HAL
open science

Etude et Réalisation d'un Système d'Aide à la Mise au Point en Programmation Logique

Christian Debarbieri

To cite this version:

Christian Debarbieri. Etude et Réalisation d'un Système d'Aide à la Mise au Point en Programmation Logique. Intelligence artificielle [cs.AI]. Ecole Nationale Supérieure des Mines de Saint-Etienne; Université Jean Monnet - Saint-Etienne, 1990. Français. NNT : . tel-00831323

HAL Id: tel-00831323

https://theses.hal.science/tel-00831323

Submitted on 6 Jun 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ÉCOLE NATIONALE SUPÉRIEURE
DES MINES DE SAINT ÉTIENNE

N° d'Ordre 37 ID

THESE

Présentée par

Christian DEBARBIERI

Ingénieur Civil des Mines

Pour obtenir le titre de

DOCTEUR

De L'Université de Saint Étienne
et de L'École Nationale Supérieure des Mines de Saint Étienne

Spécialité : Intelligence Artificielle

Étude et Réalisation d'un Système d'Aide à la Mise au Point en Programmation Logique

Soutenue à Saint-Etienne le 25 Avril 1990

Composition du Jury

Président :
Monsieur J. CHOURAQUI

Rapporteurs :
Messieurs
G. FERRAND
J.P. DELAHAYE

Examinateurs :
Madame Y. AHRONOVITZ

Messieurs C. BERTIN
H. COELHO
B. PEROCHE

ÉCOLE NATIONALE SUPÉRIEURE
DES MINES DE SAINT ÉTIENNE

N° d'Ordre 37 ID

THESE

Présentée par

Christian DEBARBIERI

Ingénieur Civil des Mines

Pour obtenir le titre de

DOCTEUR

**De L'Université de Saint Étienne
et de L'École Nationale Supérieure des Mines de Saint Étienne**

Spécialité : Intelligence Artificielle

Étude et Réalisation d'un Système d'Aide à la Mise au Point en Programmation Logique

Soutenue à Saint-Etienne le 25 Avril 1990

Composition du Jury

Président :
Monsieur J. CHOURAQUI

Rapporteurs :
Messieurs
G. FERRAND
J.P. DELAHAYE

Examinateurs :
Madame Y. AHRONOVITZ

Messieurs C. BERTIN
H. COELHO
B. PEROCHE

Partie 1 5
1. Avant Propos 7
1.1 Historique 7
1.2 Généralités sur Prolog 9
1.3 But du projet 12
1.4 Plan de la Thèse 14
2. Théorie du Premier Ordre 15
2.1 Définition du langage 15
2.2 Théorie sémantique 19
2.3 Unification 25
2.3.1 Substitution 25
2.3.2 Algorithme d'unification 27
2.4 Théorie syntaxique 28
2.5 Programmes définis 31
2.5.1 Définitions 31
2.5.2 Résolution SLD 32
2.6 Programmes normaux 35
2.7 La négation 36
2.7.1 Le monde clos 36
2.7.2 Complétion d'un programme 37
2.7.3 La négation par l'absurde 39
2.8 La coupure 39
3. Mise au point 41
3.1 Les problèmes 41
3.2 Qu'est-ce-qu'une mise au point déclarative 43
3.2.1 Cas d'erreurs en Prolog 43
3.2.2 Techniques de recherche déclaratives 44
3.3 Conclusion 48
Partie 2 49
4. Les traces 51
4.1 Principe 51
4.2 Sémantique de la résolution 52
4.3 Trace de Byrd 53
4.3.1 La trace 54
4.3.2 Extensions 55
4.3.3 Remarques 56
4.3.4 Commentaires 57
4.4 Trace de Boizumault 58
4.4.1 La trace 59
4.4.2 Commentaires 60
4.5 Trace de Eisenstadt 60
4.5.1 Exemple de trace 61
4.5.2 Extension 62
4.5.3 Commentaires 63
4.6 Conclusion sur les traces 63
5. La détection d'erreurs 65
5.1 Principe 65
5.2 "Algorithmic Program Debugging" 65
5.2.1 Solution fausse 65

5.2.2 Solution insatisfaite 69
5.2.3 Bouclage du programme 72
5.2.4 Conclusion 73
5.3 Rational Debugging 73
5.3.1 Solution fausse 74
5.3.2 Absence de solution 75
5.3.3 Remarque 77
5.4 Autres méthodes 77
5.4.1 Declarative Error Diagnosis 77
5.4.2 Top-Down Diagnosis 78
5.4.3 Error Diagnosis in Logic Programming 78
5.4.4 Select And Query 78
5.4.5 Méthodes Déductives 79
5.5 Conclusion sur la détection d'erreurs 79
6. Extensions 81
6.1 La solution fausse 81
6.1.1 Analyse du problème 81
6.1.2 Algorithme de recherche 84
6.2 La solution insatisfaite 86
6.2.1 Principe 86
6.2.2 Réalisation 87
6.2.3 Une autre approche 89
6.2.4 L'extension du coupe-choix 90
6.3 La négation 93
6.4 Extensions 97
6.4.1 Importance 97
6.4.2 Prédicats logiques et méta-logiques 97
6.4.3 Prédicats extra-logiques 98
6.5 Conclusion sur les extensions 100
Partie 3 101
7. Méthodologie 103
7.1 Quelle mise au point ? 103
7.2 Un exemple 103
7.3 Une mise au point statique ? 106
7.4 Une mise au point dynamique ? 107
7.5 Conclusion sur la méthodologie 110
8. La détection d'erreurs 113
8.1 Le bouclage 113
8.1.1 Évaluation d'un but 113
8.1.2 Le problème de la dérivation infinie 114
8.1.3 Détection d'une récursivité 115
8.1.4 La comparaison de deux buts 117
8.1.5 La similitude de deux buts 117
8.2 Vérification d'une résolution 119
8.2.1 Pourquoi une autre approche ? 119
8.2.2 Principe 120
8.2.3 Les prédicats méta-logiques 122
8.2.4 La coupure 122
8.2.5 La négation par l'absurde 123

8.2.6 Conclusion 124
9. Réalisations 125
9.1 Un méta-interprète 125
9.1.1 Un Méta-Interprète de Prolog pur 125
9.1.2 Généralisation 126
9.1.3 Le coupe-choix 128
9.1.4 La négation 131
9.1.5 Le but initial 132
9.1.6 Méta Prolog : le méta-interprète 132
9.1.7 Application 134
9.1.8 Intérêt 140
9.2 La détection des boucles 140
9.2.1 La gestion des buts ancêtres 141
9.2.2 La similitude de deux buts 142
9.2.3 Le programme 143
9.2.4 Exemples 144
9.2.5 Conclusion 146
9.3 La vérification d'une résolution 148
9.3.1 L'algorithme 148
9.3.2 Exemples 151
9.3.3 Conclusion 152
9.4 Conclusion sur la réalisation 152
Partie 4 155
10. Le matériel 157
10.1 Interface graphique 157
10.1.1 Architecture de NeWS 158
10.1.2 Programmer avec NeWS 159
10.1.3 L'interface utilisateur 160
10.2 Le système Prolog 162
11. L'application 165
11.1 L'interface utilisateur 166
11.2 Le système de détection d'erreurs 167
11.3 Exemple 168
11.4 Une trace interactive 174
11.5 Une session de mise au point 176
11.6 Conclusion sur l'application 179
Partie 5 181
Annexe 1 187
Annexe 2 195
Annexe 3 199
Annexe 4 205

Partie 1

Introduction

1. Avant Propos

1.1 Historique

La programmation logique s'est développée au début des années 1970 par la conjonction de deux projets de recherche différents : *la démonstration automatique de théorèmes et le traitement de langages naturels ou artificiels.*

La contribution d'Alain Colmerauer provient de son intérêt pour le traitement automatique du langage naturel, alors que celle de Robert Kowalski a pour origine sa compétence en logique et la démonstration automatique de théorèmes [Colmerauer-73, Kowalski-74].

Les travaux effectués par Herbrand ont contribué au développement de la démonstration automatique de théorèmes au début des années 1960 [Herbrand-31,67]. Mais sa méthode de déduction était impraticable par un système informatique. Ces travaux aboutirent à la publication par Robinson d'un important papier sur la technique de déduction, introduisant la règle de résolution [Robinson-65]. La résolution est une règle d'inférence particulièrement adaptée à la déduction automatique sur un ordinateur.

En 1970, Colmerauer et Kowalski aboutirent à l'idée fondamentale que *la logique pouvait être utilisée comme langage de programmation.* Ainsi, le premier interprète Prolog, pour Programmation en logique, fut développé à l'université d'Aix-Marseille et écrit en langage ALGOL-W par Roussel [Colmerauer-73]. Puis une version améliorée et plus influente écrite en FORTRAN donna naissance à Prolog-1.

