Utilisation des modes directionnels dans la résolution

Olivier Oudot

THESE

présentée par

Olivier OUDOT

pour obtenir le titre de **DOCTEUR**

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(arrêté ministériel du 5 juillet 1984)

Spécialité : INFORMATIQUE

==========

UTILISATION DES MODES DIRECTIONNELS DANS LA RESOLUTION

==========

Date de soutenance : *30 Novembre 1987*

Composition du Jury :	*Président*	S.Krakowiak
	Rapporteurs	B. Lepape
		L.Trilling
	Examinateurs	J. Mossière
		M. Van Caneghem
		J. Briat

Thèse préparée au sein du Laboratoire de Génie Informatique à l'Université Scientifique Technologique et Médicale de Grenoble

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Daniel BLOCH **Année 1987**

Vice - Présidents : René CARRE
Jean-Marie PIERRARD

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BESSON Jean	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
BUYLE-BODIN Maurice	ENSERG
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHERUY Arlette	ENSIEG
CHIAVERINA Jean	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Mar	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Pierre	ENSIEG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSEEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PAUTHENET René	ENSIEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SERMET PIERRE	ENSERG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
PLA Fernand
GHIBAUDO Gérard
HAMAR Sylvaine
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CAILLET Marcel
CARRE René
FRUCHART Robert
JORRAND Philippe
LANDAU Ioan
MARTIN

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BINDER Gilbert
BONNET Roland
BORNARD Guy
CALMET Jacques
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
JOUD Jean Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MERMET Jean
MUNIER Jacques
SENATEUR Jean-Pierre
SUERY Michel
TEDOSIU
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

BERNARD Claude
CHATILLON Catherine
CHATILLON Christian
COULON Michel
DIARD Jean-Paul
FOSTER Panayotis
HAMMOU Abdelkader
MALMEJAC Yves
MARTIN GARIN Régina
SAINTFORT Paul
SARRAZIN Pierre
SIMON Jean-Paul
TOUZAIN Philippe
URBAIN Georges

E.N.S.E.R.G

BOREL Joseph
CHOVET Alain
DOLMAZON Jean-Marc
HERAULT Jeanny

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

BOIS Daniel
DARVE Félix
MICHEL Jean-Marie
ROWE Alain
VAUCLIN Michel

E.N.S.I.M.A.G

BERT Didier
COURTIN Jacques
COURTOIS Bernard
DELLA DORA Jean
FONLUPT Jean
SIFAKIS Joseph

E.F.P.G

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEMOULIN Eric
DEVINE
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M.MERMET
Directeur des Etudes et de la formation: Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

PROFESSEURS DE 1ère CATEGORIE

COINDE Alexandre	Gestion
GOUX Claude	Métallurgie
LEVY Jacques	Métallurgie
LOWYS Jean-Pierre	Physique
MATHON Albert	Gestion
RIEU Jean	Mécanique-Résistance des matériaux
SOUSTELLE Michel	Chimie
FORMERY Philippe	Mathématiques Appliquées

PROFESSEURS DE 2ème CATEGORIE

HABIB Michel	Informatique
PERRIN Michel	Géologie
VERCHERY Georges	Matériaux
TOUCHARD Bernard	Physique Industrielle

DIRECTEUR DE RECHERCHE

LESBATS Pierre	Métallurgie

MAITRE DE RECHERCHE

BISCONDI Michel	Métallurgie
DAVOINE Philippe	Géologie
FOURDEUX Angeline	Métallurgie
KOBYLANSKI André	Métallurgie
LALAUZE René	Chimie
LANCELOT Francis	Chimie
LE COZE Jean	Métallurgie
THEVENOT François	Chimie
TRAN MINH Canh	Chimie

Personalités habilitées à diriger des travaux de recherche

DRIVER Julian	Métallurgie
GUILHOT Bernard	Chimie
THOMAS Gérard	Chimie

Professeurs à l'UER de Sciences de Saint-Etienne

VERGNAUD Jean-Maurice	Chimie des Matériaux et Chimie Industrielle

à Judith
à mes parents

Je tiens à remercier

Sacha Krakowiak, Professeur à l'Université Scientifique, Technologique et Médicale de Grenoble, pour avoir accepté de présider le jury de cette thèse, de même que pour m'avoir accueilli dans son équipe à mon arrivée.

Laurent Trilling, Professeur à l'Université Scientifique, Technologique et Médicale de Grenoble, pour avoir consacré beaucoup de temps à apporter ses critiques à ce travail.

Brice Lepape, Ingénieur de Recherche à la Communauté Economique Européenne, pour avoir apporté son jugement sur cette thèse.

Jacques Mossière, Professeur à l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, et Directeur du Laboratoire de Génie Informatique, pour m'avoir reçu au sein de son laboratoire et pour avoir bien voulu diriger ce travail.

Michel Van Caneghem, Ingénieur au Centre National de la Recherche Scientifique, pour avoir accepté de participer au jury de cette thèse.

Jacques Briat, qui a bien voulu me guider par son expérience. Les discussions avec lui ont été passionnées mais très constructives. Qu'il soit également remercié pour l'aide matérielle apportée à la réalisation de cette thèse.

Yann Rouzaud, pour m'avoir constamment critiqué et conseillé par de longs échanges au cours desquels se sont concrétisées la plupart des idées de ce travail, de même que pour l'aide qu'il a apporté à la rédaction de cette thèse.

Tous les membres de l'équipe FLOP, de même que tous les occupants du batiment B, pour leur constante gentillesse qui permet de maintenir une ambiance de travail sympathique.

Toutes les personnes qui ont contribué, de près ou de loin, au bon accomplissement de ce travail.

Cette thèse a pu être effectuée avec le support d'une BDI (PRC-C3) CNRS.

Table des matières

Introduction

Les langages de programmation les plus couramment utilisés actuellement (Pascal, C, etc.) ont été conçus comme une abstraction des machines de type Von Neuman. Dans ces langages, la programmation d'un algorithme exige de l'utilisateur la traduction de l'expression mathématique de son problème en un enchaînement d'expressions d'affectation à des variables, et d'expressions de contrôle sur l'état de ces variables.

Ce travail est facilité par un ensemble de recettes ou de procédés semi-automatiques de transformation de certaines classes de problèmes en des algorithmes qui les résolvent. Les caractéristiques dominantes de ces classes sont :

- le domaine généralement fini et connu des données à manipuler,
- l'existance "a priori" d'une solution du problème.

Cette approche a trouvé sa limite notamment avec le développement des applications de type "intelligence artificielle", qui ne vérifient pas ces propriétés. Il n'est donc pas anormal que les langages actuellement utilisés ou développés pour ces applications procèdent d'une démarche inverse. La solution d'un problème y est décrite par l'expression des propriétés de la solution souhaitée. L'algorithme est automatiquement fourni par l'exploitation des propriétés, par l'intermédiaire d'un principe opératoire qui n'est autre que l'interpréteur du langage d'expression de ces propriétés.

Ainsi, Lisp est un langage dit "fonctionnel", où les propriétés s'expriment comme des compositions de fonctions. Le principe opératoire, la **Réduction**, est la règle d'évaluation des fonctions.

Algorithme = Fonctions + Réduction

Le langage Prolog est un langage dit "logique", où les propriétés s'expriment comme des formules logiques réduites aux clauses de Horn. Le

principe opératoire en est la **Résolution** *[Robinson 65]*.

Algorithme = Logique + Résolution

En dépit d'un important gain de rapidité de programmation, l'utilisation d'un principe très général comme la Réduction ou la Résolution n'est pas sans poser de sérieux problèmes d'efficacité, notamment dans les domaines propres aux autres langages de programmation déjà existants.

Lisp, langage plus ancien que Prolog, l'a en partie résolu en introduisant un grand nombre de traits provenant des langages classiques. Ce ne peut être le cas de Prolog qui est beaucoup trop éloigné de ces langages. Cette amélioration n'est donc possible que par un travail de fond, permettant d'accroître l'efficacité de la mise en oeuvre de la Résolution.

Le domaine de données du langage Prolog est l'univers des termes, c'est à dire des structures arborescentes comportant des constantes (atomes) ou des variables (inconnues au sens mathématique). Les connaissances sur un domaine donné s'expriment comme des propositions clausales (logique du premier ordre) sur des termes.

Un problème est posé comme une propriété à vérifier sur des termes. Sa résolution consiste à déterminer, par une série d'essais/erreurs, quelles sont les valeurs des variables pour lesquelles la propriété est vérifiée. La série d'essais/erreurs est dirigée par le parcours d'un *arbre de recherche*, défini à partir des clauses du programme et de la propriété à vérifier. Ce parcours est réalisé par l'intermédiaire d'un opérateur unique et général, l'**unification**, qui réalise notamment l'affectation des variables.

L'un des problèmes majeurs de la mise en oeuvre de la Résolution réside dans le fait que l'arbre de recherche peut (c'est en fait souvent le cas) contenir des branches infinies, dues à la définition récursive de certains prédicats. Son exploration, généralement effectuée suivant une *stratégie de parcours* fixe et unique qui constitue la *partie contrôle* du langage, peut donc entrainer un bouclage du processus d'interprétation.

L'implantation de Prolog n'est donc pas sans poser un grand nombre de problèmes, parmi lesquels on peut citer :

- L'utilisation d'un opérateur général, l'unification, se traduit par une vitesse d'exécution excessivement lente pour des problèmes ne nécéssitant pas cette généralité.

- La représentation des termes et plus particulièrement des variables exige l'utilisation d'un grand espace mémoire.

- Une stratégie de parcours de l'arbre de recherche fixe et systématique est insuffisante et conduit trop souvent à un bouclage du processus de résolution.

- L'absence de contrôle explicite rend la programmation de certains problèmes très difficile.

Par exemple, l'une des limitations de Prolog réside dans l'insuffisance du modèle logique qui ne permet pas d'exprimer certains faits, notamment les inégalités de termes. Ces insuffisances sont actuellement palliées par l'utilisation d'opérateurs de contrôle explicite primaires. Cette technique n'est malheureusement pas à la portée du programmeur moyen, c'est pourquoi de nombreuses études se consacrent à l'extension du langage en permettant de traiter notamment les inéquations ou les arbres infinis *[Colmerauer 83]*.

Un autre exemple est la difficulté de prendre en compte les aspects pragmatiques que sont les entrées/sorties qui correspondent à des effets de bord sur la résolution, et qui sont incompatibles avec le modèle logique de Prolog. La prise en compte de ces problèmes se fait par l'intermédiaire d'opérateurs de contrôle et d'opérateurs extra-logiques.

Toutefois, les avantages du langage, les besoins de langages de programmation spécifiques de l'intelligence artificielle, et la compétition industrielle déclenchée par le projet japonais d'ordinateurs de cinquième génération ont provoqué le lancement de nombreuses études destinées à corriger ces inconvénients. Diverses voies sont explorées pour tenter d'améliorer le langage.

Une première voie se consacre à l'exploitation de propriétés extraites par l'analyse de programmes. Dans ce domaine, les deux grands créneaux d'application sont les suivants :

- L'optimisation de l'unification. En effet, des mesures effectuées sur des programmes Prolog *[Matsumoto 85]* montrent que l'unification dans toute sa généralité est bien souvent inutile. La connaissance de propriétés sur les termes permet de la spécialiser pour des contextes d'utilisation précis. C'est une propriété de ce type qui est utilisée dans les compilateurs Prolog *[Warren 77]* dont l'efficacité avoisine celle de Lisp. Les *modes*, introduits par David Warren, sont maintenant d'une utilisation générale dans la plupart des compilateurs du marché. Ces

modes sont actuellement indiqués par le programmeur et peuvent provoquer des erreurs à l'exécution si la propriété n'est pas vérifiée. Un effort important est fait à l'heure actuelle pour, soit vérifier la correction de telles propriétés, soit les extraire par analyse du programme.

- La mise en oeuvre de stratégies de parcours de l'arbre de recherche adaptées pour :
  - éviter le parcours des branches infinies,
  - éviter les calculs inutiles, c'est à dire le parcours de branches dont on peut dire à l'avance qu'elles aboutissent à un échec.

  Pour ce faire, deux techniques sont utilisées pour sélectionner les "bonnes" branches de l'arbre : un choix dynamique, effectué pendant l'exécution du programme, ou un choix statique, effectué préalablement, notamment en exploitant les modes. La première technique offre l'avantage de pouvoir s'adapter à toutes les situations mais présente l'inconvénient de ralentir le processus de résolution. La deuxième technique s'adapte mieux au cadre de la compilation mais reste imprécise quand la sélection des branches est dépendante des données.

Une autre voie, plus récente, consiste à tenter d'exploiter le parallélisme lié au non-déterminisme intrinsèque du langage. Cette nouvelle possibilité rajoute plusieurs dimensions au problème déjà complexe de l'implantation de la résolution. Actuellement, bien que quelques progrès significatifs aient été réalisés, les recherches se heurtent à des problèmes nouveaux, surtout au niveau de la définition d'architectures spécialisées et de noyaux de systèmes adaptés à ce genre de réalisation.

Le but de cette thèse est de présenter une technique d'optimisation de la Résolution basée sur l'exploitation d'une catégorie particulière de modes : les **modes directionnels**, dont une caractéristique importante est qu'ils sont indépendants de la stratégie de résolution, contrairement aux modes de Warren. Cette technique permet d'aborder un grand nombre de catégories d'optimisations, particulièrement en utilisant la restriction que sont les **modes stricts** qui autorisent notamment :

- La simplification de l'unification qui peut être restreinte à une *égalité*.

- Un contrôle statique de la stratégie de résolution qui permet d'accroître son efficacité en réduisant la taille des arbres de recherche.

- La simplification de la structure des termes qui permet une réduction de l'espace mémoire nécessaire.

- La transformation des programmes Prolog en programmes fonctionnels. Ceci autorise l'emploi de techniques de résolution mieux connues, ainsi que plus simples et plus efficaces.

Plan de la thèse

Après une brève présentation du langage Prolog, le chapitre 1 expose les bases théoriques du langage, puis aborde les différentes techniques de contrôle de la stratégie de résolution.

Les principales méthodes d'optimisation actuellement présentées dans la littérature sont exposées dans le chapitre 2. Nous y présentons notamment les modes, ainsi que la notion d'**expansion de l'unification** qui est une transformation de base pour l'utilisation de ces techniques.

Le chapitre 3 aborde le sujet principal de cette thèse : les modes directionnels. Nous y présentons tout d'abord une notion plus générale, les modes structurels, puis nous exposons les différents avantages que présentent l'exploitation des modes directionnels dans la résolution, en traitant plus particulièrement de leur restriction, les modes stricts. Enfin nous donnons un algorithme permettant une production automatique des modes directionnels.

Le chapitre 4 présente la réalisation d'un compilateur Prolog, Starlog, basé sur une utilisation exclusive des modes stricts. Son implantation a été réalisée pour évaluer les possibilités offertes par l'exploitation des modes stricts dans le cadre de la compilation de Prolog.

CHAPITRE 1

Principes de la programmation logique

L'implantation d'un langage de programmation comme Prolog peut être considérée comme une mise en oeuvre d'un démonstrateur de théorèmes exprimés sous forme de clauses, par l'application du Principe de Résolution *[Robinson 65]*. Kowalski et Van Emden *[Kowalski 76]* en ont fourni un premier modèle théorique, en s'appuyant sur une restriction de l'ensemble des clauses, les clauses de Horn.

Un démonstrateur de théorèmes ne peut toutefois pas être apparenté à un langage de programmation à part entière, dans la mesure où il n'est pas utilisable sans y introduire de multiples extensions ; celles-ci se décomposent essentiellement en trois catégories :

- Opérateurs de communication avec le monde extérieur (entrées/sorties) et de manipulation de données liées à des opérateurs cablés existants (opérateurs arithmétiques par exemple).

- Opérateurs de contrôle du processus de résolution. Par cet intermédiaire, il est également possible de programmer de nouveaux opérateurs, comme la négation, qui n'apparaissent pas tels quels dans Prolog.

- Opérateurs de modification dynamique de programmes. Ces opérateurs permettent d'ajouter ou de retirer dynamiquement de nouvelles clauses.

La première catégorie d'extensions nécessite la connaissance par le programmeur de la stratégie de résolution utilisée. Il est en effet important que les entrées/sorties soient effectuées dans un certain ordre. En outre, la deuxième catégorie suppose une déformation possible de cette stratégie en fonction des besoins du programmeur.

Après avoir abordé brièvement les aspects les plus intéressants du langage, nous allons présenter les bases théoriques de la programmation

logique. Nous exposerons alors les différentes techniques utilisées pour modifier la stratégie standard ainsi que leur différents avantages et inconvénients.

1. Présentation du langage

Le langage Prolog est un langage ayant un très haut niveau d'abstraction comparativement aux langages procéduraux classiques. L'une de ses principale caractéristique réside dans le fait que l'algorithme nécessaire à la résolution d'un problème donné n'a pas besoin d'être fourni explicitement par le programmeur.

Le travail du programmeur consiste essentiellement à décrire un **domaine**, composé d'**objets**, ainsi que les **connaissances** qu'il a sur ce domaine, représentées sous forme de **relations** entre les objets.

Supposons par exemple que l'on ait besoin de traiter d'un problème portant sur les liens familiaux. On dispose d'un certain nombre de faits constituant la filiation suivante :

Jean est le père de Paul
Jean est le père de Marc
Paul est le père de Pierre

Cet ensemble de connaissances peut être représenté en Prolog par l'ensemble de **faits** suivant :

père(Paul, Jean) ←.
père(Marc, Jean) ←.
père(Pierre, Paul) ←.

qui est une représentation des relations liant les objets Paul, Marc, Jean et Pierre.

D'autre part, nous savons que le grand-père d'une personne est définie comme le père du père de cette personne. Cette connaissance s'exprime en Prolog sous forme de la **clause** suivante :

grandpère(f, g) ← père(f, p), père(p, g).

où les variables f, p et g représentent respectivement un fils, un père et un grand-père. Cette clause peut se lire : "g est le grand-père de f si p est le père de f et g le père de p".

Pour savoir qui est le père de Pierre, il suffit en Prolog de poser la question suivante :

← père(Pierre,, x).

où la variable x représente un objet quelconque vérifiant la relation. L'interprète Prolog répond alors : $x = Paul$.

Si nous désirons connaître le père de Jean, la question sera :

← père(Jean, x).

et la réponse : *faux*.

Ceci exprime le fait qu'il n'y a pas de père pour Jean dans la mesure où il n'a pas été défini dans les faits.

Pour connaître les fils de Jean, la question se formule par :

← père(x, Jean).

à laquelle l'interprète répond par les deux solutions : $x = Paul$ et $x = Marc$

Ces exemples illustrent l'intérêt de la programmation relationnelle, et particulièrement de la propriété de **réversibilité** qu'elle offre, dans la mesure où la relation *père* se comporte fonctionnellement aussi bien pour obtenir un père qu'un fils.

Si nous désirons savoir qui est le grand-père de Pierre, la question est :

← grandpère(Pierre, x).

et provoque la réponse : $x = Jean$.

La clause définissant un grand-père peut également être considérée comme définissant un petit-fils et utilisée comme telle. Ainsi, la question :

← grandpère(x, Jean).

voit la réponse : $x = Pierre$.

Enfin, il est également possible d'obtenir tous les couples de personnes tels que l'une est le grand-père de l'autre en posant la question :

← grandpère(x, y).

qui produit la solution unique : $x = Pierre, y = Jean$.

Ces exemples très simples mettent en évidence les avantages principaux du langage Prolog, où l'expression d'un problème sous forme de clauses permet :

- de ne pas exprimer explicitement l'algorithme nécessaire à la résolution du problème,

- de définir en un seul exposé le moyen de résoudre plusieurs types de problèmes.

Ces caractéristiques sont malheureusement loin d'être vérifiées pour toutes les catégories de programmes qu'il est possible d'écrire dans ce langage. La suite de ce chapitre nous en montre quelques exemples et présente quelques solutions adoptées pour y remédier.

2. Bases théoriques

Nous rappelons ici la terminologie utilisée dans la programmation logique. Certaines définitions peuvent être trouvées dans leur contexte général dans *[Chang 73]*, ou de façon plus orientée vers Prolog dans *[Sterling 86]*.

2.1. Les clauses de Horn

On suppose disposer des ensembles suivants :

- un ensemble de variables (notées en minuscules),
- un ensemble de symboles fonctionnels dotés d'une arité (les constantes, ou symboles fonctionnels d'arité zéro, seront notées en commençant par une majuscule),
- un ensemble de symboles de prédicats.

Soit T l'ensemble des **termes** définis par :

- une variable est un terme ;
- si f est un symbole fonctionnel d'arité n et t1,..,tn sont des termes, alors f(t1,..,tn) est un terme.

Un **terme structuré** est un terme comportant au moins un symbole fonctionnel d'arité non nulle.

Un **terme complexe** est un terme structuré comportant au moins une variable (La littérature anglaise emploie parfois "complex term" dans le sens de "terme structuré").

Un **terme clos** est un terme ne comportant aucune variable. L'ensemble des termes clos forme l'**Univers de Herbrand**.

Un **littéral** positif (ou formule atomique) est de la forme P(t1,..,tn) où P est un **prédicat** d'arité n et t1,..,tn des termes appelés arguments du prédicat. Un littéral négatif est un littéral positif précédé du symbole de négation ¬.

Une **clause** est une disjonction de littéraux négatifs et positifs dont les variables sont universellement quantifiées :

$$A_1 \vee A_2 \vee \ldots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m \qquad m,n > 0$$

où Ai et Bj sont des formules atomiques. Les Bj sont les **conditions** de la clause et les Ai sont ses **conclusions**. On peut également noter cette clause :

$$A_1 \vee A_2 \vee ... \vee A_n \leftarrow B_1 \wedge B_2 \wedge ... \wedge B_m$$

ou

$$A_1, A_2, ..., A_n \leftarrow B_1, B_2, ..., B_m$$

Le symbole "$\leftarrow$" correspond à l'implication et peut se prononcer "si".

Si m = 0, cette formule est une **assertion inconditionnelle**.
Si n = 0, il s'agit d'une **réfutation**.
Si m = 0 et n = 0, il s'agit de la clause vide notée [].

Une **clause de Horn** est une clause contenant au plus un littéral positif ($n \leq 1$). Sa forme générale est donc de trois types:

type 1 :	$A \leftarrow B_1, B_2, ..., B_m$	(clause d'inférence)
type 2 :	$A \leftarrow$	(assertion)
type 3 :	$\leftarrow B_1, B_2, ..., B_m$	(réfutation)

On appelle **programme** un ensemble formé de clauses de type 1 et 2. La résolution est généralement déclanchée sur la formulation d'une clause de type 3 ou **question.**

2.2. L'Unification

2.2.1. *Substitution et Instanciation*

Un **élément de substitution** est un couple (x,t), noté souvent x/t, où x est une variable quelconque et t un terme quelconque différent de x.

Une substitution σ est un ensemble fini d'éléments de substitution tel que deux éléments de substitutions ne portent pas sur les même variables :

$$\sigma = \{ x_i/t_i,\ i=1..k \text{ avec } x_i \neq x_j \text{ si } i \neq j \}$$

Une substitution σ représente une fonction unaire définie sur T et à valeurs dans T. On écrit alors $tb=\sigma(ta)$ et on dit que tb est l'**instanciation** de ta

par σ. Si σ={xi/ti, i=1..k}, tb s'obtient en remplaçant simultanément dans ta chaque occurrence de xi par ti.

Par exemple, si σ={x/A, y/x} alors σ(f(x,y))=f(A,x)

La composition σ1 o σ2 de deux substitutions est une substitution. Cette opération est associative et possède un élément neutre σ={}.

2.2.2. *Unification et Filtrage*

Deux termes t1 et t2 sont **unifiables** s'il existe une substitution σ telle que σ(t1)=σ(t2). On dit alors que σ est **unificateur** de t1 et t2.

Exemple : Soit t1=f(x,y), t2=f(u,A) et σ1={x/B, y/A, u/B}

$$\sigma1(t1) = \sigma1(t2) = f(B,A)$$

t1 et t2 sont unifiables, σ1 est un unificateur de t1 et t2.

Unifier deux termes consiste à trouver un unificateur de ces deux termes, ou indiquer qu'ils ne sont pas unifiables.

Une substitution circulaire est une substitution contenant un élément x/t tel que t contient au moins une occurrence de x. De telles substitutions ne peuvent jamais être des unificateurs. Le test dit **d'occurrence** (*occur check*) permet de les détecter et de provoquer l'échec de l'unification.

Il est possible de construire une infinité d'unificateurs de t1 et t2, comme par exemple σ2 = {x/A, y/A, u/A} qui est aussi unificateur de t1 et t2.

Un unificateur μm est dit **plus général unificateur** pour l'unification de t1 et t2 si, quelquesoit μ un unificateur de t1 et t2, il existe une substitution σ telle que μ = σ o μm.

Pour notre exemple, si μ = {x/u, y/A}, alors μ(t1) = μ(t2) = f(u,A).
μ est une substitution minimale pour l'unification de t1 et t2.

Si deux termes t1 et t2 sont unifiables, alors il existe un plus général unificateur μg (Théorème de l'unification). De plus, tout autre plus général unificateur peut être obtenu à partir de μg par changement de variables.

Une notion proche de l'unification est le **filtrage**. Un terme t1 filtre un terme t2 s'il existe une substitution σ telle que $\sigma(t1)=t2$. Dans le filtrage, le terme t2 doit donc être une instance du terme t1. Il diffère notamment de l'unification par le fait qu'il ne nécessite pas le test d'occurrence.

Nilsson *[Nilsson 71]* définit également la notion de **reconnaissance de forme** (*pattern-matching*) comme étant une restriction du filtrage, où le terme t2 doit être clos. La reconnaissance de forme est également une restriction de l'unification dans la mesure où quelque soit un unificateur σ, $\sigma(t) = t$ si t est un terme clos. De plus, elle ne nécessite pas non plus de test d'occurrence.

Le terme "pattern-matching" englobe en fait de nombreuses significations en fonction du domaine dans lequel il est utilisé. Par la suite, nous l'utiliserons dans le sens défini par Nilsson.

2.2.3. *L'unification en Prolog*

Les interprètes Prolog utilisent généralement la structure d'arbre pour représenter les termes. L'unification de deux termes t1 et t2 apparaît alors comme la construction, si elle est possible, d'un arbre qui est l'**instance commune la plus générale** des deux arbres t1 et t2.

Les algorithmes d'unification consistent par conséquent à trouver les valeurs que doivent prendre les variables présentes dans deux arbres pour que ceux-ci coïncident *[Colmerauer 83]*. Faire une unification équivaut d'en d'autres termes à résoudre un système d'équations ; si ce système d'équations est insoluble, les termes ne sont pas unifiables.

Par exemple, unifier t1=f(y, g(A, x)) et t2=f(x, g(y, B)) équivaut à résoudre le système d'équations:

$$y = x$$
$$y = A$$
$$x = B$$

qui n'a pas de solutions $\Rightarrow$ t1 et t2 ne sont pas unifiables

Peu d'interprètes mettent en oeuvre le test d'occurrence en raison du coût exponentiel de son utilisation. Toutefois, si le test d'occurrence provoque un échec de la résolution dans le strict cadre de la logique du premier ordre, A.Colmerauer a introduit la notion d'unification d'**arbres rationnels** (forme particulière d'arbres infinis) qui permet d'étendre cette logique et de prendre en compte de tels cas *[Colmerauer 83]*.

2.3. Résolution des clauses de Horn

L'exécution d'un "programme" composé d'un ensemble de clauses de Horn est une **réfutation** (preuve par l'absurde), utilisant le Principe de Résolution *[Robinson 65]* comme règle d'inférence.

Soit A un ensemble d'axiomes (clauses du programme) et Q une hypothèse que l'on veut prouver à partir de A. On peut soit dériver Q en appliquant la règle d'inférence à partir des clauses de A ($A \supset Q$), soit dériver la clause vide à partir de l'union de A et de la négation de Q.

$$A \supset Q \quad \Leftrightarrow \quad A, \neg Q \supset [\,]$$

La restriction du Principe de Résolution aux clauses de Horn nous conduit au fonctionnement suivant :

- Soit à réfuter la conjonction de littéraux (appelés aussi sous-buts) dotée d'une liste de substitutions Σ (initialement vide) :

$$L_1 \wedge \ldots \wedge L_{i-1} \wedge L_i \wedge L_{i+1} \wedge \ldots \wedge L_m \quad + \quad \Sigma$$

et la clause :

$$P \leftarrow Q_1, \ldots, Q_n.$$

- Si P est unifiable avec Li, il existe une substitution σ d'une variante de P, obtenue en renommant ses variables de manière à ce qu'aucune d'elles n'apparaisse dans la résolvante courante (conjonction de sous-buts), et des variables de Li les rendant identiques. On peut alors appliquer la substitution et construire la résolvante :

$$\sigma\,(L_1 \wedge \ldots \wedge L_{i-1} \wedge Q_1 \wedge \ldots \wedge Q_n \wedge L_{i+1} \wedge \ldots \wedge L_m) \quad + \quad \sigma \circ \Sigma$$

On dit alors que l'on **efface** le littéral Li.

- S'il n'existe pas une telle clause, la Résolution s'arrête et l'on peut affirmer que la réfutation initiale est vraie, c'est à dire que l'hypothèse est fausse.

La Résolution appliquée aux clauses de Horn est une règle d'inférence

qui se décompose donc en deux étapes:

- Choix d'un sous-but dans la résolvante
- Choix d'une clause dont l'entête est unifiable avec le sous-but considéré

Ces deux étapes forment un pas de résolution que l'on appelle **inférence logique**.

Prouver par réfutation une conjonction de littéraux revient à effacer tous les littéraux qui la composent, par applications successives du Principe de Résolution (dérivation de la clause vide). La substitution associée à la dernière résolvante (clause vide) représente alors **une** solution au problème.

On démontre *[Chang 73]* que la Résolution est **complète**, c'est à dire qu'elle génère toujours la clause vide à partir d'un ensemble insatisfaisable (contradictoire) de clauses ; autrement dit, si une question Prolog est "vraie" (démontrable au sens classique), il existe une suite finie d'inférences logiques qui permet de la démontrer. De plus, la résolution est **correcte**, c'est à dire qu'elle ne génère pas la clause vide à partir d'un ensemble satisfaisable (non contradictoire) de clauses ; autrement dit, si une question Prolog est "fausse", il n'existe pas de chemin qui permette de la démontrer.

2.4. Interprétation d'un programme Prolog

Le terme Prolog désigne historiquement la première réalisation, par l'équipe d'Alain Colmerauer à Marseille, d'un interprète de programmes écrits sous forme de clauses de Horn *[Roussel 75]*. La tendance actuelle consiste à généraliser ce terme pour identifier tous les langages basés sur ce formalisme, bien que les stratégies d'implémentation soient très souvent différentes. Par la suite, nous employons le terme Prolog dans ce sens.

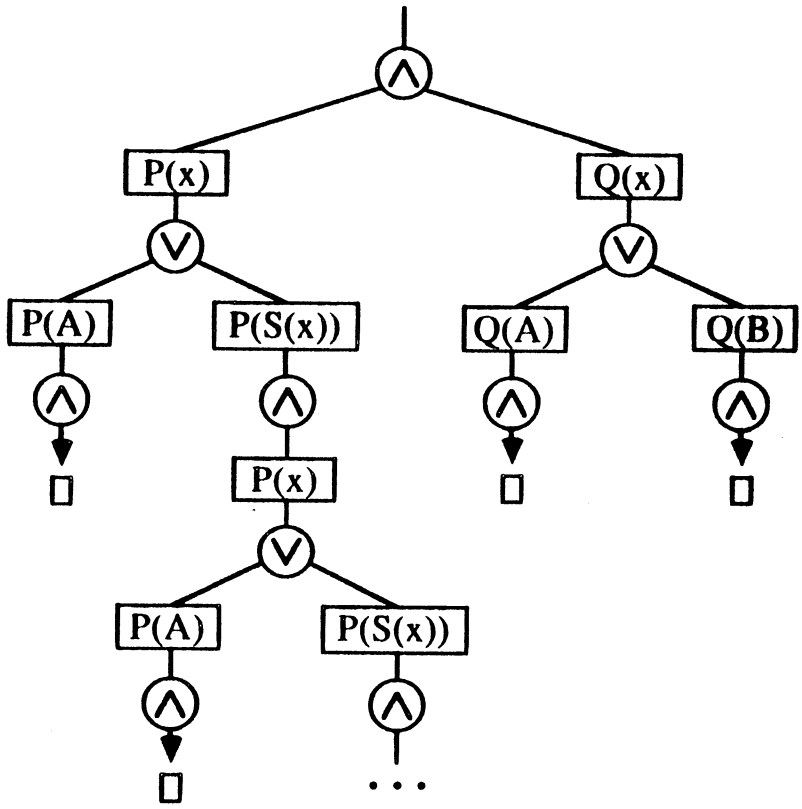
2.4.1. *Arbre de recherche*

Le mécanisme de résolution de Prolog effectue un parcours de l'**arbre et/ou**, éventuellement infini, défini statiquement par les clauses du programme *[Kowalski 79]*.

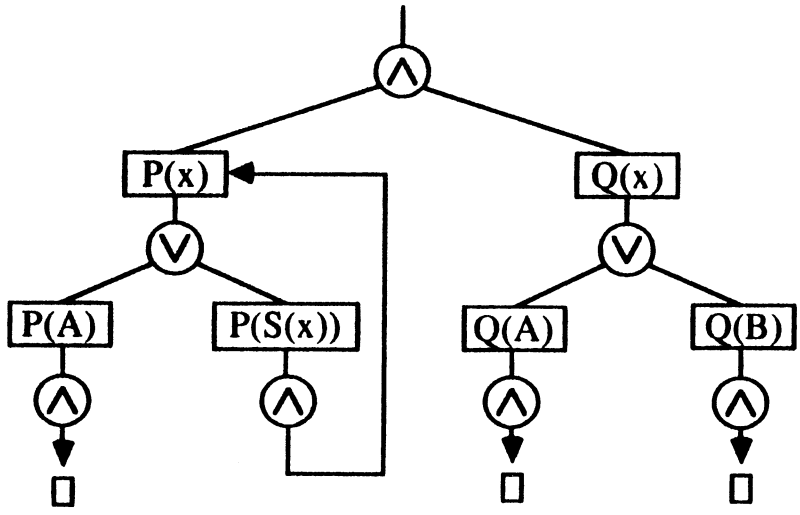
Exemple : soit le programme Prolog

$Q(A) \leftarrow .$
$Q(B) \leftarrow .$
$P(A) \leftarrow .$
$P(S(x)) \leftarrow P(x).$
$\leftarrow P(x), Q(x).$

L'arbre et/ou associé à ce programme est :



Le **graphe et/ou** est obtenu à partir de l'arbre et/ou en fusionnant les noeuds étiquettés par le même sous-but :

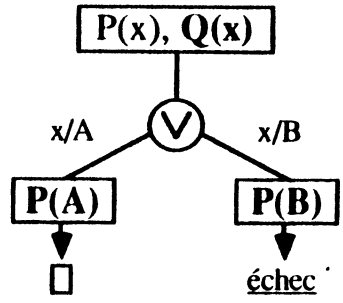


A chaque noeud du graphe (ou de l'arbre) et/ou est associé la sémantique suivante :

noeud et : Il est considéré comme vrai si tous ses fils le sont.
noeud ou : Il est considéré comme vrai si au moins l'un de ses fils l'est.

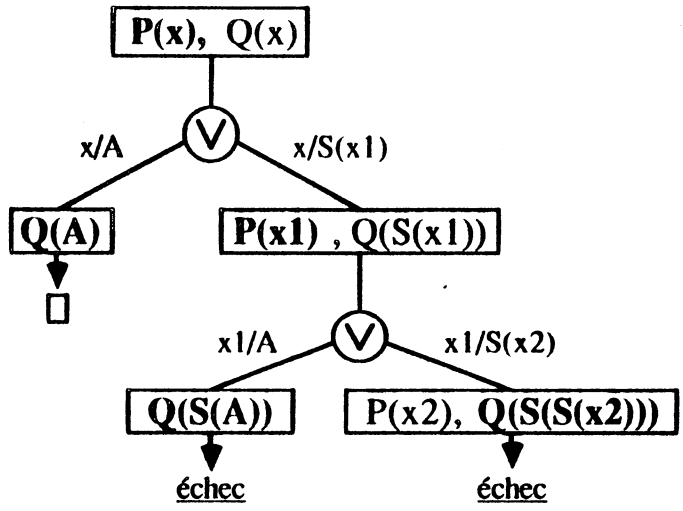
Un défaut important des arbres et/ou est qu'ils suggèrent une certaine indépendance entre la résolution des différents sous-buts d'une conjonction, ce qui est faux. En effet, la résolution simultanée de plusieurs sous-buts n'est possible que s'ils ne partagent aucune variable, de manière à ne pas instancier simultanément une même variable à deux termes incompatibles.

