Machines spécialisées et programmation en logique

Gilles Berger Sabbatel

THESE

présentée à l'Institut National Polytechnique de Grenoble

pour obtenir le titre de **DOCTEUR D'ETAT ES SCIENCES**

par **Gilles BERGER-SABBATEL**

MACHINES SPECIALISEES ET PROGRAMMATION EN LOGIQUE

Soutenue le 29 juin 1988 devant la Commission d'Examen

Composition du jury :

Président :	Monsieur J. MOSSIERE
Examinateurs :	Monsieur F. ANCEAU
	Monsieur J. DELLA DORA
	Monsieur P. JORRAND
	Monsieur J. ROHMER
	Monsieur M. SCHOLL

Thèse préparée au sein du Laboratoire TIM3.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

ésident: Daniel BLOCH

ce-Présidents: B. BAUDELET
H. CHERADAME
R. CARRE
J.M PIERRARD

Année universitaire 1984-1985

Professeur des Universités

ICEAU	François	E.N.S.I.M.A.G
RIBAUD	Michel	E.N.S.E.R.G
RRAUD	Alain	E.N.S.I.E.G
UDELET	Bernard	E.N.S.I.E.G
SSON	Jean	E.N.S.E.E.G
IMAN	Samuel	E.N.S.E.R.G
OCH	Daniel	E.N.S.I.E.G
IS	Philippe	E.N.S.H.G
NNETAIN	Lucien	E.N.S.E.E.G
NNIER	Etienne	E.N.S.E.E.G
UVARD	Maurice	E.N.S.H.G
ISSONNEAU	Pierre	E.N.S.I.E.G
YLE BODIN	Maurice	E.N.S.E.R.G
VAIGNAC	Jean-François	E.N.S.I.E.G
ARTIER	Germain	E.N.S.I.E.G
ENEVIER	Pierre	E.N.S.E.R.G
ERADAME	Hervé	U.E.R.M.C.P.P
ERUY	Arlette	E.N.S.I.E.G
IAVERINA	Jean	U.E.R.M.C.P.P
HEN	Joseph	E.N.S.E.R.G
UMES	André	E.N.S.E.R.G
RAND	Francis	E.N.S.E.E.G
RAND	Jean-louis	E.N.S.I.E.G
ICI	Noël	E.N.S.I.E.G
NLUPT	Jean	E.N.S.I.M.A.G
ULARD	Claude	E.N.S.I.E.G
NDINI	Alessandro	U.E.R.M.C.P.P
UBERT	Claude	E.N.S.I.E.G
NTIL	Pierre	E.N.S.E.R.G
ERIN	Bernard	E.N.S.E.R.G
YOT	Pierre	E.N.S.E.E.G
NES	Marcel	E.N.S.I.E.G
INIER	Jean-Michel	E.N.S.I.E.G
JSSAUD	Pierre	E.N.S.I.E.G
JOUBERT	Jean-Claude	E.N.S.I.E.G
JOURDAIN	Geneviève	E.N.S.I.E.G
LACOUME	Jean-Louis	E.N.S.I.E.G
LATOMBE	Jean-Claude	E.N.S.I.M.A.G
LESIEUR	Marcel	E.N.S.H.G
LESPINARD	Georges	E.N.S.H.G
LONGEQUEUE	Jean-Pierre	E.N.S.I.E.G
LOUCHET	François	E.N.S.E.E.G
MASSELOT	Christian	E.N.S.I.E.G
MAZARE	Guy	E.N.S.I.M.A.G
MOREAU	René	E.N.S.H.G
MORET	Roger	E.N.S.I.E.G
MOSSIERE	Jacques	E.N.S.I.M.A.G
PARIAUD	Jean-Charles	E.N.S.E.E.G
PAUTHENET	René	E.N.S.I.E.G
PERRET	René	E.N.S.I.E.G
PERRET	Robert	E.N.S.I.E.G
PIAU	Jean-Michel	E.N.S.H.G
POLOUJADOFF	Michel	E.N.S.I.E.G
POUPOT	Christian	E.N.S.E.R.G
RAMEAU	Jean-Jacques	E.N.S.E.E.G
RENAUD	Maurice	U.E.R.M.C.P.P
ROBERT	André	U.E.R.M.C.P.P
ROBERT	François	E.N.S.I.M.A.G
SABONNADIERE	Jean-Claude	E.N.S.I.E.G
SAUCIER	Gabrielle	E.N.S.I.M.A.G
SCHLENKER	Claire	E.N.S.I.E.G
SCHLENKER	Michel	E.N.S.I.E.G
SERMET	Pierre	E.N.S.E.R.G
SILVY	Jacques	U.E.R.M.C.P.P
SOHM	Jean-Claude	E.N.S.E.E.G
SOUQUET	Jean-Louis	E.N.S.E.E.G
VEILLON	Gérard	E.N.S.I.M.A.G
ZADWORNY	François	E.N.S.E.R.G

Professeurs Associés

ACKWELDER	Ronald	E.N.S.H.G
YASHI	Hirashi	E.N.S.I.E.G
PURDY	Gary	E.N.S.E.E.G

Professeurs Université des Sciences Sociales (Grenoble II)

LIET	Louis	
CHATELIN	Françoise	

Chercheurs du C.N.R.S

RRE	René	Directeur de recherche
JCHART	Robert	Directeur de recherche
RRAND	Philippe	Directeur de recherche
CHAUD	Georges	Directeur de recherche
LIBERT	Michel	Maître de recherche
SARA	Ibrahim	Maître de recherche
MAND	Michel	Maître de recherche
DER	Gilbert	Maître de recherche
RNARD	Guy	Maître de recherche
VID	René	Maître de recherche
PORTES	Jacques	Maître de recherche
OLE	Jean	Maître de recherche
NOUX	Damien	Maître de recherche
ORD	Dominique	Maître de recherche
GUELIN	Pierre	Maître de recherche
HOPFINGER	Emil	Maître de recherche
JOUD	Jean-Charles	Maître de recherche
KAMARINOS	Georges	Maître de recherche
KLEITZ	Michel	Maître de recherche
LANDAU	Ioan-Dore	Maître de recherche
LASJAUNIAS	Jean-Claude	Maître de recherche
MERMET	Jean	Maître de recherche
MUNIER	Jacques	Maître de recherche
PIAU	Monique	Maître de recherche
PORTESEIL	Jean-Louis	Maître de recherche
THOLENCE	Jean-Louis	Maître de recherche
VERDILLON	André	Maître de recherche
SUERY	Michel	Maître de recherche

Personnalités habilitées à diriger des travaux de recherche
(Décision du Conseil Scientifique)

E.N.S.E.E.G.

ALLIBERT	Colette
BERNARD	Claude
BONNET	Roland
CAILLET	Marcel
CHATILLON	Catherine
CHATILLON	Christian
COULON	Michel
DIARD	Jean Paul
EUSTATHOPOULOS	Nicolas
FOSTER	Panayotis
GALERIE	Alain
HAMMOU	Abdelkader
MALMEJAC	Yves (CENG)
MARTIN GARIN	Régina
NGUYEN TRUONG	Bernadette
RAVAINE	Denis
SAINFORT	(CENG)
SARRAZIN	Pierre
SIMON	Jean Paul
TOUZAIN	Philippe
URBAIN	Georges (Laborato des ultra-réfracta ODEILLO).

E.N.S.E.R.G.

BARIBAUD	Michel
BOREL	Joseph
CHOVET	Alain
CHEHIKIAN	Alain
DOLMAZON	Jean Marc
HERAULT	Jeanny
MONLLOR	Christian

E.N.S.I.E.G.

BORNARD	Guy
DESCHIZEAUX	Pierre
GLANGEAUD	François
KOFMAN	Walter
LEJEUNE	Gérard
MAZUER	Jean
PERARD	Jacques
REINISCH	Raymond

E.N.S.H.G.

ALEMANY	Antoine
BOIS	Daniel
DARVE	Félix
MICHEL	Jean Marie
OBLED	Charles
ROWE	Alain
VAUCLIN	Michel
WACK	Bernard

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

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

U.E.R.M.C.P.P.

CHARUEL	Robert

C.E.N.G.

CADET	Jean
COEURE	Philippe (LETI)
DELHAYE	Jean Marc (STT)
DUPUY	Michel (LETI)
JOUVE	Hubert (LETI)
NICOLAU	Yvan (LETI)
NIFENECKER	Hervé
PERROUD	Paul
PEUZIN	Jean Claude (LE
TAIEB	Maurice
VINCENDON	Marc

Laboratoires extérieurs :

C.N.E.T.

DEMOULIN	Eric
DEVINE	R.A.B.
GERBER	Roland
MERCKEL	Gérard
PAULEAU	Yves

I.N.S.A. Lyon

GAUBERT	C.

****

Personnalités habilitées à diriger des travaux de recherche
(Décision du Conseil Scientifique)

E.N.S.E.E.G.

LIBERT	Colette
RNARD	Claude
NNET	Roland
ILLET	Marcel
ATILLON	Catherine
ATILLON	Christian
ULON	Michel
DIARD	Jean Paul
EUSTATHOPOULOS	Nicolas
FOSTER	Panayotis
GALERIE	Alain
HAMMOU	Abdelkader
MALMEJAC	Yves (CENG)
MARTIN GARIN	Régina
NGUYEN TRUONG	Bernadette
RAVAINE	Denis
SAINFORT	(CENG)
SARRAZIN	Pierre
SIMON	Jean Paul
TOUZAIN	Philippe
URBAIN	Georges (Laboratoire des ultra-réfractaires ODEILLO).

E.N.S.E.R.G.

RIBAUD	Michel
REL	Joseph
OVET	Alain
CHEHIKIAN	Alain
DOLMAZON	Jean Marc
HERAULT	Jeanny
MONLLOR	Christian

E.N.S.I.E.G.

RNARD	Guy
SCHIZEAUX	Pierre
ANGEAUD	François
KOFMAN	Walter
LEJEUNE	Gérard
MAZUER	Jean
PERARD	Jacques
REINISCH	Raymond

E.N.S.H.G.

EMANY	Antoine
S	Daniel
RVE	Félix
MICHEL	Jean Marie
OBLED	Charles
ROWE	Alain
VAUCLIN	Michel
WACK	Bernard

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

RT	Didier
LMET	Jacques
URTIN	Jacques
COURTOIS	Bernard
DELLA DORA	Jean
FONLUPT	Jean
SIFAKIS	Joseph

U.E.R.M.C.P.P.

CHARUEL	Robert

C.E.N.G.

DET	Jean
EURE	Philippe (LETI)
LHAYE	Jean Marc (STT)
PUY	Michel (LETI)
JOUVE	Hubert (LETI)
NICOLAU	Yvan (LETI)
NIFENECKER	Hervé
PERROUD	Paul
PEUZIN	Jean Claude (LETI)
TAIEB	Maurice
VINCENDON	Marc

Laboratoires extérieurs :

C.N.E.T.

MOULIN	Eric
VINE	R.A.B.
GERBER	Roland
MERCKEL	Gérard
PAULEAU	Yves

I.N.S.A. Lyon

GAUBERT	C.

****

Je tiens a remercier:

Jacques Mossière, professeur à l'INPG, et directeur de l'ENSIMAG, qui m'a fait l'honneur de présider le jury de cette thèse,

François Anceau, directeur de la division architecture des ordinateurs au centre de recherche Bull de Louveciennes, qui a dirigé mes débuts dans la carriere de chercheur, et est a l'origine des travaux décrits ici,

Jean Rohmer, directeur de Bull-CEDIAG, qui a été rapporteur de cette thèse, et m'a aidé à en améliorer la premiere version. Il est également à l'origine de certaines idées du projet Opale.

Michel Scholl, de l'INRIA, rapporteur de cette thèse, et dont les critiques constructives m'ont permis d'en améliorer la version finale.

Jean Della Dora, professeur à l'ENSIMAG, directeur du laboratoire TIM3 où se sont déroulé les travaux décrits ici, et qui a accepté d'être membre de mon Jury.

Philippe Jorrand, directeur du LIFIA, qui a accepté d'être membre de mon Jury.

Je tiens a remercier aussi:

Bernard Courtois, directeur de recherches au CNRS, qui a présidé aux destinées de l'équipe de recherche en architecture des calculateurs de TIM3, assumant la succession de François Anceau après son départ à la Bull.

Tous ceux qui ont contribué au projet Opale, et plus particulièrement Weidong Dang, et Jean Christophe Ianeselli, qui n'ont pas ménagé leurs peines, malgré un contexte parfois difficile.

Le secrétariat de l'équipe d'architecture des calculateurs, qui m'a efficacement aidé dans l'organisation matérielle de la soutenance,

Tous mes camarades de l'équipe d'architecture des calculateurs, pour les nombreux échanges de vues que nous avons eu, et pour l'ambiance sympathique qu'ils ont contribué à créer dans l'équipe.

Et enfin, tous mes amis, dont le soutien ne m'a jamais manqué durant toutes les années de préparation de cette thèse.

A Marie Hélène.

RESUME

L'objet de cette thèse est la conception de systèmes informatiques basés sur des unités spécialisées dans des fonctions ou des types de traitement particuliers. Nous nous intéressons plus particulièrement à la programmation en logique comme type de langage d'un tel système.

L'intérêt des machines spécialisées et les problèmes de communication et de coopération dans un système à unités spécialisées (distribution fonctionnelle) sont d'abord discutés. Nous nous intéressons ensuite aux machines bases de données, en présentant le projet OPALE, qui vise à la conception d'une machine bases de données adaptée au traitement des connaissances. Les deux points essentiels de ce projet sont la définition d'une stratégie de recherche orientée vers le traitement d'ensembles de solutions, et un algorithme, implémentable en matériel, permettant d'unifier des ensembles de buts avec des termes lus sur disque, ceci en suivant le débit de transfert du disque.

Enfin, nous abordons le problème général de la conception de machines pour la programmation en logique. Diverses approches sont présentées et discutées à partir de mesures et évaluations sur l'activité d'un interpréteur Prolog.

Mots cles:

Machines spécialisées, programmation en logique, machines base de données, unification.

ABSTRACT

The aim of this thesis is to discuss the design of computer systems based on units specialized in particular functions or kinds of processing. We focus on logic programming as a language for such systems.

We first discuss the interest of specialized machines, and the problems related to communication and cooperation in distributed functions systems. Then, we focus on data base machines, and present the project OPALE, which aims at designing a data base machine adapted to knowledge processing. The keypoints of this project are the definition of a search strategy oriented toward the processing of sets of solutions, and an algorithm (implementable in hardware) which allows the unification of sets of goals with terms read from a disk, following the disk transfer rate.

Last, we discuss about the general problem of designing machines for logic programming. Various approaches are presented and discussed, taking into account some measurements on a Prolog interpreter.

Keywords:

Specialized machines, logic programming, data base machines, unification.

TABLE DES MATIERES

Chapitre III: MACHINES BASES DE DONNEES

Chapitre IV: PROLOG ET BASES DE DONNEES

Chapitre I

INTRODUCTION

I.1. EVOLUTION DE L'UTILISATION DES ORDINATEURS:

Conçus à l'origine pour des traitements essentiellement numériques, les systèmes informatiques se sont de plus en plus orientés vers des traitements non numériques, avec des applications telles que bases de données, documentation, traitement de textes, messagerie, systèmes experts, CAO, etc...

Il en résulte qu'on voit se développer dans les systèmes des services qui représentent des fonctions complexes: systèmes de gestion de bases de données (SGBD), systèmes de gestion de fichiers, éditeurs de textes, systèmes de messagerie, fonctions graphiques, etc...

La communication avec l'utilisateur devient aussi plus complexe pour la machine, alors même qu'elle tend à devenir plus simple pour l'utilisateur: systèmes interactifs, langages puissants et proches de l'utilisateur, interfaces évolués (écrans pleine page, communication graphique, interfaces vocaux, etc...). On voit donc s'agrandir le fossé entre un matériel basé sur la logique binaire et un utilisateur peu enclin à s'adapter à la machine.

Un système peut ainsi être vu comme un réseau de services ou fonctions structurés en couches, les programmes d'un niveau donné n'étant que des utilisateurs des fonctions de niveau inférieur. Bien souvent, l'utilisateur lui même ne fait qu'utiliser des fonctions prédéfinies. On a donc une longue chaîne de fonctions successives entre l'utilisateur et les ressources physiques de la machine.

Se rapprocher plus des fonctions réclamées par l'utilisateur n'est plus possible dans le cadre d'une machine générale et partagée, sous peine d'une utilisation peu efficace des circuits de la machine. On voit alors se développer des machines spécialisées: machines bases de données, machines LISP pour l'intelligence artificielle, etc... [BER 80, ROH 80, SAN 81] Au dessus des langages évolués, qui peuvent prétendre à une certaine universalité, on rentre dans le domaine spécifique de chaque type d'application ou de système, avec des besoins propres à chacun.

Un autre aspect important dans l'évolution future de l'utilisation de l'informatique est l'intégration poussée dans les activités d'une collectivité de travail. La disponibilité du système devient alors essentielle. Le très grand nombre d'utilisateurs connectés en permanence, souvent pour des activités très modestes (saisie de textes, par exemple), fera qu'il sera difficile de les gérer et de les servir de façon satisfaisante sur un système centralisé et partagé.

Enfin, la nature même des données manipulées va subir d'importantes modifications. Des types de données nouveaux apparaissent: sons, images, textes... Leur taille pourra être très grande (une image de télévision représente plus d'un mégaoctets). Les traitements effectués sur ces données pourront être très complexes.

I.2. EVOLUTION DE LA TECHNOLOGIE:

Les 15 dernières années ont vu des progrès fantastiques des technologies liées à l'informatique.

Les microprocesseurs permettent maintenant d'aborder les applications les plus complexes. On peut s'attendre, dans l'avenir, à voir apparaître des processeurs encore plus puissants, et mieux adaptés à certains types de traitements (Lisp, Prolog). Avec les progrès des mémoires, on peut maintenant disposer dans un ordinateur de bureau de volumes de mémoire qui étaient ceux de grosses machines il y a une quinzaine d'années. Ceci vient conforter l'utilisation des langages et techniques de l'intelligence artificielle, gros consommateurs de mémoire. Les bases de données bénéficient également de ce progrès, puisqu'on peut maintenant stocker en mémoire primaire des volumes d'information qui n'étaient autrefois possible que sur disque magnétique.

Les progrès des outils et méthodes de conception des circuits intégrés permettent de réduire fortement le coût du développement d'un circuit. Il en résulte que l'on peut maintenant envisager la conception de circuits très spécialisés.

L'évolution est similaire dans le domaine des périphériques, avec souvent des conséquences importantes: ainsi, la quasi disparition des terminaux papiers, et le développement des écrans graphiques (ou non graphiques) entraînent une augmentation des échanges de données dans le dialogue homme machine, de par la volatilité du support visuel de l'information, et de par l'augmentation de la rapidité de ces échanges.

Les mémoires secondaires connaissent également un développement rapide, avec une miniaturisation et une diminution des coûts des disques magnétiques, qui permettent d'en équiper les ordinateurs les plus simples. Par contre, les progrès dans le domaine des performances semblent beaucoup plus lents. L'émergence de technologies concurrentes au disque magnétique ne semble pas encore se préciser: les mémoires à bulles semblent stagner et se cantonner dans des domaines d'application bien précis (systèmes portables ou embarqués), et le disque optique numérique semble encore rencontrer des difficultés.

I.3. DISTRIBUTION FONCTIONNELLE

Les perspectives d'évolution des systèmes informatiques au cours des prochaines années amènent à remettre en question la structure même des systèmes.

On peut distinguer trois directions:

- Poursuivre l'évolution vers des machines partagées de plus en plus puissantes, éventuellement assistées de processeurs "frontaux" ou "dorsaux" pour exécuter des fonctions généralement très limitées et de bas niveau (gestion physique de périphériques, par exemple).
- Distribuer le traitement entre un certain nombre de machines, éventuellement mini ou microordinateurs. C'est l'approche réseaux locaux.

- Distribuer le traitement entre un certain nombre de processeurs spécialisés. C'est l'approche distribution fonctionnelle, que nous développerons dans la suite.

La première solution reste la plus couramment envisagée. Cela est dû au fait qu'il reste plus économique et aussi efficace de réaliser une fonction par programmation sur une machine puissante et universelle, que de l'exécuter sur une machine spécialisée.

La seconde solution est issue des travaux effectués sur les réseaux de téléinformatique, dont ils reprennent un certain nombre de résultats. Elle consiste à interconnecter un certain nombre d'ordinateurs, mono ou multiutilisateurs, plus éventuellement quelques serveurs effectuant des fonctions particulières. Les relations entre les machines sont relativement simples: transferts de fichiers, échanges de courrier, soumission de travaux a distance, logins distants. L'utilisation des réseaux locaux actuels correspond le plus souvent à cette approche.

La distribution fonctionnelle consiste à répartir le traitement entre un certain nombre de processeurs spécialisés dans une fonction bien précise: on trouve alors une véritable relation de coopération entre les machines, non limitée à des échanges de fichiers, ou des soumissions de travaux ponctuels. Les couches application et interface utilisateur elles mêmes sont réparties entre un certain nombre de postes de travail individuels qui doivent être aussi économiques et fiables que possible. Avec l'évolution de la technologie, on peut espérer disposer ainsi d'une puissance de calcul largement suffisante pour la plupart des applications. La distribution fonctionnelle n'implique pas nécessairement une distribution géographique des machines: celles ci peuvent être regroupées et reliées, par exemple, par un bus rapide. Peu d'applications de ce type semblent exister à l'heure actuelle, si ce n'est, dans une certaine mesure, les systèmes de fichiers répartis tels que NFS (SUN Microsystèmes).

Un système peut ainsi être réalisé par interconnexion des éléments nécessaires à l'application, ces éléments étant interchangeables, pour permettre au système d'évoluer en fonction des besoins. Le but essentiel de la distribution fonctionnelle est de permettre une intégration maximale des fonctions au niveau du matériel, et pas nécessairement de répartir géographiquement les moyens de traitement.

I.4. PROLOG:

Défini en 1975 à l'université d'Aix-Marseille par l'équipe de A. Colmerauer, le langage PROLOG est un langage non procédural basé sur la logique des prédicats du premier ordre [ROB 65, KOW 74]. On en donne une présentation détaillée dans les annexes 1 (langage) et 2 (méthode d'interprétation).

Un certain nombre de raisons nous amènent à nous intéresser à ce langage, et à s'y référer au cours de cette thèse:

- son intérêt dans le domaine en expansion de l'intelligence artificielle, grâce à son aspect déductif.

- le fait qu'il est basé sur un formalisme théorique, plutôt que sur un modèle de machine pré-existant. Ceci implique qu'il est difficile de l'interpréter de façon efficace, difficulté qui pourrait vraisemblablement être levée grâce à des machines mieux adaptées.
- le fait qu'il unifie les notions de programmation et de bases de données: il devrait en particulier être très adapté à la représentation et au stockage des connaissances.

PROLOG nous apparaît donc comme le prototype d'un nouveau style de programmation, et à ce titre, son impact sur l'architecture des calculateurs devrait être important.

Pour des informaticiens expérimentés, PROLOG apparaît très difficile à apprendre. On peut trouver à cela deux raisons: d'une part une syntaxe sans doute encore un peu rustique, et d'autre part la distance assez grande entre PROLOG et les langages de programmation procéduraux classiques. Un certain nombre d'hybrides de LISP et de PROLOG ont donc vu le jour, pour rapprocher la programmation en logique des modèles de programmation plus courants. Mais il conviendrait aussi de considérer des expériences d'apprentissage de la programmation en logique à des non informaticiens: en effet, les informaticiens ont souvent une tendance assez forte à se référer aux styles de programmation qui leur sont familiers, ce qui constitue une perte de temps pour l'apprentissage de PROLOG.

I.5. OBJET DE LA THESE:

Dans cette introduction, nous avons voulu montrer que de nouveaux problèmes posés par l'utilisation des systèmes informatiques appellent la recherche de nouvelles solutions. Nous avons indiqué quelques approches, de façon non exhaustive.

Cette thèse est le résultat d'une démarche scientifique dont le fil conducteur est l'idée que, plutôt que de concevoir des machines universelles, il peut être intéressant d'interconnecter et de faire coopérer des machines spécialisées. Cette idée nous a amené d'abord à une réflexion générale sur la notion de système fonctionnellement réparti, qui fera l'objet du prochain chapitre dont l'objet essentiel est de placer la thèse dans son contexte général.

L'émergence de Prolog et de l'intelligence artificielle nous a ensuite amené à nous intéresser à son utilisation à divers niveaux des systèmes, et plus particulièrement à celui des bases de données. Ceci a donné lieu au Projet Opale [GBS 82], qui fera l'objet des chapitres suivants. Ce projet vise en effet à la conception d'une machine bases de données orientée vers le langage Prolog. Deux problèmes importants sont mis en relief: la définition d'une stratégie de recherche adaptée au cas des bases de données, et l'implémentation de l'unification: ces problèmes sont en fait liés, dans la mesure où on va s'attacher à traiter des ensembles de solutions au niveau de l'unification. Ces deux points constituent le coeur de la thèse.

Enfin, nous avons cherché à généraliser l'étude de l'interprétation de Prolog: ceci fait l'objet du dernier chapitre, où nous discuterons, à la lumière d'évaluations, de quelques approches pour la conception de machines Prolog.

Chapitre II

SYSTEMES FONCTIONNELLEMENT DISTRIBUES

II.1. INTRODUCTION:

Comme nous l'avons établi au chapitre précédent, cette thèse a pour objet général la conception de systèmes à unités spécialisées. Avant d'aborder le domaine des machines bases de données qui sera plus particulièrement approfondi, l'objet de ce chapitre est de discuter de l'intérêt de cette approche, et d'un certain nombre de problèmes qu'elle pose. Ceci passe par une critique des notions de machine universelle et de langage universel, que nous amorcerons dans la prochaine section, à partir de quelques exemples. Nous verrons ensuite un certain nombre de problèmes liés aux systèmes fonctionnellement répartis.

L'intérêt des processeurs spécialisés semble évident en termes de performances. Pourtant, relativement peu de machines spécialisées ont vu le jour à l'heure actuelle, malgré un assez grand nombre de projets de recherche.

Une raison peut être le fait que beaucoup de problèmes restent à résoudre pour la conception de machines spécialisées, y compris dans le domaine très avancé des machines bases de données. Une autre difficulté peut concerner la compatibilité avec les systèmes préexistants.

On peut aussi se poser la question de l'efficacité d'une machine dédiée à une seule fonction, alors qu'une machine universelle pourra avoir un taux d'utilisation plus élevé, du fait même du partage entre divers types d'activité. La distribution va de plus entraîner un morcellement des mémoires, et une charge de communication qui diminuent le rendement du matériel.

Enfin, beaucoup de problèmes restent ouverts au niveau de la communication et de la coopération entre machines.

II.2. LIMITES DES MACHINES UNIVERSELLES:

II.2.1. les langages machine:

On peut représenter un système informatique et son activité sous forme d'un secteur (fig. II.1.a.): l'angle du secteur représente la variété des applications, et le rayon représente le niveau des applications. La pointe du secteur (partie hachurée) est le langage de la machine. On peut considérer que le logiciel réalise une projection du niveau application sur le niveau langage machine (fig. II.1.b).

Pour limiter le fossé entre le matériel et les applications, les constructeurs proposent des machines dont les instructions deviennent très puissantes et variées. On rencontre alors le problème suivant: plus on élève le niveau du langage machine, plus il devient difficile de rendre générales les instructions (risque de figer les structures de données). Le nombre des instructions complique généralement leur décodage, et leur complexité même

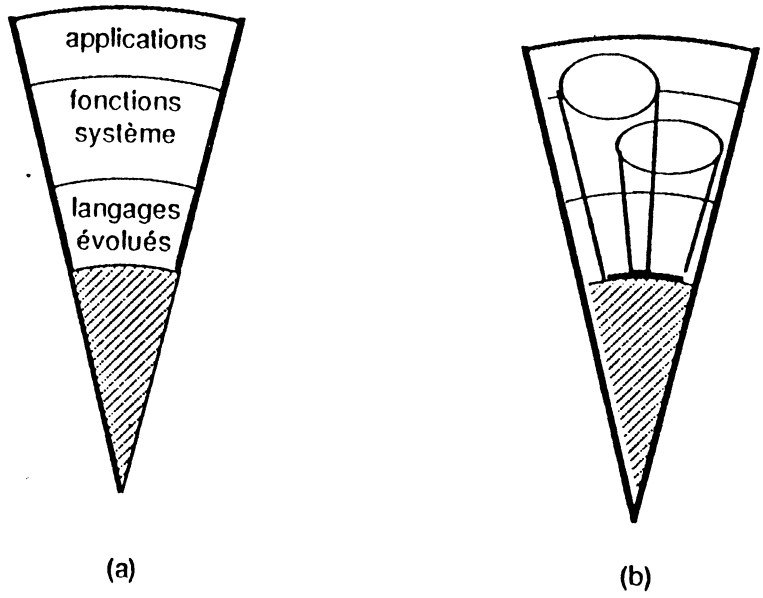


Figure II.1.

est un facteur de coût.

Une façon d'augmenter le niveau des langages machine à coût constant consiste donc à spécialiser les machines (fig. II.2.).

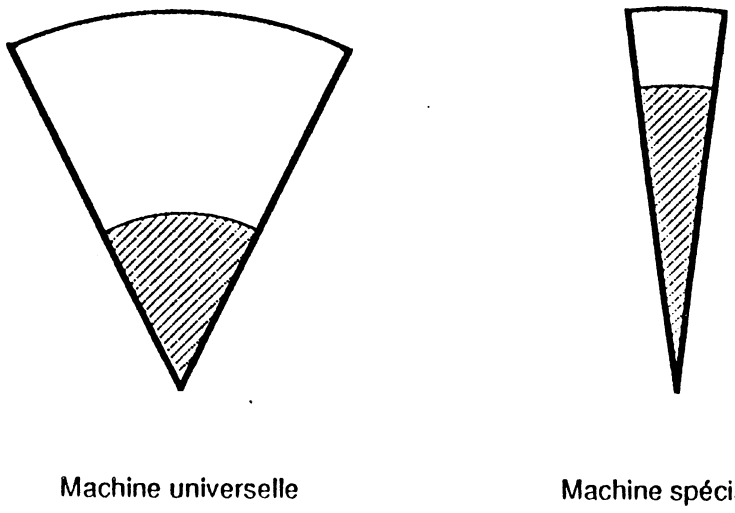


Figure II.2.

II.2.1.1. Exemple: le cas du VAX:

On trouve dans deux communications [WIE 82, CLA 82] des statistiques sur l'utilisation des instructions du VAX de DEC:

- 64,6% du traitement est effectué par 20 opérations (sur 304),
- 60,5% des opérations sont utilisées au moins une fois,
- sur 18 classes d'instructions, 5 ne représentent ensemble que 1,09% des instructions utilisées,
- 87,3% des instructions exécutées font au plus 6 octets (sur un maximum de 32),
- 96% des instructions exécutées ont au plus 3 opérandes sur un maximum de 6.

On peut considérer que le nombre d'opérandes d'une instruction et sa longueur forment une mesure empirique de sa complexité, ce qui confirme l'idée que les instructions les plus complexes sont souvent les moins utilisées. Cependant, les instructions sont groupées par familles d'instructions similaires utilisant des ressources physiques communes (format d'opération, types d'opérandes...). C'est donc souvent la famille qu'il faut considérer plutôt qu'une opération isolée.

Si on regroupe dans un tableau l'ensemble des mesures fournies dans les articles référencés (annexe 4), on trouve 82 instructions au total parmi les 20 instructions les plus utilisées de chaque jeu d'essai (en terme de temps d'exécution). Parmi ces instructions, 41 ne sont utilisées que par un seul jeu d'essais, avec souvent des taux d'utilisation non négligeables. A quelques exceptions près, les instructions de plus de trois opérandes sont peu utilisées, ou ne concernent qu'un jeu d'essais.

Ceci confirme donc l'idée qu'une bonne part des possibilités d'une machine universelle est sous employée, même si certaines instructions peu utilisées peuvent avoir tout de même une certaine importance (cas des instructions REMQUE et INSQUE pour le traitement des files d'attente).

II.2.1.2. Recherches de sous chaînes:

La recherche de sous-chaînes dans un texte est un exemple d'un type de traitement assez répandu, qu'on peut considérer comme le type même du traitement non numérique. L'algorithme le plus simple et le plus évident est l'algorithme "naïf". Programmé en assembleur sur une machine telle que le MC 68000, cet algorithme prend en moyenne une dizaine d'instructions, soit une quinzaine d'accès mémoire, par octet comparé.

Sur le VAX, il existe l'instruction MATCH, qui exécute cette recherche. C'est une opération assez complexe, qui demande 4 arguments et modifie 4 registres au retour. Si une partie suffisante de la chaîne à rechercher se trouve dans un registre, elle devrait entraîner environ 1 accès mémoire par octet traité (sans doute moins, la mémoire étant accédée par mots de 32 bits), soit un gain d'un rapport 15 environ.

Dans les statistiques précédemment mentionnées, cette instruction n'apparaît pas. On peut donc considérer que son utilisation est assez marginale pour les applications considérées. Par ailleurs, cette instruction n'est pas applicable au cas de recherches de plusieurs sous chaînes. Une instruction sophistiquée peut donc manquer de généralité, et être trop peu utilisée pour justifier les ressources matérielles nécessaires à sa réalisation, du point de vue des performances de la machine.

Par contre, il existe d'autres algorithmes logiciels [KNU 77, BOY 77] qui permettent de meilleures performances et peuvent être étendus au cas de recherches plus complexes.

L'implémentation matérielle de tels algorithmes a été largement étudiée et appliquée au cas des machines bases de données [ROH 80]. Par contre, dans le cas de machines universelles, ils sont trop complexes pour faire l'objet d'implémentations matérielles, compte tenu de leur utilisation.

II.2.1.3. Machines RISC:

La constatation que les instructions les plus complexes sont souvent sous utilisées a donné le jour aux machines RISC [PAT 81,PAT 85]. Les machines RISC (Reduced Instruction Set Computers - Ordinateurs à jeu d'instructions réduit) ont un jeu d'instructions assez rudimentaire, qui se situe à mi-chemin entre les langages d'assemblage courants et la microprogrammation: on part en effet de l'idée que les machines sont maintenant utilisées par l'intermédiaire de compilateurs qui savent très mal utiliser le jeu d'opérations des machines. Utiliser de façon efficace un jeu d'instructions simple est donc un premier principe. Le second principe consiste à exécuter le plus efficacement possible ce jeu d'instructions, par l'intermédiaire de structures matérielles sophistiquées, rendues possibles par la simplicité même du langage: en particulier, le fait de se passer de mémoire de microprogrammes (la partie contrôle de la machine étant câblée) permet, à volume de ressources matérielles équivalent, de disposer de caches importants.

Cette conception a donné le jour à des machines performantes, par rapport à leur coût. Les programmes sont cependant plus encombrants. En fait, on peut assimiler la démarche des machines RISC par rapport aux machines "classiques" à la compilation par rapport à l'interprétation: les machines classiques interprètent par microprogramme un langage complexe, alors que les machines RISC préfèrent pousser plus loin le processus de traduction logicielle.

Ce type de machine semble fournir une alternative à la spécialisation matérielle des machines (tout au moins la spécialisation par microprogrammation). Il semble cependant qu'elles posent un certain nombre de problèmes: en particulier, leurs performances reposent de façon assez critique sur l'efficacité du cache. Or, cette efficacité peut se trouver compromise dans certains types d'applications, ou dans certains cas d'utilisation: programmes possédant peu de localité, ou multiprogrammation amenant des commutations de processus fréquentes.

Enfin, il resterait, pour certaines applications, à comparer les machines RISC avec des machines langage (machines LISP), ou des machines spécialisées dans une fonction (machines bases de données).

II.2.2. Les langages de haut niveau:

Dans la précédente section, nous avons vu les problèmes posés par l'utilisation des machines universelles, et en particulier, la difficulté qu'il y a à augmenter le niveau des instructions d'une machine sans en réduire la généralité - donc sans la spécialiser.

Les machines langage peuvent alors fournir une autre façon d'aborder le problème. La question est alors: existe-t'il un langage réellement universel, et si oui, existe-t'il une

façon universelle de l'interpréter?.

La plupart des langages de programmation prétendent à l'universalité. Cependant, chaque langage est plus ou moins bien adapté à certains types de problèmes, ou certaines méthodes de travail... Le langage a tout faire, est, à notre avis, un mythe: tout au plus peut on espérer arriver à un nombre limité de langages appartenant à des classes de langages orthogonales (langages algorithmiques, fonctionnels, logiques...), le choix entre les langages étant affaire de goût, ou d'adaptation à l'application.

Une machine langage n'est donc vraisemblablement pas universelle, et, par exemple, une machine Prolog se trouvera pénalisée si elle a à exécuter des calculs qui relèveraient plutôt d'un langage tel que Fortran: or, les calculs numériques ne sont pas forcément rares dans les applications de l'intelligence artificielle. De plus, les langages tels que LISP ou PROLOG font souvent appel à un grand nombre de fonctions (ou prédicats) prédéfinis, ce qui revient souvent à programmer certains types d'opérations dans le langage d'implémentation de l'interpréteur.

Enfin, la méthode d'interprétation d'un langage elle même peut différer d'une application à l'autre: dans les prochains chapitres, nous en verrons un exemple avec l'utilisation de Prolog pour les bases de données.

II.2.3. Le coût du partage:

La notion de machine universelle est liée à la notion de partage de la machine entre plusieurs utilisateurs. Ceci amène la notion de protection, qui nécessite un minimum d'outils matériels: protection mémoire, protection des périphériques, mode de fonctionnement privilégié, contrôle des appels au système.

Le problème de la communication et du partage des données doit être résolu par le système d'exploitation, ou par un nouvel ensemble de moyens matériels, ce qui augmente la complexité des systèmes. De plus, la charge de gestion de la machine contribue à en réduire l'efficacité.

Un certain nombre de problèmes demeurent dans le cadre d'un poste de travail mono-utilisateur multitâches, mais le taux de partage de la machine est alors plus faible, et il y a partage entre une tâche de premier plan, prioritaire, et une ou plusieurs tâches d'arrière plan. La tâche de premier plan ne passera la main aux autres qu'à l'occasion d'entrées sorties, ou des périodes d'inactivité (délais de réflexion de l'utilisateur interactif). Enfin, la protection vis à vis des autres utilisateurs n'intervient qu'au niveau des échanges avec d'autres machines, et il suffit de protéger l'utilisateur contre ses propres erreurs, ce qui peut être plus simple.

II.2.4. Evolution des puissances de traitement:

Pour répondre à la demande d'une puissance de traitement accrue, l'augmentation de la puissance des machines peut suivre deux axes:

- augmentation de la rapidité d'un jeu d'instructions classique en utilisant des techniques de pipe-line, de cache, en augmentant la rapidité des circuits, etc... Une telle augmentation reste limitée par l'efficacité des caches et les accès à la mémoire, et les limitations de la technologie. On peut s'attendre à ce qu'il existe une limite au delà de laquelle l'augmentation du coût des machines est démesurée par rapport à l'amélioration des performances.
- augmentation de la puissance des instructions, mais nous avons vu que des instructions sophistiquées risquent d'être sous utilisées.

L'évolution de la gamme des microprocesseurs de Motorola peut être considérée comme significative (d'après les estimations du fabricant): Entre le MC6800 et le MC6809, le rapport de complexité est estimé à 3 et le rapport de performances est estimé à 3,5. Entre le MC6809 et le MC68000, le rapport de complexité est estimé à 5, et le rapport de performances à 3. Entre ces machines, les différences portent sur la capacité opérative (de 8 à 32 bits), l'adressage, le nombre de registres, et enfin quelques instructions (multiplication, division, ...).

Le 68020 permet le gain d'un nouveau rapport 3 par rapport au 68000 en passant la capacité opérative a 32 bits au lieu de 16, en utilisant un cache d'instructions, et en augmentant la vitesse de l'horloge. De nouveaux gains sont possibles par amélioration du cache, en particulier, mais ils sont limités (68030).

Il est intéressant de constater que le MC 68000 se situe assez près de machines beaucoup plus grosses. Ainsi, d'après nos évaluations effectuées sur des programmes en Pascal, le rapport des temps d'exécution CPU entre le MC68000 (Exormacs sous Versados) et le CII-HB 68 (sous Multics) serait de 5 pour des opérations non numériques et de 15 pour des opérations arithmétiques sur 32 bits. Ce rapport de performances est confirmé si on considère non des jeux d'essai mais des applications. D'autre part, un interpréteur Prolog écrit en C sous UNIX (*) montre un rapport de temps d'exécution CPU de 2 entre une SM 90 (MC 68000) et un VAX 780. Ces rapports de performance sont faibles si on les compare aux rapports de complexité (et de coût) des processeurs.

II.3. PROCESSEURS SPECIALISES:

L'approche processeurs spécialisés va consister à concentrer des ressources matérielles sur un ensemble limité de problèmes spécifiques. On espère ainsi obtenir de meilleures performances, par une utilisation plus intensive de ces ressources.

On distingue plusieurs voies pour la conception de processeurs spécialisés:

- microprogrammation de processeurs classiques: ceci a été utilisé par exemple dans le projet EGPA [BOD 81], où des possibilités de traitement associatif sont ajoutées à un processeur préexistant. L'utilisation de processeurs en tranches microprogrammables est également une option assez fréquente.

(*) Unix est - comme chacun sait - une marque déposée des laboratoires BELL.

- conception de toutes pièces de processeurs complets: cette voie est peu abordée pour les traitements non numériques, en raison du coût et de la complexité d'une telle étude.
- utilisation de microprocesseurs et d'opérateurs matériels spécifiques. Le microprocesseur permet d'exécuter des fonctions de bas niveau, ainsi que la gestion et le séquencement de l'ensemble de la machine. Les opérateurs spécialisés se présentent alors comme des circuits périphériques du microprocesseur. En reprenant l'analogie du secteur présentée en II.2.1.1, on a donc la structure présentée sur la figure II.3.

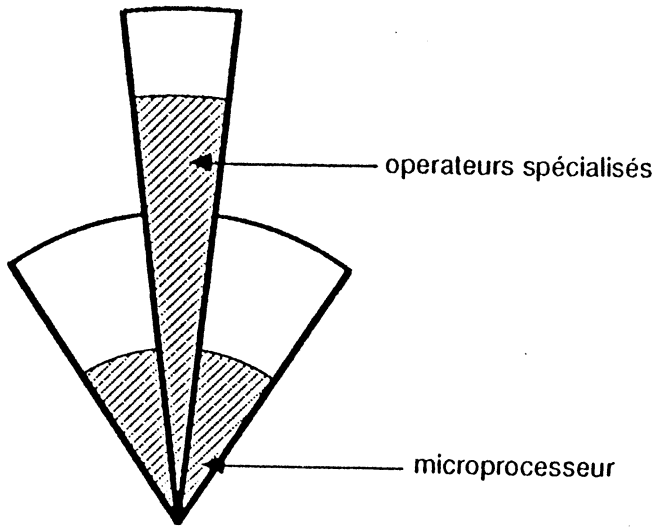


Figure II.3.

Les avantages attendus des processeurs spécialisés sont multiples.

- *performances:* L'adaptation du matériel est un premier facteur de performances. Un deuxième point est l'adaptation du logiciel de base, et en particulier de la gestion de la machine, à la fonction à exécuter. On peut s'attacher à optimiser, non l'utilisation globale de la machine, mais l'utilisation des ressources lentes ou coûteuses, comme par exemple les disques dans une machine bases de données.
- *sécurité:* L'isolement de la fonction sur une machine spécialisée est un facteur de sécurité important. La programmation n'en est pas accessible à l'utilisateur, et le logiciel peut être éventuellement stocké en PROM. Le contrôle et la protection peuvent être assurés efficacement et simplement par logiciel. Ce point est toutefois lié plus à la séparation physique des unités qu'à leur spécialisation matérielle.
- *coût:* On peut espérer qu'un matériel spécialisé sera plus simple, à performances équivalentes, qu'une machine universelle, et que la technologie permette le développement à coût réduit d'opérateurs spécialisés.

- *compatibilité:* Une machine spécialisée bien conçue devrait être connectable à n'importe quel système, moyennant éventuellement l'écriture d'un logiciel d'interface. Pour bien des fonctions, l'écriture d'un tel logiciel devrait être sensiblement moins coûteuse que le transport complet d'une fonction logicielle.

La gestion de la machine peut être assurée par la fonction elle même, et non constituer une couche logicielle séparée, ceci dans la mesure où l'efficacité s'en trouve améliorée. Ainsi, un processus peut exécuter des entrées sorties sans contrôle, ou décider lui même de l'instant où il passe la main à un autre processus, ou effectuer lui même les allocations de mémoire primaire. Le contrôle peut donc être assoupli et rendu plus efficace, dans la mesure où les transitions entre processus sont plus rares et demandent moins de contrôles.

II.4. DISTRIBUTION FONCTIONNELLE

Un système fonctionnellement réparti sera, pour nous, constitué de deux types de processeurs [POU 76]:

- des processeurs utilisateurs, ou postes de travail, permettant l'exécution des travaux utilisateurs,
- des processeurs fonctionnels, spécialisés dans l'exécution de fonctions telles que la gestion de bases de données, la gestion de périphériques d'impression, etc...

L'ensemble de ces processeurs sera relié par l'intermédiaire d'un réseau de communication. La communication entre les stations, et la répartition des activités sera basée sur la notion de *fonction*.

Une telle conception est susceptible d'apporter un certain nombre d'avantages:

- Modularité du système, dont chaque élément peut être adapté aux besoins indépendamment des autres.
- Possibilité de rapprocher certaines ressources de leurs utilisateurs (imprimantes par exemple).
- Permettre une certaine tolérance aux pannes, modulable en fonction des besoins et des moyens des utilisateurs, par redondance d'éléments, ou adoption d'éléments tolérants aux pannes.
- Offrir des performances élevées, en ne mettant la puissance de traitement que là où elle est réellement nécessaire, et en l'utilisant au mieux, et par exploitation du parallélisme.

La contrepartie de ces avantages est évidemment les problèmes liés au contrôle d'un tel système réparti: ce point sera abordé brièvement dans la suite.

II.4.1. Postes de travail:

Les postes de travail sont les points d'accès au système pour les utilisateurs. Ils pourront exécuter localement un certains nombre de travaux: bureautique, messagerie, saisies de données, etc... Ils seront également les frontaux pour l'accès aux processeurs

fonctionnels: prétraitement des travaux, traitement des résultats, etc... Ils pourront enfin établir de véritables dialogues avec des processeurs fonctionnels, par exemple pour l'accès à des bases de données.

La technologie permet de disposer de configurations "minimales" assez importantes: un processeur 16 bits et 1 Mégaoctets de mémoire tiennent sur une seule carte. Un tel poste de travail peut être guère plus coûteux qu'un terminal.

Un poste de travail peut être dépourvu de mémoire secondaire, en particulier pour des raisons de coût. La mémoire secondaire est aussi une contrainte pour l'utilisateur qui doit veiller lui même (et généralement mal) à la sauvegarde de ses informations, et à la manipulation des disques. Enfin, c'est le plus souvent un point faible de la machine.

Sur le plan des performances, il convient de comparer les performances des unités de mémoire secondaire avec celles des réseaux locaux (tableau II.1.). Le tableau II.2. présente les temps de transfert (en millisecondes) pour des longueurs de bloc de 256 à 64 K octets. On considère des minidisques durs avec déplacement de bras par moteur pas à pas, ce type d'unité étant le plus économique semble le plus approprié pour des microordinateurs mono-utilisateurs. Pour les mémoires à bulles, on considère l'utilisation parallèle de 8 circuits de 1 Mbit. Enfin, on considère un réseau local de type ETHERNET supposé relativement peu chargé. Dans de telles conditions, on peut négliger les temps d'attente en émission.

type d'unité	débit	temps d'accès	capacité
disques souples	0,5 Mb/s	200 ms	1 Mo
mini disques durs	10 Mb/s	80 ms	10 Mo
mémoires à bulles	0,8 Mb/s	5 ms	1 Mo
gros disques durs	20 Mb/s	20 ms	1000 Mo
réseau local	10 Mb/s	-	-

Tableau II.1: performances des unités de disque

	volume de données (en octets)				
Type d'unité	256	1K	4K	16K	64K
disques souples	284	296	345	542	1648
mini disques durs	90	91	94	106	154
mémoires à bulles	7,5	15	45	165	645
gros disque dur	25	25,5	27	33	57
réseau local	0,25	1	4	16	64

Tableau II.2: temps d'accès aux blocs d'information (en millisecondes)

On voit sur ce tableau que les mémoires secondaires utilisables sur des microordinateurs sont au mieux égales en débit à ce que l'on peut attendre d'un serveur fichier

performant accédé par le biais d'un réseau local. De fait, sur SM90, on s'aperçoit que l'accès à un fichier distant par l'intermédiaire du réseau (peu chargé) est très peu ralenti par rapport à l'accès à un fichier local. En fait, les performances de la mémoire secondaire sur micro-ordinateur dépendent fortement de celles du coupleur, les performances du disque étant généralement sous utilisées.

Enfin, le développement des capacités des mémoires (vives et mortes) permet de diminuer beaucoup le besoin d'accès aux mémoires secondaires: en particulier, il est possible de conserver en mémoire primaire tout un noyau système, une bibliothèque de sous programmes, et quelques utilitaires de base.

Cette conception a déjà trouvé une application avec le SUN, poste de travail Unix, dont une version dépourvue de mémoire secondaire peut être utilisée par le biais d'un réseau local Ethernet.

II.4.2. Processeurs fonctionnels:

Un processeur fonctionnel permet l'accès à des ressources qui, de par leur nature, leurs performances, ou leur coût doivent être partagées. Ces ressources peuvent être:

- de la mémoire secondaire, permettant, en particulier, le partage des informations.
- des imprimantes: l'utilisation d'une imprimante rapide mais partagée est plus efficace, pour de grosses listes, qu'une imprimante non partagée mais plus lente. Un processeur d'impression devrait être doté de mémoire secondaire pour assurer le spooling.
- des ports de communication avec des réseaux de téléinformatique.
- des moyens de traitement spécifiques de certaines classes de problèmes, etc...

II.4.3. Exemple:

Supposons un système sur lequel une application importante soit un système expert communiquant en langage naturel avec l'utilisateur. Cette application peut etre structurée autour de 3 modules essentiels: l'analyse des phrases rentrées par l'utilisateur, le système expert proprement dit, et la synthèse des réponses du système expert. Le système expert à son tour va comporter le moteur d'inférences, la base de faits, la base de connaissances, et éventuellement un module de calcul numérique si le domaine d'expertise le nécessite. L'analyse va accéder a un dictionnaire et aux règles de grammaire du langage, et sans doute aussi à la base de faits et à la base de connaissances. La synthèse des réponses accèdera elle aussi au dictionnaire et aux règles de grammaire.

L'ensemble de ces fonctions peut être exécuté par une seule machine suffisamment puissante, ou encore, dans l'approche fonctionnellement distribuée, par un ensemble de processeurs fonctionnels et par le poste de travail de l'utilisateur. Par exemple:

- le poste de travail va exécuter l'analyse des phrases rentrées par l'utilisateur, et la synthèse des réponses en langage naturel. Ces deux étapes étant en principe séparées par l'exécution du moteur d'inférence, il y a peu de parallélisme entre elles, et il n'y a donc pas d'inconvénient à les exécuter sur un même processeur. L'analyse du texte pourra se faire éventuellement en parallèle avec la saisie, et les informations peuvent

être transmises au moteur d'inférence au fure et à mesure de leur obtention.

- Le moteur d'inférences peut être exécuté sur une machine spécialisée (machine à inférence).
- Le dictionnaire, les règles de grammaire, la base de faits et la base de connaissances peuvent être gérés par une machine spécialisée.
- Enfin, le calcul numérique peut être effectué lui aussi sur une unité spécialisée (calcul vectoriel, par exemple).

Nous avons donc là un exemple de traitement qui peut être réparti sur 4 machines. Selon l'importance de chaque type de traitement, on peut choisir de recourir ou non à un processeur spécialisé.

II.5. UTILISATION DES RESSOURCES:

Dans les systèmes classiques, le partage permet l'utilisation optimale des ressources, et la communication entre les utilisateurs. Ceci permet de mettre à la disposition des utilisateurs les ressources nécessaires au traitement efficace des plus gros problèmes, et d'en assurer une utilisation correcte.

Le partage d'informations et la communication sont assurés dans les systèmes répartis par les réseaux locaux, et par l'utilisation de stations fonctionnelles. Cependant, cette approche ne permet pas une utilisation optimale des ressources. Cette utilisation devient peu critique en ce qui concerne les processeurs et la mémoire primaire, vu le coût réduit des micro-ordinateurs, le problème est plus complexe en ce qui concerne la mémoire secondaire.

II.5.1. Utilisation de la mémoire secondaire:

Un premier motif pour le partage des mémoires secondaires est le partage d'informations entre plusieurs utilisateurs. Il en existe d'autres:

- coût au bit plus réduit, par le partage de grosses unités de mémoire secondaire,
- utilisation optimale de l'espace disque.
- maintenance simplifiée, par regroupement de grosses unités de disques.

Ces raisons ont perdu de leur importance avec l'évolution de la technologie. Par contre, dans les systèmes centralisés, la mémoire secondaire dispose généralement de peu d'intelligence, et le parallélisme entre les unités est généralement loin d'être exploité de manière optimale.

Les unités de disques actuelles ont un débit de 10 Mbits/s. Ceci représente à peu près un accès mémoire par microseconde, soit une charge importante sur la mémoire primaire. Compte tenu du partage de cette mémoire avec le ou les processeurs et d'autres canaux d'entrées sorties, très peu de disques peuvent lire ou écrire en parallèle (généralement un seul). Le seul parallélisme que l'on rencontre en général dans les systèmes centralisés consiste à avoir plusieurs disques qui effectuent des déplacements de bras, alors qu'un seul transfère des données.

Dans beaucoup de systèmes, cela a peu d'importance, car les disques sont accédés par secteurs de taille relativement courte: 256 à 1024 octets. Sur un accès disque, on va avoir 256 à 1024 µs de transfert, alors que les déplacements de bras peuvent durer en moyenne 25 millisecondes, et l'attente rotationnelle environ 8 millisecondes. Les disques transfèrent donc en moyenne pendant moins de 1% du temps, et l'intérêt de transferts parallèles est négligeable.

Cependant, il est souvent intéressant d'enchaîner des transferts sur plus d'un secteur: recherches dans les bases de données relationnelles, traitements de textes, chargements de programmes, etc... Considérons par exemple la lecture d'une piste de disque: le transfert dure 16 millisecondes, pour 33 millisecondes d'attente, soit près de 33% du temps. On voit alors que, si ce type d'opération est fréquent, il y a intérêt à avoir un parallélisme maximal entre les unités de disque.

L'approche système réparti permet en grande partie un tel parallélisme, puisqu'on peut dédier un processeur et une mémoire privée à chaque unité de disque. Le traitement effectué au niveau du processeur fonctionnel devrait limiter le débit des informations; cependant il existe un risque de goulot d'étranglement entre une station qui comporte un ou plusieurs disques qui débitent à 10 Mbits/s ou plus, et un réseau qui débite en principe à 10 Mbits/s. De plus, le débit effectif obtenu au niveau des coupleurs Ethernet est souvent plus réduit: de l'ordre de quelques centaines de Kbits par seconde. L'objet des prochains paragraphes sera d'évaluer ce risque dans le cas de quelques fonctions et applications courantes.

II.5.1.1. bases de données:

Les bases de données sont accédées par l'intermédiaire d'une machine spécialisée chargée d'exécuter une sélection des données à transmettre. Cette machine assure donc une réduction du flux de données entre le disque et le réseau.

Pour accéder à un ensemble de données, il faudra exécuter plusieurs accès disque, et extraire des données lues un certain taux d'informations utiles. Le facteur de sélectivité d'une requête est donc le taux:

$$\frac{\textit{information lue}}{\textit{information utile}}$$

Pour des BD relationnelles, on considère généralement des facteurs de sélectivité de 10 à 10000 ce qui autorise des configurations de plusieurs dizaines de disques accédés en parallèle sans risque majeur d'engorgement du réseau.

Certains types d'accès vont cependant entraîner un facteur de sélectivité plus faible, allant jusqu'à 1, mais le débit sera alors limité par la capacité du processeur fonctionnel, le débit du réseau, et surtout la capacité de traitement de l'unité destinataire (impression, visualisation...). De plus, ces informations ne seront pas nécessairement contiguës sur le disque, d'où un morcellement des accès à la mémoire secondaire.

Considérons maintenant la lecture de 20 pistes d'un cylindre: on considère que l'enchaînement de la lecture sur 2 pistes n'est pas instantané (réajustement éventuel de la position des têtes, synchronisation d'horloges,...), et on a donc une rotation d'attente pour chaque piste.

Durée des 20 accès:

déplacement de bras:		25 ms
délai rotationnel:	8,3 + 16,7 x 20 =	342 ms
lectures:	16,7 x 20 =	334 ms
TOTAL:		691 ms

Considérons maintenant 20 accès d'une piste. La durée d'un accès s'évalue comme suit:

déplacement de bras:		25,0 ms
délai rotationnel:		8,3 ms
lecture:		16,7 ms
TOTAL:	50,0 ms x 20 =	1000 ms

On a donc une perte de 46% par rapport à une lecture continue, perte qui est acceptable, compte tenu du fait qu'il s'agit d'un cas relativement exceptionnel.

Cependant, en fonction du type d'unité disque et de coupleur, on peut avoir un enchaînement immédiat de l'accès à deux pistes d'un même cylindre. Dans ce cas, l'accès aux 20 pistes ne durera que: 25 + 8,3 + 16,7 x 20 = 367 ms. On a donc alors un rapport de performances de l'ordre de 3, ce qui est beaucoup moins acceptable.

Il faut aussi remarquer que le traitement demandé à une machine base de données sera souvent plus complexe que la seule sélection, et comprendra souvent des enchaînements d'opérations relationnelles et de tris: on arrive alors à un débit effectif beaucoup plus réduit au niveau du réseau.

II.5.1.2. fichiers de programmes:

La gestion des fichiers de programme pose des problèmes particuliers. Il s'agit en effet du transfert séquentiel d'un gros volume d'information, dont l'utilisateur espère une durée réduite.

La lecture d'un gros programme peut créer un goulot d'étranglement au niveau du système de communication. Il est donc souhaitable de minimiser les transferts, en adoptant plusieurs solutions: chargement à la demande, utilisation de modules ou de sous programmes résidents dans les postes de travail, etc...

Le temps de résidence des programmes interactif étant relativement long, l'intervalle entre deux chargements de programmes par un même utilisateur peut être assez long. Ainsi, sur nos SM90, l'enregistrement des commandes exécutées donne un temps moyen de résidence des programmes de l'ordre de 8 minutes, pour une taille moyenne de 300K (l'utilisation de Lisp et Prolog est importante sur ces machines). Ceci nous donne pour un programme "typique" un débit moyen effectif de l'ordre de 5K bits/s pour le chargement des programmes.

Le chargement des programmes pose donc peu de problèmes de débit global du réseau, mais plutôt des problèmes de répartition de ce débit qui peuvent être limités par des techniques de programmation et de chargement adaptées.

II.5.1.3. compilations:

La compilation n'a pas forcément une très grosse importance sur un système en exploitation où le travail utile consiste à exécuter des applications. Cependant, c'est une fonction fortement liée à la mémoire secondaire par l'importance de ses entrées sorties: lecture du texte source, production du code objet, production du listing, éventuellement fichiers de travail et chargement du compilateur. On peut en cela la considérer comme représentative d'une classe d'applications faisant intervenir de façon relativement équilibrée traitement et entrées sorties.

D'après les statistiques d'un mois de fonctionnement normal d'une SM 90, on arrive à un temps de résidence des programmes impliqués dans la compilation C de 101 minutes, pour un débit d'entrées sorties de 36 Mégaoctets environ, soit un débit moyen de l'ordre de 50K bits/s.

II.5.1.4. conséquences:

Comme nous l'avons vu, les fonctions liées à la mémoire secondaire sont très diverses, mais peu de fonctions ont réellement besoin d'être physiquement proches du disque. Les risques de goulot d'étranglement entre le disque et le système de communication sont réduits, pour autant que le débit du système de communication soit du même ordre de grandeur que le débit des disques.

La répartition de la mémoire secondaire ne répond pas à un réel besoin fonctionnel, et regrouper les unités de mémoire secondaire peut permettre d'en faciliter la maintenance, et d'optimiser les accès et la gestion de l'espace. Pour de petits systèmes, on peut donc regrouper les unités de mémoire secondaire. Lorsque la taille du système augmente, on peut envisager une distribution par fonction (base de données, textes, programmes,...).

Pour des gros systèmes, on peut être amené à utiliser des systèmes de communication à plusieurs niveaux (réseaux en grappes) (Fig. II.4.). Ceci n'est intéressant que dans la mesure où une partie importante du traffic de chaque grappe reste locale à la grappe. Le regroupement de la mémoire secondaire autour de quelques unités fonctionnelles communes a l'ensemble du réseau est évidemment contradictoire avec cela, puisque de nombreux échanges seront alors effectués sur la ou les branches du réseau qui supportent ces unités.