L'apparition d'interprètes sur de gros systèmes, tels Prolog-Dec10, puis des premiers compilateurs Prolog basés sur le fonctionnement de la machine virtuelle de Warren contribuèrent à la renommée du langage ainsi qu'a sa diffusion dans les universités puis l'industrie comme un puissant langage de l'Intelligence Artificielle [Warren-77, 83].

Le langage Prolog est fondé sur une restriction de la *logique du premier ordre* : l'ensemble des clauses de Horn. Le fait d'utiliser la logique du premier ordre comme langage de programmation était révolutionnaire parce que, jusqu'en 1972, une des utilisations de la logique en informatique était pour écrire des spécifications formelles et déclaratives. Kowalski nous montre dans son papier sur ce langage [Kowalski-74] que la logique des clauses de Horn possède une interprétation opérationnelle qui la rend très efficace comme fondement d'un langage de programmation, puis il effectue une preuve de la complétude de la méthode de dérivation sur laquelle se fondent tous les interprètes Prolog.

De plus, la définition d'une sémantique formelle d'un langage de programmation logique [Emden-Kowalski-76] permet de définir une sémantique du point fixe pour les programmes utilisant les clauses de Horn comme syntaxe formelle, mais elle permet aussi de démontrer qu'elle était identique au modèle minimal d'une sémantique opérationnelle. Cet aspect est aussi étudié dans le cadre de la norme Prolog en cours de rédaction [Deransart&all-85].

Une clause de Horn définie est une formule logique de la forme $\{A \Leftarrow A_1 \wedge A_2 \ldots \wedge A_n\}$. On peut la considérer comme la définition d'une procédure que l'on appelle un prédicat, dans laquelle chaque littéral A_i est considéré comme l'appel d'une procédure cherchant une solution de ce prédicat. Un programme Prolog se déroule lorsqu'on lui fournit un but à résoudre qui est une clause de Horn de la forme $\{\Leftarrow C_1 \wedge C_2 \ldots \wedge C_m\}$. Si la question posée au système Prolog est le but $\{\Leftarrow C_1 \wedge C_2 \ldots \wedge C_m\}$, une étape de la résolution consiste à trouver une clause $\{A \Leftarrow A_1 \wedge A_2 \ldots \wedge A_n\}$ et un indice i tels que le sous-but C_i s'unifie avec la tête A par une substitution Θ des variables du couple (A, C_i), puis à remplacer le but initial par le nouveau but $\{(\Leftarrow C_1 \ldots \wedge C_{i-1} \wedge A_1 \ldots \wedge A_n \wedge C_{i+1} \ldots \wedge C_m)\Theta\}$ qui devient ainsi le but initial de l'étape suivante. L'unification apparait comme un mécanisme de passage de paramètres, mais aussi comme un outil de sélection et de construction de données. Le programme se termine lorsqu'une étape de la résolution produit un nouveau but vide.

Une des idées principales de la programmation logique due à Kowalski [Kowalski-79] est qu'un algorithme est constitué de deux composantes ; la première composante, la logique, permet de spécifier un problème à résoudre, la seconde composante, le contrôle, permet de décrire comment on doit résoudre le problème posé.

Un système idéal de programmation logique serait de programmer dans la partie purement déclarative, c'est à dire dans la composante logique. On ne laisserait à ce système que la partie contrôle. Malheureusement, les systèmes actuels de programmation logique n'y sont pas encore parvenus et l'utilisateur doit savoir exprimer le contrôle au niveau des clauses et comment le système l'utilise pour modifier la résolution d'un problème.

1.2 Généralités sur Prolog

La programmation logique se fonde sur la théorie de la logique du premier ordre. Nous présentons ici un exemple simple de programme permettant d'illustrer quelques concepts.

Nous nous intéressons au problème de la représentation d'un arbre généalogique et à la déduction d'informations obtenues à partir de relations familiales.

Considérons un ensemble de phrases en langue naturelle correspondant à la description d'une certaine famille :

pierre est le père de hervé, de arthur et de marie,
lucien est le père de jeanne et de amélie,
jeanne est la mère de marie et de arthur,
amélie est la mère de andré,
berthe est la mère de amélie et de jeanne.

Un individu quelconque qui lira ces phrases comprendra leur signification. De plus, en appliquant certaines règles que l'individu possède par ses connaissances propres, il pourra *déduire* de ces phrases d'autres phrases en se posant des questions sur la famille, comme par exemple :

jeanne est-elle une enfant de andré ?
réponse = non
jeanne est l'enfant de qui ?
de lucien et de berthe
...

Si nous voulons disposer d'un système automatique de déduction sur ce domaine particulier, nous devons au préalable décrire ce domaine par un ensemble d'axiomes qui sera la traduction directe de phrases que nous avons considérées.

Chaque personne est un homme ou une femme. Cette information est contenue dans le genre de son prénom. Nous décrivons les individus du domaine par l'ensemble d'axiomes suivant, appelé aussi des faits :

homme(pierre)
homme(hervé)
homme(arthur)
homme(andré)
homme(lucien)
femme(berthe)
femme(marie)
femme(amélie)
femme(jeanne)

L'axiome *"homme(X)"* spécifie que l'individu de prénom X est un homme, l'axiome *"femme(Y)"* spécifie que l'individu de prénom Y est une femme. Maintenant nous pouvons déclarer les axiomes de lien de parenté entre deux individus, traduction des phrases considérées, de la façon suivante :

père(pierre, hervé)
père(pierre, arthur)
père(pierre, marie)
père(lucien, jeanne)
père(lucien, amélie)
mère(jeanne, marie)
mère(jeanne, arthur)
mère(amélie, andré)
mère(berthe, amélie)
mère(berthe, jeanne)

Où la relation *"père(X, Y)"* indique que l'individu X est le père de l'individu Y, la relation *"mère(X, Y)"* indique que l'individu X est la mère de l'individu Y.

Maintenant si nous voulons déduire de cet ensemble d'axiomes certaines relations de parenté qui peuvent être :

enfant(Enfant, Parent),
qui dit que Enfant est un enfant de Parent,

fils(Fils, Parent),
qui dit que Fils est un fils de Parent,

fille(Fille, Parent),
qui dit que Fille est une fille de Parent,

frère_sœur(X, Y),
qui dit que X est un frère ou une sœur de Y.

Nous devons traduire les règles de déductions que nous possédons sur ce domaine. Nous pouvons déjà remarquer que si l'individu Enfant est un enfant de l'individu Parent alors Parent est soit un père soit une mère de Enfant. Cette relation peut s'écrire directement sous forme de deux règles :

enfant(Enfant, Parent)
si père(Parent, Enfant).

enfant(Enfant, Parent)
si mère(Parent, Enfant).

Qui signifient que Enfant est un enfant de Parent si Parent est le père de Enfant ou si Parent est la mère de Enfant. Nous pouvons déjà déduire quelques relations simples par l'utilisation de ces règles et en appliquant des déductions à partir des axiomes. Nous pouvons poser les questions suivantes :

enfant(jeanne, andré)?
Non
% impossible à démontrer
enfant(jeanne, Parent)?
Parent = lucien % car père(lucien, jeanne) est un fait
Oui
Parent = berthe % car mère(berthe, jeanne) est un fait
Oui

Les autres relations peuvent se déduire de la même façon à partir des connaissances sur le domaine d'application, par l'ensemble des règles :

fils(Fils, Parent)
si enfant(Fils, Parent) et homme(Fils)

fille(Fille, Parent)
si enfant(Fille, Parent) et femme(Fille)

frère_sœur(X, Y)
si père(P, X)
et père(P, Y) % ils ont même père
et mère(P, X)
et mère(P, Y) % ils ont même mère
et X ≠ Y % X et Y sont différents.

Selon le même principe nous pouvons écrire d'autres relations de lien de parenté, telles que la relation *"grand_père(X, Y)"* signifiant que X est le grand père de Y, la relation *"grand_mère(X, Y)"* signifiant que X est la grand mère de Y, la relation *"oncle(X, Y)"* signifiant que X est l'oncle de Y... Nous laissons le soin au lecteur d'écrire l'ensemble des relations de parenté que nous pouvons déduire sur un arbre généalogique. Cet exemple d'application et de modélisation d'un domaine réel est une bonne introduction à la programmation logique et plus particulièrement au système de déduction automatique de théorèmes qu'est un système Prolog pur.