On préfère donc utiliser la notion d'**arbre de recherche** ; dans ces arbres, les noeuds sont étiquettés par une résolvante complète et les arcs par les substitutions réalisées pour passer d'un noeud à l'autre. L'arbre suivant est un arbre de recherche (le sous-but sélectionné dans la résolvante est placé en gras) correspondant à notre exemple :



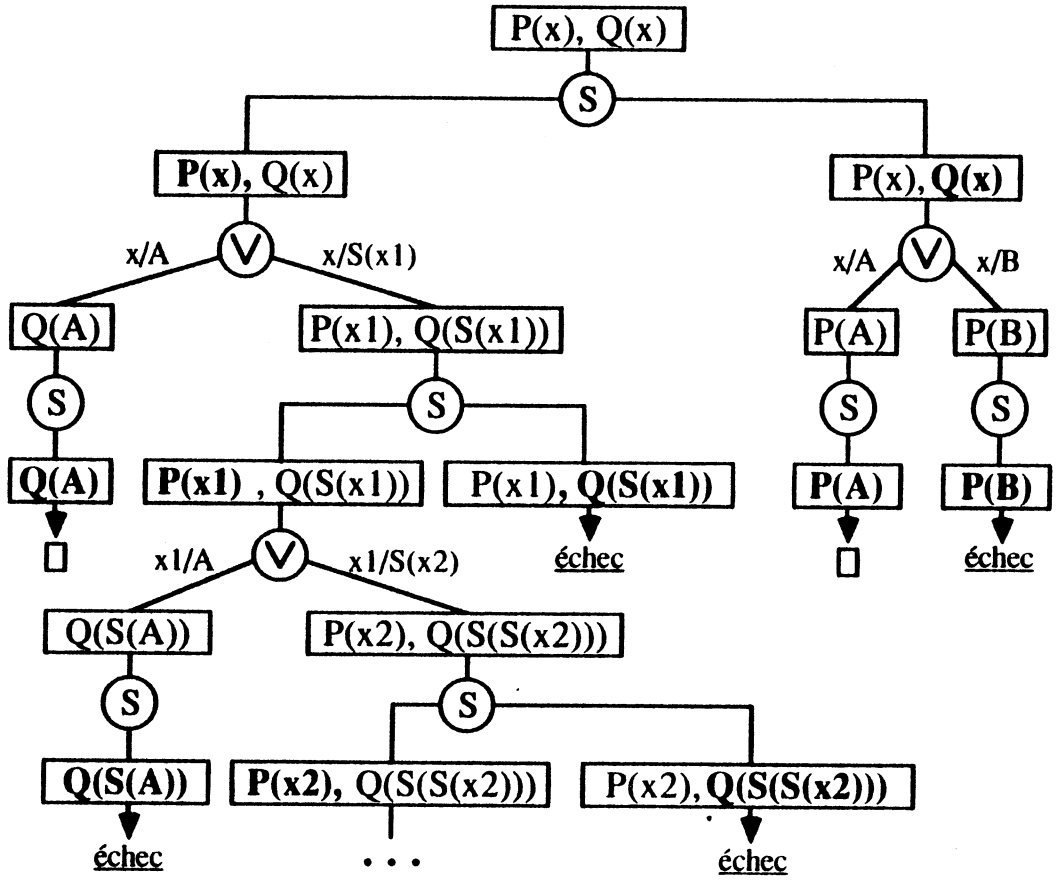
Cette représentation contient en elle-même la notion de séquencement du traitement des sous-buts de la résolvante, ce que l'arbre et/ou ne fait pas apparaître. Dans l'arbre de recherche précédent, nous avons par exemple

arbitrairement choisi de toujours substituer le dernier sous-but. En réalité, le choix est toujours possible ; ainsi, l'arbre suivant est également un arbre de recherche :



Chaque clause vide apparaissant dans un arbre de recherche représente une manière possible de satisfaire l'hypothèse formulée par la résolvante présente au sommet de cet arbre. Les instances des variables pour lesquelles la satisfaction est atteinte peuvent être obtenues en effectuant la composition des substitutions apparaissant sur les branches menant aux clauses vides. Notre arbre ne contient ici qu'une seule clause vide, représentant la solution x=A.

On peut définir l'**arbre général de recherche** comme l'union de tous les arbres de recherches possibles, cette union étant représentée par l'intermédiaire de noeuds spéciaux S (choix de sous-but). Pour notre exemple, on obtient :



Cet arbre reflète particulièrement bien les deux étapes de la Résolution : choix d'un sous-but dans la résolvante (noeud S) et choix d'une clause dont la tête s'unifie avec le sous-but (noeud V). La sémantique liée à ces deux types de noeuds de l'arbre général de recherche est la suivante :

noeud (S)

Le choix d'une seule des branches est suffisant car chaque sous-arbre contient en lui-même autant d'informations que les autres. Notamment, si l'ensemble des clauses est insatisfaisable, la clause vide est accessible dans chacun de ces sous-arbres. Par contre, les chemins d'accès à cette clause vide peuvent avoir des longueurs différentes. Pour notre exemple, le sous-arbre droit de la racine est fini alors que le gauche est infini.

noeud (V)

Le parcours de toutes les branches est parfois nécessaire car la clause vide, si elle existe, peut n'apparaître que dans l'une d'entre elles.

Un arbre de recherche est obtenu à partir de l'arbre général de recherche en fixant le choix d'une branche sur chaque noeud S atteint en parcourant toutes les branches d'un noeud V.

2.4.2. *Mise en oeuvre de la Résolution*

Implémenter la Résolution consiste à parcourir un arbre de recherche pour trouver une clause vide. En fait, il s'agit de construire cet arbre et de le parcourir simultanément (chaque noeud parcouru, ou construit, représente une inférence logique). Pour d'évidentes raisons d'efficacité, il faut parcourir cet arbre de la manière la plus "économique" possible.

L'utilisation de Prolog en tant que langage de programmation nécessite la prise en compte des contraintes suivantes :

a) *Acquisition des valeurs*

Il est clair qu'un utilisateur ne désire pas seulement savoir si son programme est vrai (insatisfaisable) ou faux (satisfaisable) mais désire également obtenir les valeurs des variables pour lesquelles il est vrai. Il est donc intéressant de rendre sous une forme lisible la liste des substitutions obtenue lorsque tous les littéraux ont été effacés.

b) *Acquisition de l'ensemble des solutions*

L'utilisation la plus générale de Prolog (ex. bases de données) s'intéresse à l'*ensemble* des valeurs des arguments pour lesquelles le programme est vrai. Ceci conduit à fournir un moyen de parcourir exhaustivement l'arbre de recherche pour atteindre **toutes** les clauses vides.

c) *Connaissance du processus de résolution*

Pour permettre la communication avec le monde extérieur, Prolog est doté d'opérateurs d'entrées/sorties, mis en oeuvre sous forme de prédicats prédéfinis. La résolution de ces prédicats dans un ordre quelconque n'est évidemment pas sans conséquences ; il est donc nécessaire que le programmeur connaisse le séquencement du processus de résolution pour maîtriser ces opérateurs.

Il est nécessaire de prendre en compte ces contraintes dans la définition d'une stratégie de parcours de l'arbre de recherche.

2.4.3. *Stratégies de parcours de l'arbre de recherche*

Le Principe de Résolution autorise deux degrés de liberté : le choix du sous-but à effacer et le choix de la clause qui permet de l'effacer. Pour fournir le contrôle de séquentialité nécessaire à une implémentation réaliste, la plupart

des interprètes sont basés sur un effacement séquentiel, dans l'ordre d'écriture des sous-buts (gauche-droite). Les seuls choix restants à effectuer se situent donc au niveau des branches **ou**. Dans ce cadre, on distingue notamment les stratégies suivantes :

profondeur d'abord (*Depth First*)

L'arbre de recherche est parcouru en profondeur en prenant une seule clause à la fois ; si la résolution n'aboutit pas à une solution, on revient à ce point de choix grâce au **retour arrière** (*backtrack*), on choisit une autre clause, et on recommence. En général, le choix des clauses s'effectue dans l'ordre de leur écriture. Une autre solution utilisée consiste à choisir en priorité les clauses de type 3 (les faits) puisqu'elles sont les seules à pouvoir effacer définitivement un littéral.

largeur d'abord (*Breadth First*)

On développe en même temps toutes les branches **ou** de l'arbre. L'arbre de recherche est donc construit en largeur, un niveau de cet arbre étant complètement construit avant le traitement du niveau suivant.

La plupart des interprètes utilisent la même stratégie (que l'on appellera par la suite "stratégie classique") basée sur les trois points suivants :

- Sélection des sous-buts dans l'ordre de leur écriture.

- Sélection des clauses dans l'ordre de leur écriture.

- Parcours de l'arbre de recherche en profondeur d'abords.

On peut remarquer que cette stratégie n'assure pas la complétude autorisée par le Principe de Résolution. En effet, un prédicat récursif à gauche peut génèrer une branche infinie à gauche de l'arbre de recherche, dont le parcours empêche de traiter l'ensemble des choix possibles. Ce problème n'apparaît pas dans la stratégie *Breadth First* mais son coût prohibitif rend son utilisation malaisée.

2.4.4. *Interprétation procédurale d'une clause de Horn*

Kowalski a introduit la notion d'interprétation procédurale *[Kowalski 74]*, dans laquelle une clause $P \leftarrow Q1,...,Qn$ est considérée comme une procédure de nom P effectuant un ensemble d'appels aux procédures $Q1,...,Qn$. Cette notion associe à la résolution d'une résolvante une sémantique particulière :

Soit la résolvante : $L_1 \wedge ... \wedge L_{i-1} \wedge P \wedge L_{i+1} \wedge ... \wedge L_m$

et la clause $P \leftarrow Q_1,..,Q_n$.

Si l'on efface le sous-but P en le substituant par la queue de la clause, on obtient la nouvelle résolvante :

$$L_1 \wedge ... \wedge L_{i-1} \wedge \mathbf{Q_1} \wedge ... \wedge \mathbf{Q_n} \wedge L_{i+1} \wedge ... \wedge L_m$$

Une **interprétation procédurale** effectue l'effacement complet de tous les sous-buts Qi avant de tenter d'effacer l'un des sous-buts Lj. En quelque sorte, la queue de clause substituée à P est une sous-résolvante qu'il faut résoudre avant de poursuivre.

On peut constater que la stratégie classique de Prolog est un cas particulier de l'interprétation procédurale.

3. Le contrôle en Prolog

Le passage du modèle logique présenté dans les paragraphes précédents à un langage de programmation à part entière n'est pas aisée. Différents problèmes doivent être résolus :

- Une stratégie de parcours de l'arbre de recherche systématique et invariable, si elle offre l'avantage de la simplicité de mise en oeuvre et de précision de déroulement d'un programme, implique l'impossibilité d'éviter de parcourir une branche infinie, bloquant ainsi le processus de résolution. L'une des conséquences directes est également une perte de la réversibilité des prédicats.

- Le modèle logique est très insuffisant pour aborder un grand nombre de problèmes réels auxquels peut être confronté le programmeur. Ainsi, il est impossible d'exprimer des négations.

Pour éviter la première catégorie de problèmes, les interprètes Prolog offrent des opérateurs permettant un contrôle de la stratégie. Une utilisation astucieuse de ces opérateurs permet de plus de combler certaines déficiences du modèle logique. Ces opérateurs de contrôle représentent en quelque sorte une méta-connaissance qui doit être associée à la connaissance d'un problème pour permettre sa résolution dans de bonnes conditions. Deux approches se distinguent *[Lebaube 86]* :

- Le *contrôle intégré*, où l'information de contrôle est mélangée au programme sous la forme de prédicats extra-logiques. C'est l'approche traditionnelle, qui offre l'avantage de la simplicité et de l'efficacité. Elle se traduit toutefois par une forte dépendance vis à vis de la stratégie et par une altération de la définition logique du programme de par la présence de l'information de contrôle.

- Le *contrôle séparé*, où l'information de contrôle est dissociée du programme lui-même et peut être considérée comme une base de méta-connaissances décrivant les contraintes à respecter lors de la résolution des prédicats *[Gallaire 82]*. Cette approche, beaucoup plus générale, évite la plupart des inconvénients du contrôle intégré mais à un coût parfois prohibitif.

Les implantations actuelles de Prolog utilisent en grande majorité la stratégie *depth first* avec sélection des sous-buts et des clauses dans l'ordre de leur écriture. Pour cette raison, nous allons présenter les opérateurs de contrôle dans le cadre de cette stratégie, en les classant en deux grandes catégories :

- Les opérateurs de contrôle du choix de clause qui, alliés aux particularités de la stratégie *depth first*, permettent notamment de définir une approche de la négation : la *négation par l'échec*.

- Les opérateurs de contrôle du choix des sous-buts, qui contribuent à améliorer la terminaison de la résolution (en évitant certaines branches infinies), et notamment la correction de la négation par l'échec.

3.1. Contrôle du choix de clause

3.1.1. *L'échec*

L'utilisation de l'échec comme outil de contrôle de la stratégie n'a de sens que dans le cadre d'un parcours *depth first* de l'arbre de recherche. Il consiste à utiliser le prédicat *fail*, qui retourne toujours faux, et par là même provoque le retour arrière du processus de résolution.

Le prédicat *fail* lui-même ne sort pas de la logique du premier ordre puisqu'il peut être défini simplement en ne faisant apparaître aucune clause pour sa définition. Toutefois, sa liaison avec le retour arrière fait qu'il est bien souvent prédéfini dans les implantations de Prolog.

Nous donnerons des exemples de son utilisation par la suite dans la mesure où il est généralement utilisé en association avec la coupure.

3.1.2. *La coupure*

La coupure, plus généralement appelée "cut" et notée "!", est l'opérateur de contrôle intégré le plus anciennement introduit dans le langage. Il est très important et largement utilisé dans la programmation. Sa définition, que nous empruntons à *[Colmerauer 83]* est la suivante :

> *L'effacement du cut a pour effet de supprimer tous les choix en attente pour tous les termes à effacer à partir de celui qui a activé la règle contenant le cut jusqu'à celui qui précède le cut dans la queue de cette règle.*

C'est donc un opérateur de très bas niveau puisqu'il ne porte que sur le séquencement de la résolution, indépendamment de toutes propriétés des substitutions réalisées.

Soit par exemple le programme Prolog :

P(A) ←.
P(B) ←.

Q(x, y) ← P(x), P(y).
R(x) ← P(x).
R(C) ←.

Q'(x, y) ← P(x), !, P(y).
R'(x) ← P(x), !.
R'(C) ←.

Les différents comportements se reflètent dans le tableau suivant :

réfutation	n-uplets solutions
←Q(x, y).	(A,A), (A,B), (B,A), (B,B)
← Q'(x, y).	(A,A), (A,B)
← R(x).	(A), (B), (C)
← R'(x).	(A)
← R'(C).	(C)

Cet opérateur ne s'intègre pas dans le cadre de la logique du premier ordre comme le montrent les deux dernières lignes du tableau. En effet, R'(C) est vrai alors que la question ←R'(x) ne retourne pas la solution x=C. L'usage de cet opérateur sans discrimination peut donc retirer tout son sens logique à un programme Prolog. Par contre, il permet notamment :

- d'éviter le parcours de branches de l'arbre de recherche jugées inutiles par le programmeur. Ceci permet d'améliorer notablement le temps nécessaire à la résolution d'un problème donné et peut permettre d'éviter le parcours de branches infinies.

- d'introduire des opérateurs inexistants dans le modèle logique, comme la négation (appelée *négation par l'échec*), en jouant sur les particularités de la stratégie de parcours de l'arbre de recherche.

3.1.2.1. *Elimination de branches inutiles*

L'utilisation pratique de la coupure consiste essentiellement à simuler le fonctionnement de la structure de contrôle si-alors-sinon par la transformation :

P = **si** A **alors** B	→	P ← A, !, B.
sinon C		P ← C.

La coupure peut donc permettre une programmation efficace en limitant le parcours de branches inutiles dans l'arbre de recherche. Ainsi, la programmation de la concaténation de deux listes, qui s'écrit normalement :

append (a.x, y, a.z) ← append (x, y, z).
append (NIL, x, x) ←.

peut s'écrire plus efficacement en :

append (a.x, y, a.z) ← !, append (x, y, z).
append (NIL, x, x) ←.

En supposant que l'on effectue la résolution de

← append(A.B.C.NIL, D.NIL, x).

la première version va tenter 4 unifications avec chaque tête de clause (soit 8 au total) alors que la version "optimisée" ne va tenter que 4 unifications avec la première tête de clause et une seule avec la deuxième (soit 5 au total).

Toutefois, une conséquence de l'utilisation du prédicat *cut* est une perte importante de la "réversibilité" des prédicats. Ainsi, la résolution de :

← append(x, y, A.NIL).

retourne pour x et y les solutions (A.NIL, NIL) et (NIL, A.NIL) pour la première version, alors qu'elle ne retourne que la solution (A.NIL, NIL) pour la version "optimisée".

Dans un certain sens, l'usage du *cut* dirige implicitement la résolution des prédicats vers une utilisation particulière ; la concaténation de listes dans sa version "pure" peut également servir à déterminer toutes les paires de listes dont la concaténation fournit une liste donnée. La version "optimisée" ne

permet plus d'effectuer cette opération dans toute sa généralité.

La plus grande utilisation de la coupure étant la simulation de la structure si-alors-sinon, certaines études *[O'Keefe 85]* proposent d'introduire explicitement cette structure dans Prolog, ce qui présente notamment l'avantage de permettre un meilleur traitement pour les techniques d'optimisation de haut niveau.

Une autre utilisation de la coupure permet de simuler des boucles *tantque* en utilisant la récursivité. Programmons par exemple la boucle *read-eval-print* d'un interprète Lisp. Une première approche pourrait être :

```
lisp ← read(x), eval(x).

eval(QUIT) ← !, print("Au revoir").
eval(x) ← eval(x, y), print(y), lisp.
```

Le problème de cette mise en oeuvre réside dans le fait que la récursivité provoque, au niveau de l'implantation, l'empilement d'un environnement important (ensemble des valeurs des variables des clauses) à chaque appel récursif. On peut utiliser à la place la méthode suivante :

```
repeat ←.
repeat ← repeat.

lisp ← repeat, read(x), eval(x, y), print(y), eq(x, QUIT), !,
        print("Au revoir").
```

Dans cette version, tant que le prédicat eq(x, QUIT) est faux, le retour arrière revient jusqu'au point de choix précédent, c'est à dire le *repeat*. Au moment où eq(x, QUIT) devient vrai, le *cut* qui le suit est franchit et le point de choix du *repeat* est annulé, provoquant ainsi l'arrêt de la boucle.

Cette méthode offre l'avantage de n'empiler que l'environnement du prédicat *repeat*, qui est en fait vide puisqu'il ne contient aucune variable. Cette utilisation de *repeat*, en tant que générateur d'une infinité de points de retour arrière, fait que ce prédicat est bien souvent prédéfini dans les interprètes Prolog.

3.1.2.2. *Négation par l'échec*

La négation par l'échec est une utilisation courante de l'opérateur de coupure. Elle consiste à supposer que ce qui n'a pas été déclaré ou prouvé vrai

est nécessairement faux (hypothèse du *monde clos*). Elle se programme en utilisant la structure si-alors-sinon de la manière suivante :

non-P =	**si** P **alors** faux	→	non-P ← P, !, *fail.*
	sinon vrai		non-P ←.

La programmation de l'inégalité de deux termes devient donc :

```
eq(x, x) ←.
neq(x, y) ← eq(x, y), !, fail.
neq(x, y) ←.
```

Le fonctionnement de ce programme reste cohérent tant que les deux arguments sont des termes clos. S'ils ne le sont pas, on peut aboutir à de graves incohérences, illustrées par l'exemple suivant :

← eq(x, A), eq (y, B), neq(x, y).	réussit, ce qui est normal.
← neq(x, y), eq(x, A), eq (y, B).	échoue, ce qui ne l'est pas.

3.2. Contrôle du choix de sous-but

Les opérateurs de contrôle du choix des sous-buts ont été définis pour tenter d'apporter une solution aux problèmes suivants :

- La réversibilité des prédicats, qui peut être diminuée soit par la stratégie de résolution, soit par l'utilisation du *cut*, comme dans l'exemple de *append* donné précédemment.

- La correction des prédicats définis à base du *cut*, comme la négation par l'échec.

- La correction de certains prédicats prédéfinis comme les prédicats arithmétiques qui demandent pour leur utilisation un minimum de connaissances. Ainsi, le prédicat plus(x, y, z), qui définit la relation z=x+y, nécessite qu'au moins deux arguments soient connus.

De nombreuses approches ont été suivies pour pallier ces problèmes ; nous ne décrirons ici que les plus représentatives. Les opérateurs proposés se décomposent en deux catégories :

- Les opérateurs qui conservent l'interprétation procédurale. Le contrôle se restreint dans ce cas à un réordonnancement des littéraux des queues de clauses. C'est le cas des opérateurs *var* et *novar*.

- Les opérateurs ne conservant pas l'interprétation procédurale. Dans ce cas, l'effacement d'un sous-but peut être effectué bien au delà de la queue de clause où il apparait. C'est le cas des opérateurs *geler* et *wait*.

3.2.1. *Les opérateurs Var et Novar*

Soit le programme suivant :

reverse(a.x, r) ← reverse(x, y), append(y, a.NIL, r).
reverse(NIL, NIL) ←.

et les deux buts :

← reverse(A.B.NIL, x). (1)
← reverse(x, A.B.NIL). (2)

La résolution du premier but se termine bien en produisant la solution x=B.A.NIL. Par contre, la résolution du deuxième ne se termine pas car la sélection du sous-but reverse(x, y) en premier crée une branche infinie à gauche dans l'arbre de recherche, que la stratégie *depth-first* va essayer de parcourir. Inversement, si la programmation de ce prédicat est :

reverse(a.x, r) ← append(y, a.NIL, r), reverse(x, y).
reverse(NIL, NIL) ←.

le processus de résolution boucle pour le premier but et se termine correctement pour le second.

On constate donc que l'ordre de sélection des sous-buts doit être modifié en fonction du type d'appel que l'on effectue.

Les opérateurs *var* et *novar* (parfois appelés *free* et *nofree*) offrent une solution simple à ce type de problème. Ce sont des méta-prédicats dont la sémantique est la suivante :

novar(x) retourne vrai si l'argument x est soit une constante, soit un terme structuré. On dit alors que x est **lié**.

var(x) retourne vrai si x n'est pas une constante où un terme structuré. On dit alors que x est **libre**.

Notre prédicat *reverse* peut être programmé en utilisant ces opérateurs pour permettre sa réversibilité. Il devient alors :

reverse(a.x, r) ← *var*(x), append(y, a.NIL, r), reverse(x, y).
reverse(a.x, r) ← *novar*(x), reverse(x, y), append(y, a.NIL, r).
reverse(NIL, NIL) ←.

La première clause de *reverse* est remplacée par deux clauses dont chacune s'applique à un contexte d'utilisation différent. Les opérateurs *var* et *novar* sont utilisés pour sélectionner l'une ou l'autre, en fonction du contexte dans lequel est réellement appelé le prédicat.

Ces deux opérateurs ont l'avantage d'offrir un certain contrôle du choix des sous-buts, local à la queue d'une clause, sans provoquer de changement dans l'interprète. Toutefois, ils obligent le programmeur à écrire plusieurs versions d'un même prédicat en fonction des différents modes d'utilisation.

3.2.2. *L'opérateur Geler*

Le *geler* a été introduit dans Prolog-II *[Colmerauer 82]* et pallie notamment au problème des copies multiples d'un même prédicat. Le *geler* permet de retarder l'effacement d'un sous-but en attendant l'instanciation d'une variable particulière. L'effacement du sous-but *geler*(v, t), où v est une variable et t un sous-but, s'effectue de la manière suivante :

- Si la variable v est liée, alors on efface le but t de manière normale.

- Si la variable v est libre, le but *geler* est effacé mais l'effacement de t est retardé jusqu'au moment où v sera instanciée à un terme autre qu'une variable (c'est à dire quand *novar*(v) deviendra vrai).

L'utilisation du *geler* provoque donc un profond changement de la stratégie de sélection des sous-buts qui doit tenir compte explicitement des directives fournies par le *geler*. Toutefois, contrairement à *var* et *novar*, il permet de sortir de l'interprétation procédurale. Soit par exemple le programme suivant :

permutation(x1, x2, x3) ← différents(x1, x2, x3), objet(x1),
objet(x2), objet(x3).

différents(x1, x2, x3) ← diff(x1, x2), diff(x2, x3), diff(x1, x3).

diff(x, y) ← *geler*(x, *geler*(y, neq(x, y))).

objet(A) ←. objet(B) ←.

objet(C) ←. objet(D) ←.

où le prédicat *permutation* définit les permutations possibles de 3 objets parmi A, B, C et D.

La résolution du but ←permutation(x1,x2,x3) en tenant compte des directives *geler* s'effectue alors avec un effacement des sous-buts dans l'ordre suivant :

objet(x1), objet(x2), **diff**(x1, x2), objet(x3), **diff**(x2, x3), **diff**(x1, x3).

On peut constater que l'interprétation procédurale n'est plus vérifiée puisqu'un sous-but *diff* a été placé entre deux sous-buts *objet*.

Nous avons introduit dans cet exemple le prédicat *diff* qui vérifie l'inégalité de deux termes sur la base du prédicat *neq*, fondé lui-même sur la négation par l'échec. Nous avons vu précédemment que le prédicat *neq* ne produit des résultats corrects que si ses deux arguments sont des termes clos. Or, le *diff* retarde la résolution du *neq* en vérifiant seulement que ces deux arguments ne sont pas des variables libres ; ceci est insuffisant dans un contexte général. Dans notre exemple de la permutation, le problème ne se présente pas car on est certain que les arguments, produits par le prédicat *objet*, sont des termes clos.

Une programmation correcte du *diff* implique donc un parcours récursif des termes pour tester l'inégalité. Sa programmation en Prolog n'étant pas aisée, Prolog-II par exemple l'incorpore en tant que prédicat prédéfini.

3.2.3. *L'opérateur Wait*

L'opérateur *wait* (attente), introduit par L.Naish dans Mu-Prolog *[Naish 85]*, *[Naish 86]*, représente une généralisation du *geler*, abordé en vue du contrôle séparé. L'utilisation de cet opérateur est donc basée sur une base de méta-connaissance qui contient dans ce cas des *déclarations d'attente*.

Une déclaration d'attente est une déclaration de la forme *wait*(P(a1,...,an)) où P est un nom de prédicat et ai une valeur parmi {0,1}. Il peut y avoir plusieurs déclarations d'attente pour un même prédicat.

Lorsque un but P(x1,...,xn) va être effacé, il est préalablement unifié avec une tête de clause. Un argument xi de ce but est dit *construit* si dans cet argument apparaît une variable unifiée avec un terme non-variable. Après l'unification, on peut obtenir l'ensemble des arguments *construits*. L'effacement du but est alors autorisé si il existe une déclaration *wait*(P(a1,...,an)) telle que :

$$\forall xi \textit{ construit},\ ai=1$$

Dans le cas contraire, l'unification est annulée et l'effacement du but est retardé jusqu'à ce qu'il vérifie cette condition. Si aucune déclaration d'attente ne porte sur le prédicat, l'effacement est toujours accepté.

Pour le prédicat *append*, les déclarations d'attente ont la forme :

wait(append(1,1,0)), *wait*(append(0,1,1)).
append (a.x, y, a.z) ← append (x, y, z).
append (NIL, x, x) ←.

Ainsi, les buts ←append(x, B.NIL, A.B.NIL) et ←append(A.NIL, B.NIL, x) seront acceptés alors que le but ←append(x, B.NIL, y) sera retardé.

La réversibilité du prédicat *reverse* est également assurée grâce aux déclarations d'attente sur *append*.

reverse(a.x, r) ← append(y, a.NIL, r), reverse(x, y).
reverse(NIL, NIL) ←.

Le *wait* présente des avantages multiples sur le *geler*. D'une part, il évite de préciser le contrôle dans chaque clause susceptible d'entraîner le processus de résolution dans une boucle infinie et, d'autre part, une production automatique des déclarations d'attente *[Naish 85]* est possible et peut permettre au programmeur de se consacrer entièrement à l'exposé logique de son problème, sans se préoccuper de l'aspect contrôle.

La production automatique des déclarations d'attente s'effectue en observant les différences entre l'entête du prédicat et son éventuel appel récursif. Si un argument de l'appel récursif est *plus général* que le même argument de l'entête (c'est à dire constitué d'une partie de ce dernier), il s'agit d'un argument dit *critique*. La production des déclarations d'attente est alors telle qu'elles n'autorisent la construction que d'un seul argument critique à la

fois. Les deux déclarations d'attente du prédicat *append* données ci-dessus sont donc facilement produites.

On peut toutefois constater quelques insuffisances. Ainsi, par exemple, il est impossible de définir des déclarations d'attente pour le prédicat *reverse* de manière à ce que les buts reverse(A.NIL,x), reverse(x,A.NIL) et reverse(A.NIL,B.NIL) soient effacés immédiatement et que, par contre, reverse(x,y) soit retardé. De plus, la production des déclarations d'attente des prédicats dont la récursivité n'est pas directe est beaucoup plus complexe.

4. Conclusion

Le langage Prolog est donc un langage potentiellement très puissant mais qui souffre, dans sa mise en oeuvre, de plusieurs défauts. Ceux-ci sont liés pour une part à l'insuffisance du modèle logique lui-même, qui rend notamment difficiles les communications avec le monde extérieur, et pour une autre part à la trop grande séparation entre la partie logique d'un programme et sa partie contrôle.

Les opérateurs de contrôle explicite ont permis de résoudre la plupart de ces problèmes. Toutefois, leur utilisation est très malaisée, en raison de leur trop grande dépendance vis à vis de la stratégie de parcours de l'arbre de recherche utilisée par l'interprète.

Les opérateurs de contrôle séparé semblent plus prometteurs car ils permettent de bien distinguer l'exposé du problème de sa partie contrôle. Toutefois, une bonne utilisation de ceux-ci devrait s'effectuer sans l'intervention de l'utilisateur. Le problème du contrôle se substitue alors par celui de la production automatique de ces opérateurs.

CHAPITRE 2

Techniques d'optimisation de la Résolution

Si Prolog s'est révélé un langage idéal pour le prototypage, son manque d'efficacité a été le frein principal à son utilisation pour des applications réalistes. Ce besoin d'efficacité sans cesse croissant pose le problème crucial de l'optimisation de la résolution ; différents travaux ont été menés dans ce but.

Historiquement, les premiers travaux ont porté sur l'amélioration des interprètes, et ensuite sur les compilateurs. Le résultat principal a été la définition d'une machine abstraite adaptée à la résolution de Prolog, la "Warren Abstract Machine" *[Warren 77]*, cible de la plupart des compilateurs actuels. Cette machine permet la prise en compte d'optimisations élémentaires comme l'optimisation de l'appel terminal *[Warren 80]* ou la spécialisation de l'unification en fonction de contextes précis.

Les travaux les plus récents portent en grande partie sur une amélioration de la gestion mémoire par récupération de l'espace inutile *[Bruynooghe 82]*. Le projet Mali *[Bekkers 85]* propose une solution matérielle à ce problème grâce à une carte mémoire dotée d'un ramasse-miette qui fonctionne en parallèle avec le processus de résolution. Toutefois, les optimisations possibles dans le cadre de la machine de Warren et de la compilation "naïve" y correspondant semblent avoir atteint leurs limites.

Les travaux les plus récents portent sur l'amélioration de la compilation elle-même. Il s'agit essentiellement :

- d'identifier les propriétés d'un programme qui peuvent être source d'optimisations,

- de déterminer statiquement ces propriétés pour un programme donné ou bien de transformer celui-ci de façon à les faire apparaître.

Un exemple d'une telle transformation est la conversion d'un prédicat récursif linéaire en un prédicat récursif terminal *[Azibi 86]* ; l'optimisation de l'appel terminal a alors pour effet de le transformer en programme quasiment itératif.

Ces propriétés permettent principalement :

- de connaître statiquement les principales caractéristiques des termes à différents points du programme ; ceci permet de spécialiser l'opérateur d'unification en fonction de ces caractéristiques. C'est en grande partie l'objectif des *modes* et des *types*.

- de résoudre statiquement les problèmes de choix de sous-but, en recherchant un ordre qui conduit à un coût de résolution minimal. Ceci peut être obtenu notamment en éliminant le parcours de branches infinies.

Ce chapitre aborde essentiellement ces deux aspects. Nous présentons tout d'abord les différents points critiques d'un interprète classique de Prolog. Nous décrivons alors plus particulièrement les optimisations portant sur l'unification elle-même, en utilisant pour cela la notion d'expansion. Nous exposons enfin les bases permettant de sélectionner un meilleur ordre de résolution des sous-buts, problème traité de façon plus complète dans le chapitre suivant.

La plupart de ces optimisations s'appliquent en supposant, naturellement, que le programme ne se modifie pas dynamiquement par l'intermédiaire de "assert" (ajout dynamique de clause) ou "retract" (retrait dynamique de clause). Nous ferons la même supposition dans la suite de cette thèse.

1. Les constituants d'un interprète Prolog

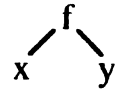
Un interprète classique de Prolog est composé des parties suivantes :

- L'unificateur, qui effectue l'unification d'un but avec un entête de clause.

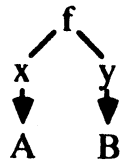
- Le moteur d'inférence lui-même qui s'occupe de sélectionner un but, et une clause pour tenter son effacement. Dans la plupart des cas, ce choix s'effectue dans l'ordre d'écriture du programme.

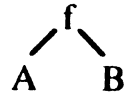
- Le retour arrière, qui remonte dans l'arbre de recherche jusqu'au plus récent point de choix pour parcourir un autre chemin.

La réalisation du retour arrière nécessite le moyen de retrouver l'environnement du point de choix où l'on retourne ; cet environnement est composé de la résolvante courante et de la substitution correspondant à ce point. Pour ce faire, les substitutions sont généralement réalisées par des chaînages liant les variables à leur valeur. Le retour arrière n'a donc qu'à défaire ces liens lors du retour à un point de choix.

Exemple : soit T le terme 

Si une unification provoque la substitution {x/A, y/B}, le terme résultant est :



alors qu'il pourrait être 

Cette caractéristique implique qu'un terme n'est pratiquement jamais directement accessible ; la valeur réelle d'une variable ne peut être obtenue qu'en parcourant les chainages pour aboutir aux feuilles. Cette opération s'appelle le *déréférençage*.

Des mesures comportementales de l'interprétation de Prolog *[Berger 86]* permettent de mettre en évidence les coûts relatifs des différentes opérations réalisées. Les résultats, acquis pour une version améliorée de Prolog I, montrent les proportions suivantes :

30% dans le *déréférençage*, essentiellement utilisé par l'unification,
25% dans l'unification elle-même,
13% dans le retour arrière,

12% dans le moteur de l'interprète,
et le reste dans diverses procédures, dont les prédicats prédéfinis.

Ces mesures peuvent bien sûr varier d'une implantation à une autre, mais elles montrent toutefois l'importance de l'unification qui, avec le déréférençage qu'elle utilise, occupe plus de la moitié du temps de résolution.

Cette constatation implique naturellement que l'unification est le premier point sur lequel doit porter une optimisation locale. La suite présente diverses techniques pour en optimiser l'algorithme en fonction du contexte d'utilisation, puis nous verrons dans le chapitre suivant comment le déréférençage lui-même peut être éliminé.

