Résumé

PROLOG est parfaitement adapté au traitement des clauses de Horn mais ne permet pas de traiter naturellement les problèmes liés à la déduction pour les logiques non-classiques, les théories équationnelles, les prédicats flous. Tous ces différents cas réclament des moyens différents de calculer un nouveau but à partir du but courant.

Nous définissons dans cette thèse un mécanisme général, le système TARSKI, permettant de développer des extensions de PROLOG :

- Réécriture complète des mécanismes d'inférences et développement d'un nouveau formalisme permettant de traiter des clauses de Horn généraslisées.
- Développement d'une machine abstraite pour implanter des règles de résolution paramétrables.
- Etude et implantation du parallélisme.

Abstract

PROLOG is just what is needed to handle the Horn clause fragment of first order logic, but can not represent in a simple way non-classical logic deduction, equationnal theories, fuzzy predicates. All these different cases need different ways of computing a new goal from an existing one.

We present here a general framework, the TARSKI system, for developping extensions of PROLOG :

- Complete rewriting of the inference mechanism and development of a new formalism to handle generalization of Horn clauses.
- Development of an abstract machine.
- Implementation of parallelism.

Je tiens à remercier :

- Mrs Martti Penttonen, professeur à l'université de Joensuu (Finlande), Christian Percebois, maîiitre de conférences à l'université Paul Sabatier et Patrick Salle, professeur à l'Institut National Polytechnique de Toulouse qui étaient rapporteurs de cette thèse. Les critiques et conseils qu'ils m'ont prodigués m'ont permis de mener ce travail à bien.
- Mr Jean Vignolles, Professeur à l'université Paul Sabatier, qui a bien voulu présider le jury de thèse.
- Mrs Guy Cousineau, Professeur à l'Ecole Normale Superieure de Paris, et Jean-Marc Garot, chef du Centre d'Etudes de la Navigation Aérienne, qui ont accepté de participer à ce jury de thèse.
- Mon directeur de thèse, Luis Fariñas Del Cerro. Il m'a tout appris de la logique modale, des techniques de résolution en logique non-classique, et m'a toujours consacré tout le temps nécessaire à l'avancement de mon travail. Il a constamment fait preuve d'une grande disponibilité et d'une inappréciable gentillesse.
- L'ensemble de l'équipe "Formalisation du raisonnement", et tout particulièrement Andreas Herzig qui aura passé beaucoup de son temps à m'expliquer simplement des choses bien compliquées.
- Le Centre d'Etudes de la Navigation Aérienne et l'Ecole Nationale de l'Aviation Civile, qui ont assuré ma survivance matérielle.
- L'ensemble des gens qui ont partagé le même bureau que moi au fil de ces dures années de labeur : Paul-Henry Mourlon, Michel Sabatier et Philippe Quéinnec.
- Mes parents, qui ont assuré la correction orthographique de ce document, entre autres choses.

Table des matières

I Fondements **19**

1 Déduction et résolution **21**
1.1 Rappels . . . 21
1.1.1 Clauses de Horn . . . 21
1.1.2 Principe de déduction . . . 21
1.1.3 Principe de résolution . . . 22
1.1.4 Forme procédurale du principe de résolution . . . 22
1.1.5 Résolution SLD . . . 22
1.1.6 Nouvelle formalisation du principe de résolution . . . 23
1.1.7 Extension au calcul des prédicats . . . 24
1.2 Extensions des clauses de Horn et de la résolution SLD . . . 25
1.2.1 Clauses de Horn généralisées . . . 26
1.2.2 Extension de la résolution SLD . . . 26

2 Modèles et implantations existants **29**
2.1 Implantations "classiques" de PROLOG . . . 29
2.1.1 Historique . . . 29
2.1.2 Les PROLOG interprétés . . . 30
2.1.3 PROLOG et compilation :la machine de Warren . . . 35
2.2 PROLOG et parallélisme . . . 36
2.2.1 Parallélisme OU . . . 37
2.2.2 Parallélisme ET . . . 39
2.2.3 Langages gardés . . . 40
2.2.4 Approches mixtes . . . 41
2.3 Méta-programmation . . . 42
2.3.1 Templog . . . 42
2.3.2 Chronolog . . . 43
2.3.3 RACCO . . . 44
2.3.4 Gödel . . . 44
2.4 Conclusion . . . 45

II Spécifications de l'automate **47**

3 Objets manipulés **49**
3.1 TARSKI vu comme un automate formel . . . 49

3.2 Les clauses 49
3.2.1 Les atomes 49
3.2.2 Les prédicats 50
3.2.3 Les variables 50
3.2.4 Les doublets et les listes 50
3.2.5 Les opérateurs 50
3.2.6 Définition formelle des clauses et objets manipulés 51
3.2.7 Interprétation 51
3.3 Règles de résolution 52
3.3.1 Les règles générales 53
3.3.2 Règles réflexives 54
3.3.3 Règles de terminaison 54
3.3.4 Extensions à la forme générale 55
3.4 Les règles de réécriture 55
3.5 Conclusion 55

4 Moteur d'inférence **57**
4.1 Mécanisme général 57
4.1.1 L'unification 57
4.1.2 Cycle de fonctionnement 58
4.2 Sélection des clauses 59
4.3 Sélection des règles de résolution 59
4.4 Exécution des règles de résolution 60
4.5 Réécriture de la résolvante 61
4.6 Terminaison de la résolution 61
4.7 Conclusion 61

5 Problèmes de contrôle **63**
5.1 Contrôle lors de la sélection des règles 63
5.1.1 Utilisation de l'extension procédurale 64
5.1.2 Construction d'un graphe de résolution 64
5.1.3 Enumération exhaustive et indexation 67
5.1.4 Le parallélisme intrinsèque de la résolution 70
5.2 Contrôle lors de l'exécution du programme 72
5.2.1 *cut* classique 72
5.2.2 *KUT* 72
5.3 Exemple 73

III Conception **79**

6 Conception **81**
6.1 Introduction 81
6.2 Choix effectués 81
6.2.1 Partage de structures contre copie de structures 81
6.2.2 Les contraintes du parallélisme 82
6.3 Architecture générale 83

6.4 La machine de niveau 0 84
6.4.1 Les objets de base 85
6.4.2 La pile des opérandes 85
6.4.3 les clauses 86
6.4.4 Les environnements 86
6.4.5 La résolvante 86
6.4.6 La question 86
6.4.7 Les points de choix 86
6.4.8 La pile de trainée 87
6.4.9 La tables des noms de prédicat 87
6.4.10 Les fonctions 87
6.4.11 Les registres 87
6.5 La machine de niveau 1 88
6.5.1 Déréférencement 88
6.5.2 Unification 88
6.5.3 Prédicats pré-définis 88
6.6 La machine de niveau 2 89
6.6.1 Sélection de clauses 89
6.6.2 Sélection et exécution de règles 91
6.7 Le moteur d'inférence 94
6.7.1 Fonctionnement général 94
6.7.2 Succès 97
6.7.3 Echec total 98
6.7.4 Justification de certains choix 98
6.8 Le jeu d'instruction de la machine TARSKI 99

7 Conception du parallélisme 101
7.1 Modèle de parallélisme 101
7.1.1 Principes généraux 101
7.1.2 La machine parallèle élémentaire 102
7.1.3 Modifications par rapport à la machine classique 102
7.2 Différentes géométries de réseau 105
7.2.1 Réseau en anneau 105
7.2.2 Réseau "du haut vers le bas" 106
7.2.3 Réseau en étoile 106
7.2.4 Réseau totalement interconnecté 107
7.2.5 Conclusion 107

IV Implantation 109

8 Implantation de la machine séquentielle 111
8.1 Choix effectués 111
8.1.1 Choix du langage d'implantation 111
8.1.2 Efficacité versus lisibilité 112
8.2 Architecture 113
8.3 Piles génériques étendues 113

8.3.1 But . . . 113
8.3.2 Entités exportées . . . 115
8.4 Les tables génériques . . . 115
8.4.1 Entités exportées . . . 116
8.5 Les types de base . . . 116
8.5.1 Entités exportés . . . 117
8.5.2 Implantation . . . 123
8.6 Les registres . . . 124
8.6.1 Entités exportées . . . 124
8.7 L'unificateur . . . 125
8.7.1 Entités exportées . . . 125
8.7.2 Implantation . . . 125
8.8 Les prédicats pré-définis . . . 125
8.8.1 Entités exportées . . . 125
8.8.2 Implantation . . . 126
8.9 Sélection de clause . . . 127
8.9.1 Entités exportées . . . 127
8.9.2 Implantation . . . 127
8.10 Sélection et exécution de règles . . . 128
8.10.1 Entités exportées . . . 128
8.10.2 Implantation . . . 129
8.11 Le moteur d'inférence . . . 131
8.11.1 Entités exportées . . . 131
8.11.2 Implantation . . . 131
8.12 L'analyseur syntaxique . . . 132
8.12.1 Entités exportées . . . 132
8.12.2 Implantation . . . 133
8.13 Debugging . . . 133
8.14 Conclusion . . . 133

9 Implantation du parallélisme 135
9.1 Contraintes matérielles . . . 135
9.2 Choix effectués . . . 135
9.2.1 Le problème du langage . . . 135
9.3 Communication inter-processeurs . . . 136
9.3.1 Choix du type de communication . . . 136
9.3.2 Protocole de communication . . . 137
9.3.3 Implantation de la couche basse . . . 137
9.3.4 Protocole de transfert . . . 138
9.3.5 Objets véritablement transférés . . . 138
9.4 Analyse de performances . . . 138

V Typologie et exemples d'applications 141

10 Typologie des règles de résolution 143
10.1 Forme 1 . . . 143

10.2 Forme 2 144
10.3 Forme 3 144
10.4 Forme 4 146
10.5 Forme 5 146
10.6 Forme 6 148
10.7 Forme 8 148
10.8 Forme 9 149
10.9 Forme 10 150
10.10Forme 11 151
10.11Remarques 152

11 Exemples d'applications **153**
11.1 Rappels 153
11.2 Implantation des règles de résolution 154
11.2.1 Enchaîinement des règles 156
11.2.2 Conclusion 158
11.3 Règles de réécriture 158
11.4 Le problème du contrôle :le KUT 160
11.4.1 Implantation des règles de résolution 161
11.4.2 Enchaîinement des règles 161
11.4.3 Règle de réécriture 162
11.5 Exemple :le problème des sages et des chapeaux 162
11.5.1 Enoncé du problème 162
11.5.2 Formalisation dans multi-S4 163
11.5.3 Traduction en langage TARSKI 164
11.6 Influence de l'ordre des règles 164
11.7 Logique des modules 165
11.7.1 Règles 165
11.7.2 Implantation des règles de résolution 165
11.7.3 Le chaîinage des règles de résolution 167
11.7.4 Règle de réécriture 167
11.7.5 Conclusion 167
11.8 Logique floue 167
11.8.1 La règle de réécriture 167

VI Conclusion **171**

VII Annexes **175**

A Détails d'implantation **177**
A.1 Les piles génériques étendues 177
A.1.1 Paramètres de généricité 177
A.1.2 Implantation 177
A.2 Les tables génériques 178
A.2.1 Paramètres de généricité 178

A.2.2 Entités exportées 179

B Le partage de structures **181**
B.1 principes généraux 181
B.1.1 Représentation d'un environnement 181
B.1.2 Notation 182
B.1.3 Exemple d'utilisation du partage de structures 182
B.2 Backtrack et pile de trainée 189
B.3 Implantation des variables 190

Liste des figures

2.1 Fonctionnement du moteur PROLOG 31
2.2 Règles de résolution TSLD 43

4.1 Cycle du moteur d'inférence 58

5.1 Graphe de règles 66
5.2 71

6.1 Architecture du système 83
6.2 Conception du moteur d'inférence 95

7.1 Distribution vers plusieurs processeurs 103
7.2 Distribution vers un seul processeur 104
7.3 Réseau en anneau 105
7.4 Réseau "du haut vers le bas" 106
7.5 Réseau en étoile 106
7.6 Réseau totalement interconnecté 107

8.1 Architecture détaillée des paquetages 114
8.2 Implantation d'une clause 120

10.1 Exécution de la forme 2 145
10.2 Exécution de la forme 2 147

A.1 Piles sans éléments dupliqués 178

B.1 Environnement de la question 182
B.2 Environnement de la question et de la première clause 183
B.3 Environnement de la question et de la première clause 184
B.4 Environnement de la question et de la première clause 184
B.5 Environnement de la question et de la première clause 184
B.6 Environnement de la question et de la première clause 185
B.7 Environnement de la question et des deux clauses 186
B.8 Environnement de la question et des deux clauses 186
B.9 Environnement de la question et des deux clauses 187
B.10 Environnement de la question et des deux clauses 187
B.11 Environnement de la question et des deux clauses 188
B.12 Environnement de la question et des deux clauses 189
B.13 Environnement après backtrack 190

Liste des tableaux

5.1 Système élémentaire 65
5.2 Développement et indexation des règles de S1 69
5.3 Règles □ contre ◇ 71
5.4 Implantation du KUT 73

6.1 Tableau des règles possibles 92
6.2 Matrice de changement d'états du moteur 96

7.1 Règles □ – ◇ 102

9.1 Temps CPU et système utilisé dans un réseau "du haut vers le bas" 139
9.2 Temps CPU et système utilisé dans un réseau "du haut vers le bas" 139

11.1 Règles de S1 155
11.2 Développement des règles du KUT KUT 161
11.3 Influence de l'ordre des règles 165
11.4 Logique des modules 166
11.5 Formes pour la logique des modules 166
11.6 Logique floue 168

Introduction

Dans le but de se rapprocher le plus possible du raisonnement humain, les systèmes de programmation logique doivent être capables de traiter différents concepts comme le temps, la croyance, la connaissance, etc...

Prolog est parfaitement adapté au traitement des clauses de Horn de la logique du premier ordre, mais qu'en est-il si nous souhaitons traiter des problèmes de logique non-classiques, organiser notre programme en modules, traiter des théories équationnelles comme les prédicats flous? Tous ces différents cas réclament des moyens différents de calculer un nouveau but à partir du but courant.

Des solutions théoriques ont été développées pour chacun des cas énumérés ci-dessus, et des extensions particulières ont été largement développées dans la littérature ([BK82], [GL82], Tokio [FKTMO86], N-PROLOG [GR84], Context Extension [MP88], Templog [Bau89a], Temporal Prolog [Sak89] [Sak87]).

Chacune de ces solutions implante un méta-interprète PROLOG, spécifique au système qu'il va traiter. Mais il y a de nombreux inconvénients à la technique des méta-interprètes : ils sont extrêmement lents, et notoirement inefficaces pour la compilation. Pour aller plus loin, et développer des extensions de PROLOG efficaces, il faut payer un prix élevé, car chaque cas conduit à développer un système différent : si nous développons un système pour le raisonnement temporel, nous ne pourrons pas faire de raisonnement sur le savoir ou la connaissance.

Le but est donc de définir un mécanisme général dans lequel l'utilisateur peut développer "son" extension de Prolog. Ce travail s'appuie sur quatre principes :

1. Nous conservons comme base les mécanismes fondamentaux de la programmation logique classique : recherche en profondeur d'abord, backtracking, unification.

2. Le pas d'inférence, qui définit comment un nouveau but est calculé à partir du but courant, doit être complètement paramétrable par l'utilisateur : la logique classique n'apparaîıt plus que comme un cas particulier du formalisme général de notre système.

3. Il doit être également possible de paramétrer des opérations de réécriture sur les buts calculés.

4. Le système doit être *efficace*.

Les trois premiers points furent les éléments fondateurs d'un méta-interprète général, MOLOG ([FdC86], [ABFdC+86], [dCH88], [Esp87b], [Esp87a]). Nous présentons ici les différents principes et techniques développés ([Bri87], [AG88], [AHLM92]) pour atteindre le but fixé par le point 4 :

- Réécriture complète des mécanismes d'inférence et développement d'un nouveau formalisme plus adapté au traitement des clauses de Horn généralisées.

- Développement d'une machine abstraite.
- Etude et implantation du parallélisme.

Ce document est divisé de la façon suivante :

- Dans la première partie, nous nous situerons dans le cadre général, à la fois théorique et pratique, de la programmation logique.
- Dans la deuxième partie nous développerons en détail les spécifications du système : formalisme, objets manipulés, moteur d'inférence, notion de contrôle.
- Dans la troisième partie, nous présenterons les choix de conception effectués.
- Dans la quatrième partie, nous détaillerons l'implantation des versions séquentielles et parallèles.
- Dans la cinquième partie nous présenterons quelques exemples d'application (concepts modaux, flous, possibilistes...)

Ce document étant également destiné à servir de référence technique pour les personnes appelées à poursuivre ce travail, il était nécessaire de détailler certains points techniques. Afin de ne pas rendre la lecture trop difficile, ces discussions ont été renvoyées en annexe.

Partie I

Fondements

Chapitre 1

Déduction et résolution

Nous supposons connu dans ce chapitre l'ensemble des résultats concernant le calcul propositionnel et le calcul des prédicats (théorie des modèles, théorie de la démonstration, adéquation, complétude, décidabilité).

Nous allons rapidement reprendre les fondements théoriques de la programmation logique (clauses de Horn, principe de déduction, principe de résolution, résolution SLD...) puis étendre ces notions pour arriver à la définition des clauses de Horn généralisées et au principe de paramétrisation de la résolution qui sont à la base de TARSKI[1].

1.1 Rappels

1.1.1 Clauses de Horn

Définition 1.1 (Clauses de Horn) *Une clause de Horn est une disjonction de littéraux* $p_1 \vee p_2 \ldots \vee p_n$ *comportant au plus un littéral positif.*

Il existe trois types de clauses de Horn :

- *Celles comportant un littéral positif et au moins un littéral négatif, appelées clauses de Horn* strictes.
- *Celles comportant exactement un littéral positif et aucun littéral négatif, appelées clauses de Horn* positives.
- *Celles ne comportant que des littéraux négatifs, appelées clauses de Horn* négatives.

1.1.2 Principe de déduction

Le principe de déduction ramène toute preuve de déduction à une preuve d'*inconsistance*, permettant de se concentrer uniquement sur les mécanismes de preuve de consistance et de validité.

Théorème 1.1 (Principe de déduction) *Si* A *est la conséquence valide d'un ensemble* S *de formules alors* $S \cup \{\neg A\}$ *est inconsistant.*

[1] Toulouse Abstract Reasonning System for Knowledge Inference

1.1.3 Principe de résolution

Le principe original de résolution est de Robinson [Rob65].

Théorème 1.2 (Principe de résolution) *Soit N une forme normale, C_1 et C_2 deux clauses de N. Soit p un atome tel que $p \in C_1$ et $\neg p \in C_2$. Soit la clause $R = C_1 \setminus \{p\} \cup C_2 \setminus \{\neg p\}$. Alors les formes normales N et $N \cup R$ sont logiquement équivalentes.*

La clause R définie ci-dessus est appelée *résolvante* de C_1 et C_2.

La condition énoncée par le principe de résolution est nécessaire et suffisante. (c.a.d un ensemble de clauses est inconsistant si et seulement si la clause f[2] figure dans l'ensemble des conséquences logiques de notre ensemble de clauses).

1.1.4 Forme procédurale du principe de résolution

Appliqué aux clauses de Horn, le principe de résolution à une forme procédurale simple :

1. Si f est dans N, l'ensemble est inconsistant et la résolution est terminée.
2. Choisir une clause C et une clause P telles que P est une clause de Horn positive réduite à p et C une clause contenant $\neg p$.
3. Calculer la résolvante R de P et C.
4. Reprendre l'algorithme en remplaçant N par $(N \setminus \{C\}) \cup R$.

1.1.5 Résolution SLD

Il y a plusieurs façons d'appliquer mécaniquement le principe de résolution de Robinson. Nous allons décrire ici la méthode appelée résolution-SLD par [AvE82]. SLD signifie *Linear resolution with Selection function for Definite clauses* ([KK71]).

La méthode SLD est à la base de la résolution PROLOG, nous allons donc nous y attarder.

Définition 1.2 (Règle de calcul) *On appelle règle de calcul une fonction R définie d'un ensemble de clauses de Horn N négatives (buts) vers un ensemble d'atomes, telle que la valeur de la fonction pour une clause N_k de N est un atome p tel que p est dans N_k.*

Définition 1.3 (Résolvante) *Considérons un ensemble P de clauses de Horn C_k positives ou strictes. Soit G_i une clause de Horn négative (but). Prenons $G_i = \neg A_1 \vee \ldots \vee \neg A_m \vee \ldots \vee \neg A_p$ et $C_k = \neg B_1^k \vee \ldots \vee \neg B_q^k \vee A^k$. On dit alors que G_{i+1} est dérivé de (est la résolvante de) C_k et de G_i via la règle R (définie ci-dessus) si :*

1. *A_m est l'atome sélectionné par R pour G_i.*
2. *$A_m = A^k$*
3. *$G_{i+1} = \neg A_1 \vee \ldots \vee \neg A_{m-1} \vee \neg B_1^k \vee \ldots \vee \neg B_q^k \vee \neg A_{m+1} \vee \ldots \vee \neg A_p$*

[2] La clause f est la clause *false*.

Définition 1.4 (Dérivation SLD) *Soit P un ensemble de clauses de Horn positives ou strictes. Soit G une clause de Horn négative. Une dérivation SLD de de $P \cup \{G\}$ par R est la suite finie ou infinie $G_0 = G, G_1, \ldots$ et une suite de clauses de P $C_1, C_2, \ldots$ telles que G_{i+1} est dérivée de G_i et C_{i+1} par R.*

Définition 1.5 (Réfutation SLD) *Une réfutation SLD est une dérivation SLD finie telle que $G_N = \emptyset$. On dit alors que la réfutation est de longueur n.*

La recherche d'une réfutation SLD est donc identique à un parcours d'arbre. De chaque nœud de l'arbre peuvent partir plusieurs branches, qui correspondent aux différentes clauses sélectionnables pour l'atome A_m de G sélectionné par R.

Un résultat intéressant est l'indépendance de la règle R pour la résolution[3]. PROLOG utilise comme règle de sélection la règle dite *leftmost first*, ou *plus à gauche en premier*.

Ce qui n'est, en revanche, pas indifférent, c'est la stratégie de parcours de l'arbre. S'il existe, pour un programme P et un but G une résolution SLD fini, alors un parcours de l'arbre en largeur d'abord la trouvera, alors qu'un parcours en profondeur d'abord peut se perdre dans une branche infinie. PROLOG utilisant en général des stratégies en profondeur d'abord détruit la complétude de la résolution SLD.

1.1.6 Nouvelle formalisation du principe de résolution

Nous pouvons utiliser le formalisme du calcul des séquents de Gentzen pour décrire le mécanisme de résolution SLD. Nous verrons que ce formalisme est particulièrement bien adapté au mécanisme de TARSKI.

La construction d'une résolvante SLD s'écrit alors :

$$A \vee L, ?B \vee \neg L \vdash ?A \vee B$$

Nous pouvons lire ceci : si la clause est $A \vee L$ et le but $B \vee \neg L$ alors la résolvante sera $A \vee B$.

En fait, suivant le même schéma, il est possible de définir récursivement la construction de la résolvante SLD en appliquant les règles de résolution suivantes[4] :

- $$\frac{E \vee A, ?B \vee D \vdash ?E \vee C \vee D}{A, ?B \vdash ?C}$$

  Cette règle se lit : La résolvante de la clause $E \vee A$ et du but $B \vee D$ est $E \vee C \vee D$ si la résolvante de A et de B est C.

- $$p, ?\neg p \vdash ?\emptyset$$

  Cette règle se lit : la résolvante de p et $\neg p$ est la clause vide.

- Les deux règles précédentes génèrent des résolvantes de la forme $A \vee \emptyset$. Pour parvenir à conclure la résolution, nous devons simplifier la forme $A \vee \emptyset$ en A. Nous devons donc adjoindre la règle de simplification[5] :

  $$A \vee \emptyset \rightsquigarrow A$$

[3] Pour l'ensemble des résultats théoriques concernant la résolution SLD aussi bien en calcul propositionnel qu'en logique des prédicats, voir [Llo84].

[4] $\emptyset$ représente ici la clause vide.

[5] nous parlerons plus tard de *règle de réécriture*

Cette définition récursive de la construction de la résolvante, généralement sous-entendu dans toute résolution SLD "à la PROLOG" est **la base du fonctionnement de** TARSKI. En effet, le principe fondamental de TARSKI est la possibilité de pouvoir paramétrer les règles de résolution. Nous allons revenir sur ce point dans la section suivante.

1.1.7 Extension au calcul des prédicats

Les résultats précédents s'étendent aux clauses de Horn en logique du premier ordre en introduisant les notions de substitution et d'unification[6].

1.1.7.1 Substitution

Intuitivement, substituer un terme t_1 à un terme t_2 dans une formule F consiste simplement à remplacer le terme t_2 par le terme t_1.

Formellement, on pose :

Définition 1.6 (Substitution) *Une substitution est une application de l'ensemble des variables dans l'ensemble des termes. On notera généralement σ une substitution.*

Définition 1.7 (Instance) *Le terme $\sigma(t)$ obtenu en remplaçant dans le terme t toutes les variables x_i par $\sigma(x_i)$ est une instance de t. Un terme t_2 est une instance de t_1 ssi il existe une substitution σ tel $t_2 = \sigma(t_1)$. On note alors $t_2 \prec t_1$.*

Définition 1.8 (Terme complètement instancié) *Un terme t est dit complètement instancié ssi il ne possède aucune variable.*

Remarquons que la relation $\prec$ que nous avons définie est une relation d'ordre partiel sur l'ensemble des termes T, au renommage des variables libres près[7].

Définition 1.9 (Plus grande borne inférieure) *Considérons deux termes s et t, et les ensembles $S = \{s_i \mid s_i \prec s\}$ et $T = \{t_i \mid t_i \prec t\}$.*

Si ces ensembles ont une intersection $S \cap T$ non vide, alors cet ensemble est totalement ordonné pour $\prec$ et admet un plus grand élément car il est discret et borné supérieurement. On appelle cet élément la plus grande borne inférieure de la paire (s, t).

Voyons sur un exemple à quoi correspond ce théorème. Soit les termes :

- $t = f(x, y, g(a))$
- $s = f(z, g(b), w)$

La plus grande borne inférieure de s et t est le terme $r = f(x, g(b), g(a))$. Notons que nous avons bien $r \prec t$ et $r \prec s$. Remarquons également que l'ensemble $S \cap T$ comprend d'autres éléments tels $f(a, g(b), g(a))$. Notons également que les termes t et s ne sont pas comparables par $\prec$[8].

[6] Nous ne revenons pas sur les problèmes de skolémisation et supposons connu le théorème montrant l'équivalence de la consistance d'une formule et de sa forme de Skolem.

[7] Une construction rigoureuse demanderait que l'on définisse une relation d'équivalence $\approx$ sur T telle que $t_1 \approx t_2$ ssi il existe un renommage des variables qui transforment t_1 en t_2. Nous pourrions alors travailler en toute rigueur sur l'ensemble quotient $T/\approx$.

[8] S'ils l'étaient, la plus grande borne inférieure serait évidemment un des deux termes !

1.1.7.2 Unification

Définition 1.10 (Littéraux unifiables) *Deux littéraux sont dits unifiables s'ils possèdent une instance fondamentale commune.*

Rien de remarquable dans cette définition.

Nous allons maintenant établir le principe de résolution sur les instances fondamentales :

Théorème 1.3 (Principe de résolution) *Soit N une forme normale, C_1 et C_2 deux clauses de N. Soit $l_1 \in C_1$ et $\neg l_2 \in C_2$, tels que l_1 et l_2 soient unifiables.*

Soit la clause $R' = C_1 \setminus \{l'\} \cup C_2 \setminus \{\neg l'\}$ où l' est une instance commune quelconque de l_1 et l_2. Alors les formes normales N et $N \cup R'$ sont logiquement équivalentes.

Ce théorème se déduit immédiatement du théorème de résolution pour le calcul propositionnel.

Nous allons prendre comme clause l' la plus grande borne inférieure de l_1 et l_2. Nous savons alors qu'il existe une substitution σ telle que $l' = \sigma(l_1) = \sigma(l_2)$.

Soit la clause R définie alors par :

$$R = \sigma(C_1) \setminus \{l'\} \cup \sigma(C_2) \setminus \{\neg l'\}$$

La clause R est alors une solution satisfaisante. Intuitivement, disons que nous lions de la façon la plus générale possible les variables des deux littéraux entre elles.

1.1.7.3 Algorithme de résolution

L'algorithme de résolution s'étend également très aisément :

Soit N une forme normale :

1. Si $f \in N$, alors la résolution est terminée.
2. Prendre deux clauses c_1 et c_2 de N, telles que $l_1 \in c_1$, $\neg l_2 \in c_2$ et l_1, l_2 sont unifiables.
3. Trouver la plus grande borne inférieure de l_1 et l_2, la substitution σ et calculer la clause résolvante r[9].
4. Remplacer N par $N \cup \{r\}$.

Bien entendu, ceci s'étend immédiatement au formalisme de Gentzen tel que nous l'avons présenté tout à l'heure.

1.2 Extensions des clauses de Horn et de la résolution SLD

La base de notre langage reste celle de PROLOG. Cependant, ce langage doit être étendu par des *opérateurs de contexte*, afin de permettre de traiter d'autres logiques que la logique classique. Nous allons présenter dans cette section les principes théoriques généraux, le chapitre 3 développant précisément chacun de ces points.

[9] Signalons tout de même un "petit problème" bien connu de tous les gens qui s'intéressent à la résolution automatique. Supposons que nous ayons à unifier des termes x et $f(x)$. L'algorithme d'unification donne $f(f(f(f(\ldots(x)\ldots))))$. Notons qu'en toute rigueur, il faudrait vérifier avant chaque unification que l'on ne risque pas de se trouver dans une situation de ce type : i.e. que l'on n'essaie pas d'unifier une variable x avec un terme contenant x mais ne se limitant pas à x. Ce test, qu'il faudrait effectuer, s'appelle *test d'occurrence* (occur-check en anglais). Nous aurons l'occasion d'en reparler dans le chapitre consacré aux implantations classiques de PROLOG.

1.2.1 Clauses de Horn généralisées

Nous conservons les définitions usuelles de *termes* et de *formules atomiques* utilisées en programmation logique. Intuitivement, les clauses de Horn généralisées sont construites de façon à ce que, si nous supprimons les contextes, nous retrouvions les clauses de Horn traditionnelles.

Formellement, nous pouvons définir récursivement notre langage par :

Définition 1.11 (Contexte) $\mathtt{m}(\mathtt{t}_1, ..., \mathtt{t}_n)$ *est un contexte si* $\mathtt{m}$ *est un opérateur de contexte,* $n \geq 0$*, et* $\forall i, 1 \leq i \leq n$ *chaque* t_i *est soit un terme soit une clause de Horn généralisée non négative.*

Définition 1.12 (Clauses de Horn généralisées négatives (but)) *On note* N *l'ensemble des clauses modales de Horn négatives. On a :*

- *Si* p *est une variable propositionnelle, alors* $\neg p \in N$*.*
- *Si* $F \in N$ *et* $G \in N$ *alors* $(F \vee G) \in N$
- *Si* $F \in N$ *alors* $MOD : F \in N$ *si* MOD *est un contexte.*

Définition 1.13 (Clauses de Horn généralisées positives) *On note* P *l'ensemble des clauses de Horn positives. On a :*

- *Si* p *est une variable propositionnelle alors* $p \in P$
- *Si* $F \in P$ *alors* $MOD : F \in P$ *si* MOD *est un contexte.*

Définition 1.14 (Clauses de Horn généralisées strictes) *On note* R *l'ensemble des clauses de Horn strictes. On a :*

- *Si* $F \in R$ *et* $G \in N$ *alors* $F \vee G \in R$
- *Si* $F \in R$ *alors* $MOD : F \in R$ *si* MOD *est un contexte.*

Ceci définit complètement l'ensemble des clauses sur lesquelles nous allons opérer.

1.2.2 Extension de la résolution SLD

Nous détaillerons très largement les techniques utilisées pour étendre la résolutions SLD dans les chapitres 3 etc 4. Nous allons ici nous contenter de présenter le principe général.

Une résolution SLD étendue se déroule en 5 étapes :

Sélection de clause : Une clause est sélectionnée pour résoudre le premier sous-but de la question.

Sélection de règle : Une règle est sélectionné pour poursuivre la résolution.

Exécution de la règle : La règle qui a été sélectionnée est exécutée. L'exécution de la règle modifie la clause courante, la question courante et construit la résolvante.

Réécriture de la résolvante : Lorsqu'un premier sous-but est résolu, il faut reconstruire la question à partir de la résolvante.

Fin de la résolution : La résolution est terminée lorsque l'on attend le but *true*.

Comparons à la résolution SLD classique. Nous n'introduisons aucune nouvelle étape, nous nous contentons d'expliciter certaines étapes implicites dans la résolution en logique classique :

- La sélection de règle existe, mais elle est implicite. Il existe en effet deux règles utilisées en logique classique comme nous l'avons fait remarquer :

$$\frac{E \vee A, ?B \vee D \vdash ?E \vee C \vee D}{A, ?B \vdash ?C}$$

  Rappelons que ceci se lit : La résolvante de la clause $E \vee A$ et du but $B \vee D$ est $E \vee C \vee D$ si la résolvante de A et de B est C.

  Nous avons d'autre part la règle :

$$p, ?\neg p \vdash ?\emptyset$$

  qui se lit : la résolvante de p et $\neg p$ est la clause vide.

  Le choix est simplement fait de façon "invisible" pour l'utilisateur. Une seule règle est applicable dans un cas précis et PROLOG la choisit "silencieusement".

- L'exécution des règles existe bien entendu. Lorsqu'une règle est choisie, le système l'exécute construisant ainsi la résolvante à partir du fait et du but courant.

- Il existe une règle de réécriture implicite en logique classique :

$$A \vee \emptyset \rightsquigarrow A$$

  Cette règle est également appliquée implicitement par la résolution SLD classique pour supprimer de la résolvante le but $\emptyset$.

Chapitre 2

Modèles et implantations existants

TARSKI se trouve au confluent de trois domaines ; la programmation logique, la programmation logique parallèle et la méta-programmation. Le but de ce chapitre est de faire un point rapide sur ces trois domaines pour replacer TARSKI dans son environnement général, aussi bien au niveau des principes qu'au niveau de l'implantation.

2.1 Implantations "classiques" de PROLOG

Nous allons dans cette section décrire succinctement les implantations de PROLOG les plus classiques. Cela nous permettra de présenter les techniques générales utilisées pour toute implantation d'un système de programmation logique.

Nous ne discuterons pas des implantations plus récentes utilisant des algorithmes de résolution sous contraintes comme CLP(R) ([HMS89]) ou PROLOG III, car TARSKI n'utilise aucune de ces techniques.

Nous discuterons brièvement des principes de compilation dégagées par la machine de Warren ; en effet, si TARSKI **ne fait pas de la compilation de clauses**, elle fait de la compilation de règles de résolutions, et elle s'inspire (très librement) pour ce faire des techniques définies par Warren.

Je me suis appuyé pour écrire cette section sur les différents articles et livres de l'équipe du GIA de Luminy ([vC86], [GKPC85], [CKvC79], [Col75], [Col84]) pour les PROLOG de Marseille.

Pour les PROLOG de la famille Edimburgh, j'ai eu directement accès aux sources de C-PROLOG, qui furent d'ailleurs à la base du développement de la première version de TARSKI en C.

Enfin, l'article élémentaire mais très clair de Marti Pentonen ([Pen85]) fut pour moi une excellente introduction aux techniques de base des implantations PROLOG. Je me suis également fréquemment servi de [Cam84].

2.1.1 Historique

PROLOG a vu le jour en 1971 sur une idée d'Alain Colmerauer qui travaillait alors aux problèmes de traduction automatique. Le premier langage PROLOG a été créé par Alain Colmerauer avec l'aide de Philippe Roussel dans le cadre d'un contrat de recherche sur la communication homme-machine. La syntaxe de ce premier PROLOG est assez différente de

ce qu'elle sera par la suite ; la notion de *cut* n'existait pas encore ; en revanche, il traitait déjà le problème du test d'occurrence, et faisait de l'évaluation différée. Il était écrit en Algol-W sur IBM 360-37. Pour la petite histoire, le nom Prolog fut inventé par la femme de Philippe Roussel.

PROLOG-I fut développé à partir du PROLOG original dès 1972. Philippe Roussel introduisit, sur une idée de Boyer et Moore, la technique de partage de structures, la syntaxe fut modifiée et la forme des clauses se rapprocha de celle que nous connaissons aujourd'hui. PROLOG-I fut implanté pour la première fois en FORTRAN en 1973. La restriction aux clauses de Horn ainsi que la sémantique de PROLOG furent établies par R. Kowalski et M. van Emden en 1974 ([KvE74]). Le manuel de PROLOG-I dans sa version"définitive" fut terminée en 1975.

C'est en 1974 que David Warren s'intéresse pour la première fois à PROLOG pour la programmation de son système Warplan. En 1977, il développe le premier "compilateur" PROLOG [War77] avec Fernando Pereira et Luis Moniz Pereira[PPW79]. Il introduira la syntaxe dite syntaxe Edimburgh, reprise par la plupart des PROLOG aujourd'hui.

A partir de 1978 les implantations de PROLOG vont se multiplier, en Angleterre à l'université d'Edimburgh et à l'Imperial College de Londres, en Belgique avec les travaux de G. Roberts et M. Bruynooghe, en Hongrie avec Peter Szeredi. Il est intéressant de remarquer que David Warren et Peter Szeredi travaillent maintenant, soit 14 ans plus tard, ensemble à l'université de Bristol sur le projet Gigalips.

David Warren définira la machine abstraite qui porte son nom en 1983 ([War83]). Il s'agit d'une amélioration de son travail sur le compilateur fait en 1977. Cette machine reste aujourd'hui la référence en matière de compilation PROLOG. Elle a donnée naissance à Quintus Prolog, SB-prolog, SWI-prolog et bien d'autres implantations.

En France, PROLOG II a été développé à la suite de PROLOG-I par l'équipe d'Alain Colmerauer, et plus particulièrement implanté par Michel van Caneghem dans les années 1980-1983 ([vC86]). Il s'agit de la première version véritablement commerciale des PROLOG marseillais et elle connaîitra une grande diffusion en France dans les universités et les centres de recherche.

Depuis ces temps héroïques, les implantations de PROLOG se sont multipliés et la recherche s'est divisée en plusieurs sous-domaines : compilation, implantations parallèles, algorithmes d'unification par contraintes...Les décrire toutes demanderait probablement une encyclopédie. Nous allons nous concentrer sur les techniques de base.

2.1.2 Les PROLOG interprétés

2.1.2.1 Description sommaire du fonctionnement du moteur d'inférence

La figure 2.1 montre le fonctionnement général de tout moteur PROLOG "standard". Rappelons les étapes fondamentales :

1. On prend dans la question courante (qui est une clause de Horn négative, donc ne contenant que des littéraux négatifs) le **premier** littéral négatif.

2. On parcourt la base de clauses (faits et règles) **séquentiellement** depuis la **première** jusqu'à ce que l'on ait trouvé une clause dont le littéral positif, appelé aussi littéral de tête de la clause, est **unifiable** avec le littéral négatif que nous avons sélectionné. Si l'on

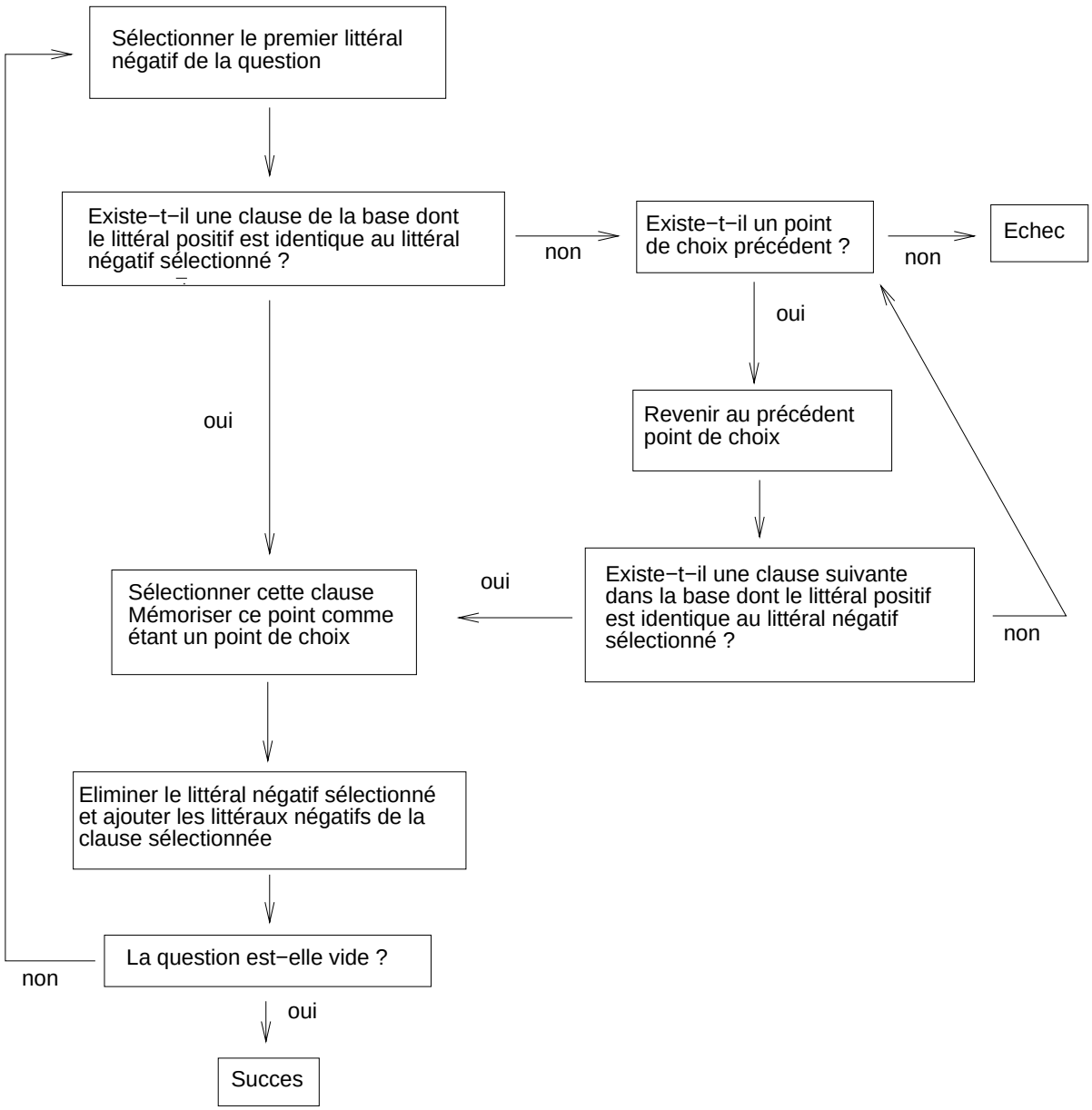


Figure 2.1: Fonctionnement du moteur PROLOG

ne peut plus en trouver, on effectue un **retour arrière** ou **backtrack**[1] en anglais. S'il n'est plus possible de backtracker, la résolution est terminée et l'on a échoué.

3. On élimine de l'ensemble des littéraux négatifs qui composaient la question le littéral négatif sélectionné, et l'on rajoute l'ensemble des littéraux négatifs de la clause sélectionnée (ensemble qui peut-être vide si cette clause est une clause de Horn positive). Le nouvel ensemble ainsi constitué est la nouvelle question.

4. Si la question est réduite à la clause vide, la résolution est terminée et l'on a réussi. Sinon, on revient à l'étape 1.

Nous allons dans la section suivante discuter en détail les points délicats de l'implantation du moteur décrit ci-dessus et les différentes solutions qu'il est possible d'adopter pour les résoudre.

2.1.2.2 Problèmes classiques pour l'implantation de PROLOG

Il est possible de réaliser des implantations de PROLOG extrêmement simples dans certains langages bien adaptés à la manipulation d'objets symboliques. Ainsi, il est possible d'écrire un interprète PROLOG complet en LISP en moins de 100 lignes ([Nil84]). Cependant, de semblables interprètes sont inefficaces et reportent toute la complexité dans le langage d'implantation (ainsi, par exemple, le problème du "ramasse-miette"est-il laissé aux bons soins de l'interprète LISP). Nous nous intéressons ici aux techniques d'implantations dans les langages procéduraux classiques.

Copie de clauses : Le premier problème difficile est la copie de clause, et le choix des structures de données pour la réaliser. Rappelons qu'à chaque sélection d'une nouvelle clause dans la résolution, il faut avoir une nouvelle copie de cette clause, car les valeurs des variables de la clause sont, bien entendu, locales à cette étape de la résolution (il ne s'agit pas de fixer une fois pour toute par l'unification la valeur des variables de cette clause pour toutes les futures sélections possibles), et l'on doit rajouter dans la résolvante la liste des buts de la partie droite de la clause avec les variables unifiées aux nouvelles valeurs.

Deux écoles sont en présence, le *partage de structures* (structure-sharing) et la *copie de structures* (structure-copying). En fait, ces mots sont relativement imprécis, et il existe des techniques de partage de structures qui font de la recopie. Détaillons les techniques les plus classiques :

La recopie totale : Cette méthode consiste à recopier l'intégralité de la clause telle qu'elle est. Cette technique est appelée *goal-stacking* par Warren, ou empilement des buts en français : en effet, la partie droite de la clause est intégralement recopiée avec les nouvelles valeurs des variables après unification. Cette technique peut apparaîitre comme naïve, et l'auteur de ces lignes la méprisa longtemps[2]. Il avait tort, et aurait mieux fait de se souvenir de la remarque de Michel van Caneghem :

> Je pense que dans certains cas cette technique peut se révéler intéressante. Dans le cas où chaque règle n'est utilisée qu'une fois, il n'y a pas lieu

[1] Certains PROLOG effectuent aussi un retour-arrière après un succès, pour chercher d'autres solutions (c'est le cas de PROLOG-II).

[2] Il n'était pas le seul d'ailleurs.

de faire de copie. **Dans celui où l'on veut faire un contrôle subtil sur l'ordre d'effacement des buts, il peut être intéressant d'avoir explicitement une liste des buts à effacer.**[3]

La dernière phrase s'applique parfaitement à TARSKI, à condition de remplacer "contrôle d'effacement" par "contrôle sur la construction". Ainsi, le choix fait pour TARSKI du partage de structures, efficace et rapide, est cependant discutable. Il aurait peut-être mieux valu adopter la technique de copie totale, car on aurait ainsi évité les contraintes sur la forme des règles de résolution. Nous y reviendrons.

Le partage de structures : Le partage de structures est une technique à la fois très simple et très uniforme dans sa mise en œuvre informatique, et pourtant peu compréhensible au premier abord. Nous la décrivons rapidement car c'est la technique adoptée pour TARSKI.

