{M}_1$ génère toutes les dérivations de longueur 1, puis de longueur 2, etc ... L'unification avec le fait de $\mathcal{M}_1$ est possible lorsque le premier argument commençant par une dérivation de la forme *[droite$_k$, gauche$_k$, droite$_{k-1}$, gauche$_{k-1}$, ... droite$_1$, gauche$_1$, but|L]* s'unifiera avec *[fait, X_k, X_k, ... X_1, X_1|LL]*. $\mathcal{M}_1$ donnera bien toutes les solutions de Π dans l'ordre d'un parcours en largeur, mais ne s'arrête pas. La première étape de la preuve est faite: Π a exactement la forme de notre méta-interpréteur quasi-itératif (à un dépliage par rapport à `choix` près, cf. théorème 8). Si maintenant on considère que le programme codé dans notre méta-programme quasi-itératif est n'importe quel méta-interpréteur connu qui prend en argument du but un couple *(programme à interpréter,but)*, on a bien un méta-(méta-)interpréteur en largeur d'abord. La deuxième partie de la preuve consiste à construire un mécanisme qui arrêtera le méta-interpréteur si le programme objet s'arrête. Ce mécanisme est construit à l'aide des fonctions de Conway et s'arrête un nombre d'étapes fini après l'obtention de la dernière solution du programme objet par le méta-interpréteur $\mathcal{M}_1$ si le programme objet s'arrête. La complexité de ce programme universel est au moins exponentielle par rapport à la complexité du programme original. $\mathcal{M}_1$ simule un parcours en largeur d'abord de l'arbre SLD du programme qu'il interprète, dans notre cas le méta-programme quasi-itératif. Sa complexité notée φ_u est majorée par (on reprend les notations précédentes):

$$\varphi_u(sol_n) \leq N_m^{\varphi_m(sol_n)} \leq 2^{\varphi_o(sol_n) \times N_o}$$

φ_u représente bien la complexité du programme universel pour l'obtention des solutions. Pour l'arrêt, la borne du nombre de nœuds à parcourir est plus grande, puisqu'on ajoute la complexité du mécanisme d'arrêt lié aux fonctions de Conway.

Théorème 6 *[DLRW94] Il existe un méta-interpréteur pour les langages à clauses de Horn sous la forme d'un programme avec une seule clause de Horn*

binaire, un fait et un but, lequel, prenant en entrée un programme à clause de Horn P, a exactement les mêmes solutions que P et s'arrête si et seulement si P s'arrête.

Théorème 7 *[DLRW94] La classe des programmes à une clause de Horn binaire et deux clauses unaires a la même puissance d'expression que les machines de Turing.*

4.4 Autour du méta-interpréteur quasi-itératif

En transformant le méta-programme quasi-itératif, on peut obtenir des résultats d'indécidabilité de l'arrêt et du vide pour quelques classes simples de programmes.

Théorème 8 *Il existe un existe un programme explicitement constructible de la forme*

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche_1) \leftarrow p(droite_1) \ . \\ p(gauche_2) \leftarrow p(droite_2) \ . \end{cases}$$

où fait est clos, tel qu'il n'existe pas d'algorithme qui décide en un temps fini si, pour un but donné clos $\leftarrow p(but)$, la résolution SLD s'arrête et s'il existe des substitutions-réponses.

De plus, ce programme est un programme universel.

Preuve Reprenons le méta-programme de la proposition 21. On sait qu'il existe un programme universel écrit avec des clauses de Horn [Tär77] qui, pour un but donné clos représentant un entier, exécute la machine de Turing de numéro cet entier. On choisit alors de coder dans `Programme` le programme universel. Mais le but du méta-programme de la proposition 21 n'est pas clos. On le remplace alors par

```
:- execute([nil,but_clos],[[nil,but_clos]-but_clos])
```

où `but_clos` est le numéro de la machine de Turing choisie par l'utilisateur. En faisant un dépliage sur l'atome `choix`, on obtient alors le méta-interpréteur suivant :

```
1. execute([],[]).
2. execute([But|ListeDeButs],
     [[But|CorpsDeRegle]-ListeDeButs|_]):-
           execute(CorpsDeRegle,Programme).
3. execute(ListeDeButs,[_|FinDuProgramme]):-
           execute(ListeDeButs,FinDuProgramme).

:- execute([nil,but_clos],[[nil,but_clos]-but_clos]).
```

□

Corollaire 1 *Le problème de l'arrêt et du vide pour les programmes constitués de deux clauses de Horn binaires, un fait clos et un but clos, est indécidable.*

Preuve Décider si un programme s'arrête ou s'il a des solutions est en général indécidable. Puisqu'il existe un méta-interpréteur de la classe qui nous intéresse, les problèmes de l'arrêt et du vide sont indécidables pour cette classe.

□

Le problème général de l'implication de clauses, $\forall \tilde{x} A \Rightarrow \forall \tilde{y} B$ où B est une clause et A un ensemble de clauses, $\tilde{x}$ sont les variables de A et $\tilde{y}$ les variables de B, est équivalent à la non-satisfiabilité d'un programme contenant l'ensemble de clauses A et un ensemble de faits clos obtenus à partir de la négation de B, et a été montré indécidable dans le cas général ainsi que pour la classe des ensembles de clauses constitués de deux clauses binaires par Schmidt-Schauss ([SS88]). Le résultat que nous obtenons ici est donc un peu plus fort puisque nous construisons deux tels ensembles A et B. Nous reparlerons des liens entre ces problèmes dans la conclusion du chapitre.