2. Optimisation de l'unification

L'unification est, par nature, un mécanisme très général dont le coût influence directement le coût de la résolution dans son ensemble ; c'est pourquoi la plupart des techniques d'optimisation s'orientent vers la spécialisation de l'opérateur d'unification en fonction du contexte d'utilisation. Parmi ses nombreuses fonctions, l'unification assure :

a) L'appel de procédure (sélection d'une clause et passage des paramètres).

b) L'unification d'arguments, c'est à dire :
- sélection de parties de termes structurés,
- construction de termes structurés,
- instanciation et affectation.

Le coût de l'unification peut être notablement réduit par l'utilisation d'opérateurs spécialisés dans chacune de ces tâches.

La partie "sélection d'une clause" peut permettre des gains importants en réduisant à l'avance le nombre de clauses avec lesquelles l'unification doit être tentée.

Dans le domaine de l'unification des arguments, la connaissance de certaines propriétés des termes permet d'utiliser des algorithmes d'unification plus spécifiques. Deux classes de propriétés, complémentaires et étroitement liées, sont source d'optimisations importantes :

- les **modes**, qui précisent le **sens** des flux de données (quels sont les termes qui consomment l'information fournie par d'autres termes),

- les **types** qui précisent la **structure** de ces termes.

L'usage d'opérateurs d'unification spécifiques nécessite la connaissance de ces propriétés. Il est possible de laisser cette tâche au programmeur qui doit alors préciser ces propriétés, mais une solution intéressante (bien que difficile et coûteuse) consiste à les inférer automatiquement à partir d'une analyse du programme source. Le problème est complexe par le fait que ces termes évoluent en fonction des instanciations des variables qu'ils contiennent.

2.1. Optimisation de l'appel de procédure

Les principales optimisations portant sur l'appel de procédure consistent à réduire le nombre de tentatives d'unifications en éliminant celles que l'on sait inutiles. Pour cela, différentes techniques peuvent être utilisées, comme par exemple l'appel de procédure indexé et la préunification.

2.1.1. *Appel de procédure indexé*

La sélection d'une clause est en principe réalisée en effectuant un parcours séquentiel de toutes les clauses du programme à la recherche d'une clause dont la tête s'unifie avec le sous-but à effacer. Une première spécialisation simple, réalisée par tous les interprètes, consiste à regrouper les clauses définissant un même prédicat. L'accès aux clauses candidates est donc *indexé* par le nom du prédicat.

Malgré cela, certains prédicats comportent un nombre de clauses assez important (surtout les prédicats de faits). Pour pallier à cet inconvénient, le compilateur Prolog-DEC10 *[Warren 77]* a étendu l'appel indexé aux arguments. Cette technique consiste à grouper les clauses d'un prédicat suivant le foncteur du premier argument de chaque clause. Lors de l'appel, le premier argument du but à effacer est analysé pour déterminer quel est le groupe de clauses qu'il faut résoudre. Deux cas se présentent :

- L'argument est une variable. Dans ce cas, toutes les clauses sont candidates à l'unification effective.

- L'argument est un terme structuré ou une constante. Dans ce cas, on sélectionne le groupe correspondant à cet index.

Soit par exemple le prédicat :

eval(plus(x, y), r) ← eval(x, u), eval(y, v), plus(u, v, r). (1)
eval(moins(x, y), r) ← eval(x, u), eval(y, v), moins(u, v, r). (2)

L'appel indexé au prédicat *eval* consiste alors à exécuter la séquence :

Si variable(arg1) **alors**	clauses-candidates = {1, 2}
sinon si fonct(arg1) = plus **alors**	clauses-candidates = {1}
sinon si fonct(arg1) = moins **alors**	clauses-candidates = {2}
sinon	clauses-candidates = { }

Cette technique peut être étendue aux arguments suivants, au prix d'une complexité supérieure. Toutefois, l'indexation sur le seul premier argument semble un bon compromis, dans la mesure où l'habitude des langages procéduraux incite à la programmation des prédicats dans une forme où les premiers arguments sont utilisés comme paramètres (donc fournis à l'appel) et les derniers comme résultats.

L'appel indexé peut également être réalisé statiquement sur les littéraux des clauses. En effet, les arguments des littéraux sont souvent des structures, ce qui permet alors d'effectuer statiquement le choix des clauses qu'il faut tenter pour ce littéral.

2.1.2. *La préunification*

Le projet OPALE *[Berger 82]* a introduit une technique originale de réduction de l'espace de recherche. Cette technique, orientée vers une utilisation de Prolog en "base de données", consiste à effectuer une **préunification** sur les têtes de clauses *[Berger 84]*.

La préunification est une opération qui compare deux termes sur la base de la condition suivante :

$\forall (N1 \in T1, N2 \in T2 \ / \ pos(N1,T1) = pos(N2,T2))$
si $N1 \neq N2$
alors N1 est une variable **ou** N2 est une variable

où pos(N,T) désigne la position d'un noeud N dans un terme T.

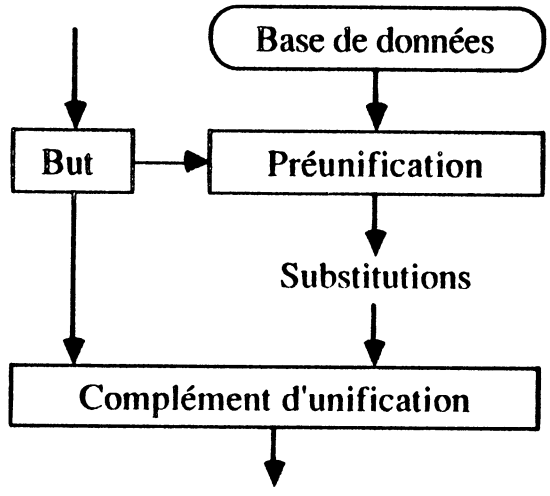
La préunification produit une liste de couples variable-valeur qui représente les substitutions nécessaires pour vérifier cette condition. Ainsi, par exemple, la préunification de :

réussit et produit les couples de substitutions {x=A, x=B}.

Il est clair que si deux termes ne sont pas préunifiables, ils ne seront pas unifiables. En cas de succès, un **complément d'unification** est nécessaire pour déterminer l'unificateur final sur la base des substitutions produites par la préunification. Ce complément d'unification permet par exemple de rejeter

l'unification des deux termes donnés ci-dessus.

L'avantage de cet opération réside dans le fait qu'elle peut être assez facilement cablée pour permettre de "filtrer au vol" les faits provenants d'une base de données.



Des mesures *[Berger 86]* ont montré que le taux d'échec de l'unification après succès de la préunification reste globalement inférieur à 20%, bien que sujet à d'importantes variations en fonction des programmes traités.

2.2. Expansion de l'unification

L'expansion de l'unification consiste à extraire explicitement une partie du système d'équations qu'il faut résoudre pour réaliser l'unification. Une notation pratique pour visualiser ce système est de le faire apparaître sous-forme de littéraux dans la queue de la clause. Cette opération s'effectue sur la base de la structure explicite des termes et provoque la génération d'un ensemble de littéraux spécialisés dans la réalisation de l'unification.

L'algorithme d'expansion est appliqué sur chaque littéral de la clause (y compris le littéral de tête) et se déroule de la manière suivante:

1) Expansion primaire : Décomposition des termes

Si l'un des arguments est un terme complexe, on le décompose récursivement en faisant apparaître un littéral spécialisé (constructeur) pour chaque sous-terme complexe du terme initial. Chacun de ces littéraux utilise une variable temporaire ti qui représente le sous-terme qu'il

reconstitue.

Par exemple, le littéral P(x.f(y,z)) se décompose en :

consf(y, z, t1), cons(x, t1, t2), P(t2)

en supposant que les prédicats constructeurs sont définis comme suit :

cons(x, y, x.y) ←.
consf(x, y, f(x, y)) ←.

2) Expansion secondaire : Mise en évidence des unifications

Si une variable x apparaît deux fois dans le littéral, on substitue l'une de ses occurrences par une variable temporaire t et on génère le littéral "unification(x,t)". De même, si une constante c est présente, on la substitue par une variable temporaire t et on engendre le littéral "unification(c, t)". La production de ces littéraux suppose la définition du prédicat :

unification(x, x) ←.

Cette transformation va donc introduire plusieurs littéraux nouveaux dans la clause ; ils seront placés avant les littéraux dont ils recomposent les arguments pour obtenir un fonctionnement identique dans une stratégie de parcours de gauche à droite. Le littéral de tête constitue une exception pour laquelle les littéraux engendrés sont placés au début de la queue de la clause.

L'expansion de l'unification nécessite la définition de certains prédicats constructeurs, fonction des symboles fonctionnels utilisés dans le programme. Le prédicat *unification* est un cas particulier de prédicat constructeur : il représente l'unification générale, qu'il faut conserver pour pouvoir unifier les variables liées dynamiquement à des termes de structure inconnue.

Si l'on applique cette méthode sur le programme :

append (NIL, x, x) ←.
append (a.x, y, a.z) ← append (x, y, z).
reverse (NIL, NIL) ←.
reverse (e.l, r) ← reverse (l, s), append (s, e.NIL, r).

L'expansion produit :

append(t1, t2, t3) ← **unification**(NIL, t1), **unification**(t2, t3).
append(t1, y, t3) ← **cons**(a, x, t1), **cons**(a, z, t3), append(x, y, z).
reverse(t1, t2) ← **unification**(NIL, t1), **unification**(t2, NIL).

reverse(t1, r) ← **cons**(e, l, t1), reverse(l, s),
cons(e, NIL, t2), append(s, t2, r).

Cette technique est utilisée dans le cadre de la compilation de Prolog. On peut en trouver une forme dérivée dans Prolog-DEC10 *[Warren 77]*, *[Warren 83]* ou dans POPLOG *[Mellish 85]*. Chaque prédicat constructeur peut être considéré comme prédéfini et dispose d'un traitement particulier au niveau de la procédure de résolution. Ils peuvent donc être remplacés par des séquences de code spécifiques, ou par des instructions spéciales dans la machine virtuelle servant d'intermédiaire à la génération effective de code. Par exemple, la machine virtuelle ZIP développée pour Prolog-X *[Clocksin 85]* ne comporte pas moins de 17 instructions d'unification différentes (essentiellement en fonction de la structure des termes), ce qui représente presque un tiers du nombre total d'instructions de cette machine.

Après l'expansion, les arguments placés en tête de clause représentent directement les paramètres effectifs passés à l'appel, sans qu'il soit nécessaire d'effectuer d'unification pour eux. En particulier, si les arguments de la tête de clause sont indépendants entre eux avant l'expansion, celle-ci ne génère aucun littéral spécialisé pour l'unification ; l'unification générale est donc remplacée par un simple appel de procédure. C'est le cas de la clause :

P(x, y, z) ←.

pour laquelle les variables x, y et z seraient liées inutilement, dans une implantation classique naïve, aux arguments passés en paramètre.

L'expansion de l'unification offre par conséquent les avantages suivants :

- Elle dissocie complètement l'unification des arguments de l'appel de procédure, ce qui autorise une spécialisation des techniques utilisées pour réaliser chacune de ces tâches.

- Elle permet une spécialisation de l'unification en tenant compte de la structure du terme local à unifier (la notion de type que nous abordons par la suite permet également de connaître la structure partielle du terme entrant). L'unification générale n'est utilisée que lorsqu'elle est indispensable et est toujours repoussée au dernier instant.

- Elle permet une factorisation des opérations à effectuer. En effet, si deux termes distincts contiennent le même sous-terme, un seul littéral peut être produit pour les représenter tous les deux.

- Elle autorise un meilleur contrôle de l'instant de construction des termes.

Le chapitre suivant montre une utilisation particulière de cette technique qui, en coopération avec les modes directionnels, permet des optimisations très intéressantes

2.3. Choix de l'algorithme d'unification

L'algorithme utilisé pour effectuer l'unification de deux termes est très coûteux s'il est utilisé dans toute sa généralité. Ainsi, l'unification des termes :

$$t1 = f(x1,..., xn, g(y0, y0),..., g(yn\text{-}1, yn\text{-}1), xn)$$
$$t2 = f(g(x0, x0),..., g(xn\text{-}1, xn\text{-}1), y1,..., yn, yn)$$

a un coût exponentiel. Il est à noter qu'il existe des algorithmes d'unification de coût linéaire *[Martelli 82]*, mais dont le coût pour de petites unifications est supérieur à celui de l'algorithme classique ; de ce fait, ils sont peu utilisés.

En réalité, l'algorithme général d'unification peut être grandement simplifié par la connaissance de propriétés sur les termes à unifier. Dans la suite, nous allons voir quels sont les différentes variantes de cet algorithme, puis nous aborderons la manière d'obtenir les propriétés qui permettent d'en sélectionner le meilleur.

2.3.1. *L'algorithme d'unification et ses dérivés*

L'algorithme général d'unification est le suivant :

```
procédure unification ( entrée : t1, t2 : termes ;
                        sortie : σ : unificateur ;
                                 unifiables : booléen )
    variables : k : entier ;
                σ1 : unificateur ;
    début
      si (t1 est une variable)
      alors
        si t1 = t2
        alors
          unifiables := vrai ;
          σ := ∅ ;
```

```
        sinon
            si (occurrence de t1 dans t2)  /* Occur Check */
            alors
                unifiables := faux ;
            sinon
                unifiables := vrai ;
                σ := {t1/t2} ;
    sinon
        si (t2 est une variable)
        alors
            unification (t2, t1, σ, unifiables) ;
        sinon  /* t1 = f(x1,...,xn) et t2 = g(y1,...,ym) */
            si (f ≠ g) ou (n ≠ m)
            alors
                unifiables := faux ;
            sinon
                unifiables := vrai ;
                σ := ∅ ;
                pour k=1 à n et tantque (unifiables) faire
                    unification (σ(xk), σ(yk), σ1, unifiables) ;
                    si unifiables alors
                        σ := σ1 o σ ;
fin
```

Nous avons vu au chapitre précédent que la reconnaissance de formes (*pattern-matching*) est une restriction de l'unification dans laquelle l'un des termes est clos. L'algorithme d'unification devient alors :

```
procédure conformité (  entrée :  t1 : terme quelconque ;
                                  t2 : terme clos ;
                        sortie :  σ : unificateur ;
                                  unifiables : booléen  )
    variables :  k : entier ;
                 σ1 : unificateur ;
    début
        si (t1 est une variable)
        alors
            unifiables := vrai ;
            σ := {t1/t2} ;
        sinon  /* t1 = f(x1,...,xn) et t2 = g(y1,...,ym) */
            si (f ≠ g) ou (n ≠ m)
            alors
```

```
            unifiables := faux ;
         sinon
            unifiables := vrai ;
            σ := ∅ ;
            pour k=1 à n et tantque (unifiables) faire
               conformité (σ(xk), σ(yk), σ1, unifiables) ;
               si unifiables alors
                  σ := σ1 o σ ;
fin
```

Si les deux termes à unifier sont clos, l'algorithme se simplifie encore et devient un simple test d'égalité :

```
procédure égalité( entrée : t1, t2 : termes clos ;
                   sortie : unifiables : booléen )
   variables : k : entier ;
   début  /* t1 = f(x1,...,xn) et t2 = g(y1,...,ym) */
      si (f ≠ g) ou (n ≠ m)
      alors
         unifiables := faux ;
      sinon
         unifiables := vrai ;
         pour k=1 à n et tantque (unifiables) faire
            égalité (xk, yk, unifiables) ;
   fin
```

On peut encore effectuer bien des spécialisations en connaissant d'autres caractéristiques des termes. Si par exemple l'un des termes est une variable, on peut utiliser l'algorithme suivant :

```
procédure affectation ( entrée : t1: variable ;
                                 t2 : termes;
                        sortie : σ : unificateur ;
                                 unifiables : booléen )
   variables : k : entier ;
               σ1 : unificateur ;
   début
      si t1 = t2
      alors
         unifiables := vrai ;
         σ = ∅ ;
      sinon
```

```
        si (occurrence de t1 dans t2)  /* Occur Check */
        alors
           unifiables := faux ;
        sinon
           unifiables := vrai ;
           σ := {t1/t2} ;
fin
```

La version la plus simple correspond à l'unification d'une variable libre avec un terme dont on est sûr qu'il ne la contient pas (un terme clos par exemple). Cette propriété évite le test d'occurrence et permet de produire :

```
procédure substitution ( entrée : t1: variable ;
                                  t2 : terme clos ;
                         sortie : σ : unificateur )
début
   σ := {t1/t2} ;
fin
```

L'utilisation de ces techniques peut être résumée dans le tableau récapitulatif suivant, fonction des propriétés des termes à unifier :

	terme complexe	**variable libre**	**terme clos**
terme complexe	unification	affectation	conformité
variable libre	affectation	affectation	substitution
terme clos	conformité	substitution	égalité

Il est clair que l'efficacité de la Résolution peut être notablement améliorée si l'on peut déterminer statiquement la classe des termes à unifier. Certains cas peuvent être déterminés simplement lorsque l'un des arguments est un terme clos. Par exemple, le premier argument de la clause suivante est toujours clos :

append(NIL, x, x) ←.

Une expansion de l'unification mettrait en évidence le "pattern-matching" correspondant à l'unification de cet argument et produirait pour :

append(t1, t2, t3) ← conformité(t1, NIL), unification(t2, t3).

Cet exemple montre comment peuvent être exploitées les caractéristiques *statiques* des termes à unifier. Toutefois, une plus grande finesse de choix peut être obtenue en connaissant des propriétés *dynamiques* sur les termes. Ainsi, si l'on peut savoir que cette clause est toujours appelée avec un terme clos comme premier argument, le "pattern-matching" utilisée pour l'unification du premier argument peut se transformer en égalité.

Nous étudions ce problème dans le paragraphe suivant, à l'aide des modes qui permettent de caractériser plus finement les propriétés exploitables pour optimiser la Résolution.

2.3.2. *Les modes*

Les modes ont été introduits par Warren *[Warren 77]* pour permettre la classification des arguments d'appel des prédicats en fonction d'un certain nombre de propriétés. Alliés aux propriétés des arguments locaux, ils permettent un meilleur choix de l'algorithme d'unification à utiliser.

En théorie, le nombre de modes possibles est infini puisqu'il dépend du nombre de propriétés que l'on peut définir sur les termes. Les modes "utilisables" sont définis par le concepteur de l'interprète ou du compilateur en fonction des optimisations qu'ils permettent de réaliser, tout en essayant d'en minimiser le nombre pour ne pas alourdir le traitement par de multiples cas particuliers.

Les propriétés utilisées par Warren sur les arguments effectifs des prédicats sont au nombre de trois :

+ : L'argument est toujours instancié (n'est pas une variable).
- : L'argument n'est jamais instancié (est une variable).
? : L'argument est parfois instancié, parfois non.

Leur utilisation pour choisir le "meilleur" algorithme d'unification est très simple ; il est résumé, pour un seul argument, par le tableau suivant :

argument formel \ *argument effectif*	**mode +**	**mode -**	**mode ?**
terme complexe	unification	affectation	unification
variable libre	affectation	affectation	affectation
terme clos	conformité	substitution	conformité

Warren définit le mode d'un prédicat d'arité n comme un n-uplet à valeurs dans {+,-,?}. Le mode implicite de tout prédicat est celui dans lequel tous les arguments ont le mode ?.

Exemple : soit le prédicat : P(x, x) ←.
L'expansion de l'unification produit : P(x, t) ← unification(x, t).

En fonction du mode fixé pour ce prédicat, l'algorithme d'unification peut être choisi avec plus de précision. Le tableau suivant montre plusieurs de ces cas :

mode	**clause**
(+,+)	P(x, t) ← unification (x, t)
(+,-)	P(x, t) ← affectation(t, x)
(-,+)	P(x, t) ← affectation(x, t)
(-,-)	P(x, t) ← affectation(x, t)

Tous les modes contenant au moins un argument en mode ? se comportent comme le premier cas. On peut noter que le dernier cas peut être simplifié puisque le test d'occurrence n'est pas utile lorsque les deux termes sont des variables libres. De toute façon, peu d'interprètes mettent en oeuvre le test d'occurrence en raison de son coût trop élevé ; Warren se place dans ce cas, chaque affectation étant alors remplacée par une substitution.

Les avantages de l'utilisation des modes vont au delà du simple choix de l'algorithme d'optimisation. Une autre conséquence importante est une meilleure gestion de l'espace mémoire. Par exemple, les variables de l'entête de clause instanciées lors de l'appel du prédicat (mode +) ne servent pas à construire un terme mais plutôt à le décomposer ; l'espace nécessaire à leur

utilisation peut donc être récupéré plus aisément.

On peut également constater l'utilité de ces modes pour améliorer l'appel de procédure indexé. En effet, si le mode du premier argument d'un prédicat est "-", cet argument sera toujours une variable lors de la résolution. Il est donc inutile d'effectuer un appel indexé sur cet argument puisqu'il conduira à une sélection de toutes les clauses du prédicat. L'indexation peut alors être effectuée sur le premier argument placé en mode "+", ou à défaut en mode "?".

Toutefois, le problème de ces modes réside dans leur unicité. En effet, la compilation ne produit qu'une séquence de code pour chaque prédicat, le mode utilisé devant englober tous les types d'appels possibles dans le programme. Ainsi, si dans un même programme le prédicat *append* est appelé par les buts append(x.NIL, y.NIL, z) et append(x, y, x.y.NIL), seul le mode (?,?,?) est utilisable pour ce prédicat, aucune optimisation n'étant alors possible.

2.3.3. *Inférence automatique de modes*

Dans les compilateurs actuels, les modes doivent être déclarés par le programmeur. Aucune vérification n'est faite et une définition incorrecte provoque des erreurs à l'exécution. Pour éviter ces problèmes, différents travaux *[Mellish 81], [Debray 85a]* proposent un mécanisme d'inférence automatique de modes à partir d'une l'analyse du programme source.

Ces mécanismes sont basés sur deux hypothèses :

- Exécution du programme dans l'ordre classique (de gauche à droite).
- Production d'un seul mode par prédicat.

Par ailleurs, ils utilisent un ensemble de modes plus complet que celui de Warren, En effet, l'utilisation des seuls modes "+", "-" et "?" est insuffisante pour autoriser une inférence de modes efficace. Ainsi, par exemple, pour la clause :

append(a.x, y, a.z) ← append(x, y, z).

la connaissance du mode "+" pour le premier argument est insuffisante pour déduire le même mode pour l'appel récursif. En effet, si l'argument effectif est le terme u.v, le premier argument de l'appel récursif est le terme attaché à la variable v ; comme on ne sait rien sur ce terme, le mode "+" n'est donc pas utilisable. Par contre, si le prédicat est appelé avec un terme clos comme premier argument, l'appel récursif s'effectue également avec un terme clos ;

dans ce cas, le mode "+" est utilisable.

Ces constatations ont ammené Mellish à préciser la notion de mode en tenant compte du degré d'instantiation des termes. Les termes sont décomposés en quatre classes :

U : Complètement non-instancié (variable).

IU : Instancié à un terme structuré dont toutes les composantes immédiates sont non-instanciées.

II : Instancié à un terme clos.

IM : Instancié à un terme structuré qui n'est ni IU ni II.

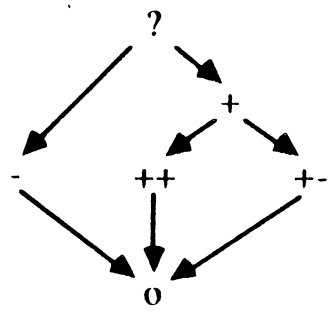
Ces différents états possibles forment un treillis qui reflète le degré d'instanciation des termes :

$$U \rightarrow IU \rightarrow IM \rightarrow II$$

Mellish définit les modes partiels à partir de ces quatre états de la manière suivante :

-	= {U}	Toujours non-instancié.
+-	= {IU}	Toujours instancié à une structure dont les arguments sont non-instanciés.
++	= {II}	Toujours instancié à un terme clos.
+	= {II, IU, IM}	Toujours instancié avec toutes les possibilités pour la structure interne.
?	= {U, IU, IM, II}	Parfois instancié, parfois non.
o	= { }	Indéfini. Ce mode sert en fait de valeur initiale pour l'algorithme d'inférence.

Ces différents modes forment également un treillis qui met en évidence un ordre partiel sur les degrés d'instanciation :



L'algorithme d'inférence de modes utilise ces modes partiels mais retourne finalement les trois modes "de base" en appliquant la transformation :

mode partiel	mode final
-	-
+-, ++, +	+
?, o	?

La définition des modes partiels permet d'inférer le mode "+" pour le premier argument de l'appel récursif de *append* si le premier argument de l'appelant est instancié à un terme clos. Toutefois, un problème subsiste pour inférer les modes dans les clauses du type :

gappend(x, y, z, r) ← append(x, y, t), append(t, z, r).

En supposant que l'on ait pu établir le mode "+" pour les trois premiers arguments de *gappend*, le premier appel à *append* se place de façon évidente en mode (+,+,-) car il s'agit de la première occurrence de la variable t. La difficulté consiste à définir le mode du premier argument du deuxième appel à *append*. En effet, ce mode est "+" si la variable t est toujours instanciée par la résolution du premier *append*, mais il reste à "-" dans le cas contraire.

L'inférence des modes dans ces clauses doit donc tenir compte de l'évolution des arguments. Pour un argument donné, on devra alors considérer deux modes distincts : l'un défini avant et l'autre après l'effacement du littéral.

2.4. Les types

Les types en Prolog représentent en fait l'association d'un domaine (sous-ensemble de l'univers de Herbrand) aux arguments des prédicats *[Mishra 84]*. Ils permettent notamment :

- Une vérification de la cohérence des programmes par le contrôle de la cohérence des domaines des termes entre eux.

- Une spécialisation de la représentation physique de chaque terme. Cette technique a notamment été utilisée par Nilsson *[Nilsson 83]* pour transformer les termes en structures de données Pascal.

- Une spécialisation encore plus grande de l'unification. En effet, l'expansion de l'unification ne permet de mettre en évidence que la structure des termes locaux à l'entête d'une clause. La connaissance des domaines permet d'avoir également des informations sur la structure des arguments avec lesquels ils vont être unifiés.

Comme pour les modes, deux solutions sont possibles pour déterminer les domaines des arguments. Soit le programmeur lui-même les fixe à l'écriture de son programme *[Turbo]*, soit un programme spécialisé permet de les produire à partir de déductions effectuées sur le texte source *[Kanamori 84]*.

Turbo-Prolog représente un bon exemple d'utilisation des domaines pour l'optimisation de la résolution. Le programmeur doit les indiquer à partir de types de base prédéfinis (entier, réel, caractère, chaîne, symbole et fichier). Des opérateurs spécialisés permettent de construire des domaines plus généraux, notamment la *liste* qui bénéficie d'un traitement spécialisé. Ces contraintes font que ce langage est parfois considéré comme plus proche de Pascal que de Prolog, mais lui permettent en revanche d'atteindre une efficacité importante.

3. Réordonnancement des sous-buts

Le chapitre 1 montre comment l'ordre de sélection des sous-buts peut influencer la taille de l'arbre de recherche, et même dans certains cas permettre de choisir un arbre fini au lieu d'un arbre infini. Le coût de la résolution étant directement lié à la taille de cet arbre, il est évident que la réduction de celle-ci conduit à un gain intéressant.

Une sélection judicieuse de l'ordre d'effacement des sous-buts peut donc permettre de choisir un "meilleur" arbre de recherche. L'exploitation de ces constatations dans le cadre général de la Résolution s'effectue en proposant un ordre de résolution des littéraux des queues de clauses différent de l'ordre d'écriture. Une bonne utilisation consiste à réordonner ces littéraux statiquement de manière à éviter des pertes de temps dues à l'évaluation dynamique des critères de sélection.

L'emploi, dans la programmation, de prédicats à effets de bord limite cet ordonnancement aux cas suivants :

- L'ordre entre deux littéraux séparés par un *cut* doit être conservé.

- L'ordre entre deux prédicats effectuant des effets de bord doit être conservé.

Nous traitons ce problème dans la suite par l'intermédiaire d'une fonction de coût qui nous permet de comparer le coût de résolution de différents ordonnancements et de choisir "le meilleur". Deux cas sont distingués :

- Les arbres de recherche toujours finis, correspondant à la résolution de prédicats non récursifs et qui ne font pas appel à des prédicats récursifs. Le problème se restreint ici à choisir le meilleur arbre parmi ceux qui sont possibles.

- Les arbres de recherche pouvant contenir des branches infinies. Dans ce cas, le problème consiste essentiellement à rendre finis ces arbres par élimination des branches infinies.

3.1. Réduction d'un arbre de recherche fini

Le choix d'un "meilleur" ordre d'effacement ne peut s'effectuer que grâce à l'utilisation d'une fonction permettant de comparer les coûts des

différents ordres possibles. Nous présentons dans ce qui suit une telle fonction, et nous discutons ensuite de son application dans le contexte indiqué.

Une première constatation est que le volume d'un arbre de recherche est dépendant de la cardinalité (nombre de solutions) des prédicats qui y figurent. Définissons la fonction *Card*(P) par :

Card(P) évalue la cardinalité *maximale* du prédicat P, c'est à dire le nombre maximum de solutions que sa résolution peut retourner. Cette cardinalité est égale au nombre de clauses vides dans un arbre de recherche correspondant au but $\leftarrow P(x1,...,xn)$, où xi est une variable, et où $\forall i,\forall j$, $xi \neq xj$ si $i \neq j$.

Si P est un prédicat défini par les clauses Ci ($1 \leq i \leq k$), sa cardinalité est alors :

$$Card\,(P) = \sum_{i=1}^{k} Card\,(Ci)$$

Si $C = P(x) \leftarrow Q1(x1),...,Qn(xn).$ est une clause de P, sa cardinalité est définie par :

$$Card\,(C) = \prod_{i=1}^{n} Card\,(Qi)$$

avec $n=0 \Rightarrow Card\,(C)=1$

La cardinalité d'une clause C est indépendante de l'ordre de résolution des Qi. Toutefois, les arbres de recherches étant différents en fonction de l'ordre choisi, le coût nécessaire à l'acquisition de ces solutions peut varier.

Remarque

Nous avons considéré ici la cardinalité d'un prédicat comme celle du but le plus général. En réalité, cette cardinalité pourrait être affinée en prenant en compte, non pas celle d'un prédicat en général, mais celle de chacun de ses buts en particulier, en fonction de la structure des termes placés en arguments. Cet aspect est abordé dans le chapitre suivant, où la cardinalité est calculée en fonction des différents modes possibles d'un prédicat.

Une première façon de sélectionner un meilleur ordre d'exécution consiste à placer les sous-buts dans l'ordre croissant de leurs cardinalités *[Naish 85]*. Cette solution représente un critère simple qui donne d'assez bons

résultats en général, surtout avec les prédicats faits. Par exemple, supposons disposer des prédicats A et B ayant respectivement une cardinalité de 1 et 1000. Considérons les deux ordres de résolution suivants :

$$\leftarrow A, B. \quad \text{et} \quad \leftarrow B, A.$$

Le premier ordre va provoquer un appel à A et un appel à B, alors que le second provoque un appel à B et mille appels à A.

En réalité, cette manière de sélectionner l'ordre des sous-buts devient insuffisante dès que les prédicats mis en jeu ne sont pas des faits. Le coût peut être en fait calculé de façon plus précise en prenant en compte le nombre d'unifications *tentées* pour effectuer la résolution.

Soit le programme Prolog :

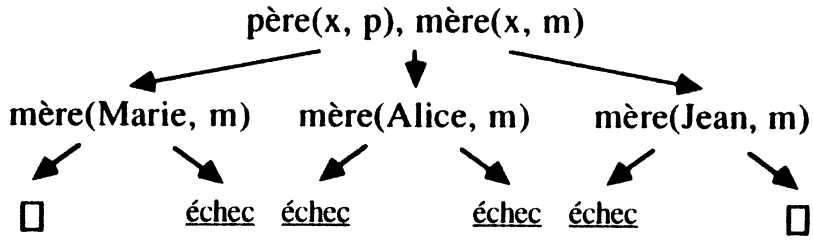
père(Marie, Pierre) ←. mère(Marie, Anne) ←.
père(Alice, Pierre) ←. mère(Jean, Marie) ←.
père(Jean, Marc) ←.

parents(x, p, m) ← père(x, p), mère(x, m).

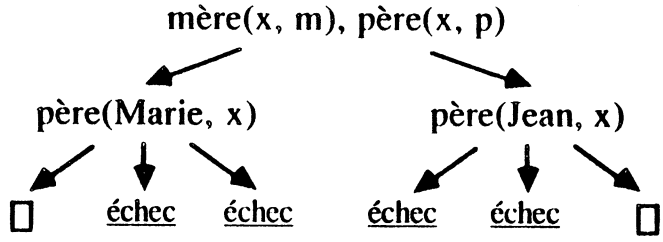
où le prédicat *parents* définit les parents *p* et *m* d'une personne *x*. Il existe deux possibilités d'ordonnancement pour cette clause :

(1) parents(x, p, m) ← père(x, p), mère(x, m).
(2) parents(x, p, m) ← mère(x, m), père(x, p).

Développons les arbres de recherche correspondant à la résolution de ←parents(x,p,m) pour les deux ordonnancements possibles donnés ci-dessus, en faisant apparaitre toutes les tentatives d'unification. Pour le premier, on obtient l'arbre :



alors que le deuxième fournit :



Pour arriver au même résultat, le premier ordonnancement tente 9 unifications (dont 5 réussies) alors que le deuxième n'en tente que 8 (dont 4 réussies). Cette deuxième solution est donc d'un coût moindre.

Définissons le coût d'un prédicat par la fonction *Coût*(P) telle que :

Coût(P) évalue le coût *maximal* de la résolution du prédicat P, c'est à dire le nombre de tentatives d'unifications nécessaires à la résolution *complète* du but ←P(x1,...,xn), où xi est une variable, et où ∀i,∀j, xi≠xj si i≠j.

Cette fonction considère chaque unification tentée comme une unité de coût. Il est évident qu'en réalité, le temps nécessaire à une unification n'est pas constant car il dépend d'une part de la taille des termes en présence, et d'autre part du nombre de variables qu'ils contiennent.

Pour un prédicat composé exclusivement de faits, le coût de sa résolution, ainsi que sa cardinalité, est théoriquement égal au nombre de faits définissant le prédicat. Pour notre exemple, nous avons donc :

$$Coût(\text{père}) = Card(\text{père}) = 3$$
$$Coût(\text{mère}) = Card(\text{mère}) = 2$$

Ce coût peut en fait être réduit par les techniques d'optimisation vues précédemment, notamment l'appel de procédure indexé. Cette évaluation peut par conséquent être affinée mais nous nous placerons ici dans un cadre plus général.

Avec cette fonction, nous serions en mesure de calculer le coût des différents ordonnancements de la clause *parents* pour permettre de sélectionner le meilleur. Le problème consiste maintenant à trouver une technique adéquate d'évaluation de ce coût.

Soit P est un prédicat formé de clauses Ci ($1 \leq i \leq k$), le coût de résolution de ce prédicat est :

$$Coût(P) = \sum_{i=1}^{k} Coût(Ci)$$

Soit C=(P←Q1,...,Qn) une clause de P. Pour une résolution dans l'ordre d'écriture de la clause, le coût de sa résolution est égal au coût de résolution de Q1, plus le produit de la cardinalité de Q1 et du coût du reste de la clause Q2,...,Qn (puisque ce reste va être résolu pour chaque solution de Q1). En notant Ci=*Card*(Qi) et Ki=*Coût*(Qi), on a donc :

$$Coût\,(C) = K1 + \sum_{i=2}^{n}(Ki * \prod_{j=1}^{i-1} Cj)$$

que l'on peut également écrire :

$$Coût(C) = K1 + C1(K2 + ... Cn\text{-}1(Kn))...)$$

Nous disposons maintenant d'une fonction permettant de calculer le coût d'un prédicat. Ainsi, le coût du prédicat *parents* est, pour les deux ordonnancements, égal à :

(1) *Coût*(parents) = 3 + 3*2 = 9
(2) *Coût*(parents) = 2 + 2*3 = 8

Ce qui indique bien que le deuxième ordonnancement est le meilleur. La cardinalité de ce même prédicat peut être obtenue par la formule donnée précédemment :

Card(parents) = 6

Si l'évaluation de coûts correspond bien à la réalité, l'évaluation de la cardinalité est très éloignée de sa valeur réelle. Ceci est dû au fait que les cardinalités des prédicats *père* et *mère* sont définis pour le cas le plus général ; or, lors de la résolution de père(x,p) dans le deuxième ordonnancement, le premier argument est toujours une constante, produite par la résolution de mère(x,m). L'observation de la structure du prédicat *père* montre que sa cardinalité est en fait égale à un si l'un de ses arguments n'est pas une variable libre. La connaissance de cette information nous permettrait d'affirmer que la cardinalité du prédicat *parents* est en fait inférieure ou égale à 2.