Supposons que le but soit `p(X,q(b))` et que nous sélectionnons la nouvelle clause :

```
p(a,X):-r(Y),t(Y,X).
```

Dans le partage de structures, un environnement est créé à chaque utilisation d'une clause. Cet environnement permet de créer une nouvelle instance de chaque variable de la clause. L'environnement est alloué sur une pile spéciale, appelée pile des environnements, ou pile des valeurs. Chaque élément de cette pile représente la valeur d'une variable et est un couple (Structure, Environnement) Un environnement pour une clause est composé d'autant de couples (Structure, Environnement) qu'il y a de variables dans la clause. L'élément Structure est un pointeur sur l'objet auquel est unifié la variable. L'élément Environnement est l'environnement dans lequel il faut évaluer la structure. Cet algorithme est décrit en détail dans l'annexe B.

La recopie simple : Cette technique est utilisée en particulier dans la machine de Warren. Chaque fois que l'on unifie une variable avec un terme, on recopie ce terme dans une pile appelée pile de recopie. Au cours de la recopie de ce terme, si on rencontre une variable, on la remplace par sa valeur. Si cette variable n'a pas de valeur, on crée une nouvelle case dans la pile de recopie et on lie la variable libre à cette nouvelle case.

Cette méthode, si elle permet un meilleur accès aux termes, rend l'interpréteur plus complexe. C'est pour cette raison que nous l'avons abandonnée.

Unification et test d'occurrence : Le principe général de PROLOG est basé sur l'algorithme de Robinson. Cependant, pour que la résolution PROLOG soit complète et consistante, il faut effectuer ce que l'on appelle l'*occur check* ou test d'occurrence en Français. Ceci consiste à vérifier en particulier que l'on essaie jamais d'unifier deux termes comme X et $f(X)$, ce qui conduit à un terme infini $f(f(f(.....(X)..)))$. Il existe plusieurs solutions pour régler ce problème :

1. Effectuer le test d'occurrence. Cela conduit à un ralentissement considérable de l'interprète PROLOG (50% dans les meilleurs cas), ce qui est difficilement admissible.
2. Accepter la notion d'arbres infinis et l'intégrer dans le langage. C'est le choix fait par l'équipe de Marseille pour tous ces PROLOGs.

[3] C'est moi qui souligne.

3. Ignorer le problème. C'est le choix fait par tous les PROLOGs de la famille Edimburgh. C'est aussi le choix le plus simple, et c'est celui que nous avons fait pour TARSKI.

Algorithmes d'unification : Il existe deux grandes classes d'algorithmes d'unification : ceux dérivés de l'algorithme de Robinson original et les systèmes d'unification par contraintes. Nous ne nous sommes pas intéressés à la seconde catégorie (utilisée dans PROLOG-III et CLP(R) par exemple). Rappelons brièvement le mécanisme de l'algorithme de Robinson que nous avons choisi : pour unifier deux termes, il commence par déréférencer récursivement chacune des variables et unifie les termes déréférencés : si ce sont deux variables, on lie la plus récente à la plus ancienne, s'il s'agit d'une variable et d'un atome, on lie la variable à l'atome.

Retour arrière et gestion des points de choix : Le retour arrière est effectué à l'aide d'une pile appelée *pile de retour-arrière* ou *pile de backtrack*. Pour chaque point de choix, on stocke dans cette pile l'ensemble des informations nécessaires à la poursuite de la résolution en cas de retour-arrière : la valeur des sommets des piles, et en particulier la valeur du pointeur de la pile de traîinée lorsque l'on utilise le modèle de partage de structures[4].

Sélection et indexation des clauses : La sélection des clauses est une étape coûteuse dans la résolution si on l'effectue avec le modèle naïf : recherche de la première clause de la base dont la tête est la même que celle de la question courante, puis en cas de backtrack, recherche par le même mécanisme de la clause suivante.

On peut implanter plusieurs optimisations pour améliorer cet état de fait. La plus ancienne consiste à chaîiner toutes les clauses qui ont le même prédicat de tête. Ainsi, lorsque l'on cherchera la clause suivante, il suffira de prendre l'élément suivant dans la liste chaîinée. Une autre technique classique, pour accélérer la recherche de la clause initiale, est l'utilisation d'une table de hachage (hash-table) sur les noms de prédicats qui, même dans le cas de grands ensembles de prédicats, permettra de trouver la clause initiale en quelques étapes seulement.

Ces deux optimisations sont implantées dans TARSKI.

Il existe une troisième optimisation, adoptée par Warren, qui consiste à indexer la recherche de la clause sur le nom du prédicat de tête **et** sur la valeur d'un ou des arguments de ce prédicat, ce qui améliore l'efficacité quand le ou les arguments sont des constantes. Cette optimisation n'a pas été retenue pour l'instant.

Gel de l'évaluation : La notion de gel de l'évaluation est liée aux PROLOG de Marseille, et a trouvé son prolongement dans les algorithmes d'unification sous contraintes. Geler un prédicat consiste à attendre pour l'évaluer qu'une variable libre ait été liée à une valeur. Ainsi, en PROLOG-II, le prédicat **dif(x,y)** qui est vrai si x est différent de y ne sera-t-il évalué que quand x et y auront reçu une valeur[5]. Cette technique n'a pas été implantée dans TARSKI.

[4] La pile de traîinée à pour fonction de conserver la trace des variables qui sont liées lors d'une unification. En cas de retour arrière, il ne faut pas oublier de marquer à *libre* les variables référencées dans cette pile. L'annexe B détaille le fonctionnement de cette pile.

[5] On trouve également en PROLOG-II le prédicat **geler**, qui permet de geler l'évaluation de son argument.

Ramasse-miettes : Tous les PROLOG sont gros consommateurs de mémoire, et les premières implantations de PROLOG furent réalisées sur des machines disposant de (relativement) peu de mémoire. On s'est donc intéressé aux différentes façons de récupérer la mémoire inutilisée.

Il faut tout d'abord souligner qu'en PROLOG, contrairement à LISP, la mémoire finit toujours par être restituée au moment du backtrack. Les problèmes de récupération de mémoire sont donc, en principe, moins critiques.

Beaucoup de techniques (distinction entre variables muettes, temporaires, locales, globales, optimisation de la récursivité terminale) ont été développées par Warren et sont maintenant implantées sur de nombreux PROLOG. Cependant, plus que de véritables techniques de GC, il s'agit plutôt d'astuces opérant sur les instances des clauses.

Il existe des techniques plus globales développées principalement par [Bru82] ; Dans l'une d'entre elles, on parcourt la liste des buts à effacer ainsi que la liste des buts en attente et l'ensemble des structures associées pour marquer les variables que l'on peut atteindre à partir de ces buts. On peut alors supprimer les variables non-marquées. Cependant, la récupération de l'espace est difficile aussi bien dans le modèle de copie de structures que dans le modèle de partage de structures. En effet, la plupart des références sont des références relatives et la suppression d'un élément dans la pile modifiera l'ensemble des références. Enfin, plus récemment, les problèmes de GC ont conduit à la mise au point de la machine MALI ([Rid92],[BCRU86],[BCRU84])[6].

Le gain que peuvent apporter ces méthodes, régulièrement améliorées, est certainement intéressant. Ceppendant, elles apparaissent comme sans intérêt immédiat pour la réalisation de TARSKI car, comme nous le verrons, le problème de la gestion de la mémoire est pour l'instant parfaitement secondaire dans notre système.

Aucune de ces techniques n'a été retenue.

2.1.3 PROLOG et compilation : la machine de Warren

Soulignons à nouveau que TARSKI et la WAM n'ont pas les mêmes buts. La machine de Warren a pour but de réaliser de la compilation de *clauses PROLOG* pour accélérer la résolution. TARSKI se contente de compiler les règles de résolution, ce qui n'a pas grand chose à voir.

En PROLOG la compilation a peu de chose à voir avec la compilation d'un langage procédural traditionnel. En effet, les objets manipulés par PROLOG n'étant que partiellement connus lors de la compilation (les variables sont libres), on ne peut réaliser qu'une compilation partielle du langage. Notons que la plupart, si ce n'est tous les PROLOG sont partiellement compilés dans la mesure où ils effectuent tous une transformation en syntaxe abstraite[7] des clauses du programme : il est bien évident que l'on ne va pas conserver une clause PROLOG sous sa forme ASCII ; elle est pré-traitée, et transformée en un objet structuré (nous reparlerons du passage en syntaxe abstraite dans TARSKI). Pour toutes les raisons pré-citées, la compilation PROLOG permet d'obtenir un gain de l'ordre de 5, contre 20 pour un langage procédural traditionnel.

[6] Le modèle MALI serait certainement adaptable à TARSKI mais cela n'est pas notre sujet.

[7] En anglais, tokenization

Il existe deux excellentes introductions à la WAM, celle réalisée par l'équipe d'Argonne ([GLLO85]) et celle écrite par Hassan Ait Kaci chez DEC ([AK91]), qui ont été d'un grand secours l'une et l'autre. J'ai également eu accès aux sources de deux PROLOG utilisant la compilation de type WAM, SB-Prolog de l'université de Stony Brooks ([DWDP88]), et SWI-Prolog de l'université d'Amsterdam ([Wie]), qui m'ont permis de voir comment est implantée pratiquement la machine de Warren. Enfin, Johan Bevemyr du SICS m'a communiqué un traceur de code WAM écrit en emacs-lisp, utilisé par le SICS pour enseigner la WAM aux étudiants, qui m'a également été bien utile.

La compilation au sens de Warren va, pour une clause donnée, créer un code de machine abstraite, qui sera par la suite interprété par le système hôte (ainsi Quintus PROLOG est un interpréteur de WAM écrit en assembleur, SICSTUS-PROLOG un interprète de WAM écrit en C) lors de la résolution. La machine abstraite est définie en termes d'instructions élémentaires, de piles de données et de registres. Je n'ai pas l'intention de la décrire ici : il y faudrait plusieurs chapitres et les rapports cités ci-dessus l'ont fait mieux que moi. Il est tout de même bon de noter certains points.

En raison du peu de clarté du rapport originel de Warren[8], plusieurs interprétations se sont faits jour, et il existe à l'heure actuelle plusieurs machines de Warren, toutes différentes. Enfin, Warren lui-même a modifié régulièrement sa machine pour améliorer son efficacité, ce qui n'a pas contribué à clarifier la situation. En fait, lorsque l'on rencontre dans un article une phrase contenant "extension de la machine de Warren" ou "implantation de la machine de Warren", cela signifie bien souvent implantation qui ressemble ou qui s'inspire de la machine de Warren.

La machine de Warren a eu un grand mérite, celle de devenir un paradigme accepté qui a permis à la communauté de la programmation PROLOG de se tourner vers d'autres problèmes, comme le parallélisme, en s'appuyant sur une connaissance commune : la machine de Warren[9]. Il n'y a pas un article, pas un poster qui n'ose parler de compilation sans se situer par rapport à la machine de Warren. Cela est cependant parfois dommage. Le monde LISP a montré qu'il était possible d'avoir plusieurs approches vis à vis des problèmes de compilation : à coté de techniques de machines virtuelles interprétées comme le LLM3 de LE_LISP ([Cha87]) on trouve également des systèmes de traduction directe en langage procédural comme KCL ([YH86]) qui traduit directement LISP en C, utilise le compilateur C et réalise ensuite un chargement dynamique du module créé, ou encore T ([RAM91]) dont le compilateur construit du code assembleur.

2.2 PROLOG et parallélisme

Nous allons dans ce paragraphe décrire rapidement les différentes techniques d'implantation parallèle de PROLOG. Pour une description plus globale nous ne pouvons qu'encourager la lecture des excellents articles publiés par TSI ([dKCRS89]). Je tiens à remercier ici l'équipe de l'Université de Bristol (projet Andorra [SCWY91b], [SCWY91a] et Aurora [CS89], [Sze89], [LWH+90], [LWH+88]), ainsi que Bogumil Hausman (problèmes de contrôle dans le parallélisme OU [Hau90]), Khayri A. M. Ali et Roland Karlson (projet Muse [AK90]) du SICS, et Geoff Sutcliffe (projet LINDA-PROLOG) de l'université de Western Australia ([Sut90]). Ils ont tous

[8] Nous parlons dans ce paragraphe de la seconde machine de Warren [War83], non de la première [War77], bien entendu.

[9] C'est un point qui fait cruellement défaut au monde de la méta-programmation où chaque équipe développe son système avec ses extensions propres, refaisant bien souvent sous une autre forme ce que d'autres ont fait. Les efforts apparaissent parfois bien désordonnés.

répondu à mes questions avec beaucoup de disponibilité et m'ont communiqué leurs différents rapports techniques, qui sont à la base de cette section.

2.2.1 Parallélisme OU

Le parallélisme OU consiste à faire exécuter en parallèle par plusieurs processeurs les différents choix qui peuvent se présenter lors de la sélection de clause. Ainsi, si nous considérons le programme suivant :

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

Un parallélisme OU consiste à faire exécuter par le premier processeur la résolution en choisissant la première clause, et parallèlement, par le second processeur, qui travaillera sur la seconde clause.

TARSKI ne fait pas de parallélisme OU au sens PROLOG sur les clauses, en revanche il fait du parallélisme OU sur les règles de résolution. L'influence des différents modèles de parallélisme OU a donc été grande dans le développement du système parallèle, et nous allons donc les discuter relativement en détail. Il faut cependant dire que les modèles dérivés de la machine de Warren ont été de peu d'utilité. Ils sont en effet complètement liés à l'architecture de la dite machine et les techniques qu'ils utilisent sont difficilement transposables à un autre système.

Il existe plusieurs problèmes de contrôle concernant le parallélisme OU. Il faut :

1. Choisir une architecture de réseau :
   - tous les processeurs peuvent-ils communiquer entre eux?
   - Y a-t-il un processeur maîıtre, ou tous les processeurs ont-ils le même statut?
   - Le système est-il un système à mémoire partagée ou les processeurs doivent-ils explicitement se transmettre les informations?
2. Adopter une stratégie de choix des clauses à paralléliser. On favorise en général les clauses se trouvant le plus près de la racine de l'arbre.
3. Choisir une technique de partage des informations nécessaires à la résolution. Ceci concerne tout particulièrement les différentes piles, et spécialement la pile de la résolvante, la pile des points de choix et la pile des environnements.

Un des grands problèmes des implantations des parallélismes OU est de bien gérer les différentes liaisons de la même variable correspondant à différentes branches PROLOG dans l'espace de recherche PROLOG.

Nous nous intéresserons tout particulièrement au modèle Kabu-Wake développé par Fujitsu au Japon. Ce modèle est probablement le plus proche du système TARSKI dans son fonctionnement.

Nous parlerons aussi du système AURORA, développé en collaboration par l'Université de Bristol, le laboratoire d'Argonne et le SICS, du système Muse, également développé par l'équipe du SICS[10].

[10] Cette équipe est à l'origine de SICStus PROLOG (une des meilleures implantations commerciales de PROLOG) comme son nom l'indique.

Le modèle Kabu-Wake : Le modèle Kabu-Wake est un modèle simple dans son principe et dans sa mise en œuvre. Il est principalement décrit dans [SSK+85] et [Mas86].

Chaque processeur du système Kabu-Wake se comporte comme un processeur standard exécutant un PROLOG interprété classique. Lorsqu'il a le choix entre n clauses, il crée un point de choix et continue la résolution normalement sur la première des alternatives. Si un processeur libre se manifeste par la suite, le processeur actif lui transmet une copie de l'ensemble des piles telles qu'elles étaient au moment de la création du point de choix, et le processeur libre va alors effectuer la résolution sur la suite de l'arbre. En un mot, le processeur libre reprend la résolution comme l'aurait fait le processeur actif en cas de retour arrière sur le point de choix.

Comme dans tous les cas de retour arrière, il y a un "petit" problème à régler : la remise à *libre* des variables qui ont pu être liées entre le moment où le point de choix a été créé et le moment où les piles sont transmises. On utilise sur la plupart des PROLOG séquentiels la pile de traîinée. Dans le modèle Kabu-Wake, on associe à chaque liaison de variable une date, qui correspond au nombre de points de choix entre la racine de l'arbre et le moment où la liaison a été effectuée. Au moment de la recopie, toutes les variables qui ont une date supérieure à la date du point de choix sont marquées *libre*. Cette technique est plus efficace que la technique classique dans le cas présent, car il faut de toute façon parcourir l'intégralité de la pile pour la recopier.

Une architecture dédiée spécialisée a été réalisée pour tester le modèle Kabu-Wake. Elle comporte 16 processeurs (68010) reliés entre eux par deux réseaux : un réseau de contrôle et un réseau de données. Les résultats sont très bons pour les problèmes ou l'espace de recherche est composé de branches de grande taille. En effet, chaque processeur se consacre alors complètement à une partie de l'arbre, et le nombre de communications est réduit.

Il est clair que le goulot d'étranglement du système Kabu-Wake est le réseau de communications, car la migration de tâches dans ce modèle est une opérations coûteuse. Si l'on parvient à avoir un réseau rapide et/ou un problème à grosse granularité, le système sera efficace. Un effet pervers du système Kabu-Wake est qu'il est d'autant plus efficace que chaque interprète particulier est lent : en effet, un interprète lent met plus de temps à effectuer une tâche, et le réseau de communication est donc moins saturé.

Nous reviendrons plus loin sur l'ensemble de ces problèmes, car ils s'appliquent presque complètement à l'implantation du parallélisme dans TARSKI.

Le système Aurora : Lors de la troisième conférence sur la programmation logique en 1986, un certain nombre de spécialistes décidèrent d'unir leurs efforts au sein d'un projet qui reçut le nom de projet Gigalips. Il y avait des chercheurs du laboratoire d'Argonne, près de Chicago, de l'Université de Bristol, et du Swedish Institute of Computer Science. Le cœur du projet fut le système Aurora.

Aurora se base sur le modèle SRI défini par David Warren en 1983, modèle qui fut raffiné par la suite. Le modèle SRI ([War87]) influença les travaux de Lusk et Overbeek qui implantèrent une version un peu différente du modèle SRI, le modèle ANL/WAM[11] ([DLO87]).

[11] ANL = Argonne National Laboratory.

Dans le modèle SRI, un groupe de *workers* (travailleurs seraient la traduction française, mais elle sonne assez mal.) coopèrent pour explorer un arbre de recherche PROLOG, en partant de la racine de l'arbre. L'arbre est défini implicitement par le programme, et doit être construit explicitement (et éventuellement supprimé) pendant l'exploration. Le premier *worker* qui entre dans une branche la construit, et le dernier *worker* qui quitte une branche la détruit. Les actions de construction et de destruction de branches constituent le véritable travail, et correspondent à la résolution et au retour-arrière en PROLOG. Quand un *worker* a terminé une partie continue du travail, appelée tâche, il parcourt l'arbre pour prendre en charge une autre tâche. Ce processus est appelé "changement de tâche " ou *scheduling*. Les *workers* essaient de maximiser le temps passé à travailler et à minimiser le temps passé à schéduler. Un *worker*, lorsqu'il effectue une tache, adopte une stratégie PROLOG standard de résolution.

L'arbre de recherches est représenté par des structures de données très proches de celle de la WAM. Les *workers* qui travaillent sur la même branche partagent les données sur cette branche. Dès que des données sont potentiellement partageables après la création d'un point de choix, elles ne peuvent plus être modifiées. Pour tourner cette restriction, chaque *worker* a un tableau privé de liaisons, dans lequel il stocke les liaisons conditionnelles, c'est a dire les liaisons vers des variables partageables. Ce tableau donne un accès immédiat à la liaison de la variable.

Dans ce modèle, avec l'aide des tableaux privés de liaisons, les opérations d'unification, et de déréférencement sont effectués avec un minimum de perte de temps par rapport à un PROLOG séquentiel classique quand un *worker* travaille. Lorsqu'il change de tâche, il doit cependant remettre à jour ses tableaux de liaison, en mettant à libre certaines variables lors de la remontée dans l'arbre et en ajoutant des liaisons lors de la descente dans une branche. Cependant, le nombre de données à transférer est moins grand que dans le modèle Kabu-Wake.

Le système Aurora impose un certain nombre de contraintes sur l'architecture du système, et en particulier la nécessité d'avoir de la mémoire partagée pour les piles de la WAM.

La BC machine (Muse) : Le système Muse a été également développé au SICS, par Khayri A. M. Ali et Roland Karlsson. Muse utilise des machines multi-processeurs à mémoire distribuée. Comme Aurora, Muse utilise également une extension de la WAM, mais contrairement au système Aurora aucune des piles de la WAM n'est partagée entre les différents processeurs, et c'est la son originalité. On peut se reporter à [AK90] pour plus de détails sur Muse. D'après les benchmarks réalisés par l'équipe de Muse, Muse serait plus rapide qu'Aurora. Il faut cependant rester prudent dans la mesure où Aurora implante PROLOG dans son ensemble alors que Muse ne traite pas le *cut*.

2.2.2 Parallélisme ET

Nous parlerons peu du parallélisme ET pur dans la mesure ou TARSKI ne l'utilise pas.

Le parallélisme ET consiste à exécuter en parallèle tous les prédicats de la queue d'une clause. Ainsi, si nous allons exécuter la clause :

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

le système exécutera en parallèle les buts $q(X)$ et $p(X)$.

Le problème principal du parallélisme ET apparaîıt dès l'exemple précédent : si X est déjà lié à un terme constant au moment de l'exécution parallèle de $p(X)$ et de $q(X)$, il n'y a pas de problème. Mais si la variable est libre, alors l'exécution parallèle de p et q risque de la lier à deux valeurs différentes. Comme il est peu raisonnable de laisser l'exécution se dérouler pour vérifier ensuite que les deux valeurs sont bien compatibles, on a développé plusieurs méthodes pour gérer ce problème :

Parallélisme ET restreint : Dans le parallélisme ET restreint une analyse statique suivie d'une génération de tests d'indépendance est faite ([DeG84], [Her86a]). Ainsi, dans le cas de la clause précédente, le compilateur génèrera un code qui, à l'exécution, vérifiera que la variable X est bien instanciée à une constante et autorisera l'exécution parallèle dans ce cas, et l'interdira sinon. Ce compilateur détectera également les prédicats indépendants[12] susceptibles d'être exécutés en parallèle.

Les pertes de temps lors de l'exécution d'un programme utilisant le parallélisme ET restreint sont limités aux tests d'indépendance des variables. Le problème est de correctement détecter la possibilité de paralléliser les différents prédicats d'une clause.

Parallélisme ET par flot : Le parallélisme ET par flot consiste à autoriser l'exécution en parallèle et à diffuser les liaisons des variables d'un processeur à un autre au fur et à mesure de la résolution[13].

On peut citer quelques exemples de PROLOG utilisant seulement le parallélisme ET dont APEX (And-Parallel Prolog EXecution) [LK88], basée sur une extension de la machine de Warren, tournant sur un Sequent Balance 21000, POPE qui utilise une architecture spécialisée ([BG87]), ou l'Abstract Machine for Restricted And parallelism ([Her86b]).

2.2.3 Langages gardés

Nous ne nous étendrons pas sur les techniques des langages gardés. En effet, nous souhaitons utiliser dans TARSKI le parallélisme intrinsèque au langage. Les techniques propres aux langages gardés nous intéressent donc fort peu.

Dans les langages gardés, le programmeur doit explicitement exprimer le caractère parallèle de son programme. Le but du langage gardé est de lui fournir les outils nécessaires à cet effet.

Les origines des langages gardés sont [CG81] qui définit le *Relational Language* et [CM81] qui définit IC-PROLOG. Les deux principes de base sont :

- Le programmeur spécifie pour chaque clause du programme une condition qui doit être satisfaite pour que le corps de la clause puisse être exécutée. Cette condition est appelée la **garde**, d'où le nom de langage gardé.
- Le non-déterminisme de PROLOG est restreint à un mécanisme appelé **don't-care nodeterminism**. Le choix d'une clause parmi un paquet de clauses définissant un prédicat ne peut pas être remis en cause, supprimant ainsi les retour-arrière.

Il existe plusieurs implantations de PROLOG gardés dont Parlog ([CG86]), Concurrent PROLOG ([Sha87]), et Guarded Horn Clauses ([Ued87]).

[12] On parle de parallélisme ET indépendant.

[13] Le parallélisme par flot est largement utilisé dans les langages gardés.

2.2.4 Approches mixtes

Andorra : Le modèle Andorra de base a pour but d'utiliser de façon transparente le parallélisme OU et une forme particulière de parallélisme ET, le parallélisme ET dépendant. Dans le modèle Andorra, les buts *déterminés*) doivent être exécutés avant les autres, et ces buts peuvent être exécutés en parallèle. Donc :

- Les buts peuvent être exécutés rapidement, et en parallèle, dès qu'ils sont déterminés. Un but est déterminé si on peut déterminer la clause unique qui permet de le résoudre. Dans le modèle Andorra originel, seule l'unification des têtes est utilisée pour détecter les buts déterminés.
- Quand aucun but n'est déterminé, un but est choisi et un point de choix est créé comme en PROLOG standard. Les points de choix peuvent être explorés en parallèle en suivant le modèle SRI décrit plus haut.

Une extension du modèle de base est en cours, qui devrait permettre une exécution parallèle de certains buts, même s'ils ne sont pas complètement déterminés.

Andorra supporte la totalité du langage PROLOG plus quelques extensions. Il faut souligner qu'Andorra utilise un interprète PROLOG et non un compilateur de type WAM.

COALA : COALA (Calculateur Orienté Acteur pour la Logique et ses Applications, [P+86], [P+87]) était initialement un modèle adapté au parallélisme OU qui a été étendu au parallélisme ET indépendant. COALA utilise le langage PROLOG pur.

Suivant l'expression des auteurs eux-mêmes "le graphe de connexion ET/OU de Kowalski ([Kow79]) constitue le cœur du modèle". Le principe du graphe de Kowalski est simple : on le construit par compilation du texte source du programme ; pour chaque prédicat apparaissant dans la partie droite d'une clause, on recherche le même prédicat apparaissant dans la partie gauche, et donc susceptible d'être utilisé comme résolvant. On crée alors un arc étiqueté par les liaisons des variables résultant de l'unification des deux prédicats. Ainsi, les arcs sont considérés comme les acteurs et l'exécution du programme correspond au dépliage du graphe statique pré-compilé : un pas de résolution consiste à sélectionner un arc et à construire la résolvante associée à partir de l'environnement fourni par l'étiquette de l'arc.

Le parallélisme OU est réalise en sélectionnant en parallèle les arcs issus d'un même littéral, le parallélisme ET en unifiant en parallèle les parties indépendantes d'un même résolvant.

Chaque processeur de la machine COALA utilise une mémoire privée et la communication entre processeurs est faite par messages,

Une implantation de COALA a été faite sur transputers, dont le modèle physique correspond bien au modèle théorique de la machine.

PEPSYS : Signalons pour mémoire le développement du système PEPSys ([WRdKS87])à l'ECRC[14]. Il s'agit d'un système qui s'inspire du modèle Kabu-Wake pour le parallélisme OU mais intègre également le parallélisme ET indépendant.

[14] European Computing Research Center

2.3 Méta-programmation

Les différents système de méta-programmation semblent vivre dans un pays du tiers monde : la natalité est particulièrement forte, et la mortalité également. On voit régulièrement apparaîitre de nouvelles extensions à PROLOG, de nouvelles techniques de contrôle, de nouvelles méthodes de méta-programmation. Chaque équipe semble prendre un malin plaisir à refaire ce qu'à fait l'équipe voisine, en changeant un opérateur et en utilisant un autre formalisme, pour un résultat des plus proches. D'autre part, la présentation de tous ces systèmes est généralement noyée avec soin dans une masse de résultats théoriques qui désespèrent le malheureux praticien que je suis : chacun de ces systèmes est bien entendu un système ad-hoc, adapté à une logique particulière (Gödel étant une des rares exceptions[15]), logique qui réclame au plus tôt sa preuve de complétude et de consistance.

Curieusement, quand on en vient aux problèmes d'implantation pratique, un silence religieux se fait. Les problèmes de bouclage sont rarement discutés, quant aux problèmes d'efficacité, ils ne sont en général même pas traités. Bien souvent, ces systèmes sont et demeurent, à l'instar des tigres de papier, des machines de papier.

Il n'existe actuellement aucun paradigme général reconnu dans le monde de a méta-programmation, par opposition au monde de la programmation PROLOG classique et parallèle qui s'appuie sur la machine de Warren. Je vais dans les quelques lignes qui suivent présenter les systèmes qui m'ont paru les plus importants, sachant que la vérité d'aujourd'hui peut parfaitement devenir la fausseté de demain.

2.3.1 Templog

Templog ([AM87], [Bau89b]) est une extension de PROLOG à la logique temporelle. Il existe plusieurs logiques temporelles. Templog utilise une logique temporelle discrète et linéaire. Le langage de Templog est une extension des clauses de Horn dans lequel on introduit trois opérateurs $\Diamond$ (peut-être), $\Box$ (toujours) et $\circ$ (ensuite). Dans Templog, l'opérateur $\circ$ peut apparaîitre n'importe où dans les clauses, l'opérateur $\Diamond$ peut apparaîitre dans n'importe quel corps de clause et l'opérateur $\Box$ dans n'importe quelle tête de clause.

On peut voir l'utilisation du langage sur un exemple. Le programme Templog suivant :

- $p(a)$.
- $\Box(\circ p(s(s(X))) : -p(X))$

définit un prédicat p tel que $p(X)$ est vrai à l'instant i tel que $X = s^{2i}(a)$

La résolution dans Templog se fait à l'aide de règles de résolution appelées règles TSLD, qui sont définis à partir des axiomes de la logique temporelle linéaire discrète (voir figure 2.2). A chaque étape de la résolution, un atome est sélectionné dans le but courant et une nouvelle résolvante est formée à partir du but courant, de la clause courante et des règles de résolution.

Templog est, dans sa méthode de résolution, très proche du mécanisme TARSKI : utilisation d'extension de clauses de Horn et résolution avec des règles de résolution dérivées des axiomes logiques.

Mais alors que TARSKI est paramétrable en fonction de la logique, Templog est un système adapté uniquement à la logique temporelle linéaire discrète.

[15] Gödel est une remarquable et véritable tentative de repenser en profondeur la programmation logique, et non un des nombreux replâtrages que l'on rencontre fréquemment.

Cond	But	Clause	Résolvante
	$\leftarrow B_1, \circ^i A, B_2$	$\circ^i A' \leftarrow B'$	$\leftarrow (B_1, B', B_2)$
$i \geq j$	$\leftarrow B_1, \circ^i A, B_2$	$\Box\, \circ^j A' \leftarrow B'$	$\leftarrow (B_1, B', B_2)$
$i \geq j$	$\leftarrow B_1, \circ^i A, B_2$	$\Box(\circ^j A' \leftarrow B')$	$\leftarrow (B_1, \circ^{i-j} B', B_2)$
$i \geq j$	$\leftarrow B_1, \Diamond(B_2, \circ^i A, B_3), B_4$	$\circ^j A' \leftarrow B'$	$\leftarrow (B_1, \circ^{i-j} B_2, B', \circ^{i-j} B_3, B_4)$
$i \geq j$	$\leftarrow B_1, \Diamond(B_2, \circ^i A, B_3), B_4$	$\Box\, \circ^j A' \leftarrow B'$	$\leftarrow (B_1, B', \Diamond(\circ^{i-j} B_2, \circ^{i-j} B_3), B_4)$
$i \geq j$	$\leftarrow B_1, \Diamond(B_2, \circ^i A, B_3), B_4$	$\Box\, \circ^j A' \leftarrow B'$	$\leftarrow (B_1, B', \Diamond(B_2, B_3), B_4)$
$i \geq j$	$\leftarrow B_1, \Diamond(B_2, \circ^i A, B_3), B_4$	$\Box(\circ^j A' \leftarrow B')$	$\leftarrow (B_1, \Diamond(\circ^{i-j} B_2, B', \circ^{i-j} B_3), B_4)$
$i \geq j$	$\leftarrow B_1, \Diamond(B_2, \circ^i A, B_3), B_4$	$\Box(\circ^j A' \leftarrow B')$	$\leftarrow (B_1, \Diamond(B_2, \circ^{i-j} B', B_3), B_4)$

Figure 2.2: Règles de résolution TSLD

2.3.2 Chronolog

Chronolog ([OW90]) est un langage de programmation basé sur la logique temporelle et sur la sémantique des mondes possibles. La logique temporelle permet de décrire des propriétés dépendant du temps de façon "directe".

Chronolog utilise deux opérateurs seulement, **first** et **next**l la logique utilisée est un peu particulière :

- first first $A \leftrightarrow$ first A
- next first $A \leftrightarrow$ first A
- first $(A \wedge B) \leftrightarrow$ (first A) $\wedge$ (first B)
- next $(A \wedge B) \leftrightarrow$ (next A) $\wedge$ (next B)
- first $\neg A \leftrightarrow \neg$(first A)
- next $\neg A \leftrightarrow \neg$(next A)
- first $(\forall x)A \leftrightarrow (\forall x)$(first A)
- next $(\forall x)A \leftrightarrow (\forall x)$(next A)

Un programme Chronolog est un ensemble de clauses de Horn dans lequel on autorise l'application des opérateurs **first** et **next** aux formules atomiques. Un exemple de programme Chronolog définissant le prédicat *fib(n)* qui à tout instant t est vrai si n est le $t+1$ nombre de fibonacci est :

```
first fib(0).
first next fib(1).
next next fib(N) <- next fib(X), fib(Y), add(X,Y,N).
```

Ce programme définit que 0 est le premier nombre de Fibonacci, 1 le nombre suivant, etc...

Une implantation efficace de Chronolog semble difficile à réaliser d'après les auteurs eux-mêmes : " une implantation efficace de Chronolog doit combiner les techniques classiques de la programmation logique (unification, backtracking) avec des techniques liées aux systèmes data-flow (marquage, mémoires associatives)".

2.3.3 RACCO

RACCO ([HNSN86]) est un langage de programmation modulaire basé sur la logique modale et la logique temporelle. Sa conception a pour but de faciliter des descriptions précises pour le contrôle de processus complexes en temps réels, où chacun des composants du système a une complète autonomie, et est en interaction avec les autres composants. Les processus sont appelés *modules* dans RACCO, mais les deux termes recouvrent le même concept.

Les principales caractéristiques de RACCO sont :

Utilisation de la logique temporelle : La description des opérations effectués par les différents processus est basé sur Temporal PROLOG ([Sak90]), qui est un langage de programmation logique permettant une représentation du temps : à chaque instant, est associé un état du monde et une valeur de vérité pour chaque prédicat qui dépend de l'instant considéré.

Interactions entre les processus : Les interactions entre processus sont faites par passages de messages, le passage de message consistant à faire prendre la valeur *vraie* à un prédicat du processus auquel on envoie un message.

Utilisation de la logique modale : Les concepts de modules, ainsi que les notions de temps et de modalités sont traités dans RACCO à l'aide d'opérateurs modaux.

RACCO, comme la plupart des systèmes décrits ici, est un système *ad hoc*. Il s'intéresse à un cas particulier d'implantation de logique non-classique, dans un but précis.

2.3.4 Gödel

Gödel ([HL91]) est un projet ambitieux dont le but est de donner un successeur à PROLOG, en comblant nombre des problèmes de PROLOG. Un des principaux desseins de Gödel est de conserver l'expressivité et la fonctionnalité de PROLOG, mais d'améliorer la sémantique déclarative du langage. Gödel fournit au programmeur tout un ensemble de facilités supplémentaires comme :

- Le typage fort
- Des opérations de contrôle plus puissantes
- Des modules
- Des possibilités de méta-programmation.

Gödel est fort éloigné de TARSKI par ses méthodes et ses buts, mais, même s'il est de peu d'intérêt pour nous, il faut souligner le grand intérêt du travail accompli par Hill et Lloyd : il repense véritablement en profondeur les techniques de programmation logique. Les auteurs souhaitent que Gödel soit à PROLOG ce que Pascal fut à FORTRAN. Je pense que l'on pourrait aussi parler de l'évolution de LISP à ML dans la famille des langages fonctionnels.

Il reste à savoir si Gödel sera effectivement implanté et quel sera son succès. Il impose au programmeur plus de contraintes que PROLOG, comme tous les langages typés, et il n'est pas sûr que nombre d'utilisateurs PROLOG, qui ont précisément choisi ce langage car il permet de programmer rapidement en sachant peu de choses seront disposés à utiliser Gödel.

2.4 Conclusion

Indiquons rapidement la place que TARSKI va occuper dans cet ensemble.

En ce qui concerne le parallélisme, TARSKI va suivre un modèle multi-séquentiel (comme les systèmes PEPsys ou Kabu-Wake) : nous allons définir une machine abstraite séquentielle. L'exécution parallèle sur plusieurs processeurs de ce système séquentiel assurera le parallélisme du système. La communication entre processeurs sera assuré par un système de messages défini de façon abstraite. L'implantation du système dépendra de l'architecture physique de l'ordinateur disponible.

Par rapport aux autres systèmes faisant de la résolution en logique non-classique, TARSKI tente de définir une extension générale du mécanisme de résolution : utilisation d'extension de clauses de Horn et résolution avec des règles de résolution dérivées des axiomes logiques.

Templog est un système relativement proche dans ses principes de TARSKI , mais il est adapté uniquement à la logique temporelle linéaire discrète.

Partie II

Spécifications de l'automate

Chapitre 3

Objets manipulés

3.1 TARSKI vu comme un automate formel

Nous souhaitons immédiatement attirer l'attention sur un point très important : bien que TARSKI ait été initialement conçu pour l'implantation de techniques de résolution en logique non-classique, nous nous attacherons à partir de maintenant à le considérer comme un automate formel manipulant un ensemble de phrases formelles (les clauses), et appliquant un mécanisme général paramétré par des règles formelles de résolution qui agissent comme des règles de réduction ou de réécriture.

Dans ce chapitre, nous allons présenter la syntaxe des différents objets utilisés par le système (clauses et règles d'inférence). Le chapitre 4 présentera le mécanisme général de fonctionnement du moteur d'inférence, et le chapitre 5 les différentes techniques de contrôle.

3.2 Les clauses

Nous allons décrire dans ce chapitre l'ensemble des objets constitutifs des clauses. Certains objets sont "classiques" et bien connus de toutes les implantations de PROLOG (prédicats, variables, atomes, listes). En revanche, nous tenons à attirer l'attention sur deux points bien particuliers :

- La notion d'**opérateur** est nouvelle et ne se superpose pas exactement à la notion d'opérateur modal.
- La syntaxe des clauses est fondamentalement différente, **y compris** pour la représentation de clauses en logique classique. Cette différence est plus profonde qu'une simple différence syntaxique, nous y reviendrons.

Il est toujours difficile de définir clairement et simplement la syntaxe d'un langage ; l'on risque soit de verser dans un formalisme trop sec, soit dans un verbiage imprécis, voire incorrect. Dans un premier temps, nous allons nous efforcer de définir informellement la structure des principaux objets en passant sous silence, par exemple, la nature de leurs arguments, avant de revenir à une définition plus formelle.

3.2.1 Les atomes

La machine doit pouvoir manipuler un certain nombre d'objets élémentaires classiques :

Les nombres entiers : les nombres entiers sont indispensables pour l'implantation de tout système un peu réaliste. Nous noterons les nombres entiers comme une suite de chiffres sans point ni virgule.

Exemples : 1, 1285, 1285993845

Les nombres réels : les nombres réels sont moins indispensables que les nombres entiers, mais comme nous souhaitons pouvoir traiter des implantations de logique floue, leur présence était obligatoire. Nous les noterons comme une séquence de chiffre, le point séparant la partie décimale de la partie entière.

Exemples : 1.2345, 3.141592654, 7583920.0

Les caractères : Nous distinguerons les caractères en les plaçant entre deux apostrophes.

Exemples : 'a', 'b', 'f'

Les constantes symboliques : Une constante symbolique est une suite de lettres ou de chiffres commençant par une **minuscule**.

Exemples : marbre, falaise, sous, jUnger, m1234, mC6

3.2.2 Les prédicats

Le nom d'un prédicat sera une constante symbolique comme définie au dessus. Un prédicat aura une arité (ou nombre de variables). Deux prédicats ayant le même nom et des arités différentes seront considérés comme différents. La syntaxe générale d'un prédicat est : **nom(arg1, arg2,..., argn)**.

Exemples : p(a,b,c), q(1,2)

3.2.3 Les variables

Les variables seront représentés par une suite de caractères contenant des lettres ou des chiffres et commençant par une majuscule.

Exemples : $X123$, Xa, XY.

3.2.4 Les doublets et les listes

Un doublet est une structure comportant deux arguments que nous noterons $[a, b]$ (ce qui correspond à la paire pointée LISP $(a.b)$). Par soucis de simplicité d'écriture, l'objet $[A, [B, C]]$ sera noté $[a, b, c]$. Suivant en cela la syntaxe PROLOG-II, la liste LISP (ab) sera représentée $[a, b, nil]$ ou nil est une constante symbolique.

3.2.5 Les opérateurs

Un nom d'opérateur est une constante symbolique précédée du symbole #. Un opérateur a, comme un prédicat, une arité (son nombre d'arguments).

Exemples : **#NEC(p)**, **#POSI(q,1)**

3.2.6 Définition formelle des clauses et objets manipulés

Nous allons maintenant donner une définition plus formelle des objets manipulés par le système.

1. Les variables, les entiers, les flottants et les constantes symboliques sont des **objets**.
2. Si A et B sont des objets, alors $[A, B]$ est un objet.
3. Si p est une constante symbolique et que a_1, a_2, ..., a_n sont des objets alors $p(a_1, a_2, \ldots, a_n)$ est un **prédicat** d'arité n.
4. Si O est une constante symbolique et que a_1, a_2, ..., a_n sont des objets alors $\#O(a_1, a_2, \ldots, a_n)$ est un **opérateur**.
5. Si A est un prédicat alors A est une forme clausale.
6. Si A est un opérateur et B est une forme clausale alors $A : B$ est une forme clausale. On dit dans ce cas que A **qualifie** B, et que B est une **tête** de A.
7. Si A est une forme clausale, alors A est un objet.
8. Si A est une forme clausale alors $A.$ est une **clause**.
9. Si A est une forme clausale, alors $?A$ est un **but**.

Voici quelques exemples :

#NEC(a) est un opérateur avec un argument. Ce n'est pas une forme clausale, ni un objet.

#POS est un nom d'opérateur.

p(1,w,2) est un prédicat à trois arguments. C'est également une forme clausale et un objet.

X12 est une variable. Ce n'est par une forme clausale, mais c'est un objet.

#POSI(a,1):p(1) est une forme clausale et un objet.

#ASSUM(#NEC(1):p(q)):r est une forme clausale et un objet.

#ASSUM(#NEC(1):p(q)):r. est une clause, mais ce n'est ni une forme clausale ni un objet.

3.2.7 Interprétation

Il est clair que la syntaxe définie ci-dessus, si elle a de nombreux points communs avec PROLOG, ne permet pas de représenter les clauses PROLOG classiques de la forme $p :- q$. Nous allons maintenant nous attarder sur notre formalisme et montrer comment il peut représenter de façon uniforme à la fois les clauses classiques et les clauses contenant des opérateurs temporels, modaux, flous, ...

Considérons la formule : $p \wedge q \rightarrow r$. Sa représentation en PROLOG (syntaxe Edimburgh) serait : $r :- p, q$. Pour représenter cette clause dans notre formalisme, nous utilisons un opérateur que nous noterons $\wedge(X)$, opérateur qui a un argument. La représentation de la

clause précédente devient alors : $\wedge(q) : \wedge(p) : r$. Ceci correspond bien à une clause valide suivant la définition du paragraphe précédent.

Considérons maintenant la formule "je sais p" : $\Box p$. Cette formule se représente par : $\mathtt{SAV} : p$. Considérons maintenant la formule "Si a sait p, alors b sait p" : $\Box(a)p \rightarrow \Box(b)p$. Cette formule se représente par : $\wedge(\mathtt{SAV}(b) : p) : \mathtt{SAV}(a) : p$

Prenons un cas plus difficile : "Si (a sait que b sait p) et (c sait q) alors c sait p", $(\Box(a)\Box(b)p) \wedge (\Box(c)q) \rightarrow (\Box(c)p)$ s'exprime par : $\wedge(\mathtt{SAV}(c) : q) : \wedge(\mathtt{SAV}(a) : \mathtt{SAV}(b) : p) : \mathtt{SAV}(c) : p$.

Il s'agit donc d'une notation préfixée, qui peut rappeler sous certains aspects la notations polonaise inversée qui note $p \wedge q$ par Kpq. Il faut pourtant remarquer un point important. Cette notation détruit la symétrie de Kpq. Ceci est parfaitement naturel dans la mesure où les différents PROLOG eux-mêmes ont en général détruit la sémantique symétrique du $\wedge$. Il est en effet bien différent en PROLOG de résoudre $p :- q, r.$ et $p :- r, q.$ Cette notation ne fait, en un sens, que le rappeler. On remarquera que les sous-buts sont placés dans l'ordre inverse de leur apparition. Ainsi $p :- r, q$ s'écrit $\wedge(q) : \wedge(r) : p$ et non $\wedge(r) : \wedge(q) : p$.

Les buts sont bien entendu représentés de la même façon que les clauses. Ainsi le but $p \wedge q$ s'écrira $\wedge(q) : p$.

Il faut enfin souligner que cette notation[1] n'a pas été adoptée légèrement. Elle présente l'immense avantage de ramener le cas modal et le cas classique au *même traitement*, à la fois pour l'écriture des règles paramétriques de résolution, comme nous allons le voir, et aussi au niveau du moteur d'inférence lui-même. L'utilisation de cette représentation a réduit considérablement la complexité du système si on le compare à ses premières implantation ([AG88], [Bri87]), et renforcé son unité, ramenant tous les aspects de la résolution au même automate.

En ce sens, on pouvait dire que MOLOG était une extension de PROLOG aux logiques non-classiques (puisqu'il conservait les règles classiques de résolution et permettait d'en adjoindre de nouvelle) ; A l'opposé, PROLOG apparaîit plutôt comme un cas particulier de TARSKI, puisque les clauses PROLOG et les règles permettant d'implanter PROLOG ne sont plus que des cas particuliers de la forme générale.

3.3 Règles de résolution

Les principes généraux pour la résolution automatique sur des clauses de Horn générales ont été développés dans le chapitre 1. Nous allons dans ce paragraphe nous attarder sur les différentes formes des règles de résolution, ainsi que sur les contraintes imposées à la forme de ces règles. Nous utiliserons pour les représenter un formalisme dérivé du calcul des séquents de Gentzen comme nous l'avons signalé au chapitre 1.

L'utilisation des règles d'inférence sera détaillée dans le chapitre 4.