Nous abordons maintenant la classe des programmes à une règle ternaire.

Théorème 9 *Il existe un programme explicitement constructible de la forme*

$$\begin{cases} p(fait_1) \leftarrow \ . \\ p(fait_2) \leftarrow \ . \\ p(gauche) \leftarrow p(droite_1), p(droite_2) \ . \end{cases}$$

où $fait_1$ est clos, tel qu'il n'existe pas d'algorithme qui décide en un temps fini si, pour un but donné clos $\leftarrow p(but)$, la résolution SLD s'arrête et s'il existe des substitutions-réponses.

De plus, ce programme est un programme universel.

Preuve L'idée de la preuve est de montrer que notre méta-interpréteur quasi-itératif du théorème précédent peut se transformer en un méta-interpréteur de la forme requise dans le théorème.

La preuve se déroule en deux étapes. Nous allons d'abord montrer qu'un programme de la forme

$$\begin{cases} p(t_0) \leftarrow \ . \\ p(t_1) \leftarrow p(t_2) \ . \\ p(t_3) \leftarrow p(t_4) \ . \\ \leftarrow p(t_5) \ . \end{cases}$$

est équivalent (au sens de l'e.o.f.) à un programme de la forme

$$\begin{cases} p(t_0) \leftarrow \ . \\ p(t) \leftarrow p(t_2) \ . \\ p(t) \leftarrow p(t_4) \ . \\ \leftarrow p(t_5) \ . \end{cases}$$

Puis nous montrerons que ce programme est équivalent à un programme de la forme

$$\begin{cases} p(t_0) \leftarrow \ . \\ p(t'_0) \leftarrow \ . \\ p(t) \leftarrow p(t_2), p(t_4) \ . \\ \leftarrow p(t_5) \ . \end{cases}$$

avec t_0 clos et t'_0 quelconque.

1. Considérons le programme suivant, noté M_1 :

$$\begin{cases} p(t_0, Y, Y) \leftarrow \ . \\ p(X, Y, Y) \leftarrow p(t_2, X, t_1) \ . \\ p(X, Y, Y) \leftarrow p(t_4, X, t_3) \ . \\ \leftarrow p(t_5, nil, nil) \ . \end{cases}$$

où *nil* est un terme clos.

Nous effectuons d'abord une étape d'indexation sur les têtes de règles (cf. proposition 10) :

$$\begin{cases} p(X, Y, Y) \leftarrow q(X, Y) \ . \\ q(t_0, Y) \leftarrow \ . \\ q(t_1, Y) \leftarrow p(t_2, t_1, t_1) \ . \\ q(t_3, Y) \leftarrow p(t_4, t_3, t_3) \ . \\ \leftarrow p(t_5, nil, nil) \ . \end{cases}$$

Une étape de dépliage en tête (cf. proposition 3) donne le programme suivant :

$$\begin{cases} p(X, Y, Y) \leftarrow q(X, Y) \ . \\ q(t_0, Y) \leftarrow \ . \\ q(t_1, Y) \leftarrow q(t_2, t_1) \ . \\ q(t_3, Y) \leftarrow q(t_4, t_3) \ . \\ \leftarrow q(t_5, nil) \ . \end{cases}$$

Le prédicat p n'est plus accessible à partir du but, on peut donc supprimer sa définition, et par la propriété de suppression des variables anonymes (cf. proposition 13) on obtient le programme noté M_2 :

$$\begin{cases} q(t_0) \leftarrow \ . \\ q(t_1) \leftarrow q(t_2) \ . \\ q(t_3) \leftarrow q(t_4) \ . \\ \leftarrow q(t_5) \ . \end{cases}$$

Ce programme est le méta-interpréteur quasi-itératif décrit dans le théorème 8. Il suffit de prendre :

$$\begin{cases} t_0 = (\texttt{[]},\texttt{[]}) \text{ ou } t_0 = (\texttt{[]},\texttt{ProgrammeClos}) \\ t_1 = (\texttt{[But|ListeDeButs]},\texttt{[[But|CorpsDeRegle]-ListeDeButs|\_]}) \\ t_2 = (\texttt{CorpsDeRegle},\texttt{Programme}) \\ t_3 = (\texttt{ListeDeButs},\texttt{[\_|FinDuProgramme]}) \\ t_4 = (\texttt{ListeDeButs},\texttt{FinDuProgramme}) \\ t_5 = (\texttt{[nil,but\_clos]},\texttt{[[nil,but\_clos]-but\_clos]}) \end{cases}$$

M_1 est équivalent (au sens de l'e.o.f.) à M_2 et appartient bien à la classe de programmes à laquelle nous voulions arriver. Néanmoins, avant de passer à la deuxième étape de la preuve, nous allons montrer qu'on peut clore le fait $p(t_0, Y, Y)$.

Considérons les arbres de dérivation de M_1 et M_2 (cf. figure 4.1). Les substitutions θ_i sont issues des systèmes d'équations des arcs immédiatement supérieurs, et les indices j des termes t_{i_j} servent à différencier deux termes t_i égaux à un renommage près.