La cardinalité d'un prédicat est par conséquent fortement dépendante des arguments présents lors de la résolution du prédicat. Tous les calculs effectués

jusqu'ici ne représentent donc que des bornes maximums considérées comme le cas le plus péssimiste. Cependant, il est important d'obtenir une évaluation de la cardinalité aussi proche que possible de la réalité dans la mesure où cette valeur intervient dans le coût des prédicats. Le chapitre suivant traite d'une catégorie de modes qui autorise le calcul de cette cardinalité d'une façon plus précise.

Remarque 1

Comparons maintenant la sélection de l'ordre par minimisation du coût avec la sélection par cardinalités croissantes. Supposons deux prédicats Q1 et Q2 ayant respectivement comme coût et cardinalité :

$$Q1 : \quad K1 = 10, \qquad C1 = 5$$
$$Q2 : \quad K2 = 1000, \qquad C2 = 10$$

Le coût réel des deux ordres possibles serait :

$$\mathit{Coût}(Q1, Q2) = 10 + 5*1000 = 5010$$
$$\mathit{Coût}(Q2, Q1) = 1000 + 10*10 = 1100$$

ce qui est en nette contradiction avec la sélection par cardinalités croissantes.

Remarque 2

La présence de *cuts* dans les clauses modifie les calculs des cardinalités et des coûts. Soit par exemple la conjonction :

$$C = Q1,...,Qi, !, Qi+1,...,Qn$$

Le franchissement d'un *cut* ramène la cardinalité de tout ce qui le précède à un. Les valeurs respectives de la cardinalité et du coût de cette conjonction sont alors :

$$\mathit{Card}(C) = \mathit{Card}(Qi+1,...,Qn)$$
$$\mathit{Coût}(C) = \mathit{Coût}(Q1,...,Qi) + \mathit{Coût}(Qi+1,...,Qn)$$

L'effet d'un *cut* n'étant pas limité à une seule clause, sa présence modifie en réalité les valeurs pour l'ensemble du prédicat. Si nous avons le prédicat :

$$P \leftarrow Q1, !, Q2.$$
$$P \leftarrow Q3.$$

ces valeurs devienent :

$$\mathit{Card}(P) = \max(\mathit{Card}(Q2), \mathit{Card}(Q3))$$
$$\mathit{Coût}(P) = \mathit{Coût}(Q1) + \max(\mathit{Coût}(Q2), \mathit{Coût}(Q3))$$

3.2. Elimination des branches infinies

Le chapitre 1 présente différentes techniques de contrôle qui permettent l'élimination de certaines branches infinies par modification de l'ordre d'effacement des sous-buts. La première de ces techniques consiste à laisser au programmeur le soin de fournir explicitement les instructions de contrôle. Ceci implique une bonne connaissance du processus de résolution, ainsi qu'une surcharge du programme avec des instructions qui en détruisent le sens logique.

Une deuxième approche consiste à éliminer automatiquement les branches infinies en analysant les prédicats récursifs, producteurs de telles branches *[Pelhat 87]*. Les déclarations d'attente de Naish *[Naish 85]* se placent dans ce cadre.

3.2.1. *Ordonnancement des littéraux*

Nous proposons dans ce qui suit l'approche d'une troisième solution, développée plus profondément dans le chapitre suivant, basée sur une extension de notre fonction de coût aux prédicats récursifs. Il faut indiquer que le coût de tels prédicats, ainsi que leur cardinalité, est potentiellement infini. Certaines études *[Sawamura 85]*, *[Mellish 85]* permettent toutefois de déterminer automatiquement si un prédicat est *déterministe* (de cardinalité inférieure ou égale à un), bien que rien ne puisse être dit sur leur coût, en général dépendant de la valeur des arguments qui limitent la profondeur de récursion. Ainsi, par exemple, le coût de résolution du prédicat *append* dépend de la taille de la liste donnée en premier paramètre.

Soit le programme :

objet(A) ←.
objet(S(0)) ←.

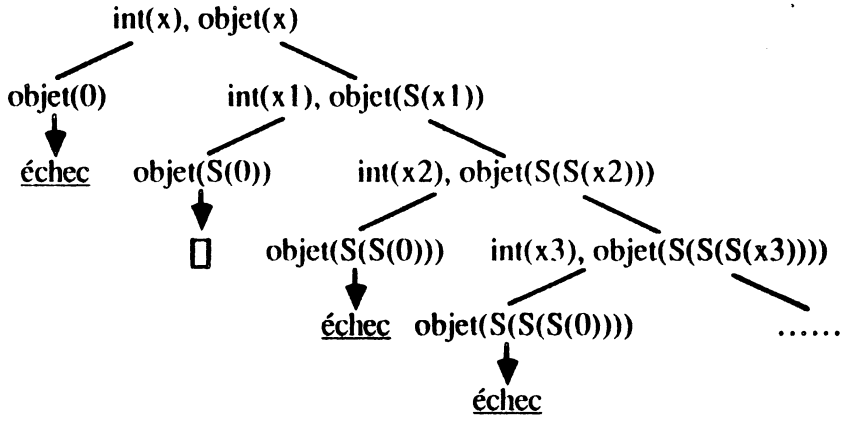
int(0) ←.
int(S(x)) ← int(x).

res(x) ← int(x), objet(x).

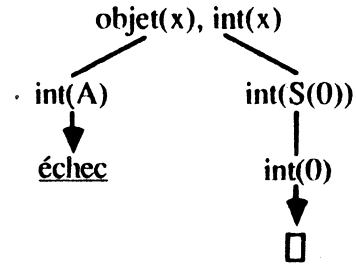
L'ordonnancement du prédicat *res* conduit à deux possibilités :

(1) res(x) ← int(x), objet(x).
(2) res(x) ← objet(x), int(x).

L'arbre de recherche associé au premier ordonnancement pour le but ←res(x) est :



alors que pour le second, il est :



Il est clair que la première solution entraîne un bouclage de la résolution car *int* va fournir tous les entiers un par un. On peut toutefois constater que ce bouclage ne s'effectue qu'après avoir trouvé la solution. Le deuxième ordonnancement aboutit par contre à un arbre de recherche fini.

Même s'il n'est pas possible de donner une borne finie au coût d'un prédicat récursif, il existe un moyen de comparer les cardinalités d'un même prédicat pour différentes valeurs de ses arguments. En notant *Card*(P(x)) la cardinalité du prédicat P pour le n-uplet d'arguments x, nous avons :

$$\forall \sigma \text{ substitution}, \mathit{Card}(P(\sigma(x))) \leq \mathit{Card}(P(x))$$

En d'autres termes, plus les arguments sont instanciés, plus la probabilité d'avoir une cardinalité finie est grande.

Cette constatation nous conduit à l'énoncé d'une règle simple qui consiste à appliquer l'appel d'un prédicat récursif le plus tard possible. Ceci offre un double avantage :

- A l'instant de sa résolution, ses arguments bénéficieront de toutes les substitutions issues de la résolution des buts qui le précèdent. Sa cardinalité aura ainsi une plus grande chance d'être finie.

- Si l'appel récursif est placé en fin de clause, son implantation peut faire l'objet de l'*optimisation de l'appel terminal [Warren 80]*.

On peut noter que, si cette règle peut souvent s'appliquer, elle n'est pas générale. En effet, dans l'exemple du *reverse* donné au chapitre 1, nous avons constaté que l'appel récursif de *reverse* doit être placé en tête dans un cas et en queue dans l'autre. On peut toutefois constater que la clause en question contient *deux* appels à des prédicats récursifs, et que dans ce cas, le problème est plutôt de savoir lequel exécuter avant l'autre. Les modes directionnels exposés au chapitre 3 donnent une solution à ce problème.

Remarque 1

Le fait qu'un prédicat soit résolu avec tous ses arguments clos ne signifie pas que sa résolution se termine. Exemple :

P(x) ←.
P(x) ← P(x).
← P(A).

Remarque 2

Le fait que la cardinalité d'un but qui appelle un prédicat récursif soit finie (prédicats déterministes par exemple) ne signifie pas que son coût le soit également. Autrement dit, il se peut très bien que sa résolution boucle. Exemple :

P(x) ← P(x).
← P(x).

Le calcul des coûts tel que nous l'avons présenté est par conséquent insuffisant pour permettre un choix correct du meilleur ordonnancement dans tous les cas. La présence d'un prédicat récursif ne permet plus d'évaluer correctement ni la cardinalité, ni le coût de résolution des prédicats qui l'appelent.

Ces évaluations peuvent cependant être affinées en exploitant certaines propriétés des termes. Les modes étudiés au chapitre suivant permettent d'affiner effectivement ces calculs ; ils permettent également de détecter le déterminisme des prédicats en fonction de différentes possibilités d'appel.

C'est ainsi qu'il est possible de détecter le déterminisme du prédicat *int* si son argument est un terme clos.

3.2.2. *Ordonnancement des clauses*

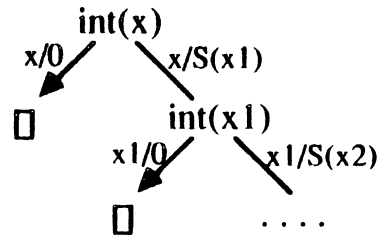
Bien qu'un arbre de recherche soit infini, il contient nécessairement une clause vide si l'ensemble de clauses est insatisfaisable. Il reste donc intéressant d'obtenir les solutions liées à cette clause vide, même si l'interprête doit boucler par la suite.

La stratégie *depth-first* effectue le parcours de ce arbre de gauche à droite, c'est à dire en sélectionnant les clauses dans l'ordre de leur écriture. Dans le cas particulier des arbres de recherche infinis, l'ordre dans lequel sont choisies ces clauses a une influence déterminante. Bien que cet ordre ne modifie en rien la taille de l'arbre, il peut permettre de placer une branche infinie à gauche dans l'arbre de recherche (donc parcourue en premier), ou à droite (alors parcourue en dernier) *[Pelhat 87]*.

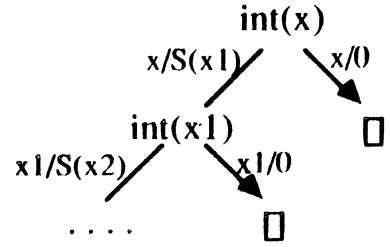
Reprenons notre prédicat *int* :

int(0) ←.
int(S(x)) ← int(x).
← int(x).

L'arbre de recherche correspondant à cet ordre d'écriture est :



alors que si les clauses de *int* sont choisies dans l'ordre inverse, on obtient :



Ces deux arbres sont tous les deux infinis et entrainent donc tous les deux un bouclage du processus de résolution. Cependant, on peut constater que le parcours de gauche à droite de cet arbre est favorable au premier cas dans la mesure où toutes les solutions sont trouvées : la boucle infinie est donc "productive". Inversement, le deuxième arbre provoquer l'entrée du processus de résolution dans une boucle infinie "improductive" si l'on utilise le même type de parcours.

Il est clair que quitte à boucler, autant produire des solutions ; il est donc préférable de toujours placer les clauses contenant des appels récursifs en dernier. Ce réordonnancement peut s'effectuer automatiquement en respectant toutefois les contraintes suivantes :

- L'ordre entre deux clauses contenant des prédicats à effets de bord doit être conservé

- L'ordre entre deux clauses séparées par une clause contenant un *cut* doit être conservé.

CHAPITRE 3

Utilisation des modes directionnels dans la Résolution

Ce chapitre représente notre apport personnel dans le domaine de l'optimisation de la résolution par l'exploitation des modes. Nous montrons en particulier que la classe de modes étudiée, les *modes directionnels*, permet d'élargir le champ des possibilités d'optimisation, ainsi que de traiter des problèmes résolus en général par des techniques de contrôle explicite.

L'utilisation des modes directionnels procède d'une démarche inverse de celle des modes de Warren. En effet, ces derniers ne sont utilisés que pour effectuer des optimisations locales lors de l'unification. Leur production *[Mellish 81]*, *[Debray 85a]* est basée sur la connaissance d'une stratégie de résolution fixe, ce qui peut se schématiser par :

Programme + Stratégie → Modes → Optimisations locales

Les modes directionnels sont par contre produits indépendamment de toute stratégie. Leur utilisation permet alors non seulement des optimisations locales du même ordre que celles réalisées par les modes de Warren, mais également de déterminer une meilleure stratégie pour la sélection des sous-buts à effacer.

Programme → Modes → Stratégie + Optimisations locales

Ce chapitre débute par une présentation formelle de la notion de modes directionnels. Puis nous présentons les différentes possibilités de transformation de programmes ouvertes par ces modes, et leur intérêt pour l'optimisation de la résolution au niveau local. Nous terminons par une description d'une technique permettant la vérification et la production automatique de ces modes.

1. Bases théoriques des modes directionnels

Nous commençons cette section par une définition générale de la notion de mode, les **modes structurels**, sur lesquels nous donnons quelques lemmes généraux utilisés essentiellement pour faciliter leur production automatique. Nous présentons ensuite un cas particulier de ces modes, principal sujet de cette thèse : les *modes directionnels*. Ces modes ont l'avantage de fixer un sens absolu aux flux de données transitant par les arguments des prédicats. On peut alors réordonner les littéraux des clauses en fonction de ces informations.

1.1. Les modes structurels

L'objectif principal des modes structurels est leur utilisation pour effectuer un réordonnancement statique des littéraux des queues de clauses. Dans ce but, il est important de connaître l'état des arguments non seulement *avant*, mais également *après* l'effacement complet d'un but. Pour ce faire, nous décomposerons un mode en deux parties : le *mode d'appel* et le *mode de retour*. De plus, l'utilisation de ces modes est facilitée si les propriétés associées sont invariantes par substitution.

Définitions

Une **propriété** p est dite **structurelle** si :

$$\forall t \text{ terme, } p(t) \Rightarrow \forall \sigma \text{ substitution, } p(\sigma(t)).$$

Un **mode structurel** (appelé simplement *mode* dans la suite) est un n-uplet (p1,...pn) de propriétés structurelles.

Remarque

Les modes de Warren ne sont pas des modes structurels dans la mesure où la propriété "est une variable" associée au mode "-" n'est pas une propriété structurelle.

Définitions

Un but P(x1,...,xn) est dit **en mode** M=(p1,...,pn) si et seulement si $\forall i \in [1..n]$, pi(xi). On notera M(x1,...,xn).

Un **mode du prédicat P** est un couple de modes (MA, MR) où MA et MR sont des modes structurels d'arité égale à celle de P. MA désigne le **mode d'appel** de P et MR le **mode de retour**.

Un mode (MA, MR) d'un prédicat P est dit **correct** si et seulement si pour tout but P(x1,...,xn) en mode MA, toute solution (σ(x1),...,σ(xn)) est en mode MR.

Remarque

Si tout but P(x1,...,xn) en mode MA n'a jamais de solution, ou si aucun but ne peut être en mode MA (si on a la propriété p telle que $\forall t \in T$, p(t)=faux), le mode (MA, MR) est toujours correct quelque soit MR.
Ces deux cas représentent des cas particulier qui, pour la clarté des démonstrations, ne seront pas pris en compte dans la suite. Le lecteur pourra toutefois vérifier que ces cas se démontrent aisément dans tous les lemmes qui suivent.

Propriété

Le mode (MA, MA) est toujours un mode correct.

Preuve

Soit P(x) un but en mode MA (x désigne ici tout le n-uplet d'arguments).
Toute solution est de la forme σ(x).
Or, les propriétés sont structurelles,
$\Rightarrow$ σ(x) est également en mode MA.

Pour un prédicat P, il existe plusieurs modes de retour corrects pour un mode d'appel donné. Dans le cadre de l'optimisation de la Résolution, il est évident que l'on s'intéresse aux propriétés qui fournissent le plus de précisions sur la structure des termes. Il nous faut donc un moyen de comparer la précision de différents modes.

L'ensemble des propriétés structurelles est muni d'une relation d'ordre partiel ≤ définie par :

$$\forall p1,p2,\quad p1 \leq p2 \Leftrightarrow \forall t \text{ terme},\ p1(t) \Rightarrow p2(t)$$

La relation ≤ peut être étendue aux modes par :

$$(a1,...,an) \leq (b1,...,bn) \Leftrightarrow \forall i \in [1..n],\ ai \leq bi$$

En effectuant son extension aux *modes de prédicats*, on définit :

$$(MA1, MR1) \leq (MA2, MR2) \Leftrightarrow MA1=MA2,\ MR1 \leq MR2$$

Définition

Un **mode minimal** (ou mode "utile") d'un prédicat P est un plus petit mode correct de P.

Un mode (MA, MR) est donc minimal si et seulement si :

$\forall MR'$, (MA, MR') correct et $MR' \leq MR \Rightarrow MR=MR'$

Définition

Soit EP un ensemble de propriétés structurelles sur les termes. EP est dit **fermé** si et seulement si :

$\forall p1,p2 \in EP, \exists p=p1 \wedge p2 \in EP$ (de plus $p \leq p1$ et $p \leq p2$)

L'opérateur de conjonction de propriétés $\wedge$ peut être étendu aux modes par :

$$(a1,...,an) \wedge (b1,...,bn) = (a1 \wedge b1,...,an \wedge bn)$$

Lemme d'inclusion

Soit P un prédicat et (MA, MR) un mode de P défini sur un ensemble EP fermé.

(MA, MR) minimal $\Rightarrow MR = MA \wedge MR$ (et donc $MR \leq MA$)

Preuve

Soit P(x) un but en mode MA $\Rightarrow$ MA(x).

Toute solution est de la forme $\sigma(x)$.

Les propriétés de MA sont structurelles $\Rightarrow$ MA(σ(x)).

De plus, (MA, MR) est un mode correct $\Rightarrow$ MR(σ(x))

$\Rightarrow$ MA(σ(x)) $\wedge$ MR(σ(x))

$\Rightarrow$ (MA, MA$\wedge$MR) est un mode correct pour P

EP est un ensemble fermé $\Rightarrow$ MA$\wedge$MR $\in$ EP*

De plus, MA$\wedge$MR $\leq$ MA et MA$\wedge$MR $\leq$ MR

Or le mode (MA, MR) est un mode minimal

$\Rightarrow$ MR = MA$\wedge$MR

Lemme de transitvité

Soit P un prédicat et (MA1, MR1), (MA2, MR2) deux modes minimaux de P définis sur un ensemble EP fermé.

$$MR1=MA2 \Rightarrow MA2=MR2$$

Preuve

Soit la clause C : Q(x) $\leftarrow$ P(x), P(x).

Si le premier sous-but est en mode (MA1, MR1) et le deuxième en mode (MA2, MR2), avec MA2=MR1, alors le mode (MA1, MR2) est correct pour Q.

Or, la clause C est équivalente à : Q(x) $\leftarrow$ P(x) (logique du 1er ordre).

Donc si (MA1, MR2) est correct pour Q, il l'est également pour P.

Or, (MA1, MR1), (MA2, MR2) sont deux modes minimaux

$\Rightarrow$ MR2 $\leq$ MA2=MR1 et MR1$\leq$MA1

$\Rightarrow$ MR1=MR2 $\Rightarrow$ MA2=MR1

Lemme de conjonction

Soit P un prédicat et (MA1, MR1) et (MA2, MR2) deux modes corrects de P sur un ensemble EP fermé. Alors :

(MA1$\wedge$ MA2, MR1$\wedge$ MR2) correct pour P

Preuve

Soit P(x) un but en mode MA1 et en mode MA2.

$\Rightarrow$ MA1(x) $\wedge$ MA2(x)

(MA1, MR1) et (MA2, MR2) corrects

$\Rightarrow$ toute solution P(σ(x)) est en mode MR1 et en mode MR2

$\Rightarrow$ MR1(σ(x)) $\wedge$ MR2(σ(x))

$\Rightarrow$ (MA1$\wedge$ MA2, MR1$\wedge$ MR2) correct pour P

Corollaire

Soit (MA, MR) un mode correct d'un prédicat P sur un ensemble EP fermé et M un mode quelconque sur EP, de même arité que P. Alors, comme (M, M) est toujours un mode correct :

(MA$\wedge$ M, MR$\wedge$ M) correct pour P

Lemme d'unicité

Soit P un prédicat et (MA, MR1) et (MA, MR2) deux modes de P définis sur un ensemble EP fermé.

(MA, MR1) et (MA, MR2) minimaux $\Rightarrow$ MR1=MR2

Preuve

Soit P(x) un but en mode MA.

(MA, MR1) et (MA, MR2) sont deux modes corrects

$\Rightarrow \forall\sigma$ substitution solution de P(x), MR1(σ(x)) $\wedge$ MR2(σ(x))

En notant MR = MR1$\wedge$MR2, EP est fermé $\Rightarrow$ MR $\in$ EP*

$\Rightarrow$ le mode (MA, MR1$\wedge$MR2) est correct

De plus, MR $\leq$ MR1 et MR $\leq$ MR2 par propriété de l'opérateur $\wedge$.

Donc, si M1 et M2 sont minimaux $\Rightarrow$ MR1 = MR2

Corollaire

Si k désigne la cardinalité de l'ensemble fermé EP, alors tout prédicat

P d'arité n dispose d'exactement k^n modes minimaux sur EP.

Preuve

Pour un prédicat P d'arité n, il existe k^n modes d'appel possibles.
Or, à tout mode d'appel correspond au moins un mode de retour correct, égal au mode d'appel.
De plus, le lemme d'unicité nous montre qu'un mode minimal est unique
⇒ Le nombre de modes minimaux solubles est égal au nombre de modes d'appel possibles, soit k^n.

1.2. Les modes directionnels

Les modes directionnels sont des modes structurels pour lesquels l'ensemble des termes se décompose en :

- termes clos.
- termes quelconques.

L'ensemble de propriétés est donc égal à {Clos, Quelconque} avec :

$$C \leq Q \quad \text{et} \quad C \wedge Q = C$$

Cet ensemble est **fermé**.

Pour un argument donné, les différentes combinaisons de mode d'appel et de retour sont donc :

MA	MR	
quelconque	clos	mode ↑
clos	clos	mode ↓
quelconque	quelconque	mode ?
clos	quelconque	inutile

La dernière ligne de ce tableau correspond à une combinaison de modes inutile en vertu du lemme d'inclusion (MR≤MA). Etant donné que nous ne nous intéressons qu'aux modes minimaux, ce mode sera écarté. Pour un argument donné d'un but donné, ces modes se décomposent donc en :

entrée (↓) : L'argument est toujours un terme clos avant l'effacement du but (et donc également après).

sortie (↑) : L'argument est un terme quelconque, mais le succès de la résolution du but provoque son instanciation à un terme clos, et ceci quelque soit le choix de clause effectué pour l'effacer.

indéterminé (?) : Autres cas.

Dans la mesure où les symboles ↓, ↑ et ? représentent les combinaisons "utiles" d'un mode d'appel et d'un mode de retour, un mode directionnel pour un prédicat P d'arité n peut être considéré comme un terme P(t1,...tn) avec ti∈ {↓, ↑, ?}.

Les modes directionnels se distinguent des modes standard de Warren sur plusieurs points :

- Ils décrivent l'état des arguments aussi bien *avant* qu'*après* la résolution d'un but.

- Ils définissent avec rigueur la direction des données, donc les dépendances de données entre les littéraux d'une clause. Ils permettent ainsi d'effectuer un réordonnancement statique, en fonction du sens des flux de données qu'ils représentent.

- Ils permettent de définir *plusieurs modes par prédicat* en distinguant le mode correspondant à chaque appel du prédicat. Contrairement aux modes de Warren qui n'autorisent qu'une seule déclaration, on peut ainsi spécialiser chaque prédicat pour plusieurs modes.

La définition des modes directionnels étant donnée, nous allons maintenant voir comment prouver leur correction.

1.2.1. *Correction des modes directionnels*

Vérifier la correction des modes est indispensable à l'élaboration d'une technique basée sur leur utilisation. Elle permet de vérifier que, pour un mode donné, la résolution du prédicat appelé avec des termes clos pour ses arguments en mode *entrée* retourne bien des termes clos pour les arguments en mode *sortie*. Elle peut être utilisée à deux niveaux :

- pour détecter les incohérences éventuelles si les modes sont déclarés par

l'utilisateur,

- pour servir de critère de validité à un algorithme d'inférence automatique de modes directionnels.

Une solution peut être trouvée grâce aux théorèmes démontrés pour les *grammaires à attributs*. Ce formalisme, développé pour être utilisé en méta-compilation, a déjà fait l'objet d'études poussées. La similitude entre un arbre d'analyse et un arbre de preuve Prolog a permis à P.Deransart et J.Maluszynski *[Deransart 84a]*, *[Deransart 84b]* d'étudier les dépendances entre données dans les programmes Prolog, à partir de la notion d'*assignation de direction*, similaire à une déclaration de modes directionnels. Cette étude a aboutit à un théorème permettant de prouver la correction de ces assignations de direction. U.S.Reddy a également travaillé sur ce sujet dans le cadre de la transformation de programmes logiques en programmes fonctionnels à l'aide des *modes définis [Reddy 84]*, *[Reddy 85]*. Nous nous sommes inspiré de ces travaux pour mettre en évidence une condition suffisante de correction des modes directionnels.

1.2.1.1. *Principe de la correction des modes directionnels*

Par définition, un prédicat modé P^m est **correct** si et seulement si la résolution d'un but de P, dans lequel tous les arguments en mode *entrée* sont clos, ne retourne que des solutions closes pour les arguments en mode *sortie*. L'ensemble des solutions de P étant l'union des solutions de chacune des clauses de P, il est clair que cette condition doit être vérifiée indépendamment pour chacune de ces clauses.

Considérons une clause quelconque C d'un prédicat modé P^m. L'utilisation des modes consiste à fixer des modes pour chaque littéral apparaissant dans cette clause. Notre clause C modée est donc de la forme :

$$P^m(x) \leftarrow P_1{}^{m1}(x_1), \ldots, P_n{}^{mn}(x_n).$$

Prouver la correction du mode m sur cette clause revient à montrer que la résolution de cette clause produit bien des solutions closes pour les arguments en mode *sortie*. Pour cela il semble nécessaire que les conditions suivantes soient vérifiées :

- Tous les modes mi des littéraux de la queue de la clause sont eux-mêmes corrects.

- Toute variable *consommée* par un littéral (c'est à dire soit en position *entrée* dans un littéral de la queue, soit en position sortie dans le littéral de tête) est *produite* par ailleurs (c'est à dire soit en position *entrée* dans le littéral de tête, soit en position *sortie* dans un littéral de la queue).

Ces conditions posées, il est également important que la résolution de cette clause soit effectuée en respectant les modes mi lors de la résolution de chaque littéral. En d'autre termes, il faut trouver un ordre de résolution tel que l'effacement du $i^{ème}$ littéral ne se produise que si tous les arguments en mode *entrée* dans mi sont clos. Un tel ordre doit donc vérifier :

- Toute variable consommée dans un littéral doit être produite *avant*, c'est à dire produite par un littéral résolu avant.

Ces trois conditions constituent une condition suffisante de correction du mode m. Toutefois, en raison de la première condition, la correction d'un mode de prédicat récursif peut dépendre d'elle-même. Ce problème est traité dans la preuve qui suit en étendant la correction à l'ensemble des clauses du programme.

1.2.1.2. *Condition de correction des modes directionnels*

Si x désigne un n-uplet d'arguments et m un mode directionnel, on désigne par :

in(m, x) l'ensemble des arguments en mode *entrée* de x

out(m, x) l'ensemble des arguments en mode *sortie* de x

ind(m, x) l'ensemble des arguments en mode *indéterminé* de x

Un littéral P(x) est dit **en mode** m si tous les éléments de in(m, x) sont clos.

Un prédicat modé P^m est **correct** si et seulement si la résolution d'un but de P en mode m, en cas de succès, ne produit que des solutions closes pour les arguments de out(m, x).

Définition

Soit C une clause modée : $P^m(x) \leftarrow P_1^{m1}(x_1), \ldots, P_n^{mn}(x_n)$.

C est **bien modée** si :

a) $\forall v$ variable, $v \in out(m,x)$, $v \notin in(m, x) \Rightarrow \exists i, v \in out(m_i, x_i)$

b) il existe un ordre strict $\prec$ (partiel) sur les $P_i^{mi}(x_i)$ tel que :

$\forall i, \forall v$ variable, $v \in in(m_i, x_i)$,

$v \notin in(m, x) \Rightarrow \exists j,\ P_j^{mj}(x_j) < P_i^{mi}(x_i),\ v \in out(m_j, x_j)$

Un prédicat est bien modé pour un mode m si toutes les clauses qui le définissent le sont pour ce mode.

Lemme

Soit C une clause bien modée : $P^m(x) \leftarrow P_1^{m1}(x_1), \ldots, P_n^{mn}(x_n)$.

Soit P(y) un but en mode m, unifiable avec P(x), tel que y et x n'aient pas de variables communes.

Soit θ un p.g.u. de x et y défini par l'algorithme d'unification.

(a) $\forall v \in out(m, \theta(y)),\ \exists i,\ v \in out(mi, \theta(xi))$

(b) $\forall i, \forall v \in in(mi, \theta(xi)),\ \exists j,\ Pj < Pi,\ v \in out(mj, \theta(xj))$

Preuve

Rappelons que toute substitution θ sur un terme x vérifie :

$\forall v \in \theta(x), \exists w \in x,\ v \in \theta(w)$ $\quad (\alpha)$

Remarquons que y en mode m $\Rightarrow \forall v \in in(m,x),\ \theta(v)$ est clos.

(a) Soit v une variable de $out(m, \theta(y)) = out(m, \theta(x))$.

$\exists w \in out(m, x),\ v \in \theta(w)$ d'après (α)

$w \notin in(m, x)$ sinon $\theta(w)$ serait clos et $v \notin \theta(w)$

$\Rightarrow \exists i,\ w \in out(mi, xi)$ car la clause est bien modée

$\Rightarrow \exists i,\ v \in out(mi, \theta(xi))$ car $v \in \theta(w)$

(b) Soit $v \in in(mi, \theta(xi))$

$\exists w \in in(mi, xi),\ v \in \theta(w)$ d'après (α)

$w \notin in(m,x)$ sinon $\theta(w)$ serait clos et $v \notin \theta(w)$

$\Rightarrow \exists j,\ Pj < Pi,\ w \in out(mj, xj)$ car la clause est bien modée

$\Rightarrow \exists j,\ Pj < Pi,\ v \in out(mj, \theta(xj))$ car $v \in \theta(w)$

Remarque : Dans le cas d'une clause d'arrêt $P^m(x) \leftarrow .$, le lemme se réduit à :

out(m, θ(x)) est clos.

Théorème

Soit $X = \{P^m\}$ un ensemble de prédicats bien modés dont les parties droites de clauses n'utilisent que des prédicats bien modés de X.
Tous les modes de X sont corrects.

Preuve

Soit $P^m(x)$ un but en mode m.

On notera $\Gamma = P_1^{m1}(x1),...,P_n^{mn}(xn)$ toute résolvante de P(x), associée à la substitution courante σ.

En supposant que $P_i^{mi}(xi) \in \Gamma$ est un but effaçable par la clause

$C : P_i^{mi}(y) \leftarrow Q_1^{\mu 1}(y1),...,Q_n^{\mu n}(yn)$. grâce à un p.g.u. θ,

On obtient une résolvante

$\Gamma' = P_1^{m1}(\theta(x1)),..., Q_1^{\mu 1}(\theta(y1)),...,Q_n^{\mu n}(\theta(yn)),...,P_n^{mn}(\theta(xn))$

associée à la substitution $\sigma' = \theta o \sigma$

On notera $\aleph = Q_1^{\mu 1}(\theta(y1)),...,Q_n^{\mu n}(\theta(yn))$

Rappelons que l'ensemble des solutions d'un but ne dépend pas de l'ordre d'effacement des sous-buts. Le théorème se réduit donc à montrer qu'il existe un choix de sous-buts tel que la propriété suivante reste vraie pour les résolvantes Γ de P(x) issues de ce choix.

(a) $\forall v \in out(m, \sigma(x)), \exists P\alpha^{m\alpha}(x\alpha) \in \Gamma, v \in out(m\alpha, x\alpha)$

(b) $\exists$ « ordre strict sur Γ,

$\forall P\alpha^{m\alpha}(x\alpha) \in \Gamma, \forall v \in in(m\alpha, x\alpha),$

$\exists P\beta^{m\beta}(x\beta) \in \Gamma, P\beta^{m\beta} \ll P\alpha^{m\alpha}, v \in out(m\beta, x\beta)$

En effet, si la propriété est vraie, elle l'est en particulier pour $\Gamma = []$, et d'après (a), out(m, σ(x)) est clos.

Cette propriété est initialement vraie ($\Gamma = P^m(x)$, $\sigma = \{\}$).

Montrons que si elle est vraie pour (Γ,σ), elle l'est également pour (Γ',σ') obtenue à partir de (Γ,σ) après une inférence logique en choisissant comme sous-but un $P_i^{mi}(xi) \in \Gamma$ sans prédécesseur pour «. Il existe un tel sous-but car « est un ordre sur un ensemble fini.

Un tel $P_i^{mi}(x_i)$ est toujours en mode mi car, d'après (b), ce sous-but ne peut pas avoir de variable dans in(mi, xi).

(a) Soit $v \in out(m, \theta o \sigma(x))$;
$\exists w \in out(m, \sigma(x)), v \in \theta(w)$ (cf. remarque du lemme)
si $w \notin out(mi, xi)$
$\Rightarrow \exists P\alpha^{m\alpha}(x\alpha) \in \Gamma, \alpha \neq i, w \in out(m\alpha, x\alpha)$ d'après (a)
$\Rightarrow \exists P\alpha^{m\alpha}(\theta(x\alpha)) \in \Gamma', v \in out(m\alpha, \theta(x\alpha))$ car $\alpha \neq i$
si $w \in out(mi, xi)$
$\Rightarrow v \in out(mi, \theta(xi))$
$\Rightarrow \exists Q\alpha^{\mu\alpha}(\theta(y\alpha)) \in \Gamma', v \in out(\mu\alpha, \theta(x\alpha))$ d'après lemme (a)

(b) Définissons la relation «' sur les sous-buts de Γ' par la fermeture transitive de l'union des relations suivantes :
l'ordre « sur Γ-Pi
l'ordre $<$ sur $\aleph$
$\forall P\alpha \in \Gamma\text{-Pi}, \forall Q\beta \in \aleph, Q\beta$ «' $P\alpha$
On montre facilement que «' est un ordre strict sur Γ'.
Il nous reste donc à prouver que «' vérifie (b).

Soit $R^{\mu}(\theta(z)) \in \Gamma', v \in in(\mu, \theta(z))$,
$\exists w \in in(\mu, z), v \in \theta(w)$ (remarque du lemme)
si $R^{\mu}(\theta(z)) \equiv P\alpha^{m\alpha}(\theta(x\alpha)), \alpha \neq i$,
$\exists P\beta^{m\beta}(x\beta) \in \Gamma, P\beta^{m\beta} \ll P\alpha^{m\alpha}, w \in out(m\beta, x\beta)$ d'après (b)
si $\beta \neq i$,
$P\beta^{m\beta}(\theta(x\beta)) \in \Gamma'$ et $v \in out(m\beta, \theta(x\beta))$
si $\beta = i$,
$\exists Q\gamma^{\mu\gamma}(\theta(y\gamma)) \in \aleph, v \in out(\mu\gamma, \theta(y\gamma))$ d'après lemme (a)
et $Q\gamma^{\mu\gamma}$ «' $P\alpha^{m\alpha}$ par définition de «'
si $R^{\mu}(\theta(z)) \equiv Q\alpha^{\mu\alpha}(\theta(y\alpha))$
$\exists Q\beta^{m\beta}(\theta(x\beta)) \in \aleph$,
$Q\beta^{m\beta} \ll Q\alpha^{m\alpha}, v \in out(m\beta, \theta(x\beta))$ d'après lemme (b)

$\Rightarrow \forall P\alpha^{m\alpha}(x\alpha) \in \Gamma', \forall v \in in(m\alpha, x\alpha)$,
$\exists P\beta^{m\beta}(x\beta) \in \Gamma, P\beta^{m\beta}$ «' $P\alpha^{m\alpha}, v \in out(m\beta, x\beta)$

Remarque

Il existe un ordre total d'effacement des sous-buts induit de « compatible avec l'interprétation procédurale. De plus, un tel ordre peut être atteint par tri des queues de clauses (ordre <).