Les règles d'inférence sont réparties en trois catégories : les règles générales, les règles réflexives et les règles de terminaison. Les règles générales sont celles qui sont usuellement

[1] Cette notation semble poser des problèmes à la plupart des gens qui la rencontrent pour la première fois. On peut tenter d'en donner la lecture "intuitive" suivante : dans le cas des faits, on peut lire l'opérateur $\wedge$ comme signifiant **SI**. La clause $\wedge(q) : \wedge(r) : p$ se lit donc : **SI q, SI r, p**. En ce qui concerne les buts, la difficulté de la compréhension vient d'un raccourci d'écriture, bien utile pour la résolution, mais peu intuitif. L'écriture correcte de la question $p \wedge q$ devrait être $\wedge(p) : \wedge(q)$, que l'on pourrait lire **SI p, SI q**, ou est-ce que p, est-ce que q : ceci est parfaitement cohérent avec l'écriture des clauses. La représentation choisie $\wedge(p) : q$ a pour intérêt de mettre en évidence le sous-but qui va être choisi pour commencer la résolution, et c'est là la raison d'être de cette écriture.

employées pour exprimer une étape de la résolution : à partir du but courant et d'un fait pris dans la base, elles construisent une partie de la résolvante et modifient le but et le fait courant ; puis elles rappellent récursivement une autre règle de résolution pour terminer de résoudre le but. Les règles réflexives procèdent comme les règles générales, mais n'utilisent pas de fait pris dans la base : elles opèrent seulement sur le but courant. Enfin, les règles de terminaison fonctionnent sur le même principe que les règle générales, mais elles terminent la résolution : elles ne rappellent pas récursivement d'autres règles.

Nous allons examiner chacune de ces trois formes.

3.3.1 Les règles générales

Les règles générales sont de la forme :

$$\frac{A, ?B \vdash ?C}{A', ?B' \vdash ?C'}$$

Ici, A et A' sont des clauses, et B, B', C, C' des buts. Comme de coutume en méta-programmation, les objets du langage objet sont représentés par des variables dans le méta-langage.

On peut lire la règle de la façon suivante : "Si le but est B et que le fait est A alors un nouveau but C peut être inféré si la résolution du fait A' et du but B' donne le but C'". Il s'agit donc d'une définition récursive de la résolution.

Un certain nombre de conditions[2] doivent être vérifiées pour que la règle soit valide au sens TARSKI (nous notons $var(X)$ l'ensemble des variables de la forme clausale X) :

- $var(A') \subset var(A)$
- A' est une tête de A ou A est une tête de A'
- C' est une variable
- C' est une tête de C

La règle :

$$\frac{\wedge(X) : A, ?B \vdash ? \wedge (X) : C}{A, ?B \vdash ?C}$$

vérifie les conditions établies ci-dessus.

Que signifie cette règle ? Si nous avons un fait qui commence par $\wedge(X)$ suivi de A (séquence quelconque), et si nous avons une question B (quelconque), alors on place dans la résolvante $\wedge(X)$ (construction de la résolvante). Le nouveau fait est *réduit* à A, et le nouveau but est toujours B. Après l'exécution de cette règle, la résolvante a donc ét partiellement construite, et l'on rappelle récursivement le moteur d'inférence avec le nouveau fait A et le nouveau but B.

[2] Il faut noter que ces conditions, placées sur les règles de résolution, ne sont pas des conditions logiques. Il existe des règles de résolution logiques ne vérifiant pas ces conditions qui sont parfaitement valables sur un plan logique. Les conditions placées sur les règles sont propres à TARSKI et ont été choisies pour des raisons d'efficacité et pour privilégier une implantation efficace. Nous y reviendrons dans la partie réservée a l'implantation du système.

3.3.2 Règles réflexives

Les règles réflexives sont de la forme :

$$\frac{\text{true}, ?B \vdash ?C}{A', ?B' \vdash ?C'}$$

Ces règles utilisent le fait spécial `true`. Ces règles doivent vérifier les conditions suivantes :

- A' est soit :
  - Une variable. Cette variable sera unifiée a un nouveau fait pris dans la base de clauses, ou
  - N'importe quelle clause construite à partir des variables de B et C et de constantes.
- C' est une variable
- C' est une tête de C.

Un exemple de règle réflexive est :

$$\frac{\text{true}, ?ASSUM(A) : B \vdash ?C}{A, ?B \vdash ?C}$$

Nous détaillerons dans le chapitre suivant le fonctionnement de notre système. Il n'est cependant pas inutile d'expliquer ce que signifie cette règle, même de façon informelle : si la question commence par $ASSUM(A)$ suivi par une séquence quelconque (B), alors on ne sélectionne pas de fait dans la base, mais on continue la résolution avec le nouveau fait A ; le nouveau but est l'ancien but réduit : B ; on ne rajoute rien dans la résolvante.

3.3.3 Règles de terminaison

Les règles de terminaison (qu'il faudrait appeler en fait règles de terminaison partielles), ont pour forme :

$$A, ?B \vdash ?C$$

Elles sont terminales pour la récursivité de la résolution, d'où leur nom. Il n'y a pas de condition à respecter sur les règles de terminaison. Un exemple de règle de terminaison valide est la règle :

$$p, ?p \vdash ?\texttt{true}$$

Cette règle signifie : Si le fait est réduit à un prédicat p et que la question est réduite au même prédicat p, alors on rajoute dans la résolvante `true` et la résolution du but courant est terminé. Lorsque l'on atteint une règle de terminaison partielle, la résolvante est complète et il faut la réécrire sous la forme d'une nouvelle question pour pouvoir continuer la résolution : c'est le but des règles de réécriture que nous allons voir dans la section 3.4.

3.3.4 Extensions à la forme générale

Toutes les règles (aussi bien les règles générales que les règles réflexives ou les règles de terminaison) admettent une *forme étendue* :

Règle si Condition

Règle est une règle conforme aux conditions énoncées dans les sections précédentes. *Condition* est un ensemble de conditions procédurales écrites à partir des instructions de la machine abstraite décrite dans le chapitre 6. Ces conditions peuvent par exemple permettre de vérifier la présence ou l'absence d'un prédicat dans la base, de forcer (ou de vérifier la possibilité de) l'unification d'une variable et d'un objet, etc...

3.4 Les règles de réécriture

Les règles de réécriture sont utilisées lors de la transformation de la résolvante en question (voir chapitre 4). Elles ont pour forme :

$$G1 \leadsto G2$$

ou $G1$ et $G2$ sont des buts. Il n'y a pas de condition sur ces règles. Ces règles admettent une extension comme les règles de résolution ; on peut ainsi leur associer une condition procédurale à vérifier où à exécuter.

Exemple de règle de réécriture :

$$X : \wedge(A) : true \leadsto X : A$$

Ceci signifie que si la résolvante se termine par $\wedge(A) : true$, alors la nouvelle question se terminera par A. Cette règle permet d'éliminer `true` de la résolvante et de dégager le nouveau sous-but à résoudre pour poursuivre la résolution. Exemple de règles de réécriture avec une condition procédurale :

$$FUZ(X_1) \ldots FUZ(X_n) : A \leadsto A \text{ et afficher}(\min(X_i))$$

3.5 Conclusion

Nous nous sommes attachés dans ce chapitre à présenter les différents objets manipulés par le système : clauses, atomes, prédicats, variables, opérateurs, règles de résolution, règles de réécriture. Dans le chapitre suivant, nous allons détailler le fonctionnement du système et montrer comment il manipule ces objets.

Chapitre 4

Moteur d'inférence

4.1 Mécanisme général

Le moteur d'inférence TARSKI reprend nombre des mécanismes du moteur d'inférence PROLOG. Cependant, il est plus complexe dans la mesure où la résolution connaîit deux étapes non-déterministes au lieu d'une : la sélection de clause (comme en PROLOG) et la sélection de la règle de résolution (qui n'a pas d'équivalent PROLOG). D'autre part, le mécanisme d'*unification*, général à tous les systèmes de résolution logique, doit être étendu pour traiter les opérateurs. C'est ce premier aspect que nous allons tout d'abord traiter avant de nous intéresser au cycle de fonctionnement du moteur d'inférence.

4.1.1 L'unification

L'unification sur les termes classiques est la même que l'unification PROLOG classique. Le cas particulier des opérateurs se définit avec la même facilité. Détaillons rapidement le mécanisme d'unification. Nous n'examinerons pas tous les n^2 cas (n représentant le nombre de types de base du langage), mais seulement les $n^2/2$ restant après élimination des symétries. Considérons donc les possibles unifications de deux objets X et Y :

Variable : Si X est une variable libre, il s'unifie à n'importe quel objet Y. S'il s'agit d'une variable liée, on est ramenée au cas de l'unification de l'objet auquel est lié X (la valeur de la variable déréférencée) et de Y.

Valeur numérique : Si X est une valeur numérique (entier ou flottant), il s'unifie à Y si Y est aussi une valeur numérique et si $X = Y$. (Rappelons une dernière fois que le cas Y variable a été traité par symétrie précédemment).

Constante symbolique : Si X est une constante symbolique, X s'unifie à Y si Y est une constante symbolique et $X = Y$.

Paire pointée : Si X est une paire pointée ($X = [A, B]$), X s'unifie avec Y si Y est une paire pointée ($Y = [C, D]$) et A s'unifie avec C et B s'unifie avec D.

Prédicat : Si X est un prédicat ($X = n1(a_1, a_2, \ldots, a_n)$), X s'unifie avec Y si Y est un prédicat ($Y = n2(b_1, b_2, \ldots, b_m)$), que $n1$ et $n2$ sont des constantes symboliques unifiables, $n = m$ et pour tout i, a_i s'unifie avec b_i.

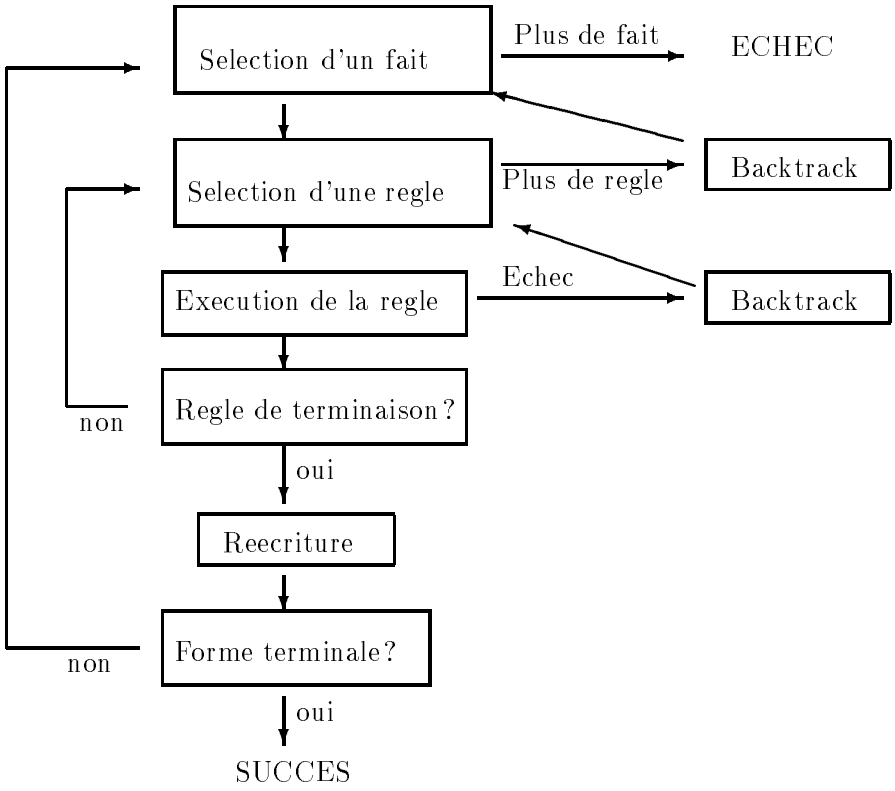


Figure 4.1: Cycle du moteur d'inférence

Forme clausale : Si X est une forme clausale $X = \#O(a_1, \ldots, a_n) : C$, X s'unifie avec Y si Y est une forme clausale, $Y = \#M(b_1, \ldots, b_m) : D$, O et M sont des constantes symboliques unifiables, $n = m$, pour tout i a_i s'unifie avec b_i et C s'unifie avec D.

4.1.2 Cycle de fonctionnement

Le cycle de fonctionnement du moteur d'inférences TARSKI est présenté sur la figure 4.1. Reprenons ce schéma :

1. La première étape est la *sélection d'une clause* permettant de commencer la résolution de la question.

2. Cette clause étant sélectionnée, on choisit une règle de résolution qui, la question et le fait étant fixés, va construire un élément de la résolvante et modifier la question et le fait courant. Cette étape se répètera jusqu'à ce qu'une règle de terminaison partielle soit sélectionnée et exécutée avec succès.

3. On utilisera alors les règles de réécriture pour construire la nouvelle question à partir de la résolvante.

4. Si la nouvelle question est une forme terminale, la résolution est terminée. Sinon, le moteur reprend son cycle de fonctionnement à l'étape 1, avec la nouvelle question.

Nous allons maintenant détailler chacune des étapes du cycle de fonctionnement du moteur.

4.2 Sélection des clauses

Sélectionner une clause est une opération qui peut se produire dans deux situations (comme en PROLOG) qu'il faut bien distinguer :

La sélection initiale : On est dans une situation de *sélection initiale* quand on doit sélectionner une clause pour commencer la résolution d'une question pour la première fois.

La sélection après backtrack : Il s'agit là de sélectionner une nouvelle clause après backtrack sur le choix de la clause précédente

Les deux opérations sont paramétrables dans TARSKI. Cependant, le seul mécanisme implanté à l'heure actuelle est le classique "premier trouvé, premier choisi" en effectuant la sélection sur le prédicat de tête de la question et du fait, qui doivent être unifiables. Insistons pourtant sur le fait qu'il serait parfaitement possible de changer le mécanisme de sélection sans modifier en quoi que ce soit le reste du système.

Exemple : soit la question : $\wedge(q) : p$ et la base de clause :

1. $\wedge(q) : p$
2. $\texttt{SAV}(a) : \wedge(p) : q$
3. $\texttt{SAV}(b) : p$

Le prédicat de tête de la question est p. La première clause sélectionnée sera donc la clause (1). En cas de backtrack, la clause sélectionnée sera (3). Si un nouveau backtrack se produit, la sélection échouera car il n'y a plus de clauses sélectionnables.

Ce mécanisme est exactement le même (sous cette forme) que la classique sélection des faits en PROLOG.

4.3 Sélection des règles de résolution

Pour une logique donnée, l'ensemble des règles de résolution est fixe. Nous allons considérer tout d'abord les règles permettant d'implanter la logique classique :

Règles d'inférence générales :

$$\frac{A, ? \wedge (D) : B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C} \tag{4.1}$$

$$\frac{\wedge(D) : A, ?B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C} \tag{4.2}$$

Règles de terminaison :

$$p, ?p \vdash ?\texttt{true} \tag{4.3}$$

Il n'y a pas de règles réflexives. Attardons nous à nouveau sur la signification de ces règles. La règle 4.1 signifie que si le fait est de la forme A (A représente ici une séquence quelconque) et que la question est de la forme $\wedge(D) : B$ (c'est à dire commence par $\wedge(D)$ suivi d'une séquence quelconque B) alors la résolvante sera $\wedge(D) : C$ si la résolvante du fait A et de la question B est C. La règle 4.2 est exactement symétrique de la règle 4.1. Quant à la règle 4.3, elle signifie que la résolvante d'un fait réduit à un prédicat p et d'un but réduit au même prédicat p est `true`, et que la résolution du but courant est terminé.

Les règles sont sélectionnées dans l'ordre dans lequel elles apparaissent dans la base de règles, comme les clauses. Pour qu'une règle puisse être sélectionnée, il faut que le fait de la règle et le fait réel soient unifiables. De même, il faut que la question de la règle et question réelle soient unifiables. Si la sélection échoue (éléments non unifiables) ou si l'exécution d'une règle échoue, on bactracke et on sélectionne la règle suivante.

Voyons cela sur un exemple. Supposons que les règles soient les règles de la logique classique, que la question soit q et le fait $\wedge(p) : r$. Le moteur commence par sélectionner la première règle de la base, 4.1. Mais la forme de la question dans la règle $\wedge(D) : B$ et la question réelle q, ne sont pas unifiables. On passe donc à la règle suivante ; ici la forme de la question est B qui s'unifie parfaitement avec q et la forme du fait est $\wedge(D) : A$ qui s'unifie avec $\wedge(p) : r$ (D s'unifie à p et A à r. La règle 4.2 est donc sélectionnée.

La sélection des règles crée un second type de points de choix, et un second non-déterminisme. C'est pour cela que nous employons le terme de double non-déterminisme dans le cas de TARSKI.

Une fois qu'une règle a été sélectionnée, il faut l'exécuter. C'est ce que nous allons développer dans la section suivante.

4.4 Exécution des règles de résolution

L'exécution des règles de résolution est l'étape fondamentale du fonctionnement du système. Considérons les règles de la logique classique présentées dans la section précédente. Nous allons examiner leur fonctionnement sur un nouvel exemple ; considérons que nous avons sélectionné le fait $\wedge(r) : p$, que la question est $\wedge(q) : p$.

La règle sélectionnée est la règle 4.1. Son exécution provoque alors les modifications suivantes :

- $\wedge(q)$ est mémorisée dans la *résolvante*.
- Le question devient p
- Le fait n'est pas modifié. Il reste $\wedge(r) : p$.

La règle 4.1 a donc pu s'exécuter complètement et a totalement réussi.

Rappelons donc que l'exécution d'une règle de résolution :

- Contribue à la construction d'un objet que nous appelons *résolvante*. La résolvante contiendra à la fin de la résolution l'ensemble des objets mémorisés par les règles de résolution. La résolvante est construite récursivement par empilements successifs.
- Modifie le fait
- Modifie la question

Le lecteur attentif a du cependant apercevoir une légère faille dans notre système qui fait que son comportement est légèrement différent de celui d'un PROLOG standard. Nous y reviendrons dans le chapitre consacré aux problèmes de contrôle (chapitre 5).

4.5 Réécriture de la résolvante

Lorsque l'on a atteint une règle de terminaison partielle, la résolvante est close et la question achevée. Il faut alors construire une nouvelle question à partir de la résolvante. C'est là le but des règles de réécriture.

En logique classique, la règle de réécriture[1] est :

$$X : \wedge(Y) : \texttt{true} \rightsquigarrow X : Y$$

Considérons la résolvante suivante : $\wedge(p) : \wedge(q) : \texttt{true}$. En appliquant la règle de réécriture on obtient la nouvelle question : $\wedge(p) : q$. La nouvelle question formée, il faut reprendre la résolution à l'étape *sélection d'une clause* comme indiqué dans le schéma 4.1.

Les règles de réécriture sont déterministes et doivent être appliquées sur la résolvante jusqu'à ce qu'aucune d'entre elles ne soient plus applicables. Il n'y a jamais de backtrack possible sur une règle de réécriture, contrairement aux règles de résolution.

4.6 Terminaison de la résolution

La résolution se termine définitivement dans deux cas :

- Lorsqu'il n'y a plus de point de choix permettant de backtracker. Il s'agit alors d'un échec.
- Lorsque la résolvante atteint une *forme finale*. Il s'agit alors d'un succès.

En logique classique, la forme finale est `true`.

Remarquons qu'il est possible de continuer la résolution même en cas de succès afin de chercher d'autres solutions. C'est d'ailleurs un des paramètres du moteur d'inférence TARSKI.

4.7 Conclusion

Nous avons détaillé dans ce chapitre le fonctionnement général du moteur d'inférence de TARSKI. Nous avons dégagé la place particulière des règles de résolution qui permettent de généraliser le classique mécanisme de résolution de PROLOG. Dans notre formalisme, PROLOG est un cas particulier du mécanisme général : comme nous l'avons montré, il suffit de fournir au système les règles de résolution de la logique classique pour qu'il se comporte comme un PROLOG standard. Dans le chapitre suivant, nous allons développer le dernier élément nécessaire au bon fonctionnement du système : les techniques de contrôle.

[1] Cette règle correspond à la règle logique vue au chapitre 1 : $A \vee \emptyset \rightsquigarrow A$

Chapitre 5

Problèmes de contrôle

5.1 Contrôle lors de la sélection des règles

Les choses ne sont hélas pas aussi simples qu'il y paraîıt au premier abord. Le lecteur attentif a du se rendre compte que les règles précédentes n'implantent pas tout à fait correctement un PROLOG standard. Rappelons les règles générales de résolution qui implantent la logique classique :

$$\frac{A, ? \wedge (D) : B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C} \tag{5.1}$$

$$\frac{\wedge(D) : A, ?B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C} \tag{5.2}$$

Supposons maintenant que le fait soit $\wedge(p) : q$ et la question $\wedge(r) : q$. Si nous appliquons nos règles à la lettre, nous allons successivement construire la résolvante : $\wedge(p) : \wedge(r)$ en appliquant successivement les règles 5.1 puis 5.2. Mais supposons que par la suite la résolution échoue. Nous devons alors backtracker sur le dernier point de choix disponible. Or, il est parfaitement possible d'appliquer d'abord la règle 5.2 puis la règle 5.1, ce qui construit la résolvante : $\wedge(r) : \wedge(p)$; cela n'est pas admissible en PROLOG.

Le problème peut se révéler beaucoup plus critique. Dans le cas précédent, résoudre $\wedge(r) : \wedge(p)$ au lieu de $\wedge(p) : \wedge(r)$ n'est pas logiquement absurde. Mais il est des cas où le backtracking peut conduire à des résultats inadmissibles.

En particulier, l'ensemble des règle permettant d'implanter le KUT[1] est :

1.

$$\frac{KUT(X) : A, ?KUT(Y) : B \vdash ?KUT(Y) : C}{KUT(X) : A, ?B \vdash ?C} \text{ et } X \neq Y$$

2.

$$\frac{KUT(X) : A, ?B \vdash ?KUT(X) : C}{A, ?B \vdash ?C}$$

[1] **Attention !** Le KUT n'a **rien de commun** avec le *cut* PROLOG standard. Il n'est donné ici qu'à titre d'exemple. On peut se contenter pour l'instant de le considérer comme un opérateur "standard" dont le comportement syntaxique est défini par les règles qui suivent. Nous reviendrons sur la sémantique du KUT ultérieurement.

3.

$$\frac{A, ?KUT(X) : B \vdash ?KUT(X) : C}{A, ?B \vdash ?C}$$

Ce qui est sous-entendu ici doit bien être compris. La règle (1) échoue si l'on a par exemple un fait de la forme $KUT(1) : A$ et une question de la forme $KUT(1) : B$. Il n'est alors pas question de backtracker pour exécuter les règles (2) et (3). La règle (2) ne doit être utilisée que lorsque l'on se trouve dans un cas où le fait est de la forme $KUT(1) : A$ et la question $\wedge(D) : B$, donc quand les deux opérateurs de tête sont différents.

Bien souvent, les règles fournies par les logiciens sont ainsi ambigües. Or, ce qui est admissible dans une communication entre personnes partageant un certain nombre de connaissances communes l'est beaucoup moins vis à vis d'un système de résolution automatique. Il apparaîıt rapidement qu'un mécanisme de contrôle est nécessaire pour diriger l'exécution des règles. Plusieurs solutions sont envisageables. Nous allons les examiner.

5.1.1 Utilisation de l'extension procédurale

Il est bien souvent possible de spécifier une forme de contrôle en utilisant une extension procédurale pour chacune des règles de résolution. Ainsi on pourrait réécrire les règles (2) et (3) de KUT de la façon suivante :

$$\frac{KUT(X) : A, ?B \vdash ?KUT(X) : C}{A, ?B \vdash ?C} \text{ et } B \text{ n'est pas de la forme } KUT(X) : Y$$

$$\frac{A, ?KUT(X) : B \vdash ?KUT(X) : C}{A, ?B \vdash ?C} \text{ et } A \text{ n'est pas de la forme } KUT(X) : Y$$

De même, la seconde règle de la logique classique :

$$\frac{\wedge(D) : A, ?B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C}$$

pourrait être réécrite en :

$$\frac{\wedge(D) : A, ?B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C} \text{ et } B \text{ n'est pas de la forme } \wedge(X) : Y$$

Ceci résout effectivement le problème mais alourdit considérablement (et inutilement) la plupart des règles de résolution. D'autre part, les possibilités de contrôle restent faibles : Il est impossible de choisir explicitement quelle règle doit être effectuée après telle autre règle, etc...

5.1.2 Construction d'un graphe de résolution

Une autre solution consiste à enrichir le fonctionnement du moteur d'inférence en adjoignant à la structure des règles un graphe de résolution. Historiquement, cette technique était utilisée dans la première version implantée en C ([AG88]) (ainsi que dans les premières versions implantées en ADA) et dérivait directement de la technique utilisée sur l'interprète écrit en PROLOG ([ABFdC+86],[Esp87b]). Celui-ci créait un pseudo-graphe à l'aide du *cut* PROLOG, lors de la sélection des règles.

Nous allons décrire rapidement cette méthode. L'idée consiste à associer à chaque règle, en plus de son code, trois champs indiquant :

Règle 1 (R1)	$p, ?p \vdash ?true$
Règle 2 (R2)	$\frac{A,?\wedge(D):B \vdash ?\wedge(D):C}{A,?B \vdash ?C}$
Règle 3 (R3)	$\frac{\wedge(D):A,?B \vdash ?\wedge(D):C}{A,?B \vdash ?C}$
Règle 4 (R4)	$\frac{A,?\Diamond(X):B \vdash ?C}{A,?B \vdash ?C}$
Règle 5 (R5)	$\frac{\Box(X):A,?\Diamond(X):B \vdash ?\Diamond(X):C}{A,?B \vdash ?C}$

Tableau 5.1: Système élémentaire

- La règle à utiliser en cas de succès de l'exécution de la règle courante.
- La règle à choisir en cas d'échec immédiat de la règle courante.
- La règle à choisir en cas de backtrack (échec différé) de la règle courante.

On dispose ainsi d'un puissant moyen de contrôle. Nous allons montrer son fonctionnement sur un exemple ; considérons le système logique décrit par les règles du tableau 5.1.

Nous pouvons alors développer le graphe de la figure 5.1.

Expliquons en détail le principe de ce graphe. A chaque règle sont associés trois champs :

un champ *Succès*: Ce champ définit la règle qui sera exécutée après la règle courante si elle réussit. La règle R2, par exemple, si elle réussit, sera suivie de la règle R1 pour vérifier que la résolution n'est pas terminée. La règle R1 au contraire à un champ à 0, car si elle réussit, la résolution est terminée.

Un champ *Echec*: Il définit la règle à utiliser si la règle courante a échoué immédiatement. On dit qu'une règle échoue immédiatement si la forme du fait sélectionnée et la forme du fait représentée dans la règle ne correspondent pas (ou respectivement, si la forme de la question courante et la forme de la question représentée dans la règle ne correspondent pas). Ainsi, si le fait sélectionné est $\Box(X) : p$ et que la règle courante est R3 (forme du fait $\wedge(D) : A$), alors il y a échec immédiat (ou échec sur la forme).

Dans le cas de la règle R3, un échec sur la forme branche simplement sur la règle suivante (R4). Dans le cas de la règle R4, s'il y a échec sur la forme, il est inutile de sélectionner R5, car la règle R5 est un cas particulier (pour les conditions sur la forme du fait et de la question). Il faut donc bien distinguer les cas d'échec sur la forme (échec immédiat) et les cas de retour arrière.

Un champ *Backtrack*: Il définit la règle qui sera sélectionnée en cas de backtrack. Il faut bien distinguer ce cas du précédent : il n'y a backtrack que s'il y a eu succès sur l'unification des formes. Ainsi, si nous prenons l'exemple de la règle R2, s'il y a backtrack, cela signifie que la question est de la forme $\wedge(D) : B$. Dans ce cas, comme nous l'avons fait remarquer précédemment, nous ne souhaitons pas utiliser la règle R3, ni aucune autre règle. Donc le champ backtrack est à 0.

En revanche, si la règle R4 backtracke, il faut essayer R5, car R4 est une forme plus générale de R5.

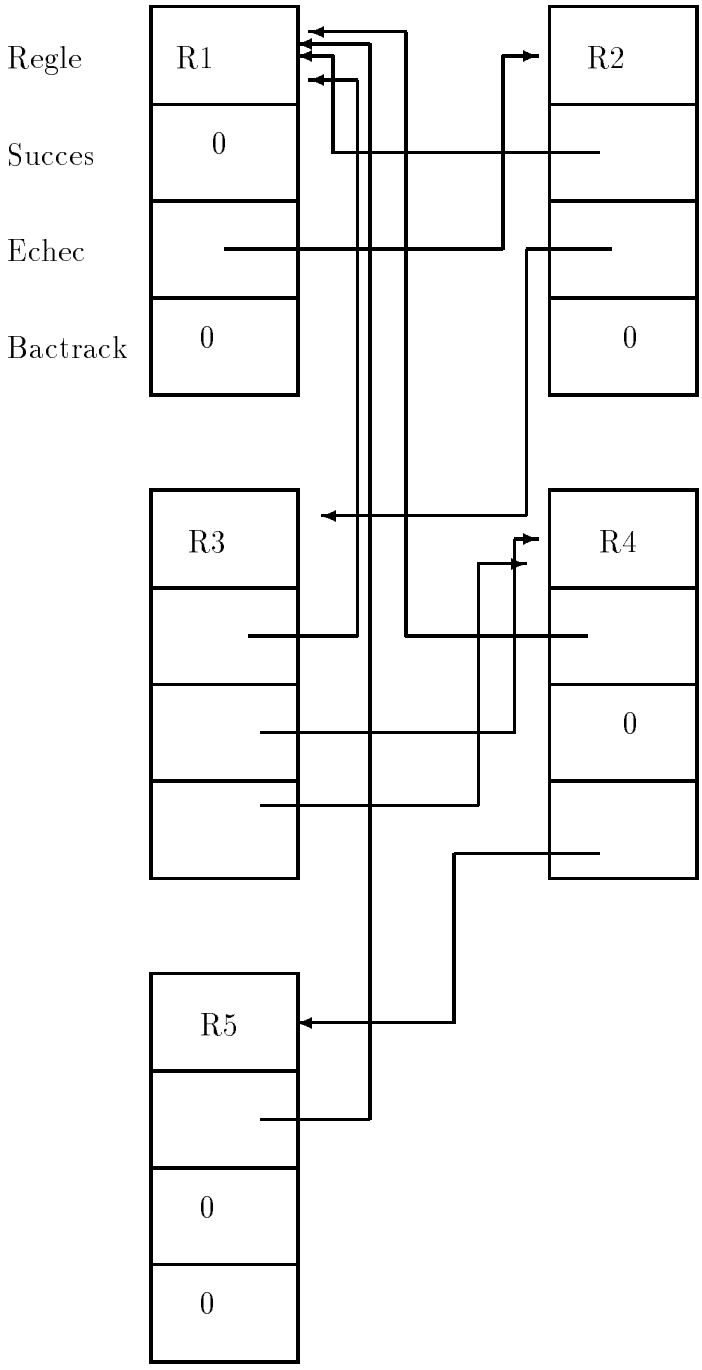


Figure 5.1: Graphe de règles

Ainsi, nous disposons d'un moyen de contrôle puissant permettant de spécifier exactement ce que le système doit faire lors de l'exécution et du backtracking. Ce mécanisme est également plus efficace car les informations données par l'échec ou le succès sur la forme du fait et de la question pour une règle permettent d'être plus sélectif pour les règles suivantes.

5.1.3 Enumération exhaustive et indexation

En fait, la méthode du graphe des règles, est à la fois trop compliquée et inefficace. C'est elle qui était en grande partie responsable de la lenteur des implantations précédentes du langage. Déjà, sur l'exemple précédent, construire le graphe est relativement délicat et demande du soin.

Il faut reprendre le problème à la base pour se rendre compte que l'on peut faire plus simple et plus efficace.

Pour une logique donnée, nous avons un nombre fini d'opérateurs que nous noterons m, plus en général l'opérateur classique $\wedge$. D'autre part, un but (ou un fait) est une séquence linéaire de ces opérateurs, qui se termine par un prédicat. Donc, à un instant donné, un fait ou un but peuvent commencer par un de ces $m+2$ objets. Donc, l'ensemble des règles de logique peuvent être exhaustivement développées en un tableau comportant au plus $(m+2)^2$ cases. Ce tableau est un tableau à deux index, l'un d'entre eux est l'opérateur en début de fait et l'autre l'opérateur en début de but. Chaque case du tableau contient la liste des règles exécutables, et bien souvent, nous allons le voir, il n'y en à qu'une.

Nous allons maintenant introduire une système de règles de résolution afin de présenter sur un exemple concret les différents problèmes de l'implantation. Nous appellerons ce système S1[2]. Ce système comprendra en plus de l'opérateur classique $\wedge$ les opérateurs $\Box$, $\Diamond$ et $\Diamond_I$. $\wedge$, $\Box$ et $\Diamond$ prennent un argument, $\Diamond_I$ prend deux arguments.

Règles de terminaison : La seule règle de terminaison est celle de la logique classique :

$$p, ?p \vdash ?\texttt{true} \tag{5.3}$$

Règles générales : Les deux premières règles générales sont les règles de résolution de la logique classique :

$$\frac{A, ? \wedge (D) : B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C} \tag{5.4}$$

$$\frac{\wedge(D) : A, ?B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C} \tag{5.5}$$

Les règles suivantes sont les règles spécifiques à S1 :

$$\frac{A, ?\Diamond(X) : B \vdash ?C}{A, ?B \vdash ?C} \tag{5.6}$$

$$\frac{\Diamond_I(X, I) : A, ?\Diamond(X) : B \vdash ?\Diamond_I(X, I) : C}{A, ?\Diamond(X) : B \vdash ?C} \tag{5.7}$$

$$\frac{\Diamond_I(X, I) : A, ?\Diamond_I(X, I) : B \vdash ?\Diamond_I(X, I) : C}{A, ?B \vdash ?C} \tag{5.8}$$

[2] Ce système n'a pas été choisi au hasard. Il permet en particulier de faire de la résolution pour le système multi-S4, système qui permet de représenter des concepts épistémiques et temporels. Les techniques de traduction de multi-S4 vers S1 et les preuves de complétude se trouvent dans [BFdCH88] et [dCH88].

$$\frac{\Box(X) : A, ?\Diamond(X) : B \vdash ?\Diamond(X) : C}{\Box(X) : A, ?B \vdash ?C} \tag{5.9}$$

$$\frac{\Box(X) : A, ?\Diamond_I(X, I) : B \vdash ?\Diamond_I(X, I) : C}{\Box(X) : A, ?B \vdash ?C} \tag{5.10}$$

$$\frac{\Box(X) : A, ?\Diamond(X) : B \vdash ?\Diamond(X) : C}{A, ?\Diamond(X) : B \vdash ?C} \tag{5.11}$$

$$\frac{\Box(X) : A, ?B \vdash ?C}{A, ?B \vdash ?C} \tag{5.12}$$

Règles de réécriture : La règle de réécriture de la logique classique reste valable :

$$X : \wedge(Y) : \texttt{true} \rightsquigarrow X : Y$$

Les règles suivantes sont spécifiques à S1 :

$$X : \Box(Y) : true \rightsquigarrow X$$

$$X : \Diamond(Y) : true \rightsquigarrow X$$

$$X : \Diamond_I(Y, I) : true \rightsquigarrow X$$

Nous allons nous intéresser à l'ensemble des règles générales et des règles de terminaison (les règles de réécriture apparaissent à un autre niveau et sont déterministes).

La méthode du développement exhaustif et de l'indexation nous donne le tableau 5.2.

Revenons sur la construction de ce tableau. Nous allons examiner en détail comment nous avons obtenu certains résultats :

- Considérons tout d'abord les trois premières lignes, qui correspondent aux cas où le fait est un prédicat. Dans ce cas, cinq cas seulement sont possibles :

  La question est un prédicat : seule la règle 5.3 est susceptible d'être utilisable. C'est la première ligne du tableau.

  La question commence par un $\wedge$: la règle 5.5 est seule applicable. C'est la deuxième ligne du tableau.

  La question commence par un $\Diamond$: la règle 5.6 est seule applicable. C'est la troisième ligne du tableau.

  La question commence par $\Diamond_I$: il n'existe aucune règle qui s'applique à ce cas. En effet, les deux seules règles susceptibles de s'appliquer sont 5.8 et 5.10. Mais 5.8 ne s'applique que si le fait commence par un $\Diamond_I$ et 5.10 ne s'applique que si le fait commence par un $\Box$.

  La question commence par un $\Box$: aucune règle ne s'applique (par construction des règles de résolution de S1, il est impossible qu'il y ait un $\Box$ dans une question).

  Ainsi, nous voyons qu'il n'y a que 3 cas possibles sur les cinq, et qu'il n'y a pour chacun de ces cas qu'une seule règle exécutable. C'est la un point très important.

- Nous allons nous intéresser aux trois dernières cases qui correspondent au cas où l'opérateur $\Diamond_I$ est en tête du fait. Il y a a priori cinq cas possibles :

Fait	*Question*	*Règles*
Pred	Pred	$p, ?p \vdash ?true$
Pred	$\wedge$	$\frac{A,?\wedge(X):B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
Pred	$\Diamond$	$\frac{A,?\Diamond(X):B\vdash ?C}{A,?B\vdash ?C}$
$\wedge$	Pred	$\frac{\wedge(X):A,?B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
$\wedge$	$\wedge$	$\frac{\wedge(X):A,?\wedge(Y):B\vdash ?\wedge(Y):C}{\wedge(X):A,?B\vdash ?C}$
$\wedge$	$\Diamond$	$\frac{\wedge(X):A,?\Diamond(Y):B\vdash ?\wedge(X):C}{A,?\Diamond(Y):B\vdash ?C}$
$\wedge$	$\Diamond_I$	$\frac{\wedge(X):A,?\Diamond_I(Y,I):B\vdash ?\wedge(X):C}{A,?\Diamond_I(Y,I):B\vdash ?C}$
$\Box$	Pred	$\frac{\Box(X):A,?B\vdash ?C}{A,?B\vdash ?C}$
$\Box$	$\wedge$	$\frac{\Box(Y):A,?\wedge(X):B\vdash ?\wedge(X):C}{\Box(Y):A,?B\vdash ?C}$
$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(Y):B\vdash ?C}{A,?\Diamond(Y):B\vdash ?C}$
		$\frac{\Box(Y):A,?\Diamond(X):B\vdash ?C}{\Box(Y):A,?B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{A,?\Diamond(X):B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{\Box(X):A,?B\vdash ?C}$
$\Box$	$\Diamond_I$	$\frac{\Box(X):A,?\Diamond_I(Y,I):B\vdash ?C}{A,?\Diamond_I(Y,I):B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond_I(X,I):B\vdash ?\Diamond_I(X,I):C}{\Box(X):A,?B\vdash ?C}$
$\Diamond_I$	$\wedge$	$\frac{\Diamond_I(Y,I):A,?\wedge(X):B\vdash ?\wedge(X):C}{\Diamond_I(Y,I):A,?B\vdash ?C}$
$\Diamond_I$	$\Diamond$	$\frac{\Diamond_I(Y,I):A,?\Diamond(X):B\vdash ?C}{\Diamond_I(Y,I):A,?B\vdash ?C}$
		$\frac{\Diamond_I(X,I):A,?\Diamond(X):B\vdash ?\Diamond_I(X,I):C}{A,?\Diamond(X):B\vdash ?C}$
$\Diamond_I$	$\Diamond_I$	$\frac{\Diamond_I(X,I):A,?\Diamond_I(X,I):B\vdash ?\Diamond_I(X,I):C}{A,?B\vdash ?C}$

Tableau 5.2: Développement et indexation des règles de S1

La question commence par un prédicat : aucune règle ne peut s'appliquer. Les deux seules règles susceptibles de s'appliquer sont 5.7 et 5.8, mais dans le premier cas, la question doit commencer par $\Diamond$, et dans le second cas, elle doit commencer par $\Diamond_I$.

La question commence par $\wedge$: on peut appliquer la règle 5.5, ce qui nous donne l'ante-pénultième case du tableau.

La question commence par un $\Diamond$: il s'agit là d'un cas intéressant. En effet, deux règles sont applicables : 5.6 et 5.7. 5.6 nous donnent le premier élément de l'avant-dernière case du tableau, et 5.7 le second élément de l'avant dernière case.

La question commence par un $\Diamond_I$: Une seule règle applicable : 5.8, qui donne la dernière case du tableau.

La question commence par un $\Box$: Aucune règle applicable.

Le lecteur curieux pourra ainsi construire toutes les autres cases du tableau. On peut en particulier attirer l'attention sur le cas où le fait commence par un $\Box$. En effet, si la question commence par un $\Diamond$, il n'y a pas moins de quatre règles utilisables simultanément.

Cette méthode présente de nombreux avantages :

- L'accès à la (ou aux) bonne(s) règle(s) est instantané. Le premier élément de la question et le premier élément du fait étant donnés, l'accès se fait directement par une méthode d'index. Il n'est plus besoin de réaliser des unifications coûteuses en temps.
- Dans de nombreux cas, il n'y a qu'une seule règle applicable pour une question et un fait donnés. Il est donc inutile de mémoriser le point de choix, puisque tout backtrack est impossible.
- Le tableau oblige à représenter clairement l'ensemble des relations entre les éléments et permet de supprimer des essais infructueux. Ainsi, si le fait commence par un $\Diamond_I$ et le fait est réduit à un prédicat, le système backtrackera immédiatement, sachant qu'il n'y a aucune règle applicable.
- Le tableau permet de représenter explicitement les règles qui sont parallélisables. Nous allons tout de suite discuter ce point plus en détail.

5.1.4 Le parallélisme intrinsèque de la résolution

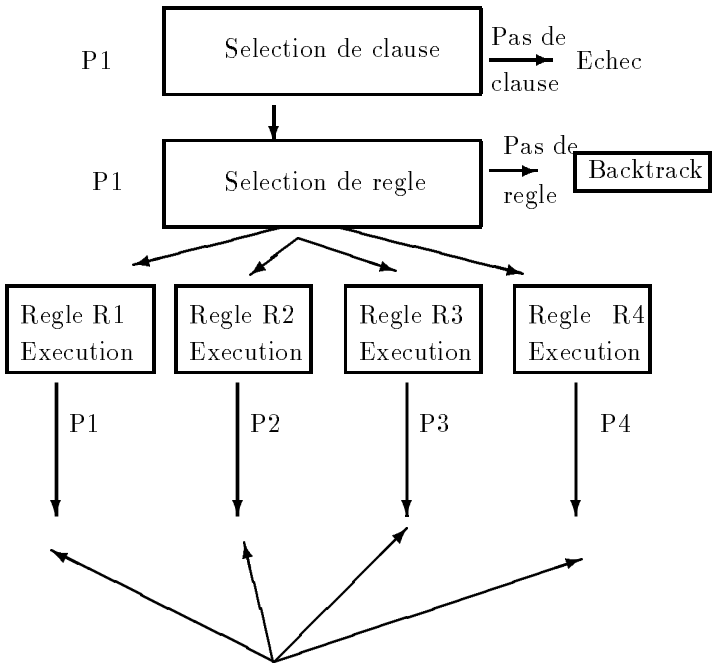
Si nous regardons dans le tableau 5.2 la ligne qui indique les règles de résolution admissibles dans le cas ou l'opérateur du fait est $\Box$ et l'opérateur de la question $\Diamond$, on constate que quatre règles différentes sont applicables (voir tableau 5.3).

Il est alors naturel de songer à diviser en quatre l'arbre de résolution en faisant exécuter chaque partie de la résolution par un agent différent (voir figure 5.2).

Ce parallélisme, implicite à la résolution même est un excellent parallélisme. Nous entendons par là qu'il n'existe aucun effet de bord possible. Une fois que la résolution a commencé sur une des quatre branches, l'agent qui l'exécute est indépendant des autres agents. Il n'y a pas de problème de "remontée des environnements" puisqu'il n'y a aucun environnement à remonter : la résolution qui s'exécute sur un processeur distant peut soit réussir, soit échouer, mais cela n'a aucune influence quant à la suite de la résolution sur le processeur père, et de fait aucune information n'est retournée, hormis l'information de terminaison de la résolution.

	Fait	Question	Règles
R1	$\Box$	$\Diamond$	$\frac{\Box(X){:}A,?\Diamond(X){:}B \vdash ?C}{A,?\Diamond(X){:}B \vdash ?C}$
R2	$\Box$	$\Diamond$	$\frac{\Box(X){:}A,?\Diamond(X){:}B \vdash ?C}{\Box(X){:}A,?B \vdash ?C}$
R3	$\Box$	$\Diamond$	$\frac{\Box(X){:}A,?\Diamond(X){:}B \vdash ?\Diamond(X){:}C}{A,?\Diamond(X){:}B \vdash ?C}$
R4	$\Box$	$\Diamond$	$\frac{\Box(X){:}A,?\Diamond(X){:}B \vdash ?\Diamond(X){:}C}{\Box(X){:}A,?B \vdash ?C}$

Tableau 5.3: Règles $\Box$ contre $\Diamond$



Chaque processeur continue la resolution sur un quart de l'arbre total

Figure 5.2:

On est ainsi dans un cas bien plus favorable que le parallélisme ET dependant PROLOG qui impose de transférer les valeurs des variables au fur et à mesure de leur liaison, et nous sommes aussi dans un cas plus favorable que le parallélisme OU PROLOG, qui doit en général tenir compte des opérateurs de contrôle comme le `cut`, ou des prédicats d'entrée-sortie.

Ici, le parallélisme est **totalement transparent pour l'utilisateur**. C'est là un point très important pour le confort d'utilisation du système. Il sera possible d'utiliser TARSKI comme un système parallèle ou comme un système séquentiel sans rien changer à la forme du programme et sans que les résultats soient modifiés en quoi que ce soit.

5.2 Contrôle lors de l'exécution du programme

5.2.1 *cut* classique

PROLOG offre à l'utilisateur une technique de contrôle classique, le *cut*. Afin de conserver à TARSKI une puissance équivalente à celle de PROLOG, le cut a été également implanté. La sémantique du cut est la même tant que les clauses logiques sont des clauses de logique classique. En revanche, sa sémantique est moins claire quand il s'applique dans le contexte des clauses de Horn généralisées.

La seule chose que l'on puisse garantir est que le *cut* supprimera l'ensemble des points de choix entre le moment où il est exécuté et le point de choix correspondant à la clause à laquelle il appartient (incluse). Ceci correspond bien au *cut* classique sur des clauses classiques.

5.2.2 KUT

La logique elle-même peut être un puissant moyen de contrôle ; nous allons dans cette section reprendre les règles décrivant l'implantation d'un opérateur logique appelé KUT et nous intéresser à sa sémantique.

Le KUT est implanté par les règles suivantes

Règles générales : Les règles générales supplémentaires apparaissent dans le tableau 5.4.