On s'aperçoit que les deuxièmes et troisièmes arguments d'un but dans l'arbre SLD de M_1 correspondent aux termes du système d'équations étiquetant l'arc du niveau immédiatement supérieur dans l'arbre SLD de M_2 : les unifications sont retardées d'un niveau de profondeur dans M_1. On voit aisément que cette propriété est vrai quelque soit le niveau de profondeur auquel on se place dans les arbres de M_1 et M_2.

Nous allons maintenant nous intéresser aux nœuds qui se dérivent directement en la clause vide. Pour cela, nous considérons que t_0 = (`[]`,`ProgrammeClos`). Soit `p(tt1,tt2,tt3)` un tel nœud de l'arbre SLD de M_1. `tt1` doit pouvoir s'unifier avec t_0, et `tt2` représente le but "principal" (c.à.d. le premier argument) du nœud précédent, tandis que `tt3` représente la tête de la règle qui s'est unifiée avec `tt2`, et dont le corps est `tt1`. Puisque `tt1` $= t_0$ est satisfiable, `tt1` ne peut être qu'un renommage de t_2, et donc `tt3` ne peut être qu'un renommage de t_1. Le système d'équations suivant est donc solvable :

$$\begin{cases} t_0 = (\texttt{[]},\texttt{ProgrammeClos}) \\ t_1 = (\texttt{[But|ListeDeButs]},\texttt{[[But|CorpsDeRegle]-ListeDeButs|\_]}) \\ t_2 = (\texttt{CorpsDeRegle},\texttt{Programme}) \\ t_3 = (\texttt{ListeDeButs},\texttt{[\_|FinDuProgramme]}) \\ t_4 = (\texttt{ListeDeButs},\texttt{FinDuProgramme}) \\ t_5 = (\texttt{[nil,but\_clos]},\texttt{[[nil,but\_clos]-but\_clos]}) \\ \texttt{tt}_1 = (\texttt{[]},\texttt{ProgrammeClos}) \\ \texttt{tt}_1 = (\texttt{CorpsDeRegle},\texttt{Programme}) \\ \texttt{tt}_3 = t_1 \\ \texttt{tt}_2 = t_1 \end{cases}$$