Corollaire

Soit X un ensemble de prédicats modés et S une fonction telle que S(X) = {prédicats de X bien modés par des modes de X}. Tout point fixe de S (tel que X=S(X)) ne contient que des modes corrects.

1.2.1.3. *Limitations*

Un modage correct de prédicat ne signifie pas que le prédicat soit soluble. En prenant par exemple le prédicat P défini par la clause :

$$P(x) \leftarrow P(x).$$

Le mode P(↑) est un mode correct alors que la résolution du but ←P(x) ne se termine jamais et ne fournit donc aucune solution.

Il est impossible de déterminer *tous* les modes corrects d'un prédicat ; en effet, la définition d'une clause bien modée ne constitue qu'une condition *suffisante*. Soit par exemple le programme Prolog :

$$Q(f(x), A) \leftarrow.$$
$$Q(g(x), y) \leftarrow.$$
$$P(x) \leftarrow Q(f(y), x).$$

Il est clair que dans tous les cas, la résolution du but ←P(x) ne fournit que des solutions closes (x=A) ; le mode P(↑) est donc correct. Toutefois, sa validité ne peut pas être prouvée à l'aide des outils mathématiques développés ci-dessus dans la mesure où sa correction dépend du fait que le prédicat Q est appelé avec un argument de la forme f(y).

De même, pour le programme suivant :

$$Q(A) \leftarrow.$$
$$P(x) \leftarrow !, Q(x).$$
$$P(x) \leftarrow.$$

le mode P(↑) est correct. Toutefois, sa validité dépend ici de la sémantique particulière du prédicat *cut* et ne peut pas être prouvée par nos outils.

1.2.2. *Classification des modes directionnels*

Les modes directionnels peuvent être décomposés en deux classes :

- Les **modes stricts**, correspondants à ceux dont le mode de retour ne contient que des arguments clos. Un mode strict d'un prédicat P d'arité n est donc un terme P(t1,...,tn) avec ti ∈ {↓,↑}.

- Les **modes non-stricts**, regroupant les modes qui ne sont pas stricts, donc ceux comportant au moins un argument en mode *indéterminé*.

Les modes stricts peuvent eux-mêmes être partagés en deux catégories :

- Les **modes stricts fonctionnels**, c'est à dire les modes stricts pour lesquels les clauses peuvent être bien modées en n'utilisant pour les sous-buts que des modes stricts, eux-mêmes fonctionnels. En d'autres termes, si X est un ensemble de prédicats bien modés avec des modes stricts et si X=S(X), alors tous les modes de X sont des modes stricts fonctionnels corrects.

- Les **modes stricts non-fonctionnels**, c'est à dire les modes stricts qui ne sont pas fonctionnels.

Par exemple, dans le programme :

P(x) ← Q(x, y). (1)
Q(A, B) ←. (2)

le mode P(↑) est un mode strict fonctionnel car la clause (1) peut être bien modée en utilisant le mode Q(↑,↑), qui est un mode strict fonctionnel.

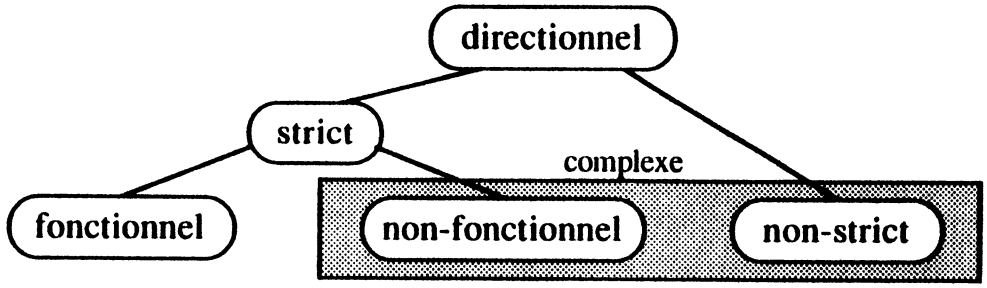
Par contre, dans le programme :

P(x) ← Q(x, y). (1)
Q(A, z) ←. (2')

le mode P(↑) est un mode strict non-fonctionnel car le bon modage de la clause

(1) nécessite l'utilisation du mode Q(↑,?) qui est un mode non-strict.

Les modes stricts non-fonctionnels et les modes non-stricts forment l'ensemble des *modes complexes*. Les modes directionnels peuvent donc se hiérarchiser en :



2. Application des modes directionnels

La connaissance des modes directionnels corrects pour les prédicats d'un programme permet d'acquérir une information importante sur l'action de la résolution d'un prédicat sur son environnement. En réalité, cette information peut être utilisée à deux niveaux :

Globalement

Les modes peuvent être utilisés pour ordonner les littéraux des queues de clauses. On peut ainsi modifier la stratégie de résolution pour une meilleure efficacité.

Localement

La connaissance des arguments clos permet de sélectionner un meilleur algorithme d'unification, et autorise également la mise en oeuvre de nouvelles techniques pour la construction des termes.

Maluszynski et Deransart *[Deransart 84a]*, *[Deransart 84b]* ont abordés le premier domaine en définissant la stratégie *Data Driven* en fonction des *assignations de direction*. Par ailleurs, dans le domaine local, Maluszynski et Komorowski *[Maluszynski 85]* ont montré que l'unification peut dans certains cas être remplacée par la *reconnaissance de formes*. Dans ce qui suit, nous étudions de manière plus approfondie les possibilités offertes par les modes directionnels dans ces deux domaines. Les aspects stratégiques sont abordés par l'intermédiaire de l'ordonnancement des queues de clauses, et les optimisations locales par l'étude des modes stricts en particulier.

Nous supposons pour l'instant disposer de tous les modes corrects minimaux (prouvés comme tels par le théorème) pour tous les prédicats d'un programme. Nous verrons plus loin comment les obtenir de manière automatique.

2.1. Ordonnancement des queues de clauses

Nous avons vu auparavant que chaque prédicat d'arité n est doté de 2^n modes minimaux, chacun de ces modes pouvant être utilisé en partie droite d'une clause. Chacun de ces modes représentant une utilisation différente du prédicat, il est intéressant de pouvoir ordonner les littéraux des queues des clauses du prédicat pour chacun d'eux. Cet ordonnancement consiste non seulement à choisir un ordre strict de résolution des sous-buts, mais également à choisir pour chaque sous-but un mode de résolution.

Or, d'après nos résultats théoriques, le "bon modage" d'une clause nous permet de déterminer un choix de mode et un tel ordre. De plus, on sait que :

- Un tel ordre existe toujours. En effet, si un mode est prouvé correct, c'est qu'il existe un bon modage de ses clauses, et par conséquent un moyen de les ordonner.

- Cette ordre vérifie la propriété : la "connaissance" (terme) consommée par un littéral est toujours produite avant (soit par un autre littéral placé avant, soit par un argument de l'entête).

Appelons **ordonnancement** une clause dont les littéraux sont ordonnés et modés de manière telle que :

- La clause est bien modée.

- L'ordre des littéraux (ordre d'exécution $<^e$) est tel que :

$$\forall L1, L2 \text{ littéraux}, \; L1 < L2 \Rightarrow L1 <^e L2$$

Un tel ordonnancement correspond à la stratégie *Data Driven* définie dans *[Deransart 84a]*, où un sous-but ne peut être effacé que si tous ses arguments en mode *entrée* sont des termes clos.

Il existe éventuellement plusieurs ordonnancements possibles en fonction des deux degrés de liberté suivants :

- Le choix des modes des littéraux, qui influence directement les relations de dépendance entre les littéraux.

- L'ordre induit des modes des littéraux n'est que partiel ; on peut donc en tirer plusieurs ordres totaux.

Dans le cadre d'une mise en oeuvre efficace de la Résolution, il est intéressant de pouvoir déterminer statiquement le meilleur ordonnancement pour chaque clause du programme, et ceci pour chaque mode des prédicats auxquels elles appartiennent.

D'une façon générale, le nombre d'ordonnancements possibles est assez restreint. Reprenons l'exemple du prédicat *reverse*, cité comme cas critique dans le premier chapitre :

append(NIL, x, x) ←.
append(a.x, y, a.z) ← append(x, y, z).

reverse(a.x, r) ← reverse(x, y), append(y, a.NIL, r).
reverse(NIL, NIL) ←.

Les modes corrects minimaux pour les prédicats *append* et *reverse* sont les suivants :

append : {(↓,↓,↓), (↓,↓,↑), (↑,↑,↓), (↓,↑,↓), (↑,↓,↓), (↓,?,?), (?, ↓,?), (?,?,?)}
reverse : {(↓,↓), (↓,↑), (↑,↓), (?,?)}

La première clause du prédicat *reverse* n'a qu'un seul ordonnancement possible pour chacun des modes stricts (↓,↑) et (↑,↓) :

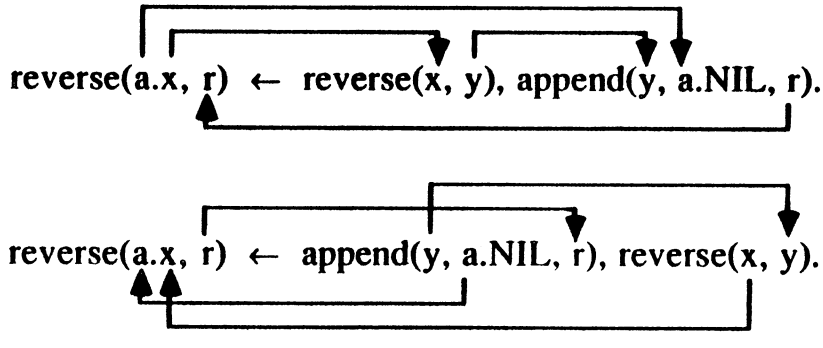
reverse(↓a.x, ↑r) ← reverse(↓x, ↑y), append(↓y, ↓a.NIL, ↑r).
reverse(↑a.x, ↓r) ← append(↑y, ↑a.NIL, ↓r), reverse(↑x, ↓y).

et deux pour chacun des modes (↓,↓) et (?,?) :

reverse(↓a.x, ↓r) ← reverse(↓x, ↑y), append(↓y, ↓a.NIL, ↓r).
reverse(↓a.x, ↓r) ← append(↑y, ↓a.NIL, ↓r), reverse(↓x, ↓y).

reverse(?a.x, ?r) ← reverse(?x, ?y), append(?y, ?a.NIL, ?r).
reverse(?a.x, ?r) ← append(?y, ?a.NIL, ?r), reverse(?x, ?y).

Pour les modes (↓,↑) et (↑,↓), les flux de données entre les différents littéraux peuvent être mieux illustrés par :



On peut constater que ces ordonnancements correspondent bien à un ordre de résolution correct pour toute configuration d'arguments correspondant au mode de l'entête. Seuls les mode (↓,↓) et (?,?) autorisent

deux ordres possibles et posent donc un problème de choix.

2.1.1. *Limitations et critères de sélection*

Les impératifs de la programmation rendent indispensable l'utilisation de certains prédicats qui permettent d'effectuer des effets de bords, comme les entrées/sorties, mais qui en revanche sortent de la logique du premier ordre. La présence de ces prédicats dans un programme influence directement l'ordonnancement des clauses puisqu'il est strictement nécessaire de préserver leur ordre mutuel.

En outre, l'usage du prédicat *cut* est devenu maintenant pratique courante dans la programmation en Prolog. De ce fait, même si les modes directionnels permettent de limiter grandement l'utilisation du *cut*, il convient de tenir compte de sa présence.

L'ordre d'écriture entre deux littéraux doit par conséquent être respecté dans les deux cas suivants :

- quand ils sont séparés par un *cut*,
- lorsqu'ils correspondent à des prédicats effectuant des effets de bord.

Par ailleurs, lorsqu'un choix existe, la sélection d'un "meilleur" ordonnancement pour un mode donné doit s'effectuer en appliquant certains critères de sélection. Parmi ces critères, les plus importants sont :

- Réduire le coût de résolution. Cette réduction peut notamment être obtenue par *élimination de certaines branches infinies.*

- Utiliser un nombre minimum de modes pour tout le programme. Ce critère vise à minimiser la taille du code généré, si chaque prédicat fait l'objet d'une génération de code spécifique pour chaque mode utilisé.

- Repousser les appels récursifs en fin de clause. Ce critère ouvre la possibilité d'appliquer l'*optimisation du dernier appel [Warren 80]*, très avantageuse lorsque cet appel est une récursion.

- Extraction d'un parallélisme maximum en fournissant le plus de sous-buts indépendants. Ce critère est examiné plus précisement dans un paragraphe ultérieur.

Ces différents critères ne sont pas forcément compatibles entre eux. En

particulier, le nombre de modes réellement utilisé est généralement supérieur au minimum possible, en raison de l'application des autres critères.

2.1.2. *Sélection par réduction du coût de résolution*

La stratégie de construction de l'arbre de recherche influe directement sur la taille de celui-ci, et donc sur le temps nécessaire à son parcours. Nous avons vu au chapitre précédent qu'une sélection judicieuse des modes et de l'ordonnancement peut conduire au choix d'un "meilleur" arbre de recherche. Nous avons également vu que le coût de ces ordonnancements est fortement dépendant des arguments des prédicats. Les modes directionnels, et particulièrement les modes stricts, offrent un bon moyen de calculer ces coûts avec une meilleure précision.

Pour ce faire, nous allons étendre les fonctions de calcul de coût présentées, au chapitre précédent, en y incluant la notion de mode. *Card* devient par conséquent une fonction évaluant la cardinalité *maximale* d'un prédicat *pour un mode donné*. Soit P un prédicat formé de clauses Ci ($1 \le i \le k$) ; pour un mode m donné, nous aurons alors :

$$Card\,(P^m) = \sum_{i=1}^{k} Card\,(Ci^m)$$

Soit C=(P←Q1,...,Qn) une clause de P dans laquelle mi désigne le mode du littéral Qi. Quelque soit le choix effectué pour les mi, et l'ordre de résolution choisi, nous avons :

$$Card\,(C^m) = \prod_{i=1}^{n} Card\,(Qi^{mi})$$

Si P est un prédicat formé de clauses Ci ($1 \le i \le k$), nous obtenons en restreignant la fonction de coût à une clause :

$$Coût(P^m) = \sum_{i=1}^{k} Coût(Ci^m)$$

Soit C=(P←Q1,...,Qn) une clause de P. En notant $Ci=Card(Qi^{mi})$ et $Ki=Coût(Qi^{mi})$, nous obtenons :

$$Coût\,(C^m) = K1 + \sum_{i=2}^{n}(Ki * \prod_{j=1}^{i-1} Cj)$$

Les modes permettent de calculer plus finement la cardinalité d'un prédicat et permettent par conséquent un calcul du coût beaucoup plus précis.

2.1.2.1. *Cas des prédicats non-récursifs*

Si un prédicat P est composé exclusivement de faits, sa cardinalité pour un mode directionnel m donné peut être évaluée en s'appuyant sur ce qui suit :

- Soit T la transformation telle que : T(Q(x), m) = Q(y), où y représente le n-uplet de tous les arguments de Q(x) en mode *entrée* dans m.

- Soit F l'ensemble de tous les faits du prédicat P.

- Soit $\mathcal{R}$ la relation telle que, si q et q' représentent deux éléments de F :

  $$q\ \mathcal{R}\ q' \Leftrightarrow T(q, m)\ \text{s'unifie avec}\ T(q', m).$$

- La relation $\mathcal{R}$ définit des classes Si. En désignant par c(Si) la fonction retournant le nombre d'éléments de la classe Si, la cardinalité du prédicat P pour le mode m vérifie alors :

  $$Card(P^m) = max(c(Si))$$

Reprenons notre exemple du chapitre 2 et évaluons plus finement ses coûts, en appliquant les résultats ci-dessus :

père(Marie, Pierre) ←. mère(Marie, Anne) ←.
père(Alice, Pierre) ←. mère(Jean, Marie) ←.
père(Jean, Marc) ←.

parents(x, p, m) ← père(x, p), mère(x, m).

La clause du prédicat *parents* peut être ordonnée et modée de deux manières différentes pour le mode (↑,↑,↑) :

(1) parents(↑x, ↑p, ↑m) ← père(↑x, ↑p), mère(↓x, ↑m).
(2) parents(↑x, ↑p, ↑m) ← mère(↑x, ↑m), père(↓x, ↑p).

Le coût de résolution des prédicats *père* et *mère* est égal au nombre des

faits qui les définissent, c'est à dire 3 et 2 respectivement, quelque soit leur mode. Leur cardinalité est par contre variable et peut être calculée plus finement :

Card(père(↑,↑)) = 3	*Card*(mère(↑,↑)) = 2
Card(père(↑,↓)) = 2	*Card*(mère(↑,↓)) = 1
Card(père(↓,↑)) = 1	*Card*(mère(↓,↑)) = 1
Card(père(↓,↓)) = 1	*Card*(mère(↓,↓)) = 1

Grâce à ces valeurs, nous pouvons calculer plus finement le coût et la cardinalité du prédicat *parents* pour les deux ordonnancements :

(1) *Coût*(parents(↑,↑,↑)) = 9	*Card*(parents(↑,↑,↑)) = 3
(2) *Coût*(parents(↑,↑,↑)) = 8	*Card*(parents(↑,↑,↑)) = 2

On peut constater que le calcul de la cardinalité donne des résultats différents suivant l'ordonnancement considéré. En effet, si les modes affinent de façon notable le calcul de la cardinalité, leur indépendance vis-à-vis de la valeur réelle des arguments oblige à considérer le pire des cas. Il est toutefois clair que la cardinalité réelle est invariable en fonction de l'ordonnancement choisi. Comme il s'agit d'une borne maximum, la cardinalité du prédicat pour un mode donné peut donc être prise comme le minimum des cardinalités calculées pour chaque ordonnancement.

Considérons maintenant le prédicat *parents* pour le mode (↑,↑,↓). Ce mode permet également deux ordonnancements qui sont :

(1) parents(↑x, ↑p, ↓m) ← père(↑x, ↑p), mère(↓x, ↓m).
(2) parents(↑x, ↑p, ↓m) ← mère(↑x, ↓m), père(↓x, ↑p).

Le calcul du coût et de la cardinalité de ces deux ordonnancements produit :

(1) *Coût*(parents(↑,↑,↓)) = 9	*Card*(parents(↑,↑,↓)) = 3
(2) *Coût*(parents(↑,↑,↓)) = 5	*Card*(parents(↑,↑,↓)) = 1

Ces résultats montrent que le deuxième ordonnancement est le meilleur et que la cardinalité ne peut dépasser 1 pour ce mode. On peut en outre constater que les résultats obtenus sont considérablement affinés par rapport à ceux obtenus sans les modes. Sur cet exemple, la plus grosse différence se situe dans le calcul de la cardinalité qui est de 6 sans les modes et qui peut être

réduite à 1 selon le mode considéré. Ce résultat se répercute évidemment sur le calcul des coûts de tous les prédicats utilisant le prédicat *parents*.

Remarques

- La cardinalité et le coût des prédicats prédéfinis est généralement égale à un pour tous les modes. En cas contraire, elles peuvent être fixées par le concepteur du système.

- Bien que ces relations ne travaillent que sur des cardinalités ou des coûts *maximums*, il est clair que ce critère de choix ne fournit qu'un moyen de minimiser le coût de résolution pour le cas le plus défavorable. Il peut parfaitement arriver qu'un ordonnancement sélectionné provoque, *pour une instanciation particulière de ses arguments*, un coût réel supérieur à celui qui aurait été obtenu avec un autre ordonnancement.

2.1.2.2. *Cas des prédicats récursifs*

Nous avons pu constater que l'élimination de certaines branches infinies peut s'effectuer automatiquement en appliquant purement la stratégie *Data Driven*, lorsqu'un seul ordonnancement est possible. C'est le cas du prédicat *reverse* pour lequel, dans la limite des connaissances apportées par les modes directionnels, le bon choix est toujours effectué.

Lorsqu'une clause qui contient un appel à un prédicat récursif peut être ordonnée de plusieurs manières, le coût est virtuellement infini pour chacun de ces ordonnancements et ne permet donc pas d'effectuer une sélection.

Bien que ce problème ne puisse avoir de réponse absolue, nous allons exposer un moyen permettant la comparaison des coûts des différents ordonnancements entre eux.

Soit, par exemple, le programme :

objet(A) ←.
objet(S(0)) ←.
int(0) ←.
int(S(x)) ← int(x).
res(x) ← int(x), objet(x).

doté de tous les modes possibles pour tous les prédicats. L'ordonnancement de la clause du prédicat *res* pour le mode (↑) autorise deux possibilités :

(1) res(↑x) ← int(↑x), objet(↓x).
(2) res(↑x) ← objet(↑x), int(↓x).

Il est clair que la première solution entraîne un bouclage de la résolution (cf. chapitre 2).

Une première manière de résoudre le problème a été donnée dans le chapitre 2 et consiste à choisir l'ordonnancement où l'appel au prédicat récursif est placé le plus loin possible dans la queue de clause. Cette solution est basée sur le fait que plus un prédicat est instancié, plus sa cardinalité décroit. Dans le contexte des modes directionnels, ceci devient :

Si m1=(MA1, MR1) et m2=(MA2, MR2) sont deux modes corrects du prédicat P, alors :

$$MA1 \leq MA2 \Rightarrow Card(P^{m1}) \leq Card(P^{m2})$$

Cette règle est suffisante dans la plupart des cas mais ne permet pas le calcul précis des coûts et cardinalités. Pour celà, une solution partielle est fournie par la *détection du déterminisme*, quipermet de savoir si la cardinalité d'un prédicat, même récursif, est inférieure ou égale à un, c'est à dire que la résolution du prédicat produit au plus une solution.

Détection du déterminisme

Sawamura, Takeshima et Kato *[Sawamura 85]* proposent une technique permettant une telle détection en se basant sur la présence de *cuts* dans le programme. Notre objectif étant d'utiliser les modes comme outil de contrôle, la présence des *cuts* devient exceptionnelle. Nous nous inspirons de la méthode pour donner un moyen de détecter le déterminisme, mais sur la base des modes. Nous parlerons donc de détermisme d'un prédicat pour un mode.

On peut constater que si un prédicat n'est pas déterministe pour un mode (c'est à dire lorsqu'il existe au moins un but qui produit plusieurs solutions), il peut tout de même l'être pour un but particulier dans ce mode. En reprenant notre prédicat *père* donné précédemment, ce prédicat n'est pas déterministe pour le mode (↑,↓) alors que le but ←père(x,Marc) ne retourne bien qu'une unique solution. On distingue donc deux catégories de déterminisme :

- Le **déterminisme global**, qui ne dépend que d'un mode.

- Le **déterminisme local**, qui est associé à un but particulier et qui dépends

donc d'un n-uplet d'arguments particulier.

Nous dirons qu'un but quelconque $P^m(x)$ est déterministe s'il est localement déterministe ou bien si le prédicat P qu'il référence est globalement déterministe pour le mode m.

En reprenant T la transformation telle que : T(Q(x), m) = Q(y), où y représente le n-uplet de tous les arguments de Q(x) en mode *entrée* dans m, un prédicat modé P^m est *globalement déterministe* si les deux conditions suivantes sont vérifiées :

- Quelque soient Q et Q' deux littéraux des têtes des clauses qui définissent P, alors :

$$Q \neq Q' \Rightarrow T(Q, m) \text{ ne s'unifie pas avec } T(Q', m)$$

- Tous les sous-buts des queues des clauses de P sont déterministes.

Un but $P^m(x)$ est *localement déterministe* si les deux conditions suivantes sont vérifiées :

- Le but P(x) ne s'unifie qu'avec l'entête d'une seule clause C de P.

- Tous les sous-buts de la queue de la clause C sont déterministes.

Utilisons ces critères pour mettre en évidence l'éventuel déterminisme des prédicats *append* et *reverse* :

append(NIL, x, x) ←.
append(a.x, y, a.z) ← append(x, y, z).

reverse(NIL, NIL) ←.
reverse(a.x, z) ← reverse(x, y), append(y, a.NIL, z).

Les modes (↓,↓,↓), (↓,↓,↑) et (↓,↑,↓) sont déterministes pour *append*. De même, les modes (↓,↓) et (↓,↑) le sont pour *reverse*. Ce critère est toutefois insuffisant pour montrer le déterminisme de *reverse* pour le mode (↑,↓).

En réalité, le fait que plusieurs ordonnancements soient possibles pour les clauses peut influer sur le calcul du déterminisme. Prenons par exemple le programme :

objet(S(0)) ←.
int(0) ←.
int(S(x)) ← int(x).
res(x) ← int(x), objet(x).

De façon évidente, le prédicat *objet* est déterministe pour les modes (↓) et (↑) alors que le prédicat *int* ne l'est que pour le mode (↓). Qu'en est-il du prédicat *res* pour le mode (↑) ? Ce prédicat peut être ordonné des deux manières suivantes :

(1) res(↑x) ← int(↑x), objet(↓x).
(2) res(↑x) ← objet(↑x), int(↓x).

Le premier ordonnancement fait référence au prédicat *int* dans le mode (↑). Or, ce mode n'est pas déterministe, et ne permet donc pas d'affirmer le déterminisme de *res*. Inversement, le second ordonnancement ne fait apparaître que des modes déterministes et permet ainsi de déduire que *res* l'est également.

Le déterminisme d'un prédicat étant indépendant de l'ordre de résolution de ses sous-buts, un prédicat est donc déterministe pour un mode s'il existe au moins un ordonnancement de ce mode pour lequel il l'est.

2.1.3. *Conclusion*

Les prédicats récursifs ou faisant appel à un prédicat récursif se caractérisent par leur coût infini : il est donc impossible d'effectuer une sélection du meilleur ordonnancement sur cette seule base. Une approche consiste à déterminer d'abord :

- La cardinalité et le coût de tous les prédicats non-récursifs du programme, pour tous leurs modes. Les meilleurs ordonnancements peuvent alors être sélectionnés pour eux.

- Les déterminismes possibles pour tous les modes et tous les prédicats récursifs du programme.

On peut alors appliquer une heuristique pour réaliser la sélection du meilleur ordonnancement pour toutes les clauses comportant un appel à un prédicat récursif. Cette heuristique consiste à fixer tous les coûts infinis à une

valeur finie arbitrairement grande, très supérieure à la valeur maximale obtenue pour les prédicats non-récursifs. On peut ainsi profiter des résultats obtenus par la détection du déterminisme.

Une autre solution, plus simple et plus pragmatique, consiste à toujours choisir, quand le choix existe, l'ordre le plus proche de l'ordre d'écriture. Cette solution, utilisée dans Starlog *[Oudot 86]*, n'enlève rien à l'intérêt pratique de la connaissance du déterminisme, notamment pour la génération de code, et semble donner d'assez bon résultats.

2.2. Intérêt particulier des modes stricts

Une facilité importante du langage Prolog est la possibilité de manipuler des termes complexes, comportant des variables. L'utilisation de ces termes peut être décomposée en deux cas :

- Fournir *explicitement* à l'utilisateur des résultats partiellement instanciés.

- Propager la connaissance à l'intérieur d'une résolvante. C'est une utilisation *implicite* et *temporaire* dans la mesure où la complexité du terme considéré décroit au fur et à mesure de l'évolution du processus de résolution, pour finalement aboutir à un terme clos.

Illustrons ce deuxième cas par l'intermédiaire du programme suivant :

eval(A, A) ←.
eval(B, B) ←.
eval(cons(x, y), u.v) ← eval(x, u), eval(y, v).
← eval(cons(A, B), r).

Les différentes étapes de sa résolution sont les suivantes :

Résolvante	**Substitutions**
← eval(cons(A, B), r).	
← eval(A, u), eval(B, v).	x/A, y/B, r/**u.v**
← eval(B, v).	u/A (⇒ r/**A.v**)
←.	v/B (⇒ r/**A.B**)

L'évolution du terme attaché à la variable r est caractéristique de

l'utilisation temporaire d'un terme complexe. En effet, l'unification effectue la substitution de r avec un terme complexe, dont les différentes composantes sont instanciés ultérieurement à des termes clos.

L'implantation des termes complexes et des structures de données dérivées *[Bruynooghe 82]* est un facteur important dans le coût de la Résolution. Nous proposons ci-après une méthode qui permet d'éliminer les termes complexes temporaires, quand cela est possible. Cette méthode, basée sur l'utilisation des modes stricts, permet une mise en oeuvre plus efficace de la résolution.

Maluszynski et Komorowski *[Maluszynski 85]* ont par ailleurs montré que les programmes Prolog pouvant être résolus en n'utilisant que des modes stricts peuvent se résoudre en remplaçant l'unification par la reconnaissance de formes (*pattern-matching*). Nous verrons dans la suite que les modes stricts fonctionnels alliés à l'expansion de l'unification permettent de transformer cette reconnaissance de formes en simple *égalité*.

2.2.1. *Modes stricts et expansion de l'unification*

Les modes stricts définissent les arguments d'un prédicat comme étant clos avant ou après la résolution de ce prédicat. Par contre, aucune restriction n'est appliquée aux termes locaux à une clause.

Dans notre exemple précédent, le mode strict (↓,↑) pour *eval* permet de savoir que le terme attaché à la variable r sera clos à la fin de la résolution d'*eval*. Il serait donc intéressant de retarder la construction de ce terme jusqu'à ce que toutes ses parties constituantes soient elles-même closes ; ceci revient à dire qu'il faut retarder l'unification de l'argument correspondant à cette variable.

L'expansion de l'unification est un bon outil pour mettre en évidence ce retardement. Par exemple, l'expansion primaire de l'unification génère pour notre programme :

consc(x, y, cons(x, y)) ←.
cons(x, y, x.y) ←.
eval(A, A) ←.
eval(B, B) ←.
eval(t1, t2) ← **consc**(x, y, t1), eval(x, u),
eval(y, v), **cons**(u, v, t2).

← eval(cons(A, B), r).

Nous faisons apparaître les littéraux produits par l'expansion dans un ordre qui n'est pas innocent ; il correspond en effet à une résolution qui évite l'utilisation d'un terme complexe temporaire. L'évolution de cette résolution est la suivante est la suivante:

Résolvante	**Substitutions**
← eval(cons(A, B), r).	
← consc(x, y, cons(A, B)), eval(x, u), eval(y, v), cons(u, v, r).	t1/cons(A, B), t2/r
← eval(A, u), eval(B, v), cons(u, v, r).	x/A, y/B
← eval(B, v), cons(A, v, r).	u/A
← cons(A, B, r).	v/B
←.	r/A.B

Le terme complexe temporaire présent dans une résolution classique est éliminé grâce à l'expansion de l'unification. Celle-ci fait apparaître la construction de ce terme sous la forme d'un sous-but que l'on peut résoudre au moment le plus approprié. Sa résolution effective n'est effectuée que lorsque les deux parties du terme à construire sont connues.

2.2.1.1. *Production des littéraux constructeurs*

Les littéraux constructeurs produits par l'expansion (cf. Chapitre 2) peuvent être insérés dans l'ordonnancement d'une clause de deux manières :

- Après avoir sélectionné le meilleur ordonnancement,

- Avant le choix d'un ordonnancement, en les traitant de façon équivalente à un littéral ordinaire.

Dans ce qui suit, nous analysons ces deux solutions ; les constructeurs sont représentés lors de cette analyse par :

nompred(sous-terme$_1$,...,sous-terme$_n$, terme construit)

Remarque

Pour les modes stricts non-fonctionnels, il se peut que certains termes de la clause soient placés en mode *indéterminé*. Dans ce cas, l'expansion de

ce terme ne permet pas d'exploiter l'avantage cité ci-dessus. Nous ne traiterons donc pas ce cas.

Production postérieure à la sélection

Une fois sélectionné un ordonnancement, la production des prédicats constructeurs est effectuée de la façon suivante :

a) Pour chaque argument constituant un terme complexe dans le littéral d'entête :
   - Si l'argument est en mode *entrée*, le littéral constructeur correspondant est placé avec le terme construit en mode *entrée* et les autres en mode *sortie*. Ce littéral est alors placé en *tête* de la queue de clause.
   - Si l'argument est en mode *sortie*, le littéral constructeur correspondant est placé avec le terme construit en mode *sortie* et les autres en mode *entrée*. Ce littéral est alors placé en *queue* de la clause.

b) Pour chaque argument constituant terme complexe dans les littéraux de la queue :
   - Si l'argument est en mode *entrée*, le littéral constructeur correspondant est placé avec le terme construit en mode *sortie* et les autres en mode *entrée*. Ce littéral est alors placé juste *avant* le littéral qui contient le terme expansé.
   - Si l'argument est en mode *sortie*, le littéral constructeur correspondant est placé avec le terme construit en mode *entrée* et les autres en mode *sortie*. Ce littéral est alors placé juste *après* le littéral qui contient le terme expansé.

Si l'un des arguments d'un littéral constructeur est une constante, il est systématiquement placé en mode *entrée*.

Exemple : Soit la clause bien modée :

$$P(\downarrow x.y, \uparrow u.v) \leftarrow Q(\downarrow x, \uparrow u:r), Q(\downarrow y, \uparrow v).$$

Elle se transforme en :

$$P(\downarrow t1, \uparrow t2) \leftarrow cons(\uparrow x, \uparrow y, \downarrow t1), Q(\downarrow x, \uparrow t3), cons(\uparrow u, \uparrow r, \downarrow t3), Q(\downarrow y, \uparrow v), cons(\downarrow u, \downarrow v, \uparrow t2)$$

Cette règle de construction assure la conservation du bon modage de la clause. Les littéraux constructeurs jouent le rôle de relais entre les littéraux produisant les termes et les littéraux qui les consomment.

Production antérieure à la sélection

L'ordonnancement des clauses nécessite dans ce cas la connaissance des modes stricts corrects correspondant aux prédicats produits par l'expansion primaire. La génération de ces modes est en réalité assez simple. En effet, seule deux possibilités existent : soit tous les sous-termes sont clos à l'appel pour permettre la construction du terme résultat, soit celui-ci est clos à l'appel pour permettre sa décomposition en sous-termes. L'ensemble des modes stricts corrects d'un prédicat constructeur vérifie donc :

tous les arguments représentant les sous-termes sont en mode *entrée*
ou
l'argument représentant le terme construit est en mode *entrée*

Pour le prédicat *cons*, les modes stricts sont par conséquent :

cons(↓,↓,↑), cons(↑,↑,↓), cons(↓,↓,↓), cons(↓,↑,↓), cons(↑,↓,↓)

La production antérieure à la sélection présente l'avantage de ne pas nécessiter de traitement particulier pour les prédicats constructeurs, insérés automatiquement à la bonne place par le traitement général de l'ordonnancement.

2.2.2. *Conséquences sur l'évolution de la résolution*

L'utilisation des modes stricts alliée à l'expansion de l'unification a de multiples conséquences sur le comportement et l'évolution de la résolution par rapport à une stratégie plus classique. Les principales conséquences sont :

- L'*unification* dans sa forme générale est totalement *supprimée* et remplacée par la reconnaissance de formes. L'expansion la spécialise encore en permettant l'utilisation de l'*égalité*.

- La *gestion mémoire* est *améliorée* grâce à un meilleur contrôle de la construction effective des termes.

- Les termes manipulés n'étant que des termes clos, l'opération de

déréférencage peut être *supprimée.*

- La perte de la bi-directionnalité des arguments entraine la perte d'une partie des informations fournies par les termes, ce qui peut parfois induire des *calculs inutiles.*

- Les programmes Prolog exécutables en utilisant seulement les modes stricts peuvent être transformés en *programmes fonctionnels*, permettant l'utilisation de la réduction au lieu de la résolution.