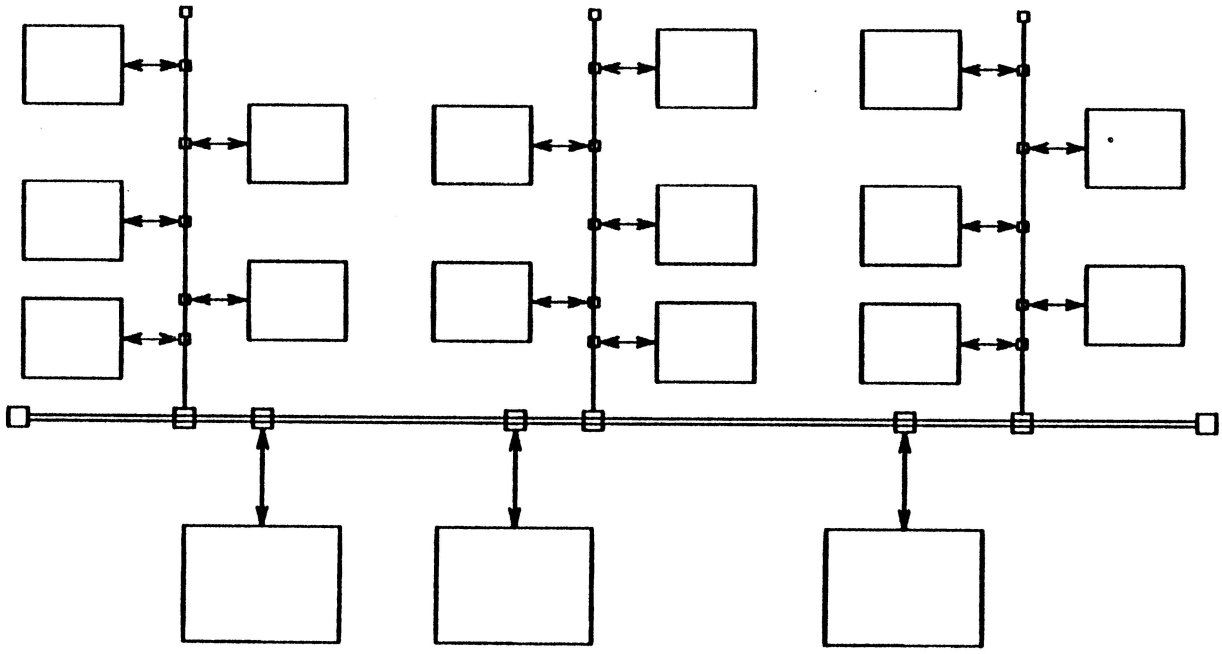


Figure II.4.

On peut alors placer des unités de mémoire secondaire au niveau des grappes, permettant de stocker les fichiers temporaires, les fichiers accédés principalement ou uniquement par les stations connectées à la grappe, et certains programmes partagés, mais fréquemment accédés, dupliqués au niveau de chaque grappe. Les programmes en exploitation étant rarement modifiés, une telle duplication est peu pénalisante.

Dans un système convenablement organisé, une grappe correspondra à un groupe de gens qui travaillent ensemble. Il sera donc possible de maintenir à l'intérieur de la grappe une proportion notable du traffic, et d'éviter une saturation des unités communes à l'ensemble du système.

II.6. SYSTEMES DE COMMUNICATION:

Il est nécessaire de définir un moyen de communication standard entre les composants d'un système fonctionnellement réparti. Certaines applications pourront nécessiter un couplage fort du type bus parallèle rapide, ou mémoire partagée, mais dans la majeure partie des cas, c'est vers les systèmes faiblement couplés du type réseaux locaux qu'il faudra s'orienter.

A l'origine du projet CRESUS [GBS 81], nous avions entamé l'étude d'un système de communication spécifique (réseau en boucle) [GB2 81]. Une autre étude dans ce domaine a également été effectuée dans l'équipe d'architecture des calculateurs de l'IMAG, avec le projet LISA [MAR 78]. Il s'avère aujourd'hui qu'il est illusoire de vouloir rivaliser avec les systèmes développés industriellement, qui sont déjà en compétition pour s'imposer comme norme. Par ailleurs, avec l'augmentation des débits et des distances de transmission, les problèmes sont de moins en moins d'ordre logique,

et de plus en plus d'ordre purement électrique. Ils échappent donc à la compétence des informaticiens, pour être de la compétence d'électroniciens spécialistes des problèmes de transmissions

Dans la suite de cette section, nous nous bornerons à présenter un des systèmes de communication les plus connus: Ethernet [MET 76, SHO 80, SHO 82]. En effet, ce système semble s'imposer dès lors qu'il s'agit d'interconnecter des machines provenant de constructeurs différents (réseaux hétérogènes), et plus particulièrement sous Unix.

Ethernet fonctionne sur le principe "CSMA/CD" (Carrier Sense Multiple Access/ Collision Detection): chaque station "écoute" en permanence le signal émis sur le câble. Lorsqu'elle veut émettre, elle attend qu'aucun signal ne soit présent sur le câble (détection de porteuse), puis commence à émettre. Il peut alors arriver que plusieurs stations commencent à émettre en même temps: ceci est une collision. Dans ces conditions, les émissions se brouillent mutuellement. Pour détecter ces collisions, chaque station compare donc en permanence le signal présent sur le câble avec ce qu'elle émet: si le signal émis diffère du signal reçu, il y a une collision. Chaque station va donc cesser d'émettre, et retentera d'émettre au bout d'un intervalle de temps pseudo-aléatoire.

Ethernet est jusqu'à maintenant utilisé essentiellement pour des tâches du type transfert de fichiers, soumission de travaux à distance, messagerie, ou logins distants. Dans ces conditions, un réseau peut supporter jusqu'à quelques centaines de stations sans problèmes notables. La présence sur le réseau de stations de travail dépourvues de mémoire secondaire (telles que le SUN) effectuant leur pagination via le réseau semble toutefois poser quelques problèmes de débit si ces stations sont nombreuses, mais on ne dispose à ce sujet d'aucunes mesures ou évaluations.

Le format des trames Ethernet comporte 26 octets de descripteurs, à quoi il faut ajouter un délai de 9,6 µs entre deux émissions et un délai minimum de retransmission de 51,2 µs en cas de collision. Il en résulte une diminution de l'efficacité du réseau (taux d'informations utiles transmises rapporté à la bande passante du réseau) dès lors que les messages sont courts, cette diminution étant encore aggravée par les risques de collision. Ceci semble poser peu de problèmes avec les applications classiques, surtout orientées vers le transfert de fichiers, mais en poserait plus avec des applications réparties qui échangeraient de nombreux messages courts.

Sur le plan de la fiabilité, dès que le nombre de machines connectées devient élevé, le risque qu'un transmetteur défectueux empêche le fonctionnement du réseau devient non négligeable.

Ethernet est prévu pour permettre l'interconnexion de plusieurs centaines de machines, sur une distance relativement longue (1,5 Km). A cet effet, il est possible d'intercaler des répéteurs dans le réseau. Un autre choix (moins grande extension, et moins de machines) permettrait de limiter la taille des descripteurs dans les trames, et de limiter le délai inter-messages, ce qui améliorerait l'efficacité pour des applications où les messages sont courts. Ceci est cohérent avec la philosophie de réseaux en grappes présentée précédemment.

II.7. MONITEUR D'EXECUTION REPARTIE

Dans un système fonctionnellement réparti, on appelera *moniteur* (ou encore noyau exécutif) l'ensemble des logiciels présents dans chaque station pour assurer une coopération aussi sûre et efficace que possible avec les autres stations. Dans cette section, on se bornera a des considérations générales sur son rôle, les problèmes qui se posent, et les solutions possibles.

Dans l'hypothèse où le réseau est constitué d'éléments hétérogènes, le moniteur a pour rôle de rendre le réseau transparent: chaque station voit un ensemble de fonctions, sans avoir à connaître leur répartition.

Ainsi, dans l'exemple de la section II.4, il est nécessaire que l'application puisse être écrite pour permettre la répartition, mais indépendamment du fait que cette répartition existe effectivement ou non: deux modules de l'application peuvent s'exécuter soit sur un même processeur, soit sur des machines différentes, la transparence au niveau application devant donc être assurée par le moniteur.

Il est commode de considérer une relation dissymétrique entre des processus utilisateurs (ou *clients*), et des processus *serveurs*, exécutant la fonction demandée. Une fonction peut appeler à son tour une autre fonction. Les synchronisations entre processus sont relativement simples: un processus client émet une requête et attend la réponse...

Le travail est réparti entre les postes de travail, supportant les processus liés aux utilisateurs, et les processeurs fonctionnels, exécutant les fonctions. Les interactions peuvent être des **dialogues**, où les processus exécutent plusieurs **étapes de traitement** séparées par des échanges de messages, ou des **services** qui se limitent à l'initialisation d'un travail distant et à un compte rendu de fin d'exécution. Une tâche répartie aura une structure hiérarchisée ayant à son sommet un processus utilisateur.

L'analogie entre la communication dans les systèmes répartis et les entrées sorties classiques amène généralement à faire communiquer les processus par l'intermédiaire de "canaux logiques", mono ou bidirectionnels, que l'on appelera ici *ports* [TES 79, BAN 80]. Une telle solution s'apparente aux paramètres formels dans les procédures, et rejoint également le principe de la communication par "tubes" dans Unix.

Une fonction sera connue dans le réseau par l'intermédiaire des ports qui lui permettent de communiquer. Un processus utilisateur peut, lui aussi, se définir des ports qu'il pourra connecter aux ports des fonctions qu'il souhaite utiliser.

Le rôle d'un port est donc de permettre l'indépendance des programmes et des composants externes avec lesquels ils sont en interaction, de faciliter la transmission des messages en mémorisant les informations concernant le connexion, et enfin de permettre au moniteur de contrôler l'activité des processus,en contrôlant la création, la suppression et l'utilisation des ports. Les ports peuvent également contenir des informations sur l'état des processus, le type d'interaction, etc....

II.7.1. Allocations de ressources:

Il n'y a pas de dynamisme dans la répartition des tâches, mais par contre, chaque processus client peut déclencher des allocations de ressources sur une ou plusieurs stations fonctionnelles. Le problème essentiel est donc le contrôle des allocations de ressources en l'absence d'un allocateur centralisé.

En fait, il n'existe de problème spécifique aux systèmes distribués que dans le cas où un processus utilisateur établit des dialogues avec plusieurs serveurs répartis sur plusieurs stations fonctionnelles, et lorsque des allocations de ressources portent sur plusieurs étapes de traitement: en effet, ce n'est que dans ce cas que peuvent apparaître des risques d'interblocages entre des processus tournant sur des stations différentes, donc non contrôlables par l'allocateur de ressources de chaque station. Un exemple de scénario d'interblocage est alors le suivant: un processus obtient une ressource A sur la station X, puis demande une ressource B sur la station Y. Ce processus est mis en attente, car la ressource B est détenue par un autre processus qui, à son tour demande la ressource A sur la station X et est mis en attente...

Un contrôle des allocations de ressources doit donc être effectué au niveau du moniteur. Ce contrôle peut être centralisé au niveau d'une seule station, la station maître, ou réparti. La solution répartie est plus complexe à mettre en oeuvre, mais elle évite de créer un goulot d'étranglement au niveau de la station maître.

Ceci demande, dans tous les cas, qu'une certaine règle du jeu soit établie et respectée par l'ensemble des processus: les allocations de ressources doivent être connues du moniteur, de façon à ce qu'il puisse appliquer une stratégie de prévention ou détection des interblocages.

Une solution pour la prévention des interblocages peut être basée sur la hiérarchisation des ressources, et l'allocation statique des ressources de même niveau. Il est possible de se baser pour cela sur une décomposition des dialogues en phases d'allocation et de traitement, la transition entre les phases étant contrôlée par le moniteur. On peut alors réinitialiser une phase d'allocation, si l'échec d'une allocation introduit un risque d'interblocage.

II.8. PROLOG ET DISTRIBUTION FONCTIONNELLE:

S'intéressant à Prolog et à la distribution fonctionnelle, il est nécessaire de voir dans quelle mesure Prolog autorise - ou impose - une approche particulière de celle-ci. Ayant par ailleurs, dans le cadre du projet Opale, défini un mécanisme d'unification capable de travailler à une vitesse compatible avec le débit d'un disque, nous avons voulu voir si un tel mécanisme peut être appliqué à d'autre contextes, tel que celui des réseaux locaux.

Dans cette section, on se propose de dégager les problèmes, et d'indiquer quelques idées générales sur la conception de systèmes répartis en Prolog, sans aller très loin toutefois: ceci nécessiterait en effet une étude à part entière.

Un programme Prolog est un ensemble de clauses permettant de vérifier des prédicats. Une clause est à son tour une suite de prédicats à vérifier. Autrement dit, la

programmation en Prolog consiste à partir d'un problème initial et à le décomposer en sous problèmes, jusqu'à obtention de problèmes suffisamment simples pour qu'ils puissent être résolus soit par des assertions (clauses sans queues), soit par des prédicats évaluables (prédicats prédéfinis). Etant donné l'énoncé d'un problème, c'est à dire un prédicat à vérifier, on sélectionne la clause à essayer par le mécanisme d'unification.

Un système Prolog réparti devra donc tirer parti de ce mécanisme naturel de décomposition des problèmes pour permettre une résolution distante de certains prédicats.

D.S. Warren propose un modèle d'exécution répartie de Prolog sur un réseau en diffusion [WAR 84]. Il s'agit d'une approche qui se veut pragmatique, dans laquelle des prédicats peuvent être soit "universels", c'est à dire que toute station du réseau est capable de le vérifier de façon autonome (c'est le cas, par exemple, des littéraux évaluables), soit "répartis", c'est à dire que les clauses permettant de la vérifier sont réparties sans duplications entre les stations du réseau. Un mécanisme de résolution basé sur la diffusion des buts et des substitutions générées est proposé dans l'article cité.

Dans un contexte de distribution fonctionnelle, c'est à dire basé sur une relation dissymétrique entre des serveurs spécialisés et des postes de travail, il nous semble qu'une organisation plus simple (et sans doute plus efficace) peut consister à distinguer, aux niveaux de chaque station, deux types de prédicats:

- des prédicats vérifiables localement, sous forme de paquet de clauses, ou de prédicats évaluables,
- des prédicats vérifiables à l'extérieur, sur le réseau.

Un même prédicat n'est alors réparti sur plusieurs stations que dans la mesure où la fonction elle même est répartie (cas de bases de données, par exemple). Il faut donc pouvoir reconnaître que certains prédicats sont définis sur une autre station, et émettre dans ce cas une demande de vérification sur le réseau. Ceci peut être effectué par l'interpréteur au niveau du traitement des prédicats non vérifiables: si un tel prédicat n'est pas défini pour l'interpréteur local (c'est à dire s'il n'existe pas de paquet de clauses ou de prédicat évaluable permettant de la vérifier), alors, trois cas peuvent se présenter:

- il s'agit d'un cas d'échec normal: pour nombre d'interpréteurs, la non définition d'un prédicat est traitée comme un échec. Cependant, il est peu pénalisant (et même, à notre avis, avantageux), de considérer ce cas comme une erreur. Quoi qu'il en soit, ce cas reste assez rare.
- il s'agit d'une erreur, auquel cas, il n'y a pas d'inconvénient à pénaliser la station locale, mais il ne faudrait pas pénaliser l'ensemble du réseau.
- il s'agit d'un prédicat vérifiable par une station distante.

Il s'agit donc pour l'interpréteur local de savoir distinguer entre ces trois cas, et de répercuter le troisième vers la ou les stations distantes concernées.

Pour effectuer cela localement, l'interpréteur devrait disposer d'une liste des stations du réseau, avec les prédicats qu'elles savent vérifier, liste qui n'existe pas forcément, ou n'est pas nécessairement disponible en totalité à tout instant. La solution consisterait donc à interroger le réseau en diffusant la question (recherche d'un serveur).

Chaque station fonctionnelle peut alors filtrer (unifier) les questions qui circulent sur le réseau (sous forme de littéraux), de façon à reconnaître les problèmes qu'elle sait résoudre (qui unifient avec les clauses qu'elle possède). La station qui reconnaît un littéral qu'elle sait résoudre peut alors retourner son identification au processus demandeur. On utilise alors le mécanisme d'unification de Prolog non seulement pour l'expression d'un problème, mais aussi pour l'identification d'un serveur sur le réseau.

Ceci suppose que l'on dispose d'un mécanisme efficace permettant l'unification de données circulant sur le réseau avec un ensemble d'en têtes de clauses. Un tel mécanisme sera présenté dans la suite de cette thèse.

A partir du moment où la station capable de vérifier le littéral est identifiée, on peut traiter les appels en insérant une clause permettant l'émission directe du littéral vers la station capable de la résoudre, de telle sorte que ceci s'insère dans la résolution sous la forme d'un prédicat évaluable.

Exemple: On considère l'émission d'une requête vers un serveur bases de données. Le littéral d'appel peut être le suivant: bdreq(R, S), ou les variables R et S permettent respectivement l'émission de la requête et le retour des résultats. A la première activation du littéral, son émission sur le réseau permet d'identifier la station MBD comme serveur compétent pour le résoudre. On doit alors insérer dans la base de clauses la clause suivante, permettant de transmettre la requête directement vers la station concernée:

bdreq(R, S) :- émettre(MBD, bdreq(R, S)).

Un problème reste donc ouvert: celui de l'échec de la recherche d'un serveur. Ceci peut signifier soit l'échec du prédicat (si l'interpréteur admet ce cas), soit une erreur, soit une indisponibilité du serveur concerné. Même si on élimine le premier cas (un prédicat inconnu est une erreur), on aboutit tout de même à une ambiguité sur la cause de l'erreur (erreur du programmeur: prédicat inconnu ou serveur inconnu, ou indisponibilité temporaire d'un serveur). Cette ambiguité peut être levée soit grâce à l'existence d'une liste des serveurs, avec les squelettes des requêtes qu'ils savent résoudre, soit par un mécanisme de time-out (on suppose alors que tout serveur qui reconnaît son nom dans un message émet une réponse, même si la requête est incorrecte).

Il peut arriver qu'une même requête puisse être résolue par plusieurs serveurs. De plus, un serveur peut retourner non une solution unique, mais un ensemble de solutions. Il existe donc un double non déterminisme, qui peut être résolu de deux façons:

- Soit on construit la liste des solutions, qui est ensuite traitée globalement,
- Soit on traite les solutions une par une, chaque nouvelle activation du littéral par backtracking entraînant alors la prise en compte d'une nouvelle solution.

La première solution correspond aux prédicats "setof" ou "bagof" introduits dans les interpréteurs Prolog d'Edimbourg. La seconde solution a été insérée dans un interpréteur Prolog pour l'accès à une base de données, dans le cadre du projet Opale.

Le non déterminisme fournit une solution simple pour dégager un premier niveau de parallélisme dans l'exécution: en effet, le traitement d'une solution peut sans difficulté se dérouler en parallèle avec l'évaluation des autres solutions. Un mécanisme de type producteur consommateur permet la synchronisation du serveur distant (producteur), et du client (consommateur).

Un parallélisme plus poussé pourrait être recherché: Prolog concurrent, exécution en pipe line (étudiée dans le cadre d'Opale), etc... Il est également intéressant de rechercher l'insertion dans le langage Prolog des mécanismes de bases de communication et de synchronisation des systèmes parallèles (énoncés gardés, par exemple).

On retrouve également dans le cas de Prolog les problèmes de conflits dans l'allocation de ressources sur des serveurs multiples. Cependant, le mécanisme inhérent de backtracking peut permettre des solutions de type détection/résolution (en cas d'interblocage, on peut forcer un processus à revenir en arrière). Ceci est possible tant que la résolution ne provoque pas de modification permanente et irréversible de l'état du système (modifications d'une base de données, par exemple). L'implémentation d'un mécanisme de journalisation permet toutefois de revenir en arrière même dans ce type de situation.

Il faut alors définir deux types de retour arrière: le retour normal, consistant à défaire des substitutions et à chercher des alternatives, et le retour forcé consistant à défaire ce qui a été fait jusqu'à un certain point, et à recommencer la résolution.

II.8.1. Conclusions:

La répartition d'applications dans le cadre de la programmation en logique pose donc un ensemble de problèmes particuliers, mais permet aussi des solutions particulières. Il semble que les mécanismes de base à adopter restent les mêmes que dans les autres types de programmation, le problème ici étant de trouver une façon cohérente de les insérer dans un interpréteur Prolog.

Trois points sont à relever:

- La décomposition naturelle en sous problèmes des programmes Prolog, favorisant la répartition, même si le parallélisme n'est pas aussi immédiatement sous-jacent qu'on le pense souvent (nous reviendrons sur ce point ultérieurement).
- La grande puissance de l'unification, qui peut intervenir dans de multiples aspects d'un système réparti: sélection et adressage d'un serveur ou d'une fonction particulière, vérification syntaxique des messages, transmission des requêtes et des données, transmission des résultats. L'unification peut même constituer en elle même tout ou partie de l'opération d'un serveur. Cependant, cette multiplicité introduit une ambiguité qui ne peut être levée que par des mécanismes complémentaires: en effet, un échec de l'unification peut être dû à des causes très diverses, et devrait être répercuté de façon différente selon le cas. Quoi qu'il en soit, on peut trouver là une application d'un dispositif matériel qui permettrait d'effectuer tout ou partie de l'unification sur un flot de données.

- Le non-déterminisme, qui constitue une forme nouvelle d'interaction entre une fonction et un client, et qui, là encore peut être traité de plusieurs façons, avec éventuellement un parallélisme entre la fonction et le client.

Chapitre III

MACHINES BASES DE DONNEES

III.1. INTRODUCTION:

Dans les chapitres précédents, nous avons établi l'intérêt d'une nouvelle approche pour la conception des systèmes informatiques: la répartition du traitement entre plusieurs machines spécialisées. Parmi les types de machines spécialisées envisageables, les machines bases de données présentent un intérêt particulier: il s'agit en effet de machines adaptées à des types de traitement essentiellement non-numériques, alors que les machines courantes sont le plus souvent essentiellement orientées vers le calcul. Avec les machines bases de données, on recherche donc la conception d'opérateurs originaux.

En effet, on trouve dans les systèmes de gestion de bases de données des catégories d'opérations pour lesquelles les machines classiques sont mal adaptées: opérations de recherche complexes sur de vastes volumes de données, traitement d'ensembles de données non résidents en mémoire primaire, etc... Cette inadaptation des architectures classiques s'est encore accrue avec les bases de données relationnelles, qui sont basées sur un modèle théorique, et non sur des algorithmes d'accès. De plus, l'importance croissante de la fonction bases de données dans les systèmes informatiques rend ses performances encore plus critiques (temps de réponse et temps CPU consommé). Ceci a donc amené à penser que la solution pouvait résider dans la conception de matériels spécialisés, ce qui nous amène au coeur du débat de cette thèse.

Tous ces points ont déjà suscité de nombreux travaux sur lesquels nous reviendrons brièvement dans la suite de ce chapitre. Pourtant, comme nous tâcherons de le mettre en évidence, tous les problèmes sont loin d'être résolus, certaines voies de recherche ont abouti à des impasses, certains problèmes importants ont été peu ou pas abordés, et il n'existe pas véritablement de vision d'ensemble de ce que doit être une machine bases de données: la recherche en est restée jusqu'ici à la définition des outils, sans même aller véritablement jusqu'à une comparaison objective de ces outils, ou à l'utilisation combinée d'outils complémentaires.

Par ailleurs, les bases de données évoluent maintenant vers de nouveaux types d'applications, souvent issues de l'intelligence artificielle: bases de données déductives, bases de connaissances, etc... Tout ceci peut influer sur le matériel, et entraîner la recherche de nouvelles solutions.

La suite de ce chapitre sera consacrée à une discussion générale sur les machines bases de données. Nous indiquerons en conclusion les grandes lignes du projet Opale, dont les points essentiels seront abordés dans les chapitres suivants.

III.2. ETAT DE L'ART:

Nous ne chercherons pas ici à effectuer une revue bibliographique complète du domaine de machines bases de données, mais plutôt à faire le point sur les problèmes abordés, les résultats acquis, et les perspectives d'avenir. Une étude plus complète, et assez récente peut être trouvée dans [SCH 85].

La grande majorité des projets dans le domaine des machines bases de données sont tournés vers le modèle relationnel, et le traitement associatif des données. Les projets se répartissent selon deux directions essentielles: le traitement "au vol" d'informations lues sur le disque, ou le traitement d'ensembles d'informations précédemment amenées en mémoire primaire.

III.2.1. Projet MAGE:

Le projet MAGE [NAV 79], mené à l'IMAG a été un des premiers projets dans le domaine des machines bases de données. Il était orienté vers des bases de données hiérarchiques de type SOCRATE (adressage calculé des informations, l'accès associatif n'étant en principe qu'un mode d'accès secondaire, réalisé par hash coding sur demande explicite à la création de la base: notion de clé).

Le principe de la machine MAGE est de dédier un ensemble de microprocesseurs au disque, pour en permettre une utilisation optimale, grâce à une méthode d'accès adaptée. La possibilité d'utiliser des microprocesseurs 8 bits pour la gestion d'une base de données avec des objectifs limités (petites bases de données pour de petits systèmes de gestion) a été démontrée. Une architecture multiprocesseurs permettant l'exécution des différentes couches logicielles de la méthode d'accès a été définie, ainsi que le mécanisme de communication entre les processeurs.

L'interface qui a été définie entre la machine bases de données et l'environnement utilisateur (processeur central à l'origine) est de type navigationnel (positionnement dans la hiérarchie par déplacements élémentaires), ce qui permettait de simplifier le dialogue, mais semble mal adapté au cas des systèmes fonctionnellement répartis (niveau trop élémentaire des interactions). Les limites de ce projet sont également celles des bases de données hiérarchiques qui semblent de plus en plus surclassées par les bases de données relationnelles.

III.2.2. Processeurs matriciels:

Les processeurs matriciels [BOD 81, BER 80, KRU 81] sont conçus pour permettre un traitement rapide d'ensembles d'informations précédemment amenés en mémoire primaire. Ils sont le plus souvent organisés en réseau matriciel d'opérateurs capables d'opérer en parallèle sur les éléments d'ensembles d'informations organisés en tableaux. Ces processeurs sont donc capables d'effectuer très rapidement les opérations de l'algèbre relationnelle sur des ensembles de données de taille relativement limitée.

Lorsque le volume d'informations augmente, le problème est alors que la totalité des informations ne tient pas dans l'ensemble mémoire + opérateurs de la machine: il faut alors échanger les informations entre le disque et la mémoire primaire, ce qui implique

des délais supplémentaires et une perte de performances. Une autre difficulté est que l'évolution de la technologie ne semble pas encore avoir favorisé ce type de conception, et l'intégration d'ensembles mémoire-logique de grande dimension (c'est à dire comparable aux volumes de mémoire primaire couramment réalisable) ne semble pas encore avoir été effectuée.

III.2.3. Machines à balayage:

Les machines à balayage sont basées sur l'idée qu'il est possible de traiter (sélectionner) les données au cours du transfert entre le disque et la mémoire du processeur de traitement. Nous distinguerons 3 générations, correspondant à des étapes de l'évolution technologique, les deux premières voies, bien que très prometteuses ayant été contrariées par l'évolution technologique.

III.2.3.1. 1ère génération:

La première génération est représentée par des machines telles que CASSM [LIP 76] OU RAP [OZK 75]. L'idée de base de telles machines repose sur le fait que, sur un disque à têtes fixes, l'ensemble des informations défile sous les têtes en 16.7 ms (durée d'une rotation du disque). Si on place une logique suffisamment rapide au niveau de chaque tête, il sera donc possible d'effectuer des sélections, même assez complexes, dans la durée d'une révolution du disque, les opérations les plus complexes (jointures...) pouvant être exécutées en quelques tours. Ceci demandait de disposer d'une électronique de lecture / écriture, et d'une logique de traitement par tête, mais on s'attendait à ce que les progrès de l'intégration rendent cela économiquement acceptable, ce qui pouvait le devenir d'autant plus que cela permettait d'atteindre des débits de traitement très élevés.

Le problème est venu du fait que les progrès dans les densités d'enregistrement des disques magnétiques étaient inapplicables aux disques à têtes fixes. Ceux ci sont donc devenus d'un coût prohibitif par rapport aux disques à bras mobile et ont assez rapidement disparu du marché.

III.2.3.2. 2ème génération:

Les disques à têtes fixes étant désormais des pièces de musées, on cherche à adapter le même principe aux disques à bras mobile, en mettant une logique de sélection par tête. Cette conception a d'ailleurs débouché sur un produit industriel, le disque Ampex à lecture parallèle (le coût de ce produit ne lui a pas permis de prendre un marché très important).

Avec l'augmentation des densités, la précision requise pour le positionnement des têtes a encore augmenté (actuellement, la largeur d'une piste représente moins de 0,03 mm), devenant sensible à des effets tels que la dilatation des disques ou des mécanismes de déplacement de bras, et demandant des précisions mécaniques extrêmes. Une telle précision n'est le plus souvent obtenue qu'au prix d'un asservissement de la position des têtes sur l'amplitude du signal lu sur le disque: les têtes sont déplacées de part et d'autre de la position nominale de la piste jusqu'à l'obtention d'un signal d'amplitude maximale.

Un tel asservissement ne peut être réalisé que sur une piste à la fois, si bien que l'on n'est en général pas assuré de pouvoir lire plus d'une piste en même temps (à moins d'une précision extrême et coûteuse de la partie mécanique du système de déplacement de bras).

Il en résulte que le principe de l'accès parallèle aux têtes d'une même unité de disques reste encore inapplicable. Le principe du filtrage n'est donc utilisable que sur un disque accédé de façon classique par un seul canal de lecture / écriture. Des évaluations de performances [BOR 81] semblent d'ailleurs montrer que le gain que l'on peut espérer retirer de l'accès parallèle aux têtes d'un disque est relativement réduit, compte tenu des possibilités d'indexation, qui restreignent le domaine de recherche, de la capacité de traitement du ou des processeurs situés en aval, et de la bande passante du moyen de communication.

III.2.3.3. 3ème génération:

Les difficultés d'expérimentation de l'accès parallèle aux têtes d'un disque ont, depuis longtemps, suscité des recherches sur le seul aspect de la sélection au vol des données.

Un certain nombre de projets ont porté sur des opérateurs cellulaires ou systoliques [PRA 81], alors qu'en France, l'intérêt s'est depuis longtemps tourné vers l'utilisation du filtrage par automates d'états finis, avec en particulier les projets TREFLE, puis VERSO [TUS 80, SCH 85, BAN 81] et SCHUSS [GON 84].

Deux conceptions existent:

- soit l'information est filtrée directement lors de la lecture sur le disque. Un premier problème est donc de suivre exactement le débit du disque, tout retard dans le traitement des données se traduisant par la perte d'une rotation du disque. Par ailleurs, l'utilisation de techniques de correction d'erreurs (de plus en plus nécessaire avec l'augmentation des densités) entraîne la nécessité de stocker au moins un secteur avant de le traiter.
- soit l'information est amenée en mémoire avant d'être filtrée, ce qui résoud les problèmes de débit et de correction d'erreur. Par contre, on réintroduit un retard entre l'accès aux données sur le disque et leur traitement. Une conception intermédiaire, généralement adoptée maintenant, est d'avoir, entre le disque et le filtre, une petite file d'attente de quelques secteurs, permettant de résoudre les problèmes de correction d'erreur et de débit, tout en minimisant le retard de traitement. En effet, ce retard peut être préjudiciable aux performances, dans le cas d'une base de données peu utilisée, ainsi que dans le cas où l'on recherche une utilisation optimale des ressources par exploitation du parallélisme intra requête.

Dans la conception même du filtre, on peut encore distinguer deux approches:

- l'une cherche à effectuer au vol la totalité de la sélection, en suivant dans tous les cas le débit du disque (ou éventuellement avec un très faible risque de retard),
- l'autre consiste à utiliser un préfiltre [ROH 81] qui a pour but de réduire de façon significative le débit de données, de telle sorte que le reste de la sélection puisse être effectué par logiciel sur un microprocesseur: on simplifie ainsi le filtre, tout en permettant des performances aussi bonnes que celles obtenues par un filtrage entièrement matériel.

Plutôt que de chercher à suivre dans tous les cas le débit du disque, il est également possible d'admettre un très faible risque que le filtrage soit plus lent que le débit des données, à condition que la seule conséquence en soit une attente du disque. La perte de performance peut en effet être globalement négligeable, alors que la simplification du filtre peut être importante.

III.2.4. Architectures multiprocesseurs:

Un certain nombre de projets sont basés sur des architectures multiprocesseurs, où le traitement peut être exécuté parallèlement sur plusieurs processeurs.

DIRECT:

Ce projet [DEW 79] repose sur une interconnexion matricielle de modules mémoires basés sur des CCD, de processeurs de requête, et de mémoires de masse (disques). Les mémoires de masse et les processeurs de requêtes sont directement contrôlés par un processeur de contrôle. Les informations à traiter sont amenées de la mémoire de masse dans les modules mémoire, puis traités par les processeurs de requête. Les CCD (qui sont en fait des mémoires à décalage) sont décalés parallèlement par une horloge commune, l'adresse courante des mémoire étant accessible par l'ensemble des processeurs de requête, ce qui permet d'optimiser les opérations.

Un apport important de DIRECT a été l'étude d'algorithmes parallèles pour l'accès et le traitement des données. Par contre, ce projet est fortement lié à la technologie des mémoires à CCD, technologie qui, à l'instar des mémoires à bulles, n'a pas eu les développements espérés: si cette technologie trouve actuellement des applications prometteuses dans le domaine de l'optoélectronique, son utilisation dans le domaine des mémoires se heurte au développement des mémoire RAM dynamiques, beaucoup plus performantes, leur volatilité ne les rendant par ailleurs pas directement applicables en tant que mémoire secondaires.

SABRE:

Ce projet [GAR 81] repose sur un ensemble de processeurs communiquant par une mémoire commune avec un ensemble d'unités de mémoires secondaires. L'apport de ce projet ne réside pas tant dans l'architecture de la machine elle même (celle ci n'ayant d'ailleurs jamais été réalisée) que dans l'étude d'algorithmes multiprocesseurs pour un certain nombre d'opérations sur les bases de données (tris, par exemple).

DELTA:

Delta fait partie du projet japonais de machines de 5ème génération [SAK 85], et vise donc à être un élément d'un système de traitement des connaissances. La machine est basée sur un ensemble de processeurs et une "unité de mémoire hiérarchisée" (UMH) intégrant mémoire primaire et mémoire secondaire. l'UMH fournit un niveau de mémorisation commun aux autres unités de Delta. Des unités relationnelles permettent d'effectuer diverses opérations de l'algèbre relationnelle. Comme toutes les autres unités de la machine, elles accèdent aux informations en mémoire secondaire par l'intermédiaire de l'UMH. D'autres processeurs assurent enfin le contrôle de la machine et l'interface avec l'extérieur.

DBC / 1012:

La machine DBC / 1012 de TERADATA est un produit industriel, et non un projet de recherches. Elle est basée sur l'interconnexion en réseau "YNET" de processeurs assurant le filtrage des données et attachés aux unités de mémoire secondaire, et de processeurs assurant le traitement du niveau supérieur des requêtes. Le réseau YNET est une arborescence binaire dont les feuilles sont les processeurs, le nombre de noeuds à traverser étant donc au pire cas égal à $2\log_2(N-1)$, où N est le nombre de processeurs.

III.3. PERSPECTIVES D'AVENIR:

III.3.1. Généralités:

Les projets précédents dans le domaine des machines bases de données ont permis la définition d'un certain nombre d'outils: architectures et algorithmes multiprocesseurs, filtres, etc...

Cependant, la grande majorité des projets restent tournés de façon quasi exclusive vers l'utilisation d'un seul de ces moyens, sans envisager l'utilisation combinée de moyens qui devraient pourtant être complémentaires. De même, peu a été fait en ce qui concerne le problème pourtant essentiel de la tolérance aux pannes.

Il ne semble pas non plus à l'heure actuelle qu'un projet de machine bases de données ait apporté la preuve de l'efficacité des moyens qu'il met en oeuvre. Il y a quelques années, beaucoup de chercheurs se sont intéressé à l'évaluation des performances dans les Machines Bases de Données. Des travaux intéressants ont été exécutés dans ce domaine [BOG 83, BOR 83], cependant, il ne semble pas à l'heure actuelle qu'il en sorte un résultat clair, ni une méthodologie d'évaluation des performances.

Dans le domaine des architectures, beaucoup a été fait sur le parallélisme, avec des machines multiprocesseurs, et quelques approches de machines data-flow [BOR 82]. Par contre, aucun modèle de machine pipe-line n'a encore été proposé, alors que ce type d'architecture - plus classique, il est vrai - pourrait semble-t-il s'adapter assez bien au problème des bases de données relationnelles.

III.3.2. évolution de la technologie:

De nombreuses études ont été faites en tablant sur la possibilité d'utiliser un parallélisme massif dans le traitement des données, et l'accès à la mémoire secondaire.

Avec l'évolution de la technologie des mémoires secondaires, on assiste à une concentration des mémoires dans des unités de capacité de plus en plus élevée: malgré l'accroissement des performances, la diminution du nombre d'unités amène à une diminution du débit d'entrées sorties effectif que l'on peut atteindre. Un exemple simple peut nous en convaincre: Considérons une base de données d'un milliard d'octets. Si cette base est répartie sur une dizaine d'unités de 100 mégaoctets, avec un temps d'accès moyen de 40 ms, et un débit de 10 Mbits/s, il sera en moyenne possible d'accéder 10 secteurs de 256 octets en 40,2 ms. Si par contre, la base de données est stockée sur une seule unité de 1 milliard d'octets, avec un temps d'accès moyen de 25 ms et un débit de 20 Mbits/s, il faudra en moyenne 25,1 ms pour accéder un seul secteur de 256 octets.

On a donc un débit effectif de quelques 64K octets par seconde dans le premier cas, et de 10K octets par seconde dans le second. Le rapport reste tout aussi défavorable si on considère l'accès à des volumes de données plus importants: pour 16K octets, on a 2,8 Mégaoctets par seconde dans le premier cas, et 0,5 dans le second.

Par ailleurs, l'évolution de la technologie des circuits intégrés apporte de plus en plus de puissance de traitement au niveau des microprocesseurs, et des mémoires de plus en plus importantes: le coût du traitement et celui de la mémoire primaire diminuent donc fortement. Compte tenu de l'évolution de la technologie des mémoires secondaires, on peut alors arriver à la conclusion qu'il est inutile de chercher à exploiter le parallélisme dans les machines bases de données, puisqu'une base de données, même très grosse, sera répartie sur un très petit nombre d'unités de mémoire secondaire, et que la puissance de traitement disponible sur un microprocesseur sera de plus en plus capable de satisfaire à elle seule les besoins des plus gros systèmes de gestion de bases de données. Il serait donc - toujours selon le même raisonnement - préférable de miser sur les volumes de mémoire primaire importants dont on peut désormais disposer.

Notre analyse est assez différente de celle ci. D'une part, un phénomène nouveau se dessine dans l'évolution de la technologie des mémoires secondaires, et n'a pas été pris en compte: il s'agit de la généralisation de très petites unités de mémoire secondaire destinées au marché des microordinateurs: unités "winchester" de quelques dizaines de mégaoctets. Avec des performances relativement modestes, mais un marché important, ce type d'unité est d'un coût très bas. L'écart en coût au bit entre ces petites unités et les plus grosses tend à se réduire, et on peut vérifier assez facilement qu'un ensemble d'unités de faibles performances accédées efficacement en parallèle peut offrir un débit d'entrées sorties très supérieur à celui que l'on obtiendrait sur une grosse unité de taille équivalente. De plus, ce type d'unités est de performances compatibles avec un contrôleur intégré, ce qui permet de réduire très significativement une partie importante du coût d'une mémoire secondaire: celui du contrôleur. Le prix à payer pour un tel parallélisme est évidemment au niveau de la complexité du logiciel chargé de le gérer.

Il est par ailleurs intéressant de noter que l'évolution de la technologie ne va pas dans le sens d'unités de plus en plus gigantesques, mais plutôt vers une compacité de plus en plus grande et un coût de plus en plus réduit des unités de volume courant. Ainsi, l'utilisation de densités de plus en plus élevées sur des disques de taille courante conduit à des fréquences de transfert de plus en plus élevées, fréquences qui deviennent difficile à contrôler sur les ordinateurs: débit du contrôleur, cadence d'accès à la mémoire primaire, etc... Des unités aussi performantes risqueraient donc de se voir limitées au marché des très gros ordinateurs, ce que nombre de fabricants de périphériques ne peuvent pas admettre. Une solution - adoptée par CDC sur un de ses disques de grande capacité - est alors de réduire la vitesse de rotation des disques. L'inconvénient est alors que l'on augmente ainsi le délai rotationnel, et donc que l'on réduit les performances.

Il ne semble pas que l'on assiste à un accroissement spectaculaire du volume stocké sur les plus grosses unités. On peut par contre constater la multiplication d'unités de disque compactes (5 pouces 1/4) atteignant le volume de stockage de ce qui était jadis de très grosses unités (100 à 300 méga octets).

Sur ce point, on peut donc tirer provisoirement les conclusions suivantes:

- Le cadences de transfert ne devraient augmenter qu'à la mesure des performances moyennes de la logique de contrôle. Il serait peu raisonnable, pour un constructeur, de produire une unité de disque qui ne soit contrôlable que par des matériels ultra performants, et donc chers et d'un marché restreint.
- L'évolution devrait prendre en compte les questions de performances, le gigantisme et la concentration allant, comme nous l'avons vu, à l'encontre de la rapidité des unités, et du débit global de la mémoire secondaire.
- On peut s'attendre essentiellement à voir des unités de mémoire secondaire d'un volume de stockage de plus en plus important s'installer sur les systèmes les plus modestes.

Un autre point non encore pris en considération est celui de la tolérance aux pannes: le fait de répartir l'espace mémoire secondaire sur un ensemble d'unités ne répond pas seulement à des impératifs de volume mémoire, mais aussi à des impératifs de fiabilité. En effet, si une base de données est stockée sur un seul disque, en cas de panne de celui ci, c'est toute la base de données qui est hors service, et vraisemblablement le système complet. Si au contraire, la base de données est stockée sur un ensemble de disques, une panne d'un disque n'affectera qu'une petite partie de la base, le risque que des utilisateurs soient affectés est plus réduit, la restauration de la base pourra se faire rapidement, et le travail se poursuivre dans de bonne conditions de sécurité. Ce point nous semble particulièrement important dans les systèmes informatiques où les unités de mémoire secondaire sont bien souvent parmi les éléments les plus faibles sur le plan de la fiabilité: une panne d'une unité de mémoire secondaire est donc loin d'être un évênement hypothétique.

III.4. LE FILTRAGE:

III.4.1. Principe:

Le filtrage, dans son principe, dérive de l'algorithme de recherche de Knuth [KNU 77]: partant d'une question plus ou moins complexe, il s'agit d'effectuer un prétraitement permettant de la résoudre à l'aide d'un algorithme performant, éventuellement réalisable par matériel. Cet algorithme tire sa puissance du fait qu'il utilise l'information implicitement contenue dans l'état même du processus de recherche pour éviter des comparaisons redondantes.

Exemple:

On recherche le mot *abc* dans le texte *ababc*. L'algorithme de recherche va reconnaître le début du mot cherché dans les deux premières lettres du texte, puis la comparaison de *c* avec *a* échoue. L'algorithme de recherche "naïf" reprendrait la recherche à partir du début du mot et du second caractère du texte, ce qui implique une comparaison inutile: du fait qu'on a déjà reconnu le *a* et le *b*, il est inutile de reprendre la comparaison au premier *b* du texte, qui ne peut pas débuter le mot cherché. L'analyse préalable du mot recherché permet de reprendre la comparaison au point où elle a échoué, et donc de comparer le début du mot avec le second *a* du texte.

Supposons maintenant que l'on recherche le mot *babac* dans le texte abababac. *On reconnaît le début du mot recherché depuis le premier jusqu'au troisième b.* A ce point, le fait d'avoir reconnu un *b* et non un *c* nous indique qu'on a déjà reconnu le début du mot (*bab*), et qu'on peut poursuivre la comparaison à partir du caractère suivant du texte, et du quatrième caractère du mot, sans revenir en arrière dans le texte.

Ce type d'algorithme peut s'implémenter sous forme d'automate d'états finis, l'analyse de la chaîne recherchée permettant de générer les tables de transition de l'automate. Il est possible, de la même façon de rechercher non des chaînes uniques, mais plusieurs chaînes en même temps. Même dans le cas d'une exécution logicielle, l'algorithme de filtrage peut être considérablement plus efficace qu'une recherche "naïve". Une efficacité maximale pourra être obtenue lorsque la question est complexe et présente de nombreuses alternatives.

L'exécution matérielle du filtrage [ROH 81] va permettre une réduction considérable du nombre des accès mémoire, puisqu'il n'y a alors pas d'instructions à lire: les seuls accès mémoire sont les accès "utiles" aux données. Il s'agit là d'un avantage important des automates câblés, les accès mémoire restant parmi les opérations les plus lentes d'un processeur.

Le filtre va alors pouvoir s'insérer entre le contrôleur disque et la mémoire primaire. Il communique avec le contrôleur par l'intermédiaire d'une petite file d'attente (un ou deux secteurs, soit 256 à 1024 octets). Il doit être associé à un processeur qui compile ses requêtes, charge ses tables et récupère ses résultats (Fig. III.1.). Le volume de

mémoire requis par les tables du filtre est un paramètre important: en effet, ces tables ont intérêt à être physiquement séparées de la mémoire propre du processeur, car le filtre risque d'y accéder intensivement durant son exécution, et elles doivent en outre être rapides. Un volume réduit permet donc de simplifier leur réalisation: petite mémoire statique, ou encore mémoire intégrée dans le circuit de filtrage lui même.

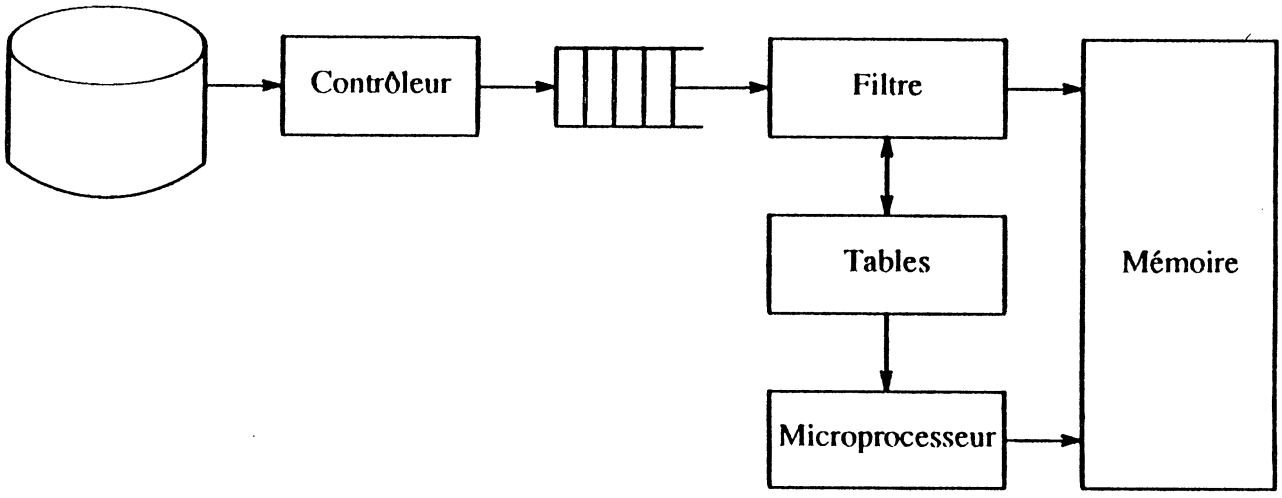


Figure III.1.

Etant donnée une requête, la recherche va se décomposer en deux opérations:

- Le prétraitement, ou compilation. Sa durée peut être éventuellement très longue, mais il n'est effectué qu'une fois, quelle que soit la longueur de la recherche.
- Le filtrage proprement dit,

Si T_c est le temps de compilation, T_f le temps de filtrage par élément de donnée, et N le nombre d'éléments de données filtrés, le temps d'exécution total sera de: $T_c + T_f \times N$.

Par contre, le temps d'exécution d'une recherche classique sera de: $T_r \times N$, où T_r est la durée élémentaire de la recherche. Pour N suffisamment grand, le temps de compilation devient donc négligeable, et le filtrage trouve alors sa pleine efficacité. S'il peut être implémenté par matériel, il permettra alors de traiter des données "au vol", c'est à dire en parallèle avec leur transfert depuis une unité de mémoire secondaire.

Beaucoup de travaux ont cherché à effectuer le filtrage des données "au vol", c'est à dire en suivant le débit du disque au niveau de l'octet. Il en résulte qu'on impose au filtre des performances très élevées pour permettre les filtrages les plus complexes, alors que de telles opérations ne représentent qu'une part extrêmement faible de l'activité du filtre. On construit donc des filtres très puissants qui passent l'essentiel de leur temps à réaliser des opérations élémentaires telles que transmission de données sans analyse, ou recherche de la fin d'un ensemble de données. Il est donc préférable de tabler sur l'existence d'une mémoire tampon entre le disque et le filtre: cette mémoire est, de toutes façons indispensable pour permettre les opérations de détection et de correction d'erreurs sur les données lues sur le disque. Il est également possible d'exécuter le filtrage en deux étapes: une première étape consistant en une première sélection des données, et la seconde,

exécutée par logiciel, achevant la sélection [ROH 81]: on compte donc sur la réduction du flux de données permise par la première étape pour donner le temps nécessaire à une sélection logicielle "au vol".

Dans un tel cas, il faut alors ajouter au temps de filtrage la durée de l'opération complémentaire nécessaire pour compléter la sélection, opération qui n'est nécessaire que sur les données présélectionnées par le filtre. Si T_s est la durée de cette opération pour un élément de donnée, et S le facteur de sélectivité du filtre, (rapport données lues / données transmises), la durée de la sélection sera alors:

$$T_c + N\ (T_f + \frac{T_s}{S})$$

L'opération complémentaire aura dans le pire cas la même complexité que l'opération de sélection complète, et dans le meilleur des cas, son exécution aura été préparée par la présélection, donc, T_s est inférieur ou égal à T_f. Le poids de l'opération complémentaire sera donc faible pour un facteur de sélectivité suffisamment grand. Tout le problème est donc de bien choisir la fonction à réaliser au niveau d'un filtre matériel, pour obtenir des performances suffisantes avec le minimum de matériel. Il est en fait possible que le meilleur compromis soit atteint avec des fonctions de filtrage matériel extrêmement élémentaires.

III.4.2. Intérêt du filtrage:

Le filtre nous paraît être une composante essentielle d'une machine bases de données pour les raisons suivantes:

- il permet de décharger le ou les processeurs de traitement d'une partie importante de leur travail, grâce à des automates mieux adaptés, et tout en profitant du transfert des données entre le contrôleur disque et la mémoire des processeurs de traitement. Si les filtres sont réalisés en VLSI, on peut exécuter le travail plus rapidement, et à l'aide d'opérateurs moins coûteux. Il est cependant nécessaire que le coût du filtre soit effectivement plus réduit que celui de processeurs classiques qui exécuteraient le même travail dans le même temps.
- il permet de réduire le flux de données entre le disque et la mémoire de traitement: un disque débitant à 10 Mbits/s consomme une bonne part de la bande passante des mémoires, au détriment des processeurs qui l'utilisent. Le filtre permet donc de n'accéder la mémoire que pour l'écriture des données utiles, c'est à dire en principe une faible partie des données lues sur disque (ceci nécessite évidemment que le filtre dispose de sa propre mémoire locale intégrée). Ceci est donc la condition nécessaire pour l'utilisation parallèle de plusieurs disques, dans une configuration multiprocesseurs. L'augmentation du volume unitaire des mémoires fait que les volumes de mémoire sont de moins en moins fractionnables, et que peu de parallélisme peut être obtenu au niveau de l'accès à la mémoire primaire. Une réduction du flux d'accès à la mémoire peut donc permettre un meilleur parallélisme entre les éléments qui la partagent.

- il permet de réduire le délai entre la lecture des données sur disque, et la disponibilité des données utiles en mémoire primaire: sans filtre, il faut transférer les données en mémoire, puis les sélectionner, et enfin les traiter. Le filtre permet de réduire ces délais en sélectionnant "au vol", avec un retard réduit: les données amenées en mémoire peuvent donc être immédiatement traitées, et les résultats intermédiaire entraîneront plus vite la génération de nouveaux accès disques. Ceci peut donc conduire à une amélioration des performances de la machine, en réduisant les risques de temps morts dans l'activité des disques et processeurs de traitement, en effet, on peut avoir un pipe line entre la production des résultats d'une opération de recherche, la consommation de ces résultats, et la génération de nouvelles requêtes. En cas de forte charge, les files d'attente de requêtes sur les disques risquent de s'allonger, ce qui peut être mis à profit pour optimiser les déplacements de bras par ordonnancement dès requêtes [DEN 67].
- il permet d'améliorer l'utilisation de la mémoire primaire, puisque celle ci ne reçoit plus que des données utiles, et n'est pas non plus accédée pour l'exécution des programmes de recherche. La réduction des délais précédemment mentionnée permet de plus de réduire le temps de résidence des données, et donc d'augmenter encore l'efficacité de l'utilisation de la mémoire. Ceci est important, malgré la diminution spectaculaire du prix des mémoires, et peut être même grâce à cette diminution: en effet, les importants volumes de mémoire aujourd'hui disponibles font que l'on peut stocker des volumes de données importants vis à vis de la capacité des disques, et donc minimiser de façon significative les échanges avec le disque. Or, il reste évident qu'un tel gain sera d'autant plus important que cet espace mémoire sera mieux utilisé.

Ceci étant dit, la question de l'efficacité d'un filtre matériel reste ouverte, en particulier vis à vis de l'exécution logicielle d'un algorithme de filtrage [SCH 85].

En fait, il est clair qu'un filtrage matériel est plus rapide qu'un filtrage logiciel, cependant, certains considèrent que l'effort que l'on consacre à la réalisation d'un filtre est tel, et les temps de conception si longs, qu'au bout du compte, on voit apparaître des microprocesseurs de puissance suffisante pour effectuer le même travail avec les mêmes performances [BOR 83]: ceci pose donc directement la question de l'intérêt des processeurs (ou opérateurs) spécialisés qui est au coeur de cette thèse.

Tout d'abord, il faut remarquer que, à savoir faire et moyens équivalents, il n'y a aucune raison pour que le temps et le coût de conception d'un opérateur spécialisé soit supérieur à celui d'un processeur banalisé. Ce dernier bénéficie d'un marché plus étendu justifiant des investissements plus importants; cependant, les progrès des techniques de CAO de la VLSI (compilateurs de silicium [ANC 83]), de réalisation des circuits (circuits multi projets [ANC 81]), et de test (microscopie électronique [COU 82]), permettent d'envisager de réaliser des circuits intégrés pour des applications bien précises. Nous considérerons donc la conception d'un filtre dans la perspective de sa réalisation en VLSI.

Il est alors intéressant de chercher à concevoir un circuit d'architecture classique (comme nous l'avons fait dans OPALE [IAN 85]), pour bénéficier d'outils standard, plutôt que de rechercher une architecture originale, même si elle est plus efficace.

Une autre approche est de rechercher un opérateur adaptable à de larges catégories de situations: filtrage sur un disque, en mémoire primaire, sur un réseau local, etc... Dans SCHUSS [GON 84], le choix a été de concevoir un processeur adaptable par microprogrammation a différents cas d'utilisation: ce processeur est, d'après ses concepteurs, plus efficace qu'un filtrage logiciel sur MC 68020.

Le deuxième point est que l'on n'a pas encore identifié de façon précise les goulots d'étranglement d'un SGBD. On ne peut donc pas encore être sûr que la sélection des informations constitue bien l'endroit où il faut mettre du matériel. Sur ce point, les mesures effectuées dans le cadre du projet VERSO avec un filtre logiciel [SCH 87] montrent que la décentralisation du filtrage n'est réellement intéressante que si le processeur de filtrage est considérablement plus rapide que le processeur central.

Le gain essentiel est alors de pouvoir travailler au rythme du disque, et donc d'utiliser ce dernier de manière optimale. Ceci nécessite également que l'utilisation du système permette une utilisation effective du disque et du filtre, autrement dit, que ceux ci ne passent pas l'essentiel de leur temps à attendre des requêtes. Autre argument en faveur du filtre: si celui ci a un coût du même ordre de grandeur qu'un processeur banalisé (ou si possible inférieur), il n'y a aucune raison de ne pas l'utiliser de manière préférentielle. Il est important de bien déterminer quelles fonctionnalités de la base de données doivent être déportées au niveau du disque (donc sur un filtre, ou à proximité immédiate du filtre). En particulier, l'introduction de possibilités de déduction, ou du traitement des connaissances, peut entraîner une complexité accrue des opérations de sélection, puisque celles ci peuvent se ramener à des unifications: ceci fera l'objet des prochains chapitres.

Par ailleurs, la possibilité de disposer d'espaces mémoire importants peut entraîner des recherches longues, même au niveau des mémoires primaires: dans la mesure où le filtre s'avère être un opérateur efficace, et où il peut être intégré, il serait alors intéressant de pouvoir l'utiliser également au niveau des recherches en mémoire primaire.

III.4.3. Filtrage et indexation:

Le filtrage permet un accès associatif à de gros volumes d'information. Cependant, il est également possible d'accéder directement à des informations par le biais d'un système d'indexation, si bien que l'on peut se poser la question de l'utilité réelle du filtrage.

Il nous semble que, pour plusieurs raisons, le filtrage et l'indexation sont complémentaires:

- D'une part, l'indexation ne permet pas (ou permet difficilement) d'accéder associativement à plus d'un champ d'une relation. Des techniques d'indexation multicritères existent, mais elles sont complexe à implémenter et ont pour effet non de sélectionner une donnée précise, mais de restreindre le domaine de recherche à un ensemble de données limité. La suite de la recherche est donc susceptible de nécessiter un système de filtrage.

- D'autre part, l'indexation nécessite un volume de données d'autant plus important que la sélection recherchée est plus précise. L'accès direct à une information va donc impliquer la consultation de volumes de données qui peuvent difficilement être conservés en totalité en mémoire primaire. De plus, cette consultation peut demander un traitement important. L'effet du filtrage est donc de permettre une recherche rapide sur des volumes de données limités (par exemple une piste de disque). L'indexation sera donc moins fine, les index pourront être moins encombrants, et tenir en mémoire. De plus, il est possible d'utiliser des techniques de filtrage pour l'accès aux index [PLA 84].

On va donc considérer que l'accès aux données peut se faire en deux étapes:

- L'indexation, qui permet de restreindre la recherche à un nombre limité de blocs de données contiguës (pistes), et qui peut s'effectuer par accès à des données résidant en mémoire primaire,
- Le filtrage, qui permet de sélectionner une ou plusieurs solutions dans les blocs de données sélectionnés.

Si les blocs de données sélectionnés par l'indexation appartiennent à des unités distinctes, les différents filtrages pourront s'exécuter en parallèle.

Dès que la recherche admet plusieurs solutions, le filtrage conduit à un accès plus rapide: ainsi, toutes les solutions contenues dans une piste sont obtenues en un accès disque (lecture d'une piste = 30 ms de délai + 16 ms d'accès = 46 ms), ce qui est plus rapide que plusieurs lectures consécutives de la même piste (deux lectures = 2 x 30 ms de délai + 2 ms d'accès= 62ms).

Par contre, l'apport de l'indexation est de permettre de répartir un ensemble de requêtes sur des zones distinctes du disque. Ceci permet de diminuer le nombre de requêtes par piste, et donc de réduire les performances requises pour le filtre (volume des tables, rapidité).

III.5. ARCHITECTURES PARALLELES:

Nous avons déjà vu pour quelles raisons une architecture répartie de la machine de bases de données était souhaitable: d'une part la recherche de performances, et d'autre part, le besoin de tolérance aux pannes. Sur ce dernier point, les travaux précédemment accomplis dans le domaine du maintien de cohérence dans les BD réparties en présence de panne [POS 83] permettent d'espérer atteindre cet objectif.

Par ailleurs, le faible coût des processeurs fait qu'il devient intéressant de répartir un traitement complexe sur un ensemble de microprocesseurs, d'opérateurs matériels et de mémoires secondaires fonctionnant en parallèle.

Une requête sur une base de données se compose le plus souvent d'une suite d'opérations sur un flux de données, le résultat d'une opération étant réinjecté dans l'opération suivante. On peut donc imaginer que des éléments distincts (filtres, disques, processeurs, mémoires primaires) peuvent être interconnectés dynamiquement pour former des réseaux de résolution de requêtes de complexité quelconque. Chaque élément

ferait donc partie d'un pool de ressources banalisées qui pourraient être sélectionnées pour s'interconnecter avec un partenaire arbitraire (Fig. III.2).