Règle de réécriture :

$$KUT(X) : true \rightsquigarrow \texttt{true}$$

Quel est l'intérêt du KUT et comment fonctionne-t-il? Le but du KUT est de contrôler l'utilisation des clauses de la base de fait : il suffit de préfixer une clause par un $KUT(i)$. i est une constante qui aura la même valeur pour tout paquet de clauses dont on veut n'utiliser qu'une seule clause dans une résolution donnée.

Voyons cela sur un exemple. Supposons que la base de clauses comporte :

$$KUT(1) : \wedge(p) : q$$

$$KUT(2) : \wedge(q) : p$$

Supposons que le but soit q. On sélectionne la première des deux clauses. Puis, on utilise la règle R1 et on empile dans la résolvante $KUT(1)$, enfin on empile $\wedge(p)$, puis `true` (résolution classique). Lors de la réécriture, on obtient la nouvelle question $KUT(1) : p$.

On sélectionne alors la deuxième clause. Il faut alors appliquer la règle R9. Les deux arguments des KUT sont différents, donc on empile $KUT(1)$ dans la résolvante. Le fait

	Fait	Question	Règles
R1	KUT	Pred	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R2	KUT	$\wedge$	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R3	KUT	$\Diamond$	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R4	KUT	$\Diamond_I$	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R5	Pred	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R6	$\wedge$	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R7	$\Box$	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R8	$\Diamond_I$	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R9	KUT	KUT	$\frac{KUT(Y):A,?KUT(X):B\vdash?KUT(X):C}{KUT(Y):A,?B\vdash?C}$ et $X \neq Y$

Tableau 5.4: Implantation du KUT

est inchangé, et la question devient p. On applique alors R1 et on empile $KUT(2)$ dans la résolvante. Le fait est alors $\wedge(q) : p$ et la question p. Après une résolution classique, la résolvante devient finalement : $KUT(1) : KUT(2) : \wedge(q) :$ `true`.

La nouvelle question est donc après réécriture : $KUT(1) : KUT(2) : q$. On sélectionne alors le fait $KUT(1) : ET(p) : q$. Il faut donc appliquer la règle R9. Mais ici les arguments des KUT sont égaux, donc la règle échoue. Comme il n'y a aucun point de backtrack, la résolution échoue.

On constate ainsi que le KUT bien employé est un moyen extrêmement puissant de supprimer les boucles infinies dans lesquelles peut aisément entrer un interprète de type PROLOG. Il faut cependant prendre soin de ne l'employer qu'à bon escient, sous peine de voir des résolutions échouer alors qu'elles devraient réussir. Si l'on considère par exemple la base de clauses suivantes :

$$\wedge(p) : \wedge(q) : t$$

$$\wedge(q) : p$$

$$KUT(1) : q$$

et que l'on pose la question t, la résolution échouera.

5.3 Exemple

Nous allons donner un exemple complet de résolution avec TARSKI. Pour cela, nous allons considérer une résolution effectuée en logique classique. Rappelons les trois règles :

1. La règle de terminaison :

$$p, ?p \vdash ?\texttt{true}$$

2. La règle de réduction de la question :

$$\frac{A, ? \wedge (D) : B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C}$$

3. La règle de réduction du fait :

$$\frac{\wedge(D) : A, ?B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C}$$

Nous allons considérer l'ensemble de clauses logiques suivants :

1.

$$p \rightarrow q$$

2.

$$r \rightarrow q$$

3.

$$r$$

La question sera :

$$p \wedge q$$

Nous allons simplement utiliser une trace commentée de l'exécution pour développer cet exemple :

Mise en forme des clauses : Nous voyons ci-dessous l'écriture au format TARSKI de la base de clause et de la question comme décrit dans les sections précédentes (l'opérateur $\wedge$ est traduit par un #IF. La clause 1 se lit : si p alors q, etc...).

```
Tarski $Revision: 1.2 $ ready.
>#IF(p):q.
>#IF(r):q.
>r.
>#IF(r):q?
```

Sélection de clause : Le premier sous-but à résoudre est q, donc le moteur sélectionne la première clause dont la tête est q, c'est à dire la clause 1 :

```
Niveau : 1
selection de clause
clause selectionnee:  1
```

sélection exécution d'une règle : Le fait est de la forme $\wedge(D) : A$ et la question de la forme $\wedge(E) : B$. Suivant le tableau 5.2, la règle applicable est donc la règle 2. Cette règle ajoute $\wedge(r)$ dans la résolvante et réduit la question :

```
Niveau:  2
selection de regle
fait:#IF(p):q
question:#IF(r):q
regle a executer: 2
execution de regle
reussi
resolvante: #IF(r):
```

Sélection et exécution d'une règle : Le fait est toujours de la forme $\wedge(D) : A$, mais la question est maintenant réduite à un prédicat. D'après le tableau 5.2, la règle applicable est donc la règle 3. Cette règle va rajouter $\wedge(D) : A$ dans la résolvante et réduire le fait :

```
Niveau:  3
selection de regle
fait:#IF(p):q
question:q
regle a executer: 3
execution de regle
reussi
resolvante: #IF(r):#IF(p):
```

Sélection et exécution de règle : Le fait et la question sont maintenant réduits à un prédicat. Donc la règle 1 est applicable. Celle ci réussit puisque les deux prédicats sont unifiables :

```
Niveau:  4
selection de regle
fait:q
question:q
regle a executer: 1
execution de regle
reussi
fin partielle
```

Réécriture : La règle 1 était une règle de terminaison partielle, donc la résolvante est close et il faut reformer une nouvelle question. On utilise donc la règle de réécriture :

```
Reecriture
Resolvante: #IF(r):#IF(p):
Nouvelle question: #IF(r):p
```

Sélection de clause et backtrack : Le système doit maintenant résoudre d'abord le nouveau sous-but p. Mais il n'y a aucune clause dans la base dont la tête est p. Le mécanisme va donc déclencher un backtrack.

```
selection de clause
pas de clause
backtrack a partir du niveau:  4
```

Sélection de règle après backtrack : Le système est donc revenu au point de choix précédent. Il s'agit d'un point de choix sur une règle. Il va donc tenter de sélectionner la règle suivante. Mais le tableau 5.2 nous montre que le cas **Pred/Pred** n'a qu'une seule règle applicable la règle 1. Donc le système ne peut trouver de règle suivante et va donc backtracker à nouveau.

```
selection de regle. Regle precedente: 1
plus de regle
backtrack a partir du niveau:  3
```

Suite de backtrack : Le système parcourt successivement tous les points de choix précédents à la recherche d'une alternative. La règle 3, comme la règle 1, est la seule règle applicable au cas considéré de même que la règle 2. Donc le système remonte jusqu'au point de choix précédent :

```
selection de regle. Regle precedente: 3
plus de regle
backtrack a partir du niveau:  2
selection de regle. Regle precedente: 2
plus de regle
backtrack a partir du niveau:  1
```

Sélection de la clause suivante : Le système vient de revenir au point de choix du niveau 1, qui est le point de choix sur la clause 1. La clause 1 a un successeur dans la base, la clause 2 qui a aussi pour prédicat de tête q :

```
Niveau : 1
selection de clause. Clause precedente: 1
clause selectionnee:  2
```

Nous n'allons pas détailler la suite de la résolution. Elle se poursuit exactement suivant le même principe jusqu'à l'instant où la question est complètement réduite.

```
Niveau:  2
selection de regle
fait:#IF(r):q
question:#IF(r):q
regle a executer: 2
execution de regle
reussi
resolvante: #IF(r):

Niveau:  3
```

```
selection de regle
fait:#IF(r):q
question:q
regle a executer: 3
execution de regle
reussi
resolvante: #IF(r):#IF(r):

Niveau:  4
selection de regle
fait:q
question:q
regle a executer: 1
execution de regle
fin partielle

Reecriture
resolvante: #IF(r):#IF(r):
Nouvelle question:#IF(r):r

Niveau: 5
selection de clause
clause selectionnee:  3

Niveau: 6
selection de regle
fait:r
question:#IF(r):r
regle a executer: 2
execution de regle
reussi
resolvante: #IF(r):

Niveau:  7
selection de regle
fait:r
question:r
regle a executer: 1
execution de regle
fin partielle

Reecriture
resolvante: #IF(r):
Nouvelle question: r

Niveau : 8
selection de clause
```

```
clause selectionnee:  3

Niveau : 9
selection de regle
fait:r
question:r
regle a executer: 1
execution de regle
fin partielle

Reecriture
resolvante:

Ok
```

Partie III

Conception

Chapitre 6

Conception

6.1 Introduction

Parler de machine abstraite en PROLOG, force irrémédiablement à comparer ce que l'on fait à la machine de Warren (WAM).

La machine TARSKI n'est pas à proprement parler une extension de la machine de Warren. Elle ne reprend pas exactement les mêmes structures de pile, elle a davantage été conçue pour être codée dans un langage de haut niveau plutôt que dans un langage de type assembleur. Enfin, le but de la machine de Warren est de faire de la compilation de clauses pour améliorer la vitesse de résolution.

Le but de la machine abstraite TARSKI est de réduire l'écart entre la forme logique des règles et leur programmation dans un langage efficace. Cependant, nous avons également souhaité l'utiliser comme spécifications du programme. Ainsi, elle doit également permettre de coder l'ensemble des processus nécessaires à la résolution, y compris les processus de base de tout langage de programmation logique comme le retour-arrière ou l'unification.

Elle opère aussi sur des piles, a un jeu d'instructions et des registres. C'est en ce sens qu'elle est proche parente de la machine de Warren, comme elle est proche de toutes les machines abstraites. La technique de compilation que nous avons choisie se rapproche des techniques utilisées dans certains LISP compilés comme KCL ([YH86]) : le langage de la machine abstraite est en fait un langage de haut niveau (C dans le cas de KCL, ADA dans le cas de TARSKI) auquel on adjoint un certain nombre de primitives ou macro-instructions. Nous allons détailler ce mécanisme dans les prochains paragraphes.

6.2 Choix effectués

Toute conception d'un système de résolution PROLOG amène à faire des choix qui déterminent l'architecture du système : on ne peut repousser jusqu'à la phase d'implantation certaines décisions.

6.2.1 Partage de structures contre copie de structures

Nous avons discuté dans le chapitre 2 des différentes méthodes classiquement utilisées pour effectuer la résolution PROLOG, et nous avons en particulier discuté des méthodes de partage de structures et de copie (partielle ou totale) de structures. Ce choix doit être fait

dès la conception car il subordonne l'ensemble des techniques d'unification et de résolution du moteur, et donc les objets mêmes qui seront manipulés par le système.

Nous avons choisi le partage de structures, principalement pour des raisons d'efficacité et surtout de simplicité du développement. D'autre part, comme le suggère Michel van Caneghem[vC86] :

> Le partage de données est plus adapté lorsque l'on réalise un interpréteur, la copie de données plus adapté lorsque l'on réalise un compilateur.

Le système TARSKI étant interprété au niveau de l'exécution des clauses[1], il nous a semblé préférable d'utiliser la technique du partage de structures.

Cette technique place des contraintes sur la forme des règles de résolution que peut traiter la machine, et donc des place des limitations sur les logiques que l'on est capable de traiter. Ces contraintes qui auraient pu être partiellement évitées si l'on avait adopté la méthode de recopie totale à la place de la méthode de partage de structures. Signalons pour notre décharge que cette méthode n'est utilisée dans aucune implantation classique de PROLOG en raison de son coût.

Notons également qu'il est relativement facile de réécrire le prototype actuel pour intégrer une technique de copie de structures à la place du partage de structures.

6.2.2 Les contraintes du parallélisme

Nous avons souhaité depuis le départ utiliser le parallélisme inhérent à la résolution sur des clauses de Horn générales. Nous devons ici choisir quelle méthode de parallélisme nous allons implanter, car cela influera sur la conception de notre système.

Un certain nombre de contraintes étaient imposées par l'environnement "logiciel et matériel" de ce travail. Une de ces contraintes était la non disponibilité de machines parallèles. Dans ces conditions, la seule façon de réaliser un système parallèle était d'utiliser des stations de travail en réseau, chaque station étant considérée comme un agent.

Enfin, nous souhaitions que le système parallèle soit aussi simple que possible. Le but de ce travail était plus une étude de faisabilité de la parallélisation qu'une implantation véritablement efficace[2].

Nous avons discuté dans le chapitre 2 des différents modèles de parallélisme classiquement utilisés en PROLOG ; de tous ces modèles, celui qui nous a paru le plus proche de ce que nous souhaitions faire était le modèle Kabu-Wake.

Rappelons que dans le système Kabu-Wake, chaque processeur se comporte comme un processeur standard exécutant un PROLOG interprété classique. Lorsqu'il a le choix entre n clauses, il crée un point de choix et continue la résolution normalement sur la première des alternatives. Si un processeur libre se manifeste par la suite, le processeur actif lui transmet une copie de l'ensemble des piles telles qu'elles étaient au moment de la création du point de choix, et le processeur libre va alors effectuer la résolution sur la suite de l'arbre. En un mot, le processeur libre reprend la résolution comme l'aurait fait le processeur actif en cas de retour arrière sur le point de choix.

[1] Il serait intéressant de développer un compilateur au sens de Warren pour les clauses, mais la tâche n'est pas aisé. En effet, en PROLOG, il n'y a pas de possibilités d'extension syntaxique du langage. Or, avec TARSKI, il est possible de paramétrer le langage avec de nouveaux opérateurs. Réaliser dans ces conditions une extension de la machine de Warren capable d'accepter tous les langages possibles semble relativement complexe.

[2] Faire du parallélisme efficace sur station de travail relié par un réseau ETHERNET à 10Mbits relèverait de l'optimisme le plus profond !

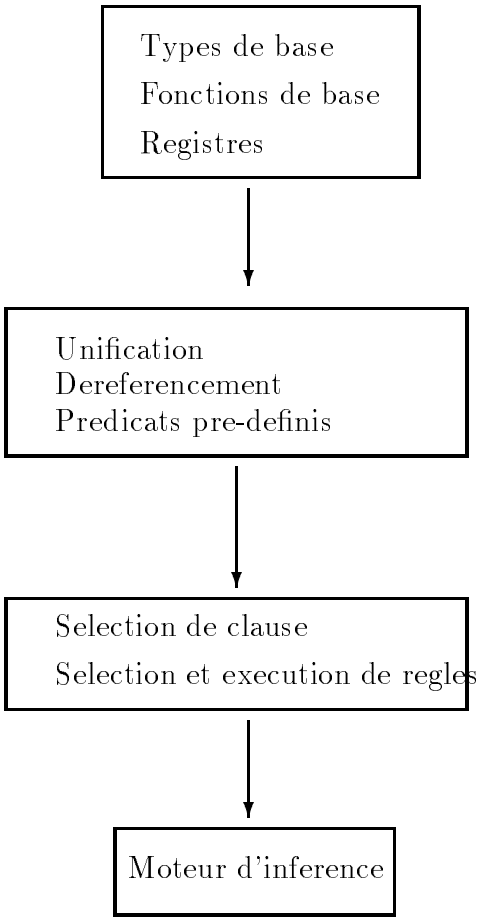


Figure 6.1: Architecture du système

Dans notre cas, il nous suffit de modifier très légèrement le mécanisme en remplaçant le point de choix au niveau des clauses par un point de choix au niveau des règles de résolution.

Ce mécanisme, ainsi que l'architecture sur laquelle nous devons l'implanter, nous impose quelques contraintes au niveau des références entre les objets qui devront être des références "pures", c'est à dire indépendantes de la machine sur laquelle s'exécute une résolution : ainsi, il sera possible de transférer les différents objets d'une machine à l'autre. Pour le reste, le système Kabu-Wake impose peu de restrictions au développement du système : le système séquentiel une fois testé et achevé, il sera possible de passer simplement au modèle parallèle : c'est la force des modèles multi-séquentiels.

6.3 Architecture générale

Nous souhaitons organiser l'architecture générale de notre système de la façon suivante (voir figure 6.1) :

- Un ensemble de types de base, de fonctions, et de variables globales formeront une première machine abstraite. Cette machine (la machine de niveau 0) permettra de mani-

puler les différents objets (copie, extraction de champs. etc...)

- Sur cette première machine abstraite, nous construisons une seconde machine abstraite (la machine de niveau 1) qui implantera les fonctions de haut niveau nécessaires à l'écriture des règles de résolution, ainsi qu'aux mécanismes de sélection de règles et de clauses (unification déréférencement, manipulation).

- Un troisième niveau comprendra la sélection et l'exécution des règles, et la sélection des clauses. C'est la machine de niveau 2.

- Le moteur d'inférence sera la quatrième et dernière partie du système.

Nous souhaitons également garder le système aussi modulaire et paramétrable que possible. En particulier, les fonctions de sélection de clause, de sélection de règles et d'exécution des règles devront être parfaitement transparentes et doivent pouvoir être modifiées en permanence sans que cela influe en rien sur le reste du système.

6.4 La machine de niveau 0

La machine de base définira les types de base, leur structure et permettra la manipulation des différents champs.

La structure générale adoptée pour chaque ensemble d'objets est une structure de "pile". Par pile, nous entendons un mécanisme de stockage permettant de gérer simplement la création et la destruction d'objets, en particulier en cas de backtrack. Ce mécanisme devra également être aussi transparent que possible. La machine de niveau 0 fournira donc également les opérations indispensables à la manipulation de ces différentes piles. TARSKI utilisera les piles suivantes que nous allons détailler :

- La pile des objets de base
- La pile des clauses
- La pile des environnements
- La pile de la résolvante
- La pile de la question
- La pile des points de choix
- La pile de trainée
- La pile des opérandes
- La table des noms de prédicats

Enfin, la machine de niveau 0 définit également les registres de la machine. La valeur des piles et la valeur des registres de la machine à un instant donné *déterminent complètement l'état de la machine*. Ceci est capital pour le parallélisme, car on pourra transférer la résolution d'un agent à un autre agent par simple copie des piles et des registres de la machine.

6.4.1 Les objets de base

Par objet de base, nous entendons l'ensemble des objets qui peuvent apparaîitre dans une clause TARSKI et qui vérifient la syntaxe défini dans le chapitre 3. Nous les opposerons aux objets généraux que manipule le programme qui sont les clauses, les environnements, la résolvante, etc...

Les entiers : A priori, TARSKI a été essentiellement conçu pour faire de la manipulation symbolique : il serait même possible de se passer des entiers pour l'opérateur $\diamond_I$, la constante de Skolem de l'opérateur pouvant être symbolique. Cependant, il est clair que nous devons au moins implanter un certain nombre de prédicats arithmétiques prédéfinis si nous voulons que notre système puisse exécuter certains benchmarks pour le comparer aux PROLOG standards.

Les flottants : Les flottants sont en revanche obligatoire, pour une raison simple : la volonté de pouvoir implanter certaines logiques comme la logique floue dont les opérateurs ont besoin d'arguments qui sont des nombres réels.

Les paires pointées : Tout système symbolique doit être capable de manipuler des listes. Nous avons adopté le mécanisme classique des listes à base de paires pointées `car` et `cdr`.

Les prédicats et les constantes symboliques : Nous avons choisi de considérer les constantes symboliques comme des prédicats sans argument.

Les opérateurs : Les opérateurs étant peu nombreux pour un système donné, et leur nom devant être connu à la compilation, leur traitement devra se faire de façon plus efficace que pour les noms de prédicats.

Les variables : Une variable, en partage de structures, se réduit à son déplacement à l'intérieur de la clause.

Les caractères : Les caractères nous permettront d'implanter les chaîines de caractères sous la forme de liste de caractères (implantation de type C-PROLOG). Ce ne sont pas des objets indispensables, et leur implantation effective pourra se faire ultérieurement.

Nous discuterons de l'implantation pratique de chacun des objets dans le chapitre 8.

6.4.2 La pile des opérandes

Un objet, tel un prédicat ou un opérateur, ayant un nombre variable d'arguments, il est impossible de stocker dans la structure de l'objet lui-même l'ensemble de ces opérandes. Il existe plusieurs techniques pour traiter ce problème[3]. Pour notre part, un objet ayant des arguments contient un pointeur sur la pile des arguments qui contient en retour une série de pointeurs sur la pile des objets ; ces pointeurs sont les adresses des arguments.

[3] C-PROLOG, par exemple, construit des objets ayant par défaut 5 arguments inclus dans la structure et utilise des listes chainées d'extensions pour représenter les arguments supplémentaires.

6.4.3 les clauses

Une clause est un arbre d'objets de base tels qu'ils sont définis ci-dessus. De plus, en partage de structures, une descripteur de clause doit également contenir le nombre de variables de la clause pour permettre de réserver l'espace nécessaire dans la pile des environnements au moment de la sélection de clause et de l'unification.

6.4.4 Les environnements

Le mécanisme de partage de structures qui a été choisi impose l'utilisation d'environnement pour l'unification et la résolution, ainsi que d'un mécanisme de mémorisation pour défaire les liaisons incorrectes au moment du retour arrière. Ce mécanisme est généralement connu sous le nom *Trailing* en anglais, ou traîiınée en français. Nous avons déjà exposé le problème lorsque nous avons décrit le partage de structures (2.1.2.2). Chaque élément de trailing indique quel élément de la pile des environnements doit être remis à libre. En cas de backtrack, on parcourt la pile de traîiınée depuis le sommet jusqu'au niveau sauvegardé au moment où le point de choix a été créé et on remet à libre l'ensemble des environnements mémorisés. Pour plus de détails, voir l'annexe B.

6.4.5 La résolvante

Comme nous avons choisi la technique du partage de structures, la résolvante aura la structure classiquement utilisée dans ce cas. On représentera chaque élément de cette pile comme un couple `(Structure, Environnement)`. `Structure` est un pointeur (au sens large[4]) sur la structure de l'objet placé dans la résolvante, et `Environnement` l'adresse (au sens large encore) de la base de l'environnement dans lequel doit être évalué cette structure.

Suivant le mécanisme indiqué dans le chapitre , chaque étape de la résolution placera un couple `(structure, Environnement)` dans la résolvante, en suivant les instructions fournies par la règle de résolution sélectionnée. Cet objet sera soit un pointeur sur un objet du fait courant, évalué dans l'environnement de ce fait, soit un élément de la question, qui est déjà un couple `(Structure, Environnement)` puisque la question est isomorphe à la résolvante (elle est formée à partir de celle-ci par les règles de réécriture).

Nous sommes donc limités sur la forme des objets que nous pouvons placer dans la résolvante : nous retrouvons ici les limitations liées au partage de structures évoquées dans le chapitre 4.

6.4.6 La question

La question sera constituée à partir de la résolvante à la fin de chaque résolution élémentaire, par application des règles de réécriture. Elle aura la même structure que la résolvante.

6.4.7 Les points de choix

Un point de choix doit contenir l'ensemble des informations nécessaires à la reprise de la résolution en cas de retour arrière du moteur d'inférence. Il faut donc pouvoir restaurer la question, la résolvante, le fait, l'environnement du fait. Il faut également récupérer l'espace

[4] Nous indiquons "au sens large" pour bien souligner qu'aucun choix n'est encore fait sur la façon dont seront implantés les "pointeurs". Il pourra s'agir de pointeurs classiques, ou de toute autre méthode.

qui peut être occupé par les divers objets, environnements, etc...crée pendant la résolution et défaire les liaisons sur les variables qui ont été générées depuis le passage par ce nœud de l'arbre de résolution. Il faut enfin reprendre effectivement la résolution à l'étape suivante, c'est à dire sur la clause suivante si le point de choix était un point de choix pour une clause, et sur la règle suivante, si le point de choix était une règle.

Pour réaliser ces différentes tâches, un point de choix mémorisera l'ensemble des *registres* déterminant l'état de la machine à un instant donné, ainsi que la position dans l'arbre de résolution (règle de résolution et clause courantes).

6.4.8 La pile de trainée

La pile de trainée est directement associée au problème du backtrack. Elle sert à remettre à *libre* au moment du backtrack des variables dont la valeur de liaison n'est plus valide. Pour tous les détails concernant l'implantation du mécanisme de trainée, on peut se reporter a l'annexe B.

6.4.9 La tables des noms de prédicat

La table des noms de prédicat a un statut particulier ; en effet, il ne s'agit pas de dupliquer le même nom de prédicat chaque fois que l'on introduit dans la base une clause contenant ce prédicat. Les noms de prédicats ne seront donc pas empilés, mais introduits dans une table, avec un test d'occurrence pour éviter les duplications.

6.4.10 Les fonctions

Les fonctions sont divisées en deux catégories :

Les fonctions opérant sur les piles d'objets : ces fonctions devront permettre l'accès à chaque élément de la pile, la création, la destruction, le remplacement d'un objet. Il faudra considérer le cas particulier des prédicats : ceux ci contrôlent en général la sélection des clauses, et l'accès aux clauses ayant le même prédicat de tête doit être rapide. Ce type de contrainte n'existe pas pour, par exemple, des entiers ou des flottants.

Les fonctions opérant individuellement sur chaque objet : Celles-ci devront fournir les primitives de lecture et d'écriture de chacun des champs.

6.4.11 Les registres

Les registres de la machine sont les variables mémorisant l'ensemble des informations nécessaires pour définir l'état de la machine (avec, bien entendu, le contenu des piles). Les registres sont modifiés par les règles de résolution, par la sélection des clauses, par le moteur d'inférence.

Les registres doivent être mémorisés à chaque point de choix pour qu'il soit possible de reprendre la résolution en cas de retour arrière.

Les registres comprennent l'ensemble des adresses du sommet des piles, l'adresse de la base de la résolvante courante, l'adresse de l'élément courant de la question, l'adresse de l'élément courant du fait, l'adresse de l'environnement du fait courant, ainsi que la clause courante et la règle de résolution courante.

6.5 La machine de niveau 1

La machine de niveau 1 doit, en s'appuyant sur la machine de niveau 0, fournir à la machine de niveau 2 tous les éléments nécessaires pour effectuer correctement les opérations de sélection de règles, de sélection de clauses, et d'exécution des règles de résolution. Le déréférencement et l'unification sont deux opérations indispensables pour effectuer ces tâches.

La présence de la définition des prédicats pré-définis peut en revanche surprendre.

De façon générale, les prédicats pré-définis sont une "verrue" de tout système PROLOG. Un PROLOG pur n'a pas besoin de prédicats pré-définis. TARSKI n'en aurait donc pas, en principe, besoin.

Il est pourtant clair que l'on ne peut les supprimer. Une implantation sans prédicat prédéfini serait dans l'incapacité de faire de l'arithmétique, ou d'exécuter un *cut*.

Les placer dans l'architecture générale est relativement délicat. Pourtant, lors de la sélection de la clause qui va servir de base à la suite de la résolution, il est nécessaire de savoir si le prédicat de tête de la question courante est un prédicat standard ou un prédicat prédéfini : dans le premier cas, il n'y a pas de sélection de clause, et le code du prédicat prédéfini est simplement exécuté, dans le second cas, on doit effectivement sélectionner une clause. Il nous a donc semblé qu'il devait se trouver à un niveau inférieur à celui des opérations de type sélection de clause. Comme d'autre part, ils doivent utiliser les types de base et les registres, nous avons choisi de les placer à ce niveau conceptuel.

6.5.1 Déréférencement

Le déréférencement est une opération classique en partage de structures. Il s'agit, étant donné un couple `(structure, Environnement)` de déterminer sur quel objet nous mène la chaîiine des différents couples *(Structure, Environnement)*.

Nous reviendrons en détail sur ce point dans le chapitre consacré à l'implantation.

6.5.2 Unification

L'unification est un problème dont nous avons déjà largement parlé. Nous reviendrons sur les modalités pratiques de l'unification dans TARSKI dans le chapitre suivant.

Rappelons une fois de plus que TARSKI utilise la technique de partage de structures (nous avons déjà largement mentionné ce point) et le classique algorithme de Robinson pour l'unification et non un algorithme de résolution par contrainte.

Pourquoi ne pas avoir choisi un algorithme de résolution par contrainte? Principalement pour des raisons de simplicité de l'implantation. Même si nous disposons des sources de CLP(R), grâce à la gentillesse de Spiro Michaylov et Joxan Jaffar, il nous a semblé préférable de nous concentrer sur les problèmes des règles de résolution plutôt que de passer une partie de notre temps à implanter un de ces nouveaux algorithmes.

6.5.3 Prédicats pré-définis

Comme nous l'avons dit précédemment, les prédicats pré-définis sont une extension au PROLOG pur, qu'il est nécessaire d'implanter pour permettre l'utilisation de techniques de contrôle comme le `cut` PROLOG classique, ou pour pouvoir effectuer certaines opérations arithmétiques. Ces prédicats ne sont pas indispensables à une implantation PROLOG, mais ils sont extrêmement utiles.

Nous avons décidé :

1. Qu'il n'était pas indispensable, dans un premier temps, d'implanter beaucoup de prédicats pré-définis. TARSKI a pour but de valider des techniques de programmation sur des clauses de Horn généralisées et non d'être un système PROLOG complet.

2. Qu'il fallait pouvoir étendre et modifier aisément les prédicats pré-définis, sans qu'il soit nécessaire de reprendre ou de recompiler quelqu'autre partie que ce soit du programme.

Nous nous restreindrons à l'implantation des types de prédicats suivants :

les prédicats arithmétiques : nous n'implanterons que les prédicats arithmétiques de base (addition, soustraction,...et tests arithmétiques).

Les prédicats de tests : nous entendons par là les prédicats permettant de déterminer si un objet est un atome, une liste, un nombre, un prédicat,...

les prédicats de contrôle : nous n'implanterons que les prédicats de contrôle les plus classique : `cut, fail, success`

"Miscellaneous predicates" : nous implanterons également le prédicat `print`, pour l'impression de la valeur d'un objet.

Pour ce qui est de l'ajout et de la modification des prédicats pré-définis, nous prendrons soin de bien séparer leur implantation du reste du programme.

6.6 La machine de niveau 2

La machine de niveau 2 s'appuiera sur les primitives des machines de niveau inférieur, et implantera les opérations de sélection de clauses et de sélection de règles. Ces opérations devront être paramétrables dans TARSKI et leur modification devra pouvoir s'effectuer avec facilité.

6.6.1 Sélection de clauses

La sélection de la première clause est la première opération réalisée par le système TARSKI lorsqu'il commence à résoudre une nouvelle question. En cas de backtrack, le système devra alors sélectionner la clause suivante pouvant permettre de résoudre également la question. Au cas où aucune clause n'est sélectionnable, la fonction de sélection de clause doit échouer.

Ainsi soit le programme PROLOG classique suivant :

```
p:-q.
q.
p:-r.
p.
```

Supposons que la question soit `p`. En utilisant le mécanisme classique de sélection de clauses PROLOG, le système doit retourner comme première clause `p:-q`, puis en cas de backtrack `p:-r`, puis, s'il y a de nouveau backtrack, `p`. L'appel suivant à la fonction de sélection doit échouer.

Nous entendons conserver, pour des raisons de simplicité, le fonctionnement "en profondeur d'abord" de PROLOG. Cependant, nous souhaitons pouvoir modifier la fonction de sélection de clause. En effet, il peut parfois être souhaitable de réaliser la sélection sur d'autres critères ou sur des critères plus contraignants que ceux utilisés classiquement dans PROLOG.

C'est pour cette raison que nous faisons en sorte de bien séparer la fonction de sélection de clauses du moteur d'inférence lui-même. En effet, il doit être possible de modifier cette fonction sans avoir à modifier en quoi que ce soit le reste du programme.

6.6.1.1 Méthode de sélection classique

La méthode de sélection classique PROLOG est la suivante :

1. On sélectionne d'abord la première clause de la base dont la tête s'unifie avec le premier prédicat de la question.

2. On sélectionne ensuite toutes les clauses de la base dont la tête s'unifie avec le premier prédicat de la question, en les prenant dans l'ordre de leur apparition dans la base de clauses.

Cette technique est évidemment applicable à TARSKI . Elle correspond à un mécanisme standard et nous l'implanterons donc en priorité.

6.6.1.2 Autres méthodes de sélection

Nous allons citer rapidement deux autres méthodes de sélection qui seraient également utilisables :

Par la valeur de l'argument du premier opérateur modal : Dans le cas de la logique des modules, il est intéressant de sélectionner la clause non seulement sur le prédicat de tête, mais également sur la valeur de l'argument du premier opérateur modal de la clause.

"Least-used first" : la sélection "least-used first" ou la moins utilisée en premier, consiste à choisir en premier parmi toutes les clauses dont le prédicat de tête correspond bien au prédicat de tête de la question, non plus la première des clauses, mais celle qui a été le moins souvent utilisée durant la résolution.

Cette technique a l'inconvénient majeur de détruire le pseudo-déterminisme PROLOG, utilisé par tous les programmeurs, et de faire disparaîıtre l'intérêt du `cut`. En revanche, elle peut permettre dans certains cas de résoudre des problèmes pour lesquels la méthode de sélection PROLOG standard échoue, car elle provoque des boucles sans fin, l'exemple classique étant le programme :

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

Ici la méthode "least-used first" permet de résoudre le problème, alors que la méthode standard échoue.

6.6.1.3 Le cas des prédicats pré-définis

Les prédicats pré-définis constituent une exception au mécanisme général de sélection de clause. En effet, lorsque l'on rencontre un prédicat pré-défini, il ne faut pas sélectionner une clause, mais simplement exécuter le code associé au prédicat.

Nous avons choisi d'implanter le cas des prédicats pré-définis au niveau de la fonction de sélection de clause et non au niveau du moteur d'inférence. C'est un choix discutable, mais il permet de masquer au moteur d'inférences les objets peu élégants que sont les prédicats pré-définis.

6.6.2 Sélection et exécution de règles

Le chapitre 5 nous a montré que l'accès aux règles de résolution devait se faire par un mécanisme d'indexation, les opérateurs de la question et du fait déterminant directement la (les) règles susceptibles d'être sélectionnées.

L'exécution de ces règles aura pour objet de modifier la question, la résolvante et le fait courant, comme indiqué dans le chapitre 4.

Il est indispensable de séparer totalement la sélection et l'exécution des règles du reste du système. En effet, les règles sont liées à une logique particulière et sont appelées à changer lorsque l'on change de logique.

Les règles doivent pouvoir s'écrire simplement à l'aide des opérations fournis par les machines de niveau inférieurs. Ainsi, il sera aisé de construire de nouvelles implantations de nouvelles logiques.

6.6.2.1 Sélection des règles

La sélection de règles sera effectué suivant le mécanisme décrit au chapitre 5 :

1. Lors du premier essai de résolution, la fonction de sélection de règles doit retourner la première règle utilisable. Cette règle est fonction de l'opérateur courant de la question et de l'opérateur courant du fait.

2. En cas de backtrack, la fonction de sélection doit retourner la règle suivante permettant de tenter de poursuivre la résolution.

Au cas où il n'y a aucune règle sélectionnable, la fonction de sélection doit échouer.

Ainsi, supposons que notre logique est déterminée par le tableau 6.1.

Si l'opérateur courant du fait est $\diamond_I$ et l'opérateur courant de la question est $\diamond$, le premier appel de la fonction de sélection de règle retournera la règle :

$$\frac{\diamond_I(Y,I) : A, ?\diamond(X) : B \vdash ?C}{\diamond_I(Y,I) : A, ?B \vdash ?C}$$

Le second appel, en cas de backtrack, retournera la règle :

$$\frac{\diamond_I(X,I) : A, ?\diamond(X) : B \vdash ?\diamond_I(X,I) : C}{A, ?\diamond(X) : B \vdash ?C}$$

et le troisième appel, s'il y a de nouveau backtrack, échouera.

Le mécanisme d'implantation doit être transparent pour la couche supérieure du système, le moteur d'inférence.

Fait	*Question*	*Règles*
Pred	Pred	$p, ?p \vdash ?true$
Pred	$\wedge$	$\frac{A,?\wedge(X):B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
Pred	$\Diamond$	$\frac{A,?\Diamond(X):B\vdash ?C}{A,?B\vdash ?C}$
$\wedge$	Pred	$\frac{\wedge(X):A,?B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
$\wedge$	$\wedge$	$\frac{\wedge(X):A,?\wedge(Y):B\vdash ?\wedge(Y):C}{\wedge(X):A,?B\vdash ?C}$
$\wedge$	$\Diamond$	$\frac{\wedge(X):A,?\Diamond(Y):B\vdash ?\wedge(X):C}{A,?\Diamond(Y):B\vdash ?C}$
$\wedge$	$\Diamond_I$	$\frac{\wedge(X):A,?\Diamond_I(Y,I):B\vdash ?\wedge(X):C}{A,?\Diamond_I(Y,I):B\vdash ?C}$
$\Box$	Pred	$\frac{\Box(X):A,?B\vdash ?C}{A,?B\vdash ?C}$
$\Box$	$\wedge$	$\frac{\Box(Y):A,?\wedge(X):B\vdash ?\wedge(X):C}{\Box(Y):A,?B\vdash ?C}$
$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(Y):B\vdash ?C}{A,?\Diamond(Y):B\vdash ?C}$
		$\frac{\Box(Y):A,?\Diamond(X):B\vdash ?C}{\Box(Y):A,?B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{A,?\Diamond(X):B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{\Box(X):A,?B\vdash ?C}$
$\Box$	$\Diamond_I$	$\frac{\Box(X):A,?\Diamond_I(Y,I):B\vdash ?C}{A,?\Diamond_I(Y,I):B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond_I(X,I):B\vdash ?\Diamond_I(X,I):C}{\Box(X):A,?B\vdash ?C}$
$\Diamond_I$	$\wedge$	$\frac{\Diamond_I(Y,I):A,?\wedge(X):B\vdash ?\wedge(X):C}{\Diamond_I(Y,I):A,?B\vdash ?C}$
$\Diamond_I$	$\Diamond$	$\frac{\Diamond_I(Y,I):A,?\Diamond(X):B\vdash ?C}{\Diamond_I(Y,I):A,?B\vdash ?C}$
		$\frac{\Diamond_I(X,I):A,?\Diamond(X):B\vdash ?\Diamond_I(X,I):C}{A,?\Diamond(X):B\vdash ?C}$
$\Diamond_I$	$\Diamond_I$	$\frac{\Diamond_I(X,I):A,?\Diamond_I(X,I):B\vdash ?\Diamond_I(X,I):C}{A,?B\vdash ?C}$

Tableau 6.1: Tableau des règles possibles

6.6.2.2 Exécution de règles

Lorsqu'une règle a été sélectionnée, il faut l'exécuter. L'exécution d'une règle va modifier le fait courant, la question courante, et la résolvante. Elle peut en fait modifier l'ensemble des structures générales de la machine, car l'exécution d'une règle peut parfaitement créer de nouveaux objets, réaliser des unifications, etc...

L'exécution d'une règle peut se terminer de trois façons :

Un échec : le système ne peut pas exécuter la règle, dans ce cas il y a backtrack et recherche de la règle suivante.

Un succès simple : on passe à la suite de la résolution avec la nouvelle question, le nouveau fait et la nouvelle résolvante.

Une terminaison : La règle est une règle de terminaison. Dans ce cas, un succès indique que la résolution courante est terminée et qu'il faut reconstruire la question en utilisant les règles de réécriture. L'exemple classique de règle de terminaison est bien entendu :

$$p, ?p \vdash ?true$$

Considérons par exemple la règle :

$$\frac{\Diamond_I(X,I) : A, ?\Diamond(X) : B \vdash ?\Diamond_I(X,I) : C}{A, ?\Diamond(X) : B \vdash ?C}$$

Son exécution va entraîiner :

1. Une unification, celle du premier argument de l'opérateur du fait et de l'argument de l'opérateur de la question.
2. La modification du fait courant, qui perd $\Diamond_I(X,I)$.
3. La modification de la résolvante sur laquelle on empile une référence vers l'opérateur de la question.

Donc, l'exécution de règles devra donc être précédée de la création d'un point de choix et de la sauvegarde des registres de la machine de façon à pouvoir restaurer l'état originel en cas de backtrack.

Il est bon de souligner deux phénomènes :

1. Dans l'exemple choisi, et d'ailleurs pour toutes les règles de S1, l'exécution d'une règle s'adapte parfaitement au mécanisme de partage de structures choisi pour TARSKI . En effet, l'empilement sur la résolvante d'un nouvel opérateur se fait ici par l'empilement d'une référence vers l'opérateur de la question, et l'empilement d'une référence vers l'environnement associé. Si nous avions adopté la technique de *goal-stacking*, ou empilement de buts, nous aurions dû ajouter dans la résolvante non pas une référence vers l'objet, mais bien une copie de l'objet lui-même.
2. Il est cependant parfois nécessaire de construire un objet durant la résolution, et, à priori, la technique de partage de structures ne permet pas de le faire. Il est cependant possible de tourner la difficulté dans certains cas. Considérons par exemple la règle suivante :

$$\frac{\Diamond_I(X,I) : A, ?\Diamond(X) : B \vdash ?\Box(X) : C}{A, ?\Diamond(X) : B \vdash ?C}$$

Il est ici parfaitement impossible d'empiler dans la résolvante une référence vers un objet de la forme $\Box(X)$ puisqu'il n'existe aucun objet de ce type, ni dans la question, ni dans le fait courant. Nous devons alors construire une nouvel objet $\Box(X)$ et réserver un nouvel environnement pour notre variable X, que nous prendrons soin d'unifier avec la variable X présente dans le fait et dans la question. Nous pouvons alors empiler dans la résolvante la référence sur ce nouvel objet et ce nouvel environnement.

Dans quel cas cette méthode ne permet-elle pas de pallier complètement aux déficiences du partage de structures? Par exemple, quand l'objet à reconstruire est le fait. Considérons maintenant la règle :

$$\frac{\Diamond_I(X, I) : A, ?\Diamond(X) : B \vdash ?\Diamond(X) : C}{\Box(Y) : A, ?\Diamond(X) : B \vdash ?C}$$

Ici, nous devrions reconstruire un nouveau fait ; mais ce nouveau fait devrait contenir un nouvel objet **contenant une nouvelle variable qui devra être évaluée dans un nouvel environnement que celui du fait**, puisque l'on n'a pas initialement prévu, lors de la réservation de l'environnement, cette nouvelle variable. Or, en partage de structures, l'environnement du fait est **unique**, et il faudrait profondément remanier le mécanisme pour pouvoir traiter la règle ci-dessus.

On comprend ainsi mieux les limitations placées sur la forme des règles dans le chapitre 4. Elles sont liées aux limitations exposées ci-dessus, limitations dues au mécanisme de partage de structures.

6.6.2.3 Les règles de réécriture

Les règles de réécriture permettent de construire une nouvelle question à partir de la résolvante quand l'on a atteint une règle de terminaison.

Les règles de réécriture sont déterministes. Il est impossible de backtracker sur une règle de réécriture. Le mécanisme de réécriture doit apparaîitre comme une boîite noire au moteur d'inférence.

Il aurait été possible de séparer les règles de réécriture des règles de résolution standard, dans la mesure où la sémantique des deux systèmes de règles est complètement différente. Cependant, il nous a semblé que, pour une personne implantant une nouvelle logique, il valait mieux que l'ensemble des objets dépendant de la logique soit regroupé dans la même entité.

6.7 Le moteur d'inférence

Le moteur d'inférence est la dernière couche du système TARSKI . Il s'appuie sur les machines de niveau inférieur pour implanter l'ensemble des processus de contrôle nécessaire au fonctionnement du système.

6.7.1 Fonctionnement général

Nous rappelons dans la figure 6.2 le schéma général de fonctionnement du moteur d'inférence.

Il est possible de concevoir le moteur de plusieurs façons :

1. On peut conserver telle quelle la structure de l'organigramme avec ses branchements, ce qui aboutira inévitablement à un codage peu élégant.

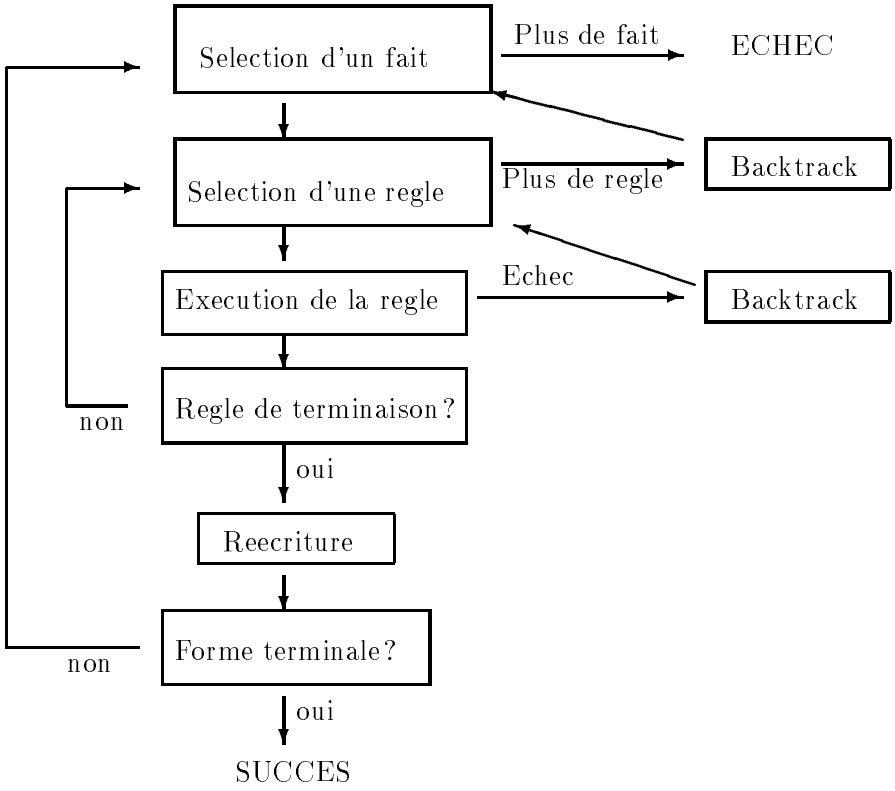


Figure 6.2: Conception du moteur d'inférence

	Etat courant			
Etat suivant	SC1	SR1	ER	BT
SC1			Terminaison	
SR1	Clause trouvée		Succès	Clause trouvée
ER		Règle trouvée		Règle trouvée
BT	Pas de clause	Pas de règle	Echec	Pas de clause/règle

Tableau 6.2: Matrice de changement d'états du moteur

2. On peut utiliser les techniques de programmation structurée.

3. On peut modéliser le système comme une machine à états.

La première implantation du moteur d'inférences dans une version précédente du système a été faite suivant la méthode (2). Cependant, la structure même du moteur s'adaptait assez mal à la programmation structurée. L'imbrication des différents niveaux rendait le code difficile à lire, et le passage d'un niveau à l'autre finissait pas disparaîitre sous l'accumulation des tests et des boucles.

Pour des raisons d'efficacité, mais aussi de lisibilité du code, nous avons donc choisi de considérer ce mécanisme comme celui d'une machine à états.