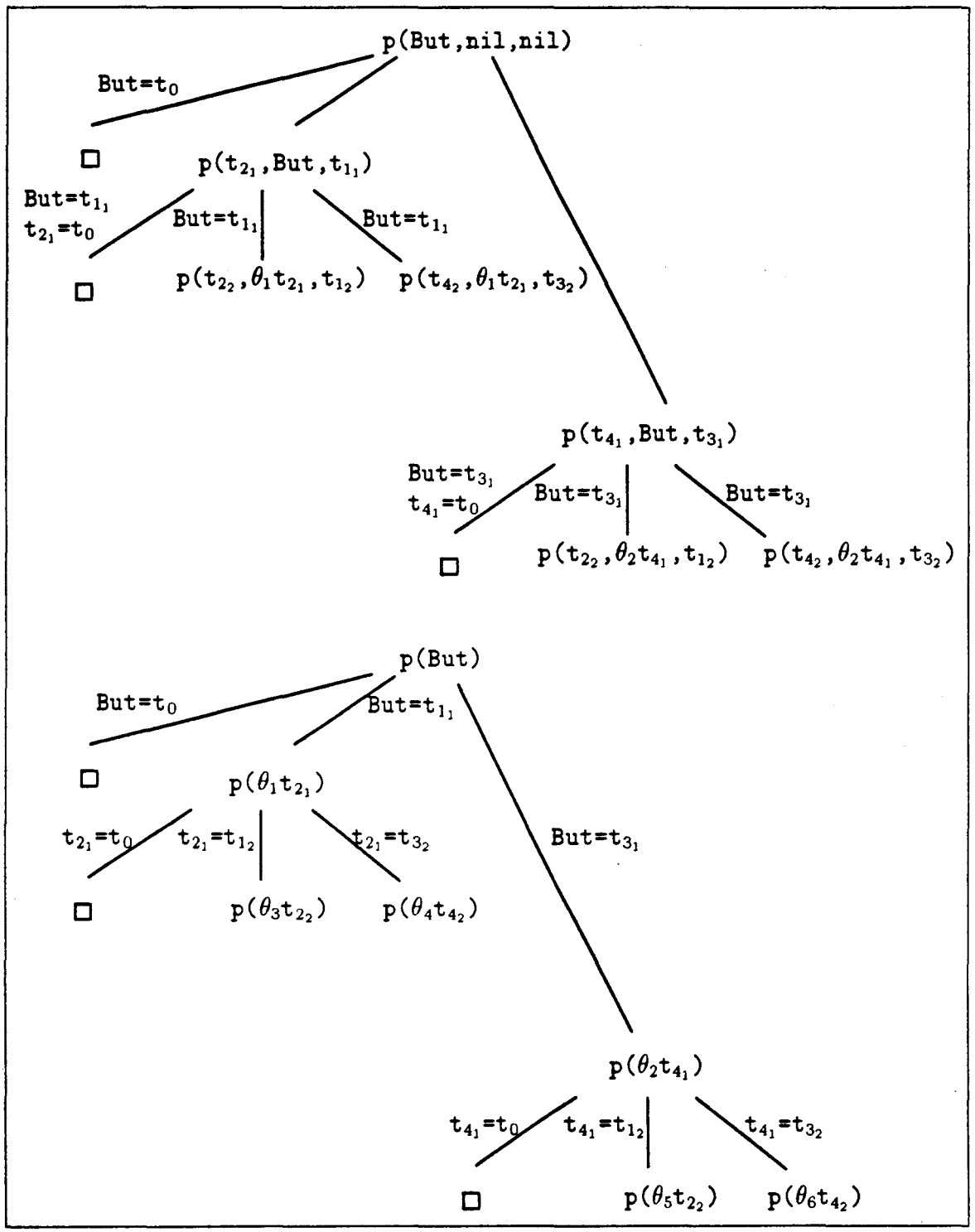


FIG. 4.1 - : Arbres SLD de M_1 et M_2.

Le deuxième argument de M_2 étant toujours une forme instancié de Programme ou d'une partie de Programme, en simplifiant le système, on obtient :

$$\begin{cases} \texttt{ListeDeButs} = [] \\ \texttt{CorpsDeRegle} = [] \\ \texttt{tt}_1 = (\texttt{[]}, \texttt{ProgrammeClos}) \\ \texttt{tt}_3 = (\texttt{[But]}, \texttt{[But|\_]}) \\ \texttt{tt}_2 = (\texttt{[But]}, \texttt{[But|\_]}) \end{cases}$$

où But est un fait de Programme. Il suffit d'apporter une légère modification à Programme pour caractériser But. Considérons Programme′ tel que Programme′ contient toutes les règles de Programme, sauf les faits qui sont remplacés par [a_i,fait|L]-L, et la règle [fait|L]-L qui est ajoutée en fin de Programme′, où fait est un symbole de prédicat d'arité 0 qui n'apparaît pas dans Programme. Programme′ est équivalent au sens de l'équivalence opérationnelle forte à Programme (il suffit d'effectuer un dépliage déterministe sur fait). Le système d'équations peut alors être encore simplifié :

$$\begin{cases} \texttt{ListeDeButs} = [] \\ \texttt{CorpsDeRegle} = [] \\ \texttt{tt}_1 = (\texttt{[]}, \texttt{ProgrammeClos}) \\ \texttt{tt}_3 = (\texttt{[fait]}, \texttt{[fait]}) \\ \texttt{tt}_2 = (\texttt{[fait]}, \texttt{[fait]}) \end{cases}$$

M_1 devient alors :

$$\begin{cases} p(t_0, (\texttt{[fait]}, \texttt{[fait]}), (\texttt{[fait]}, \texttt{[fait]})) \leftarrow \ . \\ p(X, Y, Y) \leftarrow p(t_2, X, t_1) \ . \\ p(X, Y, Y) \leftarrow p(t_4, X, t_3) \ . \\ \leftarrow p(t_5, nil, nil) \ . \end{cases}$$

avec

$$\begin{cases} t_0 = (\texttt{[]}, \texttt{ProgrammeClos}) \\ t_1 = (\texttt{[But|ListeDeButs]}, \texttt{[[But|CorpsDeRegle]-ListeDeButs|\_]}) \\ t_2 = (\texttt{CorpsDeRegle}, \texttt{Programme}) \\ t_3 = (\texttt{ListeDeButs}, \texttt{[\_|FinDuProgramme]}) \\ t_4 = (\texttt{ListeDeButs}, \texttt{FinDuProgramme}) \\ t_5 = (\texttt{[nil,but\_clos]}, \texttt{[[nil,but\_clos]-but\_clos]}) \end{cases}$$

2. Nous montrons maintenant que le programme M_1 avec deux clauses binaires et un fait clos est équivalent à un programme avec une clause ternaire et deux faits (dont un clos). Notons M_3 :

$$\begin{cases} p(t'_0, a, b) \leftarrow \ . \\ p(t', a, b) \leftarrow p(t'_2, X, Y), p(t'_4, Y, X) \ . \\ p(\_, b. a) \leftarrow \ . \\ \leftarrow p(t'_5, a, b) \ . \end{cases}$$

En appliquant la règle de spécialisation des appels (définition 15), on obtient :

$$\begin{cases} p(t'_0, a, b) \leftarrow \ . \\ p(t', a, b) \leftarrow p(t'_2, X, Y) = p(\_, a, b), p(t'_2, X, Y), p(t'_4, Y, X) \ . \\ p(t', a, b) \leftarrow p(t'_2, X, Y) = p(\_, b, a), p(t'_2, X, Y), p(t'_4, Y, X) \ . \\ p(\_, b, a) \leftarrow \ . \\ \leftarrow p(t'_5, a, b) \ . \end{cases}$$

On peut alors éliminer les égalités :

$$\begin{cases} p(t'_0, a, b) \leftarrow \ . \\ p(t', a, b) \leftarrow p(t'_2, a, b), p(t'_4, b, a) \ . \\ p(t', a, b) \leftarrow p(t'_2, b, a), p(t'_4, a, b) \ . \\ p(\_, b, a) \leftarrow \ . \\ \leftarrow p(t'_5, a, b) \ . \end{cases}$$