Comme nous venons de le voir sur cet exemple, le langage Prolog est composé de deux parties, l'algorithme de déduction (la résolution d'un but) et la représentation des connaissances du domaine d'application (les faits, les règles et les buts). La suite de cette introduction va se découper en trois parties ; la première présente les objectifs du travail de recherche que j'ai effectué, la seconde les fondements théoriques de la logique du premier ordre, la troisième une présentation des problèmes du développement d'une application en Prolog.

1.3 But du projet

Lorsqu'un nouvel utilisateur du langage Prolog commence à écrire des programmes, il se trouve devant de nouvelles difficultés de programmation s'il connaît déjà un langage impératif tel que Pascal ou C. Le cas des nouveaux programmeurs pose moins de problèmes bien qu'il s'en rapproche légèrement.

Ce qui est déroutant lorsque l'on aborde la programmation logique, c'est le fait que l'utilisateur n'a pas besoin de connaître a priori la manière dont on résout la question posée au système, du moins si l'on programme en logique pure, et bien souvent bon nombre d'utilisateurs sont rebutés par une telle approche, que l'on peut nommer "déclarativité du langage".

C'est justement cet aspect du langage, qui le rend très attrayant et qui parfois fait fuir les programmeurs automates, ceux qui ne peuvent programmer dans un langage que si celui-ci est impératif et automatique.

La logique du premier ordre est un formalisme qui hérite de la culture mathématique que nous possédons. Ainsi, toute personne ayant suivi une scolarité normale, peut utiliser la programmation logique pour formaliser certaines déductions sur divers problèmes qu'il se pose.

Cependant, le langage Prolog est plus qu'un simple langage de programmation en logique, il possède une sémantique "opérationnelle" qui lui permet de calculer des solutions à un problème donné. Ainsi, toute personne se posant une question en programmation logique, trouvera un ensemble de réponses que lui fournira un système Prolog.

La difficulté que j'ai rencontrée lorsque j'ai débuté dans ce langage, est la puissance d'expression du langage. Tout peut s'écrire en utilisant la programmation logique. Un autre aspect difficile à maîtriser est le non déterminisme de la sémantique du langage.

Pour bien programmer en Prolog, il est nécessaire de remettre en cause toute la culture informatique que l'on possède et de raisonner d'une façon nouvelle pour exprimer ses théories, par l'utilisation de la sémantique déclarative du langage, mais aussi par l'approche récursive que l'on peut comparer aux démonstrations par récurrences que nous avons tous un jour utilisées.

Cet aspect des choses peut compliquer l'écriture de programmes Prolog. Lorsque l'on écrit une application en Prolog, on utilise le modèle sous-jacent du programme, c'est le modèle théorique, pour déclarer des formules logiques. En revanche, lorsque le système Prolog recherche une solution au problème posé, c'est un but à résoudre, il utilise une sémantique qui n'est pas forcément identique à celle désirée, car bien souvent le programmeur réalise une sous-partie de la théorie. La différence entre les deux sémantiques n'est pas forcément une erreur de programmation, car il est plus simple de déclarer une partie de la théorie que de toute la réaliser pour résoudre son problème.

En revanche, lorsque la différence de ces deux sémantiques est gênante pour la justesse de la solution obtenue, on obtient ce que l'on appelle un symptôme d'une erreur du programme. Il faut donc corriger ce programme pour obtenir une bonne sémantique opérationnelle et suffisamment proche de la sémantique attendue.

Nous voyons donc que nous avons besoin d'analyser la justesse d'un programme, par les symptômes que l'on obtient sur des buts particuliers, que l'on donne au système Prolog pour tester le programme.

Tout utilisateur Prolog s'est confronté au problème de la justesse d'un de ses programmes. La technique que l'on utilise, pour corriger une erreur, est d'essayer de comprendre pourquoi il existe une différence entre les deux sémantiques. Il faut donc connaître de façon parfaite la méthode de déduction que réalise tout système Prolog pour analyser la sémantique opérationnelle.

On voit donc ici la contradiction de tout système Prolog, elle nécessite une connaissance calculatoire, au détriment de la simple connaissance déclarative. L'utilisateur ne dispose, dans les systèmes actuels, que d'une méthode de trace de la résolution Prolog permettant de suivre la résolution d'un but.

Lorsque l'utilisateur réalise une mise au point d'un de ses programmes, c'est-à-dire qu'il réalise une détection d'erreurs et une correction d'erreurs, celui-ci doit mettre en oeuvre un ensemble de raisonnement difficile à obtenir par tous les programmeurs. Il faut analyser la sémantique attendue du programme et analyser la sémantique opérationnelle pour détecter une première incohérence, fournissant l'erreur cherchée, mais aussi la cause de l'erreur.

L'objectif des systèmes de mise au point déclaratifs est de démocratiser cette phase d'analyse par comparaison, le programmeur spécifie la sémantique attendue de son programme et c'est le système qui réalise l'expertise nécessaire pour fournir la première erreur du programme à partir de cette description formelle du modèle théorique et de la sémantique obtenue.

L'objectif de cette thèse est d'étudier comment on peut réaliser un système expert d'aide à la mise au point de programmes Prolog capable d'automatiser la détection d'erreurs, mais aussi fournir un environnement de vérification et de validation de logiciel.

De plus, les évolutions du langage Prolog lui ont donné des primitives non logiques, comme la négation par l'absurde, la coupure et un certain nombre prédicats de service. Un objectif primordial de cette thèse est d'étudier comment on peut tenir compte de ces primitives pour réaliser un système de mise au point déclaratif, par l'utilisation unique de la sémantique attendue du programme, mais aussi de fournir une méthodologie d'utilisation d'une trace intelligente permettant de traiter les cas impossibles à résoudre par une détection d'erreurs.

1.4 Plan de la Thèse

Cette thèse se compose de quatre parties.

Mise en garde. Le lecteur doit se souvenir des obejctifs de cette thèse ; la mise au point du *langage* de programmation Prolog. Cette mise au point ne permet pas de répondre à celle de certaines utilisations de la Programmation Logiques, par exemple les systèmes experts.

La première partie présente une introduction aux fondements de la programmation logique et aux problèmes de mise au point en Prolog. Le premier chapitre est l'avant propos, introduisant la programmation logiques et les objectifs de cette Thèse. Le second chapitre est une présentation des fondements de la logique du premier ordre. Le troisième chapitre introduit les problèmes du développement d'applications en Prolog, et plus particulièrement la mise au point des programmes.

La seconde partie, découpée en trois chapitres, est une présentation de l'état de l'art dans le domaine de la mise au point de programmes Prolog. Le quatrième chapitre présente l'approche classique de mise au point par l'utilisation de traces de la résolution. Le cinquième chapitre présente les principes de la détection d'erreurs en Prolog, sous ces différentes approches, algorithmique, rationnelle, déductive... Ces méthodes utilisent la sémantique attendue du programme afin de réaliser une détection d'erreurs. Le sixième chapitre est l'analyse critique des extensions des méthodes de détection d'erreurs en Prolog, elle d'introduit les fondements de la méthode de vérification d'une résolution.

La troisième partie, découpée en trois chapitres, présente l'étude de la mise au point, par l'utilisation de fondements introduit dans la partie deux. Le septième chapitre aborde les problèmes d'une méthodologie de développement d'application Prolog, et plus particulièrement quelle méthode de mise au point doit-on utiliser pour réaliser un développement efficace sur des programmes Prolog. Le huitième chapitre présente l'étude formelle des méthodes de mise au point que nous devons mettre en oeuvre en Prolog, une détection de boucle pour une utilisation de la négation, une détection intelligente d'erreurs pour une mise au point avec négation et coupure. La neuvième partie décrit la réalisation "algorithmique" des principes précédents par la réalisation d'un méta-interprète fondée sur l'évaluation partielle, et l'utilisation de celui-ci pour la détection de boucles et la détection d'erreurs intelligente sur des programmes pouvant contenir des négation et des coupures.

La quatrième partie, composé de deux chapitres, est la description du prototype que nous avons développé au sein du laboratoire de l'école des Mines de Saint-Etienne. Le dixième chapitre est la description de l'environnement de travail utilisé. Le onzième et dernier chapitre décrit le prototype utilisant les algorithmes mis au point dans la partie précédente.

2. Théorie du Premier Ordre

Cette partie à été réalisé dans un objectif de référence bibliographique interne au département de l'école des Mines de Saint-Stiennes.

2.1 *Définition du langage*

La théorie du premier ordre est la réunion de diverses parties : un alphabet, un langage du premier ordre, un ensemble d'axiomes et un ensemble de règles d'inférences. Le langage du premier ordre est un ensemble de formules bien formées de la théorie. Les axiomes sont un sous-ensemble désigné des formules bien formées. L'utilisation simultanée des axiomes et des règles d'inférences permet de déduire des théorèmes de la théorie.