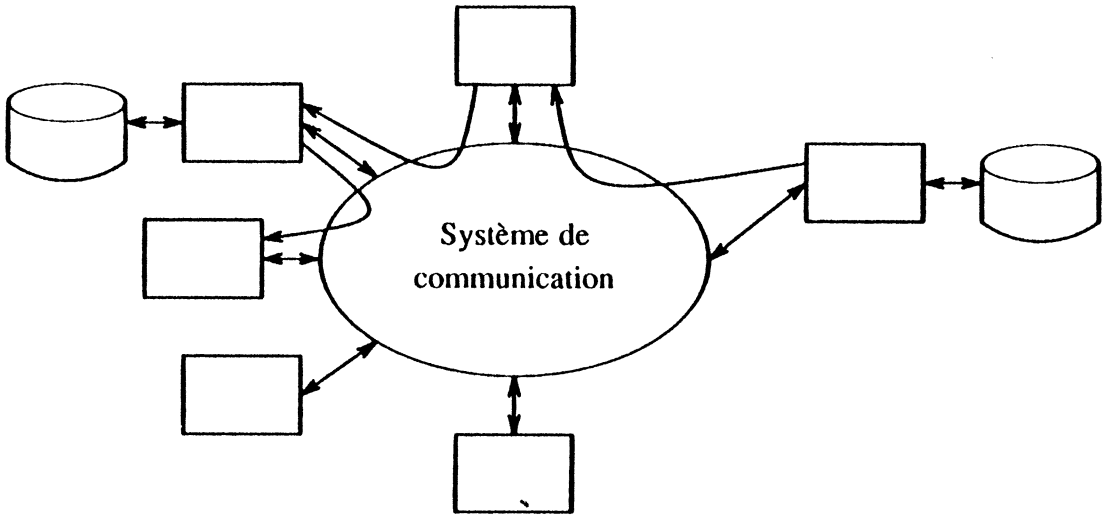


Figure III.2.

Une telle architecture serait sans doute optimale, par la totale banalisation des ressources, qui permet à la fois d'assurer la tolérance aux pannes, et une meilleure gestion des ressources. Cependant, il n'existe pas, à l'heure actuelle, de dispositif d'interconnexion qui permettrait, de façon relativement économique, une communication à haut débit entre de nombreux éléments interconnectables deux à deux. Sa gestion logicielle serait de plus assez complexe.

Il semble donc nécessaire de conserver, en particulier, l'association filtre-mémoire secondaire, et d'interconnecter, d'une part des éléments de mémoire comportant un filtre, d'autre part des éléments de traitement (microprocesseurs + mémoire), et enfin des processeurs de contrôle qui assurent la connexion avec les processeurs utilisateurs, et se chargent de la traduction des requêtes et de la gestion de l'ensemble. Cette solution présente évidemment les inconvénients qu'une panne d'un filtre empêche l'accès au disque correspondant, et d'autre part que les filtres peuvent être sous utilisés, dans la mesure où le disque auquel ils sont associés n'est pas forcément accédé en permanence. Le transfert lui même ne représente en outre qu'une petite partie de l'activité d'un disque, et l'intervalle entre deux transfert (positionnement des têtes) sera en grande partie une période d'inactivité pour le filtre. Cependant, l'intégration en VLSI du filtre peut le rendre suffisamment peu coûteux pour que cet inconvénient soit peu significatif.

III.6. LE PROJET OPALE:

Nous n'avons pas, dans les paragraphes précédents, évoqué plus avant les architectures de type matriciel, permettant le traitement de données précédemment amenées en mémoire. Ce type d'architecture pose, selon nous, le problème de la taille de tableau effectivement réalisable, alors que la capacité de stockage en mémoire primaire augmente

considérablement. Il peut donc être plus intéressant de traiter les données en mémoire primaire par filtrage.

Nous pensons cependant avoir montré que, sur le seul plan de l'architecture, toutes les questions sont loin d'être résolues et qu'il reste encore beaucoup de voies de recherche.

De plus l'évolution des besoins entraîne sans cesse l'apparition de nouveaux problèmes. Le développement des réseaux locaux a, ainsi posé quelques problèmes, mais on voit aussi apparaître le besoin de traiter des données complexes (telles que des textes), ou encombrantes (sons, images...). L'intelligence artificielle apporte également de nouveaux besoins et de nouveaux types de traitement, mais nous reviendrons sur ce point dans le prochain chapitre.

Dans le projet Opale (Organisation Parallèle Associative et Logique), nous nous sommes intéressés à l'utilisation de la logique des prédicats (Prolog), comme modèle de données et langage d'accès (*). Outre l'utilisation de la logique, les deux thèmes d'étude fondamentaux d'Opale (dont les mots clés apparaissent d'ailleurs dans le développement du sigle du projet) sont l'implémentation matérielle de l'accès associatif aux données en mémoire secondaire (filtrage), et le parallélisme dans l'accès aux données, dans la mesure où elles sont réparties sur plusieurs unités de disques.

Les objectifs sont les suivants:

- Performances: ceci passe en particulier par l'exploitation du parallélisme, d'une part dans l'accès aux disques, lorsqu'une requête porte sur plusieurs unités, et d'autre part par répartition du traitement sur un ensemble de processeurs.
- Souplesse: la conception d'Opale doit permettre de réaliser des configurations d'importance très variable, en fonction des volumes de mémoire secondaire et des performances requises. En particulier, pour des performances élevées, il est possible d'opter pour la répartition d'une base de données sur un grand nombre de petites unités plutôt que sur des unités plus importantes et moins nombreuses. Ceci conduit donc à rechercher la modularité de l'architecture.
- Tolérance aux pannes: un degré élevé de tolérance aux pannes pose des problèmes importants au niveau logiciel (autodiagnotic, isolement des éléments défaillants, redondances dans la base, journalisations, procédures de reprise, etc...). De tels problèmes sont pour l'instant hors du cadre de cette étude. Cependant, il est important que l'architecture d'une machine bases de données ne comporte pas (ou comporte peu) de points critiques sur le plan du matériel: le choix se porte donc naturellement vers une architecture répartie à base d'éléments banalisés. Il est alors possible d'introduire des éléments redondants pour permettre un fonctionnement dégradé en cas de panne d'un élément (avec reprise plus ou moins laborieuse selon le cas).

---

* Sur une idée de Monsieur Jean Rohmer, du centre de recherches Bull de Louveciennes.

III.6.1. Architecture:

Nous n'avons pas, dans le cadre de cette étude retenu l'architecture du paragraphe précédent. Celle ci présente en effet un nombre important de problèmes qu'il ne semble pas réaliste de vouloir aborder de front. Nous avons donc défini l'architecture représentée sur la figure III.3.

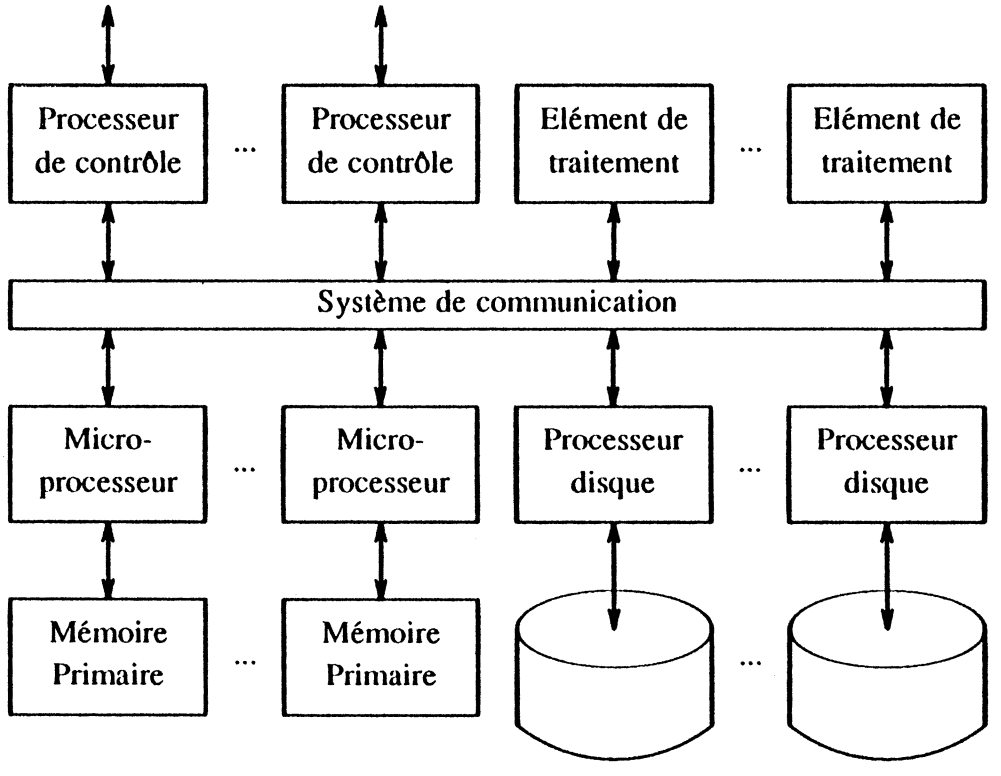


Figure III.3.

L'ensemble des éléments de la machine peut être conçu autour d'un nombre limité de composants standardisés: cartes mémoire, processeurs, auxquels s'ajouteront quelques opérateurs spécifiques (filtre).

III.6.1.1. Les processeurs disque:

Ces processeurs sont associés à chaque unité de disque, cependant, le choix, plus économique mais moins performant, de faire gérer plusieurs disques par un même processeur reste possible. Une amélioration sur le plan de la tolérance aux pannes consisterait aussi à partager deux disques entre deux processeurs (Figure III.4): sur le plan logiciel, chaque processeur prend en charge un seul disque, et ne s'occupe de l'autre qu'en cas de défaillance de l'autre processeur (une telle solution est envisagée en particulier dans la machine Dorsal 32 [ARM 83]). Les processeurs disques assurent des fonctions de bas niveau dans l'accès au disque, ainsi que la sélection des données par filtrage: ce dernier point sera détaillé dans le chapitre sur l'unification (chapitre VI).

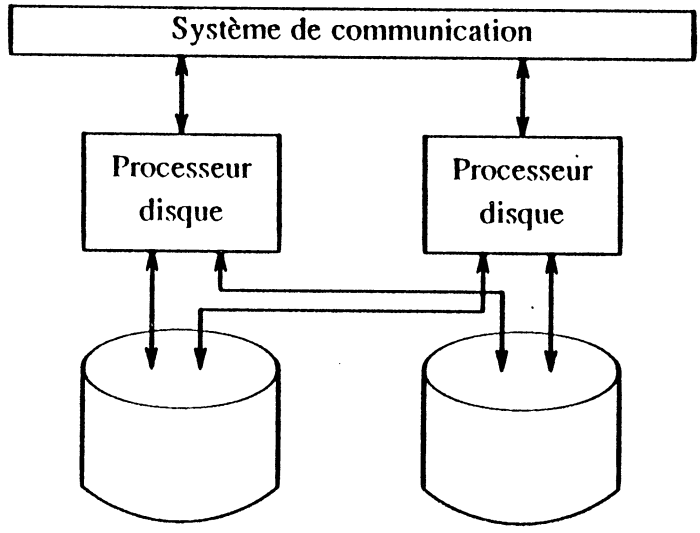


Figure III.4.

Le principe retenu au niveau du filtrage est d'avoir un tampon d'au moins deux secteurs entre le disque et le filtre: le filtre travaille donc à tout instant sur des données sans erreur de lecture (les contrôles étant effectués en fin de lecture d'un secteur) pendant que le secteur suivant est en cours de lecture. La taille exacte du buffer devra être choisie de façon à permettre d'absorber des fluctuations éventuellement importantes du temps de traitement des données par le filtre.

III.6.1.2. Les éléments de mémoire primaire:

Ces éléments seront dotés d'une mémoire primaire importante, d'un processeur, et éventuellement d'un filtre. Ils permettront le stockage de données temporaires, ou d'informations dont le volume réduit et le taux d'accès élevé justifient qu'on les stocke en mémoire RAM pour une durée relativement longue (index, schéma de la base, certaines relations ou certains paquets de règles, ...). L'idée est de pouvoir effectuer sur de tels éléments des opérations similaires à celles qui sont effectuées sur le disque, y compris le filtrage. Un filtre matériel pourra donc être utilisé aussi à ce niveau, s'il s'avère plus performant qu'un algorithme logiciel même hors des contraintes liées au disque.

III.6.1.3. Les éléments de traitement:

Ils sont composés d'un processeur et de sa mémoire, et assurent l'enchaînement des opérations. Leur rôle sera plus précisément exposé au chapitre concernant la stratégie de recherche (chapitre V).

III.6.1.4. Les processeurs de contrôle:

Il s'agit d'un ensemble de processeurs (éventuellement réduit à un seul processeur) dont le rôle est d'assurer à la fois la communication avec l'extérieur (réseau local ou processeur hôte), le contrôle de l'ensemble de la machine (y compris éventuellement des procédures de diagnostic et reprise en cas de panne), et enfin un premier niveau de traitement de requêtes (traduction sous forme interne, vérification, lancement du traitement, récupération des résultats)

III.6.1.5. Le système de communication:

Le système de communication constitue le point le plus délicat de cette architecture. Pour des configurations importantes, il doit en effet permettre d'assurer un débit très élevé, compte tenu du nombre de disques, de leur débit, et de la sélectivité des filtrages (en admettant que le flux de données issu des accès disques constitue la majeure partie des échanges sur le réseau).

Si on considère que l'accès aux disques se fait par piste, on peut avoir pour chaque accès, un délai de positionnement des têtes d'une vingtaine de millisecondes, un délai rotationnel de 8,33 ms, et un temps de transfert de 16,6 ms. Le disque transfère donc en moyenne pendant environ 36% du temps au maximum. Si on considère que le filtrage sélectionne 10% des données, le débit effectif moyen pour 100 disques débitant à 10 Mégabits par seconde est de:

100 x 10 x 0.1 x 0.36 = 36 Mégabits par seconde. Cependant, ce débit est à transmettre sur des distances relativement faibles, et le transfert peut se faire en parallèle.

En outre, le système de communication devrait:

- être modulaire, pour s'adapter aussi bien à de petites configurations qu'à de très grosses.
- être tolérant aux pannes et reconfigurable.
- permettre à un grand nombre d'éléments de prendre de manière autonome l'initiative de la communication.

La communication devra se faire par échange de messages. Il est en fait difficile, dans l'état d'avancement actuel du projet, de caractériser et évaluer la charge du système de communication afin d'effectuer un choix définitif. Une étape indispensable de la réalisation consiste donc à réaliser une première maquette de configuration restreinte, permettant d'effectuer les mesures nécessaires à la suite de l'étude. Pour une telle réalisation , la machine SM 90 [FIN 83] semble fournir une structure tout à fait adaptée. D'autres éléments sur le système de communication seront aussi présentés dans le chapitre sur la stratégie de recherche

III.6.2. La solution SM 90:

Conçue par le CNET, la SM 90 est une machine multi-microprocesseurs, de structure modulaire, et conçue pour permettre la tolérance aux pannes. Elle répond donc par elle même à un certain nombre de nos besoins. On en trouvera en annexe une description succinte.

Elle ne permet que des configurations relativement limitée par rapport à nos objectifs, puisqu'on peut avoir au plus 8 modules de traitement (MT) (pouvant prendre l'initiative de transferts sur le bus de communication), et 16 modules d'échanges (ME). Elle permet cependant de réaliser des configurations comportant plusieurs disques et éléments de traitement, et donc mettant en oeuvre un parallélisme réel. On peut donc d'une part aboutir à une véritable expérimentation matérielle d'Opale, et d'autre part, concevoir de cette façon un prototype applicable à un certain nombre d'applications: machine bases de données pour mini-ordinateurs, ou serveur bases de données pour réseau local de micro-ordinateurs, ou enfin poste de travail. Il est aussi possible d'envisager que des configurations plus importantes puissent être réalisées par interconnexion de telles machines.

Les avantages de la solution SM 90 seraient les suivants:

- résolution d'une partie des problèmes matériels, grâce à l'architecture multiprocesseur de la machine, et la possibilité de disposer d'éléments standards tels que processeurs, mémoires, etc... On peut donc se concentrer sur ce qui constitue l'originalité d'Opale: le filtre et la stratégie de recherche.
- existence de logiciels de base pour le support de la réalisation: UNIX, moniteur "MON 90", CHORUS [BAC 80].
- Compatibilité avec des outils de développement logiciel.
- Possibilité d'une première implémentation en logiciel des algorithmes de filtrage, en attendant leur implémentation matérielle.

Opale peut être réalisée sur la SM 90 de la façon suivante:

- *Les processeurs de contrôle:* ils sont connectés comme MT et peuvent donc prendre l'initiative des transferts sur le bus de communication. Ils comportent un système de communication avec l'extérieur qui devrait normalement se connecter sur leur bus local. On peut cependant admettre de les connecter par l'intermédiaire du bus de communication pour récupérer des éléments matériels préexistants.
- *Les éléments de traitement:* Ils sont eux aussi connectés comme MT et comportent une mémoire privée. La communication entre éléments de traitement et processeurs de contrôle peut se faire soit par des mémoires partagées au niveau de chaque module (mémoires locales à double accès), soit par une mémoire commune.
- *Les éléments de mémoire primaire:* Ils sont connectés comme ME, et ne peuvent donc pas prendre l'initiative des transferts sur le bus de communication. Ils communiquent avec les éléments de traitement et processeurs de contrôle par l'intermédiaire d'une mémoire d'échange.

- *Les processeurs disque:* ils sont également connectés comme ME. Ils communiquent par l'intermédiaire d'une mémoire d'échanges où ils reçoivent leurs requêtes et déposent leurs résultats. Les éléments de traitement viennent ensuite récupérer ces résultats pour les traiter. La mémoire d'échange des processeurs disque peut donc tout aussi bien assurer le stockage temporaire des résultats en attente de traitement.

Cette solution a pour inconvénient de ne pas permettre de communication directe entre processeurs disques et éléments de mémoire primaire. Cet inconvénient ne semble cependant pas rédhibitoire, une telle communication ne semblant pas rentrer dans le schéma de fonctionnement normal d'Opale. Cette architecture permettrait au maximum de connecter une dizaine de disques, compte tenu des limites imposées par le nombre maximal d'unités d'échange et de modules de traitement.

L'architecture de la SM 90 est de toutes façons mal adaptée à des configurations importantes, de par le mécanisme de communication entre les ME et les MT: le MT doit aller lire ou écrire des informations directement dans la mémoire d'échange. Le ME ne peut qu'envoyer un ,signal vers l'ensemble des MT, pour les prévenir qu'il y a quelque chose à lire. Chaque MT (concerné ou non) devra alors tester chacun des ME (ou au moins ceux avec lesquels il a quelque chose a faire), pour voir s'il y a un message qui le concerne. Pour un nombre élevé de MT et de ME, c'est donc un mécanisme assez lourd.

On peut toutefois le rendre plus léger, en adoptant la politique suivante: chaque MT est connecté logiquement à un ou plusieurs ME desquels il attend des résultats. Lorsqu'il a terminé le traitement d'un résultat, il va donc de sa propre initiative interroger la mémoire d'échange des ME pour trouver un nouveau résultat à traiter. Cependant, s'il n'y en a pas, le risque est alors de le voir cycler sur le test des différents ME, en engendrant un traffic de données important sur le SM bus.

III.6.3. Conclusion:

Dans ce bref aperçu du projet Opale, un certain nombre de problèmes ont été soulevés. Le choix du langage Prolog comme langage de base de données sera explicité et discuté au prochain chapitre. Le problème qui se pose est alors de dégager des moyens (logiciels et matériels) pour une interprétation efficace de Prolog dans un contexte de bases de données résidant pour une bonne part en mémoire secondaire. Ceci passe par la résolution de deux problèmes:

- La définition d'une stratégie de recherche adaptée, et permettant l'exploitation du parallélisme.
- L'implémentation de l'unification, qui joue un rôle de sélection des données. Cette unification, exécutable partiellement en matériel, sera effectuée au niveau des processeurs disques. L'algorithme proposé permet d'unifier un ensemble de buts en un seul accès au disque. En cela, il nécessite une stratégie de recherche permettant de dégager de tels ensembles de buts. Les deux problèmes sont donc très liés, et nous choisirons ici d'exposer en premier la stratégie de recherche, au chapitre V, puis l'algorithme d'unification au chapitre VI.

Enfin, le chapitre VII sera consacré à des mesures sur Prolog, et à une généralisation des résultats en vue de la conception d'une machine Prolog.

Chapitre IV

PROLOG ET BASES DE DONNEES

IV.1. INTRODUCTION:

Parallèlement à l'évolution matérielle des moyens de traitement des données, on voit se dessiner une évolution de la demande: alors que le relationnel commence à être utilisé dans le monde industriel, les chercheurs commencent à en entrevoir les limites et se tournent vers de nouveaux problèmes. On commence à ressentir le besoin de bases de données "déductives". On se pose également le problème des contraintes d'intégrité, permettant de détecter certaines incohérences sémantiques dans les données stockées.

Une nouvelle demande provient en outre de l'intelligence artificielle, avec, en particulier les systèmes experts: là aussi, il s'agit de pouvoir stocker et traiter, non seulement des faits, mais aussi un ensemble de règles constituant des "connaissances" (bases de connaissances).

Dans ces domaines, on est d'une façon ou d'une autre amené à dépasser le domaine du relationnel pour atteindre le domaine de la logique, et la programmation en logique (Prolog) apparaît comme une voie privilégiée. En effet, la programmation en logique constitue non seulement un outil bien adapté pour la programmation dans le domaine de l'intelligence artificielle, mais elle est en plus - de par la logique des prédicats dont elle dérive - entièrement compatible avec le modèle relationnel, par rapport auquel elle présente nombre d'avantages.

Avec l'extension de l'utilisation des bases de données à de nombreux domaines, on est également amené à vouloir traiter des données de types et de structures très diverses: sons et images, textes, arborescences, hiérarchies, etc... Or, alors que le modèle relationnel semble limité à des données atomiques, Prolog, à quelques adaptations près, est capable de traiter une grande variété de types et de structures de données. Il n'est cependant pas le seul type de langage candidat à de telles applications (Cf langages orientés objet, par exemple).

IV.2. BASES DE DONNEES ET LOGIQUE:

La liaison entre les bases de données et la logique n'est pas neuve [GAL 81]. Ainsi, les SGBD relationnels peuvent utiliser des langages prédicatifs, basés sur la logique mathématique. La logique intervient aussi dans le contrôle de l'intégrité des bases de données, en permettant l'expression de règles de cohérence.

Un deuxième point de rencontre entre logique et bases de données se situe au niveau des langages d'interrogation. En particulier, beaucoup de travaux sur l'interrogation de bases de données en langue naturelle passent par la logique [WAR 81], celle ci se trouvant, semble-t-il, au point de rencontre entre les langues naturelles et les bases de données.

Cependant, c'est avec l'émergence de nouvelles directions de recherche que la connexion entre bases de données et logique a commencé à se développer. Les bases de données déductives ont pour objectif une exploitation maximale de l'information contenue dans les données stockées, en permettant, à l'aide de règles de déduction, de déduire de nouveaux faits non explicitement stockés dans la base. Il s'agirait donc plutôt d'une extension des bases de données traditionnelles par le biais de la logique permettant d'enrichir les modes d'accès aux données et d'exploiter leur sémantique tout en limitant les redondances d'informations.

On peut citer BD-Gen [NIC 83] comme une première approche de base de données déductive; cependant, les faits déduits y sont automatiquement générés et stockés dans la base lors de toute modification, et on peut craindre que ce principe ne soit plus applicable lorsque l'ensemble de règles de déduction présente une certaine complexité: l'ensemble de faits déduits peut devenir excessivement important, et il est alors préférable de pouvoir effectuer la déduction lors de l'interrogation.

Une autre approche, suivie dans OPALE, consiste plutôt à construire un SGBD autour de la logique (par exemple Prolog), la logique jouant le rôle d'un langage d'implémentation [GAL 83].

Le domaine des systèmes experts apporte nous l'avons vu, le problème de la représentation et du stockage des connaissances. Une base de connaissances peut être vue comme un ensemble de faits et de règles. Les faits sont des données analogues aux n-uplets du relationnel, alors que les règles permettent, à partir d'un ensemble de faits (les prémices), d'arriver à une conclusion, c'est à dire de vérifier une hypothèse ou un ensemble d'hypothèses.

La logique des prédicats n'est pas la seule approche des bases de connaissances, puisqu'on trouve aussi en particulier les frames, ou les réseaux sémantiques. De plus, au sein de la logique, Prolog n'est pas la seule approche.

Il existe en effet deux démarches opposées: l'une consiste à partir de faits connus (les prémices) pour arriver aux conclusions (chaînage avant). L'autre consiste à partir d'une conclusion (hypothèse), et à chercher à la démontrer à l'aide des faits connus (chaînage arrière: c'est la démarche de Prolog). Les deux démarches ont sans doute des efficacités différentes suivant les circonstances, sans que l'une puisse être considérée comme globalement supérieure à l'autre. Le fait est toutefois que seul le chaînage arrière a, avec Prolog, donné le jour à un langage de programmation universel jusqu'à aujourd'hui. L'idéal serait sans doute de pouvoir choisir une démarche plutôt qu'une autre en fonction du problème, mais ceci représente en soi un sujet de recherche à part entière. Dans [ROH 85], J. Rohmer et R. Lescoeur présentent une méthode permettant de ramener le chaînage arrière a du chaînage avant, ceci dans le cas de relations pures (pas de règles dans la base de données).

Par ailleurs, Prolog représente une restriction par rapport à la logique du premier ordre: celle des clauses de Horn. Il ne semble pas, en fait, que cette restriction entrave beaucoup la puissance du langage; elle est, par contre, apparue indispensable pour définir une méthode d'interprétation efficace. Enfin, Prolog correspond aussi à un modèle

d'interprétation nécessaire pour définir la sémantique opérationnelle du langage. Or, cette sémantique est souvent sous-jacente, c'est à dire que le programmeur n'a pas forcément besoin d'en prendre conscience pour écrire des programmes corrects (sinon performants). Il existe aussi des cas où la sémantique du programme ne dépend pas d'une méthode d'interprétation: on considère que c'est le cas avec les bases de données, où l'on s'intéresse à des ensembles de résultats, et non à leur ordre. Il est donc possible de considérer Prolog en prenant une certaine distance par rapport à sa méthode d'interprétation.

Nous considérerons ici les bases de données logiques à travers l'approche Prolog, celle ci ayant en fait comme avantage essentiel d'établir un lien direct entre bases de données et langage de programmation. Il est ainsi possible de s'assurer une interface tout à fait naturelle entre ces deux mondes, et d'enrichir éventuellement la base de données, en associant éventuellement des actions sémantiques à certaines données.

IV.3. PROLOG ET MODELE RELATIONNEL:

L'utilisation de Prolog pour les bases de données semble assez naturelle. Beaucoup de projets s'intéressent à l'interface Prolog - BD relationnelles [CHA 82, MIN 82]. De petits ensembles de données peuvent également être gérés directement en Prolog [WAR 81]. Enfin, dans le projet japonais de 5ème génération, la conception d'une machine bases de données relationnelle est considérée comme une première approche d'une machine bases de connaissances [MUK 83]. On peut donc distinguer deux approches: l'interface Prolog - relationnel, visant essentiellement, semble-t-il, à exploiter le savoir faire acquis dans le domaine du relationnel; et l'utilisation directe de Prolog, à la fois comme modèle de données et comme langage d'implémentation.

On peut même se poser le problème de l'interface langages relationnels - Prolog, visant à exploiter la haute qualité de l'interface homme - machine que présentent les premiers pour l'accès à des bases de données fournissant les capacités de déduction de Prolog.

Il est assez facile de représenter une base de données relationnelle sous forme de paquets de clauses Prolog: chaque n-uplet d'une relation R(a,b,c) peut en effet s'écrire sous forme d'une clause: r(a,b,c).

A partir de là, on trouve aisément l'équivalent direct des opérations relationnelles:

Exemples:

li l. Intersection:r1(X,Y) :- r2(X,Y), r3(X,Y). Union:r1(X,Y) :- r2(X,Y). r1(X,Y) :- r3(X,Y). Jointure:r1(X,Y) :- r2(X,Z), r3(Z,Y). Selection:r1(X,Y,a) :- r4(X,Y,a). Projection:r1(X,Y) :- r4(X,Y,Z). Produit:r1(X,Y) :- r5(X), r6(Y). etc...

Il est également possible de combiner plusieurs opérateurs relationnels en une clause Prolog assez simple:

Exemple:

r(X,Y) :- r1(a,X,Z), r2(Z,Y,T).

Cette clause est une combinaison de sélection, jointure et projection.

Par ailleurs, Prolog présente d'autres avantages sur le relationnel:

- Manipulation de données non atomiques: les termes, de structure arborescente, qui peuvent représenter des listes.
- Définition de relations dérivées, ou déduites, voire de relations définies en partie explicitement, et en partie implicitement,
- Possibilité de représenter de façon simple des opérations telles que la fermeture transitive d'une relation (Cf l'exemple classique de la clause *ancêtre* au paragraphe IV.5.).
- Typage dynamique des données,
- Possibilité d'insertion d'actions sémantiques dans les données...

En fait l'ensemble des possibilités nouvelles introduites par Prolog mériterait à lui seul une étude particulière. Aussi, considérer le problème Prolog - BD au niveau de l'interfaçage entre un interpréteur Prolog et un SGBD relationnel nous semble très restrictif par rapport à cet ensemble de possibilités, et de plus, il introduit un niveau supplémentaire de traduction et/ou d'interprétation entre l'utilisateur et la base de données. Cela revient également à introduire une séparation assez artificielle entre le "monde Prolog", et le "monde bases de données", alors que ces mondes sont en fait très proches. C'est pourquoi, nous choisissons de considérer Prolog à la fois comme modèle de données, langage d'implémentation du SGBD, et premier niveau de langage d'accès.

Prolog constitue un modèle de bases de données au moins équivalent au relationnel [GAL 81], et d'autre part, notre hypothèse est qu'il est possible, en s'inspirant des travaux effectués dans le cadre des BD relationnelles, de définir des moyens (logiciels et matériels) pour une implémentation efficace de Prolog en tant que langage de bases de données. C'est pourquoi, notre choix, dans Opale, a été de se tourner vers l'utilisation directe de Prolog au niveau de la base de données.

Dans la suite de ce chapitre, nous tacherons de dégager les particularités, d'une part de Prolog par rapport aux bases de données relationnelles, et d'autre part des bases de données par rapport aux programmes, pour l'implémentation de Prolog: notre hypothèse est ici, en particulier, qu'une utilisation particulière de Prolog requiert des mécanismes d'implémentation spécifiques, et donc qu'une machine Prolog en tant que machine réellement universelle reste pour un temps inconcevable.

IV.4. PROLOG - BASES DE DONNEES:

IV.4.1. Spécificité vis à vis des bases de données:

Nous avons vu brièvement au paragraphe précédent qu'il était facile de traduire en Prolog les données et les opérations relationnelles. Cependant, l'introduction des nouvelles fonctionnalités de Prolog fait qu'il n'est pas possible d'utiliser strictement les

mêmes mécanismes.

Dans une base de données Prolog, on trouvera des ensembles de faits, ou axiomes, c'est à dire des clauses dont la queue est vide: ceci correspond aux relations. Par contre, on trouvera également des règles, c'est à dire des clauses dont la queue est non vide. Il est souvent tentant de considérer séparément les faits et les règles, et de prévoir des mécanismes différents, permettant, par résolution des règles, de parvenir à des ensembles d'accès sur les faits [CHA 82], résolus par les méthodes traditionnelles d'accès aux BD relationnelles.

Par ailleurs, dans une base de connaissances, il semble que le nombre de règles constituant un paquet de clauses puisse être très élevé. Les paquets de clause sont éventuellement trop importants pour tenir en mémoire primaire, et il est donc nécessaire, dans ce cas aussi, de disposer de mécanismes d'accès et de sélection performants (indexation, filtrage). Ainsi, un programme de dérivation formelle très élémentaire [COL 83] comporte déjà 9 règles de dérivation et 22 règles de simplification: un système de calcul formel plus complet peut donc comporter des ensembles de règles beaucoup plus importants, et on peut considérer qu'il s'agit bien là d'un type de base de connaissances. Le même article présente aussi un exemple (très incomplet) de programme de conjugaison du français. Pour les seuls verbes *avoir*, *être*, *aller*, plus quelques verbes en *ir*, on compte 9 règles déterminant la forme du verbe, et 9 règles de conjugaison du présent: un programme complet de traitement du français amène donc rapidement à des ensembles de règles très importants. On peut penser que le développement des systèmes experts, et leur application à des domaines de plus en plus vastes fera apparaître des bases de connaissances de taille considérable, la limite résidant en fin de compte dans la capacité d'accéder à de telles bases de clauses. Les règles peuvent par ailleurs se trouver mêlées aux faits, en constituant, par exemple, une généralisation d'un fait, qui peut être prouvé partiellement en extension et partiellement par dérivation à partir de règles.

Une autre particularité des bases de données Prolog est la possibilité d'avoir des variables dans la base de données, et ce, non seulement dans les règles, mais aussi dans les faits. Les variables ont alors valeur de quantificateur universel: r(a,X). signifie que la relation est vraie quelle que soit la valeur liée à X.

Cependant, ceci ne fournit pas directement l'ensemble des solutions en extension, c'est à dire que la question: :- r(a,b). sera évaluée à vrai, alors que la question: :- r(a,X). retournera une seule solution, avec X non lié. Il semble que ce genre de situation risque de donner lieu à des ambiguïtés, qui devraient alors être levées par des mécanismes complémentaires, permettant d'affecter un domaine de validité plus restreint à une telle variable, ou d'interdire certains types de requête.

Les variables peuvent également occuper la place d'arguments non pertinents pour certains cas, mais il n'est pas évident de pouvoir les utiliser pour dénoter, par exemple des arguments inconnus (notion de "valeur nulle" dans les SGBD relationnels). Enfin, les variables peuvent également être dépendantes, et exprimer des relations entre les arguments d'un terme: par exemple, la clause: r(X,X). signifie que r(X,Y) est vrai si ses deux arguments sont égaux (ou plutôt unifiables), et l'appel r(a,X) aura pour effet de lier X à

a. La présence de variables dans la base de données rend donc impossible l'utilisation directe des mécanismes de sélection des BD relationnelles: ce point sera développé dans la suite.

Les données de Prolog sont typées dynamiquement, c'est à dire qu'une variable peut être liée à une donnée de type quelconque, et qu'il est possible de traiter une donnée en fonction de son type. Donc, un champ d'une relation peut éventuellement recevoir n'importe quel type de données, à moins de contraintes externes supplémentaires.

Une base de données Prolog ne requiert pas, à proprement parler, de schéma ou de structure: les types sont dynamiques, et il est, en théorie, possible d'ajouter n'importe quelle clause, sans se référer à un schéma. En fait, un tel schéma reste nécessaire, mais plutôt vis à vis de la sécurité et de la détection d'erreur: il s'agit de fixer un ensemble de règles que doit vérifier toute modification de la base, de façon à permettre de détecter une erreur au niveau d'une insertion, par exemple. En effet, en Prolog, toute clause syntaxiquement correcte est réputée correcte, et on ne dispose pas d'un moyen de vérifier qu'une nouvelle clause appartient bien, par exemple, à un paquet de clauses préexistant. De même, il est nécessaire qu'une question portant, par exemple, sur une relation inexistante, soit détectée comme erronée, et non considérée comme correcte mais sans solution (ce qui serait le comportement normal de certains interpréteurs). Le schéma se confond donc avec un ensemble de règles de cohérence, utilisables autant au niveau des modifications qu'à celui des consultations.

De ces ensembles de différences notables entre Prolog et les bases de données relationnelles, il ressort que l'on ne pourra pas adopter directement les moyens de traitement des bases de données relationnelles, et qu'un travail d'adaptation est nécessaire. Par contre, nous préférerons rester dans le "monde Prolog", c'est à dire adopter le modèle de données de Prolog au niveau de la base de données, plutôt que de considérer Prolog comme un niveau hiérarchique supplémentaire entre l'utilisateur et une BD relationnelle.

IV.4.2. Spécificité vis à vis des programmes Prolog:

De même que Prolog revêt des aspects particuliers par rapport aux modèles de bases de données existants, on peut aussi dire que son utilisation dans un contexte de bases de données est très particulière par rapport aux programmes Prolog classiques.

D'une part, il existe une différence fondamentale quant à la nature et aux performances du support des données: la base de données est située le plus souvent sur disque, ce qui signifie de gros volumes de données, avec des temps d'accès relativement longs, des accès par blocs, et la quasi impossibilité de recourir à des structures de données chaînées. Par ailleurs, les temps d'accès à un secteur sont prédominants, par rapports au temps de transfert proprement dit: une fois le secteur voulu adressé, il n'est guère plus coûteux d'accéder par exemple à toute une piste.

D'autre part, d'autres différences importantes existent quant à la manière même d'utiliser le langage.

- le nombre d'alternatives des clauses sera beaucoup plus important dans un contexte de bases de données: des mesures faites sur un ensemble de programmes Prolog (Cf chapitre VII) montrent que le nombre moyen d'alternatives par clauses est de deux à trois. Par contre, dans une base de données même petite, le nombre d'alternatives par clauses variera de quelques dizaines à quelques millions. Ceci implique donc des représentations et des modes d'accès différents, et l'indexation des clauses devient quasiment indispensable.
- le taux d'échec de l'unification sera de même beaucoup plus élevé: de 30 à 60% pour l'ensemble de programmes précédemment mentionné. Dans une base de données, le taux d'échec sera en rapport avec la sélectivité des requêtes et la taille des paquets de clauses considérés: il sera donc le plus souvent supérieur à 90%, voire à 99%. L'importance relative de l'unification dans le processus d'interprétation va augmenter, car celui-ci fera de nombreuses tentatives d'unification avant qu'une unification réussie permette d'avancer dans le processus d'interprétation. Ceci va donc entraîner des techniques différentes pour l'implémentation de l'unification: nous y reviendrons plus loin.

Outre ces considérations, nous sommes amenés à faire un certain nombre d'hypothèses qui ne pourront être vérifiées que lorsque de véritables bases de données Prolog auront pu être constituées et exploitées:

- l'essentiel du volume reste représenté par les faits: on peut estimer que les faits (analogues aux relations) sont susceptibles de représenter des volumes beaucoup plus importants que les règles.
- l'ordre des résultats est le plus souvent indifférent: c'est généralement vrai dans le cas des bases de données, où les résultats finaux sont susceptibles d'être triés avant transmission à l'utilisateur.
- peu de prédicats évaluables comportant des effets de bord seront présents dans les règles: ces prédicats sont par exemple l'ajout ou la suppression de clauses, ou les entrées sorties. Ceci peut nécessiter des adaptations de l'interpréteur pour réduire ou supprimer la nécessité de tels prédicats: ainsi, l'ajout ou la suppression de clauses sont souvent les seuls moyens disponibles pour exécuter des fonctions agrégat (somme, moyenne, minimum, maximum), des comptages de solution, ou former la liste des solutions d'un prédicat, alors que quelques adaptations du langage permettraient de s'en dispenser sans pour autant créer de nouveaux effets de bord.
- la coupure est rare, ou peut être remplacée par des opérations plus adaptées: nous rediscuterons de cela dans ce chapitre.
- la complexité des clauses sera relativement réduite: on entend par là que le nombre de règles à appliquer successivement avant d'accéder à des faits sera le plus souvent réduit. L'importance de ceci est que lorsque des ensembles de règles sont complexes il peut être intéressant de les amener totalement en mémoire. On admet que ceci se produit essentiellement pour des ensembles de règles de taille réduite, et que cela peut être déterminé dès l'accès au paquet de clauses (présence d'un indicateur, par exemple).

- on suppose enfin que les dépendances entre les variables des en-têtes de clauses (occurences multiples d'une même variable) constituent rarement le critère de sélection des en-têtes de clauses: autrement dit, lorsqu'il existe des dépendances, on considère qu'il doit exister le plus souvent un autre critère de sélection (valeur constante). En fait, les dépendances servent le plus souvent à construire un résultat, plutôt qu'à sélectionner la clause à vérifier. En fait, ceci est vrai même dans le cas des programmes Prolog, comme nous le verrons au chapitre VII.

IV.5. EXEMPLES:

Dans ce paragraphe, nous allons donner quelques exemples d'utilisation de Prolog comme langage de base de données. Nous prendrons quelques exemples classiques des bases de données relationnelles, puis nous présenterons quelques exemples plus spécifiques de Prolog.

Le premier exemple est extrait de [NGU 82]: On considère une base de données contenant les deux relations suivantes:

```
etudiant (num_identification , nom , adresse)
inscription (num_identification , cours)
```

On veut répondre à la question: "*quels sont les étudiants qui habitent Grenoble et sont inscrits au cours de topographie?*".

En Prolog, cela pourra se traduire par:

```
reponse(N) :- etudiant(X, N, Grenoble), inscription(X, topographie).
```

On voit sur cet exemple que Prolog permet, grâce à un formalisme simple, une expression simple des opérations relationnelles. Par ailleurs, un optimiseur de requêtes sera éventuellement en mesure de déterminer qu'il y a de fortes chances pour qu'il y ait beaucoup plus d'étudiants habitant à Grenoble que d'étudiants inscrits au cours de topographie (même en se basant sur les rapports nombre d'étudiants/nombre de villes et nombre d'étudiants/nombre de cours); il pourra donc transformer la requête de façon à minimiser le nombre de résultats produits par le premier membre de la jointure:

```
reponse(N) :- inscription(X, topographie), etudiant(X, N, Grenoble).
```

Il faut également observer que la formulation Prolog est assez proche de l'expression de la requête en français. Des travaux sur la traduction en Prolog et l'optimisation de requêtes exprimées en langue naturelle ont été exécutés par D. Warren [WAR 81].

Par ailleurs, Prolog va permettre, par exemple, de généraliser une relation. Supposons par exemple qu'on ait la règle: "tout étudiant inscrit en maîtrise d'informatique est inscrit au cours de programmation". Ceci s'écrira en Prolog:

```
inscription(N, programmation) :- inscription(N, maîtrise_d_informatique).
```

On connaît aussi l'exemple de la clause "ancêtre" qui permet de réaliser la fermeture transitive d'une relation:

```
ancetre(X,Y) :- pere(X,Y).
ancetre(X,Y) :- pere(X,Z), ancetre(Z,Y).
```

Ce paquet de clauses exprime le fait qu'une personne X est ancêtre d'une personne Y si, soit X est père de Y, soit il existe une personne Z telle que X est père de Z et Z est ancêtre de Y.

Il est clair qu'une telle possibilité peut trouver de nombreuses applications: par exemple, en CAO, il peut être intéressant de savoir répondre à des questions du type: "quels sont les transistors dont la grille est connectée au drain du transistor X", sans pour autant avoir besoin de stocker dans la base tous les couples d'objets connectés entre eux, directement ou non. On peut également répondre à des questions bien plus complexes [MAL 83].

Un autre exemple est extrait de [COL 83]: il s'agit d'un extrait d'un petit programme de dérivation formelle, qui fait appel à un ensemble (très incomplet) de règles de dérivation et de simplification.

```
/*  règles de dérivation: derivee(A,B,C) signifie: C est dérivée de A par rapport à B */

derivee(X,X,1) :- !.
derivee(N,X,0) :- atom(N), !.
derivee(bin(O,U,V),X,bin(O,D1,D2)) :-
    op_add(O), !, derivee(U,X,D1), derivee(V,X,D2).
derivee (un(sub,U),X,un(sub,D1)) :- !,
    derivee(U,X,D1).
derivee(bin(mul,U,V),X,bin(add,bin(mul,U,D2),bin(mul,V,D1))) :- !,
    derivee(U,X,D1), derivee(V,X,D2).
derivee(bin(exp,U,N),X,bin(mul,N,bin(mul,D1,bin(exp,U,bin(sub,N,1))))) :-
    atom(N), !, derivee(U,X,D1).
derivee(un(sin,U),X,bin(mul,D1,un(cos,U))) :- !,
    derivee(U,X,D1).
derivee(un(cos,U),X,un(sub,bin(mul,D1,un(sin,U)))) :- !,
    derivee(U,X,D1).
derivee(X,Y,Z) :- write('Je ne sais pas faire!'), fail.

/*  règles de simplification pour les opérations binaires    */
/*  simp(op,X,Y,Z) signifie: op(X,Y) peut se simplifier en Z  */

simp(add,0,X,X).
simp(add,X,0,X).
simp(sub,X,0,X).
```

```
simp(sub,0,X,Y) :- simp(sub,X,Y), !.
simp(mul,0,X,0).
simp(mul,X,0,0).
simp(mul,1,X,X).
simp(mul,X,1,X).
simp(exp,X,0,1).
simp(mul,un(sub,X),Y,U) :- !,
  simp(mul,X,Y,V), simp(sub,V,U).
simp(mul,X,un(sub,Y),U) :- !,
  simp(mul,X,Y,V), simp(sub,V,U).
simp(exp,X,1,X).
simp(exp,0,X,0).
simp(exp,1,X,1).
simp(OP2,X,Y,U) :-
  dif(OP2,exp), integer(X),
  integer(Y), val(OP2,X,Y,U), !.
simp(OP2,X,bin(OP2,U,V),T) :- !,
  simp(OP2,X,U,Z), simp(OP2,Z,V,T).
simp(OP2,X,Y,bin(OP2,X,Y)).

/* règles de simplification des opérations unaires      */
/* simp(op,X,Y) signifie: op(X) peut se simplifier en Y */

simp(sub,0,0).
simp(sub,un(sub,X),X).
simp(sin,0,0).
simp(cos,0,1).
simp(OP1,X,un(OP1,X)).
```

Cet exemple nous semble intéressant, en ceci qu'il représente un embryon de base de connaissances. On constate un nombre relativement élevé d'alternatives de chaque clause, un mélange de faits et de règles, et la présence généralisée de variables dans les règles, ou dans les faits où elles ont valeur de quantificateur universel. On remarque également un usage généralisé de la coupure (!), mais celle ci a essentiellement un rôle d'optimisation en empêchant l'essai des autres alternatives lorsqu'une première solution a été trouvée. Un mécanisme plus adéquat pour la remplacer dans la presque totalité des cas consisterait à spécifier que l'on ne s'intéresse qu'à la première solution.

IV.6. LANGAGE ET TYPES DE DONNEES:

IV.6.1. Le langage:

Le langage retenu dans Opale est le langage Prolog, à quelques extensions près. Ce langage en lui même est considéré comme modèle de donnée et langage d'implémentation, mais il reste nécessaire de fournir des mécanismes complémentaires (accès multi- utilisateurs, contraintes d'intégrité, interface utilisateur) pour en tirer un SGBD convenable. Dans le cadre de cette thèse, nous en resterons au niveau du langage Prolog lui même. Notre étude ayant en fait essentiellement porté jusqu'ici sur les mécanismes de base de l'interprétation de Prolog, on ne donnera ici que des idées générales sur les extensions à apporter à Prolog dans un contexte bases de données.

Ces extensions sont assez limitées, et ont déjà fait l'objet d'études par ailleurs [GAL 83, ABI 87]. On cite couramment le prédicat évaluable "setof" (ensemble de): l'appel: setof(p(X),X,L) retourne dans L la liste des X satisfaisant le prédicat p(X). En l'absence de ce prédicat évaluable, la seule façon de construire la liste des solutions d'un littéral consiste à procéder par ajout de clauses, ce qui est très lourd. D'autres prédicats, ou l'utilisation de variable globales (non affectées par backtracking) permettraient de réaliser les fonctions agrégat.

La notion de variable anonyme, déjà présente sur plusieurs interpréteurs Prolog est également utile: il s'agit de variables qui n'apparaissent qu'une fois dans une clause, et ne sont donc pas utilisées. Leur seul rôle est donc d'occuper la place d'un argument du prédicat. La notion de variable anonyme peut alors éviter à l'interpréteur de générer des substitution qui sont inutiles. Dans une base de données, une telle notion permet d'ignorer certains "champs" d'un prédicat (selon la terminologie relationnelle), et donc de limiter le volume effectif de données manipulées: ceci correspond à la projection du modèle relationnel.

Nous verrons également ultérieurement que certains mécanismes de contrôle de la stratégie de recherche peuvent être nécessaires, pour permettre à l'interpréteur d'adopter la stratégie de recherche la mieux adaptée à la nature (base de données/programme) des clauses en cours de vérification.

Un autre ensemble d'adaptations permettrait de faciliter la programmation et d'éviter l'emploi de la coupure:

- prédicat NON: en Prolog, on utilise souvent le principe de la négation par l'échec: non(p(X)) s'écrit:

```
non(p(X)) :- p(X), !, fail.
non(p(X)).
```

  Autrement dit, si p(X) est vérifiable, on coupe le choix de la seconde alternative et on retourne échec, sinon, la coupure n'est pas exécutée, et on essaye la seconde alternative qui retourne vrai. Ce type de construction est assez fréquent en Prolog.

- possibilité de restreindre la recherche à la première solution: premier(p(X)) permettrait de ne retourner que la première solution de p(X). L'avantage d'un tel prédicat sur la formule "p(X), !", sémantiquement équivalente, est de permettre, dès l'appel de p(X), d'adopter une stratégie de recherche adaptée à la recherche d'une solution unique. On

peut de la même façon envisager un système d'annotation d'un paquet de clauses permettant d'indiquer qu'une seule clause peut admettre une solution.

- association de contraintes à une variable: un exemple en est le diff de Prolog II: diff(X,Y) retourne vrai ou faux si les variables sont liées, sinon, il place une contrainte sur les variables libres: lorsque la variable est liée, la contrainte doit être satisfaite, sous peine d'échec de l'unification. On pourrait étendre cette notion à d'autres types de contraintes:
  * relation d'infériorité ou supériorité entre variables scalaires non nécessairement numériques,
  * type (le prédicat évaluable entier(X) pourrait, si X est une variable libre, lui imposer la contrainte de n'être liée qu'à un entier),
  * etc...

Prenons l'exemple de la recherche, sur une relation R, des nuplets tels que le premier attribut soit inférieur à 20:
On peut écrire: :- r(X,Y), inf(X,20).
Ce qui revient à amener toute la relation en mémoire et à faire ensuite la sélection.
Si par contre, on peut écrire: :- inf(X,20), r(X,Y).
La contrainte placée sur X par le inf peut être vérifiée dès la lecture du nuplet, ce qui permet d'accélérer sensiblement le traitement.

Ceci peut être réalisé par d'autres moyens, tel que le constrain de D-Prolog: ce prédicat évaluable contraint un prédicat à rester vrai dans la suite de l'évaluation. La vérification est effectuée chaque fois qu'une variable contenue dans les arguments du prédicat est prise.

IV.6.2. Codage des symboles:

Comme beaucoup de langages de l'intelligence artificielle, Prolog traite essentiellement des symboles. Un symbole est un identificateur auquel une certaine sémantique est attachée par le programmeur dans le contexte d'un programme. Les symboles peuvent être associés entre eux pour dénoter des faits ou des règles. Par exemple, le prédicat "couleur(ballon, orange)" utilise trois symbole: couleur, qui est un symbole fonctionnel; ballon, qui désigne un objet; et orange, qui désigne une couleur. Le prédicat exprime, par pure convention, que la couleur du ballon est orange, mais on aurait très bien pu exprimer ce fait avec des symboles arbitraires, par exemple a(b,c). Dans le même programme, on peut trouver les mêmes symboles avec une sémantique différente, par exemple: "mange('Pierre', orange)", où orange désigne cette fois un fruit qui est mangé par une personne.

Durant l'interprétation, les symboles sont codés sous une forme interne permettant, d'une part de travailler sur des données de longueur fixe, et d'autre part d'accéder à un certain nombre d'informations concernant les symboles (en particulier, l'adresse d'un paquet de clauses, si le symbole est un nom de prédicat). La correspondance entre la forme externe du symbole et sa forme interne est assurée par un dictionnaire, accédé par

hash-coding à partir de la forme externe, ou par indexation à partir de la forme interne. Cette traduction entre la forme externe et la forme interne n'est effectuée que lors des entrées sorties.

Dans les bases de données, les données non numériques sont généralement stockées sous forme de chaînes de caractères, et il est possible, dans une base de données Prolog, de stocker les symboles sous cette forme, en assurant la traduction, si elle est nécessaire, au niveau du chargement en mémoire.

Il est aussi possible de coder les symboles de la base par l'intermédiaire d'un dictionnaire, de façon à faciliter la transition entre l'accès à la base et le traitement (résolution) en mémoire.

Un tel codage est efficace en terme d'occupation de l'espace disque: un symbole peut être codé sur 4 ou 5 octets (un tag comprenant le nombre d'arguments des symboles fonctionnels, plus trois ou 4 octets faisant référence au dictionnaire), alors que la longueur moyenne des symboles (évaluée à partir d'un ensemble de textes assez divers) est de l'ordre de 6. Ceci compense donc l'espace occupé par le dictionnaire. De plus, le codage des symboles peut comporter le nombre d'arguments du symbole, ce qui évite d'ajouter des marqueurs dans la base pour délimiter les listes d'arguments des symboles fonctionnels.

En termes de performances, l'utilisation d'un dictionnaire impose des accès disques supplémentaires au niveau des échanges avec l'extérieur (entrée des données, sortie des résultats, communication avec un autre SGBD en cas de répartition), ainsi que pour les tris, ou pour les comparaisons autres que l'égalité. Ces accès peuvent être pénalisants, mais l'accès au dictionnaire peut se faire par hash coding, et être assez bien optimisé par l'utilisation d'un cache en mémoire primaire, du filtrage, et l'exploitation du parallélisme dans l'accès (recherche groupée de plusieurs symboles.

Un avantage essentiel de l'utilisation de noms internes est enfin la possibilité de simplifier considérablement la manipulation des symboles en leur donnant une taille fixe et réduite. Les listes d'alternatives seront également plus compactes, ce qui permettra d'en accélérer le filtrage.

L'inconvénient est, par contre, que la gestion du dictionnaire est délicate pour des bases de données qui évoluent beaucoup: lorsqu'un symbole nouveau est introduit dans la base, on crée l'entrée dictionnaire correspondante, si cette entrée n'existe pas déjà. A l'opposé, lors de la suppression d'une clause, on devrait être en mesure de supprimer les entrées dictionnaire correspondant aux symboles qui n'apparaissent plus dans la base. Or, cela nécessiterait soit un parcours de toute la base pour vérifier qu'un symbole n'est plus référencé, soit d'avoir, au niveau du dictionnaire un compteur de références, incrémenté lorsqu'une nouvelle occurence du symbole est introduite dans la base, et décrémenté lorsqu'une occurence est supprimée. Ceci complique donc nettement les opérations de modification de la base. Cet inconvénient est acceptable si la base est peu modifiée, ou si les symboles utilisés appartiennent à un vocabulaire relativement fermé et limité: dans ce dernier cas, on sait que le dictionnaire cessera de croître lorsque l'ensemble du vocabulaire sera saisi.

Le choix entre l'utilisation d'un dictionnaire et le codage sous forme de chaînes de caractères va donc dépendre essentiellement de l'utilisation envisagée pour la base de données:

- Soit il s'agit d'une base de données d'utilisation classique, où Prolog n'intervient de façon faible que pour apporter quelques extensions, et il y aura intérêt à coder les symboles sous forme de chaînes.
- Soit la base de données est vue comme une extension en mémoire secondaire d'un programme Prolog, les échanges entre la BD et le programme pouvant être fréquents, et il y a alors lieu de préférer un codage par le biais d'un dictionnaire, qui facilite cette communication, l'inconvénient apporté par la gestion du dictionnaire au niveau des mises à jour étant alors compensé par la simplification des échanges.

Dans la suite, nous adopterons cette seconde démarche. Cependant, les solutions proposées pourront être assez facilement transposées à la première, par addition de mécanismes ad-hoc, déjà largement étudiés (filtrage de chaînes par automates finis).

IV.6.3. Les données:

Dans les langages de programmation procéduraux, le type des données est déterminé à la compilation. Un type est associé à une adresse mémoire, et les opérations portant sur cette adresse sont celles qui correspondent au type (dans les machines classiques). Les contrôles de validité sont donc associés à l'accès à une adresse particulière.

De même, dans les bases de données relationnelles, un type est associé aux attributs d'une relation. Les informations nécessaires sont stockées dans le schéma de la base, et les contrôles nécessaires effectués au moment des accès (consultations, mises à jour).

Il n'en est pas de même avec les langages logiques, où aucun type n'est statiquement affecté à une variable. Un argument d'un terme peut donc recevoir une valeur de type quelconque, et c'est dynamiquement, à l'exécution, qu'il faudra vérifier le type, et éventuellement adopter les types d'opérations correspondant. De plus, le programmeur peut tester le type d'une donnée, et le faire intervenir dans le traitement. Ceci nécessite que le type de la donnée puisse être identifié, et que les informations nécessaires soient présentes au niveau de chaque élément de donnée.

Les interpréteurs Prolog adoptent donc un codage explicite du type, codage qui peut par exemple prendre la forme d'un octet de préfixe. Les types reconnus peuvent être les suivants:

- symboles: le type comporte le nombre d'arguments du symbole.
- entiers
- réels: on peut utiliser un mot comportant le type et une partie du nombre, et une extension sur un second mot.
- caractères: codés sur un mot (malgré la perte de place: l'importance des caractères isolés est relativement faible).

- chaînes: codées sur une succession de mots, le premier mot comportant le type et la longueur.
- variables: codées sur un mot, comportant le type et le nom local de la variable (numéro).

Ce codage peut en outre rester ouvert pour permettre l'insertion de nouveaux types de données: vecteurs, ensembles, etc, et permettre le traitement de nouveaux types d'informations (sons, images, textes, etc...).

Un tel codage peut être adopté au niveau du stockage sur disque, où il permet de conserver la souplesse de Prolog (possibilité de données structurées en arborescence, avec typage non prédéfini).

IV.6.4. Codage des clauses:

Dans Prolog, les clauses sont stockées sous forme de listes chaînées d'alternatives. L'utilisation de chaînages sur disque n'est guère possible, aussi, dans une base de données, il faudra revenir à un stockage séquentiel.

Les clauses elles mêmes sont stockées sous forme de listes séquentielles de littéraux, ce qui convient au stockage sur disque.

IV.7. CONCLUSION

Dans ce chapitre, nous avons indiqué comment Prolog peut être utilisé comme modèle de données et langage d'implémentation d'un SGBD. Nous avons vu que l'approche Prolog diffère de l'approche relationnelle par bien des points, et que, de même, les bases de données Prolog diffèrent sur plusieurs points de la programmation en logique.

Dans la conception d'une machine, il est d'usage de se baser sur des expériences d'utilisation et sur l'évaluation de performances d'applications pré-existantes. Dans le cas des bases de données Prolog, de telles applications n'existent pas, ou alors de façon embryonnaire et expérimentale: la cause en est que l'idée même de bases de donnée Prolog est encore loin d'être évidente et qu'il n'existe pas encore d'outils efficaces pour les traiter. Il nous semble, d'ailleurs, que de tels outils passent par la définition de matériels adaptés.

Il n'est donc possible de compter sur aucune mesure, évaluation, ou expérience d'utilisation de bases de données Prolog pour la conception d'Opale. Ceci va donc nous amener souvent à nous appuyer sur les seules expériences proches disponibles, celles liées aux bases de données relationnelles et à la programmation en Prolog, même si nous avons dit qu'il existe des différences importantes entre ces problèmes et le nôtre. Notons à ce sujet que les premiers concepteurs de MBD se sont heurté au même problème, puisque les recherches dans ce domaine ont commencé pratiquement dès le début des travaux sur le modèle relationnel.

Chapitre V

STRATEGIE DE RECHERCHE

V.1. INTRODUCTION:

L'interprétation de programmes Prolog consiste à rechercher dans un espace arborescent la ou les solutions d'un problème donné. Diverses stratégies de recherche sont donc possibles, correspondant à différentes façons de parcourir une arborescence. Sur des machines séquentielles, les interpréteurs Prolog adoptent généralement la plus simple, et la plus efficace en terme d'espace mémoire nécessaire: en profondeur d'abord et de gauche à droite.

Bien que le langage, dans son principe et sa syntaxe, suggère qu'un parallélisme important peut être aisément dégagé, les interpréteurs actuels procèdent de façon purement séquentielle, et une recherche parallèle pose un certain nombre de problèmes. Beaucoup de travaux ont déjà été effectués à ce sujet, et nous présenterons plus loin quelques unes des approches proposées.

Par ailleurs, les bases de données relationnelles ont entraîné le développement d'un certain nombre de moyens pour le traitement ensembliste des données [ROH 80], et beaucoup de travaux ont également été faits sur le parallélisme dans ce domaine [DEW 79, BOR 82, GAR 81]. Nous considérons que les outils matériels des bases de données relationnelles peuvent être adaptés au cas des bases de données PROLOG, et qu'une base de données peut être répartie sur un assez grand nombre de petites unités de disque, le parallélisme (s'il est bien exploité) permettant d'obtenir des performances avantageuses, même si les performances individuelles des unités sont inférieures. Or, la stratégie de recherche des interpréteurs PROLOG classiques est assez mal adaptée à ces outils et à l'exploitation de ce parallélisme: ceci nous amène donc à rechercher de nouvelles solutions.

Dans ce chapitre, après quelques rappels sur l'interprétation des programmes Prolog, nous allons présenter quelques unes des stratégies de recherche couramment implémentées ou proposées. Nous verrons ensuite en quoi le problème des bases de données est différent, pour enfin proposer une nouvelle stratégie de recherche et indiquer son mécanisme d'implémentation.

V.2. STRATEGIES DE RECHERCHE:

V.2.1. environnements:

Une clause PROLOG a la forme générale suivante:

$$L_0 :- L_1, L_2, \cdots L_n.$$

Avec n >= 0 et $L_i = C_i(X_{1,}...,X_{j_i})$

où X_n est un terme, et $j_i \geq 0$.

Un environnement E est un ensemble de substitutions correspondant à un état donné du processus de vérification, les substitutions étant des couples variable - valeur produits par les unifications précédentes. L'application des substitutions d'un environnement E aux variables d'un littéral L va donner un prédicat, ou but que l'on notera: L/E. Un but peut donc être défini par un doublet littéral-environnement.