Nous avons retenu quatre états :

SC1 : SC1 correspond à la sélection de la première clause pour une question donnée.

SR1 : SR1 correspond à la sélection de la première règle pour un fait donné et une question donnée.

ER : ER correspond à l'exécution d'une règle. Cette étape inclut automatiquement la réécriture si la règle exécutée avec succès est une règle de terminaison.

BT : BT correspond à l'étape de Backtrack. L'opération de backtrack est la plus complexe, car elle dépend du point de choix courant :

Backtrack sur une clause : Si on backtracke sur une clause, le moteur devra, durant l'étape BT, trouver la clause suivante utilisable en appelant la primitive de niveau 2 remplissant ce service.

Backtrack sur une règle : Si on backtracke sur une règle, le moteur devra trouver la règle suivante utilisable en appelant la primitive de niveau 2 remplissant ce service.

Le tableau 6.2 décrit la matrice de changement d'état du moteur d'inférence. Nous allons le détailler.

Colonne SC1 : Lorsque le système sélectionne une clause pour résoudre une question donnée pour la première fois, deux cas sont possibles :

La clause est trouvée : dans ce cas, le moteur doit maintenant sélectionner une première règle pour continuer la résolution donc il passe dans l'état SR1.

La clause n'est pas trouvée : il y a échec et le moteur doit effectuer un retour-arrière, donc on passe dans l'état BT.

Colonne SR1 : Lorsque le moteur sélectionne une règle pour la première fois, deux cas peuvent également se produire :

La règle est trouvée : dans ce cas il faut exécuter la règle, et on passe dans l'état ER.

Il n'y a pas de règle : il faut effectuer un retour arrière, et l'on passe donc dans l'état BT.

Colonne ER : Lors de l'exécution d'une règle, trois cas peuvent se produire :

Succès et terminaison : La règle s'exécute avec succès et est une règle de terminaison. Dans ce cas, après réécriture de la question, il faudra de nouveau sélectionner une nouvelle clause. Donc on passe dans l'état SC1.

Succès simple : il faut continuer la résolution en sélectionnant une règle pour continuer à réduire le fait courant et la question courante. On passe donc dans l'état SR1.

Echec : il faut dans ce cas effectuer un retour arrière, on passe donc dans l'état BT.

Colonne BT : Il faut distinguer deux cas bien différents dans le cas d'un retour arrière :

Retour arrière sur clause : le retour arrière a lieu sur un point de choix de clause. Le moteur sélectionne alors la clause suivante. Si elle existe on passe dans l'état SR1, puisqu'il faut alors sélectionner la première règle permettant de résoudre le nouveau fait courant avec la question courante. Si elle n'existe pas, on reste dans l'état BT.

Retour arrière sur règle : le retour arrière a lieu sur un point de choix de règle. Le moteur doit alors sélectionner la règle suivante. Si elle existe, il faut l'exécuter, et on passe alors dans l'état ER. Si elle n'existe pas, on reste dans l'état BT.

Lors du début de fonctionnement du système, le moteur commence son cycle dans l'état SC1, puisqu'il s'agit de sélectionner une première clause pour résoudre la question initiale.

Le mécanisme décrit ci-dessus laisse dans l'ombre deux cas sur lesquels nous devons revenir, la terminaison du programme par un succès et la terminaison du programme par un échec total.

6.7.2 Succès

Le programme se termine par un succès quand, après avoir réécrit la question à partir de la résolvante, la nouvelle question est vide. Dans ce cas, deux possibilités peuvent se présenter, suivant les types classiques de PROLOG

- Dans les PROLOG de type PROLOG-II, la résolution se poursuit jusqu'à l'obtention de toutes les solutions.
- Dans les PROLOG de type PROLOG-C, la résolution s'arrête dès qu'une solution est trouvée.

TARSKI accepte les deux types de résolution, que l'utilisateur choisit à l'aide d'une méta-commande avant l'exécution de son programme.

La détection du succès a lieu dans l'état ER, puisque c'est dans cet état que l'on réécrit la question à partir de la résolvante, après avoir exécuté avec succès une règle de terminaison. Lorsque l'on reconstruit une question vide, on va suivant les cas :

- Passer dans l'état BT si l'on désire obtenir les autres solutions.
- Sortir du cycle du moteur d'inférence si l'on ne désire pas obtenir les autres solutions.

6.7.3 Echec total

La détection de l'échec se fait dans l'état BT, puisque c'est lorsque nous ne pouvons plus backtracker que la résolution est terminée. Dans ce cas, il suffit de quitter le cycle du moteur d'inférence.

6.7.4 Justification de certains choix

On aurait pu choisir d'autres états, ou surtout considérer plus d'états pour le moteur d'inférence.

On aurait pu en particulier adopter les états supplémentaires :

RE (Réécriture) : Cet état aurait suivi l'état ER quand la dernière règle utilisée est une règle de terminaison. En cas de succès l'état suivant aurait été SC1, en cas d'échec, BT, en cas de succès total, ST.

SCS (Sélection_clause_suivante) : L'état SCS aurait suivi l'état BT en cas de retour-arrière sur une clause. En cas de succès, l'état suivant aurait été SR1, en cas d'échec BT.

SRS (Sélection_règle_suivante) : L'état SRS aurait suivi l'état BT en cas de retour-arrière sur une règle. En cas de succès sur une règle de terminaison, on serait passé dans l'état RE, en cas de succès sur une règle classique, on serait passé dans l'état SR1, en cas d'échec dans l'état BT.

ST (Succès_total) : ST aurait suivi RE ; il aurait précédé BT dans le cas ou l'on souhaite obtenir toutes les solutions.

ET (Echec_total) : Etat terminal du système, qui suivrait BT au cas où le backtrack n'est plus possible.

On peut ainsi mieux séparer les différentes opérations de base du moteur d'inférence. C'est d'ailleurs de cette façon qu'un des moteurs d'inférence a été codé.

Cependant, cela diminue l'efficacité du moteur de façon relativement importante. Si, par exemple, nous sommes dans l'état BT sur un point de choix de clause. Nous allons d'abord passer dans l'état SCS, puis en cas d'échec dans l'état BT à nouveau. Nous exécuterons deux cycles au lieu d'un. Ce problème apparaîit également pour les retour-arrière sur les règles, ou sur la réécriture. Nous avons donc essayé de faire un compromis entre efficacité et séparation des états théoriques. Comme tout compromis il est discutable.

6.8 Le jeu d'instruction de la machine TARSKI

La compilation des règles de résolution dans le système TARSKI utilise comme langage intermédiaire un langage de haut niveau : le langage d'implantation du système : ADA.

Les instructions de la machine abstraite sont constituées de l'ensemble du langage ADA lui-même, plus un certain nombre de macro instructions fournies par les machines abstraites définies ci-dessus.

Le jeu de macro-instructions additionnelles est en fait très limité : on adjoint au langage les opérations sur les piles et les opérations d'unification et de déréférencement des variables. Ces opérations sont décrites en détail dans le chapitre suivant.

Chapitre 7

Conception du parallélisme

7.1 Modèle de parallélisme

7.1.1 Principes généraux

Nous avons choisi d'adopter comme modèle général de parallélisme le modèle Kabu-Wake décrit dans la section 2.2.1.

Rappelons rapidement que le système Kabu-Wake est composé de machines PROLOG standard. Le parallélisme est implanté au niveau des points de choix (il s'agit d'un parallélisme OU). Chaque fois qu'un processeur crée un point de choix, il continue la résolution normalement, mais si un processeur se libère, le processeur actif lui envoie une copie des piles au moment de la création du point de choix, ce qui permet au processeur libre de continuer la résolution.

Nous n'appliquerons pas le modèle Kabu-Wake tel quel. Nous en retiendrons les éléments suivants :

- Parallélisme implanté au niveau des points de choix
- Machines "standard" pour exécuter chaque fraction de l'arbre de résolution.
- Transmission de l'état de la résolution par transmission des piles et registres.

En revanche nous allons nous en distinguer sur les points suivants :

- Implantation réalisée au niveau des points de choix sur les règles et non pas sur les clauses.
- La transmission des piles se fera non pas à la demande du processeur libre, mais lors de la création du point de choix par le processeur actif, s'il existe (au moins un) processeur libre.
- Géométrie "variable"du réseau. On ne définit pas à priori la configuration et le mode de synchronisation des processeurs entre eux. On se réserve la possibilité de tester plusieurs configurations.

	Fait	Question	Règles
R1	$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(X):B\vdash ?C}{A,?\Diamond(X):B\vdash ?C}$
R2	$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(X):B\vdash ?C}{\Box(X):A,?B\vdash ?C}$
R3	$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{A,?\Diamond(X):B\vdash ?C}$
R4	$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{\Box(X):A,?B\vdash ?C}$

Tableau 7.1: Règles $\Box - \Diamond$

7.1.2 La machine parallèle élémentaire

Le principe général est simple : le choix de paralléliser la résolution pour un processeur élémentaire se fait au moment du choix de la règle de résolution. Lorsqu'il aboutit sur un point de choix de règle et qu'il y a plusieurs règles sélectionnables il envoie à un ou plusieurs processeurs libres l'ensemble des piles et des registres correspondant à l'état de la résolution courante, en leur faisant poursuivre la résolution sur la (ou les) branche(s) suivant la branche courante. Il conserve pour lui la branche courante. Deux solutions sont donc possibles dans ce cadre : Ainsi, supposons que les règles de résolution soient les règles décrites dans la table 5.3 que nous reproduisons ici dans la table 7.1. Suivant les modèles choisis, on peut rencontrer deux alternatives :

Distribution vers plusieurs processeurs : on se trouve dans le cadre du schéma de la figure 7.1. Le processeur P1 va envoyer l'ensemble des piles à chacun des processeurs disponibles et leur faire poursuivre la résolution à partir de ce point, en leur interdisant bien entendu tout retour arrière à ce niveau.

Distribution vers un seul processeur : On est la dans le cadre du schéma de la figure 7.2. le processeur actif P1 conserve pour lui la branche principale de la résolution et passe le reste de la résolution au processeur libre P2. Celui-ci peut à son tour choisir de paralléliser la suite de la résolution vers un autre processeur libre s'il s'en trouve un (P3).

7.1.3 Modifications par rapport à la machine classique

Par rapport à la machine classique, la machine parallèle ne subit presque aucune modification. Les choix effectués permettent de conserver la conception générale intacte.

Cependant certains ajouts sont nécessaires :

- Il faut écrire un paquetage de communications inter-processus.
- Les piles sont initialisées par le processeur appelant avant le début de la résolution et non par le parser.
- La machine parallèle élémentaire commencera la résolution avec le moteur d'inférences dans l'état exécution de règle (ER), au lieu de sélection de clauses (SC), comme nous

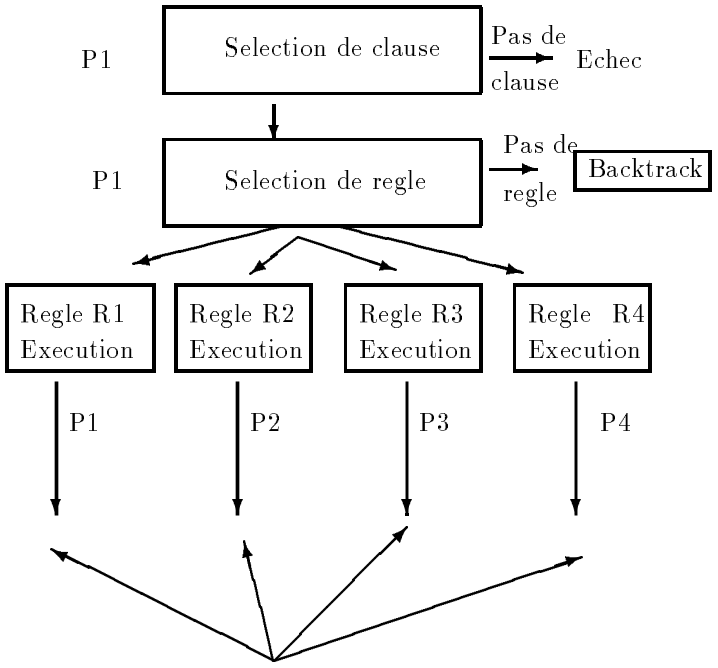


Figure 7.1: Distribution vers plusieurs processeurs

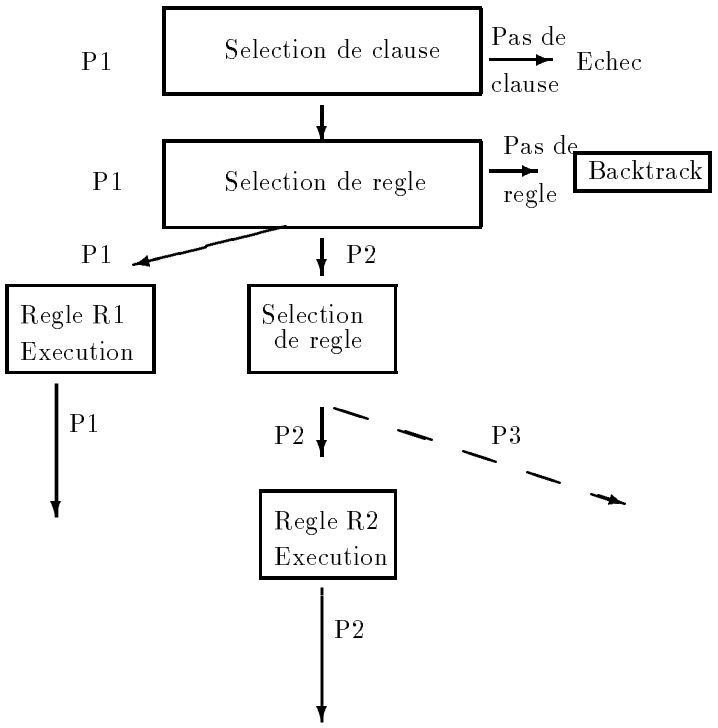


Figure 7.2: Distribution vers un seul processeur

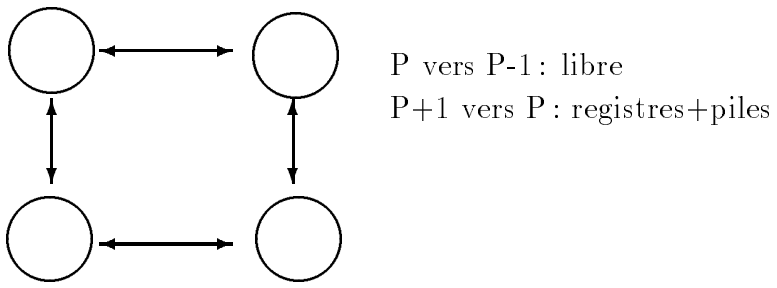


Figure 7.3: Réseau en anneau

l'avions dit dans le chapitre 6, puisqu'elle commence la résolution sur un point de choix de règles.

A l'exception de ces retouches finalement mineures, aucun changement n'intervient dans la structure du système.

7.2 Différentes géométries de réseau

Le mécanisme général de parallélisme étant choisi, reste maintenant à adopter la géométrie du réseau, c'est à dire la façon dont un processeur va distribuer la résolution vers un ou plusieurs autres processeurs[1]. Quatre solutions ont été envisagées, que nous allons maintenant décrire.

7.2.1 Réseau en anneau

Dans ce premier mécanisme, un processeur P ne peut communiquer qu'avec les processeurs $P+1$ et $P-1$. Lorsqu'il est libre, il envoie au processeur précédent un message pour le lui indiquer. Quand il veut distribuer du travail au processeur suivant il sait qu'il peut le lui envoyer immédiatement, s'il a reçu un message lui indiquant que celui-ci est libre (figure 7.3).

L'avantage de cette méthode est qu'un processeur actif n'attend jamais au moment où il envoie une tâche, puisqu'il sait instantanément si le processeur suivant est libre. L'inconvénient est qu'un processeur libre peut rester inactif plus longtemps si son prédecesseur n'a pas de travail à lui fournir.

Le cas le plus défavorable se produit non pas quand un seul processeur est actif et ne distribue pas de travail (car alors, de toute façon il n'y a pas de règles parallélisables), mais quand un processeur est actif, qu'il ne distribue pas de travail et que le processeur précédent est également actif et qu'il aurait du travail parallélisable. Alors, ce processeur ne peut le distribuer car il est bloqué par le processeur suivant, alors que les autres processeurs sont inactifs.

[1] En parlant de "réseau de communication", nous ne faisons aucune supposition sur la façon dont seront effectivement implantées la communication inter-processus. Le terme est générique.

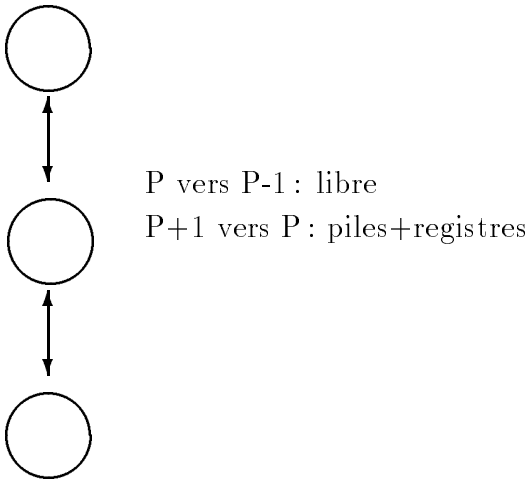


Figure 7.4: Réseau "du haut vers le bas"

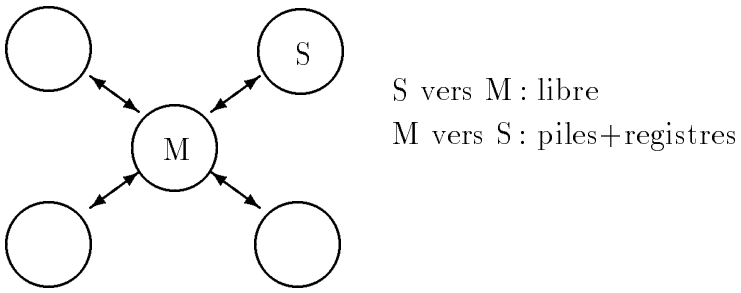


Figure 7.5: Réseau en étoile

7.2.2 Réseau "du haut vers le bas"

Le réseau "du haut vers le bas" est un réseau proche du réseau en anneau, mais où le dernier processeur ne peut envoyer de données au premier processeur (voir figure 7.4). Ce type de réseau présente de grandes qualités pratiques pour l'implantation, et c'est pour cette raison que nous le mentionnons ici. Il présente les mêmes défauts que le réseau en anneau, avec, en plus, le risque de voir la résolution "buter" sur le dernier processeur qui peut alors devenir le seul processeur actif.

7.2.3 Réseau en étoile

Dans cette structure (figure 7.5, le processeur maîiitre est le seul à pouvoir distribuer la résolution.

Ce type de structure est extrêmement délicat à mettre en œuvre. Supposons que le processeur maîiitre ait distribué du travail à tous ses processeurs esclaves ; si par malheur il termine sa résolution courante avant eux, il sera dans l'incapacité de distribuer du travail aux processeurs inactifs, et bien entendu incapable d'effectuer aucun travail.

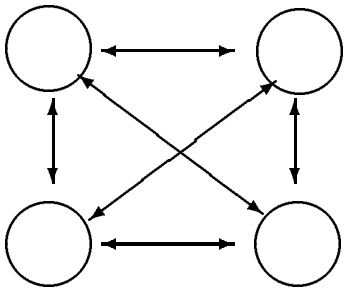


Figure 7.6: Réseau totalement interconnecté

7.2.4 Réseau totalement interconnecté

Le protocole adopté est le suivant : dès qu'un processeur L est libre, il envoie à tous les autres processeurs un message indiquant qu'il est prêt. Dès qu'un processeur P a du travail à lui donner, il lui envoie une requête d'acceptation. Le processeur libre L va alors accepter cette requête et diffuser un message à tous les processeurs pour indiquer qu'il va accepter le travail de P. Ce message permet également d'indiquer à un processeur concurrent qui avait simultanément contacté L qu'il a été évincé. Le processeur P va alors envoyer le travail à L (figure 7.6).

Les avantages de la méthode sont :

- Un processeur ayant du travail à donner sait en permanence s'il y a des processeurs libres, et lesquels. Il peut donc rapidement savoir à qui redistribuer le travail.
- Le taux d'occupation des processeurs est proche de l'optimum.

L'inconvénient majeur est le délai d'attente du processeur P lorsqu'il décide de distribuer du travail. Il doit, pour pouvoir continuer à travailler, attendre un acquittement positif ou négatif de L. D'autre part, le réseau peut être encombré par la suite des messages de contrôle.

7.2.5 Conclusion

Il est raisonnable d'examiner trois architectures de réseau :

Le réseau "du haut vers le bas" : ce type de réseau est intéressant pour sa facilité d'implantation. Ce sera le premier type de réseau que nous implanterons.

Le réseau en anneau : en plus des avantages cités ci-dessus, le réseau en anneau présente la qualité de ne pas "effondrer" le médium de transmission, car il diffuse un minimum de messages.

L'interconnexion totale : il s'agit de la méthode qui présente à priori le plus grand attrait. Cependant, dans la mesure où elle optimisera l'utilisation des processeurs et nécessitera de nombreux messages de contrôle, elle risque de provoquer un effondrement du médium de transfert, surtout si le nombre de processeurs est important et le débit du médium de transfert faible.

Partie IV

Implantation

Chapitre 8

Implantation de la machine séquentielle

8.1 Choix effectués

8.1.1 Choix du langage d'implantation

Choisir un langage d'implantation est toujours un exercice difficile. Dans notre cas, le choix se limitait à deux possibilités : C ou ADA.

C pouvait se justifier pour un grand nombre de raisons :

- La grande diffusion du langage, et donc une plus grande facilité à distribuer le produit terminé aux gens qui souhaiteraient l'utiliser.
- Le nombre élevé d'utilitaires de développement disponibles (LEX, YACC, debuggers, compilateurs produisant du code efficaces et fiables).
- La facilité d'interfaçage avec le système d'exploitation.

La première version de MOLOG développée ([AG88]) fut d'ailleurs écrite en C. Cependant, nous avons retiré de cette expérience un certain nombre d'enseignements :

- L'absence de typage fort rend le debugging difficile, et encourage une programmation approximative.
- Le langage ne possède aucune capacité multi-tâches, or nous souhaitions réaliser un système parallèle.
- Il est impossible de faire de façon esthétiquement satisfaisante certaines implantations : la généricité n'existe pas, les records à discriminant sont implantés sous une forme approximative (union).

Le langage ADA se situe presque à l'opposé de C, les avantages de l'un étant bien souvent les inconvénients de l'autres :

- ADA a un typage fort. La généricité, la notion de record à discriminants sont implantés de façon simple et agréable. Tout cela impose une programmation plus propre, et rend le debugging presque inutile.

- Le langage a des possibilités multi-tâches.

A l'opposé :

- Le langage est peu diffusé dans le monde universitaire[1], ce qui rendra la diffusion du produit plus difficile.
- Les compilateurs sont lents, souvent encore buggés. Les debuggeurs eux-mêmes sont peu fiables.
- Certains outils comme LEX et YACC n'existaient pas encore pour ADA au moment où ce travail a débuté.

Effectuer le choix dans ces conditions était un exercice bien difficile. Les raisons qui ont fait pencher le choix en faveur d'ADA sont certainement :

- Le mauvais souvenir gardé du debugging de la version C de MOLOG.
- L'environnement de travail général. L'ENAC et le CENA ayant adopté ADA, il était plus facile de trouver un "support" sur ce langage que sur C.

Pour ce qui est de l'implantation de la machine séquentielle, nous n'avons absolument pas regretté ce choix. L'écriture du programme fut facile et le debugging a peu près inexistant. Nous avons presque pu à cette occasion vérifier l'adage ADA : "un programme que l'on parvient à compiler est un programme qui marche". Le portage de la machine séquentielle sur d'autres systèmes (VAX/VMS, HP-720/HP-UX, SPARC-II/SUN-OS, SONY) et d'autres compilateurs (DECADA, ALSYS, VERDIX) ne posa absolument aucun problème : il a suffi de recompiler les sources.

Mais il faut aussi reconnaîiitre que la machine séquentielle est un programme "full ADA", sans aucun interfaçage avec le monde extérieur. Nous verrons dans le chapitre 9 que le choix d'ADA a posé bien des problèmes pour l'implantation du parallélisme, car il a fallu créer des modules d'interface avec le système d'exploitation.

8.1.2 Efficacité versus lisibilité

Lorsque l'on écrit un programme, il faut toujours faire un choix, ou trouver un compromis, entre l'efficacité du code écrit et sa lisibilité.

Dans notre cas, nous avons résolument opté pour un code lisible, au détriment de son efficacité. Nous pensons qu'il sera toujours temps d'optimiser le code par la suite, au vu des résultats des tests qui auront été faits, quitte même à changer de langage d'implantation.

Nous voyons avant tout ce travail comme un travail de validation de deux idées :

- La possibilité de développer un système souple et pourtant compilé, permettant d'implanter n'importe quel type de logique[2]
- La possibilité d'utiliser le parallélisme intrinsèque à la résolution avec des clauses de Horn généralisées et des règles d'inférence paramétriques, de façon efficace par rapport à une implantation séquentielle.

[1] On peut espérer un changement avec l'apparition d'ADAeD de l'Université de New-York, un interpréteur ADA placé dans le domaine public.

[2] Vérifiant certaines conditions tout de même...

Notre but est donc d'arriver à un système parfaitement testé, et de préférences aussi fiable que possible.

Dans ce cadre, nous tenterons d'écrire un code aussi clair et aussi fiable que possible, au détriment des performances.

8.2 Architecture

La figure 8.1 montre quels sont les différents paquetages et définit les relations d'héritage. Nous allons rapidement expliquer à quoi correspondent chacune des différentes unités de la figure 8.1.

Genpiles : Le paquetage **genpiles** est un paquetage générique définissant le type générique de piles.

Genhash : La paquetage **genhash** définit des piles génériques spécifiques évitant la duplication d'un même élément dans la pile.

Types : Le paquetage **types** implante les types de base, les piles et les opérations élémentaires associées

Registres : Le paquetage **registres** définit les registres de la machine abstraite.

Unification : Le paquetage **unification** implante l'unification et les opérations associées (déréférencement).

Opérateurs : Le paquetage **opérateurs** implante les opérations de sélection de règles et d'exécutions des règles.

Selclause : Le paquetage **selclause** implante les opérations permettant la sélection des clauses pendant la résolution.

Predef : Le paquetage **predef** implante les prédicats prédéfinis.

Moteur : Le paquetage moteur implante le moteur d'inférences lui-même.

Nous allons dans la suite de ce chapitre décrire en détail chacun de ces paquetages.

8.3 Piles génériques étendues

8.3.1 But

Les piles génériques ont pour but d'implanter l'ensemble des opérations nécessaires au fonctionnement des piles d'objets, piles d'opérandes, pile d'environnement, etc...Plutôt que de parler de piles, il vaut mieux parler de piles étendues, car le jeu d'opérations est plus large que celui du type abstrait PILE traditionnel.

Nous laissons de côté les détails de l'implantation et des paramètres de généricité : on peut le trouver dans l'annexe A. Nous nous contentons de donner les entités exportées par ce paquetage.

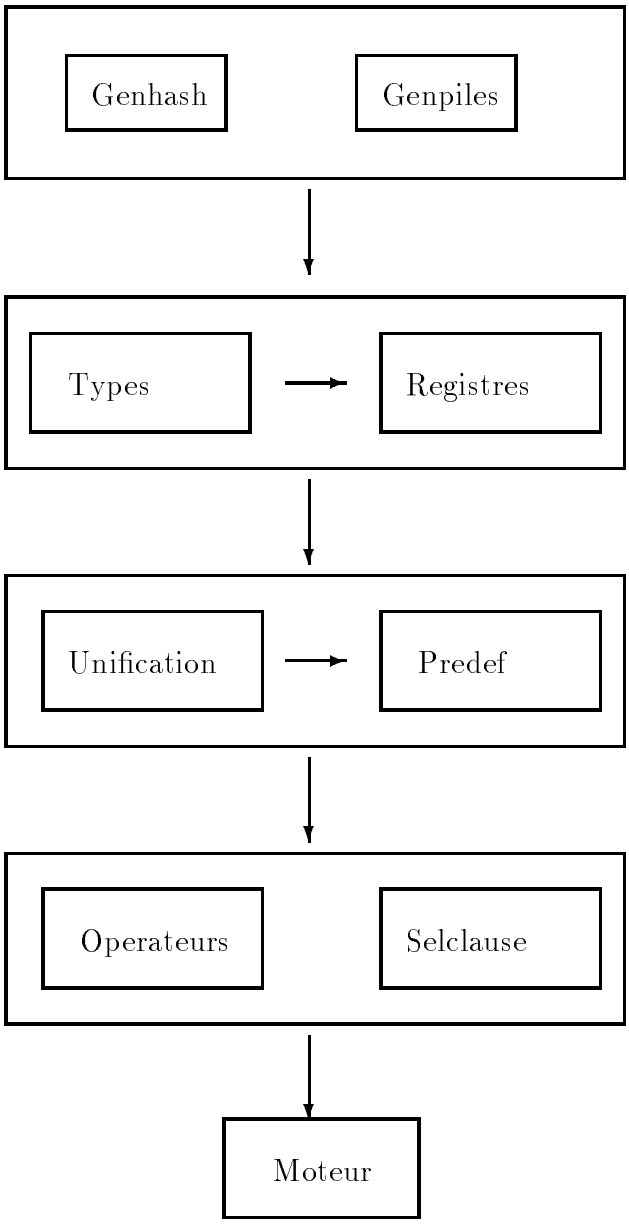


Figure 8.1: Architecture détaillée des paquetages

8.3.2 Entités exportées

8.3.2.1 Fonctions et procédures exportées

`FUNCTION empiler (x : IN objet) RETURN indice;` La fonction `empiler` prend en paramètre un objet élémentaire, le place sur le sommet de la pile et retourne son adresse.

FUNCTION reserver (n : IN positive) RETURN indice; Cette fonction prend en argument un nombre positif n. Elle réserve n cases sur le sommet de la pile et retourne l'adresse de la première de ces cases.

`FUNCTION recuperer (i : IN indice) RETURN objet;` La fonction `recuperer` prend une adresse d'objet en argument et retourne cet objet.

`FUNCTION depiler RETURN objet;` La fonction `depiler` retourne l'objet placé sur le sommet de la pile, et le retire de la pile.

`PROCEDURE modifier (x : IN objet,i : IN indice);` La procédure `modifier` prend en arguments l'adresse i d'un objet et un nouvel objet$x$. Elle remplace l'objet à l'adresse i par l'objet x.

`PROCEDURE revenir (i : IN indice);` La procédure `revenir` prend en argument une adresse i et place le pointeur du sommet de pile à cette adresse.

`FUNCTION position RETURN indice;` la fonction `position` retourne l'adresse du sommet de la pile.

8.3.2.2 Constantes exportées

La seule constante exportée est `element_null`. Elle correspond à l'adresse de base de la pile qui n'est jamais utilisée (ni utilisable). Elle correspond au pointeur nul généralement utilisé dans d'autres implantations ou d'autres langages.

8.3.2.3 Exceptions exportées

Deux exceptions sont exportées :

`indice_trop_grand : EXCEPTION;` Exception levée quand l'utilisateur tente d'accéder à un objet se situant au delà du sommet de pile.

`pile_vide : EXCEPTION;` Exception levée quand l'utilisateur tente de dépiler un objet alors que la pile est déjà vide.

8.4 Les tables génériques

Le paquetage **genhash** implante un type générique de table. Contrairement aux piles, l'ajout d'un élément ne peut jamais entraîiner de duplication. La fonction qui ajoutera un élément dans la pile retournera l'adresse de cet élément, mais ne l'empilera pas sur le dessus de la pile s'il existe déjà.

Nous nous contentons également de donner les entités exportées, les détails de l'implantation se trouvent dans l'annexe A.

8.4.1 Entités exportées

8.4.1.1 Fonctions et procédures

Certaines des fonctions fournies par le paquetage sont les mêmes que celles fournies par le paquetage standard de pile ; certaines en revanche sont spécifiques.

`FUNCTION recuperer (i : IN indice) RETURN objet;` La fonction `recuperer` prend une adresse d'objet en argument et retourne cet objet.

`PROCEDURE revenir (i : IN indice);` La procédure `revenir` prend en argument une adresse i et place le pointeur du sommet de pile à cette adresse.

`FUNCTION position RETURN indice;` la fonction `position` retourne l'adresse du sommet de la pile.

`FUNCTION ajouter (x : IN objet) RETURN indice;` La fonction `ajouter` ajoute un élément dans la pile en s'assurant qu'il n'existe pas déjà dans la pile un élément identique.

8.4.1.2 Constantes exportées

Une seule constante exportée :`element_null`, comme pour les piles génériques. Les mêmes remarques s'appliquent.

8.4.1.3 Exceptions exportées

Une seule exception exportée : `indice_trop_grand : EXCEPTION;`. Cette exception est levée quand l'utilisateur tente d'accéder à un objet se situant au delà du sommet de pile.

L'exception `pile_vide` n'a pas de raison d'être puisque l'utilisateur ne peut pas dépiler d'objets.

8.5 Les types de base

Le paquetage **types** doit exporter l'ensemble des types et objets de base.

L'implantation idéale aurait certainement constitué dans des types de base tous limités, dont l'accès aux composants aurait été fait à l'aide de fonction (une fonction par composant de la structure). Il fallait cependant faire un choix raisonnable : pour des raisons d'efficacité, et dans la mesure où ce choix ne nous a pas semblé nuire de façon sérieuse à la lisibilité[3], les types de base ont été implantés dans la partie publique du paquetage type. Ainsi, l'accès à chaque élément de structure est plus rapide.

La plupart des objets manipulés par le système sont stockés dans des piles. On associe donc à chaque type un type pointeur que l'on note `num_nom_de_l'objet`. Ces types pointeurs sont des types limités privés qui sont déclarés dans la partie privée du paquetage et dont seule la déclaration incomplète est visible dans la partie publique.

L'avantage de cacher l'implantation des types pointeurs est qu'il est possible de redéfinir complètement cette implantation sans avoir à changer le reste du programme.

Chaque type, et chaque type pointeur associé étant défini, on instancie le paquetage de pile générique qui permettra de stocker l'ensemble des objets au cours de la résolution.

Nous allons revenir plus en détail sur tout ceci.

[3] Il peut en revanche poser des problèmes de maintenabilité.

8.5.1 Entités exportés

8.5.1.1 Types exportés

Nous allons décrire l'implantation des objets en reprenant l'ordre que nous avions choisi dans le chapitre 6.

Les objets de base: Les objets de base sont l'ensemble des objets pouvant apparaîitre dans une clause. Le type objet de base est implanté comme un record à discriminant. Le discriminant est un type énumératif nommé `genre_objet` et peut prendre les valeurs `opérateur, predicat, variable, allfree, entier, flottant, cons`. Le type `elt_objet` est alors donné par la structure suivante :

```
TYPE elt_objet (genre : genre_objet := operateur) IS
  RECORD
--Chaque objet retient le numero de clause ou il apparait
     clause : num_clause;
     CASE genre IS
       -- Un operateur est decrit par son numero
       -- le nombre de ses arguments, l´adresse du premier des arguments
       -- dans la pile des operandes ainsi que l´adresse de l´objet
       -- qu´il qualifie.
        WHEN operateur =>
           nom_op    : numero_operateur;
           nb_arg_op : nombre_arguments;
           arg_op    : num_operande;
           obj_qual  : num_objet;
       -- Un predicat est decrit par son numero, son type, le nombre
       -- de ses arguments et l´adresse du premier de ces arguments.
       -- Tous les predicats ayant le meme nom ont le meme numero,
       -- grace au paquetage generique genhash
        WHEN predicat =>
           nom_pred    : num_predicat;
           type_pred   : genre_predicat := normal;
           nb_arg_pred : nombre_arguments;
           arg_pred    : num_operande;
           code        : numero_code;
       -- On ne stocke pas le nom de la variable (inutile sauf au
       -- moment de la construction de l´arbre de la clause).
       -- Une variable est donc reduite a son deplacement dans
       -- l´environnement
        WHEN variable =>
           dep : deplacement;
       --L´objet all_free est un objet special qui correspond
       --a une variable avec laquelle l´unification reussit
       --toujours. Il est necessaire pour l´implantation de la
       --constante de multi-S4: fool.
        WHEN allfree =>
```

```
            NULL;
         --Un entier est determine par sa valeur
          WHEN entier =>
            val_ent : valeur_entier;
         --Ainsi qu´un flottanr
          WHEN flottant =>
            val_flot : valeur_flottant;
         -- Un cons est une paire pointe (car,cdr)
          WHEN cons =>
            car : num_objet;
            cdr : num_objet;
      END CASE;
   END RECORD;
```

Nous n'allons pas détailler l'ensemble de cette structure, cependant certains points doivent être précisés :

- En ce qui concerne les opérateurs, les numéros d'opérateurs sont définis comme des `NEW NATURAL`. Ce choix peut surprendre, mais les numéros d'opérateurs pour une logique donnée doivent être définis de façon externe au paquetage des types de base. Si les opérateurs étaient définis dans ce paquetage, toute modification entraîinerait une recompilation complète de l'ensemble. On utilise donc un type suffisamment floue pour permettre une définition ultérieure par un paquetage approprié, le paquetage **operateurs**.
- Le champ `obj_qual` des opérateurs est le chaîinage vers l'objet suivant de la clause.
- Un certain nombre d'objets admettent des arguments (c'est le cas des prédicats, des opérateurs). Nous avons choisi la méthode suivante pour l'implantation des arguments des objets : un champ particulier (`arg_op` dans le cas des opérateurs) pointe sur un élément de la pile des opérandes[4]. L'élément pointé contient l'adresse du premier opérande de notre opérateur. L'élément suivant de la pile des opérandes est le pointeur sur le deuxième élément, et ainsi de suite. Le champ `nb_arg_op` détermine le nombre d'arguments de l'opérateur.
- Le type du prédicat est un type énumératif qui peut prendre deux valeurs `normal` ou `predefini`. Cela permet de faire la distinction entre prédicats normaux et prédicats prédéfinis. Le champ `code` est le numéro permettant d'identifier un opérateur prédéfini particulier (voir la description du paquetage **predef**.
- Le type `num_predicat` est un pointeur sur une pile sans duplications qui contient les objets de type elt_predicat. Ce type est défini par :

```
TYPE elt_predicat IS
   RECORD
      nom    : string (1 .. 20);
      clause : num_clause;
   END RECORD;
```

[4] le type opérande est simplement défini par `SUBTYPE elt_operande IS num_objet`. Ainsi, la pile des opérandes est simplement une pile de pointeur sur des objets. Nous y revenons plus loin. Il est extrêmement difficile d'éviter les références en avant dans une définition de types.

Chaque objet de cette pile contient donc le nom du prédicat (champ `nom`) ainsi que l'adresse de la première clause dans laquelle il apparaîıt comme prédicat de tête(champ `clause`). Le nom est le nom du prédicat auquel on a adjoint sous forme ASCII son nombre d'arguments. Ainsi le prédicat `toto(x,y)` a en fait pour nom `toto/2` et non `toto`, ce qui permet de distinguer deux prédicats ayant le même nom et des nombres d'arguments différents.

Il est capital que deux prédicats identiques ne puissent apparaîıtre qu'une seule fois, puisque le champ clause contient l'adresse de la première clause permettant de résoudre ce prédicat.

- Le type `valeur_entier` est défini par `TYPE valeur_entier IS RANGE - 2 ** 31 .. 2 ** 31 - 1` afin d'éviter les problèmes de dépendance au compilateur ou à la machine. Le type `valeur_flottant` est défini par `TYPE valeur_flottant IS DIGITS 16 ;` , pour les mêmes raisons.

La figure 8.2 montre un exemple complet d'implantation interne de clause[5]. Nous continuerons à nous y référer dans la suite de ce texte. Il n'y a aucune garantie particulière quant à l'ordre des objets à l'intérieur de la pile des objets. Ils seront là où le parseur les placera. Le premier objet de la clause peut parfaitement être le premier placé dans la pile, ou bien le dernier. L'ordre qui apparaîıt sur la figure 8.2 est purement circonstanciel.

Les clauses : Une clause est définie de la façon suivante :

```
TYPE elt_clause IS
   RECORD
      clause : num_objet;
      nb_var : nombre_variables;
-- Clause suivante ayant le meme predicat de tete
      next   : num_clause;
      pred   : num_objet;
   END RECORD;
```

Le champ `clause` contient le pointeur sur le premier objet de la clause. Le champ `nb_var` contient le nombre de variables de la clause (indispensable au moment de la création de l'environnement du fait pour la résolution). Le champ `next` contient l'adresse de la clause suivante ayant le même prédicat de tête que celle-ci. Ceci accélère la recherche sur les clauses dans nombre de cas. Le champ `pred` contient l'adresse du prédicat de tête de la clause. Ce champ n'est pas indispensable, car connaissant l'adresse du premier objet de la clause, il suffit de suivre la liste des liens. Cependant, cela accélère la résolution de façon notable de stocker l'adresse du prédicat de tête. Les clauses sont également stockés dans une pile, la pile des clauses.

Les environnements : La notion d'environnement est utilisé par le mécanisme de partage de structures. Un environnement est un objet extrêmement simple :

```
TYPE elt_env IS
```

[5] Cette clause ne contient pas de variable. Nous discutons du problème de l'implantation des variables et des environnements plus loin.

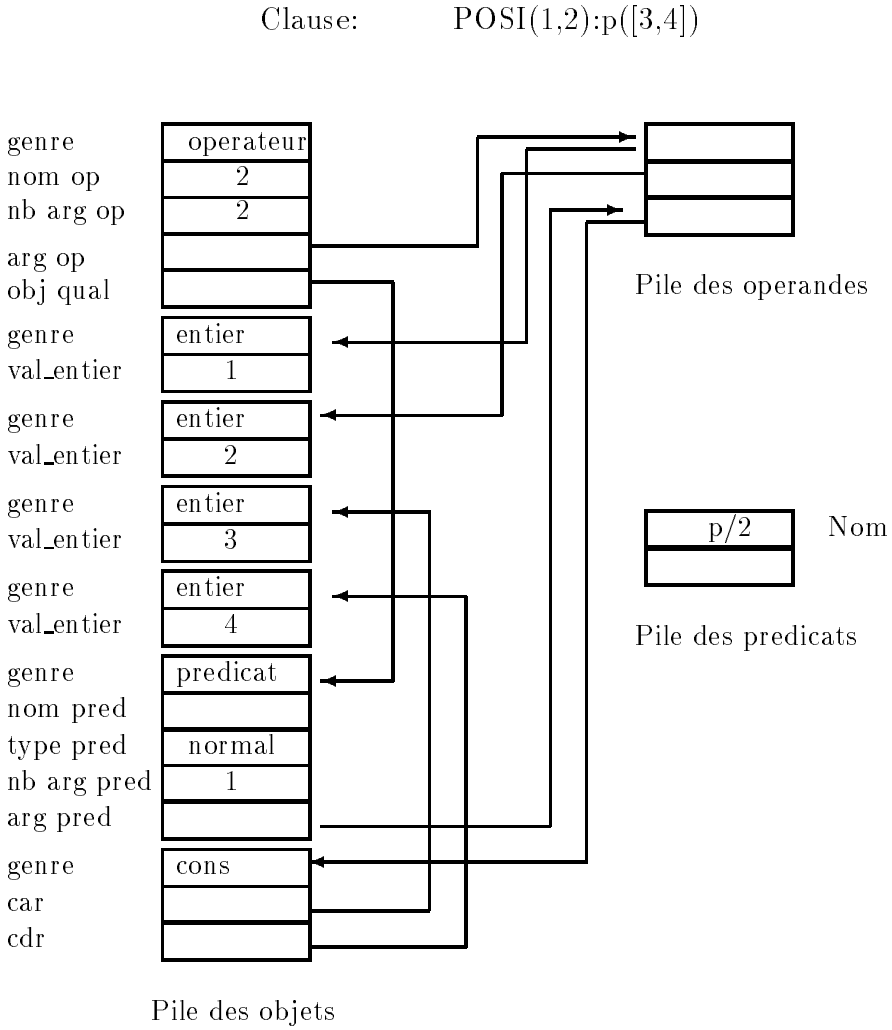


Figure 8.2: Implantation d'une clause

```
RECORD
   num_struct : num_objet;
   num_envi   : num_env;
END RECORD;
```

Nous détaillons la structure des environnements et leur utilisation dans l'annexe B.

La pile de traîiinée : Chaque élément de la pile de traîiinée a une structure extrêmement simple : `SUBTYPE elt_trail IS num_env`. En effet, la pile de traîiinée mémorise les adresses de la pile des environnements qui devront être libérées au backtrack.

La résolvante : Chaque élément de la pile de la résolvante se présente sous la forme suivante :

```
TYPE elt_res IS
   RECORD
      num_struct : num_objet;
      num_envi   : num_env;
   END RECORD;
```

Le champ `num_struct` contient l'adresse de l'objet référencé et `num_envi` l'environnement dans lequel doit être évalué cet objet.

La question : Chaque élément de la pile de la question a exactement la même structure qu'un élément de la pile de la résolvante.

Les points de choix : Les points de choix doivent stocker l'ensemble des informations nécessaires pour effectuer le backtrack. Nous avons détaillé l'ensemble des informations nécessaires dans la section 6.4.7. Un point de choix est implanté comme un record à discriminant, le discriminant étant un type énumératif pouvant prendre trois valeurs : `TYPE genre_backtrack IS (regle, clause, invalide)`. Les deux premières valeurs sont claires : `regle` correspond au cas où le point de choix est un point de choix de règle et `clause` au cas où le point de choix est une clause. Le cas `invalide` est un peu particulier. Il permet à un prédicat prédéfini ou a un opérateur modal de marquer un point de choix comme invalide, ce qui empêchera son utilisation au backtrack[6].

La structure d'un point de choix est :

```
   TYPE elt_backtrack (genre : genre_backtrack := regle) IS
      RECORD
--Base de la pile de la resolvante
         rbottom   : num_res;
--Point courant de la pile de la question
         qcurr     : num_quest;
--Point courant du fait
         fcurr     : num_objet;
--Point courant de l´environnement
         fenv      : num_env;
```

[6] Cette méthode permet en particulier d'implanter simplement le cut.

```
--Sommet de la pile de trainee
         trail_top : num_trail;
--Sommet de la pile des objets
         objet_top : num_objet;
--Sommet de la pile de la question
         q_top     : num_quest;
--Sommet de la pile de la resolvante
         r_top     : num_res;
--Sommet de la pile des environnements
         env_top   : num_env;
         CASE genre IS
            WHEN regle =>
--Numero de la regle courante
               cregle : numero_regle;
            WHEN clause =>
--Numero de la clause courante
               cclause : num_clause;
            WHEN invalide =>
               NULL;
         END CASE;
      END RECORD;
```

Les valeurs des champs vont de soi .