On effectue deux dépliages déterministes :

$$\begin{cases} p(t'_0, a, b) \leftarrow \ . \\ p(t', a, b) \leftarrow p(t'_2, a, b) \ . \\ p(t', a, b) \leftarrow p(t'_4, a, b) \ . \\ p(\_, b, a) \leftarrow \ . \\ \leftarrow p(t'_5, a, b) \ . \end{cases}$$

On effectue une étape d'indexation sur les corps de règles :

$$\begin{cases} p(X, a, b) \leftarrow q(X) \ . \\ q(t'_0) \leftarrow \ . \\ q(t') \leftarrow q(t'_2) \ . \\ q(t') \leftarrow q(t'_4) \ . \\ \leftarrow q(t'_5) \ . \end{cases}$$

p n'est plus accessible à partir du but, on peut supprimer sa définition. M_3 est bien équivalent à M_1 :

$$\begin{cases} q(t'_0) \leftarrow \ . \\ q(t') \leftarrow q(t'_2) \ . \\ q(t') \leftarrow q(t'_4) \ . \\ \leftarrow q(t'_5) \ . \end{cases}$$

il suffit de prendre

$$\begin{cases} t'_0 = (t_0, (\texttt{[fait]}, \texttt{[fait]}), (\texttt{[fait]}, \texttt{[fait]})) \\ t'_2 = (t_2, X, t_1) \\ t'_4 = (t_4, X, t_3) \\ t' = (X, Y, Y) \\ t'_5 = (t_5, nil, nil) \\ t_0 = (\texttt{[]}, \underline{\texttt{ProgrammeClos}}) \\ t_1 = (\texttt{[But|ListeDeButs]}, \texttt{[[But|CorpsDeRegle]-ListeDeButs|\_]}) \\ t_2 = (\texttt{CorpsDeRegle}, \underline{\texttt{Programme}}) \\ t_3 = (\texttt{ListeDeButs}, \texttt{[\_|FinDuProgramme]}) \\ t_4 = (\texttt{ListeDeButs}, \texttt{FinDuProgramme}) \\ t_5 = (\texttt{[nil,but\_clos]}, \texttt{[[nil,but\_clos]-but\_clos]}) \end{cases}$$

Comme dans la preuve du théorème précédent, il suffit maintenant de considérer que le programme universel à clauses de Horn est codé dans **Programme**, et que `but_clos` est l'entier codant la machine de Turing choisie par l'utilisateur. □

Corollaire 2 *Le problème de l'arrêt et du vide pour les programmes constitués d'une clause de Horn ternaire, de deux faits dont un clos et d'un but clos, est indécidable.*

Preuve Ce corollaire est directement impliqué par le théorème précédent, comme le corollaire 1 est impliqué par le théorème 8. Décider si un programme s'arrête ou s'il a des solutions est en général indécidable. Puisqu'il existe un méta-interpréteur de la classe qui nous intéresse, les problèmes de l'arrêt et du vide sont indécidables pour cette classe. □

4.5 Conclusion

Grâce à nos transformations de programmes, nous avons pu, dans un premier temps, valider le Vanilla méta-interpréteur ainsi qu'un méta-programme quasi-itératif, dont le code est restreint à deux clauses binaires et un fait clos. Ces résultats peuvent être également considérés comme des nouvelles définitions de transformations de programmes (on peut "encapsuler" un programme à l'intérieur de la structure d'un méta-programme).

La deuxième partie de nos résultats peut être mise en rapport avec des résultats récents. Dans [SS88], Schmidt-Schauss cite les trois derniers problèmes ouverts liés à l'implication de clauses :

- le problème du vide pour la classe des ensembles de clauses de Horn constitués d'une clause de Horn binaire et des règles unaires closes est-il décidable?

- $A \Rightarrow B$, pour A et B des clauses, A étant une règle ternaire, est-il décidable?

- $A \Rightarrow B$, pour A une clause de Horn et B une clause arbitraire, est-il décidable?

Le premier problème a été résolu par [DLR93a], le second par Marcinkowski et Pacholski dans [MP92], et le troisième par Marcinkowski dans [Mar93]. Les trois sont des résultats d'indécidabilité.

Notre approche par transformations syntaxiques est une approche différente des approches utilisées pour les trois problèmes de Schmidt-Schauss, et notre dernier résultat est un codage qui approche le résultat d'indécidabilité du vide de Marcinkowski. Nous espérons donc améliorer notre résultat est obtenir la clôture du deuxième fait (ou d'un nombre fini de faits), et ainsi pourvoir fournir une alternative aux approches par les fonctions de Conway ou par les Thue-systèmes.

Conclusion

Nous avons présenté une équivalence opérationnelle entre programmes logiques, et les transformations qui la préservent. Deux programmes sont dits équivalents s'ils ont le même comportement observable. Le pliage et le dépliage, qui sont les transformations de programmes classiques, préservent cette équivalence à quelques restrictions près. Ces transformations semblent assez puissantes puisqu'elles permettent d'exprimer les techniques inspirées de Gallagher et Bruynooghe [GB90]. Nous avons ainsi étudié deux utilisations particulières de ces outils: dans le cadre de l'évaluation partielle et dans le cadre de la méta-programmation.