L'alphabet de la théorie du premier ordre est constitué par : *les variables, les constantes, les symboles de fonctions, les symboles de prédicats, les connecteurs de la logique, les quantificateurs de la logique et les symboles de ponctuations.* Dans l'exposé qui va suivre nous adoptons une convention particulière de représentation de ces catégories. Pour un meilleur approfondissement de l'exposé, il peut être intéressant de lire quelques ouvrages sur la logique mathématique [Boolos-Jeffrey-80, LLoyd-87, Loveland-78, Pabion-76, Thayse-88]. De même, le livre de Hofstadter, sur le théorème de Gödel, permet de comprendre certaines notions introduites dans ce chapitre [Hofstadter-85].

La logique du premier ordre utilise un ensemble d'objets élémentaires, l'*alphabet* de la logique, pour la représentation des formules. Le choix que nous faisons pour la représentation des symboles est purement arbitraire, ils peuvent varier d'un langage à un autre, mais ils ont une même signification. Nous pouvons répartir ces objets en divers ensembles :

- Les variables seront dénommées par un nom dont la première lettre est une majuscule, par exemple *Var*.

- Les constantes seront dénommées par un nom dont la première lettre est une minuscule, par exemple *constante*.

- Les symboles de fonctions, d'arité > 0 (nombre d'arguments), seront dénommés par des noms, par exemple *f* est le symbole fonctionnel de "*f(x, Y)*".

- Les symboles de prédicats, ou symbole relationnel, d'arité ≥ 0, seront dénommés par des noms, par exemple *père* est le symbole relationnel de l'atome "*père(X, Y)*".

- Les connecteurs seront les connecteurs logiques, la négation "¬", le ou logique "∨", le et logique "∧", l'implication "⇒", l'équivalence "⇔".

- Les quantificateurs "∃" (il existe) et "∀" (quel que soit).

Le langage de la théorie du premier ordre manipule des formules logiques. Ces formules sont construites à partir de termes qui représentent les objets manipulés.

Définition :

un ***terme*** est défini de la façon suivante :

- Une constante est un terme.
- Une variable est un terme.
- Si f est un symbole fonctionnel n-aire et $(t_1, t_2..., t_n)$ n termes, $f(t_1, t_2 ..., t_n)$ est aussi un terme.

Exemple :

"0" représente l'entier nul, "s(s(s(0)))" représente le troisième successeur de l'entier nul, ce sont des termes.

Pour construire les formules de la logique du premier ordre on utilise les formules élémentaires construites à partir des symboles relationnels et des termes. Ces formules élémentaires sont appelées "formules atomiques".

Définition :

Une ***formule atomique*** est définie de la façon suivante :

- Tout symbole de prédicat d'arité 0 est une formule atomique,
- Soient R est un symbole relationnel n-aire et n termes $(T_i)_n$, $R((T_i)_n)$ est une formule atomique.

Exemple :

"plus(s(0),s(s(0)),s(s(s(0))))" est une formule atomique.

L'utilisation des formules atomiques, des connecteurs logiques et des quantificateurs permettent de construire toutes les formules utilisées dans la logique du premier ordre. Nous pouvons définir les formules par :

Définition :

Une ***formule*** est définie de la façon suivante :

- Une formule atomique est une formule.
- Si F est une formule, $(\neg F)$ est une formule.
- Si F et G sont des formules, $(F \vee G)$, $(F \wedge G)$, $(F \Rightarrow G)$ et $(F \Leftrightarrow G)$ sont aussi des formules.
- Si F est une formule, et X une variable, $(\forall X F)$ et $(\exists X F)$ sont des formules.

Exemple :

"($\forall$ X $\forall$ Y $\forall$ Z (plus(X, Y, Z) $\Rightarrow$ plus(s(X), Y, s(Z)))" est une formule.

Il est à remarquer que généralement on sous-entend les parenthèses pour une commodité d'écriture. Nous pouvons maintenant définir le langage du premier ordre construit à partir des formules.

Définition :

Le langage du premier ordre obtenu par un alphabet est l'ensemble de toutes les formules construites à partir des symboles de l'alphabet.

Exemple :

L = {{Constantes}, {Fonctions}, {Relations}, {Propositions}}, est la définition du langage du premier ordre construit à partir de l'alphabet donné.

Ainsi, {$\forall$X $\exists$Y (p(X, Y) $\Rightarrow$ q(X))}, {$\neg$ $\exists$X (p(X, a) $\wedge$ q(f(X)))} sont des formules de la logique du premier ordre.

La sémantique informelle de la proposition {$\forall$X $\exists$Y (p(X Y) $\Rightarrow$ q(X))} est :

pour tout X, il existe Y tel que si p(X, Y) est vraie alors q(X) est vraie.

Les variables occupent une place importante en mathématique. Elles permettent d'identifier des emplacements dans une formule mathématique qui pourront être occupés par des objets quelconques du domaine de référence.

Définition :

Une ***occurrence liée*** d'une variable dans une formule est une occurrence de cette variable dans la portée du quantificateur associé à la variable. Toute autre occurrence d'une variable est dite *libre*.

Exemple :

Dans la formule {$\forall$ X p(X, Y) $\wedge$ q(X)} la première occurrence de X est liée au quantificateur, la seconde est libre, l'occurrence de Y est libre.

Cette formule est aussi égale à la formule {$\forall$ Z p(Z, Y) $\wedge$ q(X)}. La variable liée X a été remplacée par une autre variable. C'est une substitution de variables.

Parmi toutes les formules que l'on peut construire en logique du premier ordre, celles qui ne possèdent pas de variables libres sont importantes, car on peut leurs attribuer une sémantique, ce sont les énoncés ou formules closes. Cette classe de formules représente les formules closes. Nous pouvons les définir par :

Définition :

Un ***formule close*** est une formule qui ne possède pas de variable libre. Toutes les occurrences des variables de cette formule sont liées (il n'y a pas de variables libres). Les formules closes sont aussi appelées formules bien formées, ou encore énoncés.

Exemple :

$\{\forall Y \exists X (p(X, Y) \wedge q(X))\}$ est une formule close.
Alors que $\{\forall X (p(X, Y) \wedge q(X))\}$ ne l'est pas car la variable Y est libre.

La logique du premier ordre manipule des formules closes pour représenter une théorie. Dans certains traitements, pour faciliter le travail de déduction automatique, on souhaite utiliser des formes particulières de formules ne faisant intervenir qu'un seul symbole de quantification. Pour cela nous devons définir ces formules particulières qui seront utilisées comme formules de bases d'un programme logique.

Définition :

Soit une formule F. La formule $(\forall F)$ dénote la ***clôture universelle*** de F telle que la proposition obtenue à partir de F en quantifiant universellement toutes les variables libres de F.

Définition :

Soit une formule F. La formule $(\exists F)$ dénote la ***clôture existentielle*** de F telle que la proposition obtenue à partir de F en quantifiant existentiellement toutes les variables libres de F.

Exemple :

Soit $F = (p(X, Y) \wedge q(X))$
$\forall F$ représente la formule close $\{\forall X \forall Y (p(X, Y) \wedge q(X))\}$,
$\exists F$ représente la formule close $\{\exists X \exists Y (p(X, Y) \wedge q(X))\}$.

Lorsque l'on manipule des formules closes on utilise des formules atomiques ou des négations de formules atomiques. Ces formules particulières représentent une classe de formules appelées les littéraux.

Définition :

On appelle ***littéral*** une formule atomique ou la négation d'une formule atomique.

Nous disposons désormais de toutes les définitions nécessaires pour construire les formules permettant d'écrire des programmes logiques. Ces formules représentent une classe particulière de formules pour lesquelles il existe un algorithme de déduction de théorèmes permettant de conclure à la validité de formules atomiques relativement à une théorie formelle.

Définition :

Une ***clause*** est une formule close de la forme $\{\forall (L_1 \vee L_2 ... \vee L_n)\}$ où chaque L_i est un littéral.

Dans la logique du premier ordre les clauses sont importantes, pour une lisibilité plus simple on notera une clause de la forme :

$$\forall (A_1 \vee A_2 \vee ... A_m \vee \neg B_1 \vee ... \neg B_n)$$

où les A_i et B_j sont des formules atomiques, par sa représentation conditionnelle suivante :

$$A_1, A_2,... A_m \Leftarrow B_1, B_2,... B_n$$

les virgules dans la conclusion ($A_1,... A_m$) représentent des disjonctions des formules atomiques A_i alors que dans l'antécédent ($B_1,... B_n$) elles représentent des conjonctions des formules atomiques B_i, le quantificateur étant sous entendu, toutes les variables sont supposées être universellement quantifiées. Ceci se justifie par le fait que la formule :

$$(A_1 \vee... A_m \vee \neg B_1 \vee... \neg B_n)$$

est équivalente à la formule :

$$(A_1 \vee... A_m \vee \neg (B_1 \wedge... B_n))$$

elle même équivalente à la formule :

$$(A_1 \vee... A_m \Leftarrow (B_1 \wedge... B_n))$$

Pour simplifier l'utilisation des formules de la logique du premier ordre et plus particulièrement pour la déduction automatique de théorèmes, il est intéressant d'écrire les formules sous forme de clauses.