Rappelons que les piles d'objets n'apparaissent pas comme objets exportés, puisqu'elles sont déclarées dans la partie privée du programme. Seules les fonctions permettant d'y accéder apparaissent dans l'interface.

8.5.1.2 Fonctions et procédures exportées

Les opérations sur les "pointeurs" : Sur tous les objets de type "pointeur" sont implantées les classiques fonctions d'incrémentation et de décrémentation. Prenons par exemple le cas du pointeur sur un élément de la pile de backtrack, le type `num_backtrack` :

```
FUNCTION "+" (adr : IN num_backtrack;
              dep : IN integer) RETURN num_backtrack;
FUNCTION "-" (adr : IN num_backtrack;
              dep : IN integer) RETURN num_backtrack;
```

Ces deux fonctions sont implantées pour l'ensemble des types "pointeur".

Les opérations sur les piles : Pour chaque type d'objet qui sera amené à être stocké dans une pile, on exporte les opérations définis dans les paquetages génériques de pile. Prenons l'exemple du type `elt_objet` et de son type pointeur associé `num_objet`. Nous avons les fonctions :

```
FUNCTION empiler (x : IN elt_objet) RETURN num_objet;
FUNCTION reserver (n : IN positive) RETURN num_objet;
```

```
FUNCTION recuperer (i : IN num_objet) RETURN elt_objet;
PROCEDURE modifier (x : IN elt_objet;
                    i : IN num_objet);
PROCEDURE revenir (i : IN num_objet);
FUNCTION position RETURN num_objet;
```

8.5.1.3 Constantes exportées

Les constantes exportées correspondent aux constantes `element_null` des paquetages de pile générique. Prenons l'exemple du type `elt_quest` et de son type pointeur associé `num_quest`. La constante `null_quest` est déclaré par :

```
null_quest : CONSTANT num_quest~;
```

8.5.1.4 Exceptions exportées

Nous devons exporter l'exception `pile_vide`, car c'est lorsque la pile de backtrack est vide (condition "normale") que la résolution est terminée. Nous déclarons donc une exception `pile_vide` dans la partie visible des spécifications.

8.5.2 Implantation

L'implantation des constantes est faite de façon extrêmement simple. Reprenons l'exemple de la constante `null_quest` :

```
null_quest : CONSTANT num_quest :=
             pile_quest.element_null;
```

Le corps des fonctions d'opération sur les piles, fonctions qui ont été déclarée dans les spécifications du paquetage, sont de simples appels aux fonctions exportées par les paquetages génériques de piles. Ainsi, prenons l'exemple de la fonction `FUNCTION empiler (x : IN elt_objet) RETURN num_objet`. Son code est :

```
FUNCTION empiler (x : IN elt_objet) RETURN num_objet IS
BEGIN
   RETURN pile_objet.empiler (x);
END empiler;
```

L'exception `pile_vide` déclarée dans les spécifications est levée dans les cas suivants :

```
FUNCTION recuperer (i : IN num_backtrack) RETURN elt_backtrack IS
BEGIN
   RETURN pile_backtrack.recuperer (i);
EXCEPTION
   WHEN pile_backtrack.pile_vide =>
      RAISE pile_vide;
END recuperer;

FUNCTION depiler RETURN elt_backtrack IS
BEGIN
```

```
   RETURN pile_backtrack.depiler;
EXCEPTION
   WHEN pile_backtrack.pile_vide =>
      RAISE pile_vide;
END depiler;
```

8.6 Les registres

8.6.1 Entités exportées

8.6.1.1 Variables globales

Le paquetage *registres* exporte un certain nombre de variables globales qui sont les registres de la machine abstraite :

Qcurr : Pointeur sur l'élément courant de la pile de la question.

Fcurr : Pointeur sur l'élément courant du fait.

Fenv : Pointeur sur l'environnement courant du fait.

CClause : Pointeur sur la clause courante.

CRule : Pointeur sur la règle d'inférence courante.

TrTop : Pointeur sur le sommet de la pile de trailing.

ObTop : Pointeur sur le sommet de la pile des objets.

BtTop : Pointeur sur le sommet de la pile de backtrack.

QTop : Pointeur sur le sommet de la question.

RTop : Pointeur sur le sommet de la résolvante.

EnvTop : Pointeur sur le sommet de la pile des environnements.

8.6.1.2 Fonctions et procédures exportées

Ce paquetage exporte deux opérations :

```
PROCEDURE sauveregistres (gen : IN genre_backtrack);
```

Cette procédure prend en paramètre le type du point de choix à créer et sauve sur la pile de backtrack l'ensemble des registres de la pile de façon à pouvoir les restaurer en cas de backtrack sur le point de choix courant.

```
FUNCTION restoreregistres RETURN genre_backtrack;
```

Cette fonction restaure l'ensemble des registres contenus dans le point de choix courant et retourne le type de ce point de choix (point de choix sur une règle ou sur une clause).

8.7 L'unificateur

Le paquetage `unify` assure l'unification de deux objets (en utilisant le mécanisme de partage de structures) ainsi que le déréférencement d'une variable.

8.7.1 Entités exportées

8.7.1.1 Fonctions et procédures

Unification : La fonction d'unification :

```
FUNCTION unify (num_objet1, num_objet2 : IN num_objet;
                num_envi1, num_envi2   : IN num_env) RETURN boolean;
```

prend en argument l'adresse de deux couples `(Structure, Environnement)`, unifie ces deux couples si cela est possible et retourne `true`. Si l'unification échoue, `unify` retourne `false`.

Déréférencement : La procédure de déréférencement prend en argument en entrée un couple `(Structure, Environnement)` et en argument de sortie retourne un couple `(Structure, Environnement)` qui est la valeur du couple d'entrée après déréférencement.

8.7.2 Implantation

Nous renvoyons le lecteur à l'annexe B pour les détails de l'implantation du partage de structures.

8.8 Les prédicats pré-définis

8.8.1 Entités exportées

8.8.1.1 Fonctions et procédures

Récupération du numéro de code : Etant donné le nom d'un prédicat, on souhaite pouvoir trouver le numéro de code associé à ce prédicat (c'est le numéro stocké dans le champ `code` de la structure `elt_objet`. La fonction :

```
FUNCTION is_predef (s : IN string) RETURN numero_code;
```

prend en argument une chaîine de caractères correspondant à un nom de prédicat, et retourne le numéro de code associé. Cette fonction est particulièrement utilisée par l'analyseur syntaxique lorsqu'il construit la structure associée à une clause.

Récupération du nom : Etant donné un numéro de code de prédicat, on souhaite pouvoir récupérer le nom de ce prédicat. La fonction :

```
FUNCTION nom_predef (num : IN numero_code) RETURN string;
```

prend en argument un numéro de code et retourne une chaîine de caractères contenant le nom du prédicat.

Exécution : Un prédicat prédéfini est un prédicat qui effectue un certain nombre d'opérations prédéfinis. La fonction :

```
FUNCTION execute_predef (num_elt  : IN num_objet;
                         num_envi : IN num_env) RETURN boolean;
```

prend en argument un pointeur sur un objet qui est un prédicat prédéfini et l'environnement dans lequel ce prédicat doit être évalué. Elle retourne un booléen qui indique si l'exécution du prédicat a été possible ou impossible.

8.8.2 Implantation

Le grand mérite de l'implantation est de laisser les noms, les numéros ainsi que le code exécuté par chaque prédicat dans le corps du paquetage. Il est ainsi possible de modifier le code, ajouter ou supprimer des prédicats prédéfinis, changer les noms des prédicats en ne modifiant que le corps de paquetage, et en ayant que le seul corps à recompiler.

Les prédicats prédéfinis actuellement implantés sont :

`fail` : Echoue toujours.

`succes` : Réussit toujours

`pred(X)` : Réussit si son argument X est un prédicat.

`atomic(X)` : Réussit si son argument X est atomique (ni une liste, ni une variable libre ou liée à une liste).

`cut` : Le classique `cut` PROLOG. Nous allons revenir sur son implantation.

`print(X)` : Affiche son argument X.

`sup(X,Y)` : Réussit si $X > Y$. Lève une exception si X ou Y ne sont pas numériques.

`inf(X,Y)` : Réussit si $Y > X$. Lève une exception si X ou Y ne sont pas numériques.

`sub(X,Y,Z)` : Essaie d'unifier Z à $X-Y$. Lève une exception si X ou Y ne sont pas numériques. Retourne `true` si l'unification réussit, `false` sinon.

`add(X,Y,Z)` : Essaie d'unifier Z à $X+Y$. Lève une exception si X ou Y ne sont pas numériques. Retourne `true` si l'unification réussit, `false` sinon.

`mul(X,Y,Z)` : Essaie d'unifier Z à $X \times Y$. Lève une exception si X ou Y ne sont pas numériques. Retourne `true` si l'unification réussit, `false` sinon.

`div(X,Y,Z)` : Essaie d'unifier Z à X/Y. Lève une exception si X ou Y ne sont pas numériques. Retourne `true` si l'unification réussit, `false` sinon.

Nous allons dire un mot de l'implantation du `cut`. Il y a deux solutions pour implanter le `cut`. La première consiste à simplement éliminer l'ensemble des points de choix entre le point de choix courant et le point de choix de la clause contenant le `cut`. La seconde consiste à marquer comme étant invalide l'ensemble de ces points de choix (d'ou le type `invalide` pour un élément de la pile de backtrack). Le second choix peut paraîıtre curieux, mais présente

l'avantage suivant : au moment du debugging, un dump de la pile de backtrack contiendra l'ensemble des informations signifiantes, y compris celles relatives à l'exécution des `cut`.

Le debugging de TARSKI étant sans doute l'opération la plus délicate, nous avons implanté les deux solutions[7].

8.9 Sélection de clause

Le paquetage de sélection de clause assure pour le moteur d'inférences l'étape paramétrable de la sélection des clauses, telle qu'elle est décrite dans la section 6.6.1.

8.9.1 Entités exportées

8.9.1.1 Fonctions et procédures

Sélection de la première clause : La fonction :

```
FUNCTION premiereclause RETURN num_clause;
```

retourne l'adresse de la première clause sélectionnable dans l'état courant.

Sélection de la clause suivante : La fonction :

```
FUNCTION clausesuivante (num : IN num_clause) RETURN num_clause;
```

prend en argument le numéro de la clause courante et retourne le numéro de la clause suivante sélectionnable.

Ces deux fonctions retournent la constante `null_clause` si aucune clause n'est sélectionnable.

8.9.2 Implantation

Ici encore, le mérite de l'implantation est d'avoir rendu totalement paramétrable la sélection de clause en isolant dans le corps du paquetage l'ensemble du code nécessaire à la sélection de clause.

La seule méthode de sélection de clause implantée est la sélection par la valeur du prédicat de tête de la clause.

La fonction `premiereclause` agit de la façon suivante :

1. Elle récupère le sommet de la question.
2. Elle vérifie s'il s'agit d'un prédicat de tête.
3. Si c'est le cas, le code associé au prédicat est exécuté, et l'on retourne le résultat approprié suivant le résultat de l'exécution.
4. S'il ne s'agit pas d'un prédicat prédéfini, elle cherche la première clause ayant comme prédicat de tête le prédicat de tête de la question. S'il n'y en a aucun, elle retourne `null_clause`.

[7] Une méta-commande permet de choisir le mode `debug` pour l'exécution, ou le mode standard.

5. Si elle trouve une clause, elle devrait simplement retourner son adresse. Cependant, pour des raisons d'optimisation, elle sauve les registres, crée un environnement pour la nouvelle clause et tente d'unifier la tête de la clause et le prédicat au sommet de la question. Si l'unification réussit, elle retourne l'adresse de la première clause, et initialise le pointeur du fait courant `fcurr`.

6. Si l'unification échoue, elle restaure les registres et appelle directement la fonction `clausesuivante`.

La fonction `clausesuivante` va se comporter de la même façon que `premiereclause`. Elle va chercher la première clause ayant le même prédicat de tête que la question courante, et qui suit la clause passée en argument. Si elle n'en trouve pas, elle retourne `null_clause`, sinon, comme `premiereclause`, elle réserve un environnement, essaie d'unifier la tête de la clause à la tête de la question et se rappelle récursivement si l'unification échoue.

8.10 Sélection et exécution de règles

8.10.1 Entités exportées

8.10.1.1 Types exportées

`TYPE resultat_execution IS (reussi, echec, fin)` : Le type `resultat_execution` est un type énumératif. Il est le type retourné par la fonction qui exécute une règle. Les trois résultats possibles de l'exécution d'une règle sont le `succes`, l'`echec` ou la `fin` de la résolution s'il s'agit d'une règle de terminaison.

`TYPE resultat_reecriture IS (ok, not_ok, fin_fin)`: Le type `resultat_reecriture` est un type énumératif. Il est le type retourné par la fonction qui réécrit la résolvante en une nouvelle question. Les trois résultats possibles de la réécriture sont `ok` si la réécriture a réussi, `not_ok` si elle a échoué, ou `fin_fin` si la résolution est terminée (la nouvelle question est vide).

8.10.1.2 Constantes exportées

La constante `null_operateur` est retournée par la fonction `is_operateur` quand aucun opérateur utilisée par cette logique ne correspond au profil passé en argument de la fonction.

La constante `plus_de_regle` est retournée par les fonctions de sélection de règles quand plus aucune règle n'est sélectionnable.

8.10.1.3 Fonctions et procédures

Les fonctions et procédures exportées peuvent être séparées en deux catégories :

Les fonctions nécessaires au fonctionnement du moteur d'inférences : Ces fonctions assurent la sélection des règles d'inférence, l'exécution de ces règles et la réécriture de la résolvante en nouvelle question. Nous allons les détailler :

Sélection de la première règle : La fonction :

```
FUNCTION premiereregle
    (num1, num2 : IN numero_operateur) RETURN numero_regle;
```

prend en paramètre le numéro de l'opérateur du fait et le numéro de l'opérateur de la question et retourne le numéro de la première règle à utiliser.

Sélection de la règle suivante : La fonction :

```
FUNCTION reglesuivante (num : IN numero_regle)
         RETURN numero_regle;
```

prend en paramètre le numéro de la règle courante et retourne le numéro de la règle qui la suit dans l'ordre du chaîinage des règles.

Exécution d'une règle : La fonction :

```
FUNCTION executeregle (num : IN numero_regle)
         RETURN resultat_execution;
```

prend en paramètre le numéro d'une règle et exécute cette règle, modifiant en conséquences les valeurs des piles et des registres de la machine. Elle retourne le résultat de l'exécution.

Réécriture de la résolvante : La fonction :

```
FUNCTION reecrit RETURN resultat_reecriture;
```

réécrit la résolvante en une nouvelle question et modifie en conséquence les registres et les piles.

Les fonctions annexes : Les fonctions annexes sont principalement utilisées par l'analyseur syntaxique, ou pour afficher certaines valeurs ;

La fonction

```
FUNCTION is_operateur  (nom : IN string;
                        nb  : IN nombre_arguments)
         RETURN numero_operateur;
```

prend en argument une chaîine de caractères représentant le nom de l'opérateur et le nombre d'arguments de cet opérateur. Elle retourne le numéro interne (destiné au champ `nom_op`) de l'opérateur.

La fonction

```
FUNCTION nom_operateur (num : IN numero_operateur)
         RETURN string;
```

prend en argument un numéro d'opérateur et retourne son nom sous forme de chaîine de caractères.

8.10.2 Implantation

8.10.2.1 Les opérateurs

Les numéros d'opérateurs sont simplement des constantes déclarées, par exemple, de la façon suivante :

```
et      : CONSTANT numero_operateur := 1;
```

Leurs noms et le nombre de leurs arguments sont placés dans deux tableaux :

```
noms    : ARRAY (premier .. dernier) OF acces_string;
nb_args : ARRAY (premier .. dernier) OF nombre_arguments;
```

ou `dernier` est une constante égale au numéro du dernier opérateur de la logique utilisée et premier est égal au numéro du premier opérateur de la dite logique. Ces tableaux peuvent être initialisés par :

```
noms (et)      := NEW string´("ET");
nb_args (et)   := 1;
```

8.10.2.2 les règles

La structure d'une règle sera la suivante :

```
TYPE regle IS
   RECORD
      next_regle : numero_regle;
      code       : numero_code;
   END RECORD;
```

Le champ `next_regle` indiquera quelle règle est utilisable après celle-ci, le champ `numero_code` est le numéro de bloc de code à utiliser pour exécuter cette règle[8]. L'ensemble des règles est représenté comme un tableau :

```
regles          : ARRAY (premiere_regle .. derniere_regle) OF regle;
```

`premiere_regle` et `derniere_regle` étant des constantes appropriées à la logique considérée. Elles sont déclarées par :

```
premiere_regle  : CONSTANT numero_regle;
derniere_regle  : CONSTANT numero_regle;
```

Comme le suggère la spécification, nous utilisons un tableau doublement indexé pour déterminer quelle première règle est utilisable en fonction de l'opérateur de la question et de l'opérateur du fait :

```
premieresregles : ARRAY (premier .. dernier,
  premier .. dernier) OF numero_regle;
```

Ainsi, étant donné le numéro de l'opérateur du fait et de la question la fonction `premiereregle` est implantée de la façon triviale suivante :

```
FUNCTION premiereregle
    (num1, num2 : IN numero_operateur) RETURN numero_regle IS
BEGIN
   RETURN premieresregles (num1, num2);
END premiereregle;
```

[8] En C, il s'agirait d'un pointeur sur le code de la règle.

La fonction `reglesuivante` est tout aussi simple :

```
FUNCTION reglesuivante (num : IN numero_regle)
         RETURN numero_regle IS
BEGIN
   RETURN regles (num).next_regle;
END reglesuivante;
```

L'exécution des règles est simple à réaliser. La fonction `executeregle` récupère le champ `code` de la règle courante et l'exécute. Ce champ utilise les instructions de la machine de niveau inférieur.

8.10.2.3 La réécriture

Théoriquement, la réécriture devait être réimplanté sous forme de règles de réécriture, chaque règle étant appelée jusqu'à ce que plus aucune règle de réécriture ne soit applicable.

Nous ne nous en sommes pas exactement tenus à ce modèle formel. Pour des raisons d'efficacité, les règles de réécriture ont été compactées en une seule fonction pour une logique donnée (la fonction `reecrit`) qui contient la totalité du code.

La façon dont la fonction `reecrit` se comporte est totalement dépendante de la logique. Nous verrons dans le chapitre 11 un exemple complet d'implantation d'une logique, ainsi que le code de la fonction `reecrit`.

8.11 Le moteur d'inférence

8.11.1 Entités exportées

Le paquetage **moteur** n'exporte qu'une seule entité, la procédure `principal`. Cette procédure sera appelée après l'analyse syntaxique. Elle commence la résolution, en supposant que toutes les piles ont été correctement initialisées par l'analyseur syntaxique.

8.11.2 Implantation

L'implantation du moteur d'inférences suit exactement le modèle décrit dans le chapitre précédent. Il s'agit d'une machine à quatre états. Le moteur commence son cycle dans l'état SC1 (voir section 6.6.2.3).

Nous allons rapidement décrire l'implantation de chacun des états.

SC1 : Le moteur appelle la fonction `premiereclause`. Si la valeur retournée est `null_clause`, il passe dans l'état **BT**, sinon il passe dans l'état **SR1**.

SR1 : Le moteur récupère le sommet de la question. S'il s'agit d'un prédicat, alors l'opérateur de la question est initialisé à `null_operateur`. S'il s'agit d'un opérateur, alors l'opérateur de la question est égal à cet opérateur. Il effectue la même opération pour le fait courant, et appelle la fonction `premiereregle` en lui passant en paramètre l'opérateur du fait et celui de la question. Si la fonction `premiereregle` retourne `plus_de_regle` alors le moteur passe dans l'état **BT**, sinon le moteur passe dans l'état **ER**.

ER : Le moteur appelle la fonction `executeregle`. Si la valeur de retour de la fonction est :

`echec:` Le moteur passe dans l'état **BT**.

`reussi:` Le moteur passe dans l'état **SR**.

`fin:` Le moteur appelle la fonction de réécriture de la résolvante `reecrit`. Suivant le résultat de cette fonction :

`ok:` Le moteur passe dans l'état **SC1**.

`not_ok:` Le moteur passe dans l'état **BT**.

`fin_fin:` Si l'option choisie est de trouver toutes les solutions, alors le moteur passe dans l'état **BT**. Si l'option choisie est de s'arrêter à la première solution, alors le programme exécute un `RETURN` rendant le contrôle à l'appelant et terminant la résolution.

BT : Dans l'état **BT**, le moteur commence par dépiler l'ensemble des points de choix marqués comme `invalide`. Puis il restaure les registres à l'aide de la sauvegarde effectuée dans le dernier point de choix valide. En fonction de la forme de ce point de choix :

`regle:` Si le point de choix est un point de choix de règle, alors le moteur appelle la fonction `reglesuivante` en lui passant en paramètre la règle courante. Si la valeur de retour de la fonction est `plus_de_regle`, le moteur reste dans l'état **BT**. Sinon, il passe dans l'état **ER**.

`clause:` S'il s'agit d'un point de choix de clause, le moteur appelle la fonction `clausesuivante` avec la clause courante comme paramètre. Si la valeur de retour de la fonction est `null_clause` alors le moteur reste dans l'état **BT**, sinon il passe dans l'état **SR**.

8.12 L'analyseur syntaxique

La méthode classique pour implanter un analyseur syntaxique est l'utilisation des classiques outils UNIX **Lex** et **Yacc**. Malheureusement, au moment où nous nous sommes lancés dans l'implantation de l'analyseur syntaxique, l'équivalent de Lex et Yacc n'existait pas pour ADA. Il nous a donc fallu implanter l'analyseur syntaxique "à la main". Par chance, la notation préfixée fait de TARSKI un langage parsable par un analyseur syntaxique LL(1). Dans ces conditions, l'implantation a été simple.

8.12.1 Entités exportées

8.12.1.1 Procédures et fonctions

Une seule procédure est exportée :

```
PROCEDURE parse (p : IN string);
```

Cette procédure prend en argument une chaîine de caractères contenant la ligne de programme à analyser, et remplit en conséquence les piles du système.

8.12.1.2 Exceptions

Deux exceptions sont exportées :

`erreur_de_syntaxe` : `EXCEPTION:` Cette exception est levée par l'analyseur syntaxique quand il rencontre une séquence incorrecte de jetons.

`operateur_inconnu` : `EXCEPTION:` Cette exception est levée quand le parseur rencontre dans le programme un opérateur non déclaré par les règles de la logique courante.

8.12.2 Implantation

L'implantation de l'analyseur syntaxique est tout ce qu'il y a de plus classique. L'analyse étant LL(1), l'analyseur syntaxique parcourt la chaîine de la gauche vers la droite et génère le code au fur et a mesure de l'analyse. Il se rappelle récursivement si nécessaire.

Nous ne nous étendrons pas sur l'analyse syntaxique, une des améliorations a apporté à TARSKI étant certainement de recoder complètement l'analyseur syntaxique avec **ALex** et **AYacc**.

8.13 Debugging

En dehors des spécifications, qui sont exactement implantées comme vu ci-dessus, nous avons intégré dans le système un certain nombre de facilités pour le debugging, ou l'explication de la résolution.

Ces facilités sont contrôlées par la valeur de deux variables booléennes, `trace` et `explain`. Ces variables peuvent être positionnées par l'utilisateur à l'aide de méta-commandes, dans le programme qu'analysera le parser. Le comportement de ces variables est le suivant :

`trace:` Quand `trace` est positionnée à `true`, chaque étape de la résolution est tracée. Cela concerne chaque étape du moteur d'inférences, ainsi que chaque sélection de clause, chaque sélection de règle, chaque exécution de règle. La trace d'un programme TARSKI peut-être extrêmement volumineuse. Ainsi, la trace du classique "wise men problem", qui ne comprend que 17 clauses, est un fichier de plus de 60 méga octets, autrement dit complètement inutilisable à la main. La commande `trace` ne peut-être utilisée que sur des résolutions relativement courtes.

`explain:` Quand `explain` est positionnée à `true`, le moteur affiche, chaque fois qu'il trouve une solution, la suite des opérations qui ont été effectuées pour parvenir à cette solution est affichée, avec l'ensemble des informations afférentes, comme si l'on avait effectué une trace sélective sur la branche partant de la racine et amenant à la solution.

Le problème du debugging avec TARSKI est extrêmement difficile. Le volume des traces est tel qu'il faudrait disposer d'outils graphiques adaptés pour pouvoir traiter correctement les fichiers de trace.

8.14 Conclusion

Nous avons dans ce chapitre décrit l'ensemble des paquetages implantant les machines abstraites introduites dans la partie "conception". Rappelons quelles sont les instructions du

système TARSKI lui-même, c'est à dire les instructions effectivement utilisées pour implanter une nouvelle logique.

Pour pouvoir traduire les règles de résolution, nous avons besoin de deux types d'opérations :

Les opérations sur les piles et les registres : toute exécution de règle exige en général une manipulation des piles de la machine (résolvante, question) et des registres (pointeur sur le fait courant, pointeur sur l'élément courant de la question, etc...)

Les opérations d'unification et de déréférencement : l'exécution d'une règle nécessite souvent d'effectuer une opération d'unification, ou une opération de déréférencement.

Au niveau de l'implantation, l'intégration de nouvelles règles se fait donc au niveau du paquetage **opérateurs**. Le mécanisme a été conçu de façon à ce que l'ajout de règles n'influe que sur le corps de ce paquetage, ce qui permet des intégrations faciles et rapides de nouvelles règles.

Chapitre 9

Implantation du parallélisme

9.1 Contraintes matérielles

Les problèmes liés au parallélisme ont déjà été largement évoqués dans les chapitres précédents. Rappelons cependant les principes généraux sur lesquels nous devons fonder notre implantation du parallélisme :

- Il faut utiliser le parallélisme intrinsèque de la résolution avec des clauses de Horn généralisées et des règles d'inférences paramétriques. Nous ne souhaitons pas implanter un parallélisme PROLOG de type parallélisme-ET ou parallélisme-OU.

- Les contraintes matérielles nous imposent de travailler sur des machines non-parallèles en réseau à faible débit (10 Mbits théoriques, 1Mb en fait). Le mécanisme utilisé devra donc limiter au maximum les échanges entre processeurs. Aucune des techniques de type mémoire partagée n'est utilisable **alors que ce mécanisme serait probablement le plus adapté**. Malgré tout, il faut éviter de trop restreindre nos choix. Si une machine parallèle venait à être disponible, il faudrait pouvoir l'utiliser sans modifier fondamentalement le mécanisme existant.

- Il faut modifier le moins possible la structure de la machine classique, de façon à circonscrire les problèmes de debugging. Les programmes parallèles ou distribués sont délicats à maîıtriser.

Ces contraintes font que notre implantation sera essentiellement **un test de faisabilité**. Nous ne pouvons espérer réaliser une implantation réellement efficace.

9.2 Choix effectués

9.2.1 Le problème du langage

On remarque un problème tout à fait regrettable : alors que le langage d'implantation (ADA) avait été choisi pour ses capacités multi-tâches, nous nous interdisons d'en profiter puisque nous allons implanter le système sur un réseau de stations et qu'il n'existe aucun exécutif ADA distribué. Pourquoi ce choix?

En fait, lors du choix du langage d'implantation (en 1988), les fabricants de compilateur annonçaient l'apparition d'exécutif distribué pour leur environnement. Malheureusement, cette

promesse n'a pas été tenue. A la fin de l'implantation de la machine classique, lorsqu'il a fallu passer a la machine parallèle, il a aussi fallu se rendre à l'évidence : une version multi-tâches ne serait pas autrement parallèle que sur le papier. Il fallait donc choisir :

- Ou développer une version multi-tâches, qui serait esthétiquement élégante, mais parfaitement inefficace.
- Ou implanter soi-même le parallélisme en distribuant explicitement l'exécution entre plusieurs machines "manuellement", sachant bien qu'ADA n'était certainement pas le langage le plus adapté à cet exercice, car il collabore très mal avec le système d'exploitation.

Nous avons choisi la seconde solution. Nous souhaitions que le parallélisme ait une influence réelle sur les temps d'exécution, même si cela impliquait une complexité supérieure pour la réalisation du système. Ce choix a occasionné pas mal de difficultés dans l'implantation, nous allons y revenir.

9.3 Communication inter-processeurs

Le problème techniquement difficile à résoudre est la communication inter-processus. Alors qu'en C le problème est relativement trivial, il n'en va pas de même en ADA.

9.3.1 Choix du type de communication

Il n'existait guère que trois alternatives pour réaliser la communication inter-processus :

- Utiliser un fichier comme médium de communication en s'appuyant sur NFS pour assurer la distribution des données. Cette solution présente l'avantage de ne pas avoir à développer d'interfaçage d'ADA avec le système. Mais elle est tellement lente et on pourrait presque dire tellement ridicule, qu'elle n'a pas été sérieusement envisagée.
- Utiliser les sockets INTERNET (les sockets UNIX ou les pipes sont évidemment inutilisables en multi-machines). Cette solution est plus complexe à mettre en œuvre. En effet, elle impose l'interfaçage d'ADA avec le monde UNIX, opération délicate pour plusieurs raisons sur lesquelles nous allons avoir l'occasion de revenir. D'autre part, elle impose de développer une couche de "mise en forme"des données à l'émission et à la réception.

  En revanche, elle garantit une vitesse de transfert aussi rapide que possible dans les circonstances considérées.
- Utiliser le mécanisme de Remote Procedure Call (RPC) pour les appels et le protocole XDR pour le passage des données. Ce système serait de loin le plus élégant dans notre cas. Un ensemble de démons Tarski tournant sur plusieurs machines et déclenchables par simple appel RPC serait à la fois propre et esthétique. Malheureusement, cette technique se heurtait à des difficultés pratiques presque insurmontables. L'implantation du mécanisme de RPC impose la possibilité de passer des adresses de procédures comme paramètres de fonction, ce dont ADA est bien incapable.

La solution retenue a donc été la solution 2.

9.3.2 Protocole de communication

Nous avons donc choisi de faire communiquer nos machines à l'aide de sockets INTERNET. Reste à choisir le protocole (UDP ou TCP) et à implanter la solution.

Le protocole UDP (User Datagram Protocol) pourrait se justifier si l'on souhaitait faire de la diffusion multi-machines sans acquittement de réception. Ce n'est pas là notre but. Nous souhaitons envoyer les données à une seule machine, et nous voulons être certains que les données sont bien parvenues. Le choix de TCP (Transfer Control Protocol) s'impose donc de lui-même.

9.3.3 Implantation de la couche basse

L'implantation des communications par sockets s'est révélée un problème difficile, sans doute parce que nous avons placé notre niveau d'exigence très haut. Nous voulions en effet implanter un système permettant d'appeler à partir d'ADA les primitives UNIX de façon transparente, mais en conservant le typage fort ADA et surtout en conservant au processus ADA son caractère multi-tâches.

Il faut revenir sur ce dernier point pour bien comprendre le problème ; un appel système UNIX, lorsqu'il est bloquant (ce qui est le cas des appels standard `read` et `write`) est bloquant pour l'ensemble du processus appelant. Or un programme ADA même multi-tâches est implanté dans un seul processus. Une tâche effectuant un appel bloquant bloque donc l'ensemble du processus et donc **toutes les autres tâches**. Ceci est inadmissible en ADA. Il fallait donc trouver une solution à ce problème.

Le résultat de nos efforts a été le système PARADISE[1]. Le but de cette thèse n'est pas de décrire ce système qui ne devait être qu'un moyen et non un but, d'autant qu'il a été décrit dans d'autres publications ([AP92]). Le développement du système a été réalisé dans un premier temps conjointement avec un technicien du CENA (Francis Preux), avant qu'il soit placé dans le domaine public. Le succès de ce système nous a obligé à en abandonner la maintenance à l'équipe du CNAM Paris (en particulier a Stéphane Bortzmeyer). PARADISE compte aujourd'hui plus de 10000 lignes d'ADA et a été porté sur une dizaine de systèmes différents. Il est utilisé par plusieurs universités et industriels importants[2].

PARADISE une fois écrit, il a été facile d'implanter le mécanisme de communication inter-processus. Nous n'étions malheureusement pas au bout de nos peines. Les premiers tests ont indiqué des pertes de performance considérables, et une analyse fine a rapidement montré que le système de transfert de données était gravement en cause : il transmettait les piles environ dix fois plus lentement que le programme C équivalent. Nous étions donc de nouveau face à un dilemme :

- Maintenir le mécanisme PARADISE, mais notre parallélisme devenait alors l'objet de la risée publique : le système parallèle avec deux processeurs allait 5 fois plus lentement que le système séquentiel traditionnel.

- Réécrire quelques centaines de lignes de C pour implanter de façon efficace le mécanisme de passage des piles d'un processus à l'autre.

[1] PARADISE signifie *Package of Asynchronous ADA Drivers for Interconnected Systems Exchange*. Je tiens à remercier Khang Vu-Tien (CAP-SESA) pour l'acronyme.

[2] Westinghouse et Verdix, où il est utilisé dans un cadre professionnel, côtoient ainsi le CNAM ([Bor91]) ou l'ENSEEIHT, où il est utilisé pour les cours systèmes.

Nous avons opté pour la seconde solution. Ceci a amélioré les performances d'un facteur dix[3].

9.3.4 Protocole de transfert

Les registres sont tout d'abord transférés en image binaire. *Cela signifie que le système ne fonctionnerait pas entre une machine "little indian" et une machine "big endian"*. On ne prend aucune précaution particulière sur l'ordre des octets.

On transfère ensuite les piles de données. Le protocole utilisée est simple : on transfère d'abord le nombre d'objets qui va être envoyé, puis les objets eux-mêmes, le tout sous forme binaire. Là encore, on suppose qu'il y a compatibilité binaire entre chacune des deux machines, l'émetteur et le récepteur.

9.3.5 Objets véritablement transférés

Le transfert de toutes les piles et de tous les registres garantit que la résolution reprendra bien au point souhaité. Mais est-il nécessaire de transférer l'ensemble des piles? Non.

En effet, il est absolument impossible de backtracker au delà du point courant qui est le point de début de résolution. Donc les piles de backtrack et de trail sont parfaitement inutiles et peuvent parfaitement ne pas être transférées. Elles ne le sont donc pas dans l'implantation courante.

Il serait possible de réaliser d'autres optimisations. En particulier, il est souvent absurde de transférer complètement les piles à chaque distribution d'une branche de l'arbre. Il vaudrait mieux n'envoyer que l'information "nouvelle". Cette opération n'est cependant pas triviale à réaliser. La seule méthode efficace consisterait sans doute à mémoriser le niveau de chaque pile au moment du transfert. Dans la suite de la résolution, on noterait pour chaque pile le point le plus bas modifié et lors du transfert suivant on ne re transférerait que les données depuis le point modifié le plus bas jusqu'au sommet de la pile. Cette optimisation n'a pas été implantée.

9.4 Analyse de performances

Nous présentons ici un exemple particulièrement significatif de la parallélisation sur TARSKI. D'autres tests ont été réalisés mais tous donnent des résultats exactement similaires.

Le tableau 9.1 montre les résultats de l'exécution du programme avec 1, 2, 3 ou 4 processeurs sur le problème classique des sages et des chapeaux sur une SPARCStation-II (voir section 11.5). Rappelons que ce problème est résolu par la machine classique en 320s de CPU. On voit donc que

- Le passage de la machine classique à la machine parallèle ne dégrade pas les performances lorsque l'on utilise un seul processeur.
- Lors du passage de 1 à 2 processeurs le gain de temps est réel, puisqu'on divise quasiment par 2 le temps de résolution. Il est assez rassurant de rencontrer ce type de résultats. Cela confirme bien le fait que, le langage étant intrinsèquement parallèle, le gain de performances doit être important.

[3] On pourrait parler d'ironie du sort : développer près de 10000 lignes de code pour ne pas s'en servir ! Mais il y a pire. La dernière version de PARADISE, qui utilise un mécanisme plus élaboré de signaux pour la transmission des données (mécanisme rendu possible par l'évolution des compilateurs), permet d'implanter la transmission de données sans perte de performance...

# de processeurs	CPU	Système	CPU	Système	CPU	Système	CPU	Système
1	319	2						
2	166	10	145	6				
3	129	24	142	50	77	17		
4	129	26	140	46	46	31	22	9

Tableau 9.1: Temps CPU et système utilisé dans un réseau "du haut vers le bas"

# de processeurs	CPU	Système	CPU	Système	CPU	Système	CPU	Système
1	323	2						
2	170	22	147	15				
3	126	53	142	80	70	15		
4	140	51	135	70	40	53	19	12

Tableau 9.2: Temps CPU et système utilisé dans un réseau "du haut vers le bas"

- En revanche, les performances avec trois ou quatre processeurs sont nettement moins bonnes, le temps système consommé augmentant régulièrement.

On retrouve ainsi un problème déjà évoqué : un réseau ETHERNET est dans l'incapacité de donner un débit suffisant pour assurer des performances correctes à un parallélisme massif. Au delà de deux processeurs, le goulot d'étranglement du système devient le médium de communication.

Il est d'autre part très difficile de faire des mesures précises. En fait, les temps obtenus varient de façon parfois importante d'une mesure à l'autre, en fonction de la charge courante du réseau. Des mesures faites dans l'après-midi alors que tous les utilisateurs travaillent sur les stations et les terminaux X montrent une dégradation encore plus nette des performances du système (voir table 9.2).

Cette incapacité à faire des mesures précises est très pénible pour réaliser une mise au point "fine" des différents paramètres de parallélisme. Nous avons par exemple tenter d'implanter les optimisations décrites quant au transfert des données de la pile (nous essayons de ne transmettre dans certains cas, que les données "utiles"). Il a été impossible de mesurer effectivement l'impact de cette amélioration (?), le système semblant dans certains cas être plus efficace et dans d'autres moins efficace.

En conclusion, les résultats que l'on obtient sont bien conformes aux prévisions. Nous pouvons conclure de ces résultats que :

- Le langage étant intrinsèquement parallèle, le parallélisme se réalise sans aucune surcharge au niveau du moteur d'inférence et du mécanisme de résolution. Le parallélisme TARSKI est un parallélisme efficace.

- Le médium de communication est inadapté. Si nous souhaitons poursuivre les travaux sur le parallélisme, il nous faudra disposer de véritables machines parallèles à mémoire partagée.

Partie V

Typologie et exemples d'applications

Chapitre 10

Typologie des règles de résolution

Nous allons dégager dans ce chapitre un certain nombre de formes classiques de règles de résolution et montrer comment implanter le code de ces formes générales.

Dans le chapitre suivant, nous montrerons des exemples de l'utilisation de ce code pour S1, la logique des modules et la logique floue.

Il faut bien remarquer que différentes logiques utilisent le même code de règle, car leurs règles de résolution ont la même typologie.

10.1 Forme 1

La forme 1 correspond au cas des règles de terminaison :

$$p, ?p \vdash ?true$$

Le code normal de la règle devrait réaliser l'unification des prédicats et, en cas de succès, empiler un élément $\emptyset$ sur la résolvante. En cas d'échec de l'unification, la règle devrait échouer. Mais, pour des raisons d'optimisation, les opérations d'unification sont réalisées lors de la sélection de la clause, donc il est inutile d'essayer à nouveau d'unifier les prédicats, puisque cela a déjà été fait.

Le code est donc extrêmement simple :

```
            DECLARE
               eltres : elt_res;
               adrres : num_res;
            BEGIN
               -- On empile l´element nul
               eltres.num_struct := null_objet;
               eltres.num_envi   := null_env;
               adrres            := empiler (eltres);
--On retourne fin et non succes, car il s´agit d´une regle de terminaison
               RETURN fin;
            END;
```

10.2 Forme 2

La forme 2 a pour résultat d'empiler l'élément courant de la question dans la résolvante et faire avancer le pointeur sur la question. Cette forme pourra s'appliquer à toutes les règles de la forme :

$$\frac{A, ?X : B \vdash ?X : C}{A, ?B \vdash ?C}$$

Ici X représente n'importe quelle séquence.

Le code d'implantation de cette forme est très simple (comme l'est le code de toutes les formes) :

```
            DECLARE
               eltres   : elt_res;
               adrres   : num_res;
               eltquest : elt_quest;
            BEGIN
--Recuperation de l´element courant de la question
               eltquest          := recuperer (qcurr);
--Stockage de cet element dans la resolvante
               eltres.num_struct := eltquest.num_struct;
               eltres.num_envi   := eltquest.num_envi;
               adrres            := empiler (eltres);
--Incrementation du pointeur sur la pile de la question
               qcurr             := qcurr + 1;
--L´ex\´ecution a reussi
               RETURN reussi;
            END;
```

Nous voyons sur la figure 10.1 l'effet de l'exécution de la règle sur les registres et sur les piles.

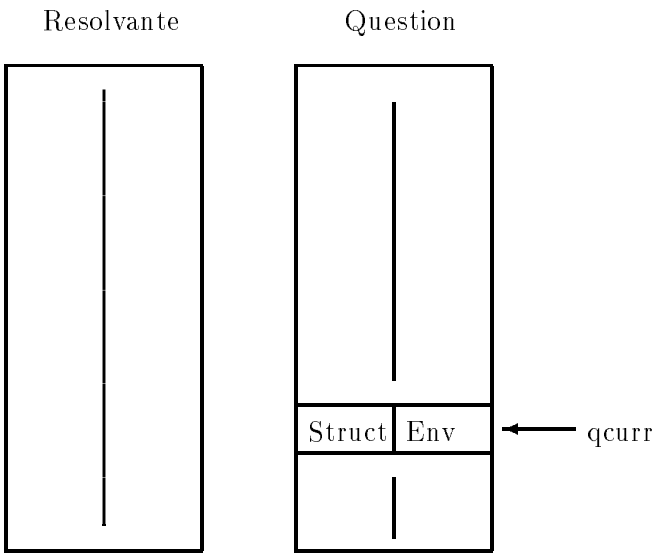
10.3 Forme 3

La forme 3 a pour résultat d'empiler l'élément courant du fait dans la résolvante et faire avancer le pointeur sur le fait. Cette forme pourra s'appliquer à toutes les règles de la forme :

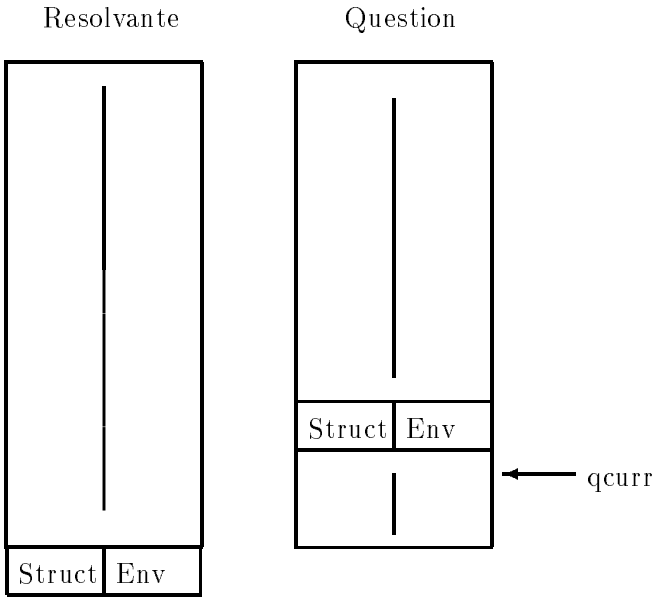
$$\frac{X : A, ?B \vdash ?X : C}{A, ?B \vdash ?C}$$

Le code est :

```
            DECLARE
               eltobjet : elt_objet;
               eltres   : elt_res;
               adrres   : num_res;
            BEGIN
--Empilage de l´objet courant du fait (repere par le registre fcurr)
--ainsi que de l´environnement du fait (registre fenv) sur la pile de
--la resolvante
```



Avant execution du code



Apres execution du code

Figure 10.1: Exécution de la forme 2

```
               eltres.num_struct := fcurr;
               eltres.num_envi   := fenv;
               adrres            := empiler (eltres);
--Recuperation de l´adresse de l´objet suivant. Il s´agit de l´objet
--qualifie par l´objet courant. Le pointeur sur le fait est
--positionne sur cet objet.
               eltobjet          := recuperer (fcurr);
               fcurr             := eltobjet.obj_qual;
               RETURN reussi;
            END;
```

Nous voyons sur la figure 10.2 l'effet de l'exécution de la règle sur les registres et sur les piles.