Pour l'application de nos transformations à l'évaluation partielle, nous avons défini un système générique d'évaluation partielle guidée par interprétation abstraite pour Prolog complet. Notre système est défini à partir des conditions minimales que doivent vérifier les informations fournies par l'analyseur statique pour pouvoir être intégrées dans un programme. Nous avons également montré que Prolog complet peut être pris en compte en restant dans un cadre logique. Les programmes Prolog complet peuvent être optimisés sans pour autant manipuler directement le code de la WAM ni ajouter d'autres prédicats extra-logiques. Néanmoins, il reste plusieurs points à développer autour de cette "plate-forme" :

- une implémentation en Prolog de notre système est en cours. Elle prend en compte l'interprétation abstraite décrite par C. Lecoutre dans sa thèse [Lec94]. Elle nous permettra de tester le système sur des programmes réels;

- puisque nous ajoutons des gardes dans les programmes, il serait intéressant de les prendre en compte à la compilation, et ainsi d'améliorer l'indexation des clauses de la WAM;

- notre système est "interactif" pour la partie transformation proprement dite. Nous allons regarder comment peuvent s'adapter des heuristiques comme, par exemple, celle proposée par Proietti et Pettorossi [PP93].

Pour la partie méta-programmation, nous avons démontré grâce à nos transformations que le Vanilla méta-interpréteur préserve l'équivalence opé-

rationnelle forte du programme qu'il interprète. Nous avons obtenu le même résultat pour un méta-programme quasi-itératif, constitué de deux clauses binaires et d'un fait clos, ou, (de façon équivalente) d'une clause ternaire et de trois faits dont un clos. Mais nous espérons en améliorer la preuve. En effet, il reste encore une étape de la preuve qui est montrée sans utiliser les transformations. Nous devrions pouvoir généraliser cette étape, et montrer qu'il s'agit en fait d'une macro-transformation de programmes qui préserve l'équivalence opérationnelle forte. Les deux codages de programmes, à l'intérieur du Vanilla-interpréteur ou du méta-programme quasi-itératif, peuvent déjà être considérés comme des macro-transformations.

Nous avons ensuite étudié ces résultats pour obtenir (ou réobtenir) des résultats d'indécidabilité du vide et de l'arrêt pour des classes de petits programmes :

- la classe des programmes constitués de deux clauses binaires, d'un but clos et d'un fait clos;
- la classe des programmes constitués d'une clause ternaire, de deux faits dont un clos et d'un but clos.

Ces problèmes sont proches de ceux de l'implication de clauses ($A \Rightarrow B$). Marcinkowski et Pacholski ont montré que dans le cas où A est une clause de Horn ternaire, l'implication est indécidable. Nous espérons, par des codages plus fins de notre second résultat, redémontrer ce cas, avec peut-être une borne sur la taille de B. Cela nous permettrait de proposer une nouvelle approche de ces problèmes.

Bibliographie

[ABFQ92] F. Alexandre, K. Bsaies, J.-P. Finance, and A. Quéré. Spes: a system for logic program transformation. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 445–447, St-Petersburg, July 1992. Springer-Verlag.

[AMP92] K.R. Apt, E. Marchiori, and C. Palamidessi. A theory of first-order built-in's of Prolog. In Springer-Verlag, editor, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science, Pisa, Italy, 1992.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B - Formal models and semantics, chapter 10, pages 493–574. Elsevier, 1990.

[AR89] H. Abramson and M.H. Rogers, editors. *Meta-Programming in Logic Programming.* Logic Programming serie. MIT Press, 1989.

[AvE82] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the Association Computing Machinery*, 29(3):841–862, July 1982.

[Azi87] N. Azibi. *TREQUASI: Un système pour la transformation automatique de programmes Prolog récursifs en quasi-itératifs.* PhD thesis, Université de Paris-Sud, December 1987.

[BB92] D. Boulanger and M. Bruynooghe. Deriving Fold/Unfold transformations of logic programs using extended OLDT-based abstract interpretation. In Clement and Lau [CL92].

[BC93] A. Bossi and N. Cocco. Basic transformation operations which preserve computed answer-substitutions of logic programs. *Journal of Logic Programming*, 16(1-2):47–87, May 1993. Special issue Partial Deduction.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association Computing Machinery*, 24(1):44–67, January 1977.

[BH87] D. R. Brough and C. J. Hogger. Compiling associativity into logic programs. *The Journal of Logic Programming*, 4(4):345–360, December 1987.

[Bil91] M. Billaud. Une sémantique dénotationnelle simple pour Prolog avec prédicats d'entrées-sorties. In *Proceedings of GULP'91*, Pisa, Italy, 1991.

[BK82] K.A. Bowen and R.A. Kowalski. Amalgamating language and meta-language. In K.L. Clark and S.-A. Tärnlund, editors, *Logic programming*, pages 154–172. Academic Press, 1982.

[BL89] K. Benkerimi and J.W. Lloyd. A procedure for the partial evaluation of logic programs. Technical Report 89-04, University of Bristol, May 1989.

[Bou91] A. Bouverot. *Comparaison entre la transformation et l'extraction de programmes logiques.* PhD thesis, Université de Paris VII, March 1991.

[BR89] M. Bugliesi and F. Russo. Partial Evaluation in Prolog: Some Improvements about Cut. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 645–660, Cleveland, Ohio, USA, 1989.