2.2 *Théorie sémantique*

La logique du premier ordre fournit des méthodes pour déduire les théorèmes d'une théorie. Les théorèmes peuvent être caractérisés comme étant les formules qui sont des conséquences logiques des axiomes de la théorie, et justifiés par les théorèmes de complétude de Gödel [Gödel-67]. L'interprétation sous-jacente suppose chaque théorème vrai. La seule règle utilisée dans les systèmes de Programmation Logique est la règle de résolution définie par Robinson [Robinson 65].

Supposons que l'on désire prouver que la formule suivante :

$$\{\exists(X_i)\ (B_1 \wedge... B_n)\}$$

est une conséquence logique d'un ensemble d'énoncés. Les systèmes actuels de résolution de preuve de théorèmes sont des systèmes basés sur une réfutation : on ajoute la négation de la formule à prouver aux axiomes et on cherche à obtenir une contradiction. Ainsi, la négation de la formule à prouver est une clause particulière dont la tête est vide, appelée un but, que nous pouvons écrire sous la forme :

$$\{\Leftarrow B_1, B_2... B_n\}$$

D'un point de vue démonstration de théorèmes, l'intérêt est de démontrer qu'une formule est une conséquence logique d'un ensemble d'énoncés. Cependant, l'utilisateur du système de réécriture s'intéresse surtout aux substitutions des variables (X_i) obtenus lors de la résolution du but, qui forme une réponse aux questions de l'utilisateur. Cependant, pour pouvoir effectuer des déductions il faut posséder un modèle d'interprétation construit à partir de l'ensemble des axiomes de la théorie.

Une interprétation permet de donner une valeur de vérité à toutes les formules. En particulier, l'interprétation donne une valeur de vérité à tous les termes. L'ensemble de ces valeurs correspondent au domaine de l'interprétation, la fonction donnant la valeur de vérité est une pré-interprétation du domaine donné. Nous pouvons la définir par :

Définition :

Une ***pré-interprétation*** J d'un langage du premier ordre L est constituée des éléments suivants :

- un domaine D,
- une assignation à toute constante de L d'un élément de D,
- une assignation à tout symbole fonctionnel n-aire de L d'une application de D^n dans D.

Exemple

Constantes = {0}, Fonctions = {s}, Relations = {plus},
J = {D={N}; {0 -> 0}, {s -> successeur}}.

Définition :

Une ***interprétation*** I d'un langage du premier ordre L est constituée d'une pré-interprétation J de L et de domaine D et de l'assignation à tout symbole de prédicat n-aire de L d'une application de D^n dans {Vrai, Faux}.

Exemple

Constantes = {0}, Fonctions = {s}, Relations = {plus},
J = {D={N}; {0 -> 0}, {s -> successeur}}.
I = {J; { ∀ X ∀ Y ∀ Z (plus(X, Y, Z) -> (Vrai si J(X) + J(Y) = J(Z); Faux sinon) }} où J(X) représente la pré-interprétation du terme X.

Définition :

Soient une pré-interprétation J de domaine D et L un langage du premier ordre. On appelle V une ***assignation de variables***, une application qui à chaque variable de L associe un élément du domaine D de J.

Définition :

Soient une pré-interprétation J de domaine D, L un langage du premier ordre et V une assignation de variables. L'***assignation de termes*** de L est définie de la façon suivante :

- l'assignation d'une variable est son assignation selon V,
- l'assignation d'une constante est son assignation selon J,
- si $a_1,..., a_n$ sont les assignations des termes $t_1,..., t_n$ et f' l'assignation selon J d'un symbole fontionnel n-aire f alors $f'(a_1,..., a_n)$ est l'assignation du terme $f(t_1,..., t_n)$.

Nous pouvons maintenant définir l'interprétation des formules à partie d'une pré-interprétation et de l'assignation des variables.

Définition :

Soient une interprétation I de domaine D, L un langage du premier ordre, V une assignation de variables. On peut donner à toute formule de L une ***valeur de vérité***, Vrai ou Faux, de la manière suivante :

- si la formule est $p(t_1,..., t_n)$ sa valeur de vérité est obtenue par la valeur de $p'(a_1,..., a_n)$, où p' est associé au prédicat p, a_i est associé au terme t_i relativement à l'interprétation I et l'assignation V.
- si la formule est l'une des formules ; $(\neg F)$, $(F \vee G)$, $(F \wedge G)$, $(F \Rightarrow G)$, $(F \Leftrightarrow G)$, la valeur de vérité de la formule est calculée par l'application des règles logiques obtenue à partir des valeurs de vérité de formules F et G.
- si la formule est $(\forall X F)$, sa valeur de vérité est Vraie si pour tous les objets d du domaine D, et si la formule F a la valeur de vérité Vrai relativement à l'interprétation I et l'assignation V donnant à X la valeur d et celles obtenues par V aux autres variables. Dans les autres cas, sa valeur de vérité est Faux.
- si la formule est $(\exists X F)$, la valeur de vérité de la formule est Vraie s'il existe un objet d du domaine D et si la formule F a la valeur de vérité Vrai relativement à l'interprétation I et l'assignation donnant à X la valeur d et celles obtenues par V aux autres variables. Dans les autres cas, sa valeur de vérité est Faux.

Remarque :

On dira qu'une formule F est vraie [resp fausse] dans l'interprétation I si sa valeur de vérité est vrai [resp faux]

Définition :

Soient une interprétation I pour un langage du premier ordre L et F une formule dans L.

- On dit que F est ***satisfiable*** dans I si $(\exists F)$ est une formule vraie dans l'interprétation I.

- On dit que F est ***valide*** dans I si ($\forall$ F) est une formule vraie dans l'interprétation I.

Remarque :

Nous pouvons donner les définitions complémentaires de celles-ci.

- On dit que F est ***non satisfiable*** dans I si ($\exists$ F) est une formule fausse dans l'interprétation I.

- On dit que F est ***non valide*** dans I si ($\forall$ F) est une formule fausse dans l'interprétation I.

Les axiomes d'une ***théorie du premier ordre*** représentent un sous-ensemble désigné de formules closes du langage du premier ordre permettant de décrire la théorie. En programmation logique, les axiomes sont représentés par les clauses de Horn. Á chaque théorie, il est possible de trouver une interprétation de telle sorte que chaque axiome soit vrai. Ces interprétations représente les modèles d'une théorie. Nous pouvons ainsi définir les modèles d'une théorie de la façon suivante :

Définition :

Soient une interprétation I pour un langage du premier ordre L et F une formule close de L. L'interprétation I est un ***modèle*** de F si la formule F est vraie relativement à I.

Définition :

Soient une théorie du premier ordre T et un langage du premier ordre L. Un ***modèle*** de T est une interprétation pour L qui est un modèle pour chaque axiome de T.

Si T admet un modèle, on dit que T est ***consistante***.

Définition :

Soient Σ un ensemble d'énoncés d'un langage du premier ordre L et I une interprétation de L. On dit que I est un ***modèle*** de Σ si I est un modèle pour chaque formule de Σ.

Cas particulier, soient $\Sigma = \{F_1, F_2, \ldots, F_n\}$ un ensemble de cardinal fini d'énoncés et I une interprétation, I est un modèle de Σ si et seulement si I est un modèle de l'énoncé $(F_1 \wedge F_2 \wedge \ldots \wedge F_n)$.

Définition :

Soit Σ un ensemble d'énoncés d'un langage du premier ordre L.

- On dit que Σ est ***satisfiable*** si L admet une interprétation qui est un modèle de Σ.

- On dit que Σ est ***valide*** si toute interprétation de L est un modèle de Σ.

Remarque :

Nous pouvons donner les définitions complémentaires de celles-ci.

- On dit que Σ est ***non satisfiable*** si aucune interprétation de L n'est un modèle de Σ.
- On dit que Σ est ***non valide*** si L admet une interprétation qui n'est pas un modèle de Σ.

Nous pouvons donner à ce niveau la définition importante de conséquence logique. Cette définition correspond aux formules qui sont des théorèmes d'une théorie. La définition est la suivante :

Définition :

Soit Σ un ensemble d'énoncés d'un langage du premier ordre L, soit F un énoncé de L. On dit que F est une ***conséquence logique*** de Σ si pour toute interprétation I de L, telle que tous les énoncés de Σ soient simultanément vrais, F est vraie dans l'interprétation I. On notera cette relation par :

$$\Sigma \models F$$

Si Σ est vide alors F est un énoncé valide, appelé tautologie et on a :

$$\models F$$

Exemple :

La formule $F = \{ p(X) \vee \neg p(X) \}$ est une tautologie.