Exemple:

Considérons le programme suivant:

```
c1(X) :- c2(Y,Z), c3(Y,Z,X).
c2(a,c).
c2(a,d).
c3(a,c,b).
c3(a,c,d).
```

La première vérification de c2 va retourner une substitution sur Y, S(Y,a), et une substitution sur Z, S(Z,c), qui constitueront l'environnement lors de l'appel de c3: {S(Y,a),S(Z,c)}. L'application de ces substitutions au littéral c3(Y,Z,X) va alors donner le but c3(a,c,X), que l'on pourra encore noter: c3(Y,Z,X) / {S(Y,a), S(Z,c)}.

Le littéral L_1 ne dépend que des variables liées lors de l'appel de la clause: on note E_0 l'environnement correspondant. L_1/E_0 va avoir comme résultat un ensemble d'environnements noté $\{E_1\}$, correspondant aux solutions du prédicat L_1/E_0.

Un littéral L_i (i > 1) va dépendre des environnements E_{i-1} solutions du prédicat L_{i-1}/E_{i-2}. Il va avoir un double rôle:

- Filtrer les environnements E_{i-1} qu'il reçoit, c'est à dire vérifier que le prédicat L_i/E_{i-1} possède au moins une solution. Il va donc par cette fonction restreindre l'ensemble des environnements (solutions partielles). C'est ainsi que, dans l'exemple précédent, le littéral c3(Y,Z,X) va éliminer l'environnement correspondant à la seconde solution de c2, car c3(a,d,X) n'admet pas de solution.
- Lier de nouvelles variables, et donc éclater certains environnements: ainsi, si L_i/E_{i-1} possède plusieurs solutions, l'environnement E_{i-1} va être éclaté en autant d'environnements E_i qu'il existe de solutions, chaque E_i étant la concaténation de E_{i-1} et d'un nouvel ensemble de substitutions. Dans l'exemple précédent, c3(a,c,X) admet deux solutions, ce qui aura pour effet de créer deux environnements: {S(Y,a),S(Z,c),S(X,b)} et {S(Y,a),S(Z,c),S(X,d)}.

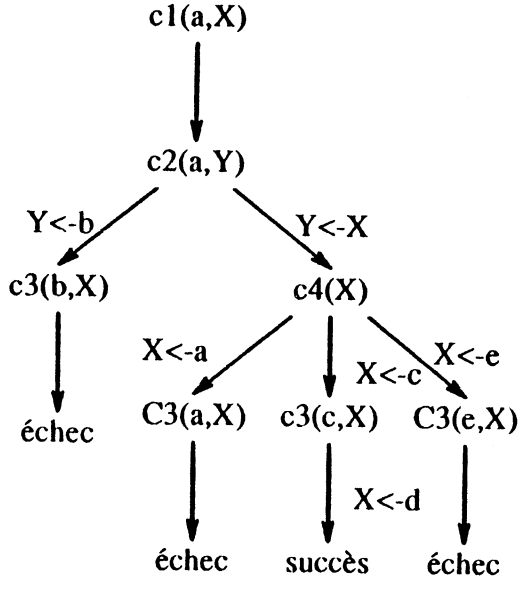
V.2.2. stratégie de recherche séquentielle:

On peut assimiler l'interprétation de PROLOG à une recherche dans un arbre abstrait (arbre de recherche) dont les branches conduisent soit à des solutions, soit à des échecs.

Exemple:

```
c1(a,X) :- c2(a,Y), c3(Y,X).
c2(a,b).
c2(a,X) :- c4(X)
c3(c,d).
c4(a).
c4(c).
c4(e).
```

L'interprétation de ce programme produit l'arbre suivant:



La façon de représenter l'espace de recherche sous forme d'arbre ést le fruit d'un premier choix: l'ordre de vérification des littéraux d'une clause (de gauche à droite). Ce choix est purement conventionnel, mais il contribue à définir la sémantique du langage, avec l'ordre d'exécution des opérations.

A partir de là, l'interprétation apparaît comme un simple parcours d'arborescence. L'algorithme le plus simple consiste alors à parcourir l'arbre "en profondeur d'abord": pour chaque noeud, on commence à descendre sur le premier fils. Lorsqu'on arrive à une feuille qui correspond à un échec, on remonte au premier noeud où il reste un (ou plusieurs) fils à traiter (retour arrière, ou backtracking), et on redescend sur le premier fils non traité. Au cours de la descente dans l'arbre, des substitutions sont produites et empilées, constituant l'environnement; elles sont dépilées et défaites en cas de retour arrière. Dans l'exemple précédent, l'environnement après le succès sur c3 est composé des substitutions: S(Y,c), S(X,d). L'interpréteur cherche donc à démontrer complètement une solution avant d'en essayer une autre. Une présentation plus complete de l'algorithme d'interprétation classique peut être trouvée en annexe II.

La seule alternative à la stratégie "en profondeur d'abord" pourrait être une stratégie "en largeur d'abord": on explore totalement un niveau de l'arborescence avant de passer au niveau inférieur. Une telle stratégie est inapplicable en pratique pour une implémentation séquentielle, car elle accumule des informations pour chaque solution partielle en cours de traitement, ce qui demande un grand volume de mémoire, pour un gain incertain.

Le modèle d'exécution de Prolog a une très grande influence sur la sémantique d'exécution des programmes, ce qui entrave l'exploitation du parallélisme:

- les littéraux successifs d'une même clause sont souvent non indépendants: dans l'exemple ci-dessus, le littéral c3 ne peut être exécuté qu'après le littéral c2, puisque ce dernier peut lier la variable Y (lui donner une valeur); ces littéraux sont donc dépendants et ne peuvent s'exécuter que l'un après l'autre. Même dans les cas où l'ordre d'évaluation est théoriquement indifférent, la cardinalité des solutions d'un littéral peut être excessivement élevée si certaines de ses variables n'ont pas été liées. Ceci est plus particulièrement vrai dans le cas des bases de données, et il existe un ordre optimal d'évaluation permettant de minimiser le nombre de solutions partielles.
- PROLOG comporte un certain nombre de mécanismes extra-logiques qui doivent généralement être exécutés dans l'ordre précédemment défini, parce qu'ils sont non réversibles par backtracking et que leur effet dépasse l'exécution de la clause où ils sont placés: c'est le cas de la coupure (!), qui supprime l'essai des alternatives de la clause en cours de vérification, des entrées-sorties, des ajouts ou suppressions de clauses.
- Pour certains problèmes, l'ordre des opérations ou des résultats peut être important: c'est le cas des applications interactives, par exemple, où les interactions avec l'utilisateur doivent se faire dans un certain ordre pour avoir un sens.
- Pour de très gros problèmes, l'espace de recherche sera énorme, et on risque d'arriver à une explosion combinatoire du nombre d'alternatives qui peuvent être explorées simultanément, et donc du nombre de processus.

Dans la prochaine section, nous verrons quelques unes des approches classiques pour une interprétation parallèle de Prolog. On peut cependant déjà observer que l'interprétation séquentielle de PROLOG peut entraîner un travail inutile.

Considérons par exemple la clause suivante:

c(X, Y, Z) :- c1(X), c2(X, Z), c3(X, Y).

Pour chaque solution de c2, une vérification de c3 est entraînée. En particulier, si, pour un X donné, il existe plusieurs valeurs de Z, on exécutera autant de fois la vérification de c3 avec la même valeur de X. Ceci peut être particulièrement pénalisant lorsque on recherche un ensemble de solutions, ou lorsque c3 n'est pas vérifié pour la valeur de X correspondante: on ne passera à une autre solution pour X (retour arrière sur c1) qu'après avoir épuisé les solutions de c2 pour une valeur de X donnée. Ce genre de situation a amené certains chercheurs à rechercher des méthodes de backtracking intelligent permettant de revenir à un littéral qui est susceptible de modifier les arguments du

littéral à vérifier [PEA 82].

V.2.3. Stratégies de recherche parallèle:

De nombreux travaux ont été faits pour rechercher des modèles d'interprétation parallèle de PROLOG [UCH 83]. L'approche la plus commune [CON 81] consiste à distinguer deux formes de parallélisme: le parallélisme ET, qui cherche à vérifier en parallèle les littéraux d'une même clause, et le parallélisme OU, qui cherche à vérifier en parallèle les différentes alternatives d'une clause qui unifient avec un littéral d'appel. On distingue également le parallélisme d'unification, tendant à dégager le parallélisme au sein même de l'opération d'unification, mais il est assez généralement admis maintenant que cette forme de parallélisme est peu intéressante. Il est également très tentant de rechercher des méthodes d'interprétation de type "data flow" qui, à première vue semblent correspondre assez bien au langage.

Dans ce qui suit, nous allons passer brièvement en revue ces approches. Des revues de l'état de l'art plus complètes peuvent être trouvées dans [DAN 87] et [SYR 85]. En fait, le parallélisme en Prolog fait aujourd'hui l'objet de nombreux travaux, et il est assez difficile d'en effectuer une synthèse d'ensemble.

V.2.3.1. Data flow:

Le data flow (séquencement par les données) consiste à déclencher l'exécution d'une opération dès que ses opérandes ont été évalués, et non selon un ordre fixé. La seule donnée de séquencement utile est donc la relation de dépendance entre les résultats et les opérandes des opérations. En Prolog, on peut donc envisager de déclencher la vérification d'un littéral dès que ses variables d'entrée ont été liées.

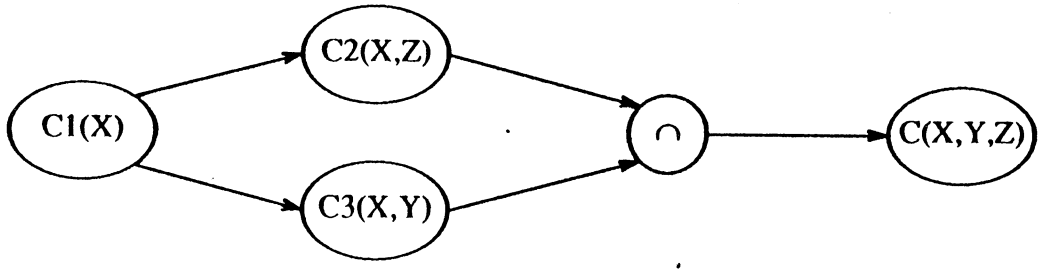
Cette approche est inadaptée à Prolog pour au moins deux raisons fondamentales: d'une part, les variables d'entrée d'un prédicat ne sont normalement pas spécifiées, et d'autre part, la structure arborescente des données manipulées fait qu'il existe des données partiellement définies. La condition de déclenchement d'une vérification est donc loin d'être simple à définir, et les approches data-flow de Prolog sont le plus souvent basées sur une redéfinition du langage lui même.

V.2.3.2. parallélisme ET:

Il consiste à vérifier en parallèle les littéraux indépendants d'une même clause, c'est à dire ceux qui ne partagent pas de variables libres. Comme nous l'avons vu, l'interprétation parallèle des littéraux d'une même clause est loin d'être évidente, puisque les variables introduisent des dépendances entre les littéraux. On doit donc, pour utiliser le parallélisme, créer un graphe de dépendance entre les littéraux de la clause:

Exemple:

c(X,Y,Z) :- c1(X), c2(X,Z), c3(X,Y).



Dans cet exemple, on peut, en principe évaluer c2 et c3 dès que c1 aura évalué X. Il peut arriver que c1 laisse la variable X libre ou partiellement instanciée. Dans un tel cas, il est possible que ce soit c2 qui lie X, et c3 ne pourra pas être exécuté avant que c2 le soit; par contre, si c2 laisse lui aussi X libre, c3 doit être exécuté de toutes façons. La condition pour la vérification de c3 est donc: X est lié ou c2 est vérifié.

Il peut arriver également que des littéraux à priori indépendants deviennent en fait dépendants de par les substitutions effectuées sur les variables:

Exemple:

p :- l1(X,Y), l2(X), l3(Y).
l1(f(a,X),g(b,X)).

X et Y étant définis par l1, on peut envisager de vérifier l2 et l3 en parallèle. Cependant, les variables X et Y sont rendues dépendantes lors de leur instanciation par l1: elles sont en effet instanciées à deux termes qui partagent une variable. La possibilité d'exécution parallèle de l2 et l3 doit donc être réévaluée pour chaque environnement traité.

L'exploitation du parallélisme ET va donc nécessiter d'analyser chaque clause pour dégager les dépendances et générer un graphe d'exécution. Ce graphe sera de plus à réévaluer pour toute instanciation des variables. Par ailleurs, le résultat n'est valable que pour les ensembles de variables (environnements) ayant satisfait chacun des prédicats, si bien qu'il faudra effectuer la jointure des ensembles de solutions de chaque littéral pour obtenir le résultat final d'une clause. Ceci représente donc un traitement complémentaire important. Une autre approche consisterait à partager une même copie de l'environnement entre des littéraux vérifiés en parallèle, mais des problèmes d'exclusion mutuelle se posent alors.

Il faut observer également que le parallélisme ET peut très facilement entraîner un travail inutile. Considérons dans l'exemple ci-dessus que pour une valeur de X, l2 ne soit pas vérifié: si l3 est vérifié en parallèle pour le Y correspondant, cette vérification aura été inutile. Or, si le programme a été écrit d'une façon optimale, ce risque est aggravé, puisque le programmeur aura vraisemblablement pris soin de mettre en premier la clause

la plus restrictive. Dans un SGBD, une telle optimisation peut, de plus, être exécutée automatiquement. Ceci entraîne donc un risque de gaspillage des ressources de la machine, risque particulièrement gênant si la machine est partagée entre plusieurs utilisateurs: le parallélisme inter-utilisateurs est alors un plus sûr garant de la bonne utilisation des ressources, et de l'obtention de performances élevées.

Le parallélisme ET semble nécessiter une adaptation du langage. Dans IC-Prolog [CLA 79], on donne à l'utilisateur la possibilité d'annoter son programme, en indiquant qu'une variable sera toujours une variable d'entrée ou de sortie, ou qu'un littéral peut être évalué dès qu'une variable donnée est évaluée ("eager consumer"), ou qu'un littéral peut n'être évalué que si une variable de sortie est "réclamée" par un autre littéral ("lazy evaluator"). Il est même possible, de cette façon, de traiter une donnée volumineuse au fur et à mesure de son évaluation par le littéral producteur. Ceci permet de dégager un parallélisme maximal, mais a pour inconvénient de reporter sur l'utilisateur la préoccupation d'annoter son programme. Il est cependant possible que cette annotation puisse être (au moins en partie) effectuée automatiquement par une analyse du programme, prenant en compte le mode d'utilisation du programme spécifié par l'utilisateur et les modes implicites de certains prédicats évaluables (entrées sorties, entre autres), et procédant par déductions successives.

Dans [DEG 85], DeGroot propose un exploitation partielle du parallélisme ET basée sur une analyse statique des programmes: le programme est alors représenté sous forme d'un graphe d'exécution faisant intervenir soit des exécutions séquentielles, soit des exécutions parallèles, soit enfin des exécution parallèles conditionnelles. Dans ce dernier cas, la condition consiste à vérifier soit que les littéraux ne comportent plus de variables libres, soit que leurs variables sont indépendantes. Le test d'indépendance est effectué de manière heuristique, afin d'en réduire le coût. Cette méthode semble permettre de dégager un réel parallélisme, en reportant une partie notable du travail au niveau de la compilation, et en n'effectuant dynamiquement que des tests simples, donc peu pénalisants. Elle ne permet pas de dégager tout le parallélisme disponible, mais elle semble néanmoins être une des approches les plus réalistes du parallélisme ET.

En fait, le parallélisme ET peut se justifier dans la mesure où on dispose d'une architecture fortement parallèle. Dans ces conditions, la perte de performances introduite par le parallélisme peut être compensée largement par l'utilisation parallèle d'un grand nombre de processeurs.

Nombre d'approches du parallélisme ET nous semblent erronées en ce qu'elles considèrent les variables individuellement, alors qu'il est plus juste de ne les considérer que comme des éléments d'un environnement. Dans ce sens, les littéraux L2 et L3 de l'exemple ci-dessus ne sont pas réellement indépendants, puisque L2 va vérifier une condition sur des environnements (ou couples (X,Y)), et ne laisser passer que ceux qui vérifient cette condition.

V.2.3.3. parallélisme OU:

Dans cette forme de parallélisme [UME 83], chaque alternative d'une clause va être vérifiée par un processus parallèle, la ou les solutions retournées pouvant être à leur tour traitées par un processus parallèle. Ceci peut entraîner plusieurs niveaux de parallélisme.

Considérons l'exemple suivant:

```
c1(X,Y,Z) :- c2(X), c3(X,Y), c4(X,Z).
c2(a).
c2(b).
c2(c).
...
```

La vérification parallèle de c2 va entraîner les opérations suivantes:

- recherche des en-têtes de clauses qui unifient avec l'appel de c2,
- vérification en parallèle des alternatives non atomiques,
- traitement en parallèle des différents résultats retournés par c2, pour la vérification de c3 et c4: chaque solution de c2 va en effet entraîner une vérification de c3, puis éventuellement une ou plusieurs vérifications de c4, ces opérations pouvant être également exécutées en parallèle. Si, par exemple, c2(X) admet comme solutions: X = a, X = b, et X = c, on va pouvoir exécuter en parallèle les vérifications de c3(a,Y), c3(b,Y), c3(c,Y).

Le parallélisme OU présente moins d'inconvénients que le parallélisme ET, et permet de dégager un parallélisme important, spécialement dans le cas hautement non déterministe des bases de données. Cela pose toutefois le problème d'un grand nombre de processus parallèles, avec risque d'explosion combinatoire. Il faut donc être capable de contrôler ce parallélisme, et éventuellement de le limiter, pour éviter un effondrement des performances. Il pose également le problème de processus travaillant en parallèle sur des environnements non disjoints (les substitutions produites par les processus pères étant communes aux processus fils). Il faut alors soit utiliser une représentation des environnements basée sur la recopie, soit structurer ceux ci non en pile, mais en arborescence.

Un autre problème est lié aux prédicats possédant des effets de bord, et en particulier la coupure (suppression des choix). L'existence de tels prédicats dans un paquet de clauses implique en effet un certain ordre (éventuellement partiel) de l'évaluation, et donc une limitation du parallélisme. Nous reviendrons brièvement sur ce point dans la suite.

V.3. CAS DES BASES DE DONNEES:

Une caractéristique essentielle des bases de données est que, dans la majorité des cas, les données résident sur disque. L'optimisation des accès disques, en nombre et en durée, va donc être un critère de performances déterminant, et on peut admettre qu'avec le développement des circuits intégrés, il sera toujours possible de mettre derrière les disques une puissance de traitement suffisante, alors que les performances des disques sont une donnée du problème.

Or, le fait que les disques soient accédés par blocs rend particulièrement inadaptées les stratégies de recherche séquentielles. En effet, un accès disque avec les unités actuelles va demander environ 30 ms. Si un paquet de clause s'étend sur, par exemple, toute une piste, il faudra dans la majorité des cas accéder à l'ensemble des informations de la piste pour sélectionner toutes les solutions possibles: soit on le fait en accédant à l'ensemble de la piste en une seule fois, ce qui demandera environ 45 ms, soit on effectue plusieurs accès disques de 30 ms successifs. Une bonne utilisation du disque nécessite donc de retirer un maximum d'information de chaque accès. On peut se contenter de conserver des blocs disque en mémoire de façon à reprendre ultérieurement la recherche, cependant, on risque alors d'encombrer la mémoire primaire d'informations en grande partie inutiles.

Comme nous le verrons au chapitre suivant, l'utilisation d'un dispositif matériel de filtrage va permettre de retrouver en un balayage du disque tous les en-têtes de clause qui unifient avec un ensemble de buts. Il est donc possible de n'amener ainsi en mémoire que des données sélectionnées et utiles. Lorsque des faits sont ainsi filtrés, on obtient directement les solutions, sous forme de substitutions. On dispose ainsi d'ensembles d'environnements qui, par application des substitutions sur un littéral vont fournir des buts. Or, les buts formés par application des substitutions de plusieurs environnements sur un même littéral concerneront un même paquet de clauses, et leur vérification entraînera souvent des accès aux mêmes blocs du disque. Il est donc intéressant de pouvoir traiter ensemble ces buts, en utilisant les possibilités de filtrage d'ensembles de buts.

Exemple:

```
c(X,Y) :- l1(X,Z), l2(Z,Y).
l1(a,b).
l1(c,b).
l1(d,e).
l2(f,g).
l2(b,g).
```

La vérification de l1 retourne ici trois solutions. Appliquées au littéral l2, les substitutions produiront seulement deux buts différents: l2(b,Y), et l2(e,Y). Si ces deux buts concernent un même bloc disque, leur vérification ne demandera qu'un accès disque pour obtenir l'ensemble des solutions par filtrage (voir chapitre suivant), en supposant les descripteurs de l'espace disque et les index résidant en mémoire primaire. On aura donc obtenu l'ensemble des solutions en deux accès disques (2 X 45 ms = 90 ms), alors qu'une stratégie de recherche séquentielle nécessite 9 accès (9 X 30 ms = 270 ms): la recherche des solutions de l1 demande 3 accès, et ces solutions entraînent trois fois deux accès à l2 (un accès par alternative).

Ceci nous amène à rechercher une méthode de résolution basée sur le traitement d'ensembles d'environnements: création d'ensembles d'environnements par filtrage, et recherche d'un ensemble de buts. Chakravarthy, Minker et Tran présentent brièvement

une telle approche dans [CHA 81], le but étant pour eux d'interfacer Prolog avec des bases de données relationnelles.

Le seul parallélisme réel dans le cas d'une base de données sera celui autorisé par l'accès simultané à des unités de disque différentes, et éventuellement à des données stockées en mémoire primaire: ceci ne devrait donc autoriser qu'un parallélisme OU assez limité.

L'optimisation des accès disques en durée est basée sur un ordonnancement des accès dans l'ordre des cylindres adressés, de façon à minimiser les déplacements de bras [DEN 67]. Un tel ordonnancement est d'autant plus efficace que le nombre de requêtes en attente sur le disque est plus grand. Un effet de bord attendu de la stratégie de recherche est alors d'augmenter la taille des files d'attente, en permettant à une cible d'être plus rapidement candidate à un accès disque. Ceci permet d'arriver, dans une certaine mesure à un séquencement des opérations basé sur la disponibilité du disque.

V.4. STRATEGIE DE RECHERCHE POUR OPALE:

On peut considérer la vérification d'une clause comme un processus opérant sur des ensembles de solutions, et donc adopter un modèle ensembliste de la résolution, chaque littéral recevant l'ensemble des environnements solutions du littéral précédent, et transmettant ses solutions au littéral suivant. Les environnements résultats après la vérification du dernier littéral sont retournés comme solution au littéral appelant. Chaque littéral peut alors être traité par un processus qui opère en pipe line sur un flux de solutions. Le parallélisme réel est réduit à un parallélisme OU limité pour la vérification des clauses non atomiques. On n'a plus de backtracking, puisqu'un échec se traduit par l'élimination d'un environnement de l'ensemble des solutions.

Dans [LIN 84], G. Lindstrom présente un modèle d'exécution dit "stream based", basé sur le traitement de flots de solutions: le traitement d'un programme se ramène à un traitement parallèle de clauses "OU", et à un traitement des flots de solutions par des processus ET opérant en parallèle sur des solutions différentes (ce qui correspond à peu près à un traitement en pipe line). Si on considère le traitement d'une clause non atomique, chaque littéral va donc consommer les solutions du littéral précédant, et produire un nouvel ensemble de solutions qu'il transmet à son tour au littéral suivant, ces ensemble de solution étant ainsi consommés en pipe-line. On a donc un parallélisme OU auquel s'ajoute un parallélisme ET induit, différents littéraux pouvant traiter en parallèle une solution diff'erente.

Notre modèle (conçu parallèlement, donc sans interaction), introduit un niveau de processus supplémentaire, dû au fait que la recherche des en-têtes de clauses qui unifient avec les buts est exécutée au niveau du disque par un processeur spécifique (voir prochain chapitre). Les processus OU ne sont donc mis en jeu que lorsque la recherche sélectionne des clauses non atomiques. Le rôle du processus OU est alors de générer les processus ET nécessaires à la résolution de la clause, et de surveiller l'exécution de ces processus pour répercuter leur terminaison au niveau du processus appelant.

V.4.1. Description générale du modèle d'évaluation d'OPALE:

Une clause PROLOG pourra être résolue par un réseau de processus opérant en pipe-line sur des ensembles de solutions (Fig. V.1), chaque littéral recevant des environnements produits par le littéral précédent, et transmettant ses solutions au littéral suivant:

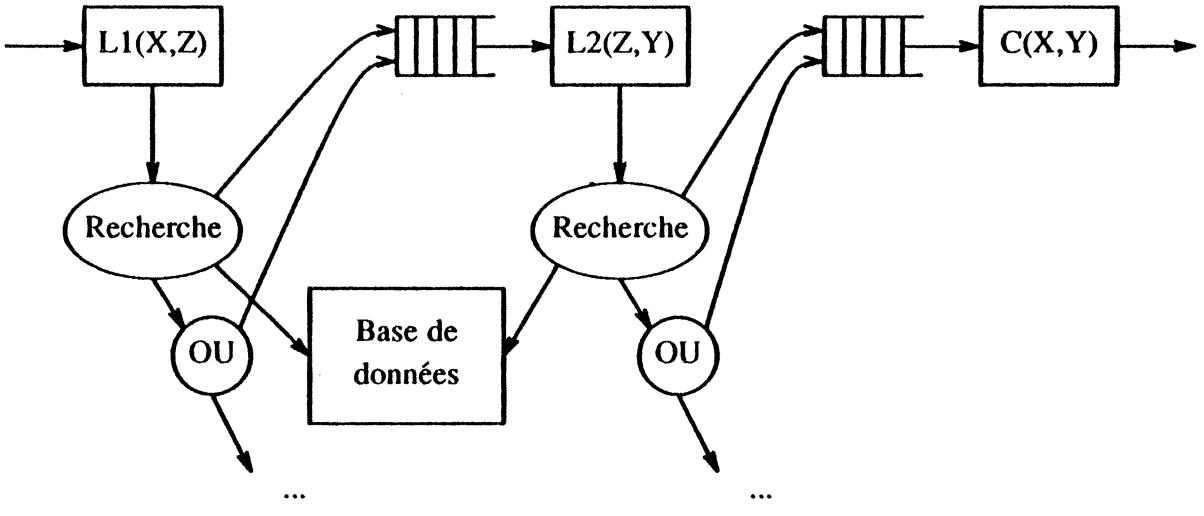


Figure V.1.

- Les processus ET: ils sont attachés à chaque littéral du corps de la clause. Ils reçoivent les environnements solutions du littéral précédent, produisent des buts par application des substitutions, et lancent des processus de recherche pour effectuer leur vérification. Pour le premier littéral d'une clause (l1(X,Z), dans l'exemple de la figure V.1), il y a un seul environnement correspondant aux substitutions générées par l'unification de l'en-tête de clause avec le littéral d'appel de la clause.
- Les processus de recherche: ils reçoivent des buts transmis par les processus ET. Ils recherchent les clauses dont l'en-tête unifie avec le but. Pour cela, ils passent par le système d'indexation pour connaître l'adresse de la ou des zones disques susceptibles de contenir les clauses recherchées, puis transmettent des requêtes de filtrage aux processeurs disques. Ils recueillent les substitutions générées par les unifications. Lorsque des clauses non atomiques sont rencontrées, ils génèrent des processus OU pour leur vérification. Pour les clauses atomiques (axiomes), ils transmettent directement les substitutions au processus qui succède à leur processus père dans le pipe-line.
- Des processus OU sont générés pour la vérification des clauses non unaires. Ils lancent un nouveau processus de vérification (nouvel ensemble de processus ET). Ils recueillent les solutions et les transmettent au successeur du processus ET qui à généré la cible qu'ils vérifient.

Notre approche permet d'exploiter trois sortes de parallélisme:

- Le parallélisme OU, dans la vérification des alternatives non atomiques,
- Le traitement parallèle de différentes solutions partielles dans les différents étages du pipe-line,
- La recherche parallèle des en-têtes de clause qui unifient avec les buts: la recherche d'un but, ou d'un ensemble de buts peut être effectuée en parallèle sur plusieurs disques, ou encore, un ensemble de buts peut faire l'objet d'une recherche groupée au cours d'un seul accès disque.

Les processus de recherche constituent la majeure différence entre notre modèle et les autres modèles pour l'interprétation parallèle de Prolog. Leur rôle est de permettre de dégager le parallélisme dans la recherche, plusieurs processus de recherche pouvant être lancés en parallèle lorsque celle-ci concerne plusieurs unités de disque. Des requêtes de filtrage sont générées par les processus de recherche et envoyées aux processeurs disques pour être exécutées. Les temps d'attente au niveau des disques (files d'attente, délais de positionnement de têtes de lecture) vont permettre l'accumulation d'ensembles de requêtes de filtrage concernant une même région du disque. Ces requêtes pourront alors être satisfaites en un seul accès disque.

En effet, le fait que le disque soit accédé par blocs dont on retire des ensembles de solutions, fait que des séries de requêtes sont générées durant des intervalles de temps assez réduit, ce qui permet d'espérer traiter des ensembles de solutions de cardinalité suffisante au niveau de chaque accès, sans qu'il soit forcément nécessaire d'introduire un dispositif de régulation pour faciliter le regroupement des accès disques.

L'asynchronisme introduit entre la recherche et le reste de la résolution permet en outre de réordonner les requêtes d'accès disques, de façon à optimiser les déplacements de bras. Notre stratégie peut donc être considérée comme "dirigée par le disque".

V.4.2. Exemples:

Un exemple plus complet permettra de faciliter la compréhension de notre stratégie. Considérons le programme suivant:

```
p(X,Y,Z) :- l1(X,Y), l2(Y,Z).
l1(a,b).
l1(b,c).
l2(b,d).
l2(c,X) :- l4 (X), l5 (X).
l2(c,X) :- l6 (X).
l4(b).
l4(a).
l5(a).
l6(b).
```

Considérons la question: *p(X,Y,Z).* Le réseau de processus entraîné par l'interprétation de ce programme est représenté sur la figure V.2. Dans cette figure, on

n'indique sur les flèches que les solutions transmises par les processus de recherche. Pour chaque processus ET, on indique le littéral qu'il traite. Pour un processus de recherche, on indique le but, et pour un processus OU, on ne met qu'une flèche vers le premier processus ET correspondant.

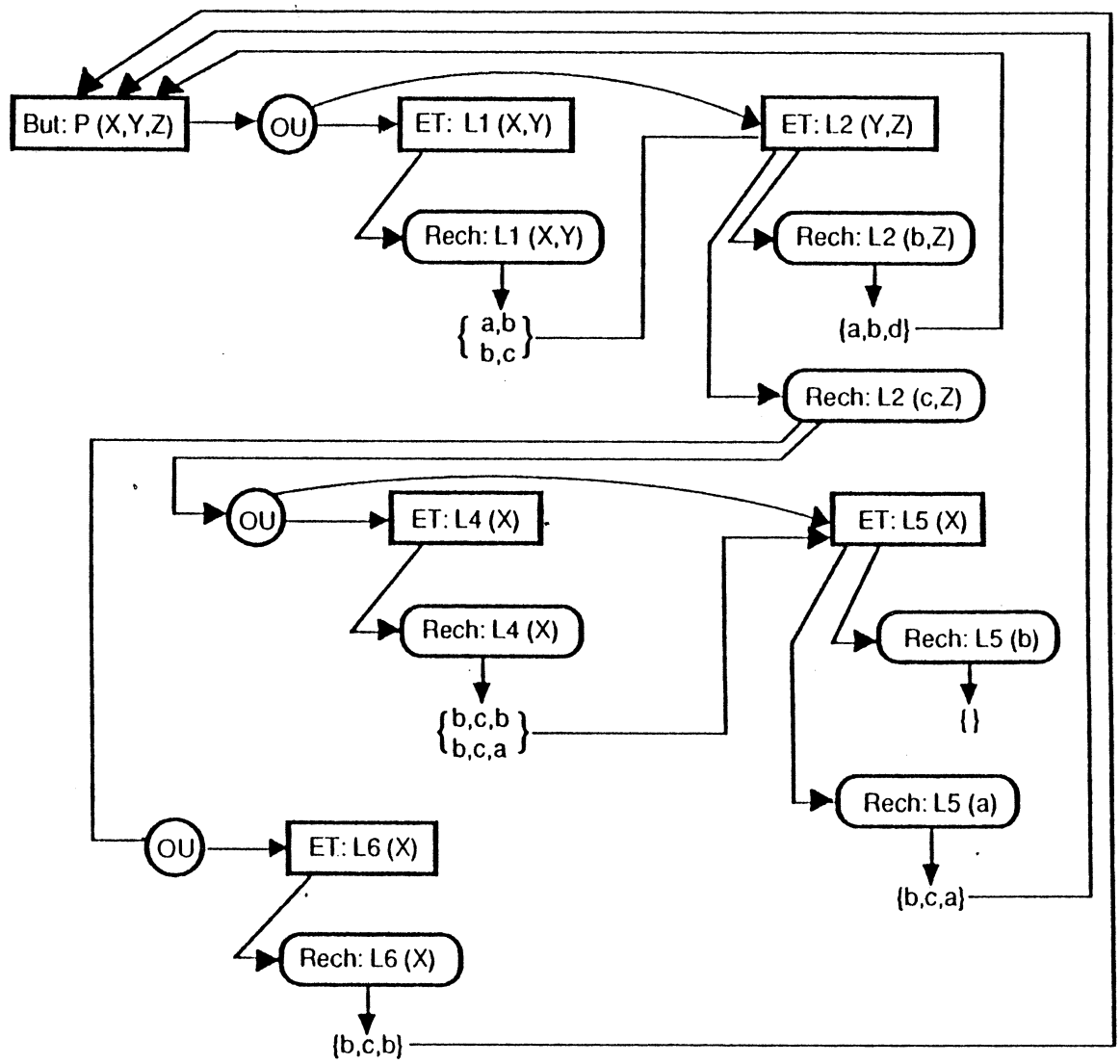


Figure V.2

- Le processus ET *l1(X,Y)* reçoit un seul environnement vide, génère un seul but *l1(X,Y)*, et crée le processus de recherche correspondant. celui ci trouve (par filtrage) deux solutions, avec les environnements *{X=a,Y=b}*. Ces solutions sont transmises au prochain processus ET *(l2(Y,Z))*.

- Le processus ET *l2(Y,Z)* applique les substitutions au littéral et crée deux processus de recherche correspondant aux buts *l2(b,Z)* et *l2(c,Z)*. Le premier processus de recherche trouve une seule clause atomique qui vérifie le but, et retourne une solution: *{X=a,Y=b,Z=d}*. Le second processus de recherche trouve deux clauses non atomiques et crée deux processus OU.
- Le premier processus OU crée deux processus ET, et envoie l'environnement *{X=b,Y=c}* au premier. Celui ci trouve deux solutions, les envoie au second processus ET, qui crée 'a son tour deux processus de recherche. Le premier ne trouve aucune solution. Le second trouve une seule solution, *{X=b,Y=c,Z=a}*, qui est retournée au littéral d'appel.
- Le second processus OU crée un processus ET, et lui envoie l'environnement *{X=b,Y=c}*. Celui ci crée un processus de recherche qui retourne la dernière solution: *{X=b,Y=c,Z=b}*.

Cet exemple montre un nombre relativement important de processus. Il est bien évident que seul un petit nombre de processus seront simultanément actifs à un moment donné.

Ainsi, le premier processus ET qui n'accède qu'à des faits se terminera très vite, vraisemblablement bien avant que les processus de recherche générés par le second processus ET aient pu accéder au disque. Ces processus de recherche, qui accèdent à la même clause, pourront éventuellement se "rejoindre" au niveau du disque, où ils donneront lieu alors à un seul filtrage.

A l'issue de ce filtrage, le premier processus de recherche sera terminé, alors que le second entraîne deux processus OU. Ceux ci peuvent éventuellement s'exécuter en total parallélisme, chacun générant à son tour des ensembles de processus ET et de processus de recherche. On pourra alors avoir deux processus de recherche portant sur deux clauses différentes. Soit ils pourront s'exécuter en parallèle sur deux disques différents, soit ils se retrouveront en concurrence sur le même disque. Dans ce dernier cas, ils seront traités dans l'ordre le plus favorable vis à vis du disque, ce qui permet en moyenne un gain de performances par rapport à une exécution strictement séquentielle.

V.4.3. Algorithmes des processus:

Dans ce paragraphe, on se bornera à donner une présentation informelle des algorithmes, sans se rattacher à un formalisme de communication prédéfini. On considère qu'il est possible à un processus:

- de recevoir des informations sans précision d'un processus émetteur, par la primitive: *recevoir(X)*, les données reçues étant retournées dans *X*. Chaque processus dispose ainsi d'un port d'entrée de messages sur lequel les messages peuvent être émis par plusieurs processus. Les messages sont traités en FIFO: ils sont stockés dans la file d'attente dans l'ordre dans lequel ils ont été émis. Le support de la communication est donc supposé préserver l'ordre d'émission des messages.

- d'émettre des informations vers un processus donné, par la primitive: *envoyer(numero_de_processus,donnée)*,
- de créer des processus fils en leur transmettant des paramètres, (primitive *créer(type_de_processus(paramètres))*, et en récupérant le numéro du processus créé.
- d'attendre que ses processus fils soient terminés (primitive *attendre*).

On utilisera aussi les primitives suivantes:

- *indexation(but)*: invoque le dispositif d'indexation de la base de données pour obtenir l'adresse disque (ou l'ensemble d'adresses) où la recherche du but doit être effectuée.
- *filtrage(but, adresse disque)*: crée un processus de filtrage qui recherche dans le bloc disque (ou les blocs disque) spécifié(s) par l'adresse (ou les adresses) les clauses dont l'en-tête unifie avec le but. Le processus de filtrage retourne la valeur FIN lorsque le filtrage est terminé. A noter que si la recherche peut être distribuée sur plusieurs disques, le processus de filtrage devra à son tour générer plusieurs processus fils, collecter leurs solutions pour les envoyer au processus de recherche, et attendre leur terminaison pour envoyer FIN.
- *appliquer(environnement, litteral)*: applique les substitutions contenues dans l'environnement aux variables du littéral, et retourne le but résultant.
- *compose(environnement, environnement)*: fusionne deux environnements, en appliquant les substitutions de l'un aux variables de l'autre et vice versa. Ainsi, *compose({X=b(Y,Z), T=a}, {Y=c, Z=T})* produit l'environnement: *{X=b(c,a), Y=c, Z=a, T=a}*.

V.4.3.1. processus ET:

Un processus ET reçoit des environnements, et applique les substitutions à un littéral L. PS est le processus suivant dans le pipe-line.

```
processus ET(L);
debut
  recevoir(PS);  /* numéro du processus suivant */
  recevoir(environnement_E);  /* recevoir le premier environnement */
  tant_que (environnement_E ≠ FIN) ;
  debut
    /* FIN indique qu'il n'y a pas ou plus de solution */
    B = appliquer(environnement_E,L);
    creer(recherche(B, PS, environnement_E));  /* démarrer un processus de recherche */
    recevoir(environnement_E); /* recevoir un nouvel environnement */
  fin;
  /* traitement de la dernière solution: attendre la fin des processus de recherche fils */
  attendre;
  /* signaler la fin du flot de solutions au processus suivant */
  envoyer(PS, FIN);
```

fin.

V.4.3.2. Processus de recherche:

Les processus de recherche reçoivent des buts B, et le numéro d'un processus PS auquel ils doivent transmettre leurs résultats.

```
recherche(B, PS, E);
debut
  filtrage (BI, indexation(B)) ; /* lancement du processus de filtrage */
  /* filtrage retourne:
     - des substitutions S
     - des queues de clause Q */
  recevoir({S,B});
  tant_que (S ≠ FIN)
  debut
    E2 = compose(E,S);
    /* si la clause est atomique, on envoie le résultat au processus destinataire */
    si (Q est vide) alors envoyer(PS, E2);
    Sinon creer(processus OU(PS, E2, Q));
    recevoir({S,B}); /* recevoir un nouveau résultat de filtrage */
  fin;
  /* attendre la fin des processus OU fils, ce qui permet au processus
       pere de savoir qu'aucune nouvelle solution n'est a attendre de ses
       processus fils, et donc d'envoyer FIN a son successeur */
  attendre;
fin.
```

V.4.3.3. processus OU:

Le processus OU reçoit en paramètre un environnement E, une queue de clause Q considérée ici comme une liste de NQ littéraux notés L[I], et le numéro du processus auquel il doit transmettre ses résultats, PS.

```
processus OU(PS, E, Q);
debut
  NQ = nombre de littéraux dans Q ;
  /* L [I] = Ième littéral dans Q */
  Pour I = 1 a NQ faire NUM_PROC[I] = creer(processus ET(L[I]));
  /* envoyer a chaque processus le numéro de son successeur */
  pour I = 1 a NQ - 1 envoyer(NUM_PROC[I], NUM_PROC[I + 1]);
  envoyer(NUM_PROC[NQ], PS); /* Processus ET destinataire */
  envoyer(NUM_PROC[1], E);  /* environnement pour 1er processus ET */
  envoyer(NUM_PROC[1], FIN);
  attendre;
```

fin.

V.4.4. Optimisations:

La stratégie de recherche que nous venons de définir peut conduire à un volume de communication important, du fait de la transmission des environnements entre les processus. Il est donc important de limiter au maximum les transferts de données inutiles: c'est la raison pour laquelle le modèle de communication que nous avons adopté permet le transfert direct d'une information entre un ou plusieurs processus producteurs et un processus consommateur, sans que ces informations soient collectées par un processus intermédiaire - ce qui serait requis par un certain nombre de modèles de communication interprocessus, tel celui d'Unix, par exemple.

Le transfert des environnements correspond en fait au transfert des résultats intermédiaires de la recherche: c'est donc un problème que l'on rencontre déjà dans les bases de données relationnelles. Cependant, en Prolog, un environnement (et les substitutions qui le composent) est en principe conservé pour la durée de l'évaluation de la clause. Une optimisation possible est donc de limiter la propagation des substitutions au seul domaine où elles sont utiles (dans une représentation des substitutions basée sur la copie des termes [MEL 82]).

L'optimisation proposée par Dang et Ianeselli [BER 86] repose sur une analyse des clauses permettant pour chaque littéral la classification des variables en six catégories:

- V.P (variable principale): variable libre qui sera utilisée dans la suite de l'évaluation. Ce littéral est le producteur des substitutions de la variable, ces substitutions doivent être transférées pour son successeur.
- V.S (variable secondaire): variable libre qui ne sera pas utilisée dans la suite de l'évaluation, il est inutile de générer les substitutions de la variable.
- V.I (variable immédiate): variable liée au moment de l'unification de l'en-tête de clause. La substitution peut être appliquée lors de la génération du processus ET, et n'est plus transmise dans les environnements.
- V.F (variable filtrée): variable liée par un filtrage de données dans un processus prédécesseur, elle sera utilisée dans la suite de l'évaluation. Ce littéral est le consommateur des substitutions générées par un des ses prédécesseurs . Les substitutions doivent être appliquées au squelette du prédicat pour construire les buts à vérifier. Les substitutions avec lesquelles les buts sont construits et vérifiés doivent être transférées pour son successeur.
- V.T (variable temporaire): variable liée par un filtrage de données dans un processus prédécesseur. Elle ne sera pas utilisée dans la suite de l'évaluation. Le traitement est le même que pour le type précédent, mais les substitutions seront jetées après la construction des buts.

- V.C (variable à copier): variable qui n'apparaît pas dans ce prédicat, elle est liée par un processus prédécesseur et les substitutions doivent être recopiées pour son successeur.

Prenons par exemple la clause:

c(X,Y,Z) :- l1(V,U,Z), l2(X,V,Z), l3(X,Y).

Et le littéral d'appel: c(terme,X1,X2). La classification des variables est la suivante:

Prédicat	V	U	X	Y	Z
l1	VP	VS	-	-	VP
l1	VT	-	VI	-	VF
l3	-	-	VI	VP	VC

D'autres optimisations peuvent être recherchées, mais n'ont pas été approfondies à ce jour. Considérons par exemple la clause suivante:

p(X) :- l1(X,Z), l2(Z,Y), l3(Y,X).

Pour chaque environnement reçu, le littéral l2 va transmettre éventuellement plusieurs solutions, donc plusieurs environnements comprenant la même substitution sur X et Z. Nous avons déjà vu que l'on peut éviter la transmission des substitutions de Z au delà de L2, mais il peut être aussi intéressant de ne transférer qu'une fois la substitution sur X, c'est à dire de la mettre en facteur: dans l'exemple ci dessus, supposons que le littéral L2 reçoive l'environnement: {X=a,Z=b}, et qu'il produise plusieurs solutions, avec les substitutions: Y=c, Y=d, Y=e. Il devrait donc transmettre trois fois la substitution sur X, à moins de coder les solutions de la façon suivante:

(X=a , ({Y=c},{Y=d},{Y=e})).

Une telle optimisation peut compliquer nettement le traitement des environnements, puisqu'un même processus peut recevoir des environnements totalement indépendants en provenance de plusieurs processus producteurs: il faut être en mesure de déterminer que les environnements en provenance d'un processus donné doivent être complétés par tel ensemble de solutions. Par contre, cela peut permettre d'optimiser le traitement des substitutions, en effectuant une seule fois l'application des substitutions de l'environnement ainsi mis en facteur. Il y a donc un choix que nous avons préféré laisser de coté pour l'instant, pour le reporter au niveau de l'implémentation.

V.5. IMPLEMENTATION:

La stratégie de recherche que nous venons de décrire a fait l'objet d'une expérimentation par simulation en langage OCCAM sur VAX VMS, conduite par Dang Weidong. Ce travail a fait l'objet de sa thèse de doctorat [DAN 87].

Basé sur CSP [HOA 78], le langage OCCAM a été développé par INMOS [WIL 83]. Son intérêt pour notre expérimentation réside dans le fait qu'il comporte les outils permettant de spécifier des processus parallèles communicants, ce qui permettait d'atteindre plus rapidement un résultat qu'en utilisant des outils de plus bas niveau, tels ceux d'UNIX, le but de l'expérimentation étant essentiellement une validation de la stratégie de recherche.

S'il fournit un modèle de communication commode, OCCAM a cependant imposé quelques modifications par rapport à notre modèle: ainsi, les processus de recherche ne peuvent pas transmettre directement leurs résultats au processus qui les consomme sans passer par leur processus père. Par ailleurs, l'implémentation d'OCCAM dont nous disposions ne disposait pas d'outils de mesure, ce qui a compliqué les évaluations, et entaché celles-ci d'incertitudes.

La suite de cette section fait largement référence aux travaux de Dang. Pour une description plus complète, se reporter à sa thèse.

V.5.1. L'arbre de processus:

La stratégie de recherche que nous avons précédemment décrite revient à définir un arbre de processus, dont les noeuds sont les processus ET, les processus OU, et les processus de recherche. Au niveau de l'implémentation, cet arbre se ramène à une structure de données traitée par ces processus.

En fait, l'opération effectuée par chaque processus est assez réduite: en particulier, les processus OU se contentent de créer des processus ET et d'attendre leur fin. Les processus ET à leur tour passent l'essentiel de leur temps à attendre la fin des processus de recherche qu'ils ont créé. La notion de processus joue donc essentiellement ici un rôle de synchronisation. A partir de cela, Dang remarque que l'information qui doit être stockée au niveau de l'arbre de processus est limitée aux processus ET - ou noeuds ET. L'exemple de la section précédente conduit à l'arbre de processus décrit dans la figure V.3.

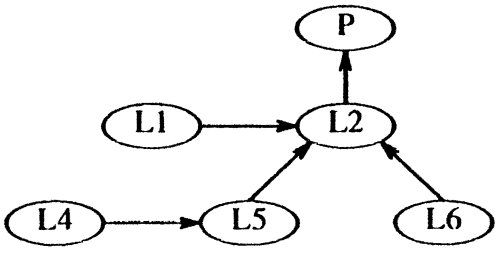


Figure V.3.

Les informations attachées à chaque noeud sont:

- Le nom du processus,
- Le squelette du prédicat,
- Le nombre de processus fils,
- Le nom du processus successeur (processus suivant dans le flot de données),
- Des informations d'état,
- La file d'attente des environnements reçus.

Les sous arbres d'un noeud sont ajoutés dynamiquement au cours de l'évaluation. Quand un processus de recherche trouve une clause non atomique, il crée un processus OU. Celui ci crée un noeud ET pour chaque littéral du corps de la clause, et écrit dans l'arbre de processus les substitutions générées par l'unification de l'en-tête de clause avec le but. Enfin, de nouveaux buts sont créés par application des substitutions sur les littéraux.

V.5.2. Répartition des processus:

L'algorithme de répartition des processus [DAN 87] a été conçu en vue de l'architecture multiprocesseurs d'OPALE, telle qu'elle est définie au chapitre III. La figure V.4 donne une vue générale de l'organisation des processus.

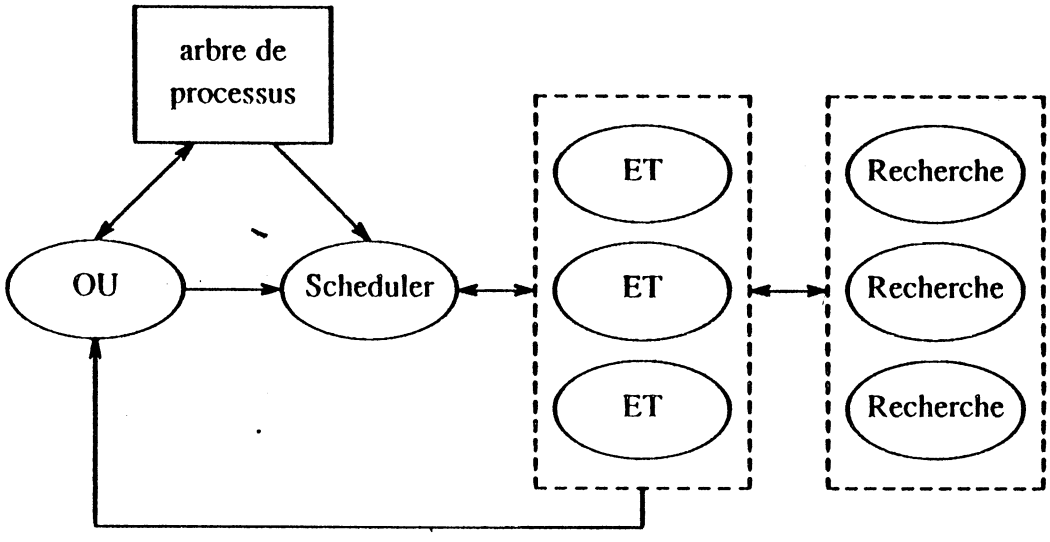


Figure V.4.

Si la base de données est stockée sur plusieurs unités de disques, il sera possible à plusieurs processus de recherche de s'exécuter simultanément. L'arbre de processus peut être stocké dans les unités de mémoire primaire qui permettent l'exécution d'accès associatifs. Les processus OU, ET, de recherche, et le scheduler sont alloués à des processeurs que l'on désignera respectivement processeurs OU, ET, de recherche, ou scheduler. Les processus de recherche déclenchent à leur tour des opérations sur les processeurs disques et les modules de mémoire primaire pour exécuter les indexations et les filtrages.

Le processus "scheduler" reçoit les requêtes en provenance de l'utilisateur, et les envoie à un processeur ET avec un environnement vide. Ce processeur crée un processus de recherche et lui transmet le but. Les résultats de la recherche sont ensuite transmis au processeur ET, qui les retransmet au processeur OU (l'implémentation en Occam ne permettant pas le passage direct du processus de recherche au processus OU).

Le processeur OU teste les données reçues: si la clause est un axiome, l'environnement reçu est ajouté à l'arbre de processus, et ceci est signalé au processeur scheduler. Sinon, de nouveaux processus ET sont créés, c'est à dire que de nouveaux noeuds sont insérés dans l'arbre de processus, et le scheduler est informé que ces processus ET peuvent être évalués.

Le scheduler a pour rôle l'organisation du pipe-line, et l'allocation des processeurs. Chaque fois qu'un processus ET est prêt à être évalué, il analyse ses successeurs et les place dans une liste de processus prêts. L'allocation des processeurs se fait en fonction du nombre de processeurs physiques disponibles, et du nombre de processus ET prêts. Les environnements générés par un processus ET sont transmis directement à son successeur, qui commence à s'évaluer dès qu'il a reçu un environnement.

L'arbre de processus peut être stocké sur plusieurs modules de mémoire primaire, pour permettre des accès multiples. Il peut aussi y avoir plusieurs processeurs OU ou processeurs schedulers si nécessaire, le système pouvant être configuré dynamiquement selon la distribution de la charge. Les processeurs OU, ET, de recherche, ou scheduler sont en effet identiques sur le plan matériel et ne sont spécialisés qu'au niveau de l'allocation. Dans sa thèse, Dang présente un algorithme permettant l'optimisation de l'allocation des processeurs aux différents processus.

Le volume de communication entre les processeurs constitue évidemment un problème pour cette stratégie. Or, il s'agit pour une bonne part d'une communication entre des processus constituant un pipe-line. Dang propose alors un double système de communication: une communication permettant à tout processeur matériel de communiquer avec n'importe quel autre processeur, et une communication en point à point reliant entre eux tous les processeurs de traitement. Selon la configuration, le premier système de communication pourra être un bus, série ou parallèle, ou un système de type commutation de circuit. Le second système peut être une ligne de communication rapide, ou encore une mémoire commune. L'algorithme d'allocation des processeurs a alors comme contrainte première d'allouer des processeurs voisins aux processus successifs d'un pipeline.

V.5.3. Résultats de l'expérimentation:

Les limitations de l'implémentation d'Occam dont nous disposions (une des premières), ainsi que des considérations de coût de calcul n'ont pas permis d'effectuer une campagne d'évaluation systématique et extensive. En particulier, l'absence d'outils d'évaluation dans le système Occam a imposé l'utilisation de méthodes heuristiques pour les évaluations (comptages).

Les résultats obtenus portent sur des configurations comportant un processeur OU, un processeur scheduler, et jusqu'à six processeurs ET et processeurs de recherche. L'exemple utilisé pour les évaluations est une base de données de relations familiales comportant trois ensembles de faits:

```
personne(Nom, Sexe, Age).

pere(Nom_pere,Nom_enfant).

mere(Nom_mere,Nom_enfant).
```

Et les règles suivantes:

```
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).

soeur(X,Y) :- parent(P,X), parent(P,Y), personne(X,feminin,_).
```

La requête: :- soeur(X,Y), print(X,Y).
a été exécutée et a retourné 17 solutions. Les résultats obtenus sont résumés dans le tableau suivant:

nombre de processeurs	2	4	6
temps d'exécution (*)	133	87	62
nombre de noeuds ET	52	52	52
Espace de travail (K Bytes)	5.7	3.6	3

(*) L'unité de temps utilisée est la durée moyenne d'un processus de recherche.

52 processus ET et 98 processus de recherche sont intervenus. Comme on pouvait l'espérer, l'accélération du traitement est à peu près proportionnelle au nombre de processeurs. Il est intéressant de noter que l'espace de travail utilisé diminue avec le nombre de processeurs (espace occupé par l'arbre de processus). Ceci est lié au fait que les processus restent en attente moins longtemps, et donc que l'espace qu'ils occupent est libéré plus rapidement, ce qui revient à diminuer le temps de résidence des résultats intermédiaires.

Pour des recherches complexes, l'espace occupé par l'arbre de processus devient important. Ce problème n'est pas nouveau dans le domaine des bases de données, car il correspond au stockage des résultats intermédiaires: ci ceux ci ne peuvent pas être conservés en mémoire primaire, il est parfaitement acceptable de les stocker temporairement sur disque.

V.5.4. Discussion:

Les évaluations présentées précédemment restent incomplètes, faute d'avoir pu les effectuer sur des exemples importants en volume ou en complexité. Il faut donc les considérer avec précaution. On peut toutefois conclure que la stratégie de recherche que nous avons présentée constitue une alternative intéressante aux stratégies classiques lorsque la base de clauses réside totalement ou partiellement sur disque.

Par ailleurs, cette stratégie ne représente pas une très grande complexité d'implémentation (cette implémentation est l'oeuvre d'une seule personne pendant quelques mois).

On ne peut cependant guère conclure en ce qui concerne l'efficacité en termes de temps d'exécution, l'interpréteur Prolog "bases de données" n'étant en effet pas comparable à un interpréteur effectuant son traitement en mémoire primaire, puisque son but est pour une bonne part d'optimiser les accès au disque, et l'implémentation effectuée restant rudimentaire au niveau de l'algorithme d'unification. D'autres résultats seront donnés au prochain chapitre, faisant intervenir une implémentation partielle de la stratégie de recherche et un algorithme d'unification d'ensembles de buts.

On remarquera aussi que si nous avons utilisé la notion de processus au niveau de la définition de la stratégie de recherche, ces processus ne se caractérisent en fait que par un faible volume de données auquel s'ajoutent les environnements manipulés. Ceci permet, au niveau de l'implémentation, de ramener le processus d'évaluation à une structure de donnée arborescente (l'arbre de processus) manipulé par un ensemble limité de processus réels correspondant aux processeurs physiques disponibles.

Le volume de données ajouté aux environnements par la gestion des processus reste assez faible: il semble en fait comparable au volume de données nécessaire dans la stratégie de recherche classique pour permettre le retour arriere (trace, empilement de points de chois, etc... Cf annexe II).

Enfin, l'utilisation d'Occam a amené a' un certain nombre de déviations par rapport à la définition initiale de la stratégie de recherche. L'utilisation d'outils de plus bas niveau en terme de communication pourrait permettre une implémentation plus fine et plus efficace.

V.6. RELATIONS AVEC UN INTERPRETEUR PROLOG:

Il semble logique de vouloir interfacer une base de données Prolog avec un interpréteur Prolog. Même si, dans le cas d'une machine spécialisée, la machine est consacrée à la base de données, il peut exister des cas qui relèvent d'une stratégie de recherche classique. Or, le changement de stratégie de recherche pose quelques problèmes de compatibilité. Un autre problème que nous avons déjà pu entrevoir concerne le traitement de certains prédicats évaluables dont on ne peut exclure totalement la présence dans une base de données, et plus particulièrement la coupure.

V.6.1. Changement de stratégie de recherche:

On peut considérer deux solutions pour le changement de stratégie de recherche: il peut se faire explicitement par le biais d'un prédicat évaluable, ou implicitement par le biais d'un attribut du paquet de clauses.

V.6.1.1. Changement de stratégie explicite:

C'est la solution que nous avons adoptée dans Opale, car elle semble la plus simple, et permet en outre de passer d'un environnement "programme", où tout est défini en mémoire primaire, à un environnement "bases de données" où la grande majorité des objets est définie en mémoire secondaire. Le problème est en effet qu'un programme Prolog serait pénalisé s'il devait connaître l'ensemble des symboles de la base de données. Le passage explicite à l'environnement bases de données permet donc, outre le changement de stratégie de recherche d'avoir accès à l'ensemble des symboles de la BD, et d'introduire les symboles lus dans l'environnement programme (stockage au niveau du dictionnaire en mémoire primaire).

On peut considérer deux types d'interface: l'un consiste à construire la liste des résultats de la requête bases de données: c'est le classique "setof". L'autre est moins conventionnel: il consiste à récupérer un par un les résultats de la recherche à l'aide d'un prédicat évaluable "backtrackable". Ce prédicat évaluable a un littéral comme argument. Au premier appel, il lance la vérification du littéral par la base de données, puis attend le premier résultat. Les résultats sont stockés dans une file d'attente où ils sont prélevés un par un lorsque le backtracking entraîne un nouvel essai du prédicat. Lorsque le dernier résultat est transmis, le prédicat retourne un échec.

Nous avons appelé "pipe" ce prédicat, pour mettre l'accent plus sur le changement de stratégie de recherche que sur le seul passage à l'environnement base de données. Le prédicat pipe a été implémenté dans Opale-r [BER 86]. Son utilisation permet donc de traiter les résultats de la base de données de la même façon que s'il s'agissait de clauses Prolog normales.

Exemple:

pipe (req (a , b, X)) lance la vérification de req sur la base de donnée. req peut être, en fait, une clause présente en mémoire (la requête), mais dont la vérification se fait en pipe-line et dans l'environnement base de données.

V.6.1.2. Changement de stratégie implicite:

Le changement de stratégie peut être non pas spécifié explicitement dans le programme, mais constituer un attribut d'un paquet de clause. L'appel d'un paquet de clauses déclaré comme étant une requête bases de données va donc impliquer automatiquement le changement d'environnement. Les résultats peuvent encore être transmis un par un, mais cette fois, c'est le mécanisme d'appel de la clause lui même qui devra s'occuper de chercher une solution en attente. On peut donc craindre qu'une telle solution n'alourdisse un peu l'interprétation, et donc diminue les performances.

V.6.2. La coupure:

Rappelons que la coupure, en Prolog, vise à éliminer les choix, c'est à dire indiquer en certains points de l'interprétation qu'aucune alternative n'est à vérifier. Elle peut jouer différents rôles: optimisation, négation, "si alors sinon", recherche de la première solution, etc... Nous la considérons comme rare dans une base de données, et on estime que d'autres mécanismes (encore à préciser) peuvent lui être substitués dans la majeure partie des cas. De plus, elle a souvent un rôle d'optimisation qui ne se justifie pas forcément avec la stratégie de recherche présentée ici.