- L'indépendance entre les sous-buts peut être détecté de façon simple, permettant d'envisager l'exploitation du *parallélisme ET.*

Nous analysons ci-après chacune de ces conséquences.

2.2.2.1. *Simplification de l'unification*

Nous n'avons abordé jusqu'ici que l'expansion primaire, qui met en évidence les constructions de termes. L'expansion secondaire, qui met en évidence les unifications effectives conduit pour sa part à des optimisation importantes en tenant compte des modes stricts.

Les modes stricts corrects du prédicat *unification* généré par l'expansion secondaire sont : (↓,↓), (↓,↑), (↑,↓). En tenant compte de la propriété "terme clos" des termes entrant, et de la décomposition des termes complexes par l'expansion primaire, ces différentes utilisations de l'unification peuvent être simplifiées en :

mode	**algorithme**
(↓,↓)	égalité
(↓,↑)	affectation
(↑,↓)	affectation

L'unification généralisée peut donc être remplacée par l'égalité ou l'affectation, en fonction du mode où elle se trouve. Nous montrons ci-après que cette dernière peut en fait être supprimée.

Production des littéraux d'unification

L'utilisation d'une affectation dans le cadre d'une forme expansée qui n'utilise que des termes clos correspond en fait à l'introduction d'un sous-but

parfaitement inutile. En effet, si une clause ordonnée contient le littéral unification(x,y) utilisé en mode (↓,↑), il peut être supprimé en substituant dans la clause toutes les occurrences de y par x (et réciproquement pour le mode (↑,↓)).

Soit par exemple la clause :

append(NIL, x, x) ←.

dont l'expansion complète nous donne :

append(t, x, y) ← unification(t, NIL), unification(x, y).

L'ordonnancement selon les modes stricts de cette clause produit pour le mode (↓,↓,↑) :

append(↓t, ↓x, ↑y) ← unification(↓t, ↓NIL), unification(↓x, ↑y).

On peut constater que le dernier littéral est inutile ; la clause peut donc être simplifiée en :

append(↓t, ↓x, ↑x) ← égalité(↓t, ↓NIL)

De même, on obtient pour le mode (↑,↑,↓) la clause :

append(↑t, ↑x, ↓y) ← unification(↑t, ↓NIL), unification(↑x, ↓y).

qui peut être simplifiée en :

append(↑NIL, ↑x, ↓x) ←.

L'expansion secondaire réalisée *avant* la sélection d'un ordonnancement peut donc conduire à la production de littéraux d'unification parfaitement inutiles (affectations). Il est donc préférable de réaliser l'expansion secondaire après sélection de l'ordonnancement de façon à ne produire que les unifications utiles, c'est à dire les tests d'égalité. Ceci peut être réalisé de la façon suivante :

Pour chaque argument en mode *entrée* du littéral d'entête ou en mode *sortie* d'un littéral de la queue :

- Si cet argument est une constante C, on la substitue par une variable t et on produit le littéral *égalité*(C, t).

- Si cet argument est une variable x qui apparait deux fois dans le littéral avec le même mode, on substitue l'une de ses occurrences par une variable t et on produit le littéral *égalité*(x, t).

Dans tous les cas, le littéral *égalité* introduit est placé en mode (↓,↓), immédiatement après le littéral traité, donc au début de la queue de clause s'il s'agit du littéral d'entête.

2.2.2.2. *Amélioration de la gestion mémoire*

Le contrôle de l'instant de construction des termes permet une meilleure gestion mémoire. En prenant, par exemple, la deuxième clause du prédicat *append* :

append(a.x, y, a.z) ← append(x, y, z).

Une résolution classique pour le mode (↓,↓,↑) construit *toujours* un terme de type x.y pour le troisième argument *avant* la résolution de la queue de la clause ; si l'un des sous-buts de celle-ci est en échec, la construction de ce terme est alors totalement inutile.

L'utilisation de l'expansion de l'unification et de l'ordonnancement produit en revanche :

append(↓u, ↓y, ↑v) ← cons(↑a, ↑x, ↓u), append(↓x, ↓y, ↑z), cons(↓a, ↓z, ↑v).

ce qui repousse cette construction en fin de clause. Celle-ci ne sera donc effectuée que si la résolution des sous-buts précédents réussit.

2.2.2.3. *Elimination du déréférençage*

L'évaluation de la forme expansée d'une clause effectue toutes les constructions de termes par l'intermédiaire des constructeurs produits par l'expansion primaire. L'association des modes stricts à ces prédicats constructeurs permet d'en adapter le fonctionnement pour chacun de ces modes.

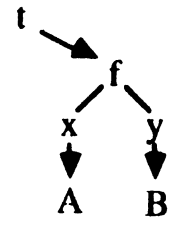
Soit le prédicat constructeur :

consf(x, y, f(x, y)) ←.

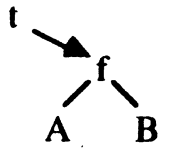
et le but :

$$\leftarrow \text{consf}(A, B, t).$$

La résolution de ce but produit comme solution t = f(A, B). Ce résultat est représenté dans une résolution classique par (cf. Chapitre 2) :



En sachant que ce but sera résolu en mode consf(↓,↓,↑), on peut se passer des variables intermédiaires x et y puisqu'on est assurés que les deux sous-termes sont clos. On peut par conséquent construire directement :



et éviter ainsi des déréférençages inutiles et coûteux.

Cette technique peut évidemment être étendue à tous les modes stricts des prédicats constructeurs. Les instructions exécutées pour la résolution du but consf(x, y, t) dans différents modes pourraient être :

mode	**instructions**
(↓,↓,↓)	**si** argument1(t) ≠ x **ou** argument2(t) ≠ y **ou** foncteur(t) ≠ f **ou** arité(t) ≠ 2 **alors** *faux*
(↓,↓,↑)	t := consf(x, y) *(Construction effective)*
(↓,↑,↓)	**si** argument1(t) ≠ x **ou** foncteur(t) ≠ f **ou** arité(t) ≠ 2 **alors** *faux* **sinon** y := argument2(t)
(↑,↓,↓)	**si** argument2(t) ≠ y **ou** foncteur(t) ≠ f **ou** arité(t) ≠ 2 **alors** *faux* **sinon** x := argument1(t)
(↑,↑,↓)	**si** foncteur(t) ≠ f **ou** arité(t) ≠ 2 **alors** *faux* **sinon** x := premarg(t), y := secarg(t)

Ceci dénote une approche plutôt *fonctionnelle* de la résolution, que nous examinons plus longuement dans le paragraphe suivant.

2.2.3.4. *Calculs inutiles*

L'utilisation des modes stricts fixe un sens absolu au flux de données qui transitent par l'intermédiaire d'un argument. Alliée à l'expansion de l'unification, il peut se produire une certaine perte d'informations dont les conséquences peuvent être importantes au niveau du nombre de calculs à effectuer *[Reddy 85]*.

Soit, par exemple, le programme :

$$P(0) \leftarrow Q(x).$$
$$P(x.y) \leftarrow R(x,y).$$
$$\leftarrow P(u.v).$$

Après expansion et ordonnancement selon les modes stricts, la question utilisateur devient :

$$\leftarrow P(\uparrow t), cons(\uparrow u, \uparrow v, \downarrow t).$$

où le premier littéral fournit des solutions qui sont filtrées par le second.

L'appel P(t) ne contient plus l'information "l'argument est une structure de la forme x.y" que l'on retrouvait dans la question originale. De ce fait, la résolution va s'effectuer sur les deux clauses qui définissent P, alors qu'une résolution classique aurait immédiatement éliminé la première. On va donc effectuer un appel inutile à Q(x), dont le coût va s'ajouter à celui de la résolution "utile". On peut constater que la sélection d'un ordonnancement dans le cas des modes directionnels en général aurait indiqué comme choix l'ordre :

$$\leftarrow cons(?u, ?v, ?t), P(\uparrow t).$$

ce qui constitue bien la meilleure solution si l'on prends comme critère de coût le nombre d'unifications tentées.

Cette situation se produit lorsque, dans la forme non expansée, un littéral contient :

- un argument structuré en mode *sortie*,
- la même variable dans deux arguments en mode *sortie*.

Dans un domaine similaire, considérons maintenant le programme suivant :

entier(0) ←.
entier(1.x) ← entier(x).
deux(x.x.0) ←.
← entier(n), deux(n).

La question peut être ordonnée de deux manières :

← entier(↑n), deux(↓n).
← deux(?n), entier(↑n).

Le premier ordonnancement, qui corresponds à un modage strict, entraîne un bouclage de la résolution, après toutefois que la solution ait été fournie. Le second fonctionne par contre parfaitement, bien qu'il utilise un mode complexe. On peut constater que, dans ce cas également, la sélection de l'ordonnancement dans le cas général des modes directionnels donne bien le deuxième ordonnancement comme meilleure solution.

Choisir systématiquement un ordonnancement en mode strict n'est donc pas toujours la garantie d'un gain de temps à la résolution. Toutefois, on peut supposer que les gains effectués par ailleurs compensent les pertes subies parfois dans ce domaine, et de ce fait choisir systématiquement les modes stricts. Cette solution a été choisie dans Starlog *[Oudot 86]*.

Une solution plus sage est de résoudre effectivement de tels cas en utilisant les modes complexes, et en revenant donc à une résolution plus classique.

2.2.2.5. *Transformation en programmes fonctionnels*

L'utilisation des modes stricts supprime, pour un mode d'appel donné, la propriété de bi-directionnalité associée à chaque argument. Par contre, la disposition d'une directionnalité absolue autorise, dans le cas des modes stricts fonctionnels, la transformation de programmes Prolog en programmes fonctionnels. On peut de ce fait utiliser la réduction au lieu de la résolution, ce qui résulte en un gain de performances intéressant.

U.S.Reddy *[Reddy 84]* propose une telle transformation qui s'effectue sur la base des modes stricts. Cette transformation considère qu'un prédicat doté d'un mode d'appel donné est une fonction admettant comme paramètres les arguments placés en mode *entrée* et qui retourne un *ensemble* de n-uplets formés des arguments placés en mode *sortie.*

Ainsi, pour une syntaxe fonctionnelle où { } désigne un ensemble et [] un n-uplet, le prédicat append dans le mode append(↓,↓,↑) deviendrait :

$$\text{append1}(x, y) = \text{append11}(x, y) \cup \text{append12}(x, y)$$
$$\text{append11}(\text{NIL}, y) = \{[y]\}$$
$$\text{append12}(a.x, y) = \{[a.z] \mid [z] \in \text{append1}(x, y)\}$$

Le même prédicat dans le mode append(↑,↑,↓) s'écrirai :

$$\text{append2}(z) = \text{append21}(z) \cup \text{append22}(z)$$
$$\text{append21}(z) = \{[\text{NIL}, z]\}$$
$$\text{append22}(a.z) = \{[a.x, y] \mid [x, y] \in \text{append2}(z)\}$$

Un interprète Prolog utilisant le retour arrière ne produit pas les solutions en un seul ensemble mais une par une, en fonction de la demande. Un effet similaire peut être obtenu dans la transformation fonctionnelle en utilisant une évaluation "paresseuse" ; toutefois, une implantation plus "brutale" est également envisageable, tel que cela a été fait en Starlog *[Oudot 86]*. La présence de prédicats produisant une infinité de solutions peut cependant conduire à la construction d'ensembles de cardinalité infinie, bien que la supression de la plupart des branches infinies rende de telles situations peu fréquentes.

2.2.2.6. *Modes stricts et parallélisme*

L'étude du parallélisme en Prolog *[Conery 81]* est fortement motivée par les constatations suivantes :

- Le coût des machines séquentielles croît exponentiellement en fonction de leur puissance alors qu'une puissance théorique équivalente peut être obtenue grâce à plusieurs processeurs de moindre puissance pour un coût global inférieur.

- Le langage Prolog semble bien s'adapter à une résolution parallèle du

fait de son non-déterminisme potentiel.

Il existe différents modèles de parallélisme pour Prolog *[Syre 85]*, *[Codognet 87]* essentiellement basés sur les deux types de parallélisme que l'on peut extraire du processus de résolution de Prolog :

Parallélisme OU

On parcourt en parallèle les différentes branches d'un noeud OU. Un prédicat est alors exécuté en lançant en parallèle la résolution de plusieurs des clauses qui le composent.

Parallélisme ET

On parcourt en parallèle les différentes branches d'un noeud ET. La résolution des sous-buts d'une clause est alors effectuée en activant plusieurs de ces sous-buts en parallèle.

Nous avons vu au chapitre 1 que la Résolution n'autorise pas le parallélisme ET, en raison de la dépendance entre les différents sous-buts d'une résolvante due aux variables qu'ils partagent. Seuls peuvent donc être résolus en parallèle les sous-buts n'ayant pas de variables en commun. Ainsi, pour la clause :

$$P(x, y) \leftarrow Q(x), R(y).$$

il semble au premier abord que les deux sous-buts Q(x) et R(y) sont indépendants, étant donné qu'ils n'ont pas de variables communes. En fait, si la question utilisateur est la suivante :

$$\leftarrow P(z, z).$$

ces deux sous-buts sont liés indirectement par la variable z, à laquelle les variables x et y sont instanciées.

Dans le cadre général de Prolog, il est donc très difficile de déterminer statiquement si deux sous-buts sont indépendants, en raison de la dépendance de cette propriété vis-à-vis des instanciations des arguments. Conery et Kibler *[Conery 85]* ont proposé une solution à ce problème, basée sur la reconnaissance des sous-buts *générateurs*, et sur un algorithme qui met en correspondance les différents flux de données qui transitent par les arguments des sous-buts. Cette reconnaissance s'effectue en partie sur la base des modes de Warren en considérant que, pour les variables d'un argument d'un sous-but donné :

- Le mode "+" signifie que le sous-but ne peut être un générateur de ces variables.

- Le mode "-" signifie que le sous-but est un générateur de ces variables.

Ces connaissances sont completées par une heuristique déterminant les sous-buts générateurs de chaque variable qui n'est pas dotée d'un tel mode. Cette solution présente toutefois l'inconvénient lié aux modes de Warren, c'est à dire de ne pas exploiter les différents modes possibles d'un prédicat, mais seulement le plus général.

Ce problème peut également être résolu en utilisant les modes stricts et l'expansion de l'unification. En effet, ces deux techniques permettent de disposer des propriétés suivantes :

- Les termes instanciés aux variables d'une clause sont toujours des termes clos. L'indépendance entre deux sous-buts peut donc être déterminée statiquement, en vérifiant tout simplement s'ils n'ont pas de variables communes au niveau du texte source.

- Les modes précisent la *direction* des données. Par conséquent, si un processus est associé à chaque sous-but, on peut déterminer avec précision quels sont les processus consommateurs et les processus producteurs, ce qui permet d'installer des canaux de communication dirigés en fonction de ces modes.

- Quand plusieurs ordonnancements sont envisageables pour une clause, il est possible de sélectionner celui qui conduit au plus grand parallélisme.

Un formalisme assez souvent employé pour représenter les réseaux de processus correspondant au parallélisme ET consiste à considérer une clause comme une boîte comportant comme entrées (resp. sorties) les arguments en mode *entrée* du prédicat (resp. *sortie*). Intéressons nous de plus près au réseau de processus contenu dans cette boîte.

Le non-déterminisme de Prolog exige que les connexions entre les différents sous-buts (processus de notre réseau) véhiculent plusieurs solutions et forment donc des flux. Il est donc indispensable d'introduire les opérateurs suivants pour les connecter :

s : qui synchronise les solutions provenant de deux flux distincts, en les supprimant si l'une des deux est fausse.

v : qui fusionne deux flux, en retournant toutes les combinaisons possibles des solutions de ces flux.

Certains modèles parallèles de Prolog se sont affranchis de cette contrainte en forçant chaque prédicat à être déterministe. C'est notamment le cas de Concurrent Prolog *[Shapiro 83]*, où une seule solution est prise en compte.

Soit, par exemple, le programme Prolog :

homme(Pierre) ←.
homme(Paul) ←.
.....
homme(Jean) ←.

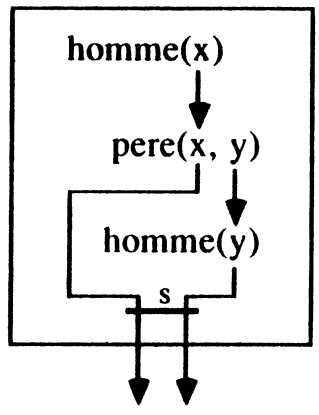
père(Pierre, Paul) ←.
père(Marie, Pierre) ←.
.....
père(Jean, Marc) ←.

fils(x, y) ← homme(x), homme(y), père(x, y).

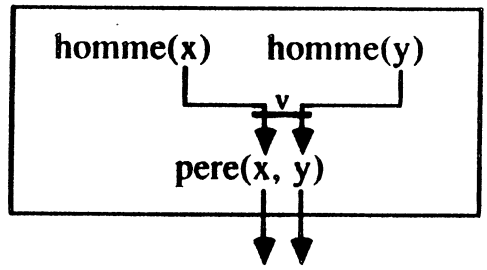
La clause de définition du prédicat *fils* en mode (↑,↑) est sujette, entre autres, aux ordonnancements suivants :

(1) fils(↑x, ↑y) ← homme(↑x), père(↓x, ↑y), homme(↓y).
(2) fils(↑x, ↑y) ← homme(↑x), homme(↑y), père(↓x, ↓y).

Le premier choix de modes produit le réseau de processus suivant :



alors que le deuxième produit :



On peut constater aisément que cette deuxième solution implique un nombre supérieur de processus réellement en parallèle.

Les modes stricts, alliés à une expansion de l'unification plus fine, permettent de mettre en évidence un plus grand parallélisme. Soit, par exemple, la clause :

$$R(x) \leftarrow P(x), Q(x).$$

où les prédicats P et Q sont quelconques et dotés de tous les modes possibles. Pour le mode R(↑), deux ordonnancements peuvent être utilisés :

$$(1)\ R(\uparrow x) \leftarrow P(\uparrow x), Q(\downarrow x).$$
$$(2)\ R(\uparrow x) \leftarrow Q(\uparrow x), P(\downarrow x).$$

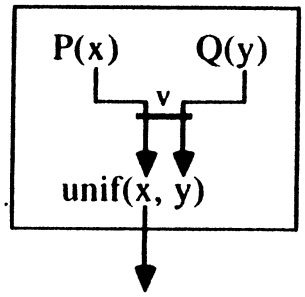
On constate facilement qu'aucun parallélisme n'est possible dans la mesure où les deux sous-buts partagent la même variable. Il serait pourtant intéressant de les rendre indépendant pour pouvoir les exécuter en parallèle. Pour ce faire, il suffit d'effectuer l'expansion secondaire qui nous permet d'obtenir pour l'exemple :

$$R(x) \leftarrow P(x), Q(y), \text{unification}(x, y).$$

et le choix de modes suivant :

$$R(\uparrow x) \leftarrow P(\uparrow x), Q(\uparrow y), \text{unification}(\downarrow x, \downarrow y).$$

auquel on peut associer le réseaux de processus :



Ce réseau présente l'avantage d'exprimer un parallélisme entre les sous-buts P et Q.

Bien qu'ils ne résolvent en rien les problèmes liés à l'implantation d'une résolution parallèle de Prolog, les modes stricts permettent cependant d'étendre le nombre de choix possibles et autorisent une plus grande liberté pour la production d'un réseau de processus. De plus, la manipulation exclusive de termes clos nous semble d'un grand intérêt pour minimiser le volume de communication entre les processus.

2.2.3. *Modes stricts fonctionnels et modes complexes*

Nous avons vu que les modes stricts offrent de multiples avantages dans le cadre de l'implantation de la Résolution. Notre expérience acquise avec Starlog nous a montré que leur utilisation *exclusive* (modes stricts fonctionnels) permet d'une part d'utiliser systématiquement et simultanément toutes les techniques d'optimisation présentées ici, mais de plus qu'elle est suffisante à la résolution de la plupart des problèmes, dans certains cas au prix d'une écriture différente, avec une approche plutôt fonctionnelle.

Toutefois, l'expression de certains problèmes est plus aisée en utilisant toute la généralité du langage *[Reddy 85]*. Comme exemple, Reddy donne le problème de la translation d'adresse que l'on trouve en compilation. Ce problème consiste à transformer une liste donnée d'éléments de deux types, *def(x)* (définition d'une variable), et *use(x)* (utilisation de cette variable). La transformation doit retourner une liste où chaque *def(x)* est remplacé par *asgn(x,n)*, où n désigne l'adresse d'implantation de la variable x allouée dans l'ordre des définitions de variables, et où les *use(x)* sont remplacés par le *use(n)* correspondant. Par exemple, la liste :

[def(A), use(A), use(B), def(C), def(B)]

doit être transformée en :

[asgn(A,1), use(1), use(3), asgn(C,2), asgn(B,3)]

Le programme réalisant cette transformation s'écrit en Prolog :

```
translate(in, out) ← map(in, out, table, 1).

map(NIL, NIL, x, y) ←.
map(def(x).in, asgn(x,n).out, table, n) ← member(asgn(x,n), table),
                                          map(in, out, table, n+1).
map(use(x).in, use(a).out, table, n) ← member(asgn(x,a), table),
                                       map(in, out, table, n).

member(a, a.x) ← !.
member(a, b.x) ← member(a, x).
```

On peut constater que ce programme effectue la translation d'adresses en un seul passage. En Starlog, ce problème doit être programmé classiquement, en effectuant deux passages, le premier pour construire la table d'assignation d'adresses, et le second pour calculer le résultat effectif.

Un autre exemple d'utilisation explicite des termes complexes est celui des *difference lists [Fribourg 87]*, qui sont une manière de représenter les listes qui autorise la concatenation de deux listes en une seule unification. Dans ce formalisme, une liste A.B.NIL est représenté sous forme de *d-list* par : d(A.B.x, x). La concatenation de deux listes peut alors s'effectuer en une seule inférence par application du prédicat *append* défini par :

```
append(d(u, v), d(v, w), d(u, w)) ←.
```

La concatenation de deux listes ne peut être programmée en Starlog qu'avec la définition classique de *append* :

```
append(NIL, x, x) ←.
append(a.x, y, a.z) ← append(x, y, z).
```

Il est donc important de conserver toute la généralité du langage Prolog. En utilisant les modes stricts à des fins d'optimisation, cette généralité peut être reconstituée grâce aux modes complexes dont le but devient :

- Permettre de déterminer quelles sont les parties d'un programme qui sont résolvables en modes stricts fonctionnels. Pour une résolvante donnée, si un sous-but se résoud en mode strict fonctionnel, tout le

sous-arbre attaché à ce sous-but se résoud alors en mode strict fonctionnel.

- Effectuer le lien entre ces différentes parties lorsque des termes complexes sont indispensables.

Dans le cadre d'une utilisation maximale des modes stricts fonctionnels, la sélection d'un ordonnancement doit alors s'effectuer de manière à faire apparaître le maximum de sous-buts en mode strict.

Un système Prolog basé sur une exploitation maximum des modes stricts fonctionnels doit nécessairement être hybride, c'est à dire doté de deux modes de résolution distincts. On peut constater que les parties qui doivent être résolues avec des techniques classiques peuvent être sujettes aux optimisations classiques qui s'y rapportent. Dans ce cadre, il est envisageable d'exploiter par exemple les modes de Warren sur les prédicats appelés en mode complexe. Dans ce cas, la production de ces modes, qui doit s'effectuer en connaissant l'ordre de résolution des sous-buts (notamment pour le mode "-"), peut être réalisée une fois les clauses ordonnées en fonction des modes directionnels.

3. Vérification et production des modes directionnels

Dans notre présentation, nous avons toujours supposé disposer des modes corrects minimaux des prédicats utilisés. Ce paragraphe est consacré à l'étude des techniques liées à l'acquisition de ces modes. Deux approches nous semblent possibles :

- Les modes directionnels des prédicats sont donnés explicitement par le programmeur. Cette solution, adoptée par la plupart des compilateurs *[Warren 77]*, est une source d'erreurs difficiles à détecter. C'est pourquoi il est intéressant de disposer d'un outil de vérification de ces déclarations.

- Les modes sont produits automatiquement à partir du programme. Cette méthode est à nos yeux la plus intéressante, bien qu'elle ait l'inconvénient d'être coûteuse à mettre en oeuvre.

Le critère de correction des modes directionnels est, dans les deux cas, le théorème démontré au début du chapitre. Ce théorème ne représentant qu'une condition suffisante, il est évident que certains modes corrects ne pourront être prouvés comme tels, ou qu'ils ne pourront être produits automatiquement. De toute façon, l'utilisation qui en est faite étant basée sur l'exploitation du bon modage d'une clause pour sélectionner un ordre d'exécution, il ne nous semble pas restrictif de considérer comme corrects les seuls modes vérifiant le théorème.

3.1. Vérification des modes directionnels

La vérification des modes directionnels déclarés par l'utilisateur consiste à contrôler leur validité par application du théorème de correction (cf. paragraphe 1). Il suffit donc de vérifier que, pour tous les modes déclarés, l'ensemble des prédicats du programme peut être bien modé en se restreignant aux seuls modes utilisés dans les déclarations. Si la vérification échoue pour un mode, le prédicat correspondant a ce mode n'a pu être bien modé pour l'une des trois raisons suivantes :

- Le mode est correct mais le bon modage n'a pu être obtenu à cause d'une insuffisance de déclaration de modes pour l'un des prédicats qu'il appelle.

- Le mode est bien correct mais ne vérifie pas la condition de correction

(qui n'est que *suffisante*).

- Il s'agit réellement d'un mode incorrect pour ce prédicat. Dans ce cas, tous les prédicats qui l'utilisent doivent à nouveau être controlés dans la mesure où leur bon modage a pu faire référence au mode en question.

La réalisation d'un tel outil de vérification de modes ne peut se concevoir que dans un environnement de programmation prêtant assistance à l'utilisateur dans la gestion de ses programmes. Deux possibilités se présentent : soit le programme complet est préalablement construit puis modé, soit on vérifie de façon incrémentale les modes proposés par le programmeur.

Dans le premier cas, l'échec du contrôle de correction d'un mode d'un prédicat entraîne la mise en doute de tous les modes des prédicats qu'il appelle. Dans certains cas, tous les modes du programme pourront être mis en doute, ce qui rends l'erreur difficilement détectable.

Dans le deuxième cas, on est obligé de considérer les modes des prédicats non encore écrits par le programmeur (et transitivement ceux qui en dépendent) comme des conjectures. Lorsque un prédicat manquant est effectivement fourni, ces conjectures doivent être vérifiées ou réfutées.

La mise en oeuvre d'un outil de vérification pose donc beaucoup de problèmes dès que le programmeur a un grand nombre de prédicats à manipuler. La solution de la production automatique semble donc nettement plus intéressante.

3.2. Production automatique des modes directionnels

La production automatique des modes directionnels est réalisée par un algorithme simple dans son principe, mais coûteux dans sa réalisation. Nous présentons dans ce qui suit l'algorithme général permettant la production des modes directionnels corrects d'un programme Prolog. Nous verrons ensuite les diverses optimisations réalisables sur cet algorithme.

La production des modes stricts fonctionnels peut être spécialisée et réalisée à moindre coût. Nous présentons un algorithme spécialisé dans ce but, ainsi qu'une méthode permettant de produire les modes complexes restants à partir des modes stricts fonctionnels.

3.2.1. *Principe de la production des modes*

L'algorithme de production automatique des modes directionnels se déduit aisément du corollaire du théorème de correction des modes. En effet, soient :

- X un ensemble de prédicats modés (où tous les littéraux des clauses de chaque prédicat sont représentés avec leurs modes)

- S une fonction tel que S(X) = {prédicats de X bien modés par des modes de X}

Le corollaire du théorème de correction des modes montre que pour tout point fixe de S (tel que X=S(X)), tous les modes de X sont des modes corrects. L'algorithme de production des modes corrects se réduit donc au calcul du plus grand point fixe de la fonction S :

- X_0 = ensemble des prédicats du programme avec toutes les combinaisons de modes possibles pour les littéraux de leurs clauses.

- $X_{n+1} = S(X_n)$

La suite définie par les X_n est décroissante (car on a toujours X_{n+1} inclus dans X_n) et bornée par ∅. On obtient donc au bout d'un temps fini un point fixe X_f (théorème du point fixe) correspondant au plus grand ensemble de modes corrects pour l'ensemble de prédicats considéré.

Ce processus est très coûteux car la cardinalité de X_0 est très grande. En effet, chaque argument pouvant prendre trois modes distincts (↑, ↓ et ?), chaque littéral dispose par conséquent de 3^n modes possibles.

Une représentation directe de X_0, où l'on donne aux littéraux toutes les combinaisons possibles pour les littéraux est irréaliste. Il est préférable de travailler à partir d'une représentation de l'ensemble des prédicats d'une part, et de l'ensemble des modes possibles pour ces prédicats d'autre part. La fonction S doit par conséquent être remplacée par une autre fonction dont le travail consiste, pour un prédicat et un mode donnés, à vérifier qu'il existe un bon modage de toutes les clauses de P pour ce mode. On a alors :

- K = (P, M) où P désigne un ensemble de prédicats et M un ensemble de modes portant sur ces prédicats.

- F une fonction telle que F(P, M) = ensemble des modes de M pour

lesquels les prédicats de P peuvent être bien modés en n'utilisant que des modes de M. F est appelée **fonction de filtrage.**

Remarque

La fonction F joue le même rôle que la fonction S mais se charge en outre de produire toutes les combinaisons possibles pour le choix des modes des littéraux des clauses. Un avantage est qu'un prédicat est considéré comme bien modé dès que la première combinaison correcte est trouvée.

L'algorithme devient :

- $K0 = (P, M0)$ où M0 désigne l'ensemble des modes possibles.

- $Kn+1 = (P, F(Kn))$

La cardinalité de l'ensemble initial, bien que grande, est ici beaucoup plus maîtrisable. En effet, si les prédicats de P sont notés P1,...,Pn, la cardinalité de M0 est alors :

$$card(M0) = \sum_{i=1}^{n} 3^{arité(Pi)}$$

Lorsque le point fixe est obtenu, l'ensemble de modes Md tel que $Md=F(P, Md)$ représente l'ensemble des modes directionnels corrects de P.

On peut noter que, bien que l'ensemble obtenu ne contienne que des modes tels que $MR \leq MA$ (dans la mesure où M0 ne contient que des modes de ce type), il ne représente pas l'ensemble des modes minimaux. Celui-ci doit être calculé à partir de Md par l'opération :

$$Mm = Minimiser(Md)$$

Etant donné que notre ensemble de propriétés $EP=\{Q, C\}$ est fermé, un mode minimal est unique. L'algorithme nous produira donc un seul mode par mode d'appel possible. Le nombre total de modes minimaux pour un prédicat d'arité n est donc exactement de 2^n.

Remarque

Les modes minimaux obtenus ne le sont que relativement au critère de correction défini par le théorème. Ce critère n'étant qu'une condition suffisante, il n'est donc pas certain qu'il s'agisse des modes réellement minimaux.

Enfin, il est important d'indiquer que les prédicats prédéfinis sont implicitement dotés de modes qui dépendent de leur sémantique. Ces modes sont par conséquent fixés par l'implantation et devront être directement utilisés par l'algorithme de production de modes directionnels. Pour tenir compte de l'ensemble Mi de ces modes prédéfinis, la fonction de filtrage F est transformée en une forme équivalente Fd telle que :

$$F(P, M) \equiv Fd(P, M, Mi).$$

3.2.2. *Algorithme de filtrage des modes directionnels*

La fonction de filtrage d'un prédicat vérifie que chaque clause du prédicat peut être bien modée en utilisant les modes définis dans l'ensemble M. En supposant disposer de l'algorithme filtrant une clause, l'algorithme général de production des modes est donc le suivant :

M = ensemble de tous les modes possibles
pointfixe := *faux*
Tant que ¬pointfixe **faire**
début
  pointfixe := *vrai*
  Mr := ∅
  Pour chaque prédicat P **faire**
    Pour chaque mode m de P dans M **faire**
      Si Pour toute clause C de P
          filtrerclause(C, m, M) = Succès
      alors Mr := Mr ∪ {m}
      sinon pointfixe := *faux*
  M := Mr
fin

L'algorithme de filtrage d'une clause se base sur l'utilisation d'un ensemble Ec contenant les variables connues commes instanciées à des termes clos a un instant donné. Par ailleurs, il utilise les fonctions suivantes :

compatible(mL, L, Ec), qui retourne vrai si toutes les variables du littéral L placées en mode *entrée* dans mL sont soit des constantes, soit présentes dans Ec.

vérifier(mL, L, Ec), qui retourne vrai si toutes les variables du littéral L placées en mode *sortie* dans mL sont soit constantes, soit présentes dans Ec.

ajouter(mL, L, Ec), qui retourne l'ensemble de variable Ec auquel sont ajoutées toutes les variables du littéral L en mode *sortie* dans mL.

retour(B), qui permet de sortir de la fonction de filtrage en lui donnant le résultat B.

Soit la clause P ← Q1, Q2,..., Qn et m le mode du prédicat P traité. L'algorithme de filtrage d'une clause est le suivant :

Pour chaque permutation [L1, L2, ..., Ln] de {Q1, Q2,..., Qn} **faire**
début
E0 = ensemble des variables de l'entête P en mode *entrée* dans m.
Pour chaque mode m1 de L1 tel que *compatible*(m1, L1, E0) **faire**
E1 = *ajouter*(m1, L1, E0)
Pour chaque mode m2 de L2 tel que *compatible*(m2, L2, E1) **faire**
E2 = *ajouter*(m2, L2, E1)
. . . . . .
Pour chaque mn de Ln tel que *compatible*(mn, Ln, En-1) **faire**
En = *ajouter*(mn, Ln, En-1)
Si *vérifier*(mP, P, En) **alors**
retour(Succès)
. . . . . .
fin
retour(Echec)

Il est intéressant d'évaluer le coût d'un tel algorithme. Si n est le nombre de littéraux de la clause, n! le nombre de permutations et mi le nombre de modes contenus dans M pour le littéral Li, le coût maximal de l'algorithme de filtrage d'une clause (en cas d'échec) est en :

$$O(n! * \prod_{i=1}^{n} mi)$$

ce qui est énorme, d'autant plus que le filtrage est effectué pour chaque mode possible d'une clause.

3.2.2.1. *Optimisation du filtrage*

De nombreuses optimisations peuvent être effectuées sur l'algorithme de filtrage lui-même, ainsi que sur la manière de l'utiliser. Parmi ces optimisations, nous allons traiter les suivantes :

- L'ensemble de modes initial peut être réduit en supprimant les modes que l'on sait déjà corrects (par exemple, mode d'appel égal au mode de retour).

- L'échec dans le filtrage d'une clause invalide les autres clauses du prédicat pour le même mode. Il est donc très intéressant de filtrer en premier les clauses ayant le moins de littéraux puisqu'elles sont les moins longues à traiter..

- Les prédicats peuvent être traités dans un ordre tel que le nombre d'applications de la fonction de filtrage soit réduit. Une approche consiste à effectuer le filtrage par composantes connexes.

- La reconnaissance du point fixe nécessite une exécution inutile de l'algorithme. Cette exécution peut être évitée dans certains cas.

Réduction de l'ensemble de modes initial

Nous avons vu dans les paragraphes précédents que tous les modes dont le mode de retour est égal au mode d'appel sont toujours corrects ; il est donc inutile de les filtrer. Ces modes correspondent, en ce qui concerne les modes directionnels, à tous ceux qui n'ont pas d'argument en mode *sortie*, seul mode dont le retour soit différent de l'appel.

Ainsi, sur les 3^n modes possibles d'un prédicat à n arguments, 2^n peuvent être déclarés corrects à l'avance. L'ensemble de modes initial M0 peut donc être décomposé en deux sous-ensembles :

M^c qui représente les modes déclarés corrects dès le départ

M^t qui représente les modes à filtrer

Le résultat du filtrage est donc $Md = M^c \cup Mi \cup Fd(P, M^t, M^c \cup Mi)$.

Il faut toutefois indiquer que les modes de M^c peuvent parfaitement être éliminés par la minimisation s'il s'avère qu'il en existe un inférieur.