Proposition :

Soit $\Sigma = \{F_1, F_2, ..., F_n\}$ un ensemble de cardinal fini d'énoncés. F est une conséquence logique de l'ensemble Σ si et seulement si la formule $\{(F_1 \wedge F_2 ... \wedge F_n) \Rightarrow F\}$ est une formule valide.

Proposition :

$\Sigma \models F$ si et seulement si $\Sigma \cup \{ \neg F \}$ est non satisfiable.
$\models F$ si et seulement si $\{ \neg F \}$ est non satisfiable.

Parmi l'ensemble de tous les modèles d'une théorie, certains d'entre eux possèdent des propriétés particulières. Ces modèles sont plus commodes pour démontrer des propriétés sur des formules (relativement à la théorie), telles que toutes propriétés vraies dans ces modèles sont vraies dans tous les modèles de la théorie. Définissons cette classe de modèles particuliers.

Définition :

Soit L un langage du premier ordre. On appelle H(L), ***l'univers de Herbrand***, tous les termes clos que l'on peut former à partir des symboles fonctionnels de L.

Exemple :

Dans le domaine formel des entiers naturels, nous avons :
H(L) = { 0, s(0), s(s(0))... }
l'univers de Herbrand représente une définition formelle de tous les entiers naturels.

Définition :

Soit L un langage du premier ordre. On appelle HB(L), la ***base de Herbrand***, toutes les formules atomiques closes que l'on peut former à partir des symboles relationnels de L et des termes de l'univers de Herbrand de L.

Exemple :

Dans le domaine formel des entiers naturels, doté seulement du symbole relationnel "plus", nous avons :
HB(L) = { plus(0,0,0), plus(s(0),0,s(0)), plus(0,0,s(0)), ... }

Définition :

Soit L un langage du premier ordre. Une ***pré-interprétation*** de Herbrand du langage L est une pré-interprétation de L telle que D = H(L), les constantes sont assignés à elles-mêmes, les symboles de fonctions de L sont assignés aux fonctions de mêmes symboles.

Définition :

Soit L un langage du premier ordre. Un ***interprétation de Herbrand*** de L est un interprétation de L basée sur une pré-interprétation de Herbrand de L.

Nous pouvons maintenant énoncer le principal théorème de Herbrand sur la conséquence logique.

Théorème de Herbrand :

Soit Σ un ensemble fini d'énoncés universellement quantifiés. Les trois affirmations suivantes sont équivalentes.

- Σ possède un modèle
- Σ possède un modèle dont la base est l'univers de Herbrand associé à Σ
- Le système de Herbrand, obtenu à partir des formules de Σ en remplaçant les variables par des éléments du domaine de Herbrand, possède un modèle lorsqu'on le considère comme un ensemble de formules du calcul propositionnel dont les formules atomiques sont les formules atomiques de la base de Herbrand. Ce modèle est appelé un modèle de Herbrand.

L'ensemble des modèles de Herbrand est donc un support intéressant pour la démonstration de certaines propriétés de la théorie du premier ordre. Cependant, il est à remarquer que seul le plus petit modèle de Herbrand présente un réel intérêt lorsqu'il existe, ce qui est le cas pour les programmes définis. En effet, ce plus petit modèle de Herbrand est l'intersection de tous les modèles de Herbrand et toute propriété démontrée à partir de ce modèle est aussi vraie pour tous les modèles d'un ensemble d'énoncés. Nous conseillons plus particulièrement de consulter les ouvrages [LLoyd-87, Delahaye-86] sur ces aspects de la théorie du premier ordre.

2.3 Unification

Comme nous l'avons vu précédemment, la résolution utilise un mécanisme de passage de paramètres par substitution des variables entre deux termes. Le principe de l'unification est de trouver une substitution d'un ensemble d'expressions telles que toutes les expressions substituées soient identiques. Le concept d'unification fut introduit en 1930 par Herbrand [Herbrand-31]. Il fut à nouveau découvert en 1963 par Robinson pour être exploité comme une technique de filtrage de la résolution, ce qui permit de réduire l'espace de recherche [Robinson-65]. Nous devons donc définir ce que sont les substitutions et l'unification.

2.3.1 Substitution

Une substitution est une application de l'ensemble des variables vers l'ensemble des termes. Elle permet de substituer un ensemble de variables dans des formules pour obtenir une autre formule. Nous pouvons définir ces applications de la façon suivante :

Définition :

Une ***substitution*** Θ est un ensemble de cardinal n de la forme $\{v_i \leftarrow t_i\}_n$ où chaque v_i est une variable, les v_i sont deux à deux distinctes, chaque t_i est un terme distinct de la variable v_i.

Nous pouvons utiliser des substitutions pour former de nouvelles formules à partir d'une formule initiale. Ces nouvelles formules ont la propriété d'être obtenues à partir d'une formule unique et permettent de définir une relation d'ordre partiel sur les formules. Cette relation d'ordre peut être définie de la façon suivante :

Définition :

Soient une substitution $\Theta = \{v_i \leftarrow t_i\}_n$ et E une expression. On appelle l'***instance*** de E par Θ l'expression $E\Theta$ obtenue à partir de E par remplacement de chaque occurrence des variables v_i apparaissant dans E par les termes t_i associés.

Exemple:

$E = p(X, s(Y)), \Theta = \{ X \leftarrow Y ; Y \leftarrow s(X) \}.$
$E\Theta = p(Y, s(s(X)))$

La relation d'ordre partiel que nous venons d'obtenir est transitive. Soient E, F et G trois formules de la logique du premier ordre, telle que F est une instance de E de substitution Θ, G est une instance de F de substitution σ. On peut alors affirmer que G est aussi une instance E de substitution $\tau=\Theta\sigma$. La substitution τ est appelé la composition des deux substitutions Θ et σ, car $E\Theta = F$, $F\sigma = G$, d'où $E\Theta\sigma = G = E\tau$.

Définition :

On appelle la ***composition de substitutions*** $\Theta\sigma$, la substitution obtenue à partir des substitutions Θ et σ telle que :
$\Theta = \{v_i \leftarrow t_i\}$
$\sigma = \{w_i \leftarrow u_i\}$
$\Theta' = \{v_i \leftarrow (t_i)\sigma\}$
$\sigma' = \sigma \setminus$ {les affectations aux v_i de Θ} % enlever les v_i de σ
$\Theta\sigma$ est la réunion de Θ'et σ'

Exemple :

$\Theta = \{X \leftarrow f(Y) ; Y \leftarrow Z\}$, $\sigma = \{X \leftarrow a ; Y \leftarrow b; Z \leftarrow Y\}$
$\Theta' = \{X \leftarrow f(b)\}$, $\sigma' = \{Z \leftarrow Y\}$
$\Theta\sigma = \{X \leftarrow f(b) ; Z \leftarrow Y\}$

Définition :

Soient E et F deux expressions. On dit que E est une ***instance*** de F s'il existe une substitution Θ telle que l'expression $F\Theta$ est identique à l'expression E.

Exemple :

$E = p(f(Y))$, $F = p(X)$, $\Theta = \{X \leftarrow f(Y)\}$

Définition :

Soient E et F deux expressions. On dit que E et F sont des ***variantes*** si E est une instance de F et si F est une instance de E.

Nous pouvons maintenant définir le concept important d'unification.

Définition :

Soient E et F deux expressions. On dit que E et F sont ***unifiables*** s'il existe une substitution Θ telle que ($E\Theta = F\Theta$). La substitution Θ est appelée ***unificateur*** de E et F.

Définition :

Soit Θ un unificateur de deux expressions E et F. La substitution Θ est dite l'***unificateur le plus général***, noté ***upg***, si :

$$E\Theta = F\Theta,$$

et

quel que soit un unificateur σ de E et F, $E\sigma$ est une instance de $E\Theta$.

Proposition :

Si deux expressions E et F sont unifiables alors elles admettent un upg et si σ et Θ sont deux upg de E et F, alors $E\sigma$ et $E\Theta$ sont des variantes.

2.3.2 Algorithme d'unification

On peut se poser la question de savoir si, pour deux formules données, il existe toujours un unificateur le plus général. Pour cela nous devons au préalable définir l'ensemble des différents de deux formules, puis l'algorithme construisant l'upg de deux formules.