10.4 Forme 4

La forme 4 a pour résultat de faire avancer le pointeur sur la question. Cette forme pourra s'appliquer à toutes les règles du type :

$$\frac{A, ?X : B \vdash ?C}{A, ?B \vdash ?C}$$

Le code est :

```
-- Fait avancer la question d´un cran
qcurr := qcurr + 1;
RETURN reussi;
```

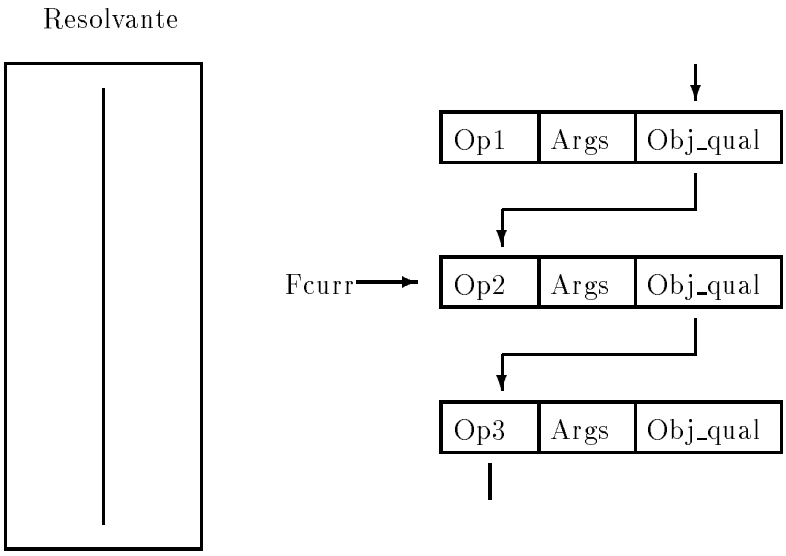
10.5 Forme 5

La forme 5 a pour résultat de faire avancer le pointeur sur le fait. Cette forme pourra s'appliquer à toutes les règles du type :

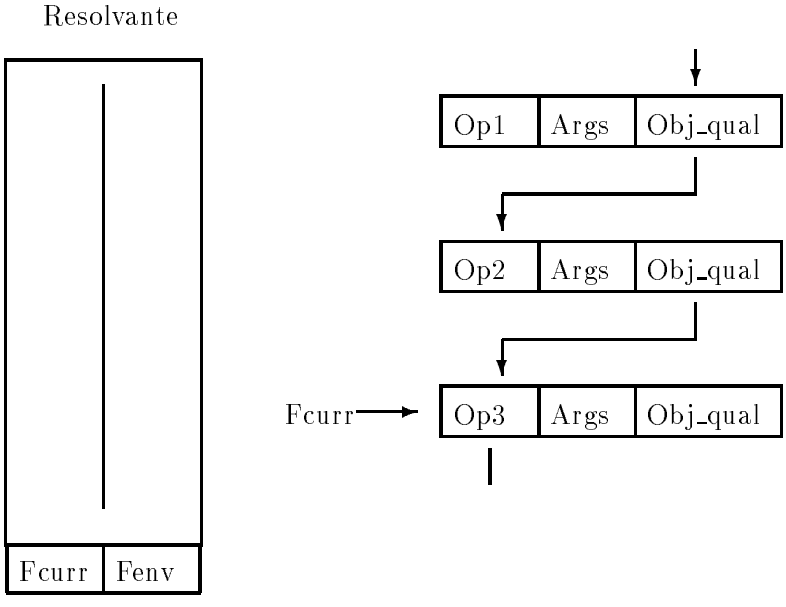
$$\frac{X : A, ?B \vdash ?C}{A, ?B \vdash ?C}$$

Le code est :

```
            DECLARE
               eltobjet : elt_objet;
            BEGIN
--Recuperation de l´element courant pointe par le registre fcurr
               eltobjet := recuperer (fcurr);
--Mise a jour de fcurr
               fcurr    := eltobjet.obj_qual;
               RETURN reussi;
            END;
```



Avant execution du code



Apres execution du code

Figure 10.2: Exécution de la forme 2

10.6 Forme 6

Cette forme unifie les deux éléments courants du fait et de la question et empile un des deux (peu importe lequel) sur la pile de la résolvante si l'unification réussit, puis fait avancer le pointeur sur le fait et le pointeur sur la question. Si l'unification échoue la règle échoue également. Cette forme s'applique à des règles du type :

$$\frac{X : A, ?X : B \vdash ?X : C}{A, ?B \vdash ?C}$$

Le code est un peu plus complexe :

```
            DECLARE
               eltobjet   : elt_objet;
               eltres     : elt_res;
               eltquest   : elt_quest;
            BEGIN
--Recuperation de l´element courant de la question
               eltquest   := recuperer (qcurr);
--Tentative d´unification de l´element courant du fait dans son
--environnement et de l´element courant de la question
               IF unify (fcurr, eltquest.num_struct, fenv, eltquest.num_envi)
                   THEN
--L´unification a reussi.
--On empile l´element courant du fait sur la resolvante
                   eltres.num_struct := fcurr;
                   eltres.num_envi   := fenv;
                   adrres            := empiler (eltres);
--On fait avancer le pointeur sur le fait
                   eltobjet          := recuperer (fcurr);
                   fcurr             := eltobjet.obj_qual;
--On fait avancer le pointeur sur la question
                   qcurr             := qcurr + 1;
--On retourne un succes
                   RETURN reussi;
               ELSE
--L´unification a \´echoue, c´est un echec total.
                   RETURN echec;
               END IF;
            END;
```

10.7 Forme 8

Cette forme[1] unifie le premier opérande du fait et le premier opérande de la question et empile l'élément courant de la question sur la pile de la résolvante si l'unification reussit, puis fait avancer le pointeur sur la question seulement.

[1] On peut se demander pour quelle raison l'on passe de la forme 6 à la forme 8. En fait, il a existé une forme 7, qui s'est révélée inutile, car traitée par une autre forme. Pour des raisons historiques, son numéro n'a pas été supprimé.

Elle s'applique à des règles de la forme :

$$\frac{O(X,Y_1,\ldots,Y_n) : A, ?M(X,Z_1,\ldots,Z_m) : B \vdash ?M(X,Z_1,\ldots,Z_m) : C}{O(X,Y_1,\ldots,Y_n) : A, ?B \vdash ?C}$$

Par exemple :
Le code est :

```
            DECLARE
               obj1, obj2           : elt_objet;
               eltres               : elt_res;
               adrres               : num_res;
               eltquest             : elt_quest;
               num_oper1, num_oper2 : num_objet;
            BEGIN
--Recuperation de l´objet courant du fait
               obj1       := recuperer (fcurr);
--Recuperation de l´element courant de la question
               eltquest   := recuperer (qcurr);
               obj2       := recuperer (eltquest.num_struct);
--On recupere le premier operande de chacun de ces objets (operande numero 0)
               num_oper1  := recuperer (obj1.arg_op);
               num_oper2  := recuperer (obj2.arg_op);
--On tente l´unification des operandes, dans l´environnement du fait pour
--l´operande du fait, dans l´environnement de la question pour l´operande
--venant de la question
               IF unify (num_oper1, num_oper2, fenv, eltquest.num_envi) THEN
--L´unification a reussi
--On empile l´element de la question sur la resolvante
                  eltres.num_struct := eltquest.num_struct;
                  eltres.num_envi   := eltquest.num_envi;
                  adrres            := empiler (eltres);
--On fait avancer le pointeur sur la question
                  qcurr             := qcurr + 1;
--On retourne un succes
                  RETURN reussi;
               ELSE
--L´unification a echoue : c´est un echec total.
                  RETURN echec;
               END IF;
            END;
```

10.8 Forme 9

Cette forme unifie le premier opérande du fait et le premier opérande de la question, empile l'élément courant du fait sur la pile de la résolvante si l'unification reussit, puis fait avancer le pointeur sur le fait seulement.

Elle s'applique à des règles de la forme :

$$\frac{O(X, Y_1, \ldots, Y_n) : A, ?M(X, Z_1, \ldots, Z_m) : B \vdash ?O(X, Y_1, \ldots, Y_n) : C}{A, ?M(X, Z_1, \ldots, Z_m) : B \vdash ?C}$$

Par exemple :
Le code est :

```
          DECLARE
             obj1, obj2           : elt_objet;
             eltres               : elt_res;
             adrres               : num_res;
             eltquest             : elt_quest;
             num_oper1, num_oper2 : num_objet;
             curr_trail           : num_trail;
          BEGIN
--Recuperation de l´objet courant du fait
             obj1       := recuperer (fcurr);
--Recuperation de l´objet courant de la question
             eltquest   := recuperer (qcurr);
             obj2       := recuperer (eltquest.num_struct);
--Recuperation du premier operande de chacun des objets
             num_oper1  := recuperer (obj1.arg_op);
             num_oper2  := recuperer (obj2.arg_op);
--Tentative d´unification des operandes
             IF unify (num_oper1, num_oper2, fenv, eltquest.num_envi) THEN
--L´unification a reussi
--On empile sur la resolvante l´element courant du fait
                eltres.num_struct := fcurr;
                eltres.num_envi   := fenv;
                adrres            := empiler (eltres);
--On fait avancer le pointeur sur l´element suivant
                fcurr             := obj1.obj_qual;
--Succes
                RETURN reussi;
             ELSE
--Echec
                RETURN echec;
             END IF;
          END;
```

10.9 Forme 10

Cette forme unifie l'élément courant du fait avec l'élément courant de la question. Si l'unification réussit alors la règle **échoue**. Si l'unification échoue alors on empile l'élément courant de la question sur la résolvante

```
          DECLARE
```

```
               eltres     : elt_res;
               adrres     : num_res;
               eltquest   : elt_quest;
            BEGIN
--Recuperation de l´element courant de la question
               eltquest   := recuperer (qcurr);
--Tentative d´unification au fait courant
               IF NOT unify (fcurr, eltquest.num_struct, fenv,
                             eltquest.num_envi) THEN
--Si l´unification echoue alors empilage de l´element courant de
--la question sur la resolvante
                  eltres.num_struct := eltquest.num_struct;
                  eltres.num_envi   := eltquest.num_envi;
                  adrres            := empiler (eltres);
--On fait avancer le pointeur sur la question
                  qcurr             := qcurr + 1;
                  RETURN reussi;
               ELSE
                  RETURN echec;
               END IF;
            END;
```

10.10 Forme 11

Cette forme unifie le premier opérande du fait et le premier opérande de la question, empile l'élément courant de la question sur la pile de la résolvante si l'unification reussit, puis fait avancer le pointeur sur le fait seulement.

Cette forme peut s'appliquer pour toutes les règles du type :

$$\frac{O(X,Y_1,\ldots,Y_n) : A, ?M(X,Z_1,\ldots,Z_m) : B \vdash ?M(X,Z_1,\ldots,Z_m) : C}{A, ?M(X,Z_1,\ldots,Z_m) : B \vdash ?C}$$

```
            DECLARE
               obj1, obj2           : elt_objet;
               eltres               : elt_res;
               adrres               : num_res;
               eltquest             : elt_quest;
               num_oper1, num_oper2 : num_objet;
            BEGIN
--Recuperation de l´objet courant du fait
               obj1       := recuperer (fcurr);
--Recuperation de l´objet courant de la question
               eltquest   := recuperer (qcurr);
               obj2       := recuperer (eltquest.num_struct);
--Recuperation du premier operande de chacun des deux objets
               num_oper1  := recuperer (obj1.arg_op);
               num_oper2  := recuperer (obj2.arg_op);
```

```
--Tentative d´unification
               IF unify (num_oper1, num_oper2, fenv, eltquest.num_envi) THEN
--Si l´unification reussit, on empile sur la resolvante l´objet
--courant de la question
                  eltres.num_struct := eltquest.num_struct;
                  eltres.num_envi   := eltquest.num_envi;
                  adrres            := empiler (eltres);
--Puis on fait avancer le fait courant
                  fcurr             := obj1.obj_qual;
                  RETURN reussi;
               ELSE
                  RETURN echec;
               END IF;
            END;
```

10.11 Remarques

La typologie ci-dessus n'est bien entendu pas complète. Cependant, elle nous a jusqu'à présent suffi pour implanter tous les exemples pratiques auxquels nous nous sommes intéressés (à l'exception d'une des règles du KUT ; nous y revenons dans le chapitre suivant).

Il faut remarquer l'intérêt qu'il y a dans le modèle TARSKI à partager entre plusieurs règles différentes ayant la même même forme le même code.

Il serait également possible de faire une typologie des règles de réécriture. Cela n'a pas été fait, et le code des règles de réécriture est actuellement implanté de façon monolithique (et peu élégante) pour chaque logique. C'est un point qu'il faudrait améliorer.

Chapitre 11

Exemples d'applications

11.1 Rappels

S1 est le système sur lequel nous nous sommes appuyés depuis le début de cet exposé pour l'ensemble de nos exemples. La raison en est simple : il présente la plupart des cas difficiles, et donc intéressants, pour l'implantation. Rappelons rapidement les règles de résolution de S1 :

$$p, ?p \vdash ?\texttt{true}$$

$$\frac{A, ? \wedge (D) : B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C}$$

$$\frac{\wedge(D) : A, ?B \vdash ? \wedge (D) : C}{A, ?B \vdash ?C}$$

$$\frac{A, ?\Diamond(X) : B \vdash ?C}{A, ?B \vdash ?C}$$

$$\frac{\Diamond_I(X, I) : A, ?\Diamond(X) : B \vdash ?\Diamond_I(X, I) : C}{A, ?\Diamond(X) : B \vdash ?C}$$

$$\frac{\Diamond_I(X, I) : A, ?\Diamond_I(X, I) : B \vdash ?\Diamond_I(X, I) : C}{A, ?B \vdash ?C}$$

$$\frac{\Box(X) : A, ?\Diamond(X) : B \vdash ?\Diamond(X) : C}{\Box(X) : A, ?B \vdash ?C}$$

$$\frac{\Box(X) : A, ?\Diamond_I(X, I) : B \vdash ?\Diamond_I(X, I) : C}{\Box(X) : A, ?B \vdash ?C}$$

$$\frac{\Box(X) : A, ?\Diamond(X) : B \vdash ?\Diamond(X) : C}{A, ?\Diamond(X) : B \vdash ?C}$$

$$\frac{\Box(X) : A, ?B \vdash ?C}{A, ?B \vdash ?C}$$

Les règles de réécriture sont :

$$X : \wedge(Y) : \texttt{true} \rightsquigarrow X : Y$$

$$X : \Box(Y) : true \rightsquigarrow X$$

$$X : \Diamond(Y) : true \rightsquigarrow X$$
$$X : \Diamond_I(Y, I) : true \rightsquigarrow X$$

Nous employons sur cet ensemble de règles la méthode de développement exhaustif décrite dans le chapitre 5 pour aboutir au classique tableau 11.1. Nous ne reviendrons pas sur sa construction.

Nous allons maintenant nous intéresser à l'implantation pratique des règles de résolution, puis des règles de réécriture.

11.2 Implantation des règles de résolution

Pour implanter les règles de résolution nous allons utiliser le code développé dans le chapitre précédent.

- Pour la règle :
$$\frac{A, ? \wedge (X) : B \vdash ? \wedge (X) : C}{A, ?B \vdash ?C}$$
nous utiliserons la forme 2. En effet, cette règle correspond exactement à :
$$\frac{A, ?X : B \vdash ?X : C}{A, ?B \vdash ?C}$$
avec X remplacé par $\wedge(X)$.

- La règle :
$$\frac{\wedge(X) : A, ?B \vdash ? \wedge (X) : C}{A, ?B \vdash ?C}$$
utilisera le code de la forme 3 :
$$\frac{X : A, ?B \vdash ?X : C}{A, ?B \vdash ?C}$$
Ici X est remplacé par $\wedge(X)$.

- La règle :
$$\frac{A, ?\Diamond(X) : B \vdash ?C}{A, ?B \vdash ?C}$$
sera implantée par le code de la forme 4 :
$$\frac{A, ?X : B \vdash ?C}{A, ?B \vdash ?C}$$
Il suffit de voir que X remplace $\Diamond(X)$.

- La règle :
$$\frac{\Box(X) : A, ?B \vdash ?C}{A, ?B \vdash ?C}$$
utilisera le code de la forme 5 :
$$\frac{X : A, ?B \vdash ?C}{A, ?B \vdash ?C}$$
Ici, X correspond à $\Box(X)$.

Fait	*Question*	*Règles*
Pred	Pred	$p, ?p \vdash ?true$
Pred	$\wedge$	$\frac{A,?\wedge(X):B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
Pred	$\Diamond$	$\frac{A,?\Diamond(X):B\vdash ?C}{A,?B\vdash ?C}$
$\wedge$	Pred	$\frac{\wedge(X):A,?B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
$\wedge$	$\wedge$	$\frac{\wedge(X):A,?\wedge(Y):B\vdash ?\wedge(Y):C}{\wedge(X):A,?B\vdash ?C}$
$\wedge$	$\Diamond$	$\frac{\wedge(X):A,?\Diamond(Y):B\vdash ?\wedge(X):C}{A,?\Diamond(Y):B\vdash ?C}$
$\wedge$	$\Diamond_I$	$\frac{\wedge(X):A,?\Diamond_I(Y,I):B\vdash ?\wedge(X):C}{A,?\Diamond_I(Y,I):B\vdash ?C}$
$\Box$	Pred	$\frac{\Box(X):A,?B\vdash ?C}{A,?B\vdash ?C}$
$\Box$	$\wedge$	$\frac{\Box(Y):A,?\wedge(X):B\vdash ?\wedge(X):C}{\Box(Y):A,?B\vdash ?C}$
$\Box$	$\Diamond$	$\frac{\Box(X):A,?\Diamond(Y):B\vdash ?C}{A,?\Diamond(Y):B\vdash ?C}$
		$\frac{\Box(Y):A,?\Diamond(X):B\vdash ?C}{\Box(Y):A,?B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{A,?\Diamond(X):B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond(X):B\vdash ?\Diamond(X):C}{\Box(X):A,?B\vdash ?C}$
$\Box$	$\Diamond_I$	$\frac{\Box(X):A,?\Diamond_I(Y,I):B\vdash ?C}{A,?\Diamond_I(Y,I):B\vdash ?C}$
		$\frac{\Box(X):A,?\Diamond_I(X,I):B\vdash ?\Diamond_I(X,I):C}{\Box(X):A,?B\vdash ?C}$
$\Diamond_I$	$\wedge$	$\frac{\Diamond_I(Y,I):A,?\wedge(X):B\vdash ?\wedge(X):C}{\Diamond_I(Y,I):A,?B\vdash ?C}$
$\Diamond_I$	$\Diamond$	$\frac{\Diamond_I(Y,I):A,?\Diamond(X):B\vdash ?C}{\Diamond_I(Y,I):A,?B\vdash ?C}$
		$\frac{\Diamond_I(X,I):A,?\Diamond(X):B\vdash ?\Diamond_I(X,I):C}{A,?\Diamond(X):B\vdash ?C}$
$\Diamond_I$	$\Diamond_I$	$\frac{\Diamond_I(X,I):A,?\Diamond_I(X,I):B\vdash ?\Diamond_I(X,I):C}{A,?B\vdash ?C}$

Tableau 11.1: Règles de S1

- La règle :

$$\frac{\Diamond_I(X,I) : A, ?\Diamond_I(X,I) : B \vdash ?\Diamond_I(X,I) : C}{A, ?B \vdash ?C}$$

sera implantée par la forme 6 :

$$\frac{X : A, ?X : B \vdash ?X : C}{A, ?B \vdash ?C}$$

Ici, X correspond à $\Diamond_I(X,I)$.

- La règle :

$$\frac{\Box(X) : A, ?\Diamond_I(X,I) : B \vdash ?\Diamond_I(X,I) : C}{\Box(X) : A, ?B \vdash ?C}$$

sera implantée par le code de la forme 8 :

$$\frac{O(X,Y_1,\ldots,Y_n) : A, ?M(X,Z_1,\ldots,Z_m) : B \vdash ?M(X,Z_1,\ldots,Z_m) : C}{O(X,Y_1,\ldots,Y_n) : A, ?B \vdash ?C}$$

Ici O correspond à $\Box$, M à $\Diamond_I$.

- La règle :

$$\frac{\Diamond_I(X,I) : A, ?\Diamond(X) : B \vdash ?\Diamond_I(X,I) : C}{A, ?\Diamond(X) : B \vdash ?C}$$

utilisera le code de la forme 9 :

$$\frac{O(X,Y_1,\ldots,Y_n) : A, ?M(X,Z_1,\ldots,Z_m) : B \vdash ?O(X,Y_1,\ldots,Y_n) : C}{A, ?M(X,Z_1,\ldots,Z_m) : B \vdash ?C}$$

Il suffit de remplacer O par $\Diamond_I$ et M par $\Diamond$.

Ainsi, le problème de l'implantation du code des règles est réglé.

11.2.1 Enchaîinement des règles

Nous ne reviendrons pas sur la technique utilisée pour le codage du chaîinage des règles, nous l'avons décrite en détail dans la section 8.10 du chapitre 6.

Nous allons tout d'abord présenter le code qui représente les accès doublement indexés sur les règles. Rappelons que les numéros représentent des numéros de règle et non des numéros de forme (ou de code). Ces numéros sont choisis de façon parfaitement arbitraires par l'utilisateur.

```
--null_operateur est le codage de PREDICAT
   premieresregles (null_operateur, null_operateur) := 1;
   premieresregles (null_operateur, et)             := 2;
   premieresregles (null_operateur, pos)            := 4;
   premieresregles (null_operateur, nec)            := plus_de_regle;
   premieresregles (null_operateur, posi)           := plus_de_regle;

   premieresregles (et, null_operateur) := 3;
   premieresregles (et, et)             := 2;
   premieresregles (et, pos)            := 3;
```

```
premieresregles (et, nec)           := plus_de_regle;
premieresregles (et, posi) := 3;

premieresregles (pos, null_operateur) := plus_de_regle;
premieresregles (pos, et)             := plus_de_regle;
premieresregles (pos, pos)            := plus_de_regle;
premieresregles (pos, nec)  := plus_de_regle;
premieresregles (pos, posi) := plus_de_regle;

premieresregles (nec, null_operateur) := 5;
premieresregles (nec, et)             := 2;
premieresregles (nec, pos)            := 6;
premieresregles (nec, nec)            := plus_de_regle;
premieresregles (nec, posi)           := 8;

premieresregles (posi, null_operateur) := plus_de_regle;
premieresregles (posi, et)             := 2;
premieresregles (posi, pos)            := 9;
premieresregles (posi, nec)            := plus_de_regle;
premieresregles (posi, posi)           := 10;
```

Le tableau suivant est le tableau des règles à proprement parler. Chaque règle est constituée de son code (qui correspond à une des formes décrites ci-dessus) et de l'adresse de la règle suivante qu'il est possible d'exécuter (si plusieurs règles sont possibles). Par défaut, il n'y a pas de règle suivante.

```
--Regle predicat,predicat
   regles (1).code        := 1;
--Regle d´elimination et d´empilage dans la resolvante de l´operateur
--de la question (par exemple predicat,et)
   regles (2).code        := 2;
--Regle d´elimination et d´empilage dans la resolvante de l´operateur
--du fait (par exemple predicat,et)
   regles (3).code        := 3;
--Regle du type Pred,POS
   regles (4).code        := 4;
--Regle du type NEC,Pred
   regles (5).code        := 5;
--Regle du type NEC,POS
--Premier exemple ou il y a plusieurs regles possibles pour resoudre
--ce cas. Bien remarquer le chainage sur la regle suivante. Les
--formes utilisees sont les successivement 8,11,5,4
   regles (6).code        := 8;
   regles (6).next_regle  := 23;
   regles (23).code       := 11;
   regles (23).next_regle := 24;
   regles (24).code       := 5;
   regles (24).next_regle := 25;
```

```
   regles (25).code       := 4;
--Cas NEC,POSI. Autre exemple ou il y a plusieurs regles possibles
   regles (8).code        := 8;
   regles (8).next_regle  := 21;
   regles (21).code       := 5;
-- Cas POSI,POS. La deux regles possibles (formes 9 et 4)
   regles (9).code        := 9; --POSI,POS
   regles (9).next_regle  := 22;
   regles (22).code       := 4;
--Regle POSI,POSI
   regles (10).code       := 6;
```

Pour bien comprendre le fonctionnement de ce mécanisme, nous allons examiner le cas où l'opérateur du fait est $\Diamond_I$ et l'opérateur de la question est $\Diamond$, en le reprenant depuis le début.

Le tableau 11.1 nous dit qu'il y a deux règles applicables :

$$\frac{\Diamond_I(X,I) : A, ?\Diamond(X) : B \vdash ?\Diamond_I(X,I) : C}{A, ?\Diamond(X) : B \vdash ?C}$$

$$\frac{\Diamond_I(Y,I) : A, ?\Diamond(X) : B \vdash ?C}{\Diamond_I(Y,I) : A, ?B \vdash ?C}$$

Si nous regardons le codage de `premiereregles`, nous voyons que dans le cas où l'opérateur du fait est $\Diamond_I$ (POSI) et l'opérateur de la question $\Diamond$ (POS), la première règle applicable est la règle 9. Si nous regardons maintenant le codage de `regle`, nous voyons que la règle 9 a pour code associé la forme 9, qui correspond bien à la règle de résolution :

$$\frac{\Diamond_I(X,I) : A, ?\Diamond(X) : B \vdash ?\Diamond_I(X,I) : C}{A, ?\Diamond(X) : B \vdash ?C}$$

Nous voyons également que la règle 9 admet une règle suivante, la règle 22. Cette règle a pour forme associée la forme 4 qui correspond bien à la règle de résolution :

$$\frac{\Diamond_I(Y,I) : A, ?\Diamond(X) : B \vdash ?C}{\Diamond_I(Y,I) : A, ?B \vdash ?C}$$

La règle 22 n'a pas de règle suivante.

L'ensemble des autres éléments de `regle` et `premieresregles` peuvent être ainsi facilement construits.

11.2.2 Conclusion

Nous avons examiné l'ensemble du code nécessaire à l'écriture des règles de résolution. Nous espérons avoir montré qu'il suffisait de peu de lignes de code pour représenter des règles logiques relativement complexes.

11.3 Règles de réécriture

Comme nous l'avons déjà dit dans la section 8.10.2.3, la partie consacrée à la réécriture n'a pas été implanté de façon conforme aux spécifications. Au lieu d'un ensemble de règles que l'on appellerait successivement, nous avons écrit un bloc de code monolithique.

Ce bloc est optimal au niveau temps d'exécution : on a gagné en temps, perdu en élégance et en lisibilité.

Le principe général que nous avons adopté est le suivant : les règles de réécriture stipulent en fait que tous les éléments entre le sommet de la résolvante et le premier opérateur $\wedge$ peuvent être éliminés, puisque nous avons simultanément : $X : \Box(Y) : true \rightsquigarrow X$, $X : \Diamond(Y) : true \rightsquigarrow X$ $X : \Diamond_I(Y, I) : true \rightsquigarrow X$

Il nous reste alors à récupérer l'argument de l'opérateur $\wedge$ et à le placer au sommet de la nouvelle question en accord avec $X : \wedge(Y) : \texttt{true} \rightsquigarrow X : Y$

La réécriture se sépare donc en trois phases :

Elimination des éléments inutiles sur le sommet de la pile de la résolvante : le fragment de code qui exécute cette opération est le suivant :

```
DECLARE
      eltres   : elt_res;
      premier  : num_res := rbottom;
      place    : num_res;
      eltobj   : elt_objet;
      dernier  : num_res := position;
BEGIN
--On commence a l´avant dernier element (le dernier est l´element vide)
      place := dernier - 1;
--On elimine tous les elements jusqu´a ce qu´on arrive au sommet de la
--pile de la resolvante ou qu´on arrive sur un ET
      WHILE place /= premier - 1 LOOP
         eltres := recuperer (place);
         eltobj := recuperer (eltres.num_struct);
         EXIT WHEN eltobj.nom_op = et;
         place := place - 1;
      END LOOP;
```

Il faut savoir s'il reste des éléments dans la résolvante. C'est le code suivant qui le teste :

```
      IF place = premier - 1 THEN
--Si la resolvante est vide, la resolution est finie
         RETURN fin_fin;
      END IF;
```

Recopie de la résolvante dans la question : Il faut maintenant recopier la nouvelle question sur la pile de la question. Pour ce faire, on recopie l'intégralité de la résolvante courante, moins le dernier élément, qui doit être remis en forme.

```
--La nouvelle question commence au dessus du sommet de la pile actuelle
      qcurr := position + 1;
      dernier:=place;
      place:=premier;
```

```
WHILE place /= dernier LOOP
   eltres              := recuperer (place);
   eltquest.num_struct := eltres.num_struct;
   eltquest.num_envi   := eltres.num_envi;
   adrquest            := empiler (eltquest);
   place               := place + 1;
END LOOP;
```

Remise en forme du dernier élément de la résolvante : Il faut maintenant réaliser l'opération de remise en forme du dernier élément.

Pour bien comprendre comment fonctionne l'opération, examinons la sur un exemple : Si l'opérateur que nous allons traiter est : `ET(POS(X):NEC(Y):p(X)` nous devons placer dans la résolvante la suite `POS(X)`, `NEC(Y)`, `p(X)`. Ceci est fait de façon générale par le code suivant :

```
eltres := recuperer (dernier);
numobj := recuperer (recuperer (eltres.num_struct).arg_op);
eltquest.num_envi := eltres.num_envi;
LOOP
   eltobj              := recuperer (numobj);
   eltquest.num_struct := numobj;
   adrquest            := empiler (eltquest);
   IF eltobj.genre = predicat THEN
      EXIT;
   ELSE
      numobj := eltobj.obj_qual;
   END IF;
END LOOP;
```

Arrivée à ce stade, la nouvelle question a été réécrite.

11.4 Le problème du contrôle : le KUT

Nous avons évoqué dans le chapitre 5 les différents problèmes de contrôle, ainsi que la nécessité d'implanter un `KUT` pour supprimer certains problèmes de bouclage spécifiques à la résolution modale.

Rappelons rapidement les règles de résolution liées au `KUT` :

1.
$$\frac{KUT(X) : A, ?B \vdash ?KUT(X) : C}{A, ?B \vdash ?C}$$

2.
$$\frac{A, ?KUT(X) : B \vdash ?KUT(X) : C}{A, ?B \vdash ?C}$$

	Fait	Question	Règles
R1	KUT	Pred	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R2	KUT	$\wedge$	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R3	KUT	$\Diamond$	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R4	KUT	$\Diamond_I$	$\frac{KUT(X):A,?B\vdash?KUT(X):C}{A,?B\vdash?C}$
R5	Pred	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R6	$\wedge$	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R7	$\Box$	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R8	$\Diamond_I$	KUT	$\frac{A,?KUT(X):B\vdash?KUT(X):C}{A,?B\vdash?C}$
R9	KUT	KUT	$\frac{KUT(Y):A,?KUT(X):B\vdash?KUT(X):C}{KUT(Y):A,?B\vdash?C}$ et $X \neq Y$

Tableau 11.2: Développement des règles du KUT KUT

3.

$$\frac{KUT(X) : A, ?KUT(Y) : B \vdash ?KUT(Y) : C}{KUT(X) : A, ?B \vdash ?C} \text{ et } X \neq Y$$

En développant exhaustivement le `KUT` par rapport aux autres opérateurs modaux, cela nous donne le classique tableau 11.2. q Il faut inclure une règle de réécriture supplémentaire : $X : KUT(Y) : true \rightsquigarrow X$

11.4.1 Implantation des règles de résolution

Les formes pour l'exécution des règles de résolution (1), (2) et (3) existent déjà, nous n'avons donc aucun code à écrire pour elles.

On constate qu'il est extrêmement facile d'implanter le `KUT` au niveau des règles de résolution. Le code à écrire est extrêmement court, et simple.

11.4.2 Enchaîinement des règles

Comme nous avons ajouté un nouvel opérateur, nous devons définir son comportement par rapport à chacun des autres opérateurs déjà existants. Cela nous donne le code suivant :

```
premieresregles (kut, null_operateur) := 3;
premieresregles (kut, et)             := 3;
premieresregles (kut, pos)            := 3;
premieresregles (kut, nec)            := 3;
premieresregles (kut, posi)           := 3;
```

```
premieresregles (null_operateur, kut) := 2;
premieresregles (et, kut)             := 2;
premieresregles (pos, kut)            := 2;
premieresregles (nec, kut)            := 2;
premieresregles (posi, kut)           := 2;

premieresregles (kut, kut) := 11;
```

D'autre part, la définition de la règle 11 est élémentaire :

```
regles (11).code       := 10;
```

11.4.3 Règle de réécriture

Il n'y a pas une ligne de code à modifier dans ce qui a été décrit dans la section 11.3. La sémantique de la règle de `KUT` est la même que celle des règles pour `POSI`, `POS` ou `NEC`.

11.5 Exemple : le problème des sages et des chapeaux

Cette section montre une application pratique de S1 au problème célèbre (au moins en IA...) des sages et des chapeaux, appelé aussi en anglais "the wise men puzzle", problème auquel nous avons déjà fait référence tout au long de ce travail.

11.5.1 Enoncé du problème

Il était une fois dans un beau royaume, dans une contrée fort lointaine, un roi qui souhaitait savoir lequel de ses trois preux chevaliers, Gauvin, Lancelot et Arthur, était le plus sage, afin de lui donner sa fille Guenièvre en mariage.

Hélas, il se faisait vieux et son esprit, aussi rouillé que son armure, ne parvenait pas à imaginer une épreuve suffisamment subtile pour départager les trois prétendants.

Mais sa fille aimait en secret Arthur. Ne souhaitant pas contrarier son père, mais voulant cependant épouser son bien-aimé, elle imagina un stratagème.

Elle vint voir un jour son père et lui dit: "Mon père je connais une épreuve qui pourra départager mes prétendants. Je vais vous la soumettre. Si vous l'acceptez, je vous demanderai seulement une petite faveur."

Le roi fut enchanté par l'intelligence de sa fille et accepta son idée d'enthousiasme.

"Alors, mon père lui dit-elle, vous poserez l'énigme à Arthur en dernier", ce que le roi accepta.

Il fit donc venir ses conseillers dans la grande salle du palais

Il leur mit à chacun un chapeau sur la tête de telle façon que chacun pouvait voir la couleur du chapeau de son voisin, mais ne pouvait voir la couleur du sien.

Alors le roi s'adressa aux prétendants en ces termes:

"Vous portez tous un chapeau, soit blanc, soit noir ; je puis vous dire qu'il y a dans cette pièce une personne au moins qui a un chapeau blanc."

Puis il s'adressa au premier et lui demanda: "Ton chapeau est-il blanc?"

"Hélas Sire, lui répondit-il, je ne sais"

Alors il s'adressa au deuxième: "Ton chapeau est-il blanc?"

"Hélas Sire, je ne puis répondre"
Enfin, il s'adressa au troisième: "Ton chapeau est-il blanc?"
"Oui Sire, je sais qu'il est blanc"
Et c'est ainsi qu'Arthur épousa Guenièvre[1]

11.5.2 Formalisation dans multi-S4

Nous allons tout d'abord écrire le problème simplement avant de le traduire dans notre langage. En effet, les expressions en logique multi-S4 devront être traduites dans le système S1.

Nous allons tout d'abord exprimer que tout le monde sait que si le chapeau de Gauvin est noir et que le chapeau d'Arthur est noir alors le chapeau de Lancelot est blanc :

```
NEC(_):((noir(gauvin) et noir(arthur)) -> blanc(lancelot) ).
```

`NEC` signifiant toujours "Savoir" `NEC(_)` signifie "tout le monde sait".

Nous devons écrire les deux clauses symétriques de celle-ci :

```
NEC(_):((noir(lancelot) et noir(arthur)) -> blanc(gauvin) ).
NEC(_):((noir(gauvin) et noir(lancelot)) -> blanc(arthur) ).
```

Nous allons maintenant exprimer que *tout le monde sait que s'il est compatible avec les connaissances de Lancelot que le chapeau d'Arthur soit blanc, alors le chapeau d'Arthur est blanc.* Cela se comprend très bien. En effet, lancelot voit le chapeau d'arthur. Donc il a sur sa couleur une certitude totale. Donc s'il est possible pour lui que ce chapeau soit blanc, c'est qu'il est blanc. On note ceci :

```
NEC(_):(POS(lancelot):blanc(arthur) -> NEC(lancelot):blanc(arthur))
```

Il nous faut bien entendu exprimer toutes les clauses similaires à celle-ci. Il y en a onze de plus. Nous ne les donnons pas. Elles sont construites sur le même modèle en développant toutes les combinaisons possibles de (arthur, lancelot, gauvin)[2].

Nous allons voir maintenant comment les réponses des trois sages ont fourni des information à Arthur. La réponse de Gauvin nous a donné une indication supplémentaire :

```
NEC(arthur):NEC(lancelot):POS(gauvin):noir(gauvin)
```

Ceci peut se lire : `Arthur sait que lancelot sait qu´il est possible pour gauvin qu´il ait un chapeau noir`. En effet, Gauvin ayant parlé le premier, Lancelot qui a parlé le second sait qu'il peut avoir un chapeau noir, puisqu'il n'a pas pu répondre, et Arthur sait que Lancelot le sait, puisque Lancelot a parlé avant Arthur[3]

Enfin, la réponse de Lancelot donne une indication supplémentaire :

```
NEC(arthur):POS(lancelot):noir(lancelot)
```

Ceci peut se lire : `Arthur sait qu´il est possible pour lancelot qu´il ait un chapeau noir`. En effet, Lancelot ayant parlé avant Arthur, Arthur sait qu'il n'a pas pu répondre, donc qu'il peut avoir un chapeau noir.

[1] Qui ne tardera pas à le tromper avec Lancelot. Mais je m'égare...

[2] On aurait fort envie d'écrire une seule clause contenant deux variables, de la forme: `NEC(_):(POS(X):blanc(Y) -> NEC(X):blanc(Y), X<>Y)`. Mais nous sommes dans multi-S4, pas en logique modale d'ordre 1.

[3] Je conseille au lecteur de respirer à fond et de relire lentement la phrase précédente...

11.5.3 Traduction en langage TARSKI

La traduction des clauses précédentes dans le langage TARSKI est relativement simple, à condition de ne pas oublier de skolémiser les opérateurs `NEC` et `POS` quand cela est nécessaire, et de passer l'ensemble du problème en syntaxe préfixée.

Le premier groupe de trois clauses devient :

```
#NEC(_):#ET(noir(arthur)):#ET(noir(gauvin)):blanc(lancelot).
#NEC(_):#ET(noir(arthur)):#ET(noir(lancelot)):blanc(gauvin).
#NEC(_):#ET(noir(lancelot)):#ET(noir(gauvin)):blanc(arthur).
```

Le second groupe de douze clauses :

```
#NEC(_):#ET(#POS(lancelot):blanc(arthur)):#NEC(lancelot):blanc(arthur).
#NEC(_):#ET(#POS(gauvin):blanc(arthur)):#NEC(gauvin):blanc(arthur).
#NEC(_):#ET(#POS(gauvin):noir(lancelot)):#NEC(gauvin):noir(lancelot).
#NEC(_):#ET(#POS(gauvin):blanc(lancelot)):#NEC(gauvin):blanc(lancelot).
#NEC(_):#ET(#POS(gauvin):noir(arthur)):#NEC(gauvin):noir(arthur).
#NEC(_):#ET(#POS(lancelot):noir(arthur)):#NEC(lancelot):noir(arthur).
#NEC(_):#ET(#POS(lancelot):blanc(gauvin)):#NEC(lancelot):blanc(gauvin).
#NEC(_):#ET(#POS(lancelot):noir(gauvin)):#NEC(lancelot):noir(gauvin).
#NEC(_):#ET(#POS(arthur):blanc(gauvin)):#NEC(arthur):blanc(gauvin).
#NEC(_):#ET(#POS(arthur):noir(lancelot)):#NEC(arthur):noir(lancelot).
#NEC(_):#ET(#POS(arthur):blanc(lancelot)):#NEC(arthur):blanc(lancelot).
#NEC(_):#ET(#POS(arthur):noir(gauvin)):#NEC(arthur):noir(gauvin).
```

Enfin les dernières informations apportées par les réponses des deux premiers prétendants sont :

```
#NEC(arthur):#NEC(lancelot):#POSI(gauvin,0):noir(gauvin).
#NEC(arthur):#POSI(lancelot,1):noir(lancelot).
```

Nous avons maintenant codé l'ensemble du problème. Le système TARSKI est capable de le résoudre dans un temps raisonnable (moins de 1'45 en configuration parallèle sur deux HP-720). Les temps de résolution de ce problème ont déjà été discutés dans les chapitres précédents.

11.6 Influence de l'ordre des règles

Nous avons vu que lorsque le moteur d'inférences atteint un point de choix multiples au niveau des règles, il peut paralléliser l'exécution si la machine parallèle est installée.

Mais si l'on n'utilise que la machine séquentielle, les règles sont exécutées dans l'ordre dans lequel elles sont chaîinées par `regles`. Il m'a donc paru intéressant de voir si l'ordre de chaîinage des règles avait une influence sur le temps d'exécution du programme. Il était bien sûr impossible de tester toutes les configurations, mais, sur l'exemple classique du "wise men puzzle" simplifié nous avons testé l'ensemble des configurations décrites dans le tableau 11.3. Les chiffres dans les trois premières colonnes indiquent dans quel ordre est exécuté chacune des formes permettant de résoudre le cas des opérateurs considérés.

On voit clairement que l'ordre des règles a une grande importance, le temps d'exécution pouvant varier du simple à plus du triple.

NEC,POS	NEC,POSI	POSI,POS	Temps
8,11,5,4	8,5	9,4	27.81
8,11,5,4	8,5	4,9	28.18
8,11,5,4	5,8	9,4	28.33
8,11,4,5	8,5	9,4	28.36
8,4,11,5	8,5	9,4	28.37
11,8,5,4	8,5	9,4	45.39
5,11,4,8	8,5	9,4	71.85
5,4,8,11	5,8	4,9	77.00
4,5,11,8	8,5	9,4	91.27
4,5,11,8	5,8	4,9	91.37

Tableau 11.3: Influence de l'ordre des règles

On constate que lorsque les formes les plus contraignantes sont exécutées d'abord la résolution est plus rapide, alors que l'exécution du code associé est plus long (il faut exécuter une unification dans la forme 8, alors que la forme 4 est simplement une incrémentation de variable).

La principale raison que nous voyons est que les formes 8 ou 9 peuvent échouer et "coupent" alors l'arbre de résolution en provoquant un backtrack, alors que les formes 4 et 5 ne peuvent jamais échouer.

11.7 Logique des modules

11.7.1 Règles

Les règles de résolution de la logique des modules sont :

$$\frac{M(X) : C, ?M(X) : G \vdash ?M(X) : NG}{C, ?G \vdash ?NG}$$

$$\frac{C, ?M(X) : G \vdash ?M(X) : NG}{C, ?G \vdash ?NG}$$

Si nos faisons un développement exhaustif, nous obtenons le tableau 11.4.

La règle de réécriture supplémentaire est :

$$Y : M(X) : true \rightsquigarrow Y : true$$

11.7.2 Implantation des règles de résolution

L'implantation des règles de résolution est immédiate. En effet, les formes utilisables sont représentées dans le tableau 11.5. Les formes portent les numéros standards que nous leur avons donné dans le chapitre précédent.

Ainsi **l'implantation des règles de résolution ne demande l'écriture d'aucun code spécifique**.

Fait	*Question*	*Règles*
Pred	Pred	$p, ?p \vdash ?true$
Pred	$\wedge$	$\frac{A,?\wedge(X):B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
Pred	M	$\frac{A,?M(Y):B\vdash ?M(Y):C}{A,?B\vdash ?C}$
$\wedge$	Pred	$\frac{\wedge(X):A,?B\vdash ?\wedge(X):C}{A,?B\vdash ?C}$
$\wedge$	$\wedge$	$\frac{\wedge(X):A,?\wedge(Y):B\vdash ?\wedge(Y):C}{\wedge(X):A,?B\vdash ?C}$
$\wedge$	M	$\frac{\wedge(X):A,?M(Y):B\vdash ?\wedge(X):C}{A,?M(Y):B\vdash ?C}$
M	Pred	
M	$\wedge$	$\frac{M(Y):A,?\wedge(X):B\vdash ?\wedge(X):C}{M(Y):A,?B\vdash ?C}$
M	M	$\frac{M(X):A,?M(X):B\vdash ?M(X):C}{A,?B\vdash ?C}$

Tableau 11.4: Logique des modules

Fait	*Question*	*Règles*
Pred	Pred	Forme 1
Pred	$\wedge$	Forme 2
Pred	M	Forme 2
$\wedge$	Pred	Forme 3
$\wedge$	$\wedge$	Forme 2
$\wedge$	M	Forme 3
M	Pred	
M	$\wedge$	Forme 2
M	M	Forme 6

Tableau 11.5: Formes pour la logique des modules

11.7.3 Le chaîinage des règles de résolution

Le chaîinage des règles de résolution ne présente aucun intérêt particulier. Aucun couple (Fait, Question) ne peut donner lieu à une parallélisation et la résolution est déterministe. Il est inutile de revenir dessus

11.7.4 Règle de réécriture

Si le code de S1 est disponible, nous n'avons rien à modifier, car la forme de la règle de réécriture de la logique des modules est exactement celle des règles de S1.

11.7.5 Conclusion

L'implantation de la logique des modules décrites ici est un travail totalement trivial qui n'a pris qu'une vingtaine de minutes.

11.8 Logique floue

La logique floue présente l'intérêt d'avoir une règle de réécriture "non-standard" par rapport à toutes celles que nous avons vues jusqu'ici.

La règle de réécriture de la logique floue est :

$$FUZ(X_1)\ldots FUZ(X_n) : A \rightsquigarrow A \text{ et afficher}(\min(X_i))$$

Les règles de résolution sont en revanche extrêmement simples :

$$\frac{FUZ(X) : A, ?B \vdash ?FUZ(X) : C}{A, ?B \vdash ?C}$$

$$\frac{A, ?FUZ(X) : B \vdash ?FUZ(X) : C}{A, ?B \vdash ?C}$$

Le classique développement exhaustif nous donne le tableau 11.6.

Remarquons simplement que, comme le `KUT`, le FUZ se place toujours en tête de la résolvante. **Ceci nous garantit qu'à la fin de la résolution, l'ensemble des opérateurs FUZ sera en tête de la résolvante.**

Les formes nécessaires à l'implantation du FUZ sont déjà disponibles (les formes 2 et 3 suffisent). Nous ne nous attarderons par sur le problème de l'implantation des règles de résolution, qui est élémentaire.

11.8.1 La règle de réécriture

Rappelons la forme particulière de la règle de réécriture :

$$FUZ(X_1)\ldots FUZ(X_n) : A \rightsquigarrow A \text{ et afficher}(\min(X_i))$$

Ceci va nous obliger à modifier le code que nous avons jusque là utilisé. Cependant, comme nous allons le voir, les modifications sont tout à fait mineures.

En fait sur les trois parties de la réécriture décrites dans le chapitre précédent, section 11.3, seule la première partie (élimination des éléments inutiles sur le sommet de la pile de la résolvante) doit être légèrement remaniée.

Voici le nouveau code :

Fait	*Question*	*Règles*
Pred	Pred	$p, ?p \vdash ?true$
Pred	$\wedge$	$\frac{A,?\wedge(X):B\vdash?\wedge(X):C}{A,?B\vdash?C}$
Pred	FUZ	$\frac{A,?FUZ(Y):B\vdash?FUZ(Y):C}{A,?B\vdash?C}$
$\wedge$	Pred	$\frac{\wedge(X):A,?B\vdash?\wedge(X):C}{A,?B\vdash?C}$
$\wedge$	$\wedge$	$\frac{\wedge(X):A,?\wedge(Y):B\vdash?\wedge(Y):C}{\wedge(X):A,?B\vdash?C}$
$\wedge$	FUZ	$\frac{\wedge(X):A,?FUZ(Y):B\vdash?FUZ(Y):C}{\wedge(X):A,?B\vdash?C}$
FUZ	Pred	$\frac{FUZ(Y):A,?B\vdash?FUZ(Y):C}{A,?B\vdash?C}$
FUZ	$\wedge$	$\frac{FUZ(Y):A,?\wedge(X):B\vdash?FUZ(Y):C}{A,?\wedge(X):B\vdash?C}$
FUZ	FUZ	$\frac{FUZ(X):A,?FUZ(Y):B\vdash?FUZ(X):C}{A,?FUZ(Y):B\vdash?C}$

Tableau 11.6: Logique floue

```
      rbottom := position + 1;
      eltres  := recuperer (dernier);
      IF eltres.num_struct /= null_objet THEN
         RAISE erreur_reecriture;
      END IF;

      place := dernier - 1;
--Ajout logique floue
--Initialise la variable qui contiendra la valeur minimale.
      mini:=1.0;
--Fin ajout

      WHILE place /= premier - 1 LOOP
         eltres := recuperer (place);
         eltobj := recuperer (eltres.num_struct);
         IF eltobj.genre /= operateur THEN
            RAISE erreur_reecriture;
         END IF;
         EXIT WHEN eltobj.nom_op = et;
--Ajout logique floue
--Si l'operateur elimine est un operateur FUZ, alors prendre
--le minimum de l'argument de l'operateur et de la variable mini
         IF eltobj.nom_op = fuz THEN
    numobj   := recuperer(eltobj.arg_op);
    eltobj   := recuperer(numobj);
    IF mini>eltobj.val_flot THEN mini:=eltobj.val_flot; END IF;
 END IF;
```

```
--Fin ajout
         place := place - 1;
      END LOOP;
```

On voit que l'apport de code se réduit 6 lignes. Le problème est en effet trivial : il suffit de prendre le minimum sur la suite des opérateurs FUZ, tous placés en tête de la résolvante.