Filtrage par ordre de taille

Le filtrage des clauses par ordre croissant de taille offre des avantages importants sur deux plans :

- Le coût du filtrage d'une clause est notamment proportionnel à la factorielle du nombre de littéraux de cette clause. Etant donné que l'échec du filtrage d'une clause invalide le filtrage des clauses suivantes, il est intéressant de filtrer les plus courtes en premier.

- Ce sont généralement les faits qui filtrent le plus de modes.

Soit, par exemple, le prédicat :

append(NIL, x, x) ←.
append(a.x, y, a.z) ← append(x, y, z).

Il existe au départ 27 possibilités de modes. Or, la première clause filtre à elle seule 15 modes parmi les 19 non minimaux.

Soit P un ensemble de prédicats. On peut découper cet ensemble en sous-ensembles $P_0,...,P_n$ tels que P_i contient toutes les clauses de P comportant i littéraux (n représente le nombre de littéraux de la plus grande clause). Le filtrage peut alors s'effectuer à moindre coût par :

$$F(P, M) = F(P_n, F(P_{n-1}, ..., F(P_1, F(P_0, M))...))$$

Filtrage par composantes connexes

Les optimisations proposées jusqu'ici sont locales à un passage de l'algorithme de filtrage. Or, la production des modes corrects nécessite l'application répétitive de cet algorithme jusqu'à obtenir un point fixe. Il est donc également très intéressant de minimiser le nombre d'appels à cette fonction, ce qui peut être obtenu en filtrant les prédicats dans un ordre précis.

Prenons comme exemple un programme composé des prédicats *append* et *reverse* déjà donnés précédemment. Les mode corrects de *reverse* dépendent des modes corrects de *append*. Supposons que le filtrage s'effectue en commençant par le prédicat *reverse*. Au premier tour du filtrage, seuls les modes incorrects de *append* sont éliminés ; en effet, aucun mode de *reverse* n'est supprimé car, à cet instant, tous les modes de *append* sont encore considérés comme corrects. Il faut donc une deuxième application du filtrage pour éliminer les modes incorrects de *reverse*. Ce problème ne se produit évidemment pas si le prédicat *append* est traité en premier.

Soit $P \leftarrow Q_1,...,Q_n$ une clause du prédicat P. Les modes corrects de P

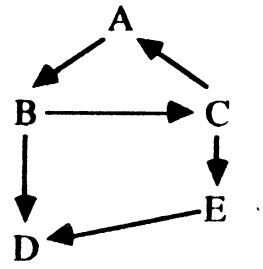
dépendent de la connaissance des modes corrects de Q1,...,Qn. Il est donc inutile de tenter la production des modes de P avant celle des modes des Qi.

A partir d'un programme Prolog, on peut obtenir son *graphe des appels*, défini par :

- Les noeuds du graphe sont les noms des prédicats du programme.

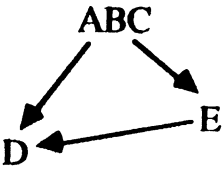
- Un noeud A est relié à un noeud B par un arc si une clause du prédicat A contient un littéral faisant appel au prédicat B.

Soit le graphe d'appels suivant :



Ce graphe nous indique que la production des modes pour un prédicat donné doit être effectuée après avoir traité tous les prédicats qu'il joint par un arc. Il est évident qu'un problème se pose en cas de cycle, ce qui indique que tous les prédicats du cycle devront être traités en même temps.

Si l'on remplace chaque composante fortement connexe par un noeud unique, le graphe des appels sera transformé en un graphe sans cycles :



Les cycles ayant disparu, on peut appliquer une stratégie de production consistant à traiter les modes à partir des puits du graphe, en remontant les arcs. Dans ce cas, le programme n'est plus traité dans son ensemble mais composante connexe par composante connexe. Ainsi, par exemple, les modes corrects du prédicat D sont produits intégralement en premier, suivis par ceux du prédicat E, et enfin par ceux des prédicats A, B et C simultanément.

Algorithme de détermination des composantes connexes

Les composantes fortement connexes d'un graphe dirigé peuvent être déterminées en appliquant un algorithme classique dans le domaine *[Tarjan 72]*. Nous avons toutefois développé un algorithme spécifique, non optimal, mais suffisant pour notre problème. Cet algorithme est le suivant :

- Pour chaque prédicat du programme, construire un ensemble contenant le nom du prédicat. Par parcours récursif, ajouter à cet ensemble tous les prédicats qu'il appelle, directement ou indirectement. Pour notre exemple, nous obtenons :

  {A, B, C, D, E}, {B, A, C, D, E}, {C, A, B, D, E}, {D}, {E, D}

- Trier la liste d'ensembles obtenue par ordre croissant de cardinalité. Pour notre exemple, nous obtenons :

  {D}, {E, D}, {A, B, C, D, E}, {B, A, C, D, E}, {C, A, B, D, E}

- Pour chaque ensemble x de la liste et à partir du premier, retirer des ensembles qui suivent tous les éléments contenus dans x. Notre ensemble devient :

  {D}, {E}, {A, B, C}, { }, { }

- La liste de composantes connexes est obtenue en retirant les ensembles vides :

  {D}, {E}, {A, B, C}

De plus, cette liste est triée de manière telle qu'une composante est toujours placée avant celles contenant des prédicats qui font référence à l'un de ses éléments.

Si $\mathbb{M}(Q, Mc)$ désigne la fonction de production des modes corrects des prédicats de Q, en connaissant les modes déjà corrects Mc, la production des modes corrects d'un programme P peut être effectuée à partir de la liste de ses composantes connexes $C_1,...,C_k$, en appliquant :

$$modes(P) = \mathbb{M}(C_k, \mathbb{M}(..., \mathbb{M}(C_0, Mi)...))$$

Elimination de la détection du point fixe

La condition d'arrêt de l'algorithme repose sur la détection du point fixe, lorsque $K_n=F(K_n)$. Ceci induit une exécution inutile de l'algorithme, qui peut

être évitée si l'on peut savoir à l'avance que le point fixe est atteint.

Cette connaissance est difficile à obtenir lorsqu'on traite le programme complet, ou une composante connexe complète, puisqu'elle dépends notamment de l'ordre de traitement des clauses et de la longueur des cycles dans les composantes connexes.

Toutefois, le filtrage d'une clause dont tous les littéraux ne font référence qu'à des prédicats dont les modes corrects sont déjà connus (donc soit prédéfinis, soit traités antérieurement dans une autre composante connexe) peut être effectuée en une seule fois. En effet, tout passage ultérieur sur cette clause ne filtre aucun mode supplémentaire puisque les informations nécessaires au filtrage ne sont pas modifiées par le premier passage.

Donc, au moment du traitement d'une composante connexe, le filtrage doit s'effectuer une première fois sur toutes les clauses de la composante. Ensuite, la recherche du point fixe peut être effectuée en ne filtrant que les clauses où apparaissent des littéraux référençant des prédicats contenus dans cette composante connexe.

Les opportunités d'application de cette optimisation sont nombreuses, puisqu'elle permet notamment de ne filtrer qu'une seule fois toutes les clauses fait du programme.

3.2.3. *Production des modes stricts fonctionnels*

La génération automatique des modes stricts fonctionnels représente un cas particulier qui permet de nombreuses optimisations. En effet, deux points de l'algorithme peuvent subir d'importantes modifications :

- L'ensemble de modes initial M0 ne contient que des modes stricts. Sa cardinalité est donc beaucoup plus faible :

$$\text{card}(M0) = \sum_{i=1}^{n} 2^{\text{arité}(Pi)}$$

à laquelle peuvent être retirés les n modes stricts toujours corrects (de la forme $(\downarrow)^*$).

- Les modes stricts ont, par rapport aux modes directionnels, l'avantage d'avoir toujours le même mode de retour (tous les arguments clos). De ce fait, une fois qu'un littéral est sélectionné par l'algorithme de filtrage, il est inutile de revenir sur le choix de son mode puisque leurs modes de

retour sont tous identiques.

3.2.3.1. *Algorithme de filtrage des modes stricts fonctionnels*

Soit la clause P←Q1,...,Qn et MP le mode du prédicat P traité. L'algorithme de filtrage des modes stricts fonctionnels est le suivant :

```
L = liste des littéraux Q1,...,Qn
E = ensemble des variables de l'entête P en mode entrée dans MP.
Tant que non vide(L) faire
début
   trouvé := faux
   Pour chaque littéral Q de L et tant que non trouvé faire
      Pour chaque mode MQ de Q et tant que non trouvé faire
         Si compatible(MQ, Q, E) alors
            trouvé := vrai
            E := ajouter(MQ, Q, E)
   Si non trouvé alors
      retour(Echec)
   L := retirer(Q, L)
fin
Si vérifier(MP, P, E) alors
   retour(Succès)
retour(Echec)
```

où les fonctions externes référencées sont :

vide(L), retourne vrai si la liste de littéraux L est vide.

retirer(Q, L), retourne la liste de littéraux L à laquelle a été retiré le littéral Q.

Le coût de cet algorithme est très inférieur à celui utilisé pour les modes directionnels. En effet, si mi représente le nombre de modes contenus dans M pour le littéral Qi, le coût maximal de l'algorithme est en :

$$O(n * \sum_{i=1}^{n} mi)$$

Il est à remarquer que l'ensemble des modes stricts produit n'a pas besoin d'être minimisé dans la mesure où il n'existe qu'un mode de retour pour un mode d'appel donné.

3.2.4. *Production des modes complexes*

Les modes complexes n'étant qu'une classe particulière des modes directionnels, il est clair que leur production peut s'effectuer en même temps que les modes stricts fonctionnels, en appliquant l'algorithme général que nous avons présenté. Toutefois, nous avons constaté que la production des modes stricts fonctionnels peut être effectuée à un coût très inférieur ; il nous semble donc intéressant de s'appuyer sur cette technique pour traiter les modes complexes.

Nous proposons ci-après une méthode de production des modes complexes basée sur la connaissance préalable des modes stricts fonctionnels. Une première stratégie de production des modes directionnels consiste à :

- Générer l'ensemble des modes stricts possibles. Cet ensemble se décompose en Ms^t et Ms^c (modes à filtrer et modes déjà connus).

- Filtrer cet ensemble avec la procédure de filtrage spécialisée Fs. On obtient alors un ensemble de modes stricts fonctionnels corrects :

$$Ms = Ms^c \cup Fs(P, Ms^t, Ms^c \cup Mi)$$

- Générer l'ensemble des modes complexes possibles (non-stricts et stricts non-fonctionnels, c'est à dire non produits par l'algorithme précédent). Comme pour les modes stricts fonctionnels, il est possible d'en retirer les modes toujours corrects (mode d'appel égal au mode de retour). On obtient alors les ensembles Mc^t et Mc^c.

- Filtrer Mc^t avec la procédure de filtrage générale. On obtient :

$$Mc = Mc^c \cup Fd(P, Mc^t, Mc^c \cup Ms \cup Mi)$$

- L'ensemble final des modes directionnels minimaux est alors égal à :

$$Md = Ms \cup Minimiser(Mc^c \cup Mc^t)$$

Par ailleurs, le lemme de transitivité (cf. paragraphe 1) nous indique que le mode de retour d'un mode minimal ne peut être égal au mode d'appel d'un autre mode minimal (sauf si ce dernier a un mode de retour identique à son mode d'appel). Il est donc inutile de tenter de filtrer de tels modes car ils seront nécessairement éliminés soit par le filtrage, soit par la construction de l'ensemble des modes minimaux. On peut ainsi réduire notablement la cardinalité de l'ensemble initial de modes à traiter.

La stratégie de production des modes directionnels devient :

- Produire l'ensemble des modes stricts fonctionnels Ms.

- Générer l'ensemble des modes complexes possibles. On obtient un ensemble Mc^g.

- Eliminer de Mc^g tous les éléments dont le mode de retour est égal au mode d'appel d'un mode strict. On obtient un ensemble $Mc^{g'}$.

- Décomposer $Mc^{g'}$ en Mc^c et Mc^t.

- Filtrer Mc^t avec la fonction de filtrage Fd. On obtient un ensemble Mc'.

- Minimiser l'union $Mc' \cup Mc^c$. On obtient ainsi l'ensemble de modes complexes Mc.

Soit, par exemple, l'ensemble de prédicats :

append(NIL, x, x) ←.
append(a.x, y, a.z) ← append(x, y, z).
reverse(NIL, NIL) ←.
reverse(a.x, z) ← reverse(x, y), append(y, a.NIL, z).
gappend(x, y, z, t) ← append(x, y, a), append(a, z, t).
egala(A, x) ←.

La cardinalité des ensembles apparaissant dans les différentes étapes de la stratégie exposée sont les suivantes :

	append	**reverse**	**gappend**	**egala**	
Ms^c	1	1	1	1	
Ms^t	7	3	15	3	
Ms	**5** (3)	**3** (1)	**9** (7)	**2** (2)	Filtrage Fs
Mc^g	22	6	72	7	
$Mc^{g'}$	8	2	26	5	
Mc^c	3	1	7	2	
Mc^t	5	1	19	3	
Mc'	0 (5)	0 (1)	0 (19)	1 (2)	Filtrage Fg
Mc	**3** (0)	**1** (0)	**7** (0)	**2** (1)	Minimisation
Md	8	4	16	4	

où nous avons placé entre parenthèses le nombre de modes éliminés par l'opération indiquée à droite.

Détaillons le contenu de ces ensembles pour les prédicats *reverse* et *egala* :

	reverse	**egala**
Ms^c	(↓,↓)	(↓,↓)
Ms^t	(↑,↓) (↓,↑) (↑,↑)	(↑,↓) (↓,↑) (↑,↑)
Ms	(↑,↓) (↓,↑) (↓,↓)	(↑,↓) (↓,↓)
Mc^g	(↑,↑) (↑,?) (↓,?) (?,↓) (?,↑) (?,?)	(↓,↑) (↑,↑) (↑,?) (↓,?) (?,↓) (?,↑) (?,?)
$Mc^{g'}$	(↑,↑) (?,?)	(↓,↑) (↑,↑) (↑,?) (?,?)
Mc^c	(?,?)	(?,?) (↓,?)
Mc^t	(↑,↑)	(↓,↑) (↑,↑) (↑,?)
Mc'	Ø	(↑,?)
Mc	(?,?)	(↑,?) (↓,?) (car (↑,?) ≤ (?,?))
Md	(↑,↓) (↓,↑) (↓,↓) (?,?)	(↑,↓) (↓,↓) (↑,?) (↓,?)

3.2.4.1. *Production économique des modes complexes*

Malgré toutes ces optimisations, la production des modes complexes reste très coûteuse. Il est cependant possible de produire un ensemble de modes complexes corrects à coût moindre en ne considérant que les modes tels que :

- le mode d'appel est égal au mode de retour,
- le mode d'appel n'est pas le mode d'appel d'un mode strict fonctionnel.

Bien sûr, ces modes, appelés modes *économiques*, ne sont pas minimaux, mais les deux points suivants peuvent justifier un tel choix :

- L'utilité pratique des modes complexes est moins importante par rapport à celle des modes stricts fonctionnels. Ils peuvent être considérés comme effectuant simplement la transition permettant d'inclure les modes stricts fonctionnels dans un cadre d'utilisation général.

- Le coût de production des modes économiques est pratiquement nul.

En outre, pour la plupart des prédicats, l'ensemble des modes économiques est effectivement minimal. Ainsi, pour le prédicat *append*,

l'ensemble des modes complexes économiques est égal à {(?,?,?), (↓,?,?), (?,↓,?)}, ce qui correspond exactement aux modes complexes minimaux de *append*. Par contre, pour le prédicat *égala*, l'ensemble des modes économiques est égal à {(↓,?), (?,?)} alors que l'ensemble des modes complexes minimaux est en réalité {(↓,?), (↑,?)}.

3.3. Traitement de la question utilisateur

Les modes directionnels ont pour but de permettre une compilation fine, adaptée à chaque situation particulière. En réalité, l'utilisation d'un programme fait que toutes ces situations (tous ces modes) ne sont pas employées. Ceci dépend des questions posées par l'utilisateur.

La première solution consiste à contraindre l'utilisateur à ne poser qu'une seule question, laquelle sera compilée en même temps que le programme. Dans ce cas, la question est un prédicat à part entière, dont le nom constitue pour le système celui du prédicat de "démarrage". Cette solution présente les caractéristiques suivantes :

- L'ordonnancement du prédicat de démarrage peut être effectué comme pour tous les autres prédicats. Le traitement est donc complètement standardisé.

- Le mode dans lequel est implicitement placé ce prédicat est celui dont le mode d'appel est (Q,...,Q) (aucun argument connu à l'appel). Une solution, utilisée par Starlog, consiste à ne pas mettre d'arguments à ce prédicat ; dans ce cas, l'affichage des solutions est à la charge du programmeur et se fait en utilisant les prédicats prédéfinis d'entrées/sorties.

- Un parcours récursif du programme à partir de ce prédicat permet de déterminer les modes réellement employés, et de ne produire du code que pour ceux-ci, économisant ainsi un espace mémoire important.

Une deuxième solution consiste à laisser l'utilisateur à un niveau d'interprétation classique, lui permettant l'écriture de toutes les questions qu'il désire. Cette solution se différencie principalement de la précédente par les points suivants :

- Chaque question doit nécessairement être modée et ordonnée avant sont exécution.

- Tous les ordonnancements pour tous les modes pour tous les prédicats doivent être conservés, étant donné que toutes les formes de questions sont possibles. On peut toutefois imaginer une compilation de ces ordonnancements "à la demande", en fonction des besoins du programmeur.

Une variante de cette seconde solution consiste à déclarer à la compilation les prédicats appelables par l'utilisateur. Dans ce cas, on peut compiler ces prédicats pour tous leurs modes, et ne compiler les autres que pour les modes référencés par les prédicats accessibles à l'utilisateur.

CHAPITRE 4

Application des modes stricts : Starlog

Dans les chapitres précédents, nous avons présenté le concept de modes directionnels et leur intérêt pour l'amélioration de l'efficacité de la résolution. Ces gains d'efficacité ont été confirmés par la mise en oeuvre d'un prototype basé sur ces développements : Starlog *[Oudot 86]*.

Starlog est un langage de type Prolog, restreint à l'utilisation exclusive des modes stricts fonctionnels. Ces modes, de même que la propriété de déterminisme, doivent être explicitement fournis par le programmeur pour chaque prédicat. Le système contrôle la cohérence de ces déclarations avant de les utiliser en conjonction avec l'expansion de l'unification, pour le réordonnancement des clauses du programme, et ensuite pour sa compilation.

Le système est constitué d'un interprète et d'un compilateur. Il a été réalisé en langage C sur Vax 780 sous le système UNIX 4.2, et ensuite porté sur d'autres machines UNIX (Sun, ...).

Nous avons d'autre part développé différents outils d'inférence de propriétés pouvant être exploitées dans un cadre plus général :

- Production de tous les modes directionnels minimaux d'un programme Prolog.

- Détection, à partir de ces modes, de ceux pour lesquels les prédicats sont déterministes.

Ce chapitre décrit les mécanismes utilisés dans Starlog pour mettre en oeuvre la résolution en modes stricts. Une étude comparative de performances permet d'apprécier les gains réalisables grâce à cette technique.

1. Architecture du système

La manipulation exclusive de termes clos tend à favoriser l'utilisation d'une représentation de données similaire à celle de Lisp. De ce fait, l'ensemble du système s'articule autour d'un noyau Lisp chargé de fournir les outils nécessaires à la manipulation des objets prédéfinis : chaînes de caractères, nombres et listes.

Les listes et les nombres sont alloués dans un espace fini, géré par une technique de liste libre réactualisée à chaque récupération mémoire (*garbage collector*). Les chaînes de caractères sont allouées dans un tas par l'intermédiaire d'une table de dispersion (*hash-code*). L'algorithme de récupération procède par marquage et permet la récupération de **tous** les types d'objets gérés par le noyau.

Outre les dispositifs d'allocation et de récupération de l'espace mémoire, le noyau Lisp fournit un ensemble de fonctions permettant de construire et d'accéder à ces objets. On y retrouve la plupart des fonctions de base d'un interprète Lisp : cons, car, cdr, etc. Il fournit également certaines fonctions plus complexes, utilisées par les prédicats prédéfinis, comme la concaténation et l'inversion de listes, la décomposition d'une chaîne en liste de caractères, etc.

Les termes manipulés dans Starlog n'étant que des termes clos, les listes, les chaines et les nombres trouvent leur représentation directement avec les objets du noyau. Les termes fonctionnels sont représentés d'une manière similaire à celle employée en Lisp pour les appels de fonctions, c'est à dire une liste dont le premier élément est le symbole fonctionnel lui même, et le reste, ses arguments. Toutefois, pour distinguer une liste d'un terme fonctionnel, le symbole fonctionnel est d'un type particulier, ce qui permet d'effectuer la distinction, notamment dans l'impression. Cette solution offre l'avantage de ne traiter qu'une seule catégorie d'objets structurés, les listes, et par la même de ne manipuler qu'un seul prédicat constructeur dans l'expansion, le *cons*.

Autour de ce noyau, le système Starlog est essentiellement composé d'un interprète et d'un compilateur produisant du langage C sur la base des clauses modées, expansées et ordonnées. Le choix de ce langage comme cible du générateur de code a été motivé par plusieurs raisons :

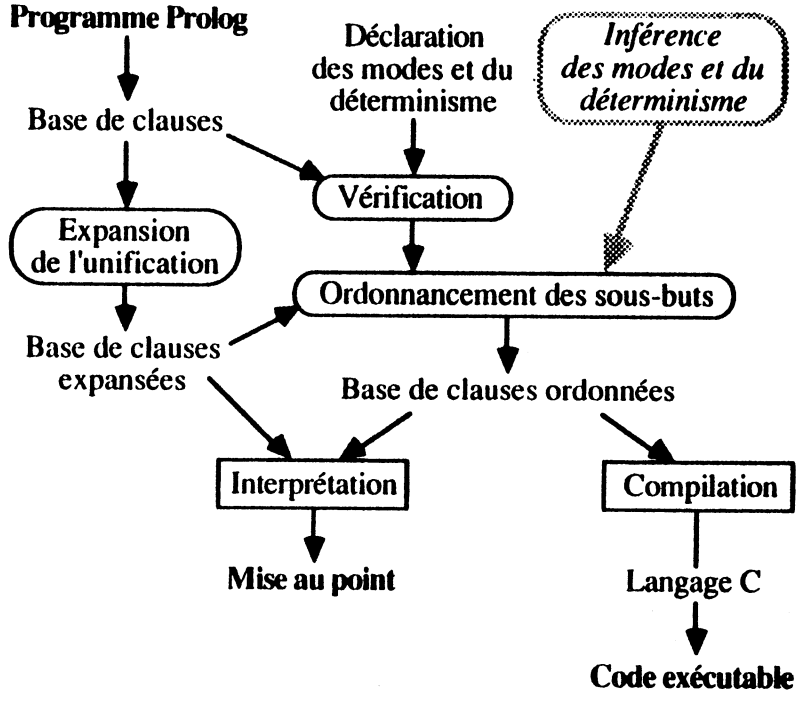
- Il simplifie notablement la génération de code dans son ensemble et permet notamment une écriture rapide du compilateur.

- Il assure une grande portabilité du fait de sa large diffusion et évite ainsi

l'écriture des générateurs de code adaptés à chaque cible.

- Le code machine généré par le compilateur C est assez efficace et reste "compétitif" vis à vis d'une production directe de code machine.

La structure générale du système Starlog est la suivante :



1.1. Stratégie de résolution

Dans Starlog, nous avons choisi de mettre totalement en oeuvre la transformation des programmes Prolog en programmes fonctionnels, à la manière de Reddy *[Reddy 85]*. Ainsi, à un prédicat donné correspond une fonction dont les paramètres sont les arguments en mode *entrée* du prédicat, et le résultat une liste des n-uplets solutions pour les arguments en mode *sortie*.

L'un de nos premiers objectifs étant l'efficacité, nous avons choisi d'utiliser une évaluation de ces fonctions très simple et brutale. En effet, les listes de solutions sont retournées *dans leur intégralité* par ces fonctions ; l'effet de "retour arrière", qui peut être obtenu par l'utilisation d'une évaluation paresseuse, est donc totalement absent.

Une conséquence importante de ce choix est que la résolution d'un prédicat comportant un nombre infini de solutions provoque la construction

d'une liste infinie à l'évaluation de la fonction associée ; la résolution s'arrête alors pour débordement d'espace mémoire. L'apparition de ce genre de problèmes est toutefois limitée par la stratégie d'ordonnancement des sous-buts qui tend à éviter les arbres de recherche infinis.

1.2. Compilation

Notre compilateur travail sur des prédicats dont toutes les clauses ont été expansées et ordonnées pour un mode correct (chaque argument est donc une variable ou une constante).

Soit un prédicat P composé des clauses C1,...,Cn, et m un mode strict de P. Soit *argin* et *argout* les ensembles des variables respectivement en mode *entrée* et en mode *sortie* de P. Le code généré par la compilation du prédicat P pour le mode m a la structure suivante :

$$Pm(argin) = \bigcup_{i=1}^{n} Ci(argin)$$

La compilation d'une clause C ← Q1,...Qn produit :

```
Ci(argin) : Liste_solutions ;
            Resul : Liste_solutions ;
            début
               Resul = nil ;
               Pour chaque solution de Q1
                  Pour chaque solution de Q2
                        . . .
                     Pour chaque solution de Qn
                        Resul = ajouter(valeur(argout), Resul) ;
                        . . .
               retour(Resul)
            fin
```

Si l'on nomme *environnement* l'ensemble des valeurs des variables de la clause, la résolution de chaque sous-but apporte de nouvelles informations à cet environnement par affectation des variables. Lorsque tous les sous-buts sont résolus, une solution finale est extraite en prenant dans l'environnement obtenu la valeur de toutes les variables en mode *sortie* dans l'entête de clause.

En réalité, ce schéma de compilation peut être simplifié en permettant l'intégration du code des clauses dans la fonction qui représente le prédicat. Analysons, par exemple, la compilation du prédicat *append* dont l'expansion et l'ordonnancement pour le mode (↓,↓,↑) produisent (cf. chapitre 3) :

append(↓u, ↓y, ↑y) ← égalité(↓u, ↓NIL).
append(↓u, ↓y, ↑v) ← cons(↑a, ↑x, ↓u), append(↓x, ↓y, ↑z), cons(↓a, ↓z, ↑v).

En prenant la notation [x1,...,xn] pour représenter un n-uplet, la compilation de ce prédicat donne :

```
append_ees (u, y) : Liste_solutions ;
                termes u, y ;

              a, x, z, v : termes ;
              Resul : Liste_solutions ;

           début
              Resul = NIL ;

              Pour chaque [] de égalité(u, NIL)          --- Clause 1 ---
                 Resul = ajouter([y], Resul)

              Pour chaque [a, x] de cons_sse(u)          --- Clause 2 ---
                 Pour chaque [z] de append_ees(x, y)
                    Pour chaque [v] de cons_ees(a, z)
                       Resul = ajouter([v], Resul)

              retour(Resul)
           fin
```

Les prédicats rajoutés par l'expansion peuvent être facilement prédéfinis et bénéficier ainsi d'un traitement particulier. Notre programme peut alors être simplifié en :

```
append_ees (u, y) : Liste_solutions ;
                termes u, y ;

              a, x, z, v : termes ;
              Resul : Liste_solutions ;

           début
              Resul = NIL ;
```

```
Si égalité(u, NIL)
   Resul = ajouter([y], Resul)

Si est_un_cons(u)
   a = car(u) ; x = cdr(u)
   Pour chaque [z] de append_ees(x, y)
      v = cons(a, z)
      Resul = ajouter([v], Resul)

retour(Resul)
fin
```

Les méta-prédicats ou prédicats d'ordre supérieur sont, dans leur ensemble, difficilement compilables. Par exemple, le prédicat *assert* (insertion dynamique d'une clause dans la base) nécessite soit une génération directe de code (solution choisie par BIM-Prolog *[Janssens 85]*), soit un noyau d'interprète permettant de résoudre les clauses insérées. Ce problème n'a pas été abordé dans Starlog qui n'effectue que la compilation *statique* des clauses de la base.

Le compilateur génère un programme C complètement autonome. Ceci impose la connaissance du point de lancement du programme. En Starlog, ce point de lancement est arbitrairement choisi comme étant le prédicat de nom *main*, devant être fourni par l'utilisateur.

1.2.1. *Code généré*

Toutes les variables locales du prédicat compilé sont représentées par autant de variables locales C de type *liste* ; ce choix impose, et c'est le cas, que le ramasse-miettes soit capable de parcourir la pile dans laquelle sont placées ces variables C.

Pour des raisons pragmatiques, la fonction C générée pour un prédicat ne retourne pas la liste de solutions mais leur nombre ; les solutions elles-mêmes sont rangées dans une pile (PTAND) servant à leur stockage. Une autre pile (PTOR) est utilisée pour le stockage provisoire des solutions du prédicat tant que celui-ci n'est pas complètement résolu. Ces solutions sont replacées ensuite dans PTAND.

Le code généré pour le prédicat *append* en mode (↓,↓,↑) est :

```
int append_ees (U, Y)
typeliste U, Y ;
{
   int NBSOL = 0 ;

   /*--- Clause 1 ---*/
   if (U == NIL)
   {  NBSOL++ ;
      *(PTOR++) = Y ;          /* empilement de la solution */
   }

   /*--- Clause 2 ---*/
   {  typeliste A, X, Z, V ;
      if (celcons(U))
      {  A = car(U) ;
         X = cdr(U) ;
         {  int res ;
            for (res = append_ees (X, Y) ; res ; res--)
            {  Z = *(PTAND++) ; /* récupération d'une solution */
               V = cons (A, Z) ;
               NBSOL++ ;
               *(PTOR++) = Z ;   /* empilement de la solution */
   }  }  }  }

   /*--- Echange et retour ---*/
   {  int i ;
      for (i = NBSOL ; i ; i--)
         *(--PTAND) = *(--PTOR) ;
   }
   return (NBSOL) ;
}
```

La compilation du même prédicat pour le mode (↑,↑,↓) génère :

```
int append_sse (V)
typeliste V ;
{
   int NBSOL = 0 ;

   /*--- Clause 1 ---*/
   NBSOL++ ;
   *(PTOR++) = NIL ;  /* empilement de la solution */
   *(PTOR++) = V ;

   /*--- Clause 2 ---*/
   {  typeliste A, X, Y, Z, U ;
      if (celcons(V))
      {  A = car(V) ;
         Z = cdr(V) ;
         {  int res ;
            for (res = append_sse (Z) ; res ; res--)
            {  X = *(PTAND++) ;
               Y = *(PTAND++) ;  /* récupération d'un couple */
```

```
                U = cons (A, X) ;
                NBSOL++ ;
                *(PTOR++) = U ;  /* empilement de la solution */
                *(PTOR++) = Y ;
    } } } }

    /*--- Echange et retour ---*/
    { int i ;
      for (i = NBSOL ; i ; i--)
          *(--PTAND) = *(--PTOR) ;
    }
    return (NBSOL) ;
}
```

1.2.2. *Optimisations du code généré*

Une analyse du code généré pour le prédicat *append* en mode (↓,↓,↑) montre clairement que certaines manipulations de pile sont inutiles ; en effet, ce prédicat est déterministe dans ce mode et ne renvoie qu'une solution au plus. Comme ce mode ne comporte également qu'un seul argument en mode *sortie*, il est possible de le retourner directement comme résultat de la fonction. Le code généré devient alors :

```
int append_ees (U, Y)
typeliste U, Y ;
{
   /*--- Clause 1 ---*/
   if (U == NIL)
      return (Y) ;

   /*--- Clause 2 ---*/
   { typeliste A, X, Z, V ;
     if (celcons(U))
     { A = car(U) ;
       X = cdr(U) ;
       if (Z = append_ees (X, Y))
       { V = cons (A, Z) ;
         return (V) ;
   } } }

   /*--- Fin ---*/
   return (FAUX) ;
}
```

La génération d'un tel programme suppose la connaissance de la propriété de déterminisme pour le prédicat traité. Dans le cas de Starlog, cette propriété est spécifiée par le programmeur, qui peut l'associer à chaque mode indépendamment.

La connaissance des types peut également être source d'optimisations intéressantes au niveau de la production du code. Ainsi, si l'on peut savoir de façon certaine que le premier argument est toujours une liste bien formée (c'est à dire NIL ou une cellule de liste), on peut alors générer :

```
int append_ees (U, Y)
typeliste U, Y ;
{
   /*--- Clause 1 ---*/
   if (U == NIL)
      return (Y) ;

   /*--- Clause 2 ---*/
   {  typeliste A, X, Z, V ;
      A = car(U) ;
      X = cdr(U) ;
      Z = append_ees (X, Y) ;
      V = cons (A, Z) ;
      return (V) ;
} }
```

que l'on peut finalement transformer en :

```
int append_ees (U, Y)
typeliste U, Y ;
{
   /*--- Clause 1 ---*/
   if (U == NIL)
      return (Y) ;

   /*--- Clause 2 ---*/
   return (cons(car(U), append_ees(cdr(U), Y))) ;
}
```

Ces deux dernières formes ne sont pas accessibles en Starlog car les outils d'inférence ou de déclaration de type n'ont pas été développés. On peut également noter que l'on peut encore améliorer ces programmes en utilisant des techniques d'optimisation globale classiques (élimination de la récursivité par exemple).

1.3. Interprétation

L'interpréteur du système Starlog utilise la même stratégie de résolution que celle utilisée pour la génération de code. Toutefois, il a été doté d'une option, essentiellement utilisée pour la mise au point, permettant une résolution sans utilisation des déclarations de modes. Pour ce faire, une

stratégie originale d'ordonnancement dynamique des sous-buts a été utilisée.

Cette stratégie d'ordonnancement dynamique est fondée sur l'hypothèse que la résolution d'un prédicat appelé dans un mode incorrect n'aboutit pas. Un mode correct se caractérisant par le fait que le prédicat ne retourne que des n-uplets complètements instanciés, le non-aboutissement peut se manifester de deux façons différentes :

- La résolution du prédicat se termine mais les n-uplets solution sont incomplètement instanciés.

- La résolution ne se termine pas : le prédicat se met à boucler en se rappellant récursivement avec les mêmes arguments.

L'interprétation d'une conjonction consiste à tenter l'effacement des sous-buts dans l'ordre de leur écriture, et à remettre en cause cet ordre si la résolution de l'un d'eux n'aboutit pas. La détection de non-aboutissement est effectuée de façon simple, en vérifiant si l'on se trouve dans l'une des deux situations précédentes :

- La détection du bouclage est réalisée à l'aide d'une pile contenant les arguments des différents appels récursifs à un prédicat. A chaque nouvel appel, cette pile est parcourue pour déterminer si l'appel n'a pas été effectué précédemment.

- Le test de n-uplet insuffisamment instancié s'effectue de façon triviale en contrôlant qu'au moins un argument solution n'est pas un terme clos.

Il faut indiquer que la stratégie présentée est purement pragmatique, aucune preuve formelle n'ayant été réalisée sur son bon fonctionnement.

1.3.1. *Algorithme de résolution*

En conséquence des contrôles indiqués ci-dessus, la résolution d'un sous-but (appel à un prédicat) peut retourner deux types de résultat :

Succès total
Dans ce cas, le sous-but est définitivement effacé. Les n-uplets solution sont retournés dans une liste, vide si le prédicat produit un échec.