L'existence d'une coupure dans un paquet de clauses doit se traduire par un strict ordonnancement des opérations. Les clauses doivent être accédées dans l'ordre (pas de parallélisme entre les unités disque), et les alternatives doivent être vérifiées les unes après les autres (pas de parallélisme OU). En cas de rencontre de la coupure, on efface de la file d'attente les alternatives restant à traiter. Ceci demande donc que la présence de la coupure dans un paquet de clauses soit signalée de façon à utiliser la stratégie de recherche convenable avant même la lecture de la coupure. Il s'agit donc d'avoir un attribut du paquet de clauses indiquant que l'ordre normal des opérations doit être conservé (ce qui peut s'appliquer à d'autres cas que la coupure). Il ne s'agit pas pour autant d'un retour pur et simple à la stratégie de recherche séquentielle, puisqu'il reste possible dans une certaine mesure de rechercher plusieurs en-têtes de clause lors d'un accès disque, et de rechercher des ensembles de cibles, la contrainte étant seulement de conserver l'ordre des résultats.

Exemples:

Considérons le paquet de clauses suivant:

```
c(a, X).
c(b, X).
c(c, X) :- !, c1(X, Y).
c(d, X) :- c2(d, X).
c(Y, X) :- c3(X, Y).
```

L'effet de la coupure dans la troisième clause est de définir deux classes dans le paquet de clauses: Les trois premières clauses sont toujours vérifiées, et les deux dernières ne sont vérifiées qu'en cas d'échec de la troisième (non unification de l'en tête de clause). On peut conserver au niveau de chaque classe la stratégie de recherche ensembliste , et accéder à l'ensemble des clauses en un seul accès disque (dans la mesure où cela ne coûte pas beaucoup plus cher).

Considérons maintenant ce qui se passe à l'intérieur d'une clause:

```
c(X) :- c1(X, Y), c2(Y, Z), !, c3(Z, T), c4(T, X).
```

Ici encore, la coupure définit deux sections à l'intérieur de la clause. Seule la première solution de c2 est utile: donc, pas de vérification globale à ce niveau. c1 par contre pourra transmettre plusieurs solutions à c2 avant qu'il puisse être vérifié, mais les solutions doivent être transmises à c2 dans l'ordre normal défini par la

stratégie "depth first". On ne peut donc pas au niveau de la vérification de c2 introduire de parallélisme OU ou de recherche parallèle sur plusieurs disques.

Il est en fait possible de maintenir l'ordonnancement des opérations en présence de OU parallélisme, si l'on introduit une synchronisation entre les processus: chaque processus OU ne transmet ses résultats que lorsque son prédécesseur a terminé de transmettre les siens. Le parallélisme n'existe alors que dans la mesure ou les processus peuvent conserver en mémoire les résultats, en attendant de pouvoir les transmettre: l'espace mémoire disponible limite donc le parallélisme effectif. La synchronisation entre les processus pourrait être assurée par leur père commun: dans notre modèle, le processus de recherche qui les a crées. Tout ceci suppose bien évidemment un overhead de communication et de synchronisation: on comprendra donc que l'on préfère s'en passer en considérant que ceci n'est éventuellement nécessaire que dans un nombre limité de cas.

V.6.3. Clauses récursives:

Le problème des récursivités est important dans les bases de données logiques. Ce point n'a pas fait l'objet d'une étude approfondie à ce jour. Il est certain que notre stratégie peut résoudre des clauses comportant des récursivités, mais il est probable que de telles clauses ne sont pas résolues de manière efficace. Dang propose dans sa thèse un algorithme permettant d'optimiser les cas de récursivité les plus simples (récursivités directes, avec présence possible d'une ou de plusieurs récursivités dans la clause) [DAN 87].

V.7. CONCLUSIONS

La stratégie de recherche présentée ici permet un accès performant à des bases de données Prolog, en utilisant une technique d'unification au vol d'ensemble de buts. Cette technique fera l'objet du prochain chapitre.

On peut ainsi obtenir un accès optimal aux disques, sans avoir besoin de stocker de gros volumes de données à chaque étape du traitement. Un alternative à la stratégie proposée serait en effet d'utiliser une stratégie de recherche classique, mais en mémorisant les solutions obtenues lors de chaque accès disque, le retour arrière permettant de venir récupérer le résultat suivant. Notre solution permet de limiter le volume de résultats intermédiaires stockés, et de limiter la durée de transit des informations en mémoire primaire, donc d'optimiser l'utilisation de celle ci. De plus, ceci est obtenu directement au niveau de la technique d'interprétation, sans rajouter une couche fonctionnelle au système.

Par contre, le fait que les résultats partiels ne soient pas en principe stockés en totalité fait qu'il ne sera pas possible de réutiliser un résultat précédemment amené en mémoire.

La technique du pipe line permet de tirer un parti maximum des ressources disponibles, avec un parallélisme réel relativement faible (accès simultané à plusieurs disques, ou traitement sur plusieurs processeurs). Ceci permet donc une implémentation relativement légère, dans la mesure où nous avons vu qu'il était possible de limiter assez

fortement le nombre réel de processus.

Le traitement en pipe line des littéraux a été introduit dans un interpréteur Prolog (appelé OPALE-R, le R signifiant "réduit") utilisant l'algorithme d'unification qui sera exposé dans le prochain chapitre. Dans cette implémentation, on s'est limité à l'accès à des bases de faits, l'accès aux règles étant une étape ultérieure du projet. Cette implémentation réduite démontre cependant l'intérêt et l'efficacité combinée de cette stratégie de recherche et de l'algorithme d'unification.

La suite du travail devrait donc consister en une implémentation de la stratégie de recherche dans un interpréteur Prolog utilisant l'algorithme d'unification.

Chapitre VI

UNIFICATION

Avertissement:

Les travaux présentés dans ce chapitre sont le résultat d'une collaboration entre l'auteur et Jean Christophe Ianeselli, et ont fait l'objet de sa thèse de troisième cycle. Il n'est pas toujours possible d'attribuer avec précision a l'un d'entre nous la paternité des idées présentées. Toutefois, on peut dire que l'idée générale de décomposition de l'unification, puis de la préunification est de moi même, J.C Ianeselli ayant contribué à préciser l'algorithme (codage des listes de buts, génération de la structure englobante, traitement des listes). Il a également effectué la programmation en C de l'algorithme et son évaluation, avec l'aide de Weidong Dang, et travaillé à la définition matérielle du filtre. Ceci étant précisé, ces travaux seront présentés de manière impersonnelle dans la suite, pour éviter d'en alourdir la présentation.

VI.1. ALGORITHME:

VI.1.1. Introduction:

Le problème que l'on va maintenant se poser consiste à exécuter au vol l'unification entre un ensemble de buts, et un flux de données lu sur un disque: l'ensemble des buts est disponible dès le début de l'opération, alors que les en-têtes de clauses défilent les uns après les autres au fur et à mesure de leur lecture sur le disque. A tout instant, on a donc bien un ensemble de buts à unifier avec un seul en-tête de clause. Nous avons vu au chapitre précédent l'importance d'une telle opération.

L'unification consiste à comparer deux termes, incluant des variables, et à tenter de les rendre égaux en substituant les variables avec une valeur. Elle a pour résultat la liste des substitutions à effectuer sur les variables pour identifier les deux termes. Considérons la grammaire suivante, définissant un terme:

<Terme> ::= <Atome> | <Symbole_fonctionnel> (< Suite-de-termes>)
| variable
<Suite-de-terme> ::= <Terme> | <Terme>,<Suite-de-termes>
<Atome> ::= Constante_symbolique | <Donnée_élémentaire>
<Symbole_fonctionnel> ::= Constante_symbolique
<Donnée_élémentaire> ::= nombre | caractère | chaîne

Un algorithme d'unification est le suivant:

```
fonction unifie(t,u);
debut
    si t est un atome alors
        si t = u alors retour(succès) sinon retour(échec);
    si t est une variable alors
    debut
        substituer t par u dans toutes ses occurrences;
        retour(succès);
```

```
    fin;
    si u est une variable alors
    debut
        substituer u par t dans toutes ses occurrences;
        retour(succès);
    fin;
    si t est un arbre et u est un arbre alors
    debut
        si symbole_fonctionnel(u) ≠ symbole_fonctionnel(t) alors retour(échec);
        si nombre_de_fils(u) ≠ nombre_de_fils(t) alors retour(échec);
        pour i = 1 jusqu'a nombre_de_fils(t) faire
        debut
            ft = fils(t,i); fu = fils(u,i);
            si unifie(ft,fu) = échec alors retour(échec);
        fin;
    fin sinon retour(échec);
fin.
```

- La fonction fils(t,i) retourne le ième fils du terme t.
- La fonction nombre_de_fils(t) retourne le nombre de fils du terme t.
- La fonction symbole_fonctionnel(t) retourne le symbole fonctionnel du terme t.

Exemple:

On notera U(t1,t2) l'unification de deux termes t1 et t2, et S(a,b) la substitution de deux termes dont un au moins est une variable.

L'unification:	U(terme(a(e,f),b(X,c(Y,g)))), terme(Z,b(T,c(k,g))))
	(X,Y,Z et T étant des variables)
produit les substitutions:	S(Z,a(e,f)); S(Y,k); S(X,T)

Les noeuds d'un terme sont définis par leur symbole fonctionnel et leur arité (nombre de fils). Deux noeuds sont donc égaux s'ils ont même symbole fonctionnel et même arité. Les feuilles peuvent être des symboles (d'arité nulle), des variables ou des données.

L'unification a fait l'objet de travaux assez nombreux, ceci principalement dans le cadre des systèmes de réécriture, ou la démonstration de théorèmes. Ainsi, l'algorithme de Martelli et Montanari [MAR 82] s'intéresse à la résolution de systèmes d'équation, c'est à dire à la recherche d'un unificateur (ensemble de substitutions) commun à un ensemble d'unifications. Autrement dit, étant donné N paires de termes, trouver l'ensemble de substitutions (s'il existe) permettant d'unifier les termes deux à deux. D'autres travaux relatifs à l'unification ont été effectués en France, en particulier par Huet [HUE 76] et par Kirchner [KIR 85].

Il semble cependant que, dans le cadre de la programmation en logique, l'algorithme de Robinson soit d'un usage quasi général: en effet, il s'agit là le plus souvent (comme nous le verrons au chapitre VII) de cas d'unifications très simples, pour lesquels la simplicité de l'algorithme compte bien plus que sa performance intrinsèque. Le cas des bases de données reste très proche de cette problématique, et le problème pour nous réside plus dans la possibilité d'implémentation matérielle, et d'exécution "au vol" de l'opération.

VI.1.2. Unification dans OPALE:

Le problème fondamental est le traitement des substitutions. En effet, une variable peut être répétée plusieurs fois dans un terme, et il est nécessaire, pour exécuter l'unification complète, de répercuter immédiatement toute substitution sur l'ensemble des occurrences de la variable, de sorte qu'elle soit prise en compte dans la suite de l'unification lorsque la variable est à nouveau rencontrée.

Ceci complique très fortement l'unification, et il est nécessaire de faire appel à des algorithmes ou à des structures de données complexes pour résoudre le problème. Dans les interpréteurs Prolog, ceci se complique du fait qu'on ne modifie pas le terme, de façon à pouvoir "défaire" les substitutions au backtracking.

La solution est de considérer le terme avec l'ensemble des substitutions produites. Chaque fois que l'on rencontre une variable dans le terme, on cherche s'il existe une substitution sur cette variable, et si oui, on l'applique. Donc, au niveau de l'accès au terme, on doit passer par le mécanisme de substitutions, et lorsqu'une substitution est produite, elle est stockée dans une table. Les variables sont codées sous forme d'un index dans cette table.

Un terme substitué à une variable peut à son tour comporter des variables. La substitution devra donc comporter non seulement un pointeur sur ce terme, mais également un pointeur sur la table permettant de définir ces variables. Une substitution est donc un doublet comportant un pointeur sur un terme (modèle de terme), et un pointeur sur la table où sont définies éventuellement ses variables. Dans Prolog, les tables de substitutions correspondent à des états du contexte et sont empilées pour permettre le backtracking.

La représentation des substitutions dans les interpréteurs Prolog se fait selon deux techniques [BRU 82]. La plus populaire jusqu'à aujourd'hui est probablement le partage de structure [BOY 72]: la substitution d'une variable par un terme composé est représentée par un doublet comportant:

- un pointeur sur le "squelette" du terme dans le programme (c'est à dire la représentation du terme dans le code lui-même). Ce squelette peut comporter des variables, libres ou non.
- un pointeur sur l'environnement correspondant, permettant d'accéder à la liaison de ces variables.

La seconde technique est la recopie [MEL 82]: celle ci consiste à empiler une copie du terme. Si ce terme comporte des variables liées, les termes correspondants sont également recopiés. La substitution des variables libres consistera donc à les remplacer par un pointeur sur la copie de terme correspondante. La comparaison des deux techniques ne montre pas de supériorité absolue d'une technique sur l'autre, leur intérêt respectif étant fonction de la machine hôte, et de l'importance relative que l'on accorde à la rapidité et à l'utilisation de la mémoire.

Dans une unification, on considère généralement au moins deux contextes: le contexte correspondant au but (clause appelante), et le contexte correspondant à la clause appelée. La figure VI.1. présente, de façon très simplifiée le résultat de l'unification des termes: t(a(b,X),Y) et t(X,d(e,f)), en utilisant le partage de structure.

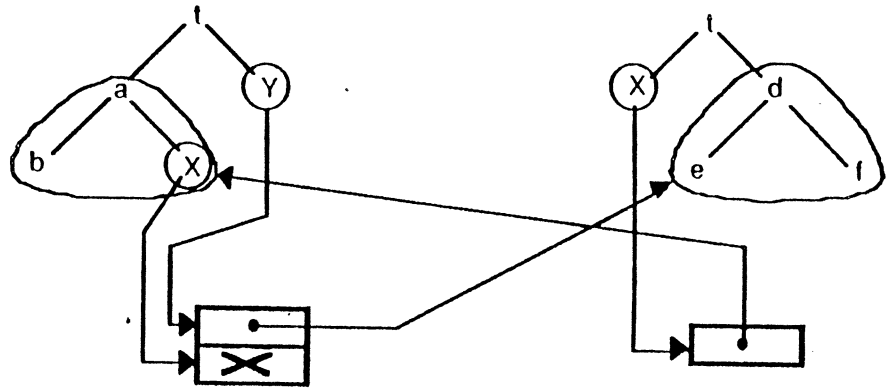


Figure VI.1.

L'utilisation d'une telle structure de données est donc très lourde, et il n'est guère concevable de réaliser une telle opération au vol sur un disque: la substitution d'une valeur à une variable peut en effet entraîner une modification complexe des buts, ce qui n'est pas réalisable par un automate fini. De plus, l'unification globale d'un ensemble de buts avec un en-tête de clause semble difficilement concevable. On va donc chercher à décomposer l'unification en une opération préliminaire exécutable au vol par un opérateur implémenté en matériel (la préunification), et une opération complémentaire, exécutée par un algorithme implémenté en logiciel.

VI.1.3. Préunification:

Quel que soit un élément (noeud ou feuille) N d'un terme T, on définit sa position, notée *pos(N,T)* comme étant le chemin à parcourir depuis la racine pour l'atteindre: ce chemin peut être représenté par la suite des rangs de chaque élément parmi les fils du noeud père. Ce codage correspond en fait à la succession des valeurs de i (indice des fils) dans les niveaux de récursivité de l'algorithme d'unification.

Si R est la racine du terme T, alors $pos(R,T) = \{\}$. Si N_i est un noeud ou une feuille de T, N_{i-1} est son noeud pere, $rang(N)$ est le rang du noeud N parmis les fils de son noeud père, et *concat* est l'opérateur de concaténation de listes, alors $pos(N_i,T)=concat(pos(N_{i-1},T),rang(N_i))$

Exemple:

La position du noeud d'étiquette *l* dans le terme
a(b(e,f),c(g(j,k,l),h),d(i)) est: $\{2,1,3\}$.

Une condition nécessaire (mais non suffisante) pour que deux termes T_1 et T_2 soient unifiables est la suivante:

quelque.soit ($N_1 \in T_1, N_2 \in T_2$ tq pos(N_1, T_1) = pos(N_2, T_2)),
Si $N_1 \neq N_2$ alors (N_1 est une variable ou N_2 est une variable).

La condition est nécessaire d'après l'algorithme de l'unification, mais elle est non suffisante.

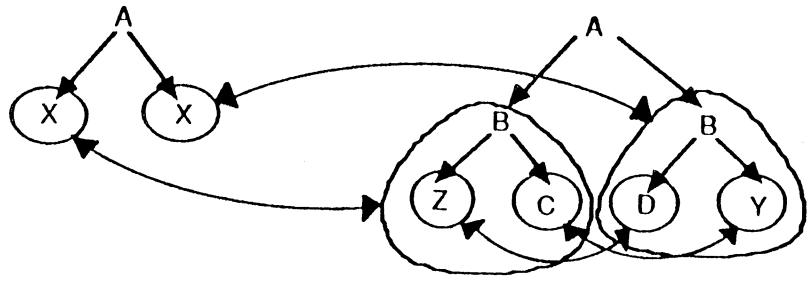
Contre exemple:

Les deux arbres *a(X,X)* et *a(b,c)* remplissent la condition
mais ne sont pas unifiables.

On appellera condition de préunification cette condition, et deux arbres qui la remplissent seront dits préunifiables. Il est clair que si 2 arbres sont préunifiables, et si chacun d'eux ne contient qu'une occurrence de chaque variable (ses variables sont indépendantes), alors ils sont unifiables. Rappelons qu'en Prolog, les variables sont locales, ce qui signifie qu'un but et un en-tête de clause ne peuvent partager de variables que du fait des liaisons générées lors de l'unification.

Deux arbres préunifiables sont unifiables si chacun des sous arbres substitués à une même variable sont unifiables entre eux (d'après l'algorithme d'unification). En effet, dans l'algorithme d'unification, si la première occurrence d'une variable d'un terme est substituée avec un sous terme T, à la seconde occurrence, c'est ce sous terme T qui sera à son tour unifié avec le sous terme correspondant de l'autre terme.

Exemple:



On appelle préunification l'opération consistant à vérifier que deux arbres sont préunifiables et à produire les couples variable- sous arbre qui les distinguent. En d'autres termes, la préunification est l'opération qui, étant donné deux arbres retourne un échec si deux éléments constants de même position sont différents, et sinon retourne le désaccord des deux termes (liste de leurs différences). L'algorithme de préunification est identique à l'unification, à ceci près que les substitutions ne sont pas répercutées sur toutes les occurrences des variables dans les arbres en cours d'unification: on peut donc simplifier considérablement l'implémentation de l'opération, ou la représentation des termes.

Lorsque deux arbres sont préunifiables, on peut faire appel à l'unification pour vérifier s'ils sont unifiables, les unifications portant alors sur des sous arbres sont donc plus simples. Ces nouvelles unifications peuvent à nouveau faire appel à la préunification. Comme un arbre fini ne comporte qu'un nombre fini de sous arbres, l'unification peut se ramener à un nombre fini de préunifications. Ceci équivaut en fait à un simple changement dans l'ordre de parcours de l'arbre: on traite d'abord les arbres initiaux, puis les parties crées par les substitutions, et ainsi de suite. En fait, dans le cas général, on utilisera l'unification classique dès la seconde étape du traitement.

La décomposition préunification-unification peut être en fait assimilée à une réécriture des buts et des en-têtes de clauses: les occurrences multiples d'une même variable sont remplacées par une variable unique, puis on unifie entre elles les variables substituées à une même variable.

Exemple:

Soit l'en-tête de clause: c(X,a(b,Y),c(X,Z,Y)):-...
On peut le réécrire: c(X1,a(b,Y1),c(X2,Z,Y2)):- X1=X2, Y1=Y2,...
L'opérateur "=" dénotant l'unification.

Pour un but, les unifications sont, de la même façon insérées en tête de la liste des buts à vérifier, donc devant la queue de la clause appeleée.

VI.1.4. Discussion:

L'introduction de la préunification repose sur l'idée que les dépendances entre variables jouent un faible rôle dans la sélectivité de l'unification. Cette idée est confirmée par une série de mesures effectuées sur l'interprétation de Prolog, comme nous le verrons au prochain chapitre: 80% des cas d'échec de l'unification correspondent également à des échecs de la préunification. En outre, la complexité de l'association est le plus souvent très faible: dans 75% des cas, elle se ramène à la génération de substitutions, et dans moins de 10% des cas, elle implique une unification entre des termes non atomiques.

On rencontre en fait deux cas:

- Soit les dépendances ont pour rôle de sélectionner un terme comportant deux champs de valeurs unifiables: chercher à unifier le but t(X,abc,X) avec un en-tête de clause revient à vérifier que le premier et le troisième argument de l'en-tête de clause sont unifiables.
- Soit elle a pour rôle de construire un résultat en transférant la valeur d'un argument dans un autre argument: il semble que ce cas soit le plus fréquent.

VI.2. UNIFICATION ET BASES DE DONNEES:

VI.2.1. Problème:

Un aspect important de l'unification dans les bases de données résulte de l'utilisation de disques, et de la stratégie de recherche adoptée. Lorsque l'on accède au disque, il est nécessaire d'en retirer un maximum d'informations.

Ceci a deux conséquences:

- Lors d'un accès disque, on recherchera non pas le premier terme unifiable dans un paquet de clauses, mais tous les termes unifiables dans le bloc (piste, ou groupe de secteurs) accédé.
- De même, on s'attachera à rechercher non pas un seul terme but, mais un ensemble de buts portant sur le même paquet de clauses. L'algorithme doit donc permettre une unification d'un ensemble de buts avec un flux de données.

Le fait de vouloir exécuter l'unification très rapidement rend impossible le recours à une structure de données complexe pour l'accès aux termes. On admettra dans ce qui suit que les buts sont disponibles, au niveau de la base de données, sous forme de termes plats, c'est à dire définis sans faire appel à des substitutions: ainsi, le terme *a(X,Y)*, accompagné des substitutions: *X <- d, Y <- c(Z,d)* est mis sous la forme: *a(d,c(Z,d))*.

Dans une base de données, on peut considérer que la sélectivité de l'unification sera très forte, autrement dit, beaucoup de tentatives d'unification seront des échecs, même si l'indexation permet dans une large mesure de réduire cette proportion. Il semble nécessaire de s'intéresser principalement à l'aspect sélection, la manipulation des substitutions n'étant nécessaire que dans les cas où l'unification réussit. Il sera intéressant d'avoir un algorithme efficace (éventuellement câblé) permettant d'éliminer une grande partie des termes non unifiables, même si un travail conséquent mais rare doit ensuite

être exécuté pour terminer l'unification.

Ainsi, si: U est la durée moyenne d'une unification complète, T le taux d'échec de la sélection, S la durée moyenne d'une sélection, C la durée moyenne du traitement complémentaire en cas de sélection, et U_2 la durée moyenne de l'unification avec sélection, On a alors: $U_2=S+(1-T)xC$. Si on prend par exemple: $T = 0,9$; $S = 0,2\ U$ et $C = U$; on obtient $U_2=0,3U$.

On voit donc que cette décomposition de l'unification est susceptible d'apporter un gain appréciable en performances, directement lié au facteur de sélectivité (rapport entre le volume de données lues et le volume de données transmises) et à l'efficacité de l'algorithme de sélection (le traitement complémentaire étant dans notre exemple supposé de même coût que l'unification totale). Ce gain sera évidemment encore plus important si les deux opérations peuvent être exécutées en pipe-line par deux processeurs.

VI.2.2. Indexation:

L'accès à de gros volumes d'information résidant sur disque implique généralement l'utilisation d'une indexation permettant de limiter le domaine de recherche en fonction d'un ou de plusieurs éléments de données.

La présence de variables dans les en-têtes de clauses complexifie l'indexation, puisque une même cible peut unifier soit avec des en-têtes de clauses qui présentent les mêmes valeurs constantes, soit avec des en-têtes de clauses qui présentent des variables. Un stockage séparé peut être rendu nécessaire, avec l'utilisation de techniques d'indexation différentes.

La technique couramment utilisée dans les interpréteurs Prolog consiste à indexer sur le symbole fonctionnel du premier argument de l'en-tête de clause: c'est la solution la plus simple, qui a l'inconvénient d'être inefficace dès que la question ne fixe pas la valeur de ce symbole, ou lorsque le premier argument a le même symbole fonctionnel pour un grand nombre de clauses (cas des listes).

Il est également possible de recourir à des techniques d'indexation multicritères, permettant de cerner un certain ensemble d'éléments à partir de la valeur d'un ou plusieurs arguments: grâce au filtrage, on n'a pas besoin de connaître l'adresse disque précise de la donnée cherchée, mais l'adresse d'un bloc (piste) qui la contient.

Beaucoup de travaux ont été faits sur les méthodes d'indexation mono ou multicritères, ainsi que sur l'indexation dans le cas particulier de Prolog. Wise et Power présentent une méthode d'indexation consistant à coder l'ensemble des éléments des en-têtes de clause sur une chaîne de bits, en allouant quelques bits pour chaque élément [WIP 84]. Cette technique autorise la présence de variables dans les en-têtes de clauses.

VI.3. FILTRAGE PAR AUTOMATES D'ETATS FINIS:

Dans les bases de données relationnelles, le filtrage par automates d'états finis apparaît comme un outil bien adapté. Il existe cependant deux différences essentielles entre le filtrage des bases de données relationnelles et l'unification:

- manipulation de données structurées en arborescences, et non plus seulement de données atomiques: On peut dire que, sur ce point, les travaux effectués, en particulier dans le projet VERSO [BAN 80, TUS 80], ont montré qu'il est possible de balayer des arborescences à l'aide d'automates d'états finis. Dans les bases de données, la structure est généralement prédéfinie et son analyse peut être confiée à l'automate lui même. En PROLOG, ce sont les données elles mêmes qui indiquent la structure: en particulier dans Opale, l'arité d'un terme est indiquée par l'octet de préfixe du symbole fonctionnel. Lorsqu'une donnée non atomique (arbre) est associée à une variable d'un terme but, il va donc falloir décoder la structure des informations pour reconnaître la fin d'un terme à transmettre. Ceci peut être effectué par un petit automate ajouté à l'automate de recherche.
- possibilité d'avoir des variables, non seulement dans les buts (où elles représentent les données inconnues à extraire de la base) mais aussi dans la base de données.

Les variables dans la base de données peuvent apparaître dans des règles: on doit bien admettre que, dans une base de connaissances, les ensembles de règles de même nom puissent être suffisamment importants pour que le filtrage soit nécessaire, ceci étant d'autant plus vrai que la présence de ces variables peut diminuer beaucoup l'efficacité de l'indexation.

Les variables peuvent également apparaître dans des faits où elles pourront avoir valeur de quantificateur universel: la clause *C(a,b,X).* signifie que le fait est vrai quelle que soit la valeur associée à X.

Un exemple peut être donné dans le domaine du traitement des langues naturelles: on considère une base de donnaissance permettant de reconnaître des mots d'un langage. Pour chaque mot du dictionnaire, on aura la racine du mot, sa forme de base (l'infinitif, pour un verbe), la terminaison, et ses règles d'accord ou de conjugaison. Pour le verbe *aimer* on aura par exemple:

mot(['a','i','m'/Termin],Termin,verbe(aimer,groupe1)).

On peut choisir de retourner l'identification des règles de terminaison, comme ci-dessus, ou de mettre l'invocation de ces règles directement dans la queue de la clause:

mot(['a','i','m'/Termin],verbe(aimer,Forme)):- conjugaison(groupe1,Termin,Forme).

où l'appel de conjugaison vérifie si Termin est une terminaison valide d'un verbe du premier groupe, et retourne la forme (personne et temps) correspondante. Le français comporte plusieurs dizaines de milliers de mots différents, il s'agit donc d'une base de données faisant intervenir des règles, ou des faits comportant des variables en nombre élevé.

Ces considérations nous amènent donc à ne pas nous limiter à un filtrage classique (sans variables dans la base de données), mais à chercher à réaliser au vol l'ensemble de l'unification.

Or, d'une part, le stockage des termes sur le disque amène à utiliser une structure de représentation simple (arborescences linéarisées avec délimiteurs), et d'autre part, l'exécution de l'unification au vol nécessite un algorithme simple pouvant être câblé.

En reprenant les résultats du paragraphe précédent, on se propose de décomposer l'unification en deux étapes:

- Une préunification exécutée par un automate câblé, qui va donc sélectionner les termes préunifiables, et produire les couples variable/valeur qui expriment les différences entre le terme lu et le ou les buts. On appellera substitutions ces couples de valeurs.
- Le reste du traitement, que nous appellerons association, consiste à contrôler et composer les substitutions, en exécutant, le cas échéant, des unifications classiques. Il concerne un flot de données qui est normalement considérablement réduit par la préunification: on considère donc qu'il peut être exécuté par logiciel.

Dans le traitement des substitutions, seuls les termes lus sur disque et substitués à une variable d'un ou plusieurs buts doivent être transmis. Lorsqu'une variable est lue sur disque, il suffit de transmettre la position correspondante dans les buts, pour éviter une transmission inutile, et qui peut être longue dans certains cas (cas où une variable d'un en-tête de clause est substituée par un gros sous terme d'un ou de plusieurs buts).

Exemples:

but:	t(a,b(c,d),e)	*On transmet X et un code permettant*
terme lu:	t(a,X,e)	*de retrouver la valeur b(c,d)*
buts:	t(a,b,c)	*On transmet X et un code permettant*
	t(a,e,c)	*de retrouver les valeurs b et e*
terme lu:	t(a,X,c)	

VI.3.1. Automates d'états finis:

Il est assez facile de vérifier qu'on peut préunifier un ensemble de buts avec un terme par utilisation d'automates d'états finis.

Prenons un terme but: $t(a_1,...,a_n)$
Où les a_i sont des atomes, des arbres ou des variables.

Reconnaître un terme qui préunifie avec celui-ci consiste donc à reconnaître:

- soit une variable
- soit le symbole fonctionnel t, suivi de successivement (pour $i = 1$ jusqu'à n
  * soit une variable
  * soit la valeur a_i
  * soit un terme préunifiable avec a_i

Le processus est récursif pour les a_i qui sont des arbres.

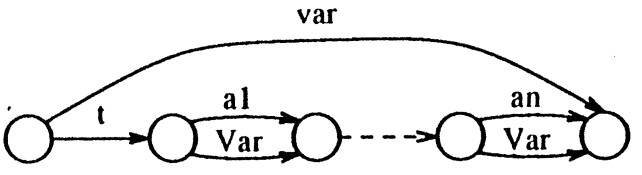
Pour les a_i qui sont des variables, on doit accepter un terme quelconque, dont on ne connaît pas la structure. Cette structure étant contenue dans les données elles mêmes, il

faut être capable de décoder le type des données et la structure des termes, ce qui peut être fait par un opérateur supplémentaire.

La reconnaissance d'un terme consiste donc à satisfaire l'expression suivante:

lire (variable ou (t et lire (a_1 ou variable) et ... et lire (a_n ou variable))).

Ce type d'expression peut être transformé en automate d'états finis:



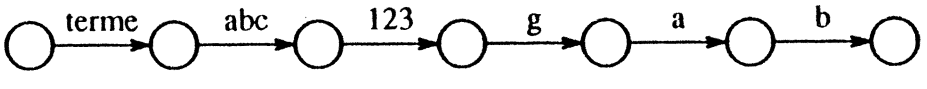
Chaque transition correspond à un élément du terme lu. Pour une variable du but, on lit un terme quelconque. Pour ignorer un argument, on doit utiliser la notion de variable anonyme de PROLOG: une telle variable n'est employée qu'une fois, et il est inutile de lui substituer une valeur.

Pour un ensemble de buts, le problème est assez simple si on ignore la possibilité de variables sur le disque: on se ramène à la vérification d'un OU sur un ensemble de critères, problème assez classique dans les BD relationnelles, la seule difficulté étant qu'il faut savoir quel but a été vérifié. Cependant, dès qu'on admet la présence de variables sur le disque, le problème se complique, et plus particulièrement lorsque plusieurs buts sont attendus. Dans la suite, nous allons détailler les différents cas susceptibles d'être rencontrés.

VI.3.1.1. Pas de variables:

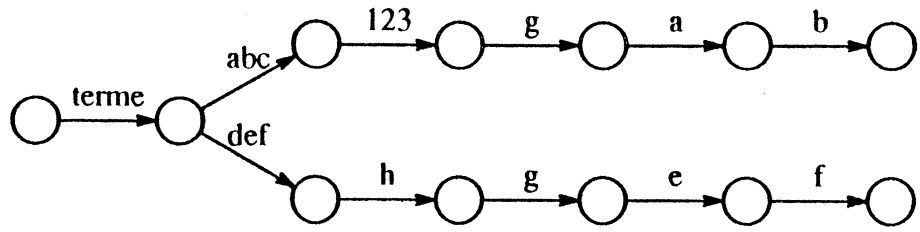
Ce cas revient à comparer deux chaînes et à décider si elles sont égales. Le cas d'un seul but donne un automate tout à fait linéaire:

but: *terme(abc,123,g(a,b))*



La recherche de plusieurs buts revient à une disjonction:

buts: *terme(abc,123,g(a,b))*
terme(def,h,g(e,f))



VI.3.1.2. Variables seulement dans les buts:

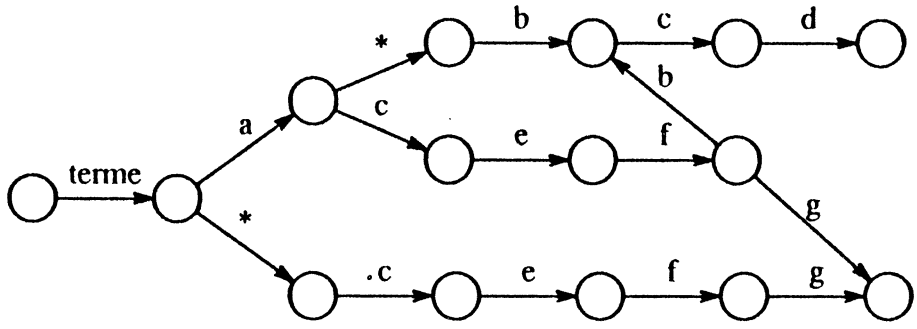
Ce cas revient à la sélection des bases de données: les constantes du but correspondent aux attributs de sélection, et les variables aux attributs dont on recherche la valeur.

La seule difficulté supplémentaire introduite est la structuration des données. En effet, en l'absence de variables, on recherche une chaîne exactement égale à une chaîne donnée, et la structuration de cette chaîne n'intervient pas. Lorsqu'il existe des variables dans les buts, on peut associer à une variable un terme complexe, qu'il faut être en mesure d'analyser pour en détecter la fin. Ceci demande en fait un automate assez simple: on utilise un compteur, initialisé à 1 (nombre de termes à lire); pour tout élément de donnée lu, on retranche 1 au compteur; et pour tout élément non atomique (symbole fonctionnel) on ajoute son nombre de fils. La fin du terme est détectée lorsque le compteur est à zéro.

L'extension au cas de plusieurs buts n'introduit encore que peu de complexité supplémentaire. La difficulté est de pouvoir déterminer lequel des buts unifie (ou préunifie) avec le terme lu en cas de succès: cette information peut être déduite de l'état final atteint par l'automate.

Exemple:

1: terme(a,X,b(c,d))
2: terme(X,c(e,f),g)



(On dénote par * l'acceptation d'une donnée (simple ou complexe) quelconque.)

Dans notre exemple, le fait que les deux buts comportent une variable dans des positions différentes introduit une complexité qui ne correspond pas au cas le plus courant dans la réalité. En effet, les différents buts seront le plus souvent engendrés par

l'application de substitutions à un même littéral, ces substitutions étant elles mêmes le produit de la vérification d'un autre littéral. Le cas courant consiste donc à avoir des buts comportant des variables aux mêmes positions:

Prenons par exemple la clause:

C(X,Y) :- L1(a,X), L2(X,c,Y).

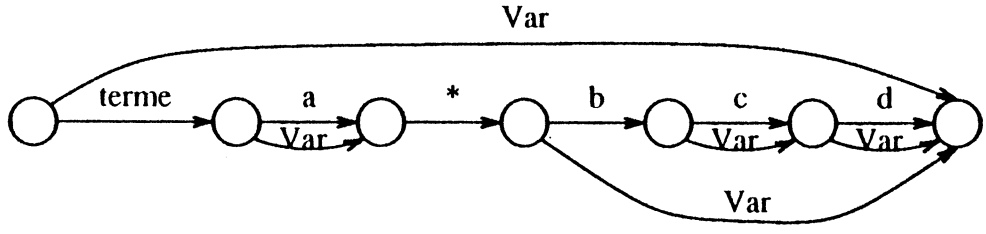
Différents buts seront produits à partir de *L2* et des substitutions sur *X* générées par *L1*. Si *X* est toujours complètement instancié (c'est à dire lié à un terme ne comportant pas de variables), tous les buts auront une variable dans la même position, correspondant à *Y*. Ce n'est que dans les cas où *X* est partiellement instancié que l'on risquera de voir apparaître des variables dans des positions différentes. Ce cas peut sans doute être considéré comme moins fréquent que le premier, mais pas exceptionnel, dès lors que l'on manipule des données non atomiques (listes, arborescences...).

VI.3.1.3. Variables dans les buts et dans les en-têtes de clauses:

Pour un seul but, le problème reste simple: pour tout état de l'automate, on doit pouvoir accepter une valeur donnée ou une variable. Ceci peut être compilé dans l'automate, en ajoutant à chaque état une transition correspondant à la lecture d'une variable.

Exemple:

terme(a,X,b(c,d))

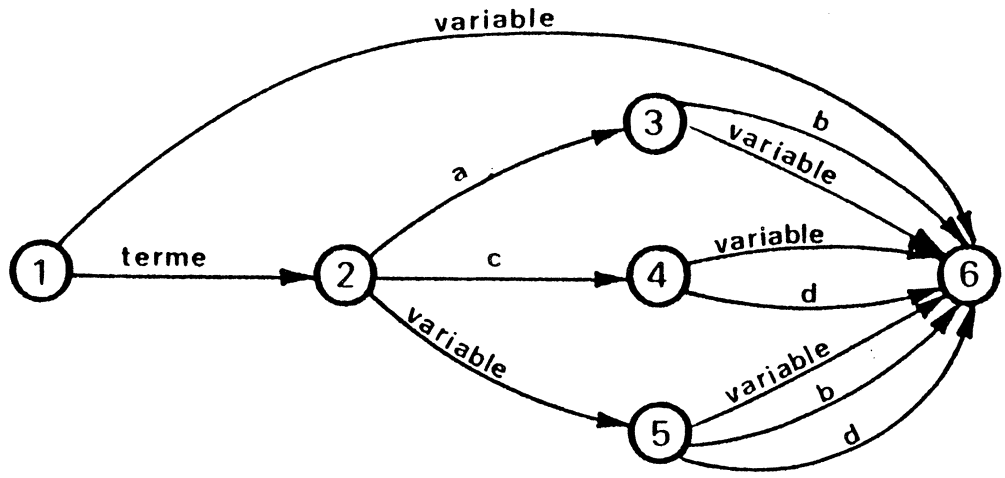


Le problème devient plus compliqué dès que plusieurs buts sont en présence: en effet, la présence d'une variable amène l'impossibilité de choix entre deux buts, et le report de ce choix à l'état suivant.

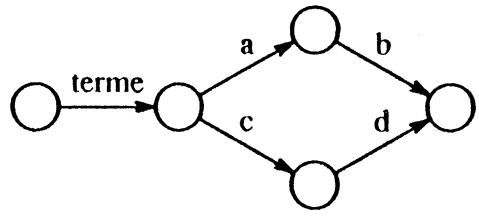
La présence de variables dans la base de données a donc pour effet de rendre les automates non déterministes. On sait qu'il est possible de transformer un automate fini non déterministe en automate fini déterministe, mais cela se fait au prix d'une forte augmentation de la complexité des automates: pour chaque noeud, il faut rajouter une transition correspondant à la lecture d'une variable; l'état atteint doit alors reposer le choix entre plusieurs buts qui n'a pas pu être effectué.

Exemple:

1: terme (a,b)
2: terme (c,d)



On comparera cet automate à celui correspondant au cas où aucune variable n'est attendue dans la base de données:



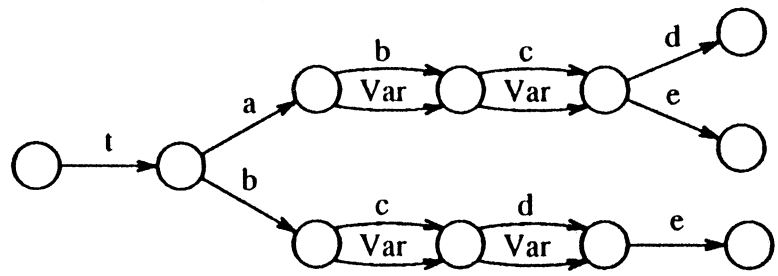
Cet exemple se place dans le cas général où les variables peuvent apparaître n'importe où dans les en-têtes de clauses. Il existe cependant des cas particuliers sensiblement plus simples, lorsque les variables interviennent aux mêmes positions pour l'ensemble des en-têtes de clauses, et que l'on connaît à l'avance les positions où se trouvent ces variables.

Exemple:

Soient les trois buts suivants:

t(a,b,c,d).
t(a,b,c,e).
t(b,c,d,e).

Si on sait que des variables apparaissent toujours en seconde et troisième position dans les en-têtes de clauses, on pourra générer l'automate suivant:



Le dernier cas favorable - et sans doute le plus courant - est enfin si on connaît un certain nombre de positions où il n'y a pas de variables. Si ces positions sont suffisantes pour distinguer les différents buts, on se ramène au cas précédent de la façon suivante:

- on ne fait aucun traitement au vol des positions où peuvent se trouver les variables,
- On vérifie après coup, dans le cas où des constantes sont lues pour ces positions, qu'elles correspondent bien à ce qui est attendu pour un des buts.

Avec l'exemple précédent, ceci correspond par exemple au cas où on sait que les premiers et derniers arguments des en-têtes de clauses ne sont jamais des variables. On peut alors générer le même automate que précédemment, mais il faudra un traitement complémentaire en cas de succès pour vérifier que ce qui est lu en 2ème et 3ème position correspond bien à ce qui est attendu pour le but correspondant.

VI.3.2. Choix d'une solution:

On se fixe comme objectif de pouvoir rechercher au vol un ensemble de buts relativement important (au moins quelques dizaines). Les automates deviennent donc très complexes, nécessitent une mémoire importante et leur compilation et leur chargement deviennent longs. Par ailleurs, on souhaite distribuer le traitement entre des opérateurs relativement simples, pour faciliter l'intégration en VLSI des éléments de la machine.

Deux solutions sont alors possibles:

- Chercher à se ramener par logiciel à un des cas favorables précédents et utiliser un automate d'états finis "classique", quitte à effectuer un traitement complémentaire par logiciel dans les cas "tordus".
- Chercher à définir un opérateur permettant d'effectuer la préunification dans tous les cas, c'est à dire un automate non déterministe.

La première solution pose plusieurs problèmes: d'une part, on n'est jamais sûr de pouvoir toujours se ramener à un cas favorable avec une sélectivité suffisante: en particulier lorsque on se trouve dans le dernier des cas précédemment étudiées (plusieurs buts, variables dans les buts et les en-têtes). D'autre part, la mise en oeuvre est complexe: il faut maintenir, pour chaque paquet de clauses, une information sur la possibilité d'avoir des variables dans chaque position, et il faut exploiter cette information pour la génération de l'automate.

Pour notre part, nous avons choisi d'explorer la seconde solution, et donc de concevoir un automate de recherche non déterministe. Ceci peut se faire normalement soit par retour arrière, soit par traitement parallèle des différentes alternatives. La nécessité

de suivre le débit du disque interdit tout retour arrière, et il faut donc être capable de parcourir en parallèle les branches de l'automate correspondant aux diverses solutions possibles à un instant donné.

Ceci implique de pouvoir gérer un ensemble de descripteurs correspondant à un ensemble d'états de l'automate, et de travailler en parallèle sur ces états. Or, il ne nous semble pas réaliste de vouloir mettre en parallèle un nombre d'opérateurs suffisant pour rechercher un grand nombre de buts. Notre approche consiste donc à décomposer le travail pour se ramener a des automates plus simples permettant d'effectuer la reconnaissance des éléments des termes, et la manipulation des ensembles de buts reconnus.

VI.4. LE FILTRAGE DANS OPALE:

Etant donné un terme T lu sur sur disque, et un ensemble de buts $\{B_i\}$, le problème que l'on se pose est de savoir si T préunifie avec un ou plusieurs buts, et lesquels.

Pour tout élément E_t de T, on peut avoir les situations suivantes:

- E_t est une variable: il peut donc être substitué avec n'importe quel sous terme de même position appartenant aux buts. E_t n'apporte donc aucune restriction à l'ensemble des buts susceptibles de préunifier. Cette situation implique la création de substitutions entre la variable E_t et les sous termes de même position dans $\{B_i\}$.
- E_t est une constante: pour chaque but B_i, on a alors à nouveau plusieurs situations:
  * Il existe un élément constant E_b de B_i qui a la même position que E_t. Dans ce cas, soit $E_b = E_t$, et B_i peut préunifier avec T, soit $E_b \neq E_t$, et B_i ne peut pas préunifier.
  * Il existe une variable V_b de B_i qui a la même position que E_t. Dans ce cas, la préunification entre T et B_i est possible et entraîne une substitution entre V_b et E_t.
  * Il n'existe aucun élément de B_i qui a la même position que E_t. Une telle situation correspond à deux cas:
    . soit E_t est un membre d'un sous terme de T substitué à une variable de B_i,
    . soit E_t appartient à un sous terme de T dont l'arité est différente de celle du sous terme de même position dans B_i. Un symbole fonctionnel étant identifié non seulement par son nom, mais aussi par son arité, cela signifie que l'on a rencontré dans T un élément constant différent de l'élément correspondant dans B_i, et donc que l'échec de la préunification a déjà été détecté.

Pour préunifier un terme avec un ensemble de buts, il suffit donc de maintenir une liste de buts susceptibles de préunifier. Pour chaque élément constant du terme, on cherche quels buts possèdent un élément constant de valeur différente, et on les élimine de la liste des buts susceptibles de préunifier.

L'exécution de la préunification va donc comporter essentiellement trois opérations:

- Analyser la structure du terme lu, détecter les variables, et coder les positions,

- Comparer la valeur des éléments constants du terme lu avec celle des éléments constants des buts pour la même position,
- Tenir à jour une liste des buts qui sont susceptibles de préunifier avec le terme en cours d'analyse.

Ces opérations peuvent être exécutées en pipe-line par trois opérateurs transmettant leur résultat vers un tampon partagé avec un microprocesseur.

VI.4.1. Implémentation:

VI.4.1.1. analyse de la structure:

Le but du premier opérateur est d'analyser la structure des termes, de coder leur position, et de déclencher certaines actions. On l'appellera donc analyseur de structure.

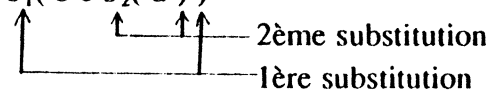
Les actions peuvent être les suivantes:

- Les variables lues sur disque ne nécessitent pas d'autre traitement: elles sont donc transmises vers un tampon de travail avec le code de leur position. La position est ici la seule donnée nécessaire pour connaître la valeur substituée à la variable du disque, puisque le but est déjà connu.
- Les données dont la position correspond à une variable d'un but sont également transmises avec un code de position. Si cette position correspond uniquement à une ou plusieurs variables des buts, il y a transmission d'un terme, sans codage de position.
- les données correspondant à des variables anonymes, ou suivant la détection d'un échec de la préunification sont ignorées: il y a donc recherche de la fin d'un terme ou d'une clause sans transmission d'un résultat.
- La queue d'une clause, lorsqu'elle existe, est transmise sans analyse.

On appelle substitutions les couples donnée lue/position. Du fait de la structure des buts, il peut y avoir imbrication de substitutions: Cette imbrication doit être gérée à l'aide d'une pile.

Exemple:

buts:	t(a,X)
	t(a,b(c,Y))
terme lu:	t(a,b(c,d))
substitutions générées:	S_1(b c S_2(d))



L'analyseur de structure reconnaît les transitions du terme:

- Passage à un noeud fils (F),
- Passage à un noeud frère (Fr),
- Retour au noeud père (P).

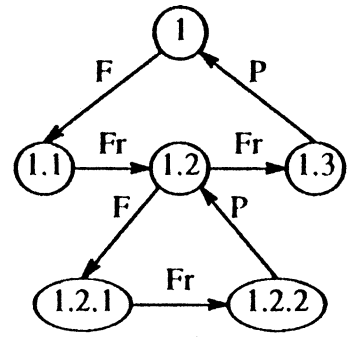
Ceci nécessite un comptage des nombres de fils, avec empilage lorsqu'un symbole fonctionnel est reconnu. L'octet de type des données est donc décodé: dans le cas d'un symbole fonctionnel, il contient le nombre de fils. L'analyseur de structure est également capable de chercher la fin d'un terme ou d'une clause.

Le codage de la position peut se faire par l'intermédiaire d'un automate d'états finis, qui reçoit en entrée les transitions "F", "Fr", "P" détectées par l'analyseur de structure.

Exemple:

Considérons les buts suivants: t (a , X , b)
t (c , d (e , Y) , X)

L'automate est le suivant:



Les suites de chiffres désignent les positions. Pour la position 1.2, on a un début de substitution correspondant à la variable *X* du premier but, mais la structure est toujours analysée à cause du second but. Pour la position 1.2.2, on génère une seconde substitution pour la variable *Y* du second but: cette substitution est imbriquée dans la première, il y a donc empilement. La nouvelle substitution doit de toutes façons être indiquée, pour faciliter l'exploitation des résultats. Si on trouvait après 1.2.2 un troisième fils, c'est à dire une position ne correspondant pas au second but, il y aurait tout de même poursuite de l'analyse, puisqu'une substitution est déjà en cours sur 1.2. Sur la position 1.3, il y a une nouvelle substitution, et donc la possibilité d'accepter n'importe quelle donnée. Enfin, une transition père à partir de cette position entraîne le retour à la position 1, et donc la reconnaissance de la fin du terme analysé.

On peut considérer que l'automate analyse une structure de terme correspondant à l'union des structures de tous les buts: on l'appelle structure englobante. Cette structure englobante est en fait intéressante, car sa génération fait partie de la compilation des buts. Dans [GBS2 84], on présente l'algorithme de génération de la structure englobante, et

une justification de l'algorithme de préunification basée sur cette notion.

Pour chaque noeud, l'automate génère un code correspondant à la position et utilisable directement pour la seconde étape (recherche des valeurs): il est donc transmis au 2ème opérateur avec la donnée lue sur le disque. Une action peut également être associée à chaque noeud (génération d'une substitution, etc...).

Lorsqu'une position n'est pas reconnue par l'automate, deux cas sont possibles:

- Soit on a affaire à un terme qui n'est unifiable avec aucun but,
- Soit la donnée lue correspond à une variable d'un but (substitution).

Ces deux cas sont distingués par l'existence d'une substitution en cours.

VI.4.1.2. Recherche:

Pour chaque élément constant du terme lu, un but ne préunifie que si l'élément de même position correspond à une variable, ou est une constante de même valeur. Donc, en fonction de la valeur lue et de la position, on aura un ensemble de buts possibles.

Le rôle de l'opérateur de recherche est donc de chercher une valeur constante parmi une liste de valeur possibles, et d'y associer une liste de buts possibles. A la fin de l'opération, l'ensemble des buts qui préunifient est l'intersection des ensembles de buts correspondant à chaque valeur lue.

L'opérateur de recherche effectue donc une simple recherche lexicographique parmi les ensembles de valeurs possibles pour chaque position. L'analyseur de structure transmet l'adresse de la liste de valeurs correspondant à cette position.

Un échec de la recherche ne correspond pas toujours à un échec de l'unification, puisque la donnée lue peut correspondre à une variable d'un ou plusieurs buts. Si tous les buts comportent des variables pour cette position, la recherche est évidemment inutile.

Diverses solutions peuvent être utilisées pour l'implémentation de la recherche:

- Recherche dichotomique câblée sur liste linéaire triée: cette solution implique un algorithme de complexité moyenne (comparaisons, divisions, indexations). L'implémentation matérielle peut toutefois être relativement complexe. Le nombre de comparaisons est en principe de $\log_2(n)$, si n est la longueur de la liste.
- Recherche par arbre binaire équilibré: ce n'est autre qu'une implémentation particulière de la dichotomie, les données étant réparties sur un arbre par ordre de valeur. L'accès est toujours en $\log_2(n)$, et l'implémentation matérielle peut être relativement simple (comparaisons, et manipulation de pointeurs). Cependant, la génération de l'arbre n'est pas très simple. C'est également une solution assez coûteuse en mémoire.
- Recherche séquentielle: cela reste une solution intéressante, en raison de l'extrême simplicité du matériel (comparaisons et compteurs). L'utilisation de mémoires statiques rapides (environ 50 ns) permet d'exécuter plusieurs dizaines de comparaisons pendant la lecture sur disque d'une donnée (4 octets = 2 à 4 µs).

- Mémoire associative: cette solution est également possible,dans la mesure où le nombre de valeurs a rechercher n'est pas excessif.

VI.4.1.3. Sélection des buts:

La sélection des buts consiste à maintenir une liste des buts et à en retirer successivement ceux qui sont reconnus ne pas pouvoir unifier avec le terme lu; autrement dit les buts qui, à une position donnée, comportent une valeur constante différente de la valeur lue.

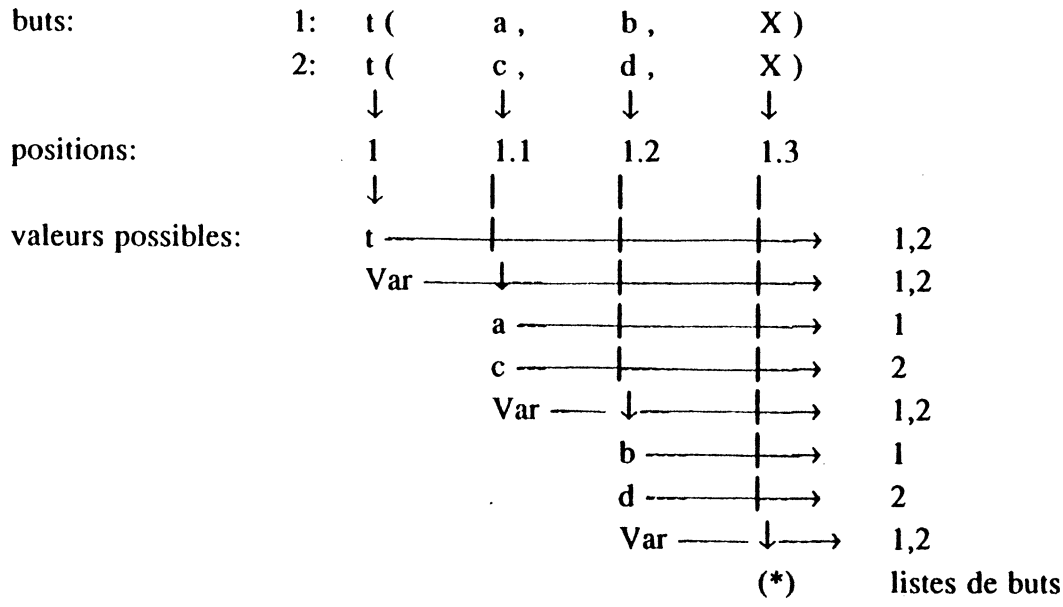
On peut proposer deux implémentations:

- La première consiste à coder les listes de buts sous forme d'un automate dont les états correspondent à des listes de buts et les transitions correspondent aux valeurs trouvées par l'opérateur de recherche: il s'agit en fait d'une forme compactée de l'automate déterministe qui effectuerait la préunification. Certaines transitions conduisent à un état d'échec (liste de buts vide). Les transitions atteintes à la fin de la préunification doivent être décodées pour fournir en clair la liste des buts qui préunifient. Le problème posé par cette solution est que la génération de l'automate n'est pas très simple.
- La seconde consiste à représenter la liste des buts sous forme d'une chaîne de bits (chaque position correspondant à un but). Pour chaque valeur lue, on doit réaliser l'intersection entre la chaîne de bits représentant l'ensemble des buts qui peuvent préunifier avec le terme lu et la chaîne de bits représentant l'ensemble des buts qui peuvent admettre cette valeur à la position correspondante. Lorsque la chaîne de bits résultante devient nulle, il y a échec de l'unification. Si en fin de lecture du terme, la chaîne est non nulle, les bits restés à un correspondent aux buts avec lesquelles le terme lu unifie.

Cette seconde solution nous semble plus simple à implémenter, tant au niveau du filtrage lui-même qu'à celui de la compilation des buts. Elle nous semble donc préférable, bien qu'elle semble à priori nécessiter plus de mémoire.

L'opérateur de recherche va donc transmettre l'adresse de la chaîne de bits correspondant à la valeur lue. Si on associe une chaîne de bits à chaque valeur possible, l'adresse à transmettre est égale à l'index de la valeur dans la table, ce qui simplifie l'opérateur de recherche.

Exemple:



(*) = valeur quelconque, substitution, pas de sélection des buts.

VI.4.2. Discussion:

Cette décomposition permet une simplification importante des automates. Il peut encore arriver que les termes recherchés soient trop complexes ou trop nombreux pour qu'on puisse effectuer l'unification au niveau du filtre et du microprocesseur qui le contrôle (problème de taille des automates). On peut alors décomposer l'opération en remplaçant certaines parties du ou des termes cherchés par des variables. On effectue le filtrage sur les termes ainsi simplifiés, puis on complète l'opération en cherchant à unifier les sous-termes remplacés avec les termes correspondants lus sur disque. Ces opérations peuvent être exécutées en pipe line par plusieurs processeurs.

Exemples:

On cherche: *terme(a,b(c,d),e,f(g,h)).*

Cela peut revenir à chercher: *terme(a,X,e,Y)* puis à vérifier que le terme substitué à *X* unifie avec *b(c,d)* et que le terme substitué à *Y* unifie avec *f(g,h).*

On cherche: $t(a, b_1, c, X)$
$t(a, b_2, c, X)$
...
$t(a, b_n, c, X)$

On peut chercher en fait: *t(a, Y, c, X)*, puis vérifier que la valeur lue pour *Y* unifie avec l'un des b_i.

Un problème non encore traité concerne les inéquations: dans le chapitre IV, nous avions fait allusion au fait que l'unification pouvait être étendue à des contraintes autres

que l'égalité, c'est à dire que l'unification peut tenir compte du fait qu'une variable ne peut être liée qu'avec une donnée inférieure ou supérieure à une certaine valeur...

En dehors de cas d'école, il ne semble pas que de telles contraintes puissent constituer un critère essentiel de sélection sur un gros ensemble de données: la sélectivité d'une inégalité est le plus souvent très faible, et un autre critère de sélection doit exister, faute de quoi la question retournera un très grand nombre de solutions. Il n'est donc pas évident à priori qu'il soit intéressant de consacrer du matériel à ce genre de problèmes, qui peuvent être traités par logiciel. Il est cependant possible d'améliorer l'opérateur de recherche, en indiquant pour chaque valeur le type de comparaison requise (égalité, inégalité, etc...).

Un point délicat concerne le filtrage sur des listes: on souhaite filtrer les en-têtes de clauses qui possèdent l'élément *E* dans une liste *L*: ceci demande normalement de lire la liste, puis d'effectuer la recherche de l'élément. On peut remarquer que, dans le cas où la liste est courte, il est possible de décomposer la recherche en éclatant la cible. En effet, chercher un élément *E* dans une liste *L*, c'est chercher une liste qui unifie avec l'une des listes suivantes:

[el|X]
[X1,el|X]
[X1,X2,el|X]
etc...

Ceci n'est évidemment possible que dans le cas où la longueur de la liste ne dépasse pas quelques unités, mais ce principe a pu être adapté par exemple à la recherche sur mots-clefs dans une base de données bibliographique. Des travaux sont entrepris pour tenter de trouver une simplification dés automates dans ce genre de situation, ce qui permettrait d'intégrer au niveau du filtre des sélections sur l'appartenance d'un élément à une liste, et donc d'augmenter considérablement l'intérêt de la manipulation de listes par Prolog.

VI.5. ARCHITECTURE DU FILTRE:

Selon l'architecture décrite au chapitre III, le filtre est un composant d'un ensemble fonctionnel lié au disque, comportant le coupleur disque, et contrôlé par un microprocesseur (Le processeur disque: Figure VI.2). On trouvera dans [GBS4 84] de plus amples détails sur l'implémentation matérielle et logicielle du filtrage.

Le microprocesseur contrôle la communication avec les autres composants de la machine, il commande le filtre et complète son travail. Ses fonctions sont donc les suivantes (Fig VI.3.):

- Il reçoit les buts transmis par d'autres éléments de la machine. Ces buts sont transmis sous une forme telle que le processeur disque puisse les connaître entièrement sans avoir recours à des substitutions stockées dans un autre processeur.