[Bru90] M. Bruynooghe, editor. *Proceedings of the second workshop on Meta-programming in Logic*, Leuven, Belgium, April 1990.

[CL91] T.P. Clement and K.-K. Lau, editors. *Logic Program Synthesis and Transformation*, Workshops in Computing, Manchester, July 1991. Springer-Verlag.

[CL92] T.P. Clement and K.-K. Lau, editors. *Logic Program Synthesis and Transformation*, Workshops in Computing, Manchester, July 1992. Springer-Verlag.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and databases*, pages 293–322. Plenum Press, New-York, 1978.

[Cla79] K.L. Clark. Predicate logic as a computational formalism. Technical Report 79/59 TOC, Imperial College, December 1979.

[CT77] K. Clark and S.-Å. Tärnlund. A first theory of data and programs. *Information Processing*, pages 939–944, 1977.

[DD91] F. Denis and J.-P. Delahaye. Unfolding, procedural and fixpoint semantics of logic programs. In Springer-Verlag, editor, *Proceedings of STACS'91*, Lecture Notes in Computer Science, pages 511–522, 1991.

[Del87] J.-P. Delahaye. *Outils logiques pour l'intelligence artificielle.* Eyrolles, Paris, 1987.

[Del88] J.-P. Delahaye. Sémantique logique et dénotationnelle des interpréteurs Prolog. *Informatique Théorique et Applications*, 22(1):3–42, 1988.

[DLR93a] Ph. Devienne, P. Lebègue, and J.-C. Routier. The emptiness problem of one binary Horn clause is undecidable. In D. Miller, editor, *Proceedings of ILPS'93*, Logic Programming Serie, pages 250–265. MIT Press, October 1993.

[DLR93b] Ph. Devienne, P. Lebègue, and J.-C. Routier. Halting problem of ome binary Horn clause is undecidable. In P. Enjalbert, A. Finkel, and K.W. Wagner, editors, *Proceedings of STACS'93*, volume 665 of *Lecture Notes in Computer Science*, pages 48–57. Springer-Verlag, February 1993.

[DLRW94] Ph. Devienne, P. Lebègue, J.-C. Routier, and J. Würtz. One binary Horn clause is enough. In *Proceedings of STACS'94*, Lecture Notes in Computer Science. Springer-Verlag, February 1994. à paraître.

[DM88] Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for Prolog. *The Journal of Logic Programming*, 5(1):61–91, March 1988.

[DM93] P. Deransart and J. Małuszyński. *A grammatical view of logic programming.* Logic Programming Serie. MIT Press, November 1993.

[DW86] Saumya K. Debray and David S. Warren. Detection and optimization of functional computations in Prolog. In Shapiro [Sha86], pages 490–504.

[Fer87] G. Ferrand. Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method. *Journal of Logic Programming*, 4:177–198, September 1987.

[FLPM89] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69:289–318, 1989.

[Fuj87] H. Fujita. An algorithm for partial evaluation with constraints. Technical Memorandum 0367, ICOT, 1987.

[GB90] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In Bruynooghe [Bru90], pages 229–244.

[GB91] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9:305–333, 1991.

[GCS88] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6(2,3):159–186, 1988. Special issue Partial Evaluation and Mixed Computation.

[Hil91] P. M. Hill. Pruning operators for partial evaluation. In Clement and Lau [CL91], pages 183–204.

[HJ92] N. Heintze and J. Jaffar. An engine for logic program analysis. In *Seventh IEEE Symposium on Logic in Computer Science*, pages 318–328, Santa-Cruz, June 1992.

[HL89] P.M. Hill and J.W. Lloyd. Meta-Programming in Logic Programming. In Abramson and Rogers [AR89], chapter 2, pages 23–51.

[HL91] P. Hill and J.W. Lloyd. The Gödel report. Technical Report 91-02, University of Bristol, Department of Computer Science, March 1991.

[HM89] T. Hickey and S. Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.

[JL86] J. Jaffar and J.-L. Lassez. Constraint logic programming. Technical Report 74, Monash University, Victoria, Australia, June 1986.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM.

[KK88] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic transformation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 413–421. ICOT, 1988.

[KKH87] T. Kanamori, T. Kawamura, and K. Horiuchi. Dectecting termination of logic programs based on abstract hybrid interpretation. Technical Report TR-398, ICOT, Tokyo, 1987.

[Kom81] J. Komorowski. *A specification of an abstract Prolog machine and its application to partial evaluation.* PhD thesis, Department of Computer Science, Linköping University, Sweden, 1981.

[Kom93] J. Komorowski. A prolegomenon to partial deduction. *Fundamentae Informaticae*, 18:41–64, 1993.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In North Holland, editor, *Proceedings of IFIP'74*, pages 569–574, Amsterdam, 1974.

[Kow79] R.A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.

[Lak89] A. Lakhotia. Promix user's manual. Technical Report CES-89-05, Case Western Reserve University, March 1989.

[LDL92a] C. Lecoutre, Ph. Devienne, and P. Lebègue. Determinacy induction by means of an abstract OLDT resolution. In *Addendum to Proceedings of WSA '92*, pages 1–8, 1992.