Définition :

Soit Σ un ensemble d'expressions simples. L'***ensemble des différents*** de Σ est défini de la façon suivante : on repère la position du symbole le plus à gauche qui n'apparait pas dans toutes les expressions de Σ et on extrait de chaque expression de Σ la sous-expression commençant à la position de ce symbole. L'ensemble de toutes ces sous-expressions est l'ensemble des différents.

Nous présentons maintenant l'algorithme d'unification, dû à Robinson, qui permet de déterminer l'unificateur le plus général entre deux expressions [Robinson-65]. Dans cet algorithme, S est un ensemble fini d'expressions simples, l'upg σ_k représente l'unificateur le plus général de cet ensemble de formules.

(1) $k=0$, σ_k = vide

(2) si $S\sigma_k$ est un singleton, la substitution σ_k est un upg de S,
sinon déterminer D_k l'ensemble des différents de $S\sigma_k$.

(3) s'il existe v et t dans D_k tels que v soit une variable n'apparaissant pas dans t, $\sigma_{k+1} = \sigma_k + \{v \leftarrow t\}$,
sinon S n'est pas unifiable.

L'étude sommaire de cet algorithme d'unification nous montre qu'il est difficile de le réaliser ainsi. Les réalisations actuelles du langage Prolog utilisent un algorithme basé sur le suivant :

Entrée : X et Y à unifier,

Sortie : un upg Θ ou échec de l'unification.

Algorithme :

```
initialiser Θ à vide,
déposer sur la pile (X, Y)
échec = faux
tant-que pile non vide et non échec faire
        dépiler (X, Y)
        selon que
```

- X et Y sont des constantes identiques, continuer
- X est une variable n'apparaissant pas dans le terme Y,
substituer X par Y dans la pile
ajouter à Θ la substitution ($X \leftarrow Y$)
- Y est une variable n'apparaissant pas dans le terme X,
substituer Y par X dans la pile
ajouter à Θ la substitution ($Y \leftarrow X$)
- $X = f((X_i)_n)$ et $Y = f((Y_i)_n)$,
déposer (X_i, Y_i) dans la pile, pour tout i
- défaut : échec = vrai

fin selon que
fin tant que
si échec = vrai retourner échec d'unification
sinon retourner Θ, l'upg de X et Y Θ.

Ainsi, cet l'algorithme d'unification permet de trouver un unificateur le plus général de deux expressions lorsqu'il existe. Dans le cas contraire, il le mentionne par un échec.

2.4 *Théorie syntaxique*

Le problème de la démonstration automatique est de trouver une méthode permettant de déduire des théorèmes. Il faut donc posséder un algorithme de déduction sur un ensemble d'axiomes permettant de prouver si une proposition est vraie dans l'interprétation. Cette partie montre le lien entre la déduction et la conséquence logique.

Définition :

Soient Σ un ensemble d'énoncés, S un ensemble d'axiomes et de règles d'inférences, l'énoncé A se ***déduit*** de Σ ($\Sigma \vdash_S A$) s'il existe une suite de n formules $(B_i)_n$ telles que :

- Soit, B_i est un axiome,
- Soit, il existe j, k < i tels que B_i se déduit de B_j et B_k par l'application d'une règle d'inférence de S,
- Soit, B_i est un énoncé de Σ,
- Et $A = B_n$.

Exemple :

Soit un langage L d'unique constante "0" et d'unique symbole fonctionnel "s" d'arité 1. L'ensemble Σ se compose de tous les énoncés de la forme :
$F(0) \wedge \forall X\, [F(X) \Rightarrow F(s(X))] \Rightarrow \forall X\, F(X)$

Par exemple, si F(X) = "$\neg X \equiv 0 \Rightarrow \exists Y\, (X \equiv s(Y))$", où $\equiv$ représente l'égalité. Si S est l'ensemble des règles d'inférences de la logique, nous pouvons effectuer une déduction de la formule $\forall X\, F(X)$ de la façon suivante :

1 $0 \equiv 0$ % axiome d'identité
2 $0 \equiv 0 \Rightarrow (\neg\, 0 \equiv 0 \Rightarrow \exists Y\, (0 \equiv s(Y)))$ % tautologie
3 $F(s(0))$ % modus ponens
4 $s(X) \equiv s(X)$ % axiome d'identité
5 $s(X) \equiv s(X) \Rightarrow \exists Y\, (s(X) \equiv s(Y))$ % axiome
6 $\exists Y\, (s(X) \equiv s(Y)$ % modus ponens sur 4 et 5
7 $\exists Y\, (s(X) \equiv s(Y) \Rightarrow F(s(X))$ % tautologie
8 $F(s(X))$ % etc...
9 $F(s(X)) \Rightarrow (F(X) \Rightarrow F(s(X)))$
10 $F(X) \Rightarrow F(s(X)))$
11 $(F(X) \Rightarrow F(s(X))\,) \Rightarrow (0 \equiv 0 \Rightarrow (F(X) \Rightarrow F(s(X))\,))$
12 $0 \equiv 0 \Rightarrow (F(X) \Rightarrow F(s(X)))$
13 $0 \equiv 0 \Rightarrow \forall X\, [F(X) \Rightarrow F(s(X))]$
14 $\forall X\, [F(X) \Rightarrow F(s(X))]$
15 $\forall X\, [F(X) \Rightarrow F(s(X))] \Rightarrow (F(0) \Rightarrow F(0) \wedge \forall X\, [F(X) \Rightarrow F(s(X))])$
16 $F(0) \Rightarrow F(0) \wedge \forall X\, [F(X) \Rightarrow F(s(X))]$
17 $F(0) \wedge \forall X\, [F(X) \Rightarrow F(s(X))]$
18 $F(0) \wedge \forall X\, [F(X) \Rightarrow F(s(X))] \Rightarrow \forall X\, F(X)$
19 $\forall X\, F(X)$

La dernière ligne correspond à la formule, $\forall X\, [\neg\, X \equiv 0 \Rightarrow \exists Y\, (X \equiv s(Y))]$ qui lorsque l'on interprète les constantes dans N, les énoncés correspondent au principe de récurrence. La formule déduite par récurrence est alors : *"tout entier non nul est le successeur d'un entier"*.

Définition :

Soit Σ un ensemble d'énoncés. L'ensemble Σ est dit ***inconsistant*** si pour tout énoncé A, $\Sigma \vdash_S A$.

Autrement dit, Σ est dit inconsistant si on peut démontrer toutes les propositions.

Proposition :

Soient Σ un ensemble d'énoncés, A un énoncé.
$\Sigma \vdash_S A$ si et seulement si $\Sigma \cup \{\neg\, A\}$ est un ensemble d'énoncés inconsistants relativement à S.

Le problème que nous pouvons nous poser est de savoir lorsqu'un énoncé se déduit d'un ensemble d'énoncés, s'il est une conséquence logique de cet ensemble d'énoncés. Dans le cas où ceci est vrai, l'ensemble S possède une propriété particulière, il est dit correct.

Définition :

Un système S est dit ***correct*** si quels que soient Σ un ensemble d'énoncés, A un énoncé, on a :

$$\Sigma \vdash_S A \Rightarrow \Sigma \models A.$$

Définition :

Un système S est dit ***complet*** si quel que soit Σ un ensemble d'énoncés, A un énoncé, on a :

$$\Sigma \vdash_S A \Leftrightarrow \Sigma \models A.$$

Cette définition complète la précédente définition sur la déduction. Ainsi, une théorie logique est intéressante lorsqu'elle est complète. Il a été démontré que la logique du premier ordre est une théorie complète [Pabion-76]. Mais aussi que la logique du second ordre n'est pas une théorie complète. Ces résultats sont des conséquences directes des théorèmes de Gödel [Gödel-67, Hofstadter-85].

Maintenant intervient un problème de décidabilité :

"existe-t-il *une méthode pour prouver qu'un énoncé se déduit ou non d'un ensemble d'énoncés ?*"

Définition :

Soient Σ un ensemble d'énoncés, R un symbole relationnel n-aire et n termes $(T_i)_n$. On dit que R est ***décidable*** si et seulement si, il existe un programme P qui pour toute valeur de $(T_i)_n$ imprime :

- oui si $R((T_i)_n)$ est vrai.
- non si $R((T_i)_n)$ est faux.

On montre qu'il est impossible de proposer un algorithme général capable de décider, en un nombre fini d'opérations, de la validité ou la non validité de n'importe quelle formule de la logique du premier ordre. C'est pourquoi on dit que la logique du premier ordre est indécidable [Church-56].

Cependant, la logique du premier ordre est semi-décidable. La semi-décidabilité d'une théorie peut être définie par :

Définition :

Soient Σ un ensemble d'énoncés, R un symbole relationnel n-aire et n termes $(T_i)_n$. On dit que R est ***semi-décidable*** si et seulement si, il existe un programme P qui pour toute valeur de $(T_i)_n$ imprime

- oui si $R((T_i)_n)$ est vrai.
- non ou tourne indéfiniment si $R((T_i)_n)$ est faux.