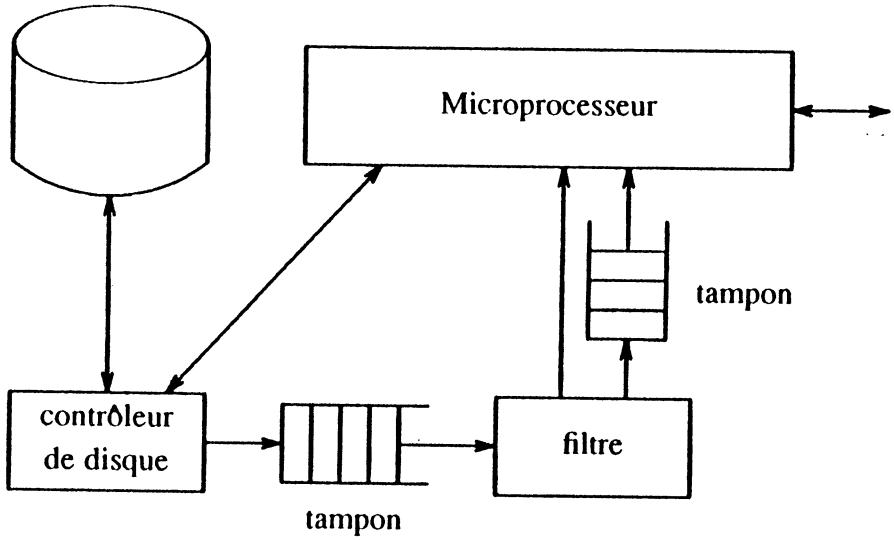


Figure VI.2

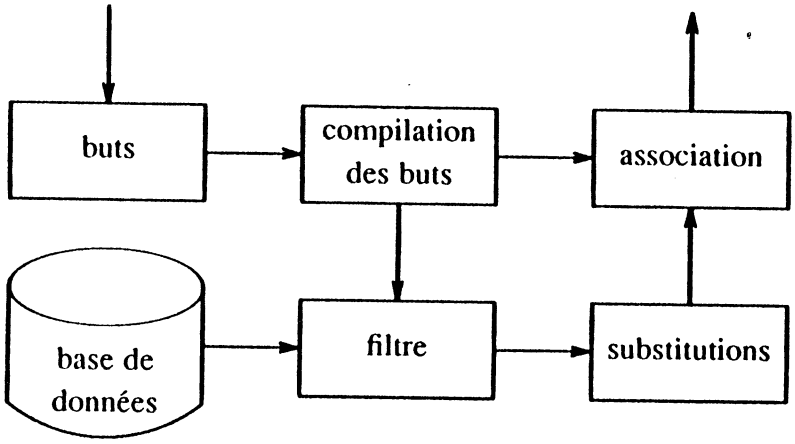


Figure VI.3.

- Il compile les buts pour générer les tables du filtre,
- Il traite les substitutions issues du filtre, élimine celles qui sont inutiles et effectue l'association. Ce traitement a pour effet d'associer aux positions des substitutions les sous termes correspondants des buts. Les substitutions sont donc composées, et leur cohérence est vérifiée, ce qui achève l'unification.
- Il retourne les résultats aux processeurs demandeurs.
- Il gère l'accès au disque, optimise l'ordonancement des requêtes, et contrôle les écritures.

Le filtre lui même est basé sur les trois opérateurs précédemment décrits, et opérant

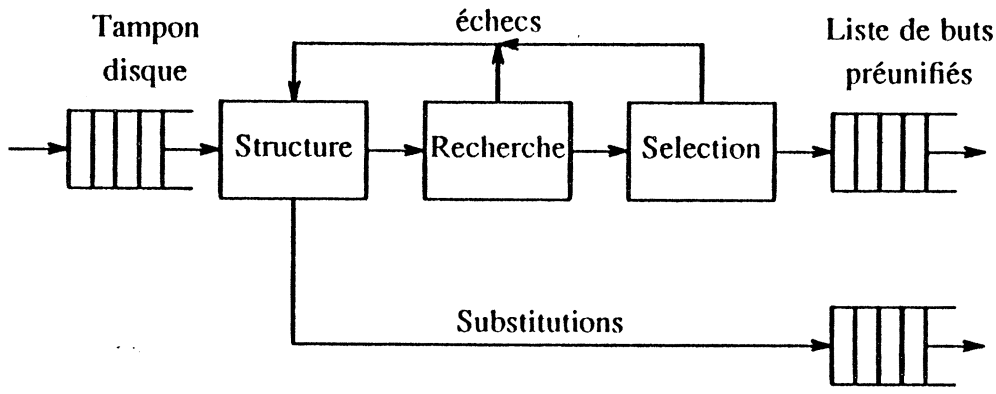


Figure VI.4.

en pipe-line (Fig VI.4.).

VI.6. EVALUATIONS:

VI.6.1. Encombrement mémoire:

Dans ce paragraphe, nous nous intéressons au volume de mémoire nécessaire pour les différents composants du filtre.

Ce volume va dépendre en particulier du nombre et de la complexité des buts. Ces données sont en fait à peu près impossibles à déterminer à priori: il faudrait pouvoir mesurer des bases de données en utilisation réelle, ce qui n'est pas envisageable dans le cadre de ce projet (même en se contentant de bases relationnelles). On va donc effectuer une évaluation formelle, puis tacher de voir quelles sont les résultats obtenus pour différentes valeurs des paramètres.

L'évaluation sera faite pour chacun des composants du préunificateur. On considérera que les opérateurs utilisent une mémoire commune de taille limitée à 64 K, ce qui donne un adressage sur 16 bits. Des solutions faisant appel à des mémoires particulières pour chaque sous opérateur sont cependant possibles, et conduiraient à un adressage plus compact.

VI.6.1.1. Analyse de structure:

La taille de l'automate va dépendre essentiellement de la taille de la structure englobante (union des structures des buts), que l'on notera S (nombre d'éléments). En tout état de cause, 256 semble une limite très raisonnable pour cette taille: des cas plus complexes sont rares, et peuvent être simplifiés en effectuant un filtrage partiel.

Etant donné le faible nombre de mots du vocabulaire d'entrée, l'utilisation d'un automate tabulé est tout indiquée. Chaque état peut être codé sur 9 octets:

- 3 pointeurs pour les transitions frère, fils, père,
- Un code d'action (1 octet),
- L'adresse d'un liste de valeurs (2 octets).

La taille mémoire requise est alors de 9S octets.

Il faut ajouter à cela une pile pour l'analyse des termes et une autre pour la gestion des substitutions imbriquées. La taille de ces piles est égale au niveau maximum d'imbrication autorisé dans les termes (noté I), soit une taille totale de 2I.

VI.6.1.2. Opérateur de recherche:

L'espace nécessaire à l'opérateur de recherche va être égal à la taille des listes de valeurs possibles pour chaque position. Celle ci va varier beaucoup en fonction des buts.

En effet, il faut se rappeler qu'un ensemble de buts sur un même paquet de clauses sera généralement généré par applications de substitutions sur certaines des variables d'un littéral. Le constantes du littéral seront donc communes à tous le buts, et seules les substitutions introduiront des différences entre les buts.

Si on considère un ensemble de N buts, on peut avoir par exemple les deux cas suivants:

- Les buts sont obtenus par applications de substitutions sur une seule variable: si les termes substitués à la variable sont atomiques, le nombre de valeurs correspondant sera égal à N. Il peut toutefois être plus grand si les termes sont non atomiques.
- Les buts sont obtenus par applications de substitutions sur plusieurs variables. Si ces substitutions sont produites de façon indépendante par plusieurs littéraux, on aura un produit cartésien des solutions de ces littéraux:

Exemple:

Soit la clause: c(X) :- l1(a,Y), l2(b,Z), l3(X,Y,Z).

Les substitutions sur Y et Z sont produites de manière indépendante par la résolution de l1 et l2; l3 reçoit donc le produit cartésien de ces ensembles de solutions. Dans un tel cas, le nombre de valeurs sera compris entre $\sqrt{N}$ et N.

On voit donc qu'il n'y a pas de relation claire entre le nombre de littéraux et le nombre de valeurs possibles. Le maximum fixé pour ce nombre devrait toutefois dépasser nettement le nombre maximum de buts que l'on souhaite pouvoir traiter en parallèle.

Chaque valeur va demander 7 octets: 5 octets pour le codage de la valeur (1 octet de tag et 4 octets de valeur), et 2 octets de pointeur sur la liste de buts correspondante. Si V est le nombre maximum de valeurs que l'on décide de traiter, la place requise sera donc de 7V.

VI.6.1.3. Sélection des buts:

La taille (en bits) de la matrice de bits est le produit du nombre de valeurs acceptées par le nombre de buts possible, soit BxV, où B est le nombre de buts possibles.

VI.6.1.4. Total:

La taille totale requise sera donc: $9S+2I+7V+\frac{BV}{8}$.

Si on se donne par exemple les valeurs suivantes: S = 256, I = 32, V = 512, B = 128, la taille totale sera de: 14144 octets. Réciproquement, avec une mémoire de 64 K octets, on a la possibilité de traiter (par exemple) 430 buts comportant 1024 valeurs différentes, avec une profondeur d'imbrication pouvant atteindre 256, et une structure englobante de 310 éléments.

La matrice de bits représentant les ensembles de buts est en fait le point le plus critique: dans notre premier exemple, elle représente 8K sur 14, et dans le second, 54 K sur 64. Des codages plus compacts pourraient être recherchés.

Notons par ailleurs que le filtrage se fera le plus souvent sur une piste de disque (si l'indexation est efficace), soit 16 à 32 K. Si la taille moyenne des clauses est de 30 à 60 octets, on en aura de 250 à 1000 sur une piste. Il semble alors assez cohérent d'admettre de ne filtrer au plus que quelques centaines de buts sur un tel ensemble de données.

VI.6.2. Performances:

On doit considérer la possibilité de traiter un élément d'un terme dans le temps de sa lecture: il s'agit en fait d'un temps de traitement moyen, puisque bien des éléments de données auront un traitement très réduit. Si on considère qu'un élément de données est codé sur 5 octets, son temps de lecture sur le disque est d'environ 4 µs avec des disques d'un débit de 10 Mégahertz. En cela, on considère que l'évolution des disques ira plus dans le sens de la compacité des unités que dans celui de l'augmentation des cadences de transfert, ceci pour rester compatibles avec le marché des micro-ordinateurs.

La seule opération qui soit exécutée sur toutes les données est la lecture et l'analyse de structure: les autres opérations (codage de position, recherche et sélection des buts) ne sont exécutées que pour les éléments constants pertinents dans la sélection. L'analyseur de structure sera donc un automate câblé: vu la relative simplicité de des opérations qu'il effectue, le débit ne doit pas poser de problèmes.

Le codage des positions sera également une opération simple et rapide (adressage et lecture de l'état de l'automate).

Le véritable goulot d'étranglement est constitué par l'opérateur de recherche. Avec une recherche séquentielle, et un temps de cycle de 50 nanosecondes pour la mémoire, on peut comparer jusqu'à 80 valeurs en 4 µs (en supposant que l'on dispose d'un comparateur permettant d'opérer en parallèle sur 5 octets). En moyenne, ceci doit permettre le filtrage de plus de 80 buts (le temps perdu dans les recherches les plus longues se rattrapant sur d'autres cas). Une implémentation plus performante de la recherche (dichotomie, mémoire associative) peut permettre de repousser cette limite. Quelques

évaluations sur une maquette seraient en fait nécessaires pour préciser plus ce point.

La sélection des buts pose à son tour peu de problèmes: il s'agit de l'intersection de deux chaînes de bits dont la longueur est égale au nombre de buts: pour 400 buts (ce qui est proche du cas extrême mentionné précédemment), l'intersection sur 8 bits demande 50 opérations, ce qui fait 2,5 µs. Ici encore, une comparaison sur 16 ou 32 bits permettrait de repousser les limites si nécessaire.

VI.6.3. Implémentation:

L'algorithme de préunification a fait l'objet d'une implémentation en C sous UNIX (sur SM 90, avec un processeur MC 68000)[BER 86]. Cette implémentation a consisté a insérer des possibilités de filtrage de bases de clauses atomiques (sans corps, mais dont l'en-tête peut contenir des variables) dans un interpréteur Prolog. On retrouve donc l'intégralité du problème de l'unification (en-têtes de clauses pouvant inclure des variables), mais sans le problème de la vérification des littéraux inclus dans la queue de la clause. Il reste toutefois possible d'exécuter en pipe-line la recherche des solutions de plusieurs littéraux consécutifs.

Deux prédicats assurent l'accès aux clauses stockées en mémoire secondaire:

- interobase([V1,...Vn], relation(V1,...Vm)): le premier argument est une liste de variables qui apparaissent toutes dans le second argument. Ce dernier est un littéral d'appel correspondant à un paquet de clauses stocké sur disque. Les solutions du littéral sont recherchées (par unification) en un accès disque, et les variables reçoivent les substitutions correspondant à la première solution. Au backtracking, le retour sur le prédicat interobase permet d'obtenir les autres solutions, ou échec s'il n'y a plus de solutions. Ce prédicat permet donc de passer d'une production d'un ensemble de solutions à une consommation non déterministe de ces solutions (par backtracking).
- pipe([V1,...Vn],c(V1,....Vm)): le premier argument a le même rôle que dans interobase. Le second spécifie une clause de la forme:

  c(V1,V2,...) :- r1(V1,...), ... ri(Vi,...).

  Les prédicats de la queue de la clause correspondent à des paquets de clauses stockés en mémoire secondaire. L'appel de ce prédicat déclenche une évaluation en pipe-line des littéraux de la clause, chaque littéral traitant de manière ensembliste les solutions du littéral précédent, d'une manière analogue à ce qui a été décrit au chapitre précédent. Les clauses résidant en mémoire secondaire étant atomiques, il n'y a cependant pas de traitement équivalent aux OU-processus. Par contre, on a bien à unifier des ensembles de buts avec des en-têtes de clauses pouvant contenir des variables.

Le but de cette implémentation était essentiellement de valider et évaluer l'algorithme de filtrage et les différentes opérations annexes (association, compilation des buts). Des détails peuvent être trouvés dans [IAN 85, BER 86, GBS2 84].

Les mesures de temps de traitement ont été faites avec l'utilitaire "prof" sous Unix, qui mesure par comptage le nombre d'appels de chaque procédure, et par échantillonage statistique (un échantillonage du compteur ordinal tous les soixantièmes de seconde) le temps passé dans chaque procédure. Moyennant certaines précautions (Cf annexe 3), on arrive à des résultats fiables et reproductibles à une faible marge d'erreur près.

On trouve dans la thèse de Ianeselli [IAN 85] une comparaison entre le filtrage implémenté en logiciel et l'algorithme classique d'unification. On considère le filtrage de 100 en-têtes de clauses, avec un facteur de sélectivité de 1% (1% des données filtrées sont sélectionnées). Les temps d'exécution comprennent, pour le filtrage, la compilation des buts et l'association. Pour un seul but, la durée du filtrage est de 280 µs pour 330 µs pour l'unification classique. Dès que le nombre de buts augmente, la durée moyenne du filtrage par but décroît de façon très importante: pour 10 buts, elle est de 41,3 µs, pour 20 buts de 29,3 µs, pour 50 buts de 21,4 µs, et pour 100 buts de 14.6 µs.

En fait, l'avantage du filtrage réside dans la possibilité de traiter simultanément plusieurs buts, possibilité qui découle de la stratégie de recherche. L'implémentation matérielle permet de réaliser cela "au vol", ce que ne permettrait pas une implémentation purement logicielle.

On peut faire les remarques suivantes, au vu des évaluations effectuées:

- La compilation des buts prend à peu près le temps d'une unification (T_u) par but, soit une moyenne de 0,5 millisecondes,
- Dans le pire cas, l'association prend également à peu près le temps d'une unification par résultat traité (en considérant que le plus souvent un résultat correspond à un seul but),

Le temps d'exécution se répartit entre les diverses opérations de la façon suivante:

recherche:	20 à 30%
sélection des buts:	10 à 15%
gestion des substitutions:	10 à 15%
analyse des termes:	5 à 10%
codage des positions:	5 à 10%
recherche de fin de clause:	10 à 15%
contrôle:	10 à 15%

Occupation du microprocesseur:

Le microprocesseur aura deux tâches essentielles: la compilation des buts, et l'association. Ces tâches sont à exécuter en parallèle avec l'accès au disque, qui se décompose en un délai de positionnement (de l'ordre de 30 ms), et un temps de transfert (de 16 ms pour une piste). Dans l'idéal, la compilation des cibles doit pouvoir prendre place durant le délai de positionnement, alors que l'association se fait durant le transfert, en parallèle avec le filtrage.

D'après les évaluations effectuées, ces objectifs peuvent être atteints pour un maximum de l'ordre de 60 buts en ce qui concerne la compilation avec un MC 68000, l'augmentation de la rapidité des processeurs permettant de repousser cette limite et de s'adapter à des disques plus rapides.

Pour l'association, le temps estimé de 0,5 ms par opération permet de traiter 32 clauses en 16 ms: pour une sélectivité de 10, cela nous fait donc 320 clauses filtrées, ce qui est également correct. On peut donc considérer qu'un microprocesseur 16 bits tel que le 68000 permet d'effectuer le traitement requis sans risque exagéré d'attente du disque.

VI.7. CONCLUSIONS:

La réalisation matérielle d'un filtre VLSI qui exécute la préunification pose assez peu de problèmes de performances. La taille mémoire nécessaire va de quelques K à 64 K, ce qui correspond à des volumes assez réduits selon les critères actuels: l'utilisation de mémoires statiques est possible, et il est également possible (selon les choix effectués en matière de capacité) d'intégrer directement ces mémoires dans les circuits de filtrage. Quand aux automates eux mêmes, on peut estimer que leur complexité n'excédera pas celle d'un microprocesseur tel que le MC 68000.

Un avantage de notre solution par rapport à celles qui ont été déjà proposées dans le domaine des bases de données relationnelles est évidemment de pouvoir exécuter au vol non seulement un filtrage, mais une étape significative de l'unification, et ce avec une complexité qui ne semble pas accrue de manière excessive.

L'intégration en VLSI du filtre a déjà fait l'objet d'études, particulièrement dans la thèse de Ianeselli [IAN 85], et le rapport de projet de bruno Lardoux [LAR87]: ce dernier s'est attaché à démontrer la faisabilité des différents éléments de l'opérateur de filtrage. L'ensemble serait intégrable sous forme d'un circuit de quelques dizaines de milliers de transistors (le MC68000 en comporte environ 70000). La plus grande part du circuit serait composée de mémoire, ou d'opérateurs très réguliers, dont la conception représente peu d'efforts (à condition de disposer des outils de CAO nécessaires!).

Des évaluations plus précises devraient permettre de trouver des solutions plus générales, que celle d'automates entièrement câblés. On peut également espérer accroître notre marge de sécurité en prenant en charge au niveau du filtre certaines fonctions liées à la compilation des buts, où à l'association, ce qui diminuerait le risque de ralentir le disque en cas de traitement excessivement long. Nous avons déjà noté dans un précédent chapitre que l'absence de moyens expérimentaux pour l'évaluation était un handicap, et c'est pourquoi une simulation logicielle est une étape importante pour affiner les paramètres de la réalisation matérielle. Les performances obtenues au niveau du filtrage logiciel sont déjà un point positif dans cette perspective.

Chapitre VII

MACHINES PROLOG

VII.1. INTRODUCTION:

Après avoir étudié l'interprétation de Prolog dans le cadre des bases de données, il était intéressant d'aller plus loin, en cherchant quelles voies peuvent être trouvées pour la conception d'une machine Prolog (non seulement bases de données). Une autre motivation était également de rechercher si les résultats obtenus avec les bases de données (filtrage et stratégie de recherche) peuvent être généralisés au cas de l'interprétation de programmes Prolog.

Prolog étant un langage très différent des langages utilisés jusqu'ici, il est assez naturel de mettre son relatif manque de performances sur le compte de l'inadaptation des processeurs, ce qui amène à envisager la conception de machines Prolog. Les approches pour la conception d'une telle machine sont multiples, correspondant aux grands problèmes de l'interprétation de Prolog.

Nous en distinguerons trois:

- L'approche classique: consiste à partir d'évaluations de performances et à définir les opérations de base que devrait avoir une machine pour interpréter efficacement Prolog, tout en restant d'architecture classique. A partir de là, on peut soit fournir un langage assez classique, c'est à dire d'assez bas niveau, et permettant d'implémenter Prolog de façon efficace par logiciel (compilation ou interprétation), soit implémenter directement un interpréteur Prolog par microprogrammation. Le premier cas est, par exemple, celui de la machine WAM [WAR 83, TIC 84], bien que le langage WAM soit d'un niveau assez élevé par rapport aux langages machines usuels. Le second cas et celui de la machine PSI de l'ICOT [YOK 83].
- L'approche "structure de données": dans cette approche, on considère que le problème essentiel dans l'interprétation de Prolog est la manipulation d'objets structurés en arborescences, et on conçoit donc une machine permettant de traiter efficacement ce type de structure de données. Cette approche revient généralement à concevoir des machines aptes à la manipulation de listes chaînées, et rejoint bien souvent les machines Lisp [PER 82, BEK 83, SAN 85]. Cependant, le parallélisme entre Lisp et Prolog nous semble une approche un peu réductrice, en ceci que l'aspect manipulation de listes n'est, à notre avis, qu'une partie du problème de l'interprétation de Prolog.
- L'approche "unification": consiste à concevoir la machine autour d'un dispositif réalisant l'unification, à rechercher une méthode d'interprétation bien adaptée, et à définir une architecture de machine à partir de là. Nous rattacherons à cette approche l'utilisation d'un filtre, que nous exposerons dans la suite.

Dans ce chapitre, nous ne chercherons pas à définir un processeur Prolog: beaucoup de travaux restent en effet à accomplir dans ce domaine. Notre but est plus d'apporter un commentaire sur certaines approches, de donner les éléments d'appréciation que nous

pouvons avoir, et d'indiquer quelles approches nous semblent plus prometteuses. Nous nous intéresserons plus particulièrement à la première et à la dernière approche: la seconde a déjà fait l'objet de travaux par ailleurs, et il ne nous semble pas utile de nous y engager également. Le lecteur intéressé pourra se reporter à la bibliographie. Notre analyse se fera par ailleurs sans objectifs de performances définis: de tels objectifs ne sont significatifs que par rapport à un type de réalisation de la machine.

Notons enfin que ces approches ne sont pàs antagonistes: mettant l'accent sur des aspects différents, elles sont complémentaires.

VII.2. L'APPROCHE CLASSIQUE:

Cette approche va consister à partir d'une ou plusieurs implémentations existantes de Prolog, et à voir, à l'aide de mesures, quelles opérations et quels modes d'adressage doivent être implémentés, et quels types de données doivent être manipulés pour conduire à une amélioration significative des performances de Prolog.

Un choix à effectuer est celui du niveau du langage machine offert, qui peut être soit un langages de type assembleur, soit un langage très proche de Prolog. Dans le cadre du projet japonais de la 5ème génération, la machine PSI [YOK 83] a pour langage un dialecte de Prolog spécialement conçu, en particulier pour permettre l'implémentation de systèmes d'exploitation. Il s'agit en fait de la microprogrammation de l'interpréteur Prolog.

Cette approche nous semble manquer de généralité en ce sens qu'elle fixe un modèle de langage, en restreignant les possibilités d'évolution. Or, il semble que Prolog soit susceptible encore d'évolutions nombreuses et importantes, et que d'autres types de langages de programmation en logique soient susceptibles de voir encore le jour. Par ailleurs, il peut exister, au sein des applications Prolog, des sous problèmes pour lesquels l'ouverture vers d'autres types de langage apparaît nécessaire, sans qu'il soit pour autant justifié d'y consacrer du matériel .

Par ailleurs, il n'est pas prouvé que la microprogrammation d'un interpréteur Prolog sur une architecture adaptée conduise à des résultats plus performants que la compilation de programmes Prolog sur la même architecture. Il semble important enfin qu'une machine puisse bénéficier d'un environnement important, et de possibilités de communication avec d'autres systèmes, possibilités que seul, jusqu'ici, un système tel qu'UNIX semble fournir. Nous considérerons donc, pour notre part, une machine de plus bas niveau.

Un modèle de machine Prolog est proposé par Warren dans [WAR2 77]. Il s'agit toutefois d'une machine abstraite dont le but essentiel est l'implémentation de son compilateur Prolog sur DEC 10. De fait, la définition de sa machine Prolog reste très influencée par le DEC 10. La machine WAM enfin [WAR 83] est aussi à l'origine essentiellement un langage intermédiaire pour la compilation de Prolog, mais plusieurs études d'implémentation matérielles ont été faites.

Nous disposons de plusieurs implémentations de Prolog (source et objet). Deux sont des implémentations "maisons": Prolog I+ et Yaap (cf Annexe 3), et deux sont des

versions diffusées: C-Prolog (université d'Edimbourg), et D-Prolog (société Delphia). L'étude et la réalisation d'interpréteurs Prolog semblent constituer un préliminaire important pour la conception d'une machine Prolog, car elles permettent de maîtriser les mécanismes de base de l'interprétation de Prolog, et autorisent des mesures et évaluations sur le comportement des interpréteurs. Il est cependant également intéressant de considérer le fonctionnement d'autres interpréteurs, tels que C-Prolog, à condition de disposer du source.

L'étude de Prolog I+ et Yaap constitueront donc la base de ce qui va suivre: ces interpréteurs ont fait l'objet de mesures, et leur fonctionnement nous est familier. Bien que nous nous basions sur des interpréteurs, les résultats obtenus devraient rester valides pour une bonne part dans le cas d'un compilateur, les opérations de base mises en oeuvre dans les deux cas restant les mêmes. Si pour cette première étude générale, l'étude d'interpréteurs Prolog semble suffisante, il semble cependant clair que, pour un approfondissement, l'étude d'un compilateur devienne nécessaire.

L'interpréteur Prolog I+ représente environ 1900 lignes de programme en C. Sur ce total, les prédicats évaluables représentent 1100 lignes, et le noyau de l'interpréteur représente 500 lignes, le reste étant les déclarations et l'initialisation de l'interpréteur. Or, le noyau de l'interpréteur, qui est composé de quelques procédures représente plus de 90 % du temps d'exécution.

Une constatation similaire peut être faite dans le cas de Yaap: le nombre total de lignes est de 2800 (abondamment commentées), le noyau de l'interpréteur ne représentant également que 500 lignes. Là aussi, la proportion du temps passé dans le noyau est de plus de 90 %. Ceci semble être une règle relativement générale, le noyau des interpréteurs Prolog faisant couramment environ 500 lignes, le reste étant essentiellement constitué de l'environnement et des prédicats évaluables.

L'essentiel du problème de la machine Prolog est donc concentré dans un volume de programme bien délimité. On peut alors espérer trouver un ensemble d'opérations relativement simples dont l'implémentation matérielle permettrait des gains de performance très importants, le problème restant toutefois la généralité, car il ne s'agit pas de concevoir une machine adaptée à un interpréteur spécifique.

Une première étude des interpréteurs Prolog montre que, parmi les cinq procédures les plus utilisées, trois sont très courtes (les deux autres étant l'unification et la boucle principale de l'interpréteur):

- La procédure "desc" effectue le parcours des chaînes de variables (cette opération est appelée "déréférence" par Warren): c'est un simple parcours de liste qui demande 15 instructions C et prend 30 % du temps de l'interpréteur Yaap. Cette fonction est appelée très souvent et dure en moyenne 60 microsecondes environ, sans compter la durée de l'appel (elle comporte un paramètre).
- La procédure "substitue" effectue le stockage des substitutions. Elle représente 5 instructions C et 7,4 % du temps d'exécution. Elle possède deux paramètres et dure en moyenne environ 40 microsecondes.

- La procédure "saut_terme" recherche la fin d'un littéral. Elle représente 5 instructions et 3,7 % du temps. Elle possède un paramètre et dure en moyenne 50 microsecondes.

L'ensemble de ces procédures représente donc déjà plus de 40 % du temps. Leurs opérations se ramènent le plus souvent à quelques accès mémoire, et si on les implémente en matériel (par câblage ou microprogrammation) on peut espérer les accélérer d'un rapport supérieur à dix, ce qui amène un gain global de plus de 36 % pour les performances de l'interpréteur.

Ceci est évidemment encore loin d'être suffisant pour constituer véritablement une machine Prolog, mais on peut trouver d'autres opérations fréquentes et simples, au sein de l'unification, ou de la boucle principale de l'interpréteur. Cependant, avant même la définition des opérations, c'est au niveau de la manipulation des données et de l'adressage que l'on peut espérer les gains de performances les plus significatifs. Ainsi, pour Warren, l'exploitation de certaines caractéristiques du DEC 10 constitue un des points clés des performances de son compilateur Prolog.

VII.2.1. Type des données:

Une caractéristique de Prolog est que les données ne sont pas statiquement typées: une variables peut contenir une données de type quelconque. L'interpréteur doit donc reconnaître le type de la donnée pour pouvoir effectuer le traitement qui convient. Ceci se fait généralement à l'aide d'un préfixe d'un octet ou de quelques bits indiquant le type de la donnée.

En terme de matériel, ceci conduit à une architecture "taggée", où le traitement effectué par une opération sur une donnée va être influencé par le type de cette donnée. Cette solution a été rencontrée sur diverses machines langages [SCH 78], mais on la rencontre surtout maintenant sur des machines Lisp (Symbolics 3600, LMI Lambda, Maia). Elle permet de simplifier le jeu d'opération de la machine, au prix d'un décodage plus complexe. Les conversions entre les types devront être assurées par le matériel.

La présence d'un tag limite la largeur utile des données avec les tailles conventionnelles (16 ou 32 bits). Le codage d'une dizaine de types différents va demander 4 bits, auxquels il peut être utile de rajouter quelques indicateurs. Avec une taille de mots de 32 bits, il ne reste donc que 28 bits pour stocker la donnée, ce qui est souvent insuffisant. Un tel choix ne semble intéressant que pour une machine de bas de gamme de type microprocesseur.

Pour des machines plus puissantes, le choix qui s'impose semble être d'avoir une taille de mots de 40 bits comportant un tag de 8 bits. Il est clair qu'une taille variable par octets constituerait le meilleur compromis sur le plan de l'utilisation de l'espace mémoire, mais au prix d'une partie contrôle plus complexe.

Le codage des données sur des nombres entiers de mots de 16 à 20 bits correspondrait assez bien à ce qu'il est possible de manipuler sur un microprocesseur ni trop coûteux, ni trop complexe à mettre en oeuvre (type MC 68000). Nous n'effectuerons pas de choix dans le cadre de cette thèse, mais indiquerons ce qui semble être le minimum nécessaire pour coder les différents types d'objet manipulés:

- Les caractères ne demandant que 7 bits, pourront être codés sur 16 bits, en comptant 8 bits de tag. Le traitement de caractères individuels est cependant assez rare, et l'optimisation de leur stockage semble peu importante.
- On considère qu'un nombre assez réduit de variables peut être autorisé dans une clause (32 ou 64), ce qui ne peut qu'inciter le programmeur à structurer ses programmes. Par contre, une variable devrait être stockée sous forme d'une déplacement dans la pile des instances, de façon à faciliter l'adressage. On devrait donc coder les variables sur au moins 16 bits.
- On peut limiter la longueur des chaînes à une valeur assez basse, des textes plus longs pouvant être stockés sous forme de listes. Les chaînes peuvent donc être représentées sous forme d'une longueur sur 8 à 10 bits, suivie des caractères.
- Nous avons déjà vu le cas des entiers, qu'il est souhaitable de pouvoir coder sur 32 bits.
- La norme IEEE fixe la longueur des réels à 64 bits (stockage) ou 80 bits (résultats intermédiaires des calculs). Cependant, on se contente fréquemment de réels courts codés sur 32 bits. Le codage des réels constitue cependant un réel problème si on veut rester compatible avec la norme IEEE, pour permettre l'utilisation de coprocesseurs de calcul, ou l'interface avec des bibliothèques de calcul flottant. Une solution peut être un codage indirect: on passe par l'intermédiaire d'un pointeur, codé sur 32 ou 40 bits, comportant un tag "référence à un réel", et l'adresse de la valeur correspondante, cette valeur étant codée en format IEEE.
- Les symboles doivent comporter, outre le type, leur nombre de fils, et une adresse (ou un déplacement dans un tableau). On peut coder le nombre de fils dans le type lui même: c'est la solution que nous avons adopté dans Yaap, les types de 0 à 127 représentant des symboles, d'arité égale à la valeur. Le nombre de fils est largement suffisant pour les applications rencontrées jusqu'ici, et on peut même le diminuer un peu pour avoir des tags plus courts (6 ou 7 bits), ou récupérer des bits à d'autres fins. Une autre solution est de coder le nombre de fils dans un champs séparé, mais on réduit alors le champs disponible pour l'adresse. Celle ci doit permettre l'accès au dictionnaire, qui permet d'obtenir la forme externe du symbole, et d'accéder au paquet de clauses qui lui est éventuellement associé. On peut enfin stocker le nombre de fils dans le dictionnaire, mais ceci peut poser un problème au niveau de l'unification, celle ci ayant besoin d'accéder le nombre de fils de chaque symbole traité.
- Les applications de l'intelligence artificielle demandent des espaces mémoire importants, qui sont rendus possibles par les progrès de la technologie. L'espace d'adressage doit donc être supérieur à 16 Millions d'octets, ce qui nécessite de stocker des pointeurs sur plus de 24 bits. Cependant, un adressage par mots, et relatif à un espace mémoire donné (clauses, piles, dictionnaire, etc...) peut permettre, pour de petites machines, d'obtenir un espace d'adressage suffisant avec un nombre de bits réduit.

VII.2.2. Adressage:

VII.2.2.1. Modes d'adressage:

Le problème de l'adressage est essentiel dans l'interprétation de Prolog. En effet, les interpréteurs passent une bonne partie de leur temps à accéder des informations par le biais de pointeurs et d'indexation. Ainsi, on estime que le passage de Pascal en C d'un interpréteur Prolog permet de gagner à peu près un rapport 2 dans les performances, dû en grande partie à la souplesse du mécanisme de pointeurs de C.

De même, une exploitation améliorée de l'adressage du MC 68000 dans l'interpréteur Yaap permet des performances sensiblement meilleures, malgré une capacité opérative accrue (traitements sur 32 bits au lieu de 16).

Une première constatation est que la plupart des références mémoire sont indirectes, et même fréquemment doublement indirectes (un pointeur pointe sur un pointeur qui pointe sur la donnée). L'indirection à partir d'une adresse contenue soit dans un registre, soit en mémoire semble donc une caractéristique essentielle d'une machine Prolog. La possibilité d'indirection multiple serait même encore préférable.

L'adressage relatif (base + déplacement) est également nécessaire: c'est en particulier le mode d'adressage des variables. Cependant, ici encore, le déplacement ne se trouve ni dans l'instruction elle même ni dans un registre, mais dans un opérande qui doit être recherché en mémoire.

VII.2.2.2. Codage des adresses:

Nous avons vu précédemment qu'une adresse peut être codée sur 24 ou 32 bits. Or, il est fait fréquemment usage dans l'interprétation de Prolog de couples de pointeurs, ou "molécules", comportant un pointeur vers les clauses et un pointeur vers les environnements (partage de structure).

Une représentation efficace des molécules et des adresses qui la composent est donc un point essentiel pour les performances d'une machine Prolog: on doit pouvoir manipuler séparément les éléments d'une molécule, ou manipuler globalement l'ensemble de la molécule. L'accès à l'élément de données désigné par une molécule correspond à l'opération "desc" dont nous avons déjà vu l'importance.

VII.2.2.3. Gestion mémoire:

L'interpréteur Prolog comporte plusieurs espaces distincts: les clauses, le dictionnaire, la (ou les) pile(s) des instances, la pile d'évaluation, la trace, auxquels il faut encore ajouter l'espace mémoire nécessaire au programme de l'interpréteur lui même. Ces espaces évoluent de façon relativement indépendante suivant l'application, et provoquent une fragmentation importante: certains programmes peuvent demander peu d'espace pour les clauses, mais plus pour les instances, par exemple...

Dans une machine Prolog, il est souhaitable de permettre une allocation dynamique de ces espaces. Ceci peut être effectué par logiciel, mais au détriment des performances:

on peut regrouper d'un coté des informations gérées en pile (environnements, trace, variables locales), et de l'autre les clauses, le dictionnaire et les variables globales gérées en tas.

Une autre solution serait d'avoir un mécanisme de segmentation, permettant de gérer séparément les différents espaces. Un pointeur peut alors être relatif à un espace donné (par exemple, une molécule est composée d'un pointeur vers une clause, et d'un pointeur vers une instance).

La mémoire virtuelle est une solution intéressante, qui commence à faire son apparition sur quelques microprocesseurs (par exemple, 68010 ou 68020). Cependant, la localité des références dans l'interprétation de Prolog n'est pas évidente.

Le comportement de Prolog sous mémoire virtuelle a fait l'objet de mesures [BER2 86]: dans Prolog I+ et dans Yaap, on a inséré des procédures stockant sur disque les adresses référencées dans les espaces de travail. Ces adresses ont fait ensuite l'objet de traitements visant à évaluer la localité des références, et en particulier en simulant le fonctionnement d'une mémoire virtuelle avec un algorithme de remplacement des pages de type LRU. Ces mesures démontrent un comportement assez médiocre, le taux de défauts de pages n'étant généralement acceptable que lorsque l'interpréteur dispose d'une mémoire réelle équivalente à environ 50% de l'espace qu'il utilise effectivement

Il va de soit que ces mesures pourraient être améliorées: en particulier, nos interpréteurs ne connaissent pas la notion de variables locales, qui permet de gagner en espace mémoire et aussi probablement en localité. La taille des programmes test utilisés reste modérée (178 K). Enfin, on pourrait vraisemblablement améliorer le comportement de l'interpréteur Prolog sous mémoire virtuelle au niveau des structures de données manipulées.

Les techniques classiques de gestion de la mémoire virtuelle semblent également mal adaptées, et en particulier l'algorithme de remplacement LRU. Un exemple simple peut nous en convaincre. Considérons le cas du retour arrière: celui ci provoque un dépilement qui peut amener a réaccéder des pages qui ont été swappées. Il faudra donc choisir des pages qui peuvent être remplacées pour permettre de ramener les nouvelles pages en mémoire. Ces pages ne seront vraisemblablement pas les pages du sommet de pile qui viennent d'être libérées, puisqu'elles viennent d'être accédées, mais ce pourront être d'autres pages de la pile qui risquent à leur tour d'être bientôt accédées. De plus, lorsque les pages du sommet de pile seront à leur tour swappées, elles seront réécrites en mémoire secondaire, puisqu'elles ont vraisemblablement été modifiées lors de l'avancement. Or, ces pages ne sont plus porteuses d'information utile. Elles n'ont donc pas besoin d'être réécrites, ni même d'être relues lorsque l'avancement amène à les accéder de nouveau. Elles devraient être swappées prioritairement, ce qui ne coûte rien en termes d'accès disques (sauf peut être la dernière page allouée, qui risque d'être vite a nouveau allouée en cas d'avancement).

Il devrait ainsi être possible de trouver des heuristiques de remplacement qui tiennent compte du comportement même des programmes Prolog. Ceci demande encore beaucoup de travail, et peut faire l'objet de nombreux travaux de recherche.

On admettra donc qu'il est possible de trouver des implémentations de mémoire virtuelle en Prolog qui soient efficace, à condition toutefois que le rapport de taille entre la mémoire virtuelle et la mémoire réelle reste modéré.

Enfin, un mécanisme matériel de récupération automatique de l'espace est également envisageable, et fait l'objet d'un projet à l'IRISA [BEC 83].

VII.2.3. L'unification:

Il est tentant de dire qu'une machine Prolog doit exécuter l'unification par matériel. Les choses ne sont en fait pas si simples: l'unification ne représente en soit qu'environ 25 à 30 % du temps de traitement des interpréteurs, pourcentage qui peut passer à 50 % environ si des opérations telles que la déréférence, la création des substitutions et le parcours de termes sont câblées. Par ailleurs, l'unification représente toujours une opération relativement complexe, qui peut difficilement être exécutée par le matériel.

Il peut aussi exister diverses implémentation de l'unification, avec des fonctionnalités différentes. Dans Prolog II, par exemple (et plus encore sans doute dans Prolog III), l'unification doit prendre en compte des contraintes placées sur les variables par le biais de prédicats évaluables tels que le "diff". Les compilateurs Prolog par ailleurs décomposent l'unification des en-têtes de clauses en opérations plus simples, compte tenu du fait que ces en-têtes sont connus à la compilation. Si on considère par exemple l'en tête de clause c(X,a,X), le traitement de la première occurence de X se ramène à une substitution du terme correspondant dans le but, le traitement de a se ramène à une comparaison avec le terme correspondant, ou à une substitution si celui ci est une variable, et enfin, la seconde occurence de X peut déclencher une unification, puisque on ne sait pas quelle sorte de donnée a été liée à X dans sa première occurence. Le poids de l'unification devient donc nettement plus faible.

Plutôt qu'une implémentation matérielle figée et complexe de l'unification, il paraît donc préférable de rechercher des primitives, des modes d'adressage et un codage des données qui en permettent une implémentation logicielle efficace. Ainsi, l'unification passe une grande partie de son temps à tester le type des données, et à accéder à ces données par l'intermédiaire de molécules.

VII.2.4. Opérations:

Nous avons vu, dans l'introduction, l'exemple de trois opérations qui peuvent amener un gain de performances important si on les réalise par matériel. En règle générale, on peut estimer que le temps d'exécution de ces opérations est à peu près égal au temps des accès mémoires.

VII.2.4.1. Déréférence:

Cette opération (la primitive desc de nos interpréteurs) n'est autre qu'une indirection multiple par le biais des molécules: elle consiste à suivre une chaîne de variables liées entre elles pour accéder à l'information qui leur est liée.

Les chaînes sont le plus souvent très courtes: la longueur moyenne mesurée sur divers types de programmes Prolog est de l'ordre de 1,5. En se basant sur des molécules stockées sur 4 mots de 20 bits, on aura:

- un accès à la molécule définissant l'environnement et la variable initiaux,
- 1,5 accès aux molécules de la pile des instances,
- 1,5 accès aux variables (1 mot),
- un accès à la donnée finale (2 mots),

Soit un total de 14 accès en moyenne, plus deux accès pour la lecture de l'instruction elle même. Si on compte un temps de cycle de 300 nanosecondes, l'opération prendra en moyenne 5 microsecondes, ce qui est inférieur à la durée d'un appel de sous programme avec un paramètre sur MC 68000. Le câblage de ce déréférence fait donc gagner pratiquement 30 % sur le temps d'exécution d'un interpréteur tel que Yaap.

VII.2.4.2. Substitue:

Il s'agit également d'une opération assez simple qui consiste à stocker une molécule dans la pile des instances, et à empiler l'adresse dans la trace si elle est inférieure au pointeur "environnement local". Cette opération demande jusqu'à 9 accès mémoire. On peut donc espérer gagner largement les 7,4 % du temps que représente cette procédure.

VII.2.4.3. Saut:

Cette opération consiste à rechercher la fin d'un terme: elle balaye le terme en maintenant un compte du nombre de fils à traiter. Chaque fois qu'un symbole fonctionnel est trouvé, le compte est augmenté du nombre de fils, chaque fois qu'un élément est traité, il est décrémenté. Seul le type et éventuellement le nombre de fils des données est accédé, si bien qu'un seul accès mémoire est effectué par élément de donnée. Le nombre moyen d'éléments accédés est faible: de 4 à 5, soit une durée d'exécution de 1,2 à 1,5 microsecondes.

VII.2.4.4. Autres opérations:

Les opérations précédentes concernent environ 40% du temps de traitement des interpréteurs Prolog. Il est possible de trouver d'autres opérations fréquentes dans le corps de l'interpréteur, la procédure d'unification, et les prédicats évaluables, bien que leur poids soit plus difficile à estimer.

La plus grande partie d'une procédure d'unification consiste à tester le type des éléments et à les comparer. Une opération de base consisterait alors à comparer deux éléments de données, et à distinguer trois cas:

- Les éléments sont égaux et non variables: il faut retourner le nombre de fils éventuel de l'élément,
- Les éléments sont différents et non variables (échec de l'unification),
- Un au moins des éléments est une variable,

Une autre approche du même problème est suivie dans la machine Maia, ou une instruction permet de tester deux tags.

Dans le corps de l'interpréteur, on trouve surtout des opérations de type manipulation de pointeurs, et en particulier beaucoup de doubles indirections qui bénéficieront directement d'une amélioration de l'adressage.

Des opérations d'un poids plus élevé se trouvent dans le backtracking: il s'agit d'une partie de programme relativement courte (23 lignes de C) qui est exécutée aussi souvent que l'appel d'un littéral (avancement), puisqu'à tout appel doit correspondre un retour. Nous avons évalué le pourcentage du temps passé dans cette opération, en en faisant un sous programme, ce qui permet l'analyse par prof: on s'aperçoit que la boucle principale de l'interpréteur passe presque 50% de son temps dans le backtracking.

celui ci est constitué d'opérations relativement simples, mais répétitives qu'il serait intéressant de réaliser en matériel:

- recherche d'un point de choix (parcours d'une liste chaînée en testant une condition),
- remise à zéro des substitutions à partir de la pile et de la trace.

Le câblage ou la microprogrammation de ces opérations permettraient de gagner un rapport 10 dans leur exécution, c'est à dire environ 40% sur la boucle principale de l'interpréteur.

VII.2.5. Conclusions:

Nous avons pu évaluer que 50 % du temps d'exécution de l'interpréteur peut être gagné par câblage de certaines opérations. Ce gain peut sans doute atteindre un facteur 3 à 5, si ces opérations sont complétées de modes d'adressages appropriés et de manipulations des structures de données. L'obtention d'un gain plus important se heurte à une limite qui est le nombre d'accès à la mémoire de travail requis par l'interprétation de Prolog: Prolog effectue en effet un nombre important d'accès mémoire, comme nous avons pu le vérifier au cours de nos évaluations. Cette limite doit donc être prise en compte et repoussée par l'intermédiaire de dispositifs tels que caches, mémoires entrelacées, etc...

La remarque faite au sujet de la gestion de mémoire virtuelle avec les espaces gérés en pile (inadaptation des algorithmes de remplacement) reste valable pour un cache. Ceci peut être pris en compte par l'intermédiaire d'instructions d'empilement et de dépilement qui permettraient d'indiquer au cache, lors d'un dépilement, que le mot dépilé peut être remplacé sans réécriture en mémoire, et lors de l'empilement, que le mot alloué n'a pas besoin d'être lu en mémoire. Ceci semble prévu dans la machine PSI [YOK 83].

Les considérations qui précèdent concernent essentiellement l'interprétation de Prolog. Or, il existe un rapport de performances de 10 à de 20 entre un Prolog compilé et un Prolog interprété. Le développement de l'utilisation pratique de Prolog passe donc bien par des implémentations compilées.

Nous n'avons pas eu la possibilité jusqu'ici d'effectuer des mesures et évaluations sur des compilateurs Prolog, ne disposant pas d'un tel outil. ceci est cependant prévu

dans l'avenir. En fait, il s'avère que les structures de données manipulées par les programmes compilés, et les opérations effectuées sont les mêmes que pour l'interprétation. En effet, la compilation d'un programme Prolog n'est autre que la "mise à plat" de l'interpréteur, avec élimination ou simplification de certaines opérations (tests de types de constantes, par exemple). On peut donc s'attendre à ce que le comportement des programmes compilés reste assez proche de celui des programmes interprétés, avec sans doute des variations dans le poids des opérations.

Si on considère le langage WAM, il est clair que les types d'opérations mises en jeu restent les mêmes que ceux que nous avons précédemment mentionnés.

VII.3. L'APPROCHE UNIFICATION:

Ayant conçu, dans le cadre du projet Opale un filtre capable d'effectuer la préunification au vol, il nous semble intéressant de voir si un tel filtre ne pourrait pas servir pour une machine Prolog. Nous avons cependant vu que l'exécution matérielle de l'unification, aussi performante soit elle, ne suffit pas à faire une machine Prolog de performances intéressantes, et il faut ajouter autour d'autres mécanismes, dont, peut être, quelques uns de ceux qui ont été envisagés au paragraphe précédent.

Le problème est cependant que ce filtre a été conçu pour une base de données: le filtrage est rapide, mais le temps de compilation ne peut être récupéré que si le nombre d'alternatives est suffisamment élevé: ce n'est pas le cas en programmation. De plus, le filtrage est basé sur l'hypothèse que les variables des buts et des en-têtes de clauses sont le plus souvent indépendantes, ce qui n'est peut être plus vérifié en dehors du contexte des bases de données.

Tel quel, le filtre n'est donc pas utilisable. Cependant, il est intéressant de noter que l'unification est commutative: donc, plutôt que de compiler des buts (générés dynamiquement) pour filtrer des en-têtes de clauses (statiques), il est possible de compiler les en-têtes de clause pour filtrer les buts. L'intérêt est alors que la compilation des en-têtes de clauses se fait une fois pour toutes (à l'instar des langages de programmation courants, ou de Prolog compilé), elle devient donc non significative au niveau de l'exécution du programme.

Le filtre opérant cette fois sur une mémoire primaire, on n'a plus de contraintes de temps aussi fortes sur le temps d'exécution, et la totalité des environnements sont disponibles, ce qui permet d'utiliser une représentation du type partage de structure. On peut donc avoir un pipe-line composé d'un automate exécutant l'opération "desc", le filtre, et un automate stockant les substitutions (Fig VII.1.). Cependant, les substitutions concernant des en-têtes de clauses multiples, elles ne peuvent toujours pas être répercutées dynamiquement au cours du filtrage, et on doit donc en rester à l'exécution d'une préunification.

Dans cette approche, contrairement à la première, on remet en question la méthode d'interprétation de Prolog. Effectuant une unification "en largeur", on réduit une bonne part du travail de l'interpréteur, qui consiste à enchaîner des appels d'unification.

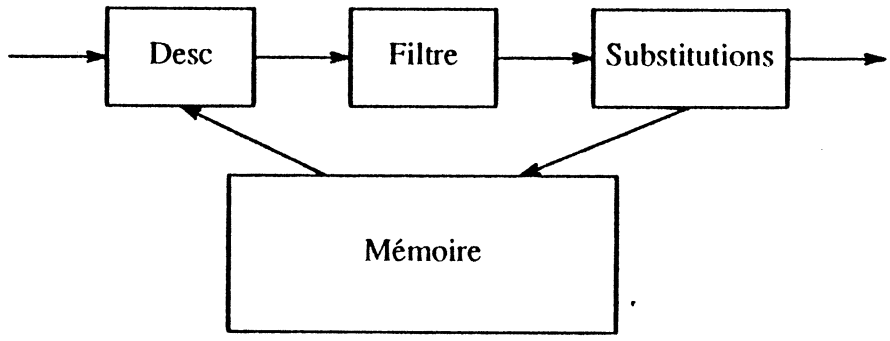


Figure VII.1.

L'efficacité de cette solution va dépendre pour une bonne part des dépendances qui existent entre les variables des buts ou des en têtes de clauses, puisque ce sont ces dépendances qui peuvent faire que des termes préunifiables ne soient pas unifiables.

VII.3.1. Evaluation des dépendances:

Des mesures ont été faites sur divers programmes Prolog: L'interpréteur Prolog I+ a été piégé pour détecter la présence de dépendances dans les opérandes de l'unification.

On distingue:

- les dépendances dynamiques (crées au cours de l'interprétation par liaison de variables indépendantes),
- les dépendances statiques (existant dans le programme source),
- les dépendances dans les buts,
- les dépendances dans les en-têtes de clauses: ces dernières étant connues lors de la compilation du programme peuvent en effet être traitées plus efficacement.

Considérons un exemple:

```
c1(X,Y) :- c2([X|L],X), c3(Y,Z), c4(Y,Z).
c2([a|b],X).
c3(X,X).
c4(a,a).
```

Dans cet exemple, dans la clause c1, le littéral c2 comporte une dépendance statique (deux occurences de la variable X), alors que les littéraux c3 et c4 n'en comportent aucune. Après vérification de c3, la clause c4 comportera une dépendance dynamique, puisque Y et Z ont été liés par la clause c3.

Les programmes utilisés pour la mesure sont les suivants:

- superviseur Prolog: le programme qui assure la lecture et le chargement des clauses Prolog. C'est donc un cas d'assez gros programme. 4 cubes: programme extrait de [COL 79]. Ce programme est assez court, mais très long à l'exécution.
- 8 reines: problème classique, avec visualisation de l'échiquier à l'écran pour chacune des solutions.
- quick sort: programme de tri rapide,
- naïve reverse: programme d'inversion de listes volontairement naïf faisant appel à la concaténation. Ce programme fait partie des jeux d'essais classiques pour l'évaluation de performances des interpréteurs Prolog [WAR 77].
- compilateur: compilateur d'un langage très simple, générant du code assembleur pour une machine hypothétique elle même très simple. Autre exemple d'un assez gros programme Prolog.

D'autre programmes ont été mesurés, mais de façon non systématique. Leurs résultats confirment ceux que l'on mentionne ici.

Le tableau VII.1 montre le pourcentage d'unifications dans lesquelles intervient au moins une dépendance, soit dans les buts, soit dans les en-têtes de clauses. Ce taux est très variable, mais généralement assez fort, pouvant aller jusqu'à plus de 50 % des unifications. Cependant, ces dépendances sont, pour l'essentiel des dépendances dans les en-têtes de clauses, les dépendances dans les buts étant beaucoup plus rares.

programme	dep. buts	dep. en-têtes
superviseur Prolog	11.7%	24.4%
4 cubes	0.0%	72.0%
8 reines	0.0%	41.0%
quick sort	1.2%	50.4%
naive reverse	4.9%	24.4%
compilateur	18.0%	24.3%

Tableau VII.1.

VII.3.2. Sélectivité de la préunification:

Le taux de dépendances n'est pas totalement significatif. Il importe donc d'évaluer plus précisément le taux d'unifications qui échouent alors que la préunification réussit: ceci peut encore se faire en piégeant un interpréteur pour lui faire exécuter une préunification avant l'unification, et mesurer le taux d'échecs respectif des deux opérations.

Ceci a été fait sur l'interpréteur Prolog I+. Quelques résultats sont présentés dans le tableau VII.2. Nous reprenons les mêmes exemples que dans le tableau VII.1, auxquels on a ajouté un programme de dérivation formelle adapté de celui présenté par Colmerauer [COL 83]. Le problème des 8 reines et celui des 4 cubes constituent en fait un cas extrême. Le compilateur constitue presque le cas minimal, et les autres programmes

semblent correspondre au cas moyen.

Pour chaque programme, on donne:

- dans la première colonne: le pourcentage de préunifications réussies (donc, le pourcentage d'unifications tentées par rapport aux préunifications).
- dans la seconde colonne: le pourcentage d'échecs de l'unification par rapport au nombre total de préunifications exécutées,
- dans la dernière colonne: le pourcentage d'échecs de l'unification par rapport au nombre d'unifications tentées (après succès de la préunification).

programme	préunif.	échecs/préunif	échecs/unif
superviseur Prolog	52.3%	3.8%	7.3%
4 cubes	49.7%	1.5%	23.2%
8 reines	88.1%	6.9%	19.1%
quick sort	58.1%	2.1%	3.6%
naive reverse	50.8%	1.7%	3.3%
compilateur	48.3%	1.3%	2.7%
dérivation	52.1%	5.5%	10.6%

Tableau VII.2.

On voit donc que la préunification permet de filtrer une grande part des cas d'échec de l'unification: plus de 90% des en têtes de clauses sélectionnés par préunification unifient également, ou en d'autres termes, sur 100 unification, 50 sont des échecs, dont 40 sont détectés par préunification. La préunification filtre donc 80% des cas d'échec.

VII.3.3. Association:

Il reste maintenant à évaluer le travail nécessaire pour compléter l'unification après préunification (association): le tableau VII.1 montre en effet que ce travail peut être nécessaire dans plus de 50 % des cas. Intuitivement, le fait que peu d'échecs soient entraînés par l'association semble indiquer que la plupart des dépendances ont pour rôle de transférer un résultat d'un argument dans un autre, plutôt que de constituer un critère de sélection en vérifiant que deux éléments sont égaux. Prenons l'exemple classique de la concaténation:

```
conc([],X,X).
conc([X|Y],Z,[X|T]) :- conc(Y,Z,T).
```

Dans la majeure partie des cas d'appel, soit le premier argument, soit le troisième sera libre. Dans la seconde clause, les deux occurences de X servent donc le plus souvent à transporter le premier élément de la première liste en tête de la troisième, ou vice versa. Il en est de même de la première clause. Par contre, si on a un appel du type: conc("ab",X,"abcd"), la dépendance jouera bien un rôle de sélection, mais ceci n'est pas ici le cas d'appel courant.

Il est difficile de vérifier cela statistiquement au niveau des buts: on n'a en effet pas de moyen simple de savoir au niveau de l'unification qu'une variable dépend d'une autre. De toutes façons, le taux de dépendances dans les buts est faible. Par contre, dans les en-têtes de clauses, on peut analyser les différents cas rencontrés dans l'unification. On sait en effet que si on rencontre une variable liée dans l'en-tête de clause, cela ne peut être que du fait d'une occurence précédente de la variable. Il est donc possible de compter les cas où une variable liée de l'en tête de clause est comparée à un élément du but, en distingant le cas où l'élément correspondant est une variable (qui sera simplement liée à la variable de l'en-tête), de celui où cet élément est un terme (qui entraîne une comparaison, et éventuellement une nouvelle unification).

Le tableau VII.3. présente les résultats des mesures effectuées sur les seules unifications consécutives à un succès de la préunification. On présente pour les divers exemples:

- le nombre moyen d'itérations de l'unification (c'est à dire le nombre d'éléments de termes traités). Rappelons qu'il s'agit ici de l'unification complète du but et de l'alternative.
- le nombre de cas (pour 100 unifications) où une variable liée de l'alternative est unifiée avec une donnée non variable du but,
- le nombre de cas (pour 100 unifications) où une telle variable est liée à un terme non atomique et entraîne donc une unification complète. On note que le nombre moyen d'argument du terme est dans tous les cas strictement égal à deux, ce qui correspond à des listes ou des chaînes de caractères.
- le nombre de cas (pour 100 unifications) où une variable liée de l'alternative correspond à une variable libre du but.

programme	iter.	lie/lie	non at.	lie/libre
superviseur Prolog.	4.6	23.9	13.0	12.3
4 cubes	5.4	30.2	0.0	19.4
8 reines	4.1	23.2	0.0	5.5
quick sort	6.2	5.0	1.1	8.5
naive reverse	5.7	4.9	1.2	11.7
compilateur	5.0	4.5	1.1	29.8
dérivation	4.1	22.4	7.1	29.8

Tableau VII.3.

Ce tableau nous montre d'abord que les unifications sont le plus souvent assez simples (de 4 à 6 éléments). Le nombre de cas où une variable liée de l'en tête de clause est unifiée avec une variable liée du but n'atteint 30% que dans le cas quasi pathologique du problème des 4 cubes. De plus, le nombre de cas où il s'agit de termes non atomiques est extrêmement faible: 13% dans le cas du superviseur Prolog, où il s'agit très vraisemblablement de traitements de chaînes de caractères.

On peut donc dire que l'association sera dans l'immense majorité des cas très simple: dans plus de 75% dés cas, elle se ramène à la génération de substitutions, et dans moins de 10% des cas, il s'agit d'une unification entre termes non atomiques.

VII.3.4. Performances:

Compte tenu de la rapidité du filtrage, et si les opérations "desc" et "subst" sont câblées, on peut espérer gagner ainsi environ 90% du temps d'exécution d'un ensemble d'opérations qui représentent 60% du temps d'exécution d'un interpréteur Prolog. Ceci demande en outre que les rares cas où des dépendances existent dans les buts puissent être aisément détectés et traités: ceci peut faire partie du traitement des substitutions.

Le filtrage permet en outre de simplifier le corps de l'interpréteur, en évitant un nombre appréciable d'appels de l'unification et de backtracking. Par ailleurs, les techniques mentionnées dans la première approche restent applicables ici. On peut donc espérer un gain de performances important par rapport à la première solution. On peut encore aller plus loin en utilisant des techniques de pipe-line ou de OU-parallélisme analogues à celles exposées dans le chapitre VIII.

Il est bien évident qu'une telle solution est plus coûteuse en mémoire que les techniques classiques d'interprétation de Prolog, puisque les solutions générées par une préunification doivent être (au moins en partie) conservées jusqu'à l'essai de la dernière solution possible, ou la rencontre d'une coupure. On peut s'attendre par contre à ce que le stockage des en-têtes de clauses sous forme de tables de filtrage soit plus compact que le stockage classique: ceci est un avantage par rapport aux programmes Prolog compilés.

Un dernier point concerne la complexité du filtre, car les paramètres adoptés dans le cas d'une base de données ne sont pas forcément appropriés pour des programmes Prolog (nombre d'en-têtes de clauses, nombre de valeurs, taille maximale de la structure...).

VII.3.5. Complexité du filtre:

Les tailles de tableaux qui ont été considérées au chapitre VI correspondent aux hypothèses des bases de données. Dans un programme Prolog, on s'attend à ce que ces tailles soient beaucoup plus réduites.

Ceci peut être estimé si on analyse un ensemble de programmes: on s'intéresse particulièrement à la taille des paquets de clauses, aux nombre total de constantes différentes dans les en-têtes d'un même paquet, et à la taille de la structure englobante des en-têtes d'un paquet de clauses.