[LDL92b] C. Lecoutre, Ph. Devienne, and P. Lebègue. Termination induction by means of an abstract OLDT resolution. In J.-P. Delahaye, Ph. Devienne, P. Mathieu, and P. Yim, editors, *Premières Journées Francophones sur la Programmation en Logique*, pages 353–373, Lille, May 1992.

[Lec94] C. Lecoutre. *Interprétation abstraite en programmation logique avec contraintes.* PhD thesis, Université de Lille 1, February 1994. à paraître.

[Llo87] J. W. Lloyd. *Foundations of logic programming.* Springer Verlag, 1987.

[LM89] Giorgio Levi and Maurizio Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, 1989. The MIT Press.

[LMM87] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J.Minker, editor, *Foundations of Deductive Databases*, pages 587–623. Morgan Kaufmann, 1987.

[LS87] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. Technical Report 87-09, University of Bristol, December 1987.

[Mah87] M.J. Maher. Equivalences of logic programs. In J.Minker, editor, *Foundations of Deductive Databases*, pages 627–658. Morgan Kaufmann, 1987.

[Mah92] M.J. Maher. *Lecture Notes on Constraint Logic Programming: Analysis, Transformation and Semantics.* Fourth International School for Computer Science Researchers, Acireale, Sicily, June 1992.

[Mar93] J. Marcinkowski. A Horn clause that implies an undecidable set of Horn clauses. 1993.

[MNL90] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.

[MP92] J. Marcinkowski and L. Pacholski. Undecidability of the Horn clause implication problem. In *Proceedings of FOCS'92*, 1992.

[MS93] K. Marriott and H. Søndergaard. Difference-list transformation for prolog. *New Generation Computing*, 11:125–157, 1993.

[O'K90] R.A. O'Keefe. *The craft of Prolog.* Logic Programming. MIT Press, 1990.

[Par90] H.A. Partsch. *Specification and transformation of programs, a formal approach to software development.* Texts and monographs in computer science. Springer-Verlag, 1990.

[PDL91] A. Parrain, Ph. Devienne, and P. Lebègue. Prolog program transformations and meta-interpreters. In Clement and Lau [CL91], pages 238–251.

[PP89] A. Pettorossi and M. Proietti. Decidability results and characterization of strategies for the development of logic programs. In Levi and Martelli [LM89], pages 539–553.

[PP90] M. Proietti and A. Pettorossi. Construction of efficient logic programs by loop absorption and generalization. In Bruynooghe [Bru90], pages 57–81.

[PP91] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation*, New-Haven, U.S.A., June 1991.

[PP93] M. Proietti and A. Pettorossi. An abstract strategy for transforming logic programs. *Fundamentae Informaticae*, 18:267–286, 1993.

[Pre92a] S. Prestwich. Paddy. Technical report, ECRC, 1992.

[Pre92b] S. Prestwich. An unfold rule for full Prolog. In Clement and Lau [CL92].

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association Computing Machinery*, 12(1):23–41, January 1965.

[Rou94] J.-C. Routier. *Terminaison, Satisfiabilité, Puissance calculatoire d'une clause de Horn binaire.* PhD thesis, Université de Lille 1, February 1994. à paraître.

[Sah91] D. Sahlin. *An automatic partial evaluator for full Prolog.* PhD thesis, Swedish Institute of Computer Science, Stockholm, March 1991.

[Sek89] H. Seki. Unfold/fold transformation of stratified programs. In Levi and Martelli [LM89], pages 554–568.

[SH90] Donald A. Smith and Timothy J. Hickey. Partial evaluation of a CLP language. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 119–153, Austin, 1990. ALP, MIT Press.

[Sha86] Ehud Shapiro, editor. *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, London, 1986. Springer-Verlag.

[She92] J. Shepherdson. Unfold/fold transformations of logic programs. *Math. Struc. in Computer Science*, 2:143–157, 1992.

[Smi91] Donald A. Smith. Constraint operations for CLP($\mathcal{FT}$). In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 760–774, Cambridge, Massachusetts London, England, 1991. MIT Press.

[SS86] L. Sterling and E. Shapiro. *The art of Prolog.* Logic Programming Serie. MIT Press, 1986.

[SS88] M. Schmidt-Schauss. Implication of clauses is undecidable. *Theoretical Computer Science*, (59):287–296, 1988.

[Sub89] V.S. Subrahmanian. A simple formulation of the theory of metalogic programming. In Abramson and Rogers [AR89], chapter 4, pages 65–101.

[Swe93] Swedish Institute of Computer Science. *SICStus Prolog user's manual*, January 1993. for SICStus Prolog 2.1.

[Tär77] S.-Å. Tärnlund. Horn clause computability. *BIT*, 17(2):215–226, 1977.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In Sten-Åke Tärnlund, editor, *Second International Logic Programming Conference*, pages 127–138, Uppsala, 1984.

[TS86] H. Tamaki and T. Sato. OLD resolution with tabulation. In Shapiro [Sha86], pages 84–98.

[VD88] R. Venken and B. Demoen. A partial evaluation system for Prolog: some practical considerations. *New Generation Computing*, 6(2,3):279–290, 1988. Special issue Partial Evaluation and Mixed Computation.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association Computing Machinery*, 23(4):733–742, October 1976.

[ZG88] J. Zhang and P.W. Grant. An automatic difference-list transformation algorithm for Prolog. In *Procceedings of ECAI'88*, pages 320–325, 1988.