Il faut également modifier le code vérifiant la terminaison de la résolution :

```
      IF place = premier - 1 THEN
--Ajout logique floue
 afficher(mini);
--Fin ajout
         RETURN fin_fin;
      END IF;
```

On ajoute simplement la ligne de code qui demande l'affichage de la variable contenant le minimum.

Partie VI

Conclusion

Conclusion

Suivant l'expression de Martti Penttonen, la réalisation d'extensions de PROLOG doit tenter de trouver un compromis entre deux objectifs contradictoires :

- d'une part, il faut améliorer l'expressivité du langage PROLOG ;
- d'autre part, il faut tenter de ne pas diminuer l'efficacité de PROLOG, qui est déjà à la limite de l'acceptable.

Face à ces deux objectifs, nous avons tenté de dégager une voie moyenne. Certes, notre système est doublement non-déterministe : a coté du classique choix sur les clauses, nous introduisons un choix sur les règles de résolution. Mais nous avons restreint les possibilités d'extensions en n'acceptant que certaines formes de règles de résolution : ainsi nous pouvons garantir un traitement efficace. D'autre part, nous avons montré que l'utilisation du parallélisme spécifique pouvait considérablement améliorer les performances. Enfin, certains systèmes logiques, comme la logique floue ou la logique des modules, ont un jeu de règles TARSKI totalement déterministes et la résolution pour ces logiques est aussi rapide que la résolution en logique classique.

Il reste cependant de nombreux problèmes à résoudre avant que TARSKI puisse être considéré comme un langage accessible au grand public. Ces problèmes constituent autant de directions de recherches futures :

Sur un plan conceptuel :

Il nous faut tenter de développer un mécanisme permettant de traduire automatiquement les règles logiques dans le code de la machine abstraite. Dans l'état courant de l'implantation, chaque nouvelle implantation de logiques demande l'écriture de code, une opération simple pour qui connaîit bien le système, mais complexe pour un utilisateur novice extérieur. Il est clair que cette étape de traduction automatique est difficile. Il faudrait sans doute développer un langage de spécification intermédiaire qui agirait comme un médiateur entre la forme logique et le code de la machine abstraite.

Sur un plan pratique : trois directions privilégiées sont à retenir :

- Il faut développer une bibliothèque de composants implantant plusieurs logiques.
- Il faut implanter le système dans un langage permettant sa diffusion au niveau universitaire, donc le réécrire en C.
- Il faut réaliser une implantation sur une véritable machine parallèle à mémoire partagée. Cela permettra d'améliorer de façon conséquente l'efficacité du langage.

Dans l'état actuel, TARSKI est un outil utile pour tester et expérimenter diverses extensions logiques. Nous pensons également avoir montré qu'il est possible de construire un langage suffisamment général pour traiter de nombreuses extensions de PROLOG, et suffisamment efficace pour que son utilisation soit acceptable.

Partie VII

Annexes

Annexe A

Détails d'implantation

A.1 Les piles génériques étendues

A.1.1 Paramètres de généricité

Une pile générique aura trois paramètres de généricité :

`TYPE objet IS PRIVATE;` Le type privé `objet` est le type que stockera la pile. Chaque élément de pile sera un élément de type `objet`.

`TYPE indice IS RANGE <>;` Le type `indice` est le type qui servira d'indice à la pile. Nous avons choisi un type entier plutôt qu'un type discret pour pouvoir utiliser les opérations $+$, $-$, etc...

`taille : IN indice;` `taille` sera la taille demandée par l'utilisateur pour la pile *initialement*.

A.1.2 Implantation

La seule originalité de l'implantation est la possibilité d'étendre dynamiquement les piles.

Une pile est implantée comme un tableau qui a au départ la taille demandée à l'instanciation. Si, au cours de l'exécution d'une instruction, il faut accéder à une adresse se situant à un indice supérieur à la taille du tableau, on réserve un tableau deux fois plus grand, on recopie le tableau initial à l'intérieur du nouveau tableau et on poursuit l'exécution de l'instruction sur le nouveau tableau[1].

Cette technique fonctionne parfaitement et permet à TARSKI de résister à la résolution de `fib(20,X)` (calcul récursif du vingtième nombre de fibonacci) alors qu'un grand nombre de PROLOG déclare forfait.

De plus cette forme d'implantation nous garantit que les références des objets dans la pile sont absolues et resteront constantes d'un processeur à un autre puisqu'il s'agit d'indice de tableaux et non pas de pointeurs (qui sont des adresses machine, donc variables d'un processeur à un autre). L'inconvénient est bien entendu la lenteur de la technique, puisque tout accès à un objet de la pile demande un appel de fonction et un chargement mémoire indirect.

[1] Cette technique n'est pas originale. Elle est utilisée en particulier par SWI-Prolog, mais elle est implantée au niveau système ce qui provoque parfois des "plantages" inattendus de l'application. Notons que cette possibilité n'est en fait quasiment jamais utilisé lors d'une exécution car les piles sont très largement dimensionnées au démarrage de l'application.

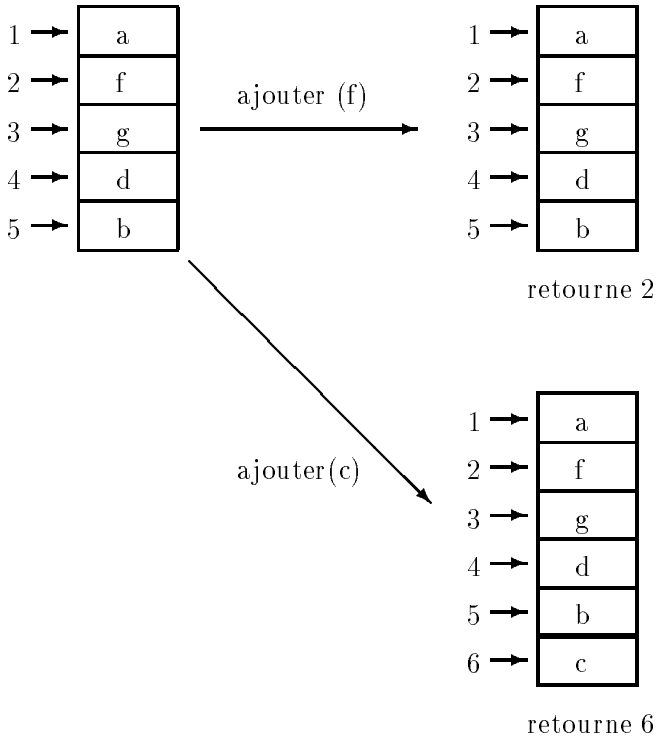


Figure A.1: Piles sans éléments dupliqués

A.2 Les tables génériques

L'implantation des tables est un peu plus complexe que l'on ne pourrait le croire. Au lieu de tester l'existence d'objets identiques, on teste l'existence d'objets ayant des clefs identiques, et on utilise une fonction annexe pour savoir lequel des deux objets ayant la même clef il faut conserver. Le principe est détaillé sur la figure A.1

Quel est l'intérêt de ce mécanisme? Il est purement technique ; les tables servent principalement à stocker les prédicats. Mais, en fait, dans notre implantation, un prédicat est un record contenant le nom du prédicat, mais aussi la clause associée à ce prédicat. Ce que nous souhaitons conserver dans la table, c'est le record contenant un pointeur sur la première clause de la base associée à ce prédicat (première clause sélectionnée pendant la résolution). Avec cette méthode, l'accès aux clauses associées à un prédicat est immédiat.

A.2.1 Paramètres de généricité

`TYPE objet IS PRIVATE;` Le type privé `objet` est le type que stockera la pile.

`TYPE indice IS RANGE <>;` Le type `indice` est le type qui servira d'indice à la pile.

`taille : IN indice; taille` sera la taille demandée pour la pile.

`TYPE hash_value IS (<>);` Le type `hash_value` est la valeur de hachage qui sera utilisé pour retrouver plus facilement les éléments susceptibles d'être égaux.

`WITH FUNCTION hash (x : IN objet) RETURN hash_value IS <>;` La fonction `hash` prend un objet en paramètre et retourne sa valeur de hachage. Elle est générique, puisque l'objet étant privé, elle ne peut être calculée que de façon externe.

`WITH FUNCTION egal (x, y : IN objet) RETURN boolean;` La fonction `egal` prend deux objets en paramètres et retourne vrai s'ils sont égaux et faux s'ils sont différents. On peut se demander pourquoi utiliser une fonction générique, puisque les types étant privés, il est possible de les comparer. En fait, nous ne devrions pas parler d'égalité, mais d'égalité pour une clef. Deux éléments seront "égaux" quand la fonction `egal` dira qu'ils le sont, même si en réalité ils sont différents. Dans un cas simple la fonction `egal` sera effectivement $=$, mais pas dans tous les cas.

`WITH FUNCTION keep (x, y : IN objet) RETURN objet;` La fonction `keep` prend deux objets en paramètre et en retourne un troisième. En pratique, cette fonction est appelée quand deux objets x (déjà dans la pile) et y (que l'on veut ajouter dans la pile) ont été déclarés égaux par `egal`.

L'objet que l'on conserve dans la pile n'est pas x ou y, mais l'objet retourné par `keep` qui remplace x dans la pile. Dans un cas simple, la fonction `keep` retournerait par exemple x ou y, mais pas dans tous les cas.

A.2.2 Entités exportées

A.2.2.1 Fonctions et procédures

Certaines des fonctions fournies par le paquetage sont les mêmes que celles fournies par le paquetage standard de pile ; certaines en revanche sont spécifiques.

`FUNCTION recuperer (i : IN indice) RETURN objet;` La fonction `recuperer` prend une adresse d'objet en argument et retourne cet objet.

`PROCEDURE revenir (i : IN indice);` La procédure `revenir` prend en argument une adresse i et place le pointeur du sommet de pile à cette adresse.

`FUNCTION position RETURN indice;` la fonction `position` retourne l'adresse du sommet de la pile.

`FUNCTION ajouter (x : IN objet) RETURN indice;` La fonction `ajouter` ajoute un élément dans la pile en s'assurant qu'il n'existe pas déjà dans la pile un élément identique.

A.2.2.2 Constantes exportées

Une seule constante exportée :`element_null`, comme pour les piles génériques. Les mêmes remarques s'appliquent.

A.2.2.3 Exceptions exportées

Une seule exception exportée : `indice_trop_grand : EXCEPTION;`. Cette exception est levée quand l'utilisateur tente d'accéder à un objet se situant au delà du sommet de pile.

L'exception `pile_vide` n'a pas de raison d'être puisque l'utilisateur ne peut pas dépiler d'objets.

Annexe B

Le partage de structures

Dans cette annexe, nous allons détailler le fonctionnement du mécanisme de partage de structures, présenter des exemples de son utilisation et expliquer en détail son implantation.

B.1 principes généraux

Le but du partage de structures est de partager les structures du programme.
Une instance d'une structure sera représentée par un couple de pointeurs `(Struct, Env)`.

- Struct pointe sur la représentation en mémoire de la structure en question.
- Env pointe sur un environnement donnant la valeur des variables de la structure.

Exemple : Soit la structure `cons(X,Y)` , l'instance `cons(a,nil)` de cette structure sera représentée par le couple de pointeurs suivant:

Structure : `cons(X,Y)`

Environnement : `X=a, Y=nil`

On voit bien que ces deux informations (structure et environnement) suffisent à définir le terme `cons(a,nil)`. Par homogénéité, tout terme sera représenté par un couple `(Struct,Env)` même les constantes pour lesquelles l'environnement n'a pas d'utilité puisque les constantes ne contiennent pas de variable (par définition).

Pour repérer sans ambiguîiité une structure dans un programme dans les exemples qui suivent, nous lui associerons un numéro unique .Par exemple, nous écrirons : `append1(nil2,Y,Y)`.

B.1.1 Représentation d'un environnement

Un environnement est alloué à chaque utilisation d'une clause. Cet environnement permet de creer une nouvelle instance de chacune des variables de la clause.

La valeur d'une variable est un couple `(Struct,Env)`. Un environnement pour une clause C est composé d'autant de couples `(Struct,Env)` qu'il y a de variables dans la clause C. On voit ici l'intérêt d'associer dans le dictionnaire des clauses à chaque clause son nombre de variables.

Ces environnements sont créés sur une pile spéciale appelée la pile des environnements. Nous notons le pointeur sur le sommet de cette pile V. Ce pointeur est sauvé dans la pile des

Figure B.1: Environnement de la question

points de choix à chaque succès dans une résolution (aussi bien succès sur le choix d'une clause que sur le choix d'une règle), ce qui permet de récuperer l'espace alloué au backtrack.

Par exemple, un environnement pour la clause : `append(nil2,Y,Y)` sera représenté par un seul couple `(Struct,Env)` (pour la variable Y). En revanche, un environnement pour la clause : `#ET(append(X,Y,Z)):append(cons4(U,X),Y,cons5(U,Z))`. sera representé par 4 couples `(Struct,Env)` (pour les variables U, X, Y, Z).

Les environnements permettent de représenter les unifications des variables d'une clause. La valeur d'une variable est un couple de pointeur : `(Struct,Env)` représentant une instance d'une structure du programme.

B.1.2 Notation

Pour simplifier l'écriture des couples, nous remplacerons le pointeur sur la structure par le nombre associé à cette structure et le pointeur sur l'environnement par le nom de l'environnement `(e0,e1,e2,...)`. Une instance de terme sera donc représentée par : `i/ej` où i est le numéro associé à la structure utilisée et `ej` celui de l'environnement.

Pour représenter une variable V d'un environnement ei, nous l'écrirons : `V/ei`

Lors des unifications, les deux termes a unifier seront représentés: `terme1/ei1` et `terme2/ei2` `ei1` et `ei2` étant les environnements respectifs dans lesquels les termes 1 et 2 sont décrits.

B.1.3 Exemple d'utilisation du partage de structures

Nous allons effectuer une résolution (élémentaire) complète pour montrer comment fonctionne le partage de structures.

1. Soit la question[1] :

   ```
   append(cons8(a9,cons10(b11,nil12)),cons13(c14,(cons15(d16,nil17)),R)
   ```

   La question contient une variable R, nous allons donc créer pour cette question un environnement ne contenant qu'une case pour R (Cette case représentera une nouvelle instance de R). Cet environnement nous l'appellerons `e0` et c'est l'environnement où nous irons chercher la valeur du résultat (ici la valeur de la variable R, voir figure B.1).

2. Pour résoudre cette question, nous allons utiliser l'ensemble de clauses vu au dessus :

[1] Rappelons que même si chaque nil est affecté d'un numéro différent, ils représentent tous le même objet nil. En fait, le numéro que nous associons à une structure correspond à son adresse dans la pile des objets. Toutes les structures sont construites les unes après les autres et ont donc des adresses différentes. En revanche, deux prédicats portant le même nom sont sémantiquement identiques.

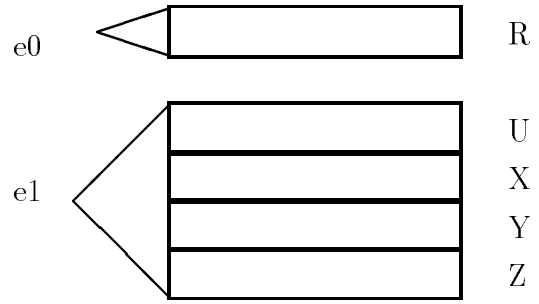


Figure B.2: Environnement de la question et de la première clause

```
#ET(append(X,Y,Z)):append(cons4(U,X),Y,cons5(U,Z)).
append(nil2,Y,Y).
```

et appliquer la première clause. Cette première clause contient 4 variables U, X, Y, Z nous allons donc créer un environnement e1 à 4 cases (Créant ainsi une nouvelle instance de chacune des variables $(U, X, YetZ)$, voir figure B.2).

3. Nous allons unifier le `cons8(a9,cons10(b11,nil12))/e0` de la question avec le `cons4(U,X)/e1` de la clause utilisée.

   Cons contre cons, l'unification réussit, unifions `a9/e0` de la question avec la variable `U/e1` du FAIT (variable U de l'environnement `e1`).

   `U/e1` étant une variable libre (non encore unifiée) nous allons l'unifier à `a9` ceci sera représenté en mettant dans la variable `U` de l'environnement courant `e1` le couple `(Struct,Env)` représentant le terme `a9/e0`, c'est à dire : `9/e0` car `a9` est repéré par le nombre `9`[2].

   Donc `U/e1` unifié au terme `a9/e0` s'écrit en mettant `9/e0` dans la variable `U` de l'environnement `e1`, `9` étant l'adresse de la structure, et `e0` l'adresse de l'environnement. Ceci nous donne l'état représenté sur la figure B.3.

4. Nous continuons notre unification. Il nous faut unifier le terme `cons10(b11,nil12)/e0` avec `X/e1` (variable `X` de l'environnement `e1`). `cons10(b11,nil12)/e0` sera comme ci-dessus désigné par `10/e0`.

   La variable `X/e1` est libre , donc unifier `X/e1` avec `10/e0` revient à mettre le couple `10/e0` dans la case `X` de l'environnement `e1`. Ceci nous donne l'état représenté sur la figure B.4.

   Nous avons donc unifié le premier argument du append.

5. Maintenant unifions `Y/e1` et `cons13(c14,cons15(d16,nil17))/e0`. Nous mettons dans la case `Y` de l'environnement `e1` le couple `13/e0` car la variable Y/e1 est libre (non encore unifiée). Ceci nous donne l'état représenté sur la figure B.5.

[2] a9 étant une constante, l'environnement n'est d'aucune utilité mais on le met pour être homogène dans les données manipulées.

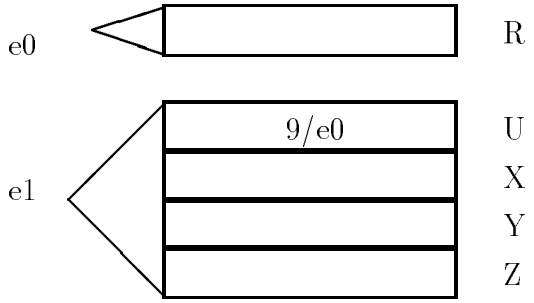


Figure B.3: Environnement de la question et de la première clause

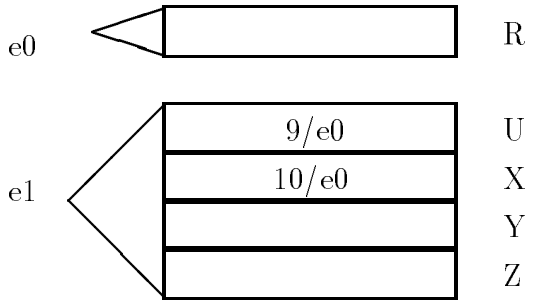


Figure B.4: Environnement de la question et de la première clause

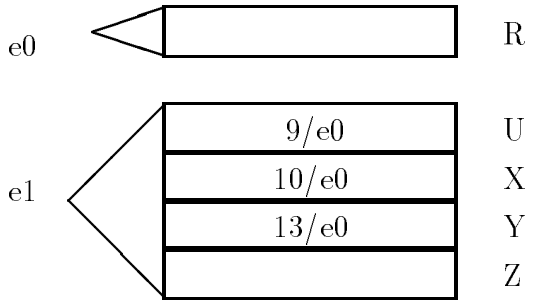


Figure B.5: Environnement de la question et de la première clause

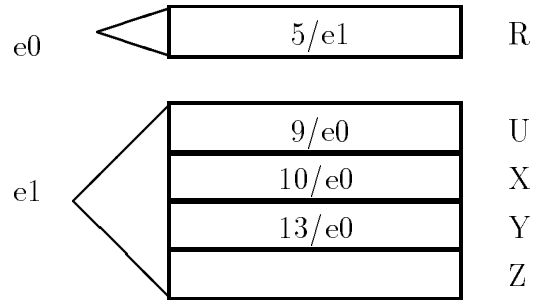


Figure B.6: Environnement de la question et de la première clause

6. Passons au 3eme argument. Nous devons unifier `R/e0` de la question avec le `cons5(U,Z)/e1` du FAIT. Nous avons un terme et une variable libre, nous allons donc mettre le terme dans la variable libre. Ceci nous donne l'état représenté sur la figure B.6.

7. Nous avons unifié la tête du FAIT avec la question. Après avoir reformé la question, nous devons maintenant résoudre `append(X,Y,Z)/e1` (`append(X,Y,Z)` dans l'environnement e1).

   Nous voyons ici l'importance de définir un terme dans un environnement car `append(X,Y,Z)` aurait une toute autre valeur dans un autre environnement où les valeurs de `X`, `Y` et `Z` seraient différentes.

   La variable `X` dans l'environnement `e1` a pour valeur le terme `10/e0` c'est à dire le terme `cons10(b11,nil12)/e0`. Nous sélectionnons la première clause. Cette clause contient 4 variables `(U X Y Z)` nous allons donc créer (empiler) un environnement à 4 cases. Ceci nous donne l'état représenté sur la figure B.7.

   Grâce à l'environnement e2, nous avons créé de nouvelles instances des variables `U, X, Y, Z`. Ces nouvelles instances sont désignées par : `U/e2 X/e2 Y/e2 Z/e2`.

8. Nous allons maintenant unifier le terme `append(X,Y,Z)/e1` avec le terme `append(cons4(U,X), Y, cons5(U,Z))/e2` (qui est la tête de la clause sélectionnée). Unifier `X/e1` et `cons4(U,X)/e2` revient à unifier `10/e0` avec `cons4(U,X)/e2` puisque `10/e0` est la valeur de `X/e1` (valeur de `X` dans l'environnement `e1`). Cette opération, qui consiste à récupérer la valeur de la variable, plutôt que d'unifier à la variable elle-même s'appelle *déréférencement.*

   `10/e0` représente le terme `cons10(b11,nil12)/e0` que nous unifions à `cons4(U,X)/e2`. Ceci nous donne l'état représenté sur la figure B.8.

9. Unifions `Y/e1` avec `Y/e2`, nous obtenons `Y/e2` reçoit `Y/e1` car `Y/e2` est une variable libre. Comme Y/e1 représente `13/e0` (déréférencement), Ceci nous donne l'état représenté sur la figure B.9.

10. Unifions `Z/e2` avec `cons5(U,Z)/e2`. `Z/e1` est une variable libre, Ceci nous donne l'état représenté sur la figure B.10. L'unification a réussit, nous continuons la résolution après avoir reforme" la question avec `append(X,Y,Z)` dans l'environnement `e2`.

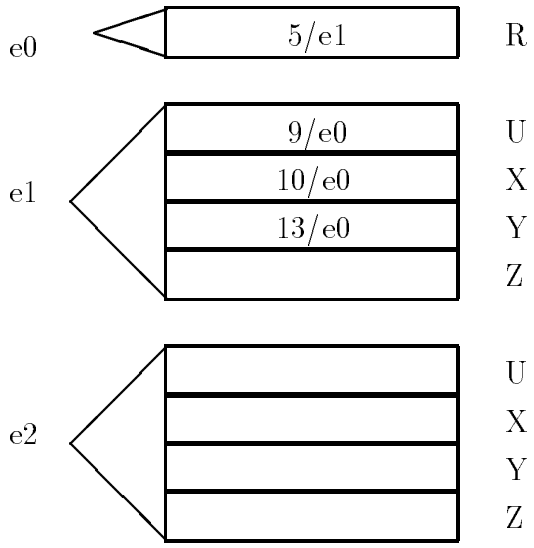


Figure B.7: Environnement de la question et des deux clauses

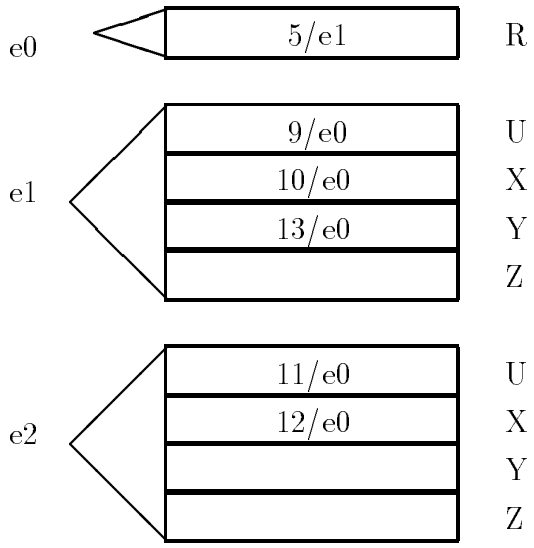


Figure B.8: Environnement de la question et des deux clauses

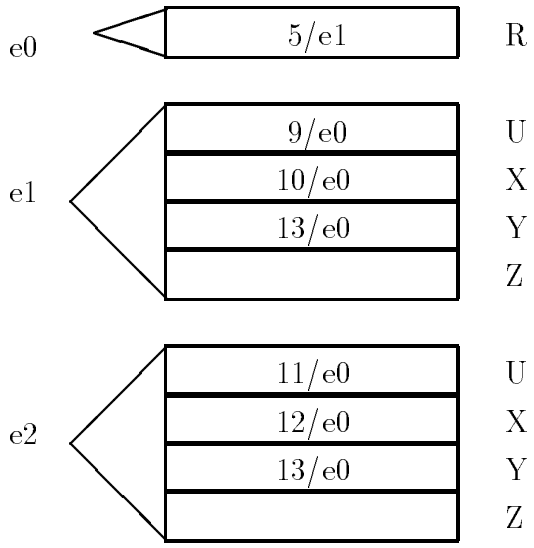


Figure B.9: Environnement de la question et des deux clauses

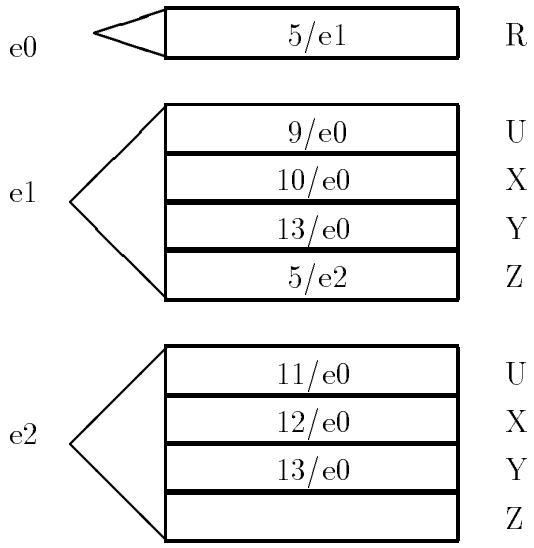


Figure B.10: Environnement de la question et des deux clauses

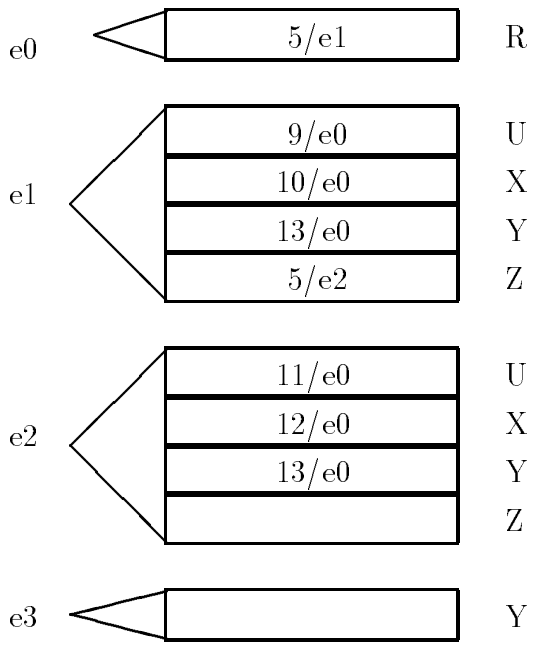


Figure B.11: Environnement de la question et des deux clauses

11. La nouvelle question est donc : `append(X,Y,Z)/e2`. `X/e2` est égal à nil nous allons utiliser la seconde clause du programme. Cette seconde clause ne contient qu'une variable nous empilons donc un environnement e3 de seulement un élément (pour la variable Y de cette clause). Ceci nous donne l'état représenté sur la figure B.11.

12. Essayons d'unifier `append(X,Y,Z)/e2` avec `append(nil2,Y,Y)/e3`. L'unification `X/e2` avec `nil2/e3` réussi car `X/e2` a pour valeur `nil12/e0` (et deux constantes sont unifiables si elle ont la même structure et les mêmes noms).

    Unifions `Y/e2` avec `Y/e3`. `Y/e3` étant une variable libre, nous l'associons à `13/e0`.

    Il nous reste à unifier `Z/e2` avec `Y/e3`. `Z/e2` est une variable libre. Nous déréférençons `Y/e3`, et nous trouvons comme valeur déréférencée `13/e0`. Ceci nous donne l'état représenté sur la figure B.12.

    L'unification a réussi, il n'y a plus de but en attente, nous avons donc achevé la résolution, le résultat se trouve dans les variables de l'environnement e0 (ici R). Or nous avons :

```
R/e0= 5/e1
5/e1 = cons5(U,Z)/e1
U/e1 = 9/e0 = a
Z/e1 = 5/e2 = cons5(U,Z)/e2
U/e2 = 11/e0 = b
Z/e2 = 13/e0 = cons(c,cons(d,nil))
```

    donc

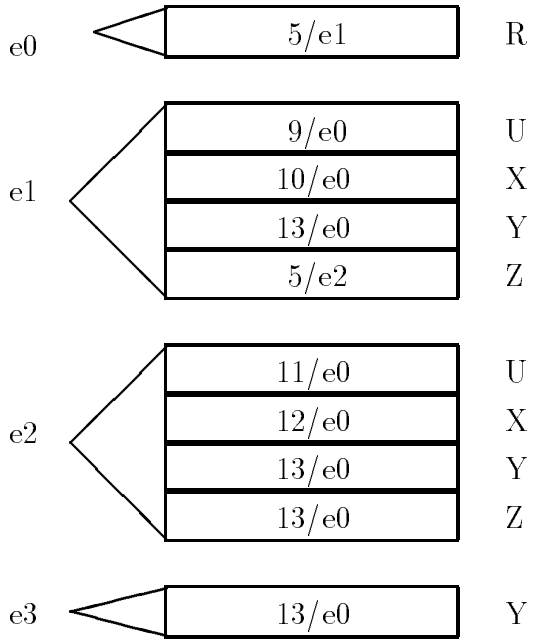


Figure B.12: Environnement de la question et des deux clauses

```
Z/e1 = cons(b, cons(c,cons(d,nil)))
R/e0 = cons(a, cons(b, cons(c,cons(d,nil))))
```

B.2 Backtrack et pile de trainée

Lorsqu'il y a backtrack en PROLOG, c'est que l'unification a échouée avec le FAIT courant,il faut donc reconstruire l'environnement tel qu'il était avant le point de choix.

Supposons que nous soyons à l'étape représentée sur la figure B.3. `U/e1` vient d'être affecté à `9/e0` mais supposons que sur l'unification suivante il y ait un échec. Dans ce cas, nous dépilons l'environnement courant (ici `e1`) et nous retrouvons la pile d'environnement telle qu'elle était avant l'utilisation du FAIT qui a provoqué l'échec.

Si nous nous plaçons à l'étape du dessin B.4, `X/e1` vient d'être affecté à `10/e0`. Si nous supposons maintenant que la résolution échoue, il suffit la aussi de dépiler l'environnement courant `e1` pour obtenir une pile comme elle l'était avant l'utilisation du FAIT courant. Il en va de même pour l'étape de la figure B.5.

Mais regardons maintenant l'étape de la figure B.6. Nous avons unifié `R/e0` et `cons5(U,Z)/e1` et pour cela, nous avons donné à la variable `R/e0` la valeur `5/e1`. Supposons que la résolution échoue. Il nous faut backtracker et donc dépiler l'environnement courant, mais si nous le faisons sans précaution, nous obtiendrons la pile représentée sur la figure B.13. `R` garde la valeur que lui avait donnée l'unification avec le FAIT qui a provoqué l'échec. Nous ne retrouvons donc pas la pile des environnements telle qu'elle était avant l'utilisation du FAIT. Pour retrouver une pile telle qu'on le souhaite, il faudrait réinitialiser la variable `R/e0`.

Pour ce faire, nous devons utiliser une pile annexe, la pile de trainée ou pile de trail, ou encore pile RESET.

Figure B.13: Environnement après backtrack

A chaque nouvelle résolution réussie on sauve le pointeur sur cette pile dans la pile des points de choix, de façon à pouvoir au backtrack revenir à l'état précédent.

Lorsque deux termes sont unifiés, on aboutit en général (si les termes ne sont pas des constantes) à la modification d'un certain nombre de variables. Parmi ces variables, celles qui seront mises dans la pile de trainée sont celles qui n'appartiennent pas à l'environnement courant `ei`, et donc celles qui appartiennent à un environnement `ej` antérieur : $j < i$.

Par exemple, dans l'étape de la figure B.6, la variable modifiée est la variable `R/e0` qui n'appartient pas à l'environnement courant `e1`. `R/e0` va donc être ajoutée sur la pile de trainée. Ainsi, lors d'un backtrack sur `e1`, la variable `R/e0` sera remise à vide et la pile sera alors comme avant l'utilisation du FAIT de l'environnement `e1`.

Un autre exemple nous est donné à l'étape du dessin B.10. Lorsque la variable `Z/e1` est unifiée avec `cons5(U,Z)/e2`, `Z/e1` est modifiée, il faut donc la placer dans la pile de trainée car `Z/e1` appartient à un environnement antérieur à e2.

Lorsque nous avons à unifier deux variables libres `X/ei` et `Y/ej`. Nous avons deux possibilité : soit mettre `X/ei` dans `Y/ej`, soit l'inverse. Mais la solution la plus simple est de mettre la variable la plus ancienne (appartenant à l'environnement le plus ancien) dans la variable la plus récente. Il n'est alors pas nécessaire de modifier la pile de trainée.

B.3 Implantation des variables

Nous avons vu qu'à chaque utilisation d'une clause on crée un environnement pour cette clause.

Dans cet environnement, il y a une case par variable de la clause ; ainsi chaque variable peut être accédée par un déplacement dans l'environnement de sa clause.

En pratique, on remplacera le nom des variables par la valeur de ce déplacement ce qui permet un accés encore plus efficace. Le nom n'est conservé dans la structure des variables que pour permettre un debugging plus facile. Par exemple, considérons la clause :

```
#ET(append(X,Y,Z)):append(cons(U,X),Y,cons(U,Z)).
```

Un environnement pour cette clause contiendra 4 cases pour U, X, Y et Z. U sera la variable de déplacement 0, X sera la variable de déplacement 1, Y sera la variable de déplacement 2, Z sera la variable de déplacement 3. Ainsi la clause sera codée par :

```
#ET(append(_1,_2,_3)):append(cons(_0,_1),_2,cons(_0,_3)).
```

Cette valeur correspond à la valeur que nous plaçons dans le champ `dep` d'un `elt_objet`.

Bibliographie

[ABFdC+86] R. Arthaud, P. Bieber, L. Fariñas del Cerro, J. Henry, and A. Herzig. Automated modal reasoning. In *Proc. of the Int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Paris, july 1986.

[AG88] Jean-Marc Alliot and Julian Garmendia. *Une implantation en C de MOLOG.* Rapport de dea, Université Paul Sabatier, 1988.

[AHLM92] Jean-Marc Alliot, Andreas Herzig, and Mamede Lima-Marques. Implementing prolog extensions: a paralle inference machine. In *Fifth Generation Computer Systems (FGCS-92)*, volume 2, pages 833–842. Institute for New Generation Computing Technology, OHMSA, Ltd, June 1992.

[AK90] Khayri A. M. Ali and Roland Karlsson. The muse or-parallel prolog model and its performance. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756, Cambridge, Massachusetts, 1990. MIT Press.

[AK91] Hassan Ait-Kaci. *The Warren abstract machine: a tutorial reconstruction.* MIT press, 1991.

[AM87] Martin Abadi and Zohar Manna. Temporal logic programming. In *International Symposium on Logic Programming.* IEEE, 1987.

[AP92] Jean-Marc Alliot and Francis Preux. Paradise : interface non bloquante des entrées-sorties unix. Compte rendu dans la revue INTERFLASH (AFCET), Mars 1992.

[AvE82] K. R. Apt and M. H. van Emden. Contribution to the theory of logic programming. *JACM*, 29(3), 1982.

[Bau89a] Marianne Baudinet. *Logic Programming Semantics: Techniques and Applications.* PhD thesis, Stanford University, feb 1989.

[Bau89b] Marianne Baudinet. Temporal logic programming is complete and expressive. In *Sixteenth ACM Symposium on Principles of Programming Language*, 1989.

[BCRU84] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. A memory management machine for Prolog interpreters. In S. Tarnlund, editor, *2nd International Conference on Logic Programming.* Uppsala university, 1984.

[BCRU86] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: a memory with a real-time garbage collector for implementing logic programming language. In *3rd symposium of logic programming*, 1986.

[BFdCH88] P. Balbiani, L. Fariñas del Cerro, and A. Herzig. Declarative semantics for modal logics. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, Tokyo, 1988.

[BG87] J. Beer and W. K. Giloi. Pope — a parallel operating PROLOG engine. In *Proceedings of the Fifth Generation Conference on Systems*, 1987.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[Bor91] Stephane Bortzmeyer. Introduction à la programmation système sous unix avec ada. Technical report, Conservatoire National des Arts et Metiers, 1991.

[Bri87] M. Bricard. *Une machine abstraite pour compiler MOLOG.* Rapport D.E.A., Université Paul Sabatier – LSI, 1987.

[Bru82] M. Bruynooghe. A note on garbage collection in PROLOG interpreters. In M. van Caneghem, editor, *Proceedings of the first international logic programming conference*, 1982.

[Cam84] J. A. Campbell, editor. *Implementations of PROLOG.* Ellis Horwood, 1984.

[CG81] K. Clark and S. Gregory. A relational language for parallel programming. In *Proceedings of the conference on functional programming languages and computer architecture.* ACM, 1981.

[CG86] K. Clark and S. Gregory. Parlog and prolog united. In *ACM transactions on programming languages*, 1986.

[Cha87] Jérome Chailloux. *LE_LISP : manuel de référence de la version 15.21.* INRIA, 1987.

[CKvC79] A. Colmerauer, H. Kanaoui, and M. van Caneghem. Etude et réalisation d'un système prolog. Technical report, GIA, Université d'Aix-Marseille II, 1979.

[CM81] K. L. Clark and F. McCabe. *Expert systems in the micro-electronic Age*, chapter The control facilities of IC-PROLOG. Edinburgh University Press, 1981.

[Col75] A. Colmerauer. Les grammaires de métamorphose. Technical report, GIA, Université d'Aix-Marseille II, 1975.

[Col84] A. Colmerauer. PROLOG, langage de l'intelligence artificielle. *La recherche*, 1984.

[CS89] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[dCH88] Luis Fariñas del Cerro and Andreas Herzig. Linear modal deductions. In E. Lusk and R. Overbeek, editors, *Proc. of the 9th Int. Conf. on Automated Deduction*, LNCS 310, pages 487–499, Argonne – USA, may 1988. Springer Verlag.

[DeG84] Doug DeGroot. Restricted and-parallelism. In Hideo Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[dKCRS89] Jacques Chassin de Kergommeaux, Philippe Codognet, Philippe Robert, and Jean-Claude Syre. Une programmation logique parallele. *Technique et Science Informatiques (TSI)*, 8(3 and 4):205–224 and 285–305, 1989.

[DLO87] T. Disz, E. Lusk, and R. Overbeek. Experiments with or-parallel logic programs. In *Proceedings of the fourth International Conference on Logic Programming.* MIT Press, 1987.

[DWDP88] Saumya K. Debray, David Scott Warren, Suzanne Dietrich, and Fernando Pereira. *The SB-PROLOG system, version 3.0.* University of Arizona, September 1988.

[Esp87a] Esprit Project p973 "ALPES". *MOLOG Technical Report*, may 1987. Esprit Technical Report.

[Esp87b] Esprit Project p973 "ALPES". *MOLOG User Manual*, may 1987. Esprit Technical Report.

[FdC86] L. Fariñas del Cerro. MOLOG: A system that extends PROLOG with modal logic. *New Generation Computing*, 4:35–50, 1986.

[FKTMO86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-Oka. Tokio: Logic programming language based on temporal logic and its compilation to prolog. In *Third Int. Conf. on Logic Programming*, pages 695–709, jul 1986.

[GKPC85] F. Giannesini, H. Kanaoui, R. Pasero, and M. Van Caneghem. *PROLOG.* InterEditions, 1985.

[GL82] M. Gallaire and C. Lasserre. Meta-level control for logic programs. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 173–188. Academic Press, 1982.

[GLLO85] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek. A tutorial on the Warren abstract machine for computational logic. Technical report, Argonne National Laboratory, 1985.

[GR84] D. Gabbay and U. Reyle. N-prolog: An extension of prolog with hypothetical implications. *Jounal of Logic Programming*, 1:319–355, 1984.

[Hau90] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[Her86a] Manuel Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 25–39. Springer-Verlag, 1986.

[Her86b] Manuel V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Departement of Computer Sciences, The University of Texas at Austin, August 1986.

[HL91] P. M. Hill and J. W. Lloyd. The Gödel report. Technical report, University of Bristol, 1991.

[HMS89] N. C. Heintze, S. Michaylov, and P. J. Stuckey. On meta programming in CLP(R). In *Proceedings 1st NACLP*. MIT Press, October 1989.

[HNSN86] T. Hattori, R. Nakajima, T. Sakuragawa, and N. Niide. Racco: a modal logic programming language for writing models of real-time process control systems. Technical report, Research Institute for Mathematical Science, Kyoto University, 1986.

[KK71] R. A. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2, 1971.

[Kow79] Robert Kowalski. *Logic for problem solving.* The computer science library — Elsevier, 1979.

[KvE74] R. Kowalsi and M. van Emden. The semantic of predicate logic as programming language. DCL Memo 73, University of Edinburgh, 1974.

[LK88] Y. J. Lin and V. Kumar. And-parallel execution of logic programs on a shared memory multi-processors. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, 1988.

[Llo84] J. W. Lloyd. *Foundations of logic programming.* Springer Verlag, 1984.

[LWH+88] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*, pages 819–830. ICOT, Tokyo, Japan, November 1988.

[LWH+90] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[Mas86] Hideo Masuzawa. Kabu wake parallel inference mechanism and its evaluation. In *FJCC IEEE*, 1986.

[MP88] Luis Monteiro and Antonio Porto. Modules for logic programming based on context extension. In *Int. Conf. on Logic Programming*, 1988.

[Nil84] M. Nilsson. *Implementations of PROLOG*, chapter The world's shortest PROLOG interpreter. Ellis Horwood, 1984.

[OW90] Memhet A. Orgun and William W. Wadge. Theory and pratice of temporal logic programming. Technical report, University of Victoria, Canada, 1990.

[P+86] Christian Percebois et al. Projet coala, articles et publications, 1984-1986. Technical report, LSI, 1986.

[P+87] Christian Percebois et al. Simulation results of a multiprocessor prolog architecture based on a distributed and/or graph. Technical Report 6, Greco-Programmation, 1987.

[Pen85] Martti Penttonen. Introduction to the implementation of prolog. Technical report, University of Turku, Finland, 1985.

[PPW79] L. M. Pereira, F. Pereira, and D. H. D. Warren. *User's guide to DEC-10 PROLOG*. DAI, University of Edinburgh, 1979.

[RAM91] Jonathan Rees, Norman Adams, and James Meehan. *The T manual (version 3.1)*. University of Yale, 1991.

[Rid92] Olivier Ridoux. MALI v06, tutorial and reference manual. Technical report, IRISA/INRIA, 1992.

[Rob65] J. A. Robinson. A machine oriented based on the resolution principle. In *JACM 12*, 1965.

[Sak87] Y Sakakibara. Programming in modal logic: An extension of PROLOG based on modal logic. In *Int. Conf. on Logic Programming*, 1987.

[Sak89] Takashi Sakuragawa. Temporal PROLOG. In *RIMS Conf. on software science and engineering*, 1989.

[Sak90] Takushi Sakuragawa. Temporal prolog. In *RIMS conference on software science and engineering*, 1990.

[SCWY91a] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*. MIT Press, 1991.

[SCWY91b] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*. MIT Press, 1991.

[Sha87] E. Shapiro. Concurrent prolog: a progress report. In E. Shapiro, editor, *Concurrent Prolog: collected papers*, pages pp157–187. The MIT Press, Cambridge, Massachusetts, 1987.

[SSK+85] Y. Sohma, Ken Satoh, Koichi Kumon, Hideo Masuzawa, and Akihiro Itashiki. A new parallel inference mechanism based on sequential processing. In *IFIP TC-10 working conference on fifth generation computer architecture*, 1985.

[Sut90] Geoff Sutcliffe. PROLOG-"LINDA: an embedding of linda in muprolog. In *Proceedings of the 4th Australian Conference on Artificial Intelligence*, 1990.

[Sze89] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[Ued87] Kazunori Ueda. Guarded horn clauses. In Ehud Shapiro, editor, *Concurrent Prolog, collected papers*, volume vol 1, pages 140–156. MIT Press, Cambridge, Massachusetts, 1987.

[vC86] Michel van Caneghem. *L'anatomie de PROLOG*. InterEditions, 1986.

[War77] D. H. D. Warren. Implementing PROLOG : Compiling predicate logic programs. report 39-40, DAI, 1977.

[War83] D. H. D. Warren. An abstract PROLOG instruction set. note 309, SRI international, October 1983.

[War87] D. H. D. Warren. The SRI model for or-parallel execution of PROLOG— abstract design and implementation issues. In *Proceedings of the Symposium Working Conference on Logic Programming*, 1987.

[Wie] Jan Wielemaker. *SWI-Prolog 1.5 reference manual.* University of Amsterdam. 1990.

[WRdKS87] Harald Westphal, Philippe Robert, Jacques Chassin de Kergommeaux, and Jean-Claude Syre. The pepsys model: Combining backtracking, and- and or-parallelism. In *Proceedings of the 4th Symposium on Logic Programming*, pages 436–448. IEEE, 1987.

[YH86] Taiichi Yuasa and Masami Hagiya. The kyoto common lisp report. Technical report, Research Institute for mathematical science, Kyoto University, 1986.