Succès partiel
La résolution du prédicat a été arrêtée pour non-aboutissement. Le

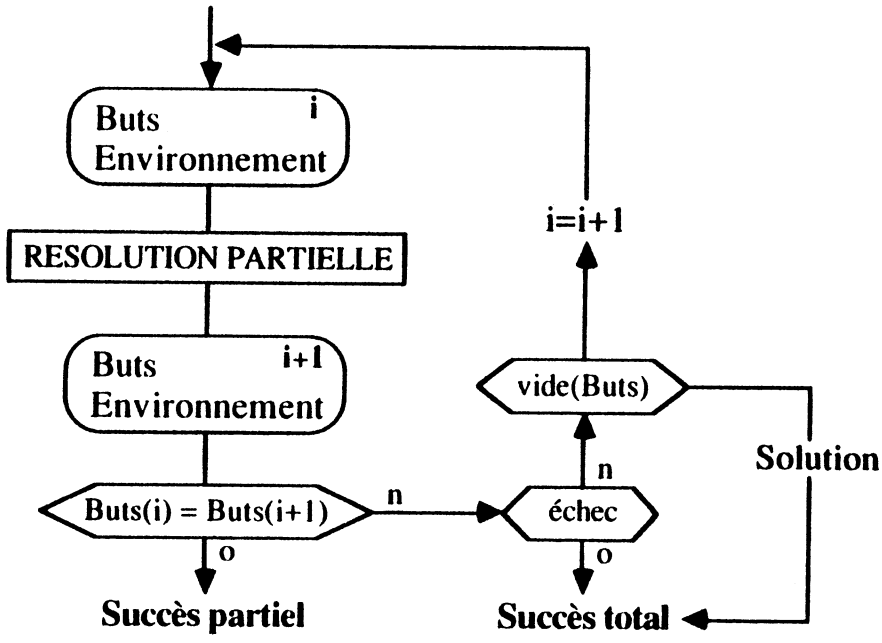
sous-but n'a donc pas pu être résolu intégralement car appelé dans un mode incorrect.

La résolution d'une clause s'effectue en partant d'un *environnement* initial, obtenu à partir des constantes de la clause et des arguments placés en entrée. Chaque but de la queue de la clause est ensuite résolu en s'appuyant sur cet environnement qui fournit les instances de ses arguments. Si le résultat est un *succès total*, les solutions qu'il fournit sont placées dans l'environnement. Si c'est un *succès partiel*, le sous-but est placé en attente d'informations complémentaires (plus d'arguments instanciés) et devient donc *réapplicable*.

L'algorithme de résolution partielle effectue la résolution de tous les sous-buts séquentiellement et retourne, en plus du nouvel environnement, une liste de sous-buts à réappliquer. Résoudre totalement une clause consiste à itérer ce processus jusqu'à ce que la liste des sous-buts réapplicables soit vide. La solution peut alors être extraite de l'environnement et retournée à l'appelant.

Pour éviter les bouclages dus à des applications successives des mêmes sous-buts toujours réapplicables, un mécanisme se charge de contrôler que la liste de sous-buts décroit à chaque itération.

Si la résolution d'un sous-but fournit plusieurs solutions, l'algorithme est appliqué indépendamment sur l'environnement de chaque solution.



Nous présentons ci-après quelques exemples de résolution de clauses

pour illustrer le fonctionnement de l'algorithme.

Soit le programme :

```
eq(x, x) ←.
Q(x, y, z, t) ← eq(x, y),    (1)
                eq(y, z),    (2)
                eq(z, t).    (3)
```

La résolution de ←Q(x, y, z, A) se déroule de la manière suivante :

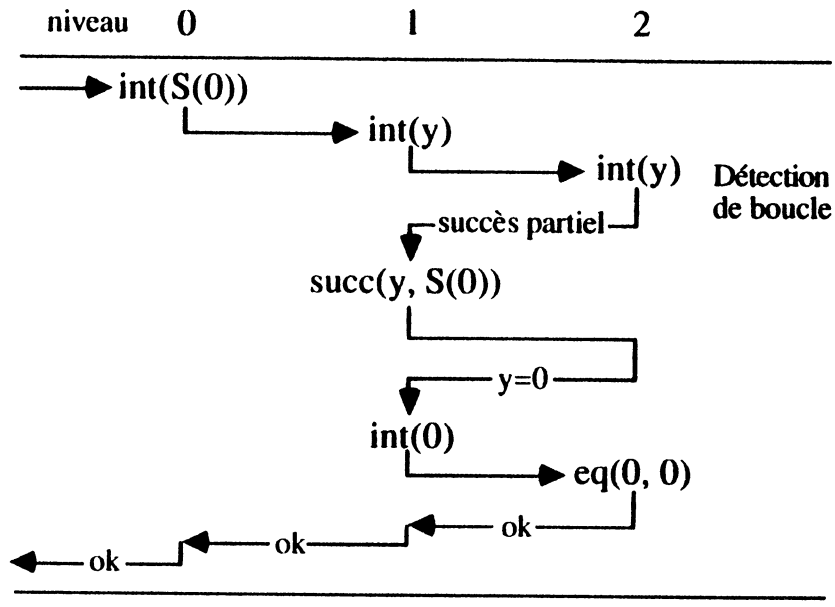
Sous-but courant	Environnement (x, y, z, t)	Liste des sous-buts	Sous-buts réapplicables
Initialisation	(x, y, z, A)	1, 2, 3	
(1) eq(x, y)	(x, y, z, A)	2, 3	1
(2) eq(y, z)	(x, y, z, A)	3	1, 2
(3) eq(z, A)	(x, y, A, A)		1, 2
Réapplication		1, 2	
(1) eq(x, y)	(x, y, A, A)	2	1
(2) eq(y, A)	(x, A, A, A)		1
Réapplication		1	
(1) eq(x, A)	(A, A, A, A)		
Solution	(A, A, A, A) ⇒ x=A, y=A, z=A		

Dans cet exemple, on voit clairement que les buts, bien qu'écrits dans l'ordre 1-2-3 sont en fait effacés dans l'ordre 3-2-1.

Pour illustrer l'arrêt par détection de boucle, considérons le programme suivant :

```
eq(x, x) ←.
succ(x, S(x)) ←.
int(x) ← eq(x, 0).
int(x) ← int(y), succ(y, x).
```

La résolution de ←int(S(0)) se déroule de la manière suivante :



Cette technique montre une efficacité suffisante (quoique faible) bien que la réapplication engendre, par son principe, beaucoup de travail inutile. Le principal problème de cette approche reste les prédicats à effets de bords (entrées/sorties entre autres) qui peuvent être effectués plusieurs fois. Ce problème est résolu par le fait que les prédicats prédéfinis correspondants ont été programmés de façon à n'agir que lorsqu'ils sont appelés dans un mode correct, entrainant alors leur effacement définitif.

2. Mesures de performances

Les mesures de performances sont en général très difficiles à réaliser car elles mettent en jeu des facteurs très variés. Toutefois, il nous a semblé important d'en effectuer pour montrer l'intérêt de nos propositions en matière d'efficacité. Nous avons essayé de prendre ces mesures dans les conditions les plus couramment employées, de façon à pouvoir comparer nos résultats avec ceux d'autres systèmes Prolog. Nous avons dû ainsi éviter l'utilisation de certaines possibilités de Starlog ; c'est le cas de certains prédicats prédéfinis dont la mise en oeuvre a été très soignée du point de vue efficacité, ce qui favoriserait notre système.

2.1. Vitesse de résolution

L'unité de mesure d'efficacité d'un interprète Prolog employée le plus couramment est le nombre d'unifications réussies par seconde ("logical inference per second" ou **lips**). Cette mesure est relativement floue dans la mesure où le temps nécessaire à une unification est variable en fonction de la taille des termes à unifier. De plus, elle ne tient pas compte des unifications échouées. Cette mesure ne doit donc être prise qu'à titre indicatif.

Les mesures ont été réalisées pour des programmes considérés comme des "benchmarks". Une comparaison est effectuée avec BIM-Prolog 1.0 *[Janssens 85]*, l'un des systèmes Prolog les plus efficaces sur le marché, doté d'un compilateur utilisant les modes de Warren. Il est également l'un des seuls pour lequel nous avons des mesures précises sur une machine dont nous disposons également.

Les tests ont été effectués sur SUN-2 (Unix 4.2) et portent sur cinq programmes choisis parmi les tests proposés dans *[Janssens 85]*. Ces programmes sont donnés ci-après, dans la syntaxe Starlog. Les variables commencent par une majuscule et les constantes par une minuscule. Les déclarations de mode utilisent le symbole "?" pour désigner le mode *entrée*, et le symbole "^" pour le mode *sortie*.

naive reverse

```
mode (main()).
main  == list30(X), cputime(T1), nreverse(X, Y), cputime(T2),
         minus(T2, T1, T), print("Temps CPU = ", T, " ms\n").
```

```
mode (nappend(?,?,^), append(^,^,?)).
nappend(A.X, Y, A.Z) == nappend(X, Y, Z).
nappend(X, nil, nil) ==.

mode (nreverse(?,^), nreverse(^,?)).
nreverse(A.X, Y) == nreverse(X, XR), nappend(XR, [A], Y).
nreverse(nil, nil) ==.

mode (list30(^)).
list30([1, 2, 3, 4, 5, 6, 7,....., 27, 28, 29, 30]) ==.
```

fibonacci

```
mode (main()).
main  == cputime(T1), fibo(17), cputime(T2),
         print("Temps CPU = ", :-(T2,T1), " ms\\n").

mode (fibo(?,^)).
fibo(0, 0) == !.
fibo(1, 1) == !.
fibo(X, Y) == minus(X, 1, X1), minus(X, 2, X2),
              fibo(X1, Y1), fibo(X2, Y2), plus(Y1, Y2, Y).
```

quick-sort d'une liste triée

```
mode (main()).
main  == liste(L), cputime(T1), qsort(L, R, nil), cputime(T2),
         minus(T2, T1, T), print("Temps CPU = ", T, " ms\\n").

mode (qsort(?,^,?)).
qsort(nil, R, R) ==.
qsort(H.T, R, X) == split(H, T, U1, U2), qsort(U1, R, H.Y),
                    qsort(U2, Y, X).

mode (split(?,?,^,^)).
split(H, H1.T1, H1.U1, U2) == lsequal(H1, H),  split(H, T1, U1, U2).
split(H, H1.T1, U1, H1.U2) == greater(H1, H), split(H, T1, U1, U2).
split(H, nil, nil, nil) ==.

mode (liste(^)).
liste([1, 2, 3, 4, 5, 6, 7,....., 48, 49, 50]) ==.
```

quick-sort d'une liste inverse

idem mais avec le prédicat liste *suivant :*

```
mode (liste(^)).
liste([50, 49, 48, 47, 46, 45,.....,5, 4, 3, 2, 1]) ==.
```

quick-sort d'une liste aléatoire

idem mais avec le prédicat liste *suivant :*

```
mode (liste(^)).
liste([482, 590, 522, 355, 836, 993, 201, 724, 149, 608, 527, 443,
       239, 841, 457, 723, 522, 1005, 789, 537, 516, 465, 309,
       288, 524, 89, 62, 263, 183, 529, 825, 522, 760, 283, 89,
       258, 639, 925, 228, 840, 626, 377, 424, 129, 820, 663,
       970, 253, 363, 468])==.
```

Les différentes mesures sont exprimées en milli-secondes et sont données pour une résolution compilée. Elles sont effectuées, dans le cas de Starlog, de deux façon distinctes, respectivement sans et avec l'exploitation de la propriété de déterminisme des prédicats.

Programme	**Starlog** sans déterminisme	**Starlog** avec déterminisme	**BIM Prolog**
naive reverse	48 ms	20 ms	20 ms
fibonacci	740 ms	460 ms	2560 ms
qsort triée	380 ms	320 ms	600 ms
qsort inverse	340 ms	220 ms	340 ms
qsort aléatoire	80 ms	40 ms	80 ms

Un facteur correctif est à apporter à ces valeurs dans la mesure où les stratégies diffèrent profondément. Pour Starlog, la valeur donnée représente le temps nécessaire au parcours de **tout** l'arbre de recherche alors que pour BIM-Prolog, elle ne représente que le temps nécessaire pour arriver à la première solution, donc sans compter les branches en attente de retour arrière.

Une analyse de ce tableau montre clairement le gain de performance obtenu en exploitant le déterminisme des prédicats. La différence, très importante dans le cas du *naive reverse*, est toutefois moindre pour le même problème sur d'autres machines telles que le Vax 780, sur lequel nous avons

également effectué ces essais.

La comparaison avec BIM-Prolog est globalement favorable à Starlog. Notons toutefois que BIM-Prolog offre beaucoup plus de fonctionnalités que Starlog, comme la compilation dynamique des *assert*, et que la comparaison n'est effectuée que sur une partie assez peu représentative d'un système complet.

On peut également remarquer que BIM-Prolog génère directement du code machine alors que Starlog effectue cette génération par l'intermédiaire du compilateur C. Une analyse effectuée sur le code généré par ce compilateur pour le 68000 permet d'envisager un gain non négligeable par génération directe de code machine. A titre indicatif, le temps nécessaire à l'empilement d'une valeur sur la pile PTAND où PTOR peut être divisé par 6 en utilisant des registres internes du processeur pour représenter ces deux pointeurs, alors que nous sommes obligés en C d'utiliser pour cela des variables globales.

A titre comparatif, nous avons également testé le *naive reverse* en utilisant la forme de programme que l'on obtiendrait par l'utilisation des types, donnée à la fin du paragraphe 2.2.1. Comme cette transformation n'est pas effectuée par Starlog, nous avons modifié à la main le programme C obtenu par une compilation normale. Les résultats obtenus montrent clairement l'intérêt de cette technique puisque le temps réalisé est de 15ms, ce qui correspond à un gain de 25%.

2.2. Espace mémoire

La mémoire nécessaire à la résolution est un facteur important d'efficacité. Nous analysons ci-après l'espace mémoire utilisé en raison de la stratégie de résolution, ainsi que l'espace occupé par le système lui-même.

2.2.1. *Espace inhérent à la stratégie*

La stratégie utilisée force le maintien des solutions au niveau de chaque littéral d'une conjonction. Il est évident que cette caractéristique peut être un grand inconvénient lorsque l'arbre de recherche est très large.

Exemple : la résolution de

P(X, Y) == append(X, Y, *[liste de 1000 éléments]*).

nécessite l'utilisation d'un espace de 500000 cellules (de l'ordre de 4 MOctets), uniquement pour maintenir l'ensemble des solutions.

Par contre, cette stratégie favorise les problèmes déterministes ayant un arbre de recherche profond car les environnements locaux sont récupérés par simple retour procédural. Ainsi, par exemple, la résolution interprétée du "naive reverse" d'une liste de 145 éléments est encore possible en Starlog avec un espace de travail de 10 Koctets (750 cellules + 1000 éléments de pile). Par contre, l'interprète C-Prolog *[Pereira 84]* s'arrête pour débordement de la pile locale (128 Koctets) pour une liste de seulement 85 éléments.

Les programmes compilés sont encore plus efficaces car les prédicats déterministes sont transformés en fonctions. Ainsi, par exemple, si l'environnement de la fonction C correspondante au prédicat *fibonacci* occupe N octets, la résolution de *fibonacci*(M) nécessite au maximum N*M octets. En supposant raisonnablement que N est inférieur à 50 octets, le calcul de fibonacci(50) nécessite donc tout au plus 2500 octets.

Si le prédicat compilé fait référence à des objets dans l'espace des listes, la limitation de cet espace peut alors engendrer une limitation dans la profondeur de récursion.

2.2.2. *Espace occupé par le système*

Le système Starlog, incluant l'interprète, le compilateur et 76 prédicats prédéfinis, occupe une place assez réduite. L'occupation mémoire du système standard est la suivante :

Volume du code = 70k (contre 175 pour C-Prolog)
Espace de 10000 cellules = 80k
Pile d'exécution de 10000 éléments = 40k
Tas de stockage des chaînes < 50k
Bibliothèque (seulement pour les programmes compilés) = 20k

Le volume mémoire occupé s'élève donc à 240k. Comparativement, l'interprète C-Prolog nécessite, pour un fonctionnement correct, un volume total proche du Méga-octet.

4. Bilan de l'expérience

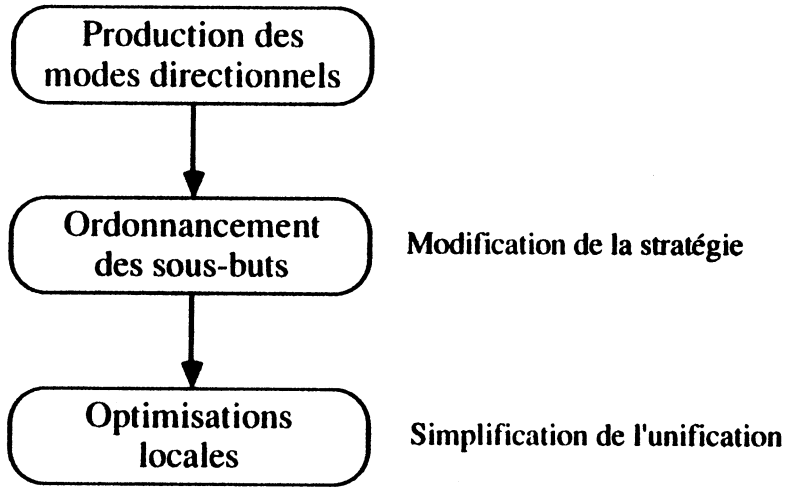
La réalisation de Starlog a suivie de près l'évolution de notre travail et contient donc encore de nombreux "exotismes" dus aux essais successifs de différentes techniques. Starlog ne peut donc en aucun cas être considéré comme un produit fini. Une amélioration notable pourrait être obtenue en intégrant au système les outils de production automatique des modes, écrits par ailleurs en Lisp. Ceci éviterait au programmeur la lourde tâche de leur définition.

Malgré tout, l'utilisation de ce prototype et les expériences que nous avons pu réaliser avec lui ont permis de montrer que l'utilisation exclusive des termes clos dans la résolution n'est pas une contrainte insurmontable. En effet, nous avons constaté que les problèmes nécessitant l'utilisation de modes complexes (non acceptés par Starlog) sont programmés délibérement dans cette optique (comme par exemple les *difference list [Fribourg 87]*). Les problèmes classiquement traités en Prolog ont donc pu être abordés en Starlog dans leur grande majorité, même si leur programmation en a été parfois un peu différente, avec une approche plus fonctionnelle.

Conclusion

1. Bilan général

Nous avons vu que les modes directionnels permettent d'aborder, par une seule technique, différents aspects de l'optimisation de la Résolution. Ceci est important à nos yeux dans la mesure où une homogénéïté globale est ainsi aisément obtenue. Leur utilisation suit le schéma suivant :



La première catégorie d'optimisations réalisables à l'aide des modes directionnels n'est pas directement liée à une implantation particulière et concerne l'aspect contrôle du langage. Dans ce domaine, les modes directionnels permettent d'extraire et d'utiliser de multiples informations dans le but de trouver une "meilleure façon" de parcourir l'arbre de recherche défini par un ensemble de clauses. Un réordonnancement statique des littéraux des queues de clauses permet alors :

- d'éviter le parcours d'un grand nombre de branches infinies,

- de réduire le coût de résolution,

- d'améliorer la réversibilité des prédicats,

- de réduire l'espace mémoire occupé,

- une meilleure génération de code à la compilation.

La seconde catégorie d'optimisations qu'il est possible d'effectuer à l'aide des modes directionnels est beaucoup plus locale et concerne l'unification elle-même. L'utilisation conjointe des modes stricts et de l'expansion de l'unification s'applique particulièrement à ce domaine, et autorise les points suivants :

- L'unification dans sa forme la plus générale peut être totalement supprimée. Les modes stricts fonctionnels permettent de la restreindre à la reconnaissance de formes, qui peut par ailleurs être transformée en simple égalité grâce à l'expansion.

- Dans le cas des modes stricts fonctionnels, les termes manipulés peuvent être réduits aux seuls termes clos, ce qui permet une implantation plus simple et plus efficace, en utilisant par exemple un noyau Lisp comme cela a été fait dans Starlog.

- Il est possible de transformer les programmes Prolog en programmes fonctionnels. Ceci permet notamment d'utiliser des techniques d'évaluation différentes de celles employées habituellement pour Prolog.

La réalisation du prototype Starlog, basé sur l'utilisation des modes stricts, a montré l'importance des gains réalisables. Toutefois, ce prototype reste spécialisé dans l'utilisation *exclusive* des modes stricts et n'aborde pas les problèmes qui nécessitent l'utilisation de techniques de résolution classiques. Les expériences effectuées avec lui ont cependant montré le large domaine d'application de cette technique.

2. Perspectives et développements

Les développements envisageables à partir de notre expérience des modes directionnels sont multiples et concernent indifféremments les aspects théoriques et pratiques de la résolution.

2.1. Développements théoriques

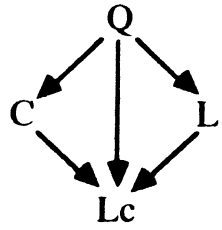
Les modes directionnels ne représentent qu'un sous-ensemble des modes structurels. Les travaux théoriques entamés ici permettent d'envisager l'extension du théorème de correction à l'ensemble des modes structurels ; des travaux sont d'ores et déjà en cours à ce propos. Ceci permettrait indirectement de fournir un algorithme de production de modes paramétrable par les propriétés structurelles désirées.

Toutefois, il est clair que seules les propriétés qui permettent une optimisation réelle au niveau de l'implantation présentent un intérêt. Ainsi, par exemple, la propriété "est un terme comportant au moins un entier" ne présente aucun intérêt pour optimiser la génération de code.

Les modes directionnels ne traitent que les propriétés Clos et Quelconque. Ceci est suffisant pour permettre une bonne exploitation des modes stricts mais ne permet pratiquement aucune optimisation lorsque l'on travaille avec des modes complexes. Or, une structure de données souvent utilisée dans les programmes Prolog est la liste :

liste = NIL
liste = terme . liste

L'intégration de cette propriété dans l'ensemble {Clos, Quelconque} apporterait des précisions appréciables sur les modes et permettrait des optimisations intéressantes sur tous les prédicats de traitement des listes (cf. chapitre 4). Le graphe de dépendance des modes deviendrait alors :



où L désigne la propriété "est une liste" et Lc "est une liste close".

La production de tels modes est alors sujette à de nombreuses optimisations ; en effet, le lemme de conjonction nous permet d'utiliser les modes directionnels pour produire rapidement des modes comportant les listes. Ainsi, par exemple, le mode du prédicat *reverse* :

$$(C, Q) \rightarrow (C, C)$$

induit, en conjonction avec la propriété L, le mode :

$$(Lc, L) \rightarrow (Lc, Lc)$$

En réalité, les propriétés utiles à la production des modes d'un programme donné sont souvent dépendantes de ce programme. En effet, les listes précédemment citées ne sont utiles en tant que propriété que si le programme les utilise effectivement. De plus, la production des modes étant fortement dépendante du nombre de propriétés à traiter, il est clair que tenter d'inférer des modes comportant des propriétés inutilisées serait d'un coût prohibitif, et par conséquent nuisible.

C'est pourquoi, il nous semble intéressant d'allier la production des modes à une connaissance des types (domaines), pouvant être obtenue automatiquement. Sur cette base, il serait alors possible de définir des propriétés utiles au programme, et d'éviter ainsi celles qui ne sont pas utilisées par celui-ci. On pourra ainsi réduire le coût de la production des modes.

2.2. Développements pratiques

Nous avons montré de manière théorique comment identifier automatiquement dans un programme Prolog les parties qui peuvent être évaluées en utilisant uniquement les modes stricts, et les parties nécessitant une résolution classique, utilisant les modes complexes. Cependant, nous n'avons pas réalisé d'implantation combinant les deux techniques et nous pouvons supposer que cette coexistence doit poser des problèmes d'interface qui n'ont pas été étudiés. Il est probable que la solution choisie dans Starlog ne puisse plus, ou alors difficilement, s'appliquer dans ce cas.

Un aspect pratique intéressant consisterait donc à étudier les différentes possibilités d'implantation combinant les deux types de modes. Ce travail pourrait aboutir à moyen terme à la réalisation d'un système intégrant, en un seul ensemble, tous les outils d'inférence décrits dans cette thèse, et exploitant au mieux les possibilités de résolution en modes stricts.

Un deuxième aspect intéressant réside dans la possibilité de transformer les programmes strictement modés en programmes fonctionnels. Ceci semble d'un grand intérêt pour envisager notamment une intégration aisée d'une partie de la logique du premier ordre dans la programmation fonctionnelle.

Dans ce domaine, l'expérience de Starlog nous a montré, d'une part que la restriction aux modes stricts n'est pas un gros handicap à l'utilisation du langage, et d'autre part qu'une implantation basée sur un noyau Lisp est réalisable. Les caractéristiques principales du langage Lisp, dont notamment la non distinction entre programme et données, en font l'un des meilleurs candidat à cette extension. Sa réalisation, écrite en Lisp, a déjà été entamée ; elle a pour but de transformer, sous certaines conditions, les prédicats Prolog en fonctions Lisp, complètement intégrables au système hôte.

3. Conclusion

L'inefficacité relative de la Résolution est essentiellement due à l'inadéquation de l'architecture des machines sur lesquelles elle est actuellement implantée. Dans ce cadre, les modes directionnels ne représentent pas en réalité une technique d'optimisation, mais plutôt un moyen d'adapter l'implantation de la Résolution à ces architectures.

On ne peut donc considérer les modes directionnels que comme un palliatif, dans l'attente d'une redéfinition complète des principes régissant l'architecture des machines. Dans ce domaine, le parallélisme semble offrir des perspectives intéressantes, bien que la conception et l'utilisation de telles machines se heurtent encore à des problèmes d'une très grande complexité.

Bibliographie

[Bekkers 85]
Y.Bekkers, B.Canet, L.Ungaro
"Projet Mali (Mémoire Adaptée aux Langages Indéterministes)"
Octobre 1985 : Bilan et perspectives, RR IRISA, Rennes, 1985

[Berger 82]
G.Berger Sabbatel, G.T.Nguyen
"Projet OPALE : Motivations et principes pour une machine Base de Données Prolog"
RR.339, IMAG Grenoble, Décembre 1982

[Berger 84]
G.Berger Sabbatel, W.Dang, J.C.Ianeselli, G.T.Nguyen
"Unification for a Prolog Data Base Machine"
Second International Logic Programming Conference, Uppsala, 1984

[Berger 84]
G.Berger Sabbatel
"Mesures comportementales sur l'interprétation de Prolog"
Actes du séminaire de Programmation Logique, Trégastel, Mai 1986

[Bruynooghe 82]
M.Bruynooghe
"The Memory Management of Prolog Implementations"
Logic Programming, Clark Tärnlund ed., Academic press 1982, 83-98

[Chang 73]
C.L.Chang, R.C.Lee
"Symbolic Logic and Mechanical Theorem Proving"
Academic Press, New York, 1973

[Chassin 86]
J. Chassin de Kergommeaux
"Machines Abstraites pour l'Implantation de Prolog"
LGI-IMAG, RR.589, Grenoble, Février 1986

[Clocksin 85]
W.F.Clocksin
"Design and Simulation of a Sequential Prolog Machine"
New Generation Computing, Vol.3, N°1,
Springer-Verlag 1985, 101-120

[Codognet 87]
P.Codognet
"Programmation Logique Parallèle 87"
Séminaire de Programmation en Logique, Trégastel, Mai 1987, 153-172

[Colmerauer 82]
A.Colmerauer
"Prolog II: Manuel de référence et modèle théorique"
Groupe Intelligence Artificielle ERA CNRS 363, Mars 1982

[Colmerauer 83]
A.Colmerauer
"Prolog, bases théoriques et développements"
Techniques et Sciences Informatiques, Vol.2(4), Juillet 1983, 271-311

[Conery 81]
J.S.Conery, D.F.Kibler
"Parallel Interpretation of Logic Programs"
Conf. on Functional Programming Languages and Computer
Architecture ACM, Portsmouth, Octobre 1981, 163-170

[Conery 85]
J.S.Conery, D.F.Kibler
"AND Parallelism and Nondeterminism in Logic Programs"
New Generation Computing, Vol.3, N°1, Springer-Verlag 1985, 43-70

[Conrad 86]
T.Conrad
"Réarrangement de clauses Prolog"
Programmation en Logique, Séminaire Trégastel 1986, 345-355

[Debray 84]
S.K.Debray
"Optimizing Allmost-Tail-Recursive Prolog Programs"
Technical Report #84/089, Dept. of Computer Science,
State University of New York at Stony Brook, 1984

[Debray 85a]
S.K.Debray
"Automatic Mode Inference for Prolog Programs"
Technical Report #85/19, Dept. of Computer Science,
State University of New York at Stony Brook, Juin 1985

[Debray 85b]
S.K.Debray
"Detection and Optimization of Functional Computations in Prolog"
Technical Report #85/020, Dept. of Computer Science,
State University of New York at Stony Brook, Août 1985

[Deransart 84a]
P.Deransart, J.Maluszynski
"Modelling Data Dependencies in Logic Programs by Attribute Schemata"
Rapport de Recherche N°323, INRIA, Juillet 1984

[Deransart 84b]
P.Deransart, J.Maluszynski
"Relating Logic Programs and Attribute Grammars"
Research Report, Department of Computer and Information Science
Linköping University, S-581 83, Linköping, Sweden, Octobre 1984

[Dincbas 85]
M.Dincbas, S.Bourgault, JP.Lepape
"Notes de cours Prolog"
Cours INRIA, Méthodes et langages de l'IA, Novembre 1985, 61-116

[Fribourg 87]
L.Fribourg
"List Concatenation via Extended Unification"
Séminaire de Programmation en Logique, Trégastel, Mai 1987, 45-58

[Gallaire 82]
H.Gallaire, C.Lasserre
"Metalevel Control for Logic Programs"
Logic Programming, Clark Tärnlund ed., Academic Press 1982, 99-106

[Giannesini 85]
F.Giannesini, H.Kanoui, R.Pasero, M.Van Caneghem
"Prolog"
InterEdition, Paris 1985

[Janssens 85]
G.Janssens
"A set of benchmarks for Prolog applied to BIM-Prolog 1.0"
K.U.Leuven, Departement Computerwetemschappen Celestijnenlaan 200A, 3030 Leuven Belgium, Octobre 1985

[Kanamori 84]
T.Kanamori, K.Horiuchi
"Type Inference in Prolog and its Applications"
ICOT Technical Report TR-095, Décembre 1984

[Kowalski 74]
R.Kowalski
"Predicate Logic as Programming Language"
Information Processing 74, North Holland 1974, 569-574

[Kowalski 76]
R.Kowalski, M.VanEmden
"The semantics of predicate logic as programming language"
Journal of the ACM, Vol.23, N°4, Octobre 1976, 733-742

[Kowalski 79a]
R.Kowalski
"Algorithm = Logic + Control"
Communications of the ACM, Vol.22, N°7, Juillet 1979, 424-436

[Kowalski 79b]
R.Kowalski
"Logic for Problem Solving"
Elsevier Science Publishing, 1979

[Lebaube 86]
P.Lebaube, B.Lepape
"Prolog et les techniques de contrôle"
Actes du séminaire de Programmation en Logique, Trégastel, Mai 1986

[Maluszynski 85]
J. Maluszynski, H.J.Komorowski
"Unification-free Execution of Logic Programs"
IEEE, Mars 1985, 78-86

[Martelli 82]
A.Martelli, U.Montanari
"An Efficient Unification Algorithm"
ACM Transactions on Programming Languages and Systems,
Vol.4, N°2, Avril 1982, 258-282

[Matsumoto 85]
H.Matsumoto
"A Static Analysis of Prolog programs"
SIGPLAN Notices, Vol.20, N°10, Octobre 1985, 48-59

[Mellish 81]
C.S.Mellish
"The Automatic Generation of Mode Declarations for Prolog Programs"
Research Report 163, Dept. of Artificial Intelligence, Edinburgh, 1981

[Mellish 82]
C.S.Mellish
"An alternative to Structure Sharing in the Implementation of a Prolog Interpreter"
Logic Programming, Clark Tärnlund ed., Academic Press 1982, 99-106

[Mellish 85]
C.S.Mellish
"Some Global Optimizations for a Prolog Compiler"
Journal of Logic Programming, 1985, 43-66

[Mishra 84]
P.Mishra
"Towards a theory of types in Prolog"
International Symposium on Logic Programming Atlantic City,
New Jersey, Février 1984, 289-298

[Naish 85]
L.Naish
"Automating Control For Logic Programs"
Journal of Logic Programming, N°3, 1985, 167-183

[Naish 86]
L.Naish
"Negation and Control in Prolog"
Lecture Notes in Computer Science, Springer-Verlag, 1986

[Nilsson 71]
N.J.Nilsson
"Problem Solving methods in Artificial Intelligence"
Mc.Graw Hill, 1971

[Nilsson 83]
J.F.Nilsson
"On the Compilation of a Domain-Based Prolog"
Information Processing, North Holland, IFIP 1983, 293-298

[O'Keefe 85]
R.A.O'Keefe
"On the Treatment of Cuts in Prolog Source-level Tools"
IEEE, Mars 1985, 68-72

[Oudot 86]
O.Oudot
"Utilisation des modes stricts dans la Résolution"
R.R.596, LGI-IMAG, Grenoble, Février 1986

[Pelhat 87]
S.Pelhat
"Les boucles dans Prolog : Structures et Origines"
Séminaire de Programmation en Logique, Trégastel, Mai 1987, 153-172

[Pereira 84]
F.Pereira
"C-Prolog Version 1.5 User's Manual"
Edinburgh Computer Aided Architectural Design, Février 1984

[Reddy 84]
U.S.Reddy
"Transformation of Logic programs into Functional Programs"
International Symposium on Logic Programming Atlantic City, New Jersey, Février 1984, 187-196

[Reddy 85]
U.S.Reddy
"On the relationship between logic and functional languages"
Logic Programming : Functions, Relations and Equations
D.DeGroot, G.Lindstrom, Prentice Hall, 1985, 3-36

[Robinson 65]
J.A.Robinson
"A Machine-Oriented Logic Based on the Resolution Principle"
Journal of the ACM, Vol.12, N°1, Janvier 1965, 23-41

[Roussel 75]
P.Roussel
"Prolog : Manuel de référence et d'utilisation"
GIA, Université d'Aix-Marseille II, Septembre 1975

[Sawamura 85]
H.Sawamura, T.Takeshima, A.Kato
"Source-Level Optimization Techniques for Prolog"
ICOT Technical Memorandum TM-0091, Janvier 1985

[Shapiro 83]
E.Y.Shapiro
"A Subset of Concurrent Prolog and its Interpreter"
ICOT Technical Report TR-003, Février 1983

[Sterling 86]
L.Sterling, E.Y.Shapiro
"The Art of Prolog"
MIT Press Series in Logic Programming, E.Shapiro editor, 1986

[Syre 85]
J.C.Syre
"Une revue de modèles parallèles pour Prolog"
Technical Report CA-06, ECRC Munich, Mai 1985

[Tarjan 72]
R:E.Tarjan
"Depth first search and linear graph algorithms"
SIAM J. Computing, 1:2 146-160, 1972

[Turbo]
"Turbo-Prolog, the natural language of artificial intelligence"
Manuel d'utilisation de Turbo-Prolog

[Warren 77]
D.H.Warren
"Implementing Prolog-Compiling Predicate Logic Programs"
R.R. 39, Dept. of Artificial Intelligence, University of Edinburgh, 1977

[Warren 80]
D.H.Warren
"An Improved Prolog Implementation wich Optimize Tail Recursion"
Logic Programming Workshop, Tärnlund éditeur, Debrecen 1980,1-11

[Warren 83]
D.H.Warren
"An Abstract Prolog Instruction Set"
Technical note 309, Artificial Intelligence Center, 1983

Résumé

Cette thèse a pour but de présenter une catégorie particulière de modes, les modes directionnels, et de mettre en évidence leur utilité pour l'optimisation de la résolution dans le langage Prolog. Ils se caractérisent essentiellement par le fait qu'ils permettent de distinguer les différentes utilisations possibles d'un même prédicat. Ces modes offrent la possibilité d'améliorer le contrôle en modifiant la stratégie de résolution standard par réordonnancement statique des sous-buts des clauses. Dans certains cas, ils autorisent une transformation des programmes Prolog (ou d'une partie) en programmes fonctionnels. Ils permettent également diverses optimisations locales, dont notamment une grande spécialisation de l'algorithme d'unification en fonction des termes à unifier. Un algorithme de production automatique de ces modes est décrit.

L'étude est concrétisée par la réalisation du compilateur Starlog, fonctionnant sur un cas particulier de modes directionnels. Les mesures de performances mettent en évidence l'intérêt pratique de cette technique.

Mots-Clés

Prolog - Compilation - Stratégie de résolution - Unification - Termes clos - Optimisations - Modes.