Ces données ont été mesurées. Il s'agit cette fois de mesures statiques effectuées à partir des programmes sources. On retrouve quelques uns des programmes précédemmment mesurés (compilateur, dérivée), plus un fichier de benchmarks regroupant entre autres les programmes de 4 cubes, des 8 reines, le quick sort et le naïve reverse. On ajoute à cela les statistiques cumulées sur l'ensemble des fichiers analysés (comprenant d'autres fichiers que ceux que l'on a mentionnés). Les résultats sont présentés dans le tableau VII.4.

programme	nb alternatives		nb constantes		taille st. engl	
	moy	max	moy	max	moy	max
benchmarks	2.4	11	6.1	43	13.8	102
compilateur	3.3	16	4.6	18	6.6	25
dérivation	3.5	21	3.5	14	5.8	28
total	2.8	25	4.2	51	7.26	102

Tableau VII.4.

Une autre donnée intéressante est la répartition des différents cas. La figure VII.2. présente les histogrammes correspondant aux différentes valeurs.

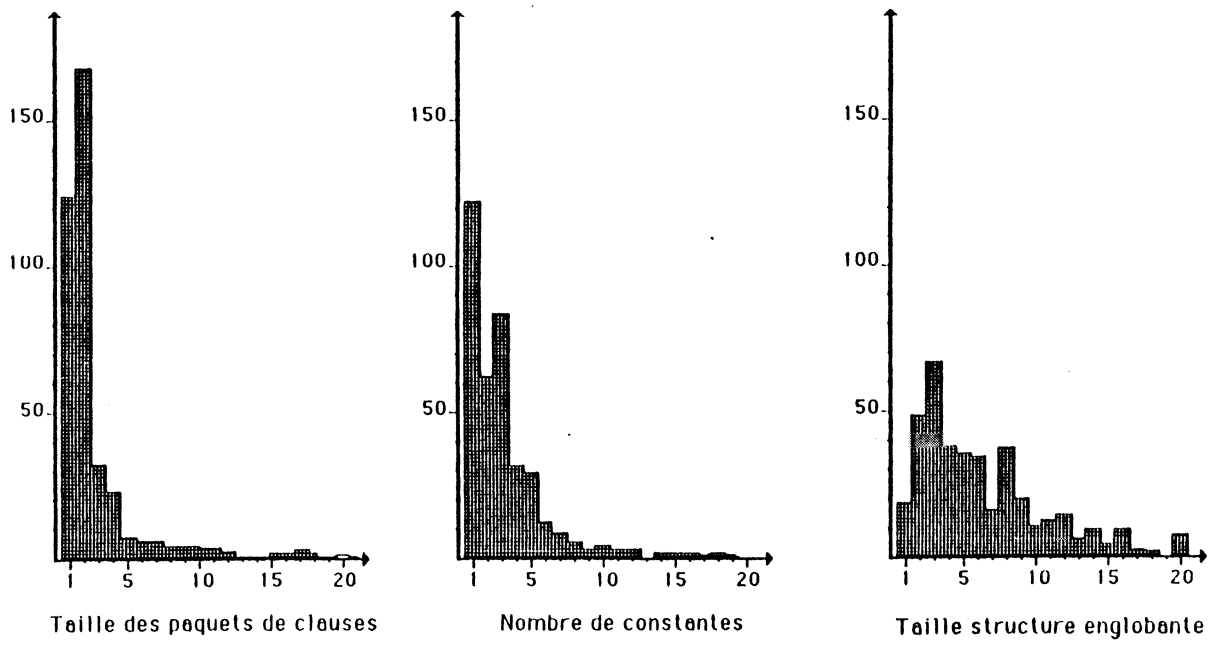


Figure VII.2.

- La taille des paquets de clauses est relativement faible, même si on met à part le cas des paquets réduits à une seule alternative qui jouent le plus souvent le rôle de sous programmes: ils peuvent recevoir un traitement particulier. Le filtre pourra donc permettre de traiter un maximum de 8 en-têtes de clauses (les clauses plus importantes pouvant être traitées en plusieurs filtrages). Compte tenu de la sélectivité de la préunification, on voit également que le problème du stockage des résultats en attente d'utilisation ne devrait pas être crucial.
- Le nombre de constantes contenues dans les en-têtes de clauses est également relativement faible: ici encore, un maximum de 8 semble suffisant, les cas plus importants pouvant être simplifiés au prix d'un traitement logiciel complémentaire. La recherche pourra donc se faire séquentiellement.

- Le cas de la structure englobante est un peu moins simple: les cas de structures de plus de 16 éléments ne sont en effet pas rares. De plus, la taille de cette structure est liée au nombre maximum d'arguments qu'un terme peut avoir. Compte tenu des programmes Prolog que l'on rencontre couramment, la taille maximum que l'on peut traiter en un filtrage devrait donc être fixée à 32.

Les états de l'automate de parcours de la structure englobante pourront donc être stockés sur au plus 32 bits, les listes de valeurs associées seront très courtes, et le plus souvent vides, enfin, la table de bits représentant les listes d'en-têtes pouvant unifier pourra être stockée sur 8 à 16 mots de 32 bits.

VII.3.6. Conclusion:

Cette étude montre l'intérêt d'une machine Prolog conçue autour d'un filtre. Un gain significatif peut être obtenu sur l'unification, et sans doute également sur l'interprétation elle même, étant donné que seules les alternatives filtrées seront essayées: on évite donc des tentatives d'unification, avec le traitement que cela implique dans le corps de l'interpréteur. Par ailleurs, le fait de n'exécuter qu'une préunification par matériel laisse la place à un complément logiciel de l'opération, permettant de faire évoluer ses fonctionnalités.

Cette approche pourrait éventuellement être exploitée au niveau d'une architecture multiprocesseurs, où le travail serait partagé entre un ou plusieurs processeurs d'unification, et un ensemble de processeurs exécutant le reste du traitement.

Cependant, cette étude est loin d'être complète, et on ne sait pas apprécier actuellement l'augmentation d'espace mémoire qu'implique le stockage de solutions multiples en attente de traitement, ni le traitement qu'implique leur gestion. De plus, nous avons vu que les paquets de clauses possèdent en fait un nombre réduit d'alternatives, si bien qu'on ne peut pas espérer qu'un nombre relativement élevé d'unifications puisse être fait en parallèle. La simplification de l'opération qui en résulte ne compense que très partiellement ce fait.

Enfin, il reste un grand nombre d'opérations que nous avons laissées dans le domaine du logiciel. Le filtre ne serait en fait qu'un opérateur annexe d'une machine qui pourrait par ailleurs être du style de ce que nous avons esquissé dans la première approche: ces approches sont donc bien complémentaires. Il resterait un travail à effectuer pour trouver éventuellement des mécanismes matériels bien adaptés pour effectuer ce traitement, et déboucher ainsi sur une architecture totalement non conventionnelle.

Une étape indispensable pour la conception d'une telle machine nous semble être la conception d'un interpréteur et/ou compilateur Prolog logiciel reposant sur ce principe, et qui permettrait de compléter les mesures. Ceci pourrait en fait être en partie réalisé à partir des éléments de la simulation logicielle d'Opale, le filtre lui-même dérivant également de celui d'Opale.

VII.4. CONCLUSION:

Dans ce chapitre, nous avons esquissé diverses manières d'aborder la conception d'une machine Prolog. Nous n'avons pas défini de machine, cependant, des éléments d'évaluation importants nous semblent se dégager.

Un point important est la place de l'unification dans l'interprétation de Prolog: la procédure d'unification ne prend en elle même qu'un quart du temps, et une machine Prolog ne saurait donc reposer uniquement sur un opérateur d'unification. D'autre part, les mesures effectuées autour de la troisième approche montrent que l'unification, si elle peut prendre des formes très complexes, est le plus souvent très simple. Cependant, on ne peut pas figer l'unification en la faisant exécuter entièrement par un opérateur matériel (sous réserve que cela soit possible), car des contraintes supplémentaires peuvent être introduites au niveau de l'unification, et il faut pouvoir les traiter par logiciel. La préunification semble cependant une solution intéressante.

On peut également considérer la possibilité de solutions faisant appel au parallélisme massif. Bien que notre étude sur ce point reste superficielle, il semble que les possibilités d'exploitation réelle du parallélisme restent limitées, au moins dans l'état actuel du domaine de recherche. On préférera donc des solutions basées sur un parallélisme limité, et l'exécution performante des opérations de base de Prolog. Par ailleurs, conformément à l'évolution de l'utilisation des machines, on préférera des solutions du type poste de travail individuel à des solutions basées sur le partage des machines: en cela, on rejoint les conclusions de notre première partie, et il semble en outre que des postes de travail "Prolog" de hautes performances pour un usage individuel soient beaucoup plus réalisables, et sans doute moins coûteux que de grosses machines partagées. Ces dernières ne sont pas sans utilité, mais elles seraient à réserver pour la résolution de problèmes de complexité exceptionnelle.

Chapitre VIII

CONCLUSIONS

Dans cette thèse, nous avons cherché à démontrer l'intérêt de la conception de systèmes organisés autour d'unités spécialisés, avant de nous intéresser plus particulièrement aux machines bases de données, puis à la programmation en logique. On peut dire que nous avons couvert un certain nombre des thèmes majeurs de la 5ème génération, sans que cela soit délibéré, d'ailleurs, puisque certains de nos travaux sont antérieurs à l'annonce du projet Japonnais.

Par ailleurs, nombre de problèmes abordés restent ouverts, certains n'ayant d'ailleurs été qu'effleurés: ceci est dû à l'aspect nouveau de certains aspects du domaine de recherche couvert, et au fait qu'une expérimentation n'a pas encore pu être menée a son terme. Il s'agit d'ailleurs d'une oeuvre de longue haleine qui devrait regrouper plusieurs équipes et disposer de moyens importants.

VIII.1. DISTRIBUTION FONCTIONNELLE:

Sur cet aspect, nous avons cherché rapidement d'une part a justifier la notion de système fonctionnellement distribué, et d'autre part à présenter quelques uns des problèmes posés, et des ébauches de solutions.

On voit maintenant les systèmes se décentraliser effectivement petit à petit. Nous n'avons pas eu les moyens nécessaires pour conduire une expérimentation dans ce domaine, mais on voit maintenant les réseaux se répandre, d'une part, et les systèmes évoluer vers la distribution des fonctions d'autre part. En particulier, on voit se répandre la notion de poste de travail: machines de puissances très diverses à vocation mono-utilisateurs. Les postes de travail présentés sont généralement prévus pour être intégrés dans un réseau, par l'intermédiaire duquel ils peuvent accéder à des ressources partagées (serveurs fichiers, serveurs d'impression, et passerelles vers un réseau de téléinformatique). On note également sur le Sun (machine UNIX à base de MC 68000) la possibilité d'avoir des machines sans mémoire secondaire capables d'accéder à un serveur fichiers via un réseau Ethernet.

Un point intéressant dans tout cela est que, semble t'il, la plus grande part des développements effectués sur les réseaux locaux le soient autour du système UNIX: le développement de ce système en train de s'imposer sur nombre de machines résoud en effet nombre de problèmes propres aux systèmes répartis, puisque l'on dispose d'un système unique sur toutes les machines. Des logiciels tels que la Newcastle Connection ou NFS, permettent à une machine d'accéder à l'ensemble des ressources des machines connectées sur un réseau comme si elles lui étaient locales, et TCP/IP permet une standardisation de la communication entre machines.

Il faut noter également que la notion de distribution fonctionnelle ne se limite pas à des machines distinctes liées par un réseau, mais qu'elle concerne également l'architecture des machines elles même: on peut en effet concevoir des machines

réparties, basées sur la coopération de plusieurs types d'unités spécialisées, et ceci sans que la distribution soit visible pour l'utilisateur. Les systèmes de communication peuvent être très divers (bus parallèle, commutation de circuits, mémoire partagée, communication en point à point,...), alors que les distances peuvent être le plus souvent réduites (les éléments de la machine sont situés dans une même salle, ou même dans une même baie). Les principes de la communication et de la coopération restent tout de même applicables, à ceci près qu'il n'existe pas forcément de problèmes de synchronisation, et que le contrôle peut être partiellement centralisé. Opale est, pour nous, un exemple d'une telle machine.

Par contre, il semble qu'encore peu de machines spécialisées aient vu le jour: les machines bases de données existantes ne sont le plus souvent qu'un processeur, ou un ensemble de processeurs dédiés à cette fonction, sans qu'on puisse parler véritablement de matériel spécialisé. Par contre, on trouve des matériels spécialisés dans des domaines tels que le calcul vectoriel ou le traitement d'images.

VIII.2. MACHINES BASES DE DONNEES:

Après des activités importantes, le domaine des machines bases de données semble avoir marqué une pause. On s'intéresse beaucoup à des problèmes d'évaluation, voire à une remise en question de la notion de machine base de données, mais peu de projets nouveaux ou d'idées nouvelles voient le jour. Bien évidemment, nous prétendons, avec le projet Opale, avoir apporté quelques idées nouvelles dans ce domaine, mais essentiellement, à notre avis, au niveau de l'utilisation de Prolog, plus que de l'originalité de l'architecture (celle d'Opale est relativement classique, ce qui n'est sûrement pas un défaut, d'ailleurs).

Cette utilisation de Prolog dans le domaine des bases de données semble encore loin de faire l'unanimité des spécialistes du domaine, et l'idée d'utiliser Prolog comme modèle de données en lieu et place du relationnel semble encore peu acceptée, du moins chez les spécialistes des bases de données, car elle trouve par ailleurs un large écho chez les spécialistes de l'intelligence artificielle ou de la programmation en logique. Elle est une des voies qui devraient permettre d'étendre significativement les possibilités des bases de données vers ce qu'attendent les spécialistes de l'intelligence artificielle.

Sans doute manque-t-il d'études théoriques à ce sujet, mais nous pensons avoir défini, et en partie validé des outils permettant l'utilisation de Prolog dans les bases de données, les problèmes essentiels étant selon nous l'unification et la stratégie de recherche, problèmes largement traités ici, et pour lesquels nous considérons avoir trouvé des solutions.

Le projet Opale rentre maintenant dans une phase expérimentale. L'algorithme d'unification a pu être validé par implémentation logicielle, et évalué dans le cadre d'un interpréteur Prolog orienté bases de données. La stratégie de recherche a fait l'objet d'une implémentation en OCCAM et d'une évaluation. L'étude de l'implémentation matérielle du filtre se poursuit - avec lenteur, vu le manque de moyens matériels.

Il semble que les étapes ultérieures du projet Opale devraient être les suivantes:

- Implémentation logicielle du système complet en logiciel, sous Unix,
- Intégration dans un environnement multiprocesseurs (SM 90),
- Déport du filtrage sur un module d'échange disques (processeur autonome),
- Accès parallèle à plusieurs disques par l'intermédiaire de plusieurs modules d'échange,
- Intégration du filtre matériel développé en parallèle.

Le problème de l'intérêt du filtre reste cependant ouvert. Le projet OPALE n'est pas encore assez avancé pour apporter un élément de réponse sur ce point, le problème étant que, si le filtrage permet un accès rapide aux données sur disque, des goulots d'étranglement situés en aval risquent de minimiser le gain de performances effectivement obtenu. Il s'agit là d'un problème important dont il ne semble pas possible de prédire à priori la réponse. Cependant, l'utilisation de matériel spécialisé (VLSI) ne nous semble pas inutile, dans la mesure où le coût de développement diminue considérablement. Tout le problème est de bien cerner le niveau des opérations à réaliser, sachant qu'une opération simple complétée par un traitement logiciel peut apporter des performances suffisantes.

VIII.3. MACHINES PROLOG:

Nous avons par ailleurs exploré quelques unes des voies possibles pour la conception d'une machine Prolog. Nous avons évalué deux voies, avec d'une part la conception d'une machine de type classique, et d'autre part une conception plus nouvelle, mais aussi plus spécialisée, avec l'utilisation d'un préunificateur matériel. Cette seconde approche semble permettre d'obtenir une machine plus performante, mais il reste beaucoup de points à éclaircir. Nous n'avons pas, en particulier, étudié les performances dans le cas de la compilation de Prolog, ni fait intervenir plusieurs méthodes d'interprétation. On ne peut pas encore prédire si le rapport complexité/performances est favorable à l'une ou l'autre approche, ce qui serait le véritable critère de comparaison. Or, il faut remarquer que la première approche aboutit à une machine optimisée pour l'exécution de Prolog, certes, mais qui reste universelle. Ici encore, l'intérêt d'une machine spécialisée reste une question ouverte.

Par ailleurs, le langage continue d'évoluer, et un point important dans la conception d'une machine Prolog semble être de ne pas bloquer ce type d'évolution.

Enfin, il reste probablement beaucoup de travaux à effectuer pour faire de Prolog le langage idéal d'une machine dite "intelligente". Ainsi, Prolog III doit présenter des extensions qui en feront un véritable système de résolution d'équations. On peut également chercher à rapprocher Prolog d'autres courants de la programmation moderne (langages orientés objets, programmation fonctionnelle...). D'autres types d'évolution peuvent concerner la stratégie de recherche (possibilité pour l'utilisateur de définir une stratégie particulière, de sélectionner des clauses en fonction de critères complémentaires de l'unification...). L'adaptation de Prolog à d'autres modèles d'interprétation (parallélisme), la définition de systèmes d'exploitation autour de Prolog restent encore des sujets de

recherche ouverts. D'autres problèmes sont sans cesse introduits par les progrès de l'intelligence artificielle, tels que le raisonnement incertain [KOD 85], ou l'apprentissage, ou encore l'utilisation directe d'autres méthodes d'inférence (chaînage avant).

Reste à savoir s'il appartient à un langage de programmation de résoudre les problèmes que l'on peut être amené à traiter, ou s'il doit être seulement l'outil permettant de mettre en oeuvre des solutions. En cela, il semble que Prolog ait donné lieu à un malentendu: le fait qu'il soit fortement inspiré des méthodes de démonstration de théorème a fait qu'on l'a souvent considéré comme un démonstrateur de théorème, un résolveur de problème, voire même un système expert. Ceci amène à considérer Prolog comme "un mauvais démonstrateur de théorème", un "mauvais système expert", etc... Tout ceci est erroné, car Prolog n'est qu'un langage de programmation, c'est à dire un outil permettant de mettre en oeuvre la solution des problèmes, et non pas de résoudre les problèmes eux mêmes. Par la rigueur de la méthode de programmation en Prolog, et par ses mécanismes sous jacents, Prolog est un outil puissant, et c'est un langage universel, même si, comme tous les langages de programmation jusqu'à aujourd'hui, il se révèle plus performant pour certains types de problèmes que pour d'autres.

Prolog n'est, par ailleurs, pas le seul langage que l'on peut dire de 5ème génération: Lisp y prétend également, de même que les langages orientés objet, les langages fonctionnels, etc... Par contre, certains mécanismes de bases de Prolog (et en particulier l'unification) apparaissent comme essentiels en intelligence artificielle. Quelque soit l'avenir du langage Prolog en lui même, on peut donc dire que les études matérielles conduites autour de ce langage concernent des problèmes beaucoup plus généraux, dont l'intérêt dépasse le cadre du langage Prolog lui même.

Les développements actuels de l'intelligence artificielle font que l'on ressent de plus en plus le besoin d'outils matériels pour l'intelligence artificielle: la popularité grandissante des machines Lisp en témoigne. C'est donc là aussi un domaine d'étude qui s'ouvre, et qui ne se limite sans doute pas aux machines Lisp ou Prolog. Il est possible par ailleurs que de telles machines ne soient qu'un point de passage vers des machines offrant au niveau matériel des fonctionnalités bien plus élevées, mais il s'agit d'un passage obligé, dans la mesure où déjà à ce niveau, des problèmes non triviaux existent, et doivent donc être résolus.

VIII.4. TECHNOLOGIE:

Extrapoler l'évolution de la technologie est un des exercices favoris des informaticiens. La littérature est pleine de publications prédisant la mort du disque magnétique au profit des mémoires à bulles (aux environs de 78), l'avènement prochain des technologies cryogéniques (effet Josephson, aux environs de 81), etc...

Certaines prospectives s'avèrent exactes, comme celles qui prédisent l'accroissement des capacités mémoires, ou à peu près exactes, comme celles qui prédisent l'augmentation de l'intégration des processeurs (d'après des prévisions d'il y a quelques années, le million de transistors devrait être chose courante aujourd'hui, mais ce n'est sans doute qu'un petit retard). Mais ces prévisions concernent l'évolution d'une

technologie donnée, et non le remplacement d'une technologie par une autre.

Il est clair que ceux qui prévoyaient le remplacement des disques magnétiques par les bulles magnétiques négligeaient deux choses:

- les bulles magnétiques étaient loin d'avoir fait leurs preuves,
- la technologie des disques magnétiques avait encore de larges capacités d'évolution.

On voit mal à l'heure actuelle quelle technologie pourrait, à court ou à moyen terme, remplacer le disque magnétique, même si d'autres technologies (disque optique) peuvent s'imposer sur certains types d'application. Ceci dit, il est difficile de concevoir des projets de recherche en informatique à long terme sans un minimum de réflexion sur ce que l'on peut attendre de l'évolution. Nous en avons fait nous même dans cette thèse.

Ainsi, la conception de la machine Opale est largement basée sur les caractéristiques des disques magnétiques. Cependant, ce qui est vrai pour le disque magnétique devrait le rester pour le disque optique, qui reste dans son principe d'accès identique au disque magnétique (mémoire rotative accessible par blocs).

Le filtre de la machine Opale est basé sur un opérateur spécialisé. Certains chercheurs considèrent que cette conception est dépassée, car des processeurs rapides permettront bientôt d'effectuer le même travail avec des performances suffisantes: nous avons déjà discuté de ce point. Pour nous, un circuit spécialisé peut, de toutes façons être plus efficace ou moins complexe qu'un processeur, mais cela ne signifie pas qu'il soit moins coûteux, si le processeur bénéficie d'un marché supérieur de plusieurs ordres de grandeur. Il y a donc là un pari technologique, qui est que les méthodes de CAO de VLSI permettront de réduire les coûts de conception dans des proportions telles qu'il sera possible à un circuit spécialisé produit en série limitée d'être plus économique qu'un processeur performant produit en grande série.

Nous n'avons pas (ou très peu) dans cette thèse considéré les possibilités offertes par les réseaux systoliques, ou les architectures à parallélisme massif. Sur le premier point, il est possible que des solutions intéressantes puissent être trouvées: c'est un aspect que nous serons amenés à envisager, et éventuellement évaluer dans l'avenir. Sur le second point, il reste beacoup à faire pour dégager un parallélisme suffisant dans l'exécution des programmes, et concevoir des architectures efficaces. C'est un domaine de recherche à part entière, que nous n'avons pas abordé jusqu'ici.

ANNEXE 1

LE LANGAGE PROLOG

1. INTRODUCTION:

L'objet de cette annexe (*) est de fournir une présentation de Prolog rapide, mais relativement complète, pour le lecteur qui ne connait pas déjà ce langage. On illustrera également, à l'aide d'un exemple, la démarche de la programmation en logique.

2. LE LANGAGE:

2.1. Principe:

La programmation classique est dite procédurale, ou impérative: toutes les étapes de l'algorithme permettant de résoudre un problème sont spécifiées. Avec la programmation en logique, par contre, on va chercher à spécifier le résultat attendu plutôt que la manière d'y parvenir.

Exemple:

Soit le problème: X appartient t'il à la liste L?

Dans un langage impératif, il faut:

* définir la structure de données représentant la liste,
* manipuler explicitement cette structure,
* parcourir la liste en cherchant la valeur X.

En Prolog, on part d'une définition du problème:

X appartient à la liste L si:

* Soit X est le premier élément de L,
* Soit X appartient au reste de la liste.

---

* Extrait d'un exposé donné le 22/3/85 aux journées micro informatiques de Grenoble (CUEFA)

Ce qui s'écrit:

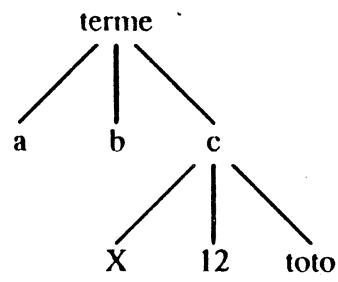
```
appartient(X, [X|L1]) :- c'est tout.
appartient(X, [Y|L1]) :- appartient(X, L1).
```

"C'est tout" est une primitive de Prolog sur laquelle nous reviendrons plus tard. La définition du problème et sa programmation en Prolog sont récursives, ce qui est une caractéristique fréquente de la programmation en logique.

2.2. Données manipulées:

Les données manipulées par Prolog sont des Un terme est une arborescence composée de données élémentaires (caractères, nombres, ...), de ou de variables. Un symbole est une donnée atomique dont la représentation externe est un identificateur.

Exemple:



Dans les programmes, les termes sont représentés sous forme parenthésée:

terme(a, b, c(X, 12, toto))

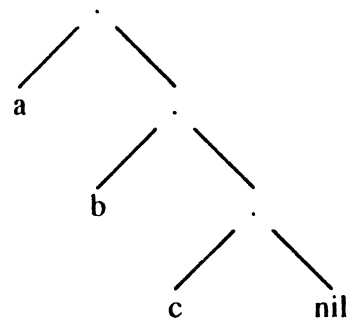
Ici, terme, toto, a et b sont des symboles. Terme et c, qui sont des noeuds de l'arborescence sont des symboles fonctionnels.

Il n'y a pas en Prolog de déclaration statique du type de donnée: une variable peut recevoir une donnée de n'importe quel type, y compris un autre terme. Certains traitements dépendront du type, d'autres devront vérifier le type de la donnée.

Certains interpréteurs Prolog permettent d'avoir des variables à la place de symboles fonctionnels: cela permet d'accéder facilement aux éléments du terme.

Les termes se prêtent à la représentation de nombreux types de données complexes.

Ainsi, une liste linéaire peut se ramener à un terme:



On peut la noter: .(a,.(b,.(c,nil)))
Ou encore: [a,b,c]

Ce type de représentation des listes sera fréquemment employé, comme nous le verrons dans ce document. La liste vide (c'est à dire la liste réduite au symbole nil) sera spécifiée par: [].

Si on considère par exemple un programme source écrit dans un langage quelconque, on pourra le représenter de plusieurs façons:

- sous forme d'une liste de caractères,
- sous forme d'une liste d'éléments syntaxiques (mots clés, identificateurs, symboles spéciaux, etc...): une telle représentation sera issue d'une analyse lexicographique.
- sous forme d'une arborescence syntaxique (décomposition du programme selon sa structure): c'est le résultat de l'analyse syntaxique.

2.3. L'unification:

L'unification est l'opération fondamentale de Prolog. Elle consiste à tenter de rendre deux termes égaux par substitution de leurs variables. Elle produit comme résultat les substitutions qui doivent être appliquées sur les variables pour identifier les termes.

Exemple:

> Soit l'unification des termes: terme(a(e, f), b(X, c(k, g))) et terme(Z, b(T, c(Y, g))) .
> Les substitutions produites sont: Z <- a(e,f), Y <- k, X <- T

On remarque dans cet exemple une substitution entre deux variables (X et T). Ces variables seront donc par la suite contraintes à prendre la même valeur. Toute substitution effectuée sur l'une affectera automatiquement l'autre.

2.4. Structure d'un programme:

Un programme Prolog est constitué de indiquant les différentes façons de vérifier des prédicats. Les clauses correspondant à un même prédicat constituent les d'un *paquet de clauses*. Une clause est à son tour composée de c'est à dire de termes qui représentent des prédicats. Le premier littéral est le *littéral de tête*: il représente le prédicat que la clause permet de vérifier. Les autres littéraux forment la queue de la clause.

Vérifier une clause revient donc à vérifier successivement tous les littéraux qui composent sa queue. Si la queue est vide, la clause est un *axiome*. L'exécution d'un programme Prolog va donc revenir à vérifier un prédicat de départ (littéral d'appel). Un prédicat est vérifié s'il existe une clause permettant de le vérifier.

Exemple:

On peut définir un grand père de la façon suivante: X est grand-père de Y s'il existe une personne Z telle que Z soit père de Y et X soit père de Z, ou Z soit mère de Y et X soit père de Z. En Prolog, cela peut s'écrire de la façon suivante:

```
grand_pere(X,Y) :- pere(Z,Y), pere(X,Z).
grand_pere(X,Y) :- mere(Z,Y), pere(X,Z).
pere(b, a).
pere(c, b).
pere(d, c).
pere(f, e).
mere(e, a).
```

Ce programme peut être appelé de diverses façons pour répondre à diverses question:

- Quels sont les couples grand-père/petit-fils: grand_pere(X,Y)
- Quels sont les grands-pères de a: grand_pere(X,a)
- Quels sont les petits fils de d: grand_pere(d, X)
- d est il grand-père de a: grand_pere(d, a)

Un programme Prolog peut donc être appelé de plusieurs façons, et retourner un ensemble de résultats (non déterminisme). Ses variables d'entrée et de sortie ne sont (en principe) pas définies.

2.5. Les variables:

Les variables d'une clause sont locales: elles ne sont connues qu'à l'intérieur de la clause où elles sont définies. Cependant, comme nous l'avons vu précédemment, des variables peuvent être liées entre elles lors des unifications, ce qui permet l'accès à des variables non locales.

Les variables ne sont affectées (substituées) qu'une seule fois (assignation unique). Une variable peut donc être ou (instanciée).

Dans nos exemples (écrits dans la syntaxe de C-prolog), une variable est dénotée par un identificateur dont les première lettre est une majuscule.

3. LA VERIFICATION

Vérifier un littéral, c'est chercher l'ensemble des valeurs de ses variables pour lesquelles il est vrai. L'interprétation d'un programme Prolog est donc une recherche dans un espace arborescent.

On part d'un littéral d'appel à vérifier. On doit alors:

- Rechercher une clause dont le littéral de tête unifie avec le littéral d'appel,
- Vérifier la clause, c'est à dire vérifier de gauche à droite les littéraux de la queue.

Le processus est récursif, les littéraux sont à leur tour traités de la même façon, jusqu'à ce qu'on trouve une clause n'entraînant pas de nouvelle vérification (axiome). Il y a succès de la démonstration si tous les littéraux peuvent être démontrés de cette façon, et échec dans le cas contraire.

En cas de succès, il y a avancement (passage au littéral suivant). Toutes les substitutions produites sont conservées et constituent l'environnement de la démonstration.

En cas d'échec, il y a retour arrière (backtracking). Un point de choix (alternative) est recherché, c'est à dire un littéral précédemment démontré et qui admet une autre solution. Les substitutions produites depuis la précédente démonstration de ce littéral sont effacées, et on essaie une nouvelle solution.

Lorsqu'une solution du littéral d'appel est trouvée, on provoque un échec pour passer à la solution suivante, de façon à produire l'ensemble des solutions.

Exemple:

Reprenons l'exemple précédent:

Littéral d'appel: grand_pere(X,a)? (quels sont les grands pères de a)

La première alternative entraîne la vérification des littéraux: :- pere(Z,a), pere(X,Z).

Le premier littéral unifie avec pere(b,a). Cette clause est un axiome. On retourne donc la substitution: Z=b. Le second littéral devient donc, par application de cette substitution: pere(X,b). Ce littéral unifie avec pere(c,b) (axiome), ce qui nous fournit la première solution: c.

Un échec est alors forcé pour passer aux autres solutions. Le retour sur pere(X,b) ne fournit aucune alternative, ni le retour sur pere(Z,a). On passe donc à la seconde alternative de grand_pere, ce qui donne:

:- mere(Z,a), pere(X,Z).

Le premier littéral fournit comme solution: Z=e, ce qui donne le littéral pere(X,e), et la solution X=f.

Il y a ensuite un nouveau retour arrière qui ne trouve pas d'autre solution, la démonstration est donc terminée.

Considérons maintenant la question suivante:

grand_pere(d,X)? (de qui d est il grand père)

La première alternative donne les littéraux: :- pere(Z,Y), pere(d,Z).

Ce qui donne les essais suivants:

Z=b, Y=a pere(d,b). échec, retour AR

Z=c, Y=b pere(d,c). 1ere solution = b
Z=d, Y=c pere(d,d). échec, retour AR
Z=f, Y=e pere(d,f). échec, retour AR

Le retour arrière ne trouve alors pas d'alternative sur père, et on passe à l'alternative suivante de grand-père:

:- mere(Z,Y), pere(d,Z).
Z=e, Y=a pere(d,e). échec, retour AR
... pas d'autre solution ...

Cette seconde requête entraîne un essai exhaustif de tous les pères et mères, ce qui n'est pas optimal: le programme, bien que fonctionnant dans n'importe quel sens favorise plutôt les questions du type "qui sont les grands pères". On pouvait écrire les clauses grand père dans l'autre sens si l'on voulait favoriser les questions du type "qui sont les petits fils":

grand_pere(X,Y) :- pere(X,Z), pere(Z,Y).
grand_pere(X,Y) :- pere(X,Z), mere(Z,Y).

Certains interpréteurs Prolog offrent des possibilités limitées pour permettre le choix d'un ordre optimal de vérification.

3.1. Prédicats évaluables:

On peut effectuer un certain nombre d'opérations avec les seuls mécanismes précédemment décrits. Cependant, ceux ci sont très insuffisants pour des applications réelles, tant au niveau des fonctionnalités qu'à celui des performances. On complète donc l'interpréteur par un certains nombre de prédicats prédéfinis, appelés prédicats évaluables.

Les prédicats évaluables sont le plus souvent spécifiques d'une implémentation de Prolog: de leur choix judicieux dépendra pour une bonne part la puissance de l'interpréteur.

Ils permettent:

-Les entrées sorties: au niveau caractère, sur les interpréteurs les plus rustiques, mais souvent aussi au niveau de la chaîne de caractère ou du terme.

-La manipulation de la base de clauses: possibilité d'ajouter ou de supprimer des clauses, ainsi que, souvent, possibilité de consulter la base de clauses.

-Le contrôle de l'interprétation: en particulier, suppression des choix: la coupure (!). Ce prédicat indique à l'interpréteur que l'on ne veut pas essayer d'alternative dans la démonstration de la clause en cours. C'est le "c'est tout" de notre premier exemple, où il sert à éviter de poursuivre le parcours de la liste après avoir trouvé l'élément cherché. La coupure peut donc servir à éviter des démonstrations inutiles. Nous en verrons d'autres utilisations dans un prochain exemple.

-L'arithmétique: elle peut être limitée aux quatre opérations sur des entiers, ou traiter des réels, ou encore porter sur l'évaluation d'une expression arithmétique présentée sous forme de terme.

-La mise au point (traces, points d'arrêt, etc...).

-Et bien d'autres choses encore...

On peut citer en outre deux prédicats remarquables (Prolog II):

-diff(X,Y) : contraint les variables à rester différentes. Une unification qui affecte ensuite X ou Y ne peut réussir que si elle respecte la contrainte: si X et Y sont liées à une même valeur, l'unification échoue.

-geler(X,p(X)) : le prédicat p(X) n'est vérifié que lorsque la variable X est liée. Si lors de l'appel de geler, la variable X est liée, le prédicat p(X) est vérifié. Sinon, il est mis en attente jusqu'à ce que X soit liée.

D'une façon générale, on peut considérer qu'un interpréteur Prolog représente environ une dizaine de pages de programme en langage évolué; par contre, l'ensemble des prédicats évaluables peut en représenter une centaine. La réalisation d'un interpréteur Prolog est donc un exercice de programmation assez simple, mais par contre, en faire un système Prolog complet et opérationnel représente un travail beaucoup plus important.

4. PROGRAMMER EN PROLOG:

La programmation en Prolog est une démarche très différente de la programmation dans les langages classiques. En fait un programme Prolog peut être vu de deux façons (sémantique):

-déclarative, c'est à dire qu'on ne s'intéresse qu'à la signification des clauses,

-procédurale, c'est à dire qu'on fait intervenir la façon dont le programme va être interprété.

Dans une grande partie des cas, la sémantique déclarative est suffisante, c'est à dire qu'on n'a pas besoin de se préoccuper de la façon dont le programme se comportera réellement.

Nous allons dans la suite tenter d'expliciter la démarche de la programmation en logique en quatre préceptes et un exemple.

4.1. Les préceptes:

Soit un problème...

-1er précepte: oublier sa culture informatique. La personne qui aborde la programmation en logique avec une bonne connaissance préalable d'autres langages de programmation tente en général de se raccrocher aux notions connues. C'est en fait une perte de temps, et cela amène à une programmation particulièrement inefficace.

-2ème précepte: trouver une représentation convenable des éléments du problème. La notion de terme est suffisamment puissante, et une bonne représentation du problème est souvent une étape importante vers sa solution.

-3ème précepte: Réfléchir sur un problème, c'est déjà le programmer. En effet, Prolog fournit les outils de formalisation et d'expression des problèmes. Bien souvent, la meilleure façon de réfléchir sur un problème consiste à le programmer directement en Prolog.

-4ème précepte: Si on ne sait pas résoudre un problème, il faut le couper en morceaux. La démarche de base de Prolog consiste en effet à décomposer un problème en sous problèmes, jusqu'à ce que ces sous problèmes soient suffisamment simples pour être résolus directement à l'aide d'axiomes ou de prédicats évaluables. Nous avons vu un exemple de cette démarche avec la recherche d'un élément dans une liste: on ne sait accéder qu'au premier élément de la liste, donc on réduit successivement la liste jusqu'à avoir trouvé l'élément cherché (succès), ou avoir épuisé la liste (échec).

4.2. exemple:

Prenons comme exemple le classique problème des huit reines: il s'agit de placer huit reines sur un échiquier de façon à ce qu'elles ne puissent pas se prendre entre elles. Ceci implique que l'on n'aie qu'une reine et une seule par ligne et par colonne, et qu'on ne trouve pas deux reines sur une même diagonale. On peut donc représenter l'échiquier sous forme d'une liste de huit chiffres de 1 à 8: chaque élément de la liste représente par exemple une colonne, sa valeur étant le numéro de la ligne où se trouve la reine. Le problème revient alors à réarranger une liste de 8 nombre, puis à imprimer le résultat sous une forme agréable.

On arrive donc à la clause suivante:

reines :- placer([1,2,3,4,5,6,7,8],[],X), sortir(X).

Le placement peut se faire en prenant les éléments un par un dans la liste originale et en vérifiant qu'ils soient bien placés par rapport aux éléments déjà placés, c'est à dire qu'ils remplissent la seconde condition (ne pas être sur la même diagonale qu'une reine déjà placée).

La clause placer va donc travailler sur trois listes: une liste d'éléments à placer, une liste d'éléments déjà placés, et une troisième liste permettant le retour du résultat final. Lorsque la première liste est vide, le résultat peut être transféré dans la troisième, et on a terminé. L'utilisation de la coupure (!) permet d'éviter un nouvel essai de la seconde alternative, essai qui échouerait, puisqu'il n'y a plus d'élément à prendre: c'est donc une simple optimisation. Sinon, si la première liste n'est pas vide, il faut:

-prendre un élément dans la première liste, en retournant une nouvelle liste d'éléments restant à placer,
-vérifier que l'élément pris soit bien placé,
-itérer le processus par récursivité pour placer le reste de la liste.

```
placer([],X,X) :- !.
placer(Non_places,Places,Resul) :-
    prendre(Elem,Non_places,Non_place2),
    bien_place(Elem,Places,1),
    placer(Non_place2,Elem.Places,Resul).
```

"prendre" rappelle dans son principe notre premier exemple: on prend soit le premier élément de la liste, soit un autre élément. La différence est que le processus est non déterministe, et qu'au retour arrière il va aller chercher un autre élément. Il retourne l'élément pris et la liste des éléments restant.

```
prendre(Element,[Element|Liste],Liste).
prendre(Element,[X|Liste],[X|Liste2]) :-
    prendre(Element,Liste,Liste2).
```

Vérifier qu'un élément soit bien placé, c'est vérifier qu'il ne soit pas pris par un élément déjà placé. Il ne doit pas être sur une même diagonale, c'est à dire que la différence des positions ne doit pas être égale à la différence en valeur absolue des valeurs. On va procéder par vérification successive des éléments déjà placés (récursivité). "bien_placé" va donc avoir trois arguments: l'élément à tester, la liste d'éléments à vérifier, et la différence de position entre l'élément à tester et le premier élément de la liste. On rencontre ici un nouvel exemple de coupure dans un rôle d'optimisation (une seule des deux unifications peut normalement réussir). On trouve le prédicat arithmétique is, qui évalue l'expression représentée par son deuxième argument et unifie le résultat avec son premier argument. Ce prédicat s'emploie sous forme infixée (X is Y, et non is(X,Y)).

```
bien_place(Element,[Tete|Liste],Dif) :- !,
    pas_pris(Element,Tete,Dif),
    Dif2 is Dif+1,
    bien_place(Element,Liste,Dif2).
bien_place(Element,[],X).
```

"pas_pris" calcule la valeur absolue de la différence des éléments, et vérifie qu'elle soit différente de la différence des positions. On utilise deux clauses, car l'interpréteur utilisé ne sait pas évaluer une valeur absolue. On trouve le prédicat coupure a ici un rôle particulier, qui est de réaliser l'équivalent d'un "si ... alors ... sinon...". Les deux clauses ont le même littéral de tête, et le choix n'est effectué que par le '<'. Dif teste la différence de ses arguments.

```
pas_pris(X,Y,Z) :-
    X < Y, !, U is Y-X, dif(U,Z).
pas_pris(X,Y,Z) :-
    U is X-Y, dif(U,Z).
```

Si l'interpréteur utilisé ne sait pas tester l'inégalité, il est possible de le programmer. Si les deux arguments unifient entre eux, c'est qu'ils ne sont pas différents: on force alors un échec, et la coupure empêche d'essayer la seconde alternative. Sinon, la seconde alternative est essayée, et retourne un succès.

```
dif(X,X) :- !, fail.
dif(X,Y).
```

On va maintenant afficher le résultat en dessinant l'échiquier sous forme d'une grille où chaque reine est symbolisée par la lettre R. On considère que la liste représente des lignes et des numéros de colonne, et on va donc travailler ligne par ligne, ce qui est plus pratique(de toutes façons, le problème est symétrique dans tous les sens). Pour chaque ligne, on va donc afficher un trait de séparation, et les cases. A la fin, on dessine le bord inférieur de l'échiquier. Write est un prédicat évaluable qui écrit un terme. "nl" passe à la ligne suivante.

```
sortir([]) :- !,
    write('+---+---+---+---+---+---+---+---+'),nl.
sortir([X,Y]) :-
    write('+---+---+---+---+---+---+---+---+'),nl,
    ecrili(X,1),sortir(Y).
```

On écrit les lignes case par case; lorsque le numéro de la case est égal à la position de la reine sur la ligne, on affiche un R. Après la huitième case, on dessine le bord de l'échiquier.

```
ecrili(X,X) :- !,
    write('! R '), Y is X+1, ecrili(X,Y).
ecrili(X,9) :- write('!'), nl, !.
ecrili(X,Y) :-
    write('!   '), Z is Y+1, ecrili(X,Z).
```

5. CONCLUSIONS:

Dans l'exemple précédent, nous espérons avoir montré quelques uns des points forts de la programmation en logique: manipulation automatique de la structure de données, programmation découlant directement de l'analyse du problème... Dans la plupart des cas, le programme est écrit sans même se préoccuper de la façon dont il est exécuté.

Certains cas demandent une compréhension un peu plus approfondie de l'interprétation, mais ils se ramènent le plus souvent à quelques "recettes": utilisation de la coupure, récursivité, etc... Avec une expérience, même réduite, du langage, certains types de traitement, tels que ceux qui impliquent des parcours de listes, deviennent quasiment immédiats.

ANNEXE 2

INTERPRETATION DE PROLOG

Il existe diverses façons de présenter l'interprétation de Prolog, de même qu'il existe diverses solutions pour l'implémentation. Nous présenterons ici l'algorithme le plus classique, employé dans Prolog I, dans l'interpréteur DEC 10 de Warren, et vraisemblablement dans nombre d'autres interpréteurs (dont bien sûr Prolog I+ et Yaap). Notre présentation s'appuyera en grande partie sur l'interpréteur Yaap, que nous avons développé. Une description succinte de cet interpréteur figure en annexe 3.

D'une façon générale, cette interprétation est un processus de recherche dans un espace arborescent composé des diverses alternatives à essayer pour arriver aux solutions. Cette recherche s'effectue d'une façon bien précise, correspondant à la définition même de Prolog (de gauche à droite et en profondeur d'abord).

Un interpréteur Prolog est basé généralement sur un ensemble d'espaces mémoire permettant de stocker les différents composants du programme, son environnement à un instant donné (contexte), et les informations nécessaires pour permettre le backtracking.

1. LE DICTIONNAIRE:

Le dictionnaire a plusieurs rôles:

- Il assure la correspondance entre la forme externe des symboles (identificateur) et leur forme interne (code). Cette forme interne n'est autre, généralement, que l'index, ou l'adresse du descripteur du symbole dans le dictionnaire. Le dictionnaire va donc comporter l'identificateur du symbole et pourra être accédé par hash coding à partir de cet identificateur.
- Il fournit un "modèle" du terme. Ce modèle va d'une part indiquer le nombre de fils, et d'autre part fournir un "squelette" de terme qui sera utilisé par certains prédicats évaluables pour synthétiser des termes.
- Il fournit une voie d'accès aux paquets de clauses: le symbole fonctionnel d'un prédicat permet, par l'intermédiaire du dictionnaire, d'obtenir l'adresse du paquet de clauses correspondant. Si ce symbole correspond à un prédicat évaluable, on trouve le code du prédicat, qui permet l'appel de la procédure de traitement correspondante. Enfin, dans Yaap, le même champs du dictionnaire permet également d'affecter une valeur à un symbole (notion de variable globale, similaire à celle que l'on trouve dans Prolog II). La distinction entre ces possibilités est faite selon la valeur d'un mot de 4 octets: positif, il s'agit d'un pointeur vers un paquet de clauses, négatif, il s'agit soit d'une variable globale, soit d'un prédicat évaluable, selon la valeur du premier octet. Ce type de codage suppose évidemment que les pointeurs restent codés sous forme

d'adresses physiques, et que l'espace d'adressage est inférieur à deux milliards d'octets. Cela ne semble pas très limitatif dans l'immédiat.

2. LE CODAGE DES CLAUSES:

Les clauses sont codées sous forme de listes chaînées d'alternatives. Les alternatives peuvent être codées sous forme de listes de littéraux. Chaque alternative va en outre comporter son nombre de variables.

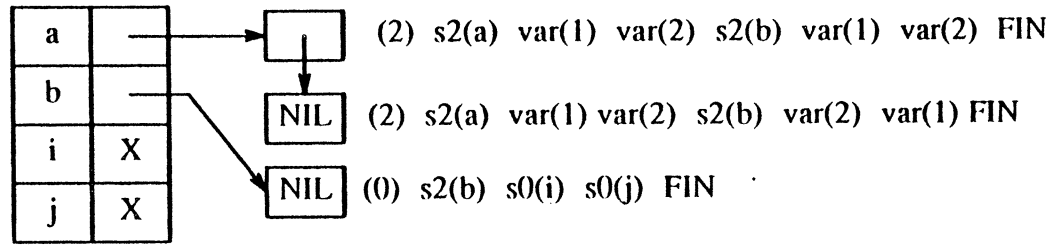
Dans Yaap, nous avons adopté un codage des données sur 4 octets, comportant un octet de type: en effet, le type des données n'est pas prédéterminé en Prolog, et il est donc nécessaire de l'indiquer explicitement dans la donnée elle même. Un type positif indique un symbole, sa valeur correspondant au nombre de fils (donc compris entre 0 et 127). Les trois octets restant sont alors l'index de l'entrée correspondante dans le dictionnaire. Un type négatif indique une variable, ou une donnée élémentaire (entier, caractère, ou chaîne).

Exemple:

Soit le programme:

```
a(X, Y) :- b(X, Y).
a(X, Y) :- b(Y, X).
b(i, j).
```

Ce programme est codé de la façon suivante:



dictionnaire clauses

3. LES VARIABLES:

Le stockage des variables représente un point clé de l'interprétation de Prolog. Le problème réside en effet d'une part dans le stockage d'objets de structure complexe pouvant être partiellement définis (les termes), et d'autre part dans la nécessité de pouvoir revenir en arrière, et donc de "défaire" très rapidement les substitutions effectuées (backtracking).

Lors de l'appel d'une clause, un espace correspondant aux variables de la clause est alloué sur une pile, d'une façon un peu similaire à un appel de procédure: on parle

d'instanciation de la variable, la pile s'appelant pile des instances. Ceci constitue ce que l'on appelle l'environnement de la clause. La substitution de données simples ne pose guère de problèmes. Les choses se compliquent lorsque l'on substitue à une variable un terme partiellement défini: ce terme va en effet comporter des variables qui pourront à leur tour être substituées dans la suite. De la même façon, un terme substitué à une variable peut être lui même déjà "construit" par l'intermédiaire de substitutions sur des variables.

La majeure partie des interpréteurs Prolog fonctionnent selon le principe du "partage de structure" [BOY 72]. Celui ci consiste a représenter une substitution entre une variable et un terme sous forme d'un couple de pointeurs: un pointeur vers un "squelette" du terme, c'est à dire la structure de terme fournie par le programme, et un pointeur vers un environnement, où sont (ou pourront être) définies les variables contenues dans le terme.

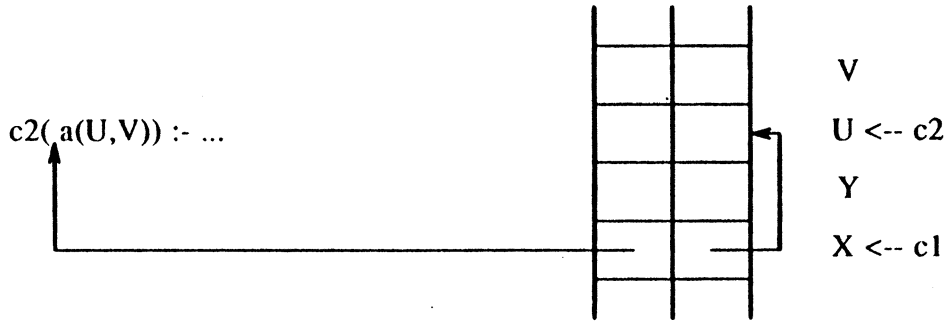
Considérons l'exemple suivant:

```
c1(X,Y) :- c2(X), c3(Y).
c2(a(U,V)) :- ... .
  ...
```

L'environnement de la clause c1 va comporter deux emplacements pour les variables X et Y: ces variables seront en fait représentées par leur index dans l'environnement (déplacement par rapport à la base de l'environnement dans la pile des instances). Lors de l'appel de la clause c2, un environnement va être affecté à la vérification de cette clause, comportant deux emplacements pour les variables U et V. Lors de l'unification de la cible c2(X) avec l'en-tête c2(a(U,V)), la variable X de la cible (supposée libre) va être substituée avec le terme a(U,V) de l'en-tête. Ceci se fait en plaçant un pointeur sur a(U,V) (dans le programme), et un pointeur sur l'environnement de la clause c2.

On obtient alors la situation représentée sur le schéma suivant:

c1(X,Y) :- c2(X), c3(Y).



De cette façon, toute substitution effectuée ultérieurement sur U et V sera automatiquement reportée sur le terme substitué à X.

Dans l'accès à un terme, il va donc falloir, pour chaque variable rencontrée, aller voir dans l'environnement correspondant si elle n'est pas liée à quelque chose. Comme plusieurs variables peuvent être liées entre elles, on peut avoir ainsi une chaîne de variables (le plus souvent très courte), qu'il va falloir parcourir pour trouver la valeur associée à la variable: c'est la primitive "descente" de Prolog I, appelée "déréférence" par Warren. Un terme est donc défini par l'intermédiaire des environnements des clauses vérifiées: les liaisons effectuées sur U et V resteront utiles tant que la liaison sur X restera elle même utile. Contrairement à un appel de procédure classique, on ne peut donc pas dépiler un environnement au retour, mais seulement lors du backtracking.

Par ailleurs, lors du backtracking, le dépilement des environnements ne suffit pas à effacer les substitutions: ainsi, lors du backtracking sur c2, l'effacement de l'environnement de c2 ne suffit pas à annuler la substitution sur X. De plus, par le biais de substitutions entre variables, la substitutions sur X peut avoir été enregistrée au niveau d'une variable de la clause qui a appelé c1. Il faut donc mémoriser de telles substitutions pour pouvoir les défaire ensuite: c'est la rôle de la trace, qui est gérée en pile, et qui permet de savoir, au backtracking, quelles variables non locales ont été affectées, pour pouvoir les remettre à l'état libre.

Ce schéma est susceptible d'améliorations. Ainsi, Warren distingue, au niveau de l'environnement les variables locales des variables globales: les variables locales sont celles qui n'apparaîssent que comme arguments au premier niveau des littéraux de la clause. Dans notre exemple précédent, U et V sont des variables globales, car elles apparaîssent au niveau d'un sous terme du littéral de tête. Par contre, X et Y sont des variables locales, car elles apparaîssent uniquement comme arguments au premier niveau des littéraux.

Les variables locales ne font en fait qu'établir un lien entre différents littéraux d'une clause, alors que les variables globales rentrent dans la construction des termes. Les variables locales deviennent donc inutiles dès que la clause où elles sont définies se termine de manière déterministe, c'est à dire sans autre solution possible (point de choix). Les variables globales restent par contre utiles jusqu'au backtracking, puisqu'elles continuent de porter des éléments du résultat en cours d'évaluation. Dans le compilateur de Warren, les variables locales et globales sont stockées dans des piles séparées, la pile locale étant dépilée lors des retours déterministes des clauses.

Il existe également des techniques alternatives basées sur la recopie des modèles de termes [MEL 82, BRU 82]: cela permet un accès direct aux éléments des termes (sans passer par des chaînes de références), au prix d'un encombrement mémoire supérieur et d'une substitution plus complexe. Toutefois, ces inconvénients semblent équilibrer les avantages, et cette méthode peut être plus performante sur certaines machines.

4. PILE D'EVALUATION:

L'évaluation va nécessiter une pile. Cette pile est empilée à l'avancement, et dépilée au backtracking: elle va donc contenir un historique de toute l'interprétation, de façon à permettre le backtracking.

Ses éléments sont les suivants:

- un pointeur sur le but en cours de vérification,
- un pointeur sur l'alternative en cours de vérification: ce pointeur sera utilisé au bacttracking pour la recherche d'une autre alternative. Une autre solution consiste donc à stocker directement l'adresse éventuelle de l'alternative suivante, ce qui peut accélérer le backtracking.
- un pointeur dans la pile des instances, sur la base de l'environnement de la cible vérifiée,
- un pointeur sur le sommet de la trace au moment de l'appel. Ce pointeur est utilisé au backtracking pour déterminer jusqu'à quel point les substitutions enregistrées dans la trace doivent être défaites.
- un pointeur sur l'entrée de pile correspondant au point d'appel de la cible en cours de vérification.

5. L'INTERPRETATION:

L'interprétation va partir d'un but de départ (littéral d'appel). L'algorithme est décrit ci après.

```
fini = faux;
tant que non fini faire
debut
    accéder au dictionnaire à partir du symbole fonctionnel du but;

    Si le but correspond à un prédicat évaluable, alors
    debut
        appeler la procédure de traitement correspondante;
        Si échec du prédicat évaluable alors aller à backtracking;
    fin;

    Sinon debut

        obtenir l'adresse de la première clause correspondant au but;
        (* dans le dictionnaire *)
appel:
        allouer les variables sur la pile des instances;
        unifier le but avec l'en tête de l'alternative;
        si échec de l'unification, alors aller à backtracking;
```

```
        si l'alternative est un axiome alors
        debut
            si le but courant a un successeur, alors celui ci devient le nouveau but courant;
            sinon
            debut    (* la clause est démontrée  *)
                rechercher dans la pile un but à démontrer;
                S'il existe un tel but, il devient le nouveau but courant;
                sinon, aller à backtracking; (*  chercher les autres solutions  *)
            fin;
            préparer une nouvelle entrée de pile pour la suite de la démonstration;
        fin;

        (* l'alternative n'est pas un axiome *)
        (* on passe à la démonstration de l'alternative *)
        le premier littéral de la queue de l'alternative devient le nouveau but courant;
        initialiser une nouvelle entrée de la pile;
        aller à finresol;

backtracking:

        (* chercher un point de choix *)
        remonter la pile jusqu'à trouver une alternative qui a un successeur;
        S'il n'en existe pas alors fini := vrai;
        sinon debut
            (* effacer les substitutions jusqu'au point de choix *)
             dépiler les instances ;
             remettre à l'état libre les variables enregistrées dans la trace;
             le but appelant de l'alternative devient le nouveau but;
             dépiler la pile d'évaluation jusqu'au nouveau but;
             la nouvelle alternative à vérifier devient le successeur de
             l'alternative précédente;
             aller à appel ;
        fin;
    fin;
finresol:
fin; (* fin du tant que  *)
```

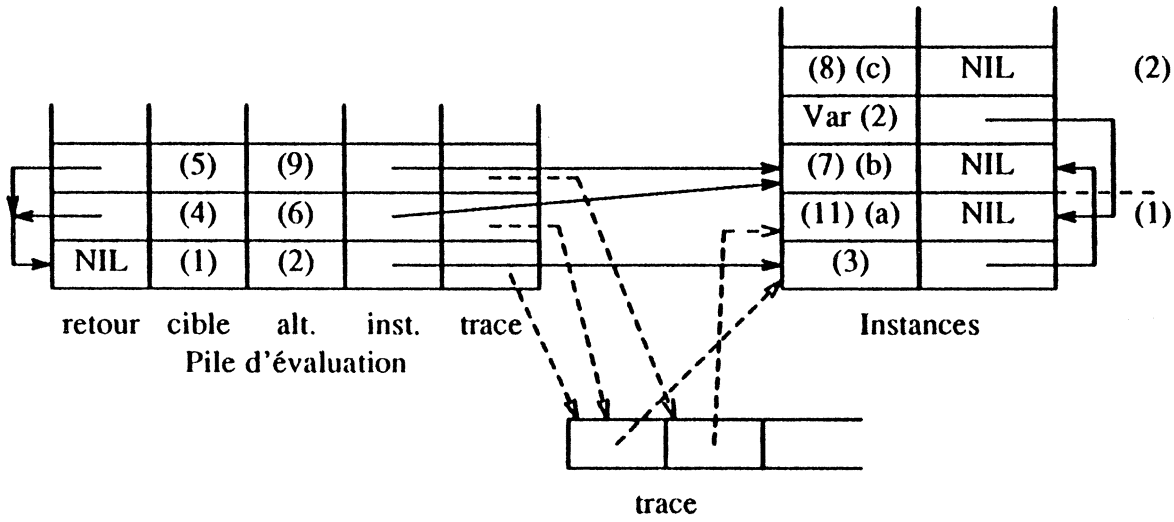
6. EXEMPLE:

Il est difficile d'illustrer clairement les mécanismes d'interprétation de Prolog: on arrive rapidement à une prolifération de pointeurs rendant tout schéma illisible et toute explication confuse...

Nous allons tout de même tenter cette gageure en restant au niveau d'un exemple simple. Nous représenterons le programme sous sa forme externe, les littéraux étant

numérotés en indice. Les chaînages vers le programme sont représentés sous forme d'indices entre parenthèses renvoyant aux numéros des littéraux.

:- $q_1(X, Y)$.
$q_2(f_3(a, X), Y)$:- $c1_4(X, Z)$, $c2_5(Z, Y)$.
$c1_6(b_7, c_8)$.
$c2_9(c_{10}, a_{11})$.



Au premier niveau de la pile, on trouve l'état de l'interprétation lors de l'activation de la clause d'appel. Les variables X et Y sont allouées aux deux premiers niveaux de la pile des instances.

L'appel du but (1) entraîne la vérification de la clause (2). L'espace nécessaire aux variables est alloué, et l'unification est effectuée. L'unification du but (1) avec l'en tête de clause (2) réussit: elle produit alors une substitution entre la variable X de (1) et le squelette de terme (3), et une substitution entre les deux variables Y. La substitution sur X est non locale, elle est donc mémorisée dans la trace.

Le but (4) est vérifié. Ceci entraîne une unification, mais pas d'allocation d'instance, car l'alternative (6) n'a pas de variable. La vérification est mémorisée au second niveau de la pile. Les deux substitutions sont locales.

L'alternative (6) étant un axiome, on passe à la vérification du but (5). L'unification entre ce but et l'en tête (9) produit une substitution sur *y, répercutée sur la variable de la clause d'appel, grâce à la liaison entre les deux variables: il s'agit donc encore d'une substitution non locale qui doit être mémorisée dans la trace.

L'alternative (9) étant un axiome, et le but (5) étant le dernier de la clause, la démonstration est terminée. On revient à la clause (1) qui est également terminée. Les variables X et Y se trouvent liées aux solutions f(a,b) et a. Il faut remarquer que la première liaison passe par l'intermédiaire de la variable X de la clause (2).

Nous n'avons pas fait figurer dans cet exemple, volontairement simplifié, de cas d'échec ou d'alternative. Il est facile de vérifier que, si un tel cas se produit, la pile

contient tous les éléments nécessaires au retour arrière et au choix d'une nouvelle solution.

7. COMPILATION:

Ce qui précède concerne l'interprétation de Prolog. Il n'existe encore, à notre connaissance que peu de compilateurs Prolog, le plus connu étant celui de Warren.

La compilation de Prolog va consister, pour chaque en-tête de clause à générer des séquences d'instructions effectuant l'unification avec les buts. Il s'agit donc d'unifications "sur mesure", contrairement à l'interprétation où on utilise un algorithme général. En particulier, on n'aura pas à tester le type des éléments de l'en-tête de clause, puisque celui-ci est connu à la compilation: pour une constante atomique, par exemple, selon le type de l'élément correspondant du but, on peut générer directement une comparaison ou une substitution.

Pour le reste, Prolog compilé va utiliser les mêmes mécanismes de pile que Prolog interprété.

Le Prolog de Warren donne en outre la possibilité de déclarer des modes: il s'agit d'indiquer au compilateur si une variable est une entrée, une sortie, ou si elle peut être les deux. Les déclarations de mode permettent de simplifier le code généré, et de gagner du temps d'exécution et de la place en mémoire. On peut leur reprocher d'augmenter le travail de l'utilisateur, et de lui imposer une préoccupation de performances que Prolog devrait supprimer. Il s'agit cependant là d'optimisations qui peuvent être réalisées une fois que le programme tourne. Par ailleurs, il est sans doute envisageable de faire exécuter ce travail par un programme (écrit en Prolog, bien sûr), qui utiliserait des déclarations de mode uniquement au niveau de la clause d'appel, plus les possibilités de déduction que permettent certains prédicats évaluables.

8. OPTIMISATIONS:

Aux mécanismes de base précédemment décrits peuvent s'ajouter quelques techniques d'optimisation. Nous avons déjà vu la possibilité de classer les variables en variables locales et variables globales.

Le backtracking sélectif est une idée présentée par L.M Pereira et A. Porto [PEA 82]. Il consiste à sélectionner l'alternative sur laquelle va être effectué le retour arrière, plutôt que d'essayer systématiquement toutes les alternatives. Ceci doit permettre d'éviter d'essayer des alternatives qui ne modifient pas la cause de l'échec.

Exemple:

Considérons la clause suivante: c1(X,Y) :- c2(Z), c3(X), c4(Z,Y).

Dans cet exemple, le retour sur la clause c3 en cas d'échec de c4 est inutile, car c3 ne modifie pas les conditions d'appel de c4: une nouvelle solution de c3 conduirait donc au même échec de c4, et il est préférable de backtracker directement sur c2.

Un deuxième type d'optimisation est la réduction des récursions en queue. En effet, la récursivité est très souvent utilisée pour simuler une itération, mais son coût est nettement plus élevé, en raison en particulier de l'encombrement des piles. Un certain nombre de travaux ont donc été effectués pour permettre, dans certains cas, de transformer des appels récursifs en itération, lorsqu'ils sont effectués par le dernier littéral de la clause (appels terminaux) [BRU 82].

Dans le cas où un paquet de clauses comporte de nombreuses alternatives, il est peu efficace de tenter d'unifier chacun des en têtes de clause avec le but. Il est alors intéressant de disposer d'un mécanisme d'indexation permettant d'accéder directement aux alternatives susceptible d'unifier. Un tel mécanisme d'indexation est proposé par Warren dans son compilateur. L'indexation est effectué sur le symbole fonctionnel du premier argument du but.

Dans C-Prolog, l'unification est en grande partie effectuée par un segment de programme placé dans le corps de l'interpréteur. Ceci permet d'avoir une unification "en ligne", adaptée à l'unification d'un but avec un en-tête de clause. Une procédure générale d'unification n'est appelée que lorsque des variables liées impliquent de poursuivre l'unification dans un autre environnement.

ANNEXE 3

OUTILS EXPERIMENTAUX:

L'étude de l'interprétation de Prolog doit s'appuyer sur un interpréteur permettant, non seulement l'utilisation du langage, mais aussi de faire des mesures. Il était donc important de disposer du source. A cette fin, nous disposons dans le projet Opale de deux outils, Prolog I+, et Yaap, que nous allons décrire dans cette annexe.

Par ailleurs, nous présenterons également brièvement la SM 90, support matériel de l'étude, et qui devrait pouvoir supporter l'implémentation matérielle d'Opale, et le système UNIX, qui, avec quelques uns des outils logiciels qu'il comporte constitue un moyen expérimental puissant.

1. Prolog I+:

Prolog I+ est un dérivé du premier interpréteur Prolog, écrit en Fortran. Cet interpréteur a été dans un premier temps traduit en Pascal UCSD sur Pascaline (128 K), et structuré, ce qui nous a permis d'en comprendre le fonctionnement. En effet, à l'époque, il n'existait pas de documentation claire sur le fonctionnement d'un interpréteur Prolog: la seule solution consistait donc à en prendre un et à voir comment il est fait.

Cet interpréteur a ensuite été traduit en C et porté sous Unix (Micromega, puis SM 90), où il a subi une série d'améliorations concernant les performances, les fonctionnalités et la capacité (taille des programmes que l'on peut traiter). Signalons au passage que les diverses traductions ont été faites d'une manière très systématique, et n'ont représenté qu'un travail relativement réduit (un peu moins de 15 jours à deux personnes pour le passage de Fortran à Pascal, sans connaissance préalable du programme). Le passage de Pascal a C était dû au fait que nous ne disposions pas, à l'époque, d'un compilateur Pascal satisfaisant sur le Micromega. Nous pouvons cependant estimer le gain de performances à environ deux, connaissant le rapport de performances des machines elles mêmes.

L'interpréteur opère sur des mots de 16 bits, ce qui permet d'obtenir des performances satisfaisantes. Les différents espaces de travail (clauses, dictionnaire, pile d'évaluation, trace et pile des instances) ont été séparés en trois tableaux, au lieu de deux initialement. Ceux ci comprennent: les clauses + le dictionnaire, la pile + la trace, et les instances. Ceci permet donc une taille maximale de 32 K pour chaque tableau, ce qui représente une capacité suffisante pour traiter des problèmes moyens. Certains archaïsmes de Prolog I ont été supprimés (simplification du codage interne de clauses), ce qui a permis des gains de place et de performances.

Par contre certaines lourdeurs sont restées: codage du dictionnaire dans la table des clauses, codage des caractères, représentation des chaînes de caractères sous forme de listes. Le système est toujours basé sur un superviseur écrit en Prolog, cependant, l'itération du superviseur est assurée par l'interpréteur: elle n'encombre donc pas la pile d'évaluation.

La syntaxe a été améliorée: suppression des fameux + et - de Prolog I, utilisation d'une flèche pour séparer l'en tête des clauses. Le traitement des erreurs a été amélioré: en cas d'erreur d'exécution, le prédicat erroné est imprimé (et non pas seulement son nom), et il y a retour au superviseur avec le cas échéant commutation de l'entrée sur le terminal. L'appel d'un prédicat indéfini provoque une erreur, mais par contre, il existe un prédicat echec qui permet de forcer un retour à faux. Contrairement à Prolog I qui ne connaissait que les lettre majuscules, Prolog I+ connaît l'ensemble des caractères ASCII imprimables.

Certains prédicats évaluables ont été améliorés, d'autres ont été ajoutés:

- possibilité de commuter explicitement l'entrée ou la sortie sur le terminal ou un fichier,
- Le prédicat évaluable ecrit (*x) permet d'écrire un caractère, une chaîne ou un nombre,
- possibilité de lecture d'une chaîne,
- Le prédicat ouvrir permet d'ouvrir (ou de réouvrir) le fichier d'entrée ou de sortie,
- Le prédicat finligne permet de tester si l'entrée courante est en fin de ligne.
- Possibilité limitée de manipuler des variables globales (non affectées par le backtracking), accessibles par l'intermédiaire de symboles. Cette possibilité permet en particulier de faire des comptages, des cumuls, etc...
- Possibilité de tester l'infériorité de caractères ou d'entiers,
- Possibilité d'appel de commandes Unix.

Prolog I+ est certainement encore loin des fonctionnalités des bons systèmes Prolog actuels, mais on pense tout de même en avoir fait un outil nettement plus agréable, et aux possibilités très améliorées.

Dans cette version, en configuration maximale, Prolog I+ demande environ 230 K octets de mémoire primaire, dont 30 K de programme. Le programme représente 1872 lignes de programme en C, dont 1078 pour les prédicats évaluables, le reste se partageant entre les déclarations (137 lignes), divers sous programmes, l'unification, le corps de l'interpréteur, et le programme principal. Le noyau de l'interpréteur (corps de l'interpréteur + unification + quelques procédures de base) occupe au maximum une dizaine de pages de programme. Les performances sont estimées à un peu plus de 500 inférences par secondes (sur SM 90, avec un processeur 68000 10 MHz).

2. Yaap:

Malgré toutes les améliorations qu'il a subi, Prolog I+ n'était pas un outil suffisant pour nos travaux. Son manque de souplesse empêchait, en particulier de pousser plus loin les améliorations, et il était peu adapté à l'étude des bases de données. Par ailleurs, le groupe "Opale" ressentait le besoin d'aborder la conception d'un interpréteur complet, c'est à dire d'affronter les choix, plutôt que de subir des choix déjà fait. Enfin, l'étude de Prolog nécessitait de pouvoir évaluer d'autres solutions que celles adoptées dans Prolog I.

Ceci nous a donc amenés à concevoir et réaliser Yaap (Yet Another Avatar of Prolog). Comme son nom l'indique peut être, Yaap se veut avant tout un outil expérimental, et n'a pas la prétention de concurrencer les interpréteurs Prolog déjà existants. Ses objectifs sont les suivants:

- Fournir une plateforme expérimentale pour interfacer avec la base de données d'Opale,
- Permettre de rechercher et évaluer les éléments de la performance dans l'interprétation de Prolog,
- Permettre d'expérimenter diverses solutions dans l'interprétation de Prolog.

L'algorithme d'interprétation de Yaap est classique, et assez proche de celui de Prolog I+. Nous avons cependant cherché à avoir des structures de données plus claires, même si c'est au détriment de l'encombrement mémoire: le dictionnaire, les instances, les clauses, la pile d'évaluation et la trace sont donc stockés dans des tableaux séparés.

Yaap a été programmé de façon modulaire, pour faciliter son évolution. On a également cherché à simplifier l'insertion de nouveaux prédicats évaluables: il suffit pour cela de modifier et de recompiler une procédure C comprenant un tableau des noms et des nombre d'arguments des prédicats évaluables, et un switch vers les procédures correspondant à chaque prédicat. On considère qu'un tel mécanisme pourrait d'ailleurs être laissé à la disposition de l'utilisateur pour lui permettre d'enrichir ou d'adapter son interpréteur: il n'a en fait (outre écrire ses prédicats évaluables), qu'à modifier et compiler cette procédure, puis refaire l'édition de liens de l'interpréteur. On peut également prévoir un ensemble de procédures permettant à un utilisateur de récupérer des paramètres et retourner des résultats sans avoir à connaître de façon précise le codage des données dans l'interpréteur.

Nous n'avons pas jusqu'ici cherché à implémenter des techniques telles que backtracking intelligent, réduction des récursivités en queue de clause, séparation pile locale - pile globale. Nous nous somme plutôt attachés à rechercher les éléments de la performance au niveau du codage des données.

Les données sont codées sur 4 octets comportant un octet de données et un octet de type. Pour les symboles, le type est constitué de son nombre d'argument: ce nombre va de 0 à 127, on peut donc reconnaître un symbole au fait qu'il peut être considéré comme un entier positif. Pour les autres types de données (variables, entiers, chaînes de caractères...), le type est fixe, et les trois octets restants représentent la donnée elle même, ou l'index de la variable. Les chaînes sont représentées sous forme d'un pointeur vers un

espace des chaînes où sont stockées leurs valeurs.

On a cherché à utiliser au maximum les pointeurs, de préférence aux index, pour tirer parti des possibilités offertes par le langage C (Cf §4.1). Le mécanisme de pointeurs est utilisé en particulier pour le chaînage des clauses, et pour les pointeurs entre la pile d'évaluation et la table des clauses. Ceci permet à Yaap d'être sensiblement plus rapide que Prolog I+ (environ 20%), alors que sa capacité opérative est accrue (travail sur 32 bits au lieu de 16). Par rapport à C-Prolog, qui tourne également sur SM 90, Yaap est à peu près 10% moins rapide, mais il est encore susceptible d'être amélioré.

Les prédicats évaluables implémentés permettent de lire des chaînes ou des caractères, d'écrire des entiers, des chaînes ou des termes complexes, de tester l'égalité formelle de deux termes, d'évaluer des expressions arithmétiques, d'ajouter des clauses... De nombreux prédicats évaluables restent à implémenter, et le seront progressivement.

Le passage à 32 bits de la taille des données et la fragmentation des espaces font que Yaap occupe plus de place en mémoire, à capacité équivalente, que Prolog I+. Cependant, les clauses et les chaînes sont codées de façon sensiblement plus compacte, ce qui permet de récupérer en partie cet inconvénient.

3. LA SM 90

La SM 90 est une machine à base de microprocesseurs développée par le CNET. Elle présente pour le projet Opale une certain nombre de caractéristiques intéressantes:

- C'est une machine multiprocesseurs,
- Elle prévoit la tolérance aux pannes,
- Elle est ouverte et très modulaire,
- Elle peut recevoir une grande variété de composants standard (processeurs, mémoires, coupleurs de périphériques),
- Elle dispose d'un moniteur en ROM accessible par l'utilisateur.

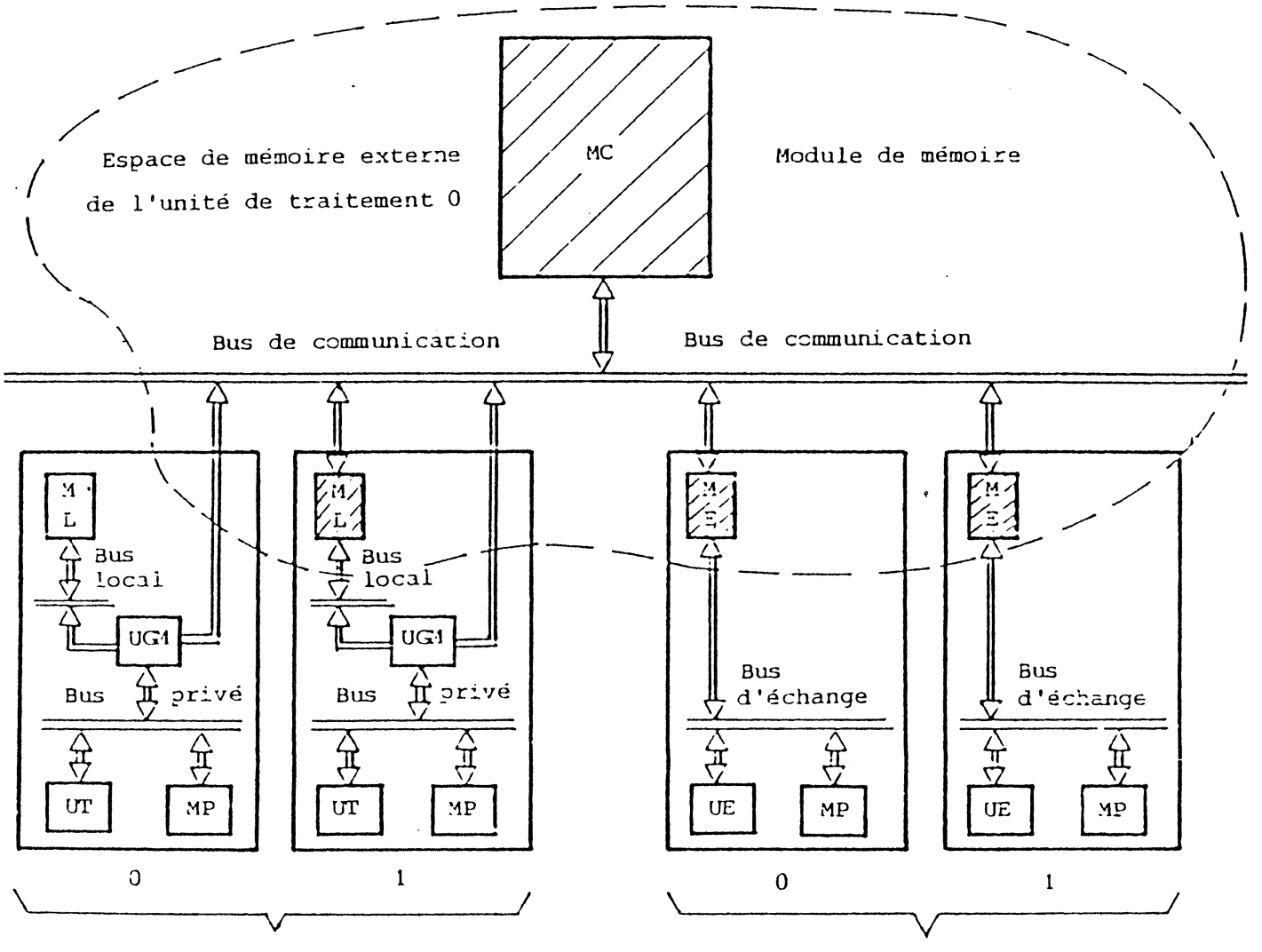
On prévoit donc que cette machine pourra nous dispenser d'un certain nombre de réalisations, en nous permettant de nous concentrer sur les points clés de l'étude (filtrage). On peut également espérer qu'elle fournisse le support d'une première valorisation du projet.

La machine est organisée autour d'un bus de communication (le SM bus) permettant à différents types de modules de communiquer (Fig. A3.1.):

- Les modules de traitement, qui peuvent prendre l'initiative des transferts sur le bus,
- Les modules d'échanges, qui ne peuvent pas prendre l'initiative de transferts,
- Les modules de mémoire commune (MC), accessibles par tous les modules de traitement.

Les modules d'échanges et de traitement comportent tous un processeur. Ils peuvent comporter en outre une mémoire privée (MP) (non accessible par les autres modules).

Les modules de traitement comportent en outre un bus local permettant de leur adjoindre une mémoire (ML) ou des périphériques locaux. Les mémoires locales peuvent



UT: unité de traitement
UE: unité d'échange
UGM: unité de gestion de la mémoire

Source: documentation CNET

Figure A3.1.

être à double accès, c'est à dire accessibles par le module de traitement par l'intermédiaire de son bus local, et accessibles par d'autres modules de traitement par l'intermédiaire du bus de communication.

Les modules d'échanges communiquent avec les modules de traitement par l'intermédiaire d'une mémoire d'échange (ME) accessible par le bus de communication ou par le bus local (ou bus d'échange) du module.

Le bus de communication comporte, outre les lignes de données et d'adresses, des lignes d'arbitrage (partage du bus), des lignes de réveil permettant aux modules d'échange de signaler qu'ils ont quelque chose à communiquer: le ou les modules de traitement devront alors aller consulter la mémoire d'échange des modules.

4. UNIX:

Il n'entre pas dans notre propos de décrire le système UNIX, déjà largement répandu. Rappelons simplement qu'il s'agit d'un système multiutilisateurs avec structure hiérarchique du système de fichiers, doté d'un langage de commande (le Shell) permettant d'enchaîner de façon simple divers travaux. La communication entre les processus se fait par un mécanisme appelé pipe: la sortie d'un processus peut être connectée à l'entrée d'un autre processus, la communication se faisant par le biais des opérations d'entrées sorties. Par le même moyen, la sortie d'un processus peut être "redirectée" sur l'entrée d'un autre, ce qui fournit un moyen simple et puissant pour enchaîner des travaux.

Nous allons ici parler plus particulièrement de deux outils que nous avons plus particulièrement utilisés, largement mentionnés dans cette thèse, et qui sont à l'origine des résultats expérimentaux cités: il s'agit du langage C et de l'utilitaire prof.

4.1. Le langage C:

Il s'agit d'un langage de haut niveau, portable, mais étudié plus particulièrement pour l'utilisation optimale d'une machine telle que le PDP 11 (ainsi que le MC 68000 qui en est très proche).

Ce langage permet une expression concise des algorithmes (parfois au détriment de la clarté), ainsi qu'une programmation structurée. Mais il permet aussi de contrôler de très près le code généré et d'arriver à un code très proche du code optimal qui pourrait être écrit directement en assembleur.

Ses caractéristiques essentielles sont les suivantes:

- pas d'imbrication des procédures: toute les procédures sont déclarées au niveau lexical le plus externe. Elles communiquent par le biais de variables externes communes, ou de paramètres. Les paramètres sont transmis par valeur, les paramètres par référence devant être faits "à la main" par le biais de pointeurs. Les procédures peuvent comporter des variables statiques, qui sont conservées d'un appel à l'autre.
- souplesse des structures de contrôle, en particulier pour les boucles for où il est possible de définir très librement les conditions initiales, la condition de fin, et l'incrémentation de la variable de contrôle.
- typage faible des données: tout type de donnée peut être considéré comme un entier, ce qui présente des inconvénients: risque d'erreurs, mais permet des conversions aisées entre les différents types, et surtout la manipulation aisée de données non statiquement typées.

- notion de pointeur souple et puissante: un pointeur peut faire référence à une donnée d'un type donné. On peut affecter à un pointeur l'adresse d'un élément d'un tableau. On peut également effectuer une addition ou une soustraction sur un pointeur: si un pointeur contient l'adresse de l'élément N d'un tableau, ajouter la valeur I à ce pointeur revient à le faire pointer sur l'élément N + I. Ceci permet un gain de performances appréciable en évitant de repasser par l'indexation lorsque plusieurs éléments d'un même tableau sont successivement adressés. Si I est une constante, l'opération se ramène à une simple addition. La soustraction de deux pointeurs revient à calculer la différence des indices des éléments adressés.
- accès simple à l'ensemble des fonctions standard de UNIX (comprenant des fonctions de calcul, d'entrées sorties, de tri, etc...).

4.2. l'utilitaire prof:

Il s'agit d'un moyen d'analyser la distribution du temps de traitement entre les diverses procédures d'un programme. Ceci est mis en oeuvre par une option du compilateur (C, Fortran, Pascal...). Les résultats sont stockées dans un fichier, puis analysés à l'aide de l'utilitaire prof.

La mesure est effectuée par deux moyens complémentaire:

- Un comptage du nombre d'appels de chaque procédure, grâce à un appel à une procédure de comptage inséré au début de chaque procédure,
- Un échantillonage statistique du compteur ordinal: tous les soixantièmes de seconde, une interruption est générée, et un compteur correspondant à la valeur du compteur ordinal est incrémenté. Les compteurs sont regroupés dans un tableau comportant un élément pour deux mots de 32 bits (ou plus).

La confrontation des nombres d'appels des procédures, de l'échantillonage du compteur ordinal, et des adresses des procédures permet à l'utilitaire prof de sortir le pourcentage du temps passé dans chaque procédure, le nombre d'appels, et la durée d'exécution moyenne.

Il s'agit d'une mesure statistique, qui n'est donc précise que dans la mesure où la durée d'exécution du programme est suffisamment longue. Lors des mesures sur Prolog, nous avons néanmoins constaté que les résultats étaient relativement précis et répétitifs.

Un autre problème est que l'échantillonage du compteur ordinal regroupe sur un même compteur plusieurs instructions. Il arrive donc que se produise un problème de voisinage entre deux procédures, des instructions de l'une étant comptabilisées à l'autre, du fait du manque de résolution spatiale de la mesure: cette inconvénient devient sensible si les procédures sont courtes et fréquemment appelées. Ainsi, nous avons eu la surprise de voir qu'une procédure de Prolog qui n'était jamais appelée se voyait gratifier d'un pourcentage d'exécution relativement élevé. Il était alors facile de rétablir la vérité, en restituant ce pourcentage à sa légitime propriétaire. Ceci nous suggère un moyen de préciser les mesures si nécessaire: il suffit d'encadrer la ou les procédures sensibles de procédures vides, jamais appelées, dont le seul rôle sera d'absorber un éventuel

"débordement" de leur voisine.

L'incidence du comptage peut n'être pas négligeable, toujours lorsque les procédures les plus courtes sont fréquemment appelées, mais elle peut être en partie évaluée, puisque le temps passé dans la procédure de comptage est évalué au même titre que le reste. Seul ne peut pas être pris en compte l'appel de cette procédure. L'interprétation des résultats demande donc un minimum de précautions.

ANNEXE 4

UTILISATION DES INSTRUCTIONS DU VAX (*)

[CLA 82] présente les pourcentages des opérations en fréquence et en temps d'exécution pour 7 cas:

- exécution d'un programme de calcul numérique en FORTRAN (Whetstone),
- exécution d'un programme en COBOL (COBOLX),
- compilation d'un programme FORTRAN,
- édition de liens du même programme,
- simulation d'une charge en temps partagé: système VMS en mode "kernel" (noyau système),
- idem en mode "exécutive" (gestion de fichiers),
- idem tous modes: la charge simulée comporte des mises à jour de fichiers indexés, un programme de multiplication de matrices, et des activités de type développement de programmes (éditions de textes, manipulations de fichiers, compilations, éditions de liens, exécution et mise au point de programmes en Basic, Fortran et COBOL).

Les auteurs de l'article observent que les divers jeux d'essai utilisent la machine de façon très différente.

Nous synthétisons ci-après les résultats dans deux tableaux:

- Le tableau II.1. présente la fréquence des 25 instructions les plus utilisées de chaque jeux d'essai. On a ajouté les résultats fournis dans [WIE 82].
- Le tableau II.2. présente le pourcentage du temps passé à exécuter les 25 opérations les plus fréquentes.

Nous avons trié les deux tableaux sur la valeur moyenne des pourcentages présentés. Nous avons indiqué en chiffres gras les nombres d'opérandes supérieurs ou égaux à 4 et les instructions utilisées par un seul jeu d'essai.

Notations:

N = nombre d'arguments
T = type d'opération:
- 1 = opérations logiques sur entiers et flottants,
- 2 = opérations arithmétiques sur entiers et flottants,
- 3 = opérations décimales,
- 4 = opérations sur chaînes de caractères,

(*) D'après Clark et Wiecceck [CLA 82, WIE 82]

- 5 = opérations sur chaînes de bits,
- 6 = index
- 7 = traitements de files,
- 8 = manipulations d'adresses,
- 9 = manipulation des registres généraux de la machine,
- 10 = branchements inconditionnels,
- 11 = branchements conditionnels,
- 12 = branchements sur valeur binaire,
- 13 = boucles et case,
- 14 = appels et retours de sous programmes,
- 15 = appels et retours de procédures,
- 16 = fonctions système,
- 17 = manipulation des registres processeur,
- 18 = fonctions spéciales.

1 = test sur compilateurs
2 = calcul numérique en FORTRAN
3 = programme COBOL
4 = compilation FORTRAN
5 = édition de liens,
6 = système VMS mode "kernel"
7 = VMS mode "exécutive"
8 = VMS tous modes.

Tableau II.1.: fréquence des instructions

code-op	N	T	1	2	3	4	5	6	7	8
movl	2	1	13.40	21.99	3.57	12.36	17.23	10.21	10.22	11.40
beql	1	11	5.10	-	2.65	4.49	2.75	5.59	8.69	5.85
bneq	1	11	5.30	-	5.29	8.72	2.66	3.99	3.60	3.07
cmpl	2	1	3.90	2.60	3.65	5.34	3.44	2.78	2.00	2.73
movab	2	8	4.40	-	2.32	5.56	6.08	1.23	-	1.97
movzbl	2	1	2.60	-	-	3.01	8.44	1.66	1.96	3.07
addl2	2	2	1.70	-	-	4.03	7.73	2.80	1.81	2.08
cmpb	2	1	2.30	-	4.97	2.21	-	1.44	4.83	2.52
rsb	0	14	-	1.54	-	1.37	1.71	4.33	3.06	2.68
incl	1	2	1.40	-	5.37	1.20	-	1.76	3.07	1.68
aobleq	3	13	-	5.08	-	1.09	5.34	-	-	2.70
cvttp	**5**	3	-	-	**13.52**	-	-	-	-	-
movzwl	2	1	-	-	-	3.10	1.86	3.25	2.24	1.99
bbs	3	12	-	-	-	-	1.21	2.40	5.25	2.77
brb	1	10	2.40	1.20	2.89	2.33	2.71	-	-	-
bbc	3	12	-	-	-	4.05	1.62	2.42	1.27	2.09
clrl	1	1	1.60	1.05	2.73	2.20	1.75	-	-	2.12

code-op	N	T	1	2	3	4	5	6	7	8
cmpw	2	1	3.90	1.24	-	2.71	-	-	2.20	1.17
ret	0	15	2.40	3.55	-	2.52	1.19	-	-	1.16
tstl	1	1	-	-	-	2.02	1.87	1.88	1.88	1.61
mulf3	3	2	-	6.42	-	-	-	-	-	2.34
blbc	2	12	-	-	-	-	1.57	1.84	3.32	1.95
pushl	1	9	2.00	-	-	2.93	-	1.36	-	1.47
calls	2	15	2.40	1.43	-	2.54	1.20	-	-	-
mull3	3	2	-	1.95	5.25	-	-	-	-	-
addf3	3	2	-	**7.15**	-	-	-	-	-	-
bgeq	1	11	-	1.38	3.33	-	-	1.88	-	-
moval	2	8	-	3.74	-	-	-	1.60	-	1.25
bsbw	1	14	-	-	-	-	1.17	2.38	1.73	1.29
subl3	3	2	-	2.47	-	-	1.44	-	2.49	-
brw	1	10	1.90	-	2.16	0.99	1.24	-	-	-
movc3	3	4	-	-	**6.19**	-	-	-	-	-
addp4	4	3	-	-	**5.78**	-	-	-	-	-
movw	2	1	2.10	-	-	1.68	-	1.66	-	-
cvtpt	5	3	-	-	**5.14**	-	-	-	-	-
movq	2	1	-	-	-	-	-	2.64	1.19	-
extzv	4	5	-	-	-	1.72	-	2.06	-	-
addf2	2	2	-	**3.54**	-	-	-	-	-	-
jmp	1	10	-	1.39	2.12	-	-	-	-	-
bleq	1	11	-	1.75	1.20	-	-	-	-	-
divf3	3	2	-	**2.79**	-	-	-	-	-	-
bgtru	1	11	-	-	-	-	1.34	-	1.43	-
cvtwl	2	1	**2.70**	-	-	-	-	-	-	-
movb	2	1	1.70	-	0.96	-	-	-	-	-
bsbb	1	14	-	-	-	-	-	1.29	1.34	-
mtpr	2	17	-	-	-	-	-	**2.62**	-	-
bcc	1	11	-	-	**2.60**	-	-	-	-	-
movc5	5	4	-	-	**2.54**	-	-	-	-	-
cmpp3	3	3	-	-	**2.49**	-	-	-	-	-
cmpc3	3	4	-	-	**2.37**	-	-	-	-	-
cvtfd	2	1	-	-	-	-	-	-	-	**2.34**
sobgtr	2	13	-	-	-	**2.15**	-	-	-	-
callg	2	15	-	**2.12**	-	-	-	-	-	-
bcs	1	11	-	-	-	-	-	-	**2.03**	-
addd2	2	2	-	-	-	-	-	-	-	**2.00**
blss	1	11	-	-	-	-	-	**1.79**	-	-
bbcc	3	12	-	-	-	-	-	-	**1.57**	-
popr	1	9	-	-	-	-	**1.57**	-	-	-
mulf2	2	2	-	**1.54**	-	-	-	-	-	-

code-op	N	T	1	2	3	4	5	6	7	8
bbsc	3	12	-	-	-	-	-	-	**1.53**	-
pushr	1	9	-	-	-	-	**1.49**	-	-	-
mull2	2	2	-	**1.46**	-	-	-	-	-	-
bgtr	1	11	**1.40**	-	-	-	-	-	-	-
jsb	1	14	-	**1.39**	-	-	-	-	-	-
clrw	1	1	-	-	-	-	-	-	**1.30**	-
addl3	3	2	-	-	-	-	**1.23**	-	-	-
polyf	3	2	-	**1.17**	-	-	-	-	-	-
subf3	3	2	-	**1.11**	-	-	-	-	-	-
cmpp4	**4**	3	-	-	**1.08**	-	-	-	-	-
ashl	3	2	-	-	-	**0.96**	-	-	-	-
TOTAL			64.60	81.05	90.17	81.28	79.84	66.86	70.01	65.30

Tableau II.2.: pourcentage du temps

code	N	T	2	3	4	5	6	7	8
movc3	3	4	-	20.82	1.14	3.96	-	23.30	13.14
movl	2	1	12.81	0.47	6.76	9.03	8.19	4.48	6.60
calls	2	15	6.55	-	21.59	11.77	-	-	7.80
ret	0	15	9.94	-	9.46	5.08	-	1.44	4.07
cvttp	**5**	3	-	**23.30**	-	-	-	-	-
movc5	**5**	4	-	11.90	4.86	2.25	1.91	-	-
bbc	3	12	-	-	6.32	2.90	3.39	3.82	2.74
bbs	3	12	-	-	-	2.15	2.94	6.44	3.11
callg	2	15	**13.01**	-	-	-	-	-	-
movzbl	2	1	-	-	2.35	5.84	1.32	1.19	1.84
addp4	**4**	3	-	**12.48**	-	-	-	-	-
aobleq	3	13	4.63	-	-	4.67	-	-	1.67
mulf3	3	2	6.04	-	-	-	-	-	3.59
cmpl	2	1	1.26	-	2.64	2.00	1.96	-	1.31
pushr	1	9	-	-	-	6.94	2.23	-	-
movzwl	2	1	-	-	2.41	1.09	2.70	1.27	1.27
cvtpt	**5**	3	-	**8.32**	-	-	-	-	-
rsb	0	14	0.94	-	-	1.08	2.57	2.03	1.59
bneq	1	11	-	0.64	3.70	1.19	1.39	1.26	-
extzv	4	5	-	-	2.70	1.17	2.83	-	1.26
bsbw	1	14	-	-	-	1.23	3.08	2.00	1.47
beql	1	11	-	-	1.46	-	1.97	2.47	1.67
movab	2	8	-	0.31	3.02	3.12	-	-	-
cmpb	2	1	-	0.81	1.21	-	-	2.93	1.47

code	N	T	2	3	4	5	6	7	8
mtpr	2	17	-	-	-	-	5.27	-	1.15
divf3	3	2	**6.28**	-	-	-	-	-	-
popr	1	9	-	-	-	4.38	1.88	-	-
movq	2	1	-	-	-	-	2.83	1.74	1.14
ediv	**4**	2	-	-	-	2.49	-	2.43	-
incl	1	2	-	0.83	-	-	1.81	1.97	-
rei	0	16	-	-	-	-	3.29	-	1.20
polyf	3	2	**4.41**	-	-	-	-	-	-
addf3	3	2	**4.39**	-	-	-	-	-	-
jsb	1	14	1.35	-	-	-	1.88	-	1.13
addl2	2	2	-	-	1.07	2.74	-	-	-
mull3	3	2	2.13	1.66	-	-	-	-	-
cmpc5	**5**	4	-	1.09	-	2.56	-	-	-
tstl	1	1	-	-	0.90	1.12	1.33	-	-
caseb	**4**	13	0.92	-	0.87	-	-	1.53	-
subl3	3	2	1.61	-	-	-	-	1.68	-
cmpw	2	1	-	-	1.88	-	-	1.38	-
blbc	2	12	-	-	-	-	-	1.97	1.03
divp	**6**	3	-	**2.99**	-	-	-	-	-
emodf	**5**	2	**2.97**	-	-	-	-	-	-
bbsc	3	12	-	-	-	-	-	1.83	1.10
movw	2	1	-	-	1.23	-	1.65	-	-
mulp	**6**	3	-	**2.79**	-	-	-	-	-
emul	**4**	2	-	-	-	1.55	-	1.17	-
bsbb	1	14	-	-	-	-	1.33	1.20	-
probew	3	16	-	-	-	-	-	**2.43**	-
cmpc3	3	4	-	**2.38**	-	-	-	-	-
brb	1	10	-	-	1.09	1.24	-	-	-
clrl	1	1	-	0.30	0.92	-	-	-	1.06
cmpp3	3	3	-	**2.20**	-	-	-	-	-
addf2	2	2	**2.11**	-	-	-	-	-	-
bbcc	3	12	-	-	-	-	-	**1.93**	-
prober	3	16	-	-	-	-	-	**1.89**	-
pushl	1	9	-	-	**1.86**	-	-	-	-
cvtfd	2	1	-	-	-	-	-	-	**1.60**
moval	2	8	**1.53**	-	-	-	-	-	-
bicl3	3	1	-	-	-	-	**1.50**	-	-
mfpr	2	17	-	-	-	-	**1.46**	-	-
addd2	2	2	-	-	-	-	-	-	**1.42**
mull2	2	2	**1.39**	-	-	-	-	-	-
remque	2	7	-	-	-	-	**1.31**	-	-
editpc	**6**	3	-	**1.25**	-	-	-	-	-

code	N	T	2	3	4	5	6	7	8
cmpv	**4**	5	-	-	-	**1.21**	-	-	-
divf2	2	2	**1.20**	-	-	-	-	-	-
sobgtr	2	13	-	-	**1.18**	-	-	-	-
cmpp4	**4**	3	-	**1.16**	-	-	-	-	-
mulf2	2	2	**1.15**	-	-	-	-	-	-
jmp	1	10	**0.75**	0.30	-	-	-	-	-
ashl	3	2	-	-	**1.01**	-	-	-	-
cvtlf	2	1	**1.01**	-	-	-	-	-	-
subf3	3	2	**0.97**	-	-	-	-	-	-
cmpzv	**4**	5	-	-	**0.89**	-	-	-	-
divd2	2	2	**0.80**	-	-	-	-	-	-
blequ	1	11	-	**0.39**	-	-	-	-	-
cvtlp	3	3	-	**0.39**	-	-	-	-	-
bgeq	1	11	-	**0.38**	-	-	-	-	-
movb	2	1	-	**0.32**	-	-	-	-	-
spanc	**4**	4	-	**0.28**	-	-	-	-	-
TOTAL			90.15	97.76	82.52	82.76	62.02	75.78	65.43

Le tableau II.1. contient 70 instructions dont 30 ne sont présentes que dans une colonne. Les pourcentages cumulés dans chaque colonne vont de 65,6 à 93,3%. 2 instructions seulement sont présentes dans les 9 colonnes, et 11 sont présentes dans au moins 7 colonnes. On note une faible utilisation des instructions de plus de 4 opérandes.

Le tableau II.2. contient 82 instructions, dont 41 ne sont présentes que dans une colonne: on voit donc s'accentuer la spécificité des instructions en fonction des applications. Les pourcentages cumulés des colonnes vont de 62,1 à 97,8%. Une seule instruction est présente dans toutes les colonnes. 9 sont présentes dans au moins 6 colonnes.

Un groupe d'instruction entier n'est utilisé que par le programme COBOLX (groupe 3: instructions décimales). Même si COBOL est d'usage général en gestion, ceci laisse tout un domaine où ce groupe d'instructions, souvent complexes, est inutilisé. Dans ce programme, les instructions décimales représentent 54,9% sur les 97,8% du tableau.

De même, le programme de calcul utilise seul un grand nombre d'instructions. Même si les opérations en virgule flottante sont souvent optionnelles sur une machine, on voit qu'elles ne sont (relativement) bien utilisée qu'à condition que la machine fasse essentiellement de l'exécution de programmes numériques, avec une faible proportion d'autres activités, telles que le développement de programmes. Dans ce programme, le pourcentage de temps passé dans les opérations flottantes est de 33,6% sur les 90,2% des instructions les plus utilisées; il y a donc une sous utilisation des opérateurs flottants.

BIBLIOGRAPHIE

[ABI 83] S. Abiteboul, M.O. Cordier, S. Gamerman, A. Verroust
Querying and filtering formatted files.
Int workshop on database machines, pp. 318-338
Springer Verlag, Munich, 1983

[ABI 87] S. Abiteboul, S. Grumbach
bases de données et objets complexes.
TSI, Vol. 6, No 5, Mai 1987

[ANC 74] F. Anceau
Contribution a l'étude des systèmes hiérarchisés de ressources dans l'architecture des machines informatiques.
Thèse d'état, INPG, Grenoble, 5 décembre 1974.

[ANC 81] F. Anceau
The French MPC
journées CAO et VLSI, Université Paris VI, juin 1981

[ANC 83] F. Anceau
CAPRI: A design methodology and a silicon compiler for VLSI circuits specified by algorithms
Caltech Conf. on VLSI mars,1983

[AMA 82] M. Amamiya, R. Hasegawa, O. Nakamura, H. Mikami
A list processing oriented data flow machine architecture.
NCC 1982

[ARM 83] J.P. Armisen
DORSAL 32: la machine base de données de Copernique.
5emes journées francophones sur l'informatique, Genève, 18-19 janvier 1983

[BAC 78] J. Backus
Can programming be liberated from Von Neumann style?
A functionnal style and its algebra of programs.
CACM. Vol. 21 No 8, Août 78

[BAC 80] J.S. Banino, A. Caristan, M. Guillemont, G. Morisset, H. Zimmerman,
CHORUS: an architecture for distributed systems.
Rapport INRIA No 42, novembre 1980.

[BAN 80] F. Bancilhon, M. Scholl
Design of a Backend processor for a database machine
Proc. ACM-SIGMOD, Santa Monica, Mai 1980

[BAN 81] F. Bancilhon, P. Richard, M. Scholl
The relationnal database machine VERSO
6th Workshop on comp. arch. for non numeric processing, juin 1981

[BEK 83] Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro
Machine PROLOG pour les applications d'intelligence artificielle: gestion de mémoire.
5emes journées francophones sur l'informatique, Genève, 18-19 janvier 1983

[BER 80] P. Bruce Berra
A summary of the state of the art in data base machines.
Journées INRIA machines bases de données, septembre 1980

[BER 86] G. Berger Sabbatel, W. Dang, J.C. Ianeselli
Mécanismes d'accès aux bases de données Prolog dans le projet Opale.
IMAG, Rapport technique No 4. Juin 1986

[BER2 86] G. Berger Sabbatel
Mesures comportementales sur l'interpretation de Prolog.
Séminaire CNET sur la programmation en logique, Trégastel, Mai 1986.

[BIT 83] D. Bitton, H. Boral, D.J. DeWitt, W.K. Wilkinson
Parallel algorthms for the execution of relationnal database operations.
ACM/TODS, Vol. 8, No. 3, pp 324-353, Septembre 1983

[BOD 81] A. Bode, W. Handler
Some results on associative processing by extending a microprogrammed general purpose processor
6th Workshop on comp. arch. for non numeric processing, juin 1981

[BOG 83] R. Bogdanowicz, M . Crocker, D.K. Hsiao, C Ryder, V. Stone, P. Strawser
Experiments in benchmarking relational database machines.
IWDM 83

[BOR 81] Haran Boral, David J. De Witt, W. Kevin Wilkinson
Performance evaluation of associative disk designs
6th Workshop on comp. arch. for non numeric processing, juin 1981

[BOR 82] Haran Boral, David J. De Witt
Applying data flow techniques to data base machines
Computer, pp. 57-63, Aôut 1982.

[BOR 83] H. Boral, D.J. De_Witt
Data Base Machines: an idea whose time has passed?
A critique of the future of data base machines
In "data base machines", Springer Verlag, septembre, 1983

[BOY 72] R.S. Boyer, J.S. Moore
The sharing of structure in theorem proving programs
In Machine intelligence 7, Edinburg university press. 1972.

[BOY 77] R.S. Boyer, J.S. Moore
A fast string searching algorithm
CACM. Vol.20 No 10, octobre 1977

[BRU 82] M. Bruynooghe
The memory management of Prolog implementations.
In "Logic Programming", Clark and Taernlund, Academic Press, 1982.

[CHA 82] U.S. Chakravarthy, J. Minker, D. Tran
Interfacing predicate logic languages and relationnal databases
1st int logic programming conference, Marseilles, septembre 1982

[CHE 85] D.R. Cheriton
Evaluating hardware support for superconcurency with the V Kernel.
Congrès AFCET "Matériels et logiciels pour la 5ème génération". Paris, Mars 1985.

[CHO 81] E. Chouraqui
Contribution à l'étude théorique de la représentation des connaissances. Le système symbolique Arches.
Thèse d'état, INPL Nancy, Octobre 1981.

[CLA 79] K.L. Clark, F.G. Mc Cabe
The control facilities of IC-PROLOG
In "Expert systems in the microelectronics age", Edinburg University Press. 1979.

[CLA 82] D.W. Clark, H.M. Levy
Measurements and analysis of instruction use in the VAX-11/780
9th symposium on Comp. Architecture, IEEE. Austin, Avril 1982

[CLO 81] W.F. Clocksin, C.S. Mellish
Programming in PROLOG
Springer Verlag.

[COD 70] E.F. Codd
A relational model of data for large shared data banks
CACM Vol.13, No6, pp 377-387, juin 1970

[COL 79] A. Colmerauer, H. Kanoui, M. Van Caneghen
Etude et réalisation d'un système PROLOG.
Université d'Aix Marseille. GIA. Rapport IRIA/SESORI No 77030.

[COL 83] A. Colmerauer, H. Kanoui, M. Van Caneghem
PROLOG: bases théoriques et développements actuels.
TSI, Vol. 2, No. 4, Juillet 1983

[CON 81] J.S. Conery, D.F. Kibler
Parallel interpretation of logic programs
ACM Conf on functionnal prog. lan. and comp. arch. Portsmouth, Octobre 1981.

[COU 81] B. Courtois, M. Marinescu, J.F. Pons.
SKALP: Skeleton Architecture for Local Processing.
IMAG. RR. 212. 1981.

[COU 82] B. Courtois
Debugging VLSI using SEM
IMAG. RR 310, octobre 1982.

[CUR 83] T.W. Curry, A. Mukhopadhyay
Realization of Efficient Non Numeric Operations Through VLSI.
VLSI 83

[DAN 83] W. Dang
Etude du système de communication pour la machine bases de données Opale.
Rapport de DEA, IMAG, Juin 1983.

[DAN 87] W Dang
Parallélisme dans une machine base de connaissances Prolog.
Thèse de docteur de l'INPG, Grenoble, janvier 1987.

[DEG 85] D. DeGroot
Alternate graph expressions for restricted and-parallelism
Compcon 1985

[DEN 67] P.J. Denning
Effects of scheduling on file memory operations
SJCC. 1967.

[DEN 79] Jack B. Dennis
The varieties of data flow computers
1st Int conf on dist computing systems, 1979

[DEW 79] David j. De Witt
DIRECT - A mutiprocessor organization fo supporting relational database management systems
IEEE Trans. on Computers, Vol C-28. No6, juin 1979

[DIT 80] D.R. Ditzel, D.A. Patterson
Retrospective on High level language computer architecture.
7th Annual symp. on Computer Architecture. IEEE. 1980.

[DON 82] P. Donz
FOLL, un dialecte du langage PROLOG.
Manuel d'utilisation. CRISS. Grenoble.

[DON 84] P. Donz, R. Hurtado
Le langage D-Prolog. Initiation aux langages de la 5ème génération
Editests, 1984.

[EIS 82] N. Eisinger, S. Kasif, J. Minker
Logic programming: a parallel approach
1st Int. Logic programming conf. Marseilles, Sept 1982.

[FEL 79] J.A. Feldman
High level programming for distributed computing
CACM,Vol.22, No6, Juin 1979, pp. 353/368

[FIN 83] U. Finger, G. Medigue
La structure multiprocesseurs SM 90
Journées francophones sur l'informatique, Geneve, Janvier 1983

[GAL 81] H. Gallaire
Impacts of logic on data bases
VLDB 81

[GAL 83] H. Gallaire
Prolog et bases de données
Séminaire CNET sur la programmation en Logique, Perros Guirec, Mars 1983

[GAR 81] G. Gardarin
An introduction to SABRE multimicroprocessor data base machine
6th Workshop on comp. arch. for non numeric processing, juin 1981

[GBS 81] G. Berger Sabbatel
Projet CRESUS: présentation générale
IMAG: RR. 263. Août 1981

[GB2 81] G. Berger Sabbatel
Projet CRESUS: service de communication.
IMAG. RR. 264, Août 1981.

[GB3 81] G. Berger Sabbatel
Projet CRESUS: Etude des systèmes de communication adaptés à des réseaux locaux de microprocesseurs.
IMAG. RR. 256. juillet 1981.

[GB2 82] G. Berger Sabbatel, Nguyen G.T.
OPALE: Une machine bases de données PROLOG.
Séminaire ADI bases de données, Toulouse, Novvembre 1982.

[GBS 82] G. Berger Sabbatel, Nguyen G.T.
Projet OPALE: Motivations et principes pour une machine bases de données PROLOG
IMAG. RR. 339. Decembre 1982

[GB1 83] G. Berger Sabbatel, Nguyen G. T.
Une machine bases de données PROLOG
Journées francophones sur l'informatique, Geneve, Janvier 1983

[GB2 83] G. Berger Sabbatel, A. Coeur, Nguyen Gia toan, P. Winninger
La machine bases de données OPALE
Séminaire CNET sur la programmation en logique. Perros Guirrec, Mars 1983.

[GB4 83] G. Berger Sabbatel, J.C. Ianeselli, Nguyen G.T.
A PROLOG Data base machine
In "Data base machines". Springer Verlag. septembre 1983

[GBS 84] G. Berger Sabbatel, W. Dang, J.C. Ianeselli, G.T Nguyen
Unification for a Prolog data base machine
2nd Int. Logic programming Conf. Uppsala, juillet 1984

[GBS2 84] G. Berger Sabbatel, J.C. Ianeselli
Un automate d'unification pour la machine OPALE.
IMAG, rapport de recherche No 465, Octobre 1984

[GBS 85] G. Berger Sabbatel, W. Dang, J.C. Ianeselli
Stratégie de recherche et unification pour la machine OPALE
Congrès AFCET "Matériels et logiciels pour la 5ème génération", Paris, Mars 1985.

[GON 84] R. Gonzales-Rubio, J. Rohmer, D. Terral
The SCHUSS filter: a processor for non numerical data processing
Int. Symp. on Computer Architecture, Ann Arbor, Juin 1984

[HOA 78] C.A.R. Hoare
Communicating sequential processes
CACM, Vol 21, No. 8, 1978

[HUE 76] G. Huet
Résolution d'équations dans des langages d'ordre 1,2, ... ω
Thèse d'état, Université de Paris VII, 1976.

[IAN 85] J.C Ianeselli
Un operateur d'unification pour une machine base de connaissances PROLOG
INPG, Thèse de 3ème cycle ,1985

[JAM 85] R. Jamier, A. Jerraya
Apollon: a datapath silicon compiler
ICCD, New York, 1985

[KIR 85] Claude Kirchner
Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles.
Thèse d'état, Université de Nancy, juin 1985

[KNU 77] D.E. Knuth, J.H. Morris Jr, V.R. Pratt
Fast pattern matching in strings
SIAM journal on computing. Vol 6 No 2, juin 77

[KOD 85] Y. Kodratoff, H. Perdrix, M. Franova
Traitement symbolique du raisonnement incertain.
Congrès AFCET "Matériels et logiciels pour la 5ème génération". Paris, Mars 1985.

[KOW 74] R. Kowalski
Predicate logic as a programming language.
IFIP. 1974

[KRU 81] F. Kruzela, B. Svensson
The Lucas architecture and its applications to relationnal data base management.
6th Workshop on comp. arch. for non numeric processing, juin 1981

[LAR 87] B. Lardoux
Définition d'un filtre matériel pour base de connaissances Prolog.
rapport de fin d'études d'ingénieur. INPG/ENSERG, Grenoble, Juin 1987

[LIN 84] G. Lindstrom, P. Panangaden
Stream-based excution of logic programs.
Int. Conf. on Logic Programming, Atlantic City, Fevrier 1984

[LIP 76] G.J. Lipovski
Architectural features of CASSM: A Context Adressed Segment Sequential Memory.
5th Symposium on Comp. Architecture.

[LIT 82] D.Litaize et al.
Hiérarchies de mémoires intégrant des mémoires à bulles magnétiques.
Congrès AFCET, 1982

[MAL 83] A. Malek
Analyse de description de schémas électriques.
Rapport de projet, INPG, Juin 1983

[MAR 78] M. Marinescu
Mecanisme de communication par bus série pour des réseaux informatiques locaux.
Thèse de 3eme cycle. IMAG. Septembre 1978

[MAR 82] A. Martelli, U. Montanari
An efficient Unification algorithm.
ACM Trans on Prog. Languages and Systems, Vol. 4. No 2, Avril 1982, pp 258-282

[MEG 84] A. Mégrelis, D. Sciamma, T. Banel
Premières mesures sur l'interpréteur de Prolog II.
Séminaire CNET sur la programmation en logique, 1984

[MEL 82] C.S. Mellish
An alternative to structure sharing in the implementation of a PROLOG interpreter.
In "Logic Programming", Clark and Taernlund, Academic Press, 1982.

[MEN 81] M.J. Menon, D.K. Hsiao
Designs and analysis of a relationnal join operation for VLSI
VLDB 81

[MET 76] R.M. Metcalfe, D.R. Boggs.
Ethernet: distributed packet switching for local computer networks.
CACM. Vol. 19 No 7. Juillet 1976.

[MIN 82] J. Minker
An experimental relationnal data base system based on logic.
In "Logic Programming", Clark and Taernlund, Academic Press, 1982.

[MUK 83] K. Murakami, T. Kakuta, N. Miyazaki, S. Shibayama, H. Yokota
A relational database machine: first step to knowledge base machine.
10th Anual Symp on Comp. Architecture, Stockholm, 1983.

[MUR 81] M. Muraszkiewicz
Concepts of sorting and projection in a cellular array
VLDB 81

[NAK 82] H. Nakashima
Prolog/KR - Language Features
1st Int logic programmin conference, Marseilles, Septembre 1982

[NAV 79] P. Navaux, G. Berger Sabbatel
A data base processor: hardware design and implementation based on MAGE
Congrès EUROMICRO. Août 1979

[NGU 82] G.T. Nguyen, F. Fernandez, L. Ferrat, Y.J. LEE
MICROBE - Manuel de référence
Rapport IMAG, Janvier 82

[NIC 83] J.M. Nicolas
Bases de données logiques et programmation en logique.
Séminaire CNET sur la programmation en logique, Perros Guirec, mars 1983.

[OZK 75] E.A. Ozkarahan, S.A. Schuster, K.C. Smith
RAP: an associative processor for data base management.
NCC-AFIPS 1975, Vol. 44, pp. 379-387

[PAT 80] D.A. Patterson, C.H. Sequin
Design considerations for single chip computer of the future
IEEE Trans on computers. Vol. C29, No 2, fevrier 1980

[PAT 81] D.A. Patterson, C.H. Sequin
RISC I: A reduced instruction set VLSI computer.
8th annual symposium on computer architecture, Mai 81

[PAT 82] D.A. Patterson, R.S. Piepho
RISC assessment: a high level language experiment
9th annual symposium on Comp. architecture. IEEE. Austin, Avril 1982.

[PAT 85] D.A. Patterson
Reduced instruction set computers
CACM, Vol 28 No 1, pp. 8-21, Janvier 1985

[PEA 82] L.M. Pereira, A. Porto
Selective backtracking
In "Logic programming", Academic press. 1982

[PER 82] C. Percebois, J.P. Sansonnet
A LISP-Machine to implement PROLOG
1st int. Logic programming conference, Marseille. Septembre 1982.

[PLA 84] D. Plateau
A pruned Trie to index a sorted file and its evaluation.
Information Systems, Vol. 9, No 2, pp 157-165, 1984

[POS 83] C. Postigo
Cohérence de copies multiples avec latence de détection d'erreur et test fonctionnel de microprocesseur.
INPG, thèse de 3ème cycle, Juin 1983.

[POU 76] G. Poujoulat
The CORAIL building block system.
EUROMICRO Newsletter, Octobre 1976.

[PRA 81] S. Pramanick
Hardware organization for non numeric processing
VLDB 81

[ROB 65] J.A. Robinson
A machine oriented logic based on the résolution principle
JACM 12, 1, decembre 1965, pp227-234

[ROB 79] J.A. Robinson
The logical basis of programming by assertion and query
In Expert systems in the microelectronics age, Edinburg University Press, 1979.

[ROH 80] J. Rohmer
Machines et langages pour traiter les ensembles de données
Thèse d'état, INPG, Grenoble, décembre 1980

[ROH 81] J. Rohmer
Associative filtering by automata: a key operator for data base machines
6th Workshop on comp. arch. for non numeric processing, juin 1981

[ROH 82] J. Rohmer
Le projet BOOMERANG
Seminaire bases de données. ADI. Toulouse. Novembre 1982

[ROH 85] J. Rohmer, R. Lescoeur
La méthode d'alexandre: une solution pour traiter les axiomes recursifs dans les bases de données déductives
Congrès RFIA 1985, pp. 915-927, AFCET, Grenoble, novembre 1985

[ROU 75] J. Roussel
PROLOG: manuel de référence et d'utilisation
Rapport, Groupe d'intelligence artificielle, Université d'Aix Marseille II, 1975

[SAK 85] H. Sakai, K. Iwata, S. Shibayama, M. Abe, H. Ito
Development of Delta as a first step to a knowledge base machine.
ICOT, rapport technique 0121, Juillet 1985

[SAN 81] J.P. Sansonnet, M. Castan, C.Percebois
Architecture of a multi language processor based on list-structured DELs
6th Workshop on comp. arch. for non numeric processing, juin 1981

[SAN 85] J.P Sansonnet
La machine pour les applications d'intelligence artificielle: MAIA.
Congrès AFCET "Matériels et logiciels pour la 5ème génération". Paris, Mars 1985.

[SCH 78] J.P. Schoellkopf
A tutorial on high level language machine for Pascal
IMAG. RR. 131, octobre 1978

[SCH 85] M. SCHOLL
Architecture pour le filtrage dans les bases de données relationnelles
INPG, Thèse d'état, Grenoble, 19 Juin 1985

[SCH 87] M. SCHOLL, V. DELEBARRE
Verso, un système relationnel N1NF à filtrage séquentiel: mesures et bilan
TSI, Vol. 6, No 5, pp. 479-490, Mai 1987

[SHO 80] J.F. Shoch, J.A. Hupp
Measured performances of an ethernet local network.
Rapport Xerox-PARC. Fevrier 1980.

[SHO 82] J.F. Shoch, Y.K. Dalol, D.D. Redell, R.C. Crane.
Evolution of the ethernet local computer network.
IEEE Computer. Aoçut 1982.

[SYR 85] J.C. Syre
Une revue de modèles parallèles pour Prolog
Séminaire CNET sur la programmation en logique. Trégastel. Mai 1985.

[TAK 84] S. Takagi et al.
Overall design of SIMPOS.
2nd Int. Logic Programming Conference. Uppsala, juillet 1984

[TAN 82] Y. Tanaka
A data stream database computer
In Computer Science and technologies, North Holland, 1982

[TES 79] J.A. Test
An interprocess communication scheme for the support of cooperating process networks.
1st Int. Conf. on dist. Comp. Systems, Huntsville, Alabama, octobre 1979.

[TIC 84] E. Tick, D. H. D. Warren
Towards a pipelined Prolog processor.
Int. Symp. on logic programming, pp. 29-40, février 1984, Atlantic City

[TR2 82] P.C. Treleaven
VLSI processor architecture.
Computer, Juin 82

[TRE 82] P.C. Treleaven, I.G. Lima
Japan's fifth generation computer systems
Computer. Août 1982

[TUS 80] D. Tuséra
Un automate à pile pour la recherche d'information.
Journées INRIA Machines bases de données, septembre 1980.

[UME 83] S. Umeyama, K. Tamura
A parallel execution model of logic program
10th Annual symposium on computer architecture, pp. 349-355, ACM, Stockholm, 1983

[UCH 83] S. Uchida
Inference machine: from sequential to parallel
10th Annual Symp. on Computer Architecture, pp. 410-416, ACM, Stockholm, 1983

[VAN 83] M.H. Van Emden
An interpreting algorithm for Prolog programs.
1st Int. logic programming conference, Marseille, septembre 1983

[WAR 75] D. Warren
PROLOG on the DEC 10 system
In Expert systems in the microelectronics age, Edinburg University Press, 1979.

[WAR 77] David H.D. Warren, L.M. Pereira, F. Pereira
PROLOG: the language and its implementation compared with LISP
ACM symposium on artificial intelligence an programming language, août 1977

[WAR2 77] David H. D. Warren
Implementing Prolog. Compiling predicate logic programs.
Edinburg University. D.A.I. Research report No 39-40, Mai 1977

[WAR 81] David H.D. Warren
Efficient processing of interactive relationnal database queries expressed in logic
VLDB 81

[WAR 83] David H.D. Warren
An abstract Prolog intruction set
SRI, technical note 309, Octobre 1983

[WAR 84] D.S. Warren, M. Ahamad, S.K. Debray, L.V. Kalé
Executing distributed Prolog programs on a broadcast network
1984 Int. Symp. on logic programming , pp. 12-21
IEEE, Atlantic City, Fevrier 1984

[WIE 82] C.A. Wiecek
A case study of VAX-11 instruction set usage for compiler execution.
Symp. on architectural support for Prog. languages and operating systems.
ACM. Mars 1982.

[WIL 83] P. Wilson
Occam architecture bases system design
Computer design, Novembre 1983

[WIP 84] M. J. Wise, D. M. W. Powers
Indexing Prolog clauses via superimposed code words and field encoded words.
Int. Symp. on logic programming, pp. 203-210, février 1984, Atlantic City

[YOK 83] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, S. Uchida
The design and implementation of a personal sequential inference machine: PSI.
New generation computing, Vol. 1, pp. 125-144, 1983

département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 5 de l'arrêté du 16 avril 1974

VU les rapports de présentation de Messieurs

. F. ANCEAU
. M. SCHOLL
. J. ROHMER

Monsieur Gilles BERGER SABBATEL

est autorisé à présenter une thèse en soutenance en vue de l'obtention du grade de DOCTEUR D'ETAT ES SCIENCES.

Fait à Grenoble, le 22 juin 1988

Le Président de l'Université Joseph Fourier

Le Président

J.J. PAYAN

Le Président de l'I.N.P.-G

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble
P.O. le Vice-Président,