Cela revient à dire que si un énoncé est valide, l'application de l'algorithme sur cet énoncé s'arrête au bout d'un nombre fini, non borné, de déductions permettant de conclure que ladite formule est valide.

2.5 Programmes définis

Nous devons maintenant caractériser un tel algorithme de déduction. Cet algorithme repose sur un ensemble de règles d'inférences particulier. De plus, on choisit un sous ensemble de la logique du premier ordre comme langage, l'ensemble des clauses de Horn.

2.5.1 Définitions

Définition :

Une ***clause de Horn défini*** est une clause de la logique du premier ordre dont la conséquence est une formule atomique.

Exemple :

la clause $\{A \Leftarrow B_1, B_2,... B_n\}$ est une clause de Horn.
la clause $\{A_1, A_2 \Leftarrow B_1, B_2,... B_n\}$ n'est pas une clause de Horn.

Définitions :

Soit une clause de Horn $\{A \Leftarrow B\}$
A est appelée la ***tête*** de la clause,
B est appelée la ***queue*** de la clause,
une clause de Horn $\{A \Leftarrow\}$ est appelée un ***fait*** ou clause unité.

La sémantique déclarative de la clause $\{A \Leftarrow B\}$ est :

"quelle *que soit l'instance des variables de cette clause, si chaque énoncé de B est vrai alors l'énoncé A est aussi vrai"*

De même la clause $\{A \Leftarrow\}$ à pour sémantique :

"quelque *soit l'instance des variables de ce fait, l'énoncé A est vrai"*

ce qui peut aussi se traduire par la clause de Horn $\{A \Leftarrow \text{vrai}\}$, où vraie représente la formule atomique toujours vraie, équivalente à la clause de Horn $\{\text{vrai} \Leftarrow\}$.

Nous pouvons maintenant définir la classe de programmes correspondant à de la programmation logique pure. On les appelle les programmes définis.

Définition :

On appelle ***Programme défini*** un ensemble fini de clauses de Horn définies.

Définition :

Un ***but*** est une clause de Horn de la forme $\{\Leftarrow B_1, B_2,... B_n\}$. C'est-à-dire une clause dont la conséquence, la tête, est vide.

Il existe une notion importante permettant de définir une partition de l'ensemble des prédicats d'un programmes, que l'on appelle le paquets de clauses qui regroupe l'ensemble des clauses relatives à un même prédicat.

Définition :

La ***définition d'un symbole de prédicat*** p d'un programme défini P est l'ensemble de toutes les clauses de Horn définies du programme P dont les têtes sont de symbole de prédicat p.

De même, lorsque l'on pose une question à un système Prolog, celui-ci fournit des substitutions correspondant à une réponse au problème posé. Il existe plusieurs types de substitutions, seules sont importantes celles produites par la résolution, car elles correspondent à une réponse correcte.

Définition :

Soient P un programme défini et B un but. On appelle ***réponse*** pour $P \cup \{B\}$ toute substitution des variables de B.

Définition :

Soient P un programme défini, B un but $\{\Leftarrow A_1 \wedge A_2 \ldots \wedge A_n\}$, Θ une réponse pour $P \cup \{B\}$. On appelle Θ une ***réponse correcte*** pour $P \cup \{B\}$ si la formule $\forall((A_1 \wedge A_2 \ldots \wedge A_n)\Theta)$ est une conséquence logique de P. Ce qui revient à dire :

$$P \models \{(A_1 \wedge A_2 \ldots \wedge A_n)\Theta\}$$

Ainsi, pour pouvoir fournir une réponse correcte à un problème nous devons posséder une algorithme permettant de fournir une réponse *calculée* qui sera aussi une réponse correcte.

2.5.2 Résolution SLD

Une résolution SLD est fondée sur une démonstration par l'absurde, appelée réfutation. Elle permet de démontrer qu'un but est une conséquence logique d'un programme. L'origine de la réfutation appliquée à la démonstration automatique de théorèmes est due à Robinson, qui lui à donné le nom de ***principe de résolution*** [Robinson-65].

Cependant, il existe plusieurs technique de réfutation plus ou moins efficace et dans le cas de la programmation logique seule la SL-résolution est importante et plus particulièrement la SLD-résolution s'appliquant sur les programmes définis.

Définition :

Soient P un ensemble de clauses de Horn, N un but de la forme $\{\Leftarrow B_1 \wedge B_2 \ldots \wedge B_n\}$. On appelle ***SLD-dérivation*** de $(P \cup \{N\})$, l'ensemble des trois suites : (N_i), (C_i) et (Θ_i) telles que :

N_i est un but,

Si N_i est non vide, $N_i = \{\Leftarrow B_1 \wedge B_2 \ldots \wedge B_m\}$ $m \geq 1$

alors

C_i est une variante d'une clause de P,

$C_i = \{A \Leftarrow A_1 \wedge A_2 \ldots \wedge A_n\}\ n \geq 0$
Θ_i est un unificateur de A et B_k
$N_{i+1} = \{(\Leftarrow B_1 \ldots \wedge B_{k-1} \wedge A_1 \ldots \wedge A_n \wedge B_{k+1} \ldots \wedge B_m)\Theta_i\}$
Si N_i est vide, la dérivation est de longueur i.

Il est à remarquer que les SLD-dérivations peuvent être finies ou infinies, et elle peuvent échouer ou réussir. Une SLD-dérivation qui réussie est une SLD-dérivation qui est de longueur finie se terminant par une clause vide (N_i est vide). Une SLD-dérivation qui échoue est une SLD-dérivation finie dont la dernière clause est non vide (il est impossible de trouver un Θ_i qui unifie une tête de clause et un littéral choisi de la clause). Ainsi, les dérivation finie sont importante, et plus particulièrement celle qui réussissent.

Définition :

On appelle ***SLD-réfutation*** toute SLD-dérivation de longueur finie qui réussi.

Nous pouvons nous demander quel est le rapport de la SLD-réfutation, permettant de calculer une suite de substitutions, avec la notion de conséquence logique donnant l'ensemble de réponses correctes. Le théorème suivant nous apporte la réponse.

Théorème :

Soient P un ensemble de clauses de Horn un but de la forme $\{\Leftarrow B_1 \wedge B_2 \ldots \wedge B_n\}$. Il existe une SLD-réfutation de $P \cup \{\Leftarrow B_1 \wedge B_2 \ldots \wedge B_n\}$ si et seulement si l'ensemble d'énoncés $(P \cup \{\Leftarrow B_1 \wedge B_2 \ldots \wedge B_n\})$ est inconsistant.

Voir [Lloyd-87] pour la démonstration de ce théorème.

Ainsi, nous possédons une méthode pour démontrer qu'un énoncé est une conséquence logique d'un ensemble d'énoncés. Il nous faut maintenant un algorithme de dérivation qui permet de trouver une SLD-réfutation lorsqu'il en existe une. Nous devons pour cela faire un choix sur la construction de la réfutation. Ces choix sont les suivants :

- Choix du littéral parmi les littéraux du but,
- Choix d'une clause lorsqu'il en existe plusieurs s'unifiant avec le littéral,
- Choix d'un unificateur.

Lors d'une recherche d'une réfutation, des dérivations sont construites jusqu'à obtenir une clause vide. L'ensemble des dérivations construites lors de la procédure de recherche d'une SLD-réfutation représente l'espace de recherche, ou ***arbre SLD***.

D'une façon générale, lors de la construction de l'arbre de recherche, on peut choisir une substitution quelconque pour l'unification du terme choisi dans le but avec la tête de la clause considérée. Cependant, seul le choix d'un unificateur le plus général permet de trouver une SLD-réfutation minimale, et ainsi de réduire l'espace de recherche.

Définition :

On appelle une ***SLD-résolution*** un algorithme de parcours de l'espace de recherche d'une SLD-dérivation.

Définition :

On appelle ***retour-arrière*** d'une SLD-résolution une modification du choix de la clause lors de la dérivation. On choisit une autre clause, la suivante dans l'ordre de déclaration, qui s'unifie avec la formule atomique.

Définition :

Soit P un ensemble de clauses de Horn. L'***ensemble des succès*** de P est l'ensemble de tous les buts B tels que $P \cup B$ possède une SLD-réfutation.

Nous pouvons maintenant donner la définition de réponse calculée, correspondant à la production d'une solution par un système Prolog.

Définition :

Soient P un programme défini, B un but. Une ***réponse calculée*** pour $P \cup B$ est la substitution $\Pi(\Theta_i)_n$ telle que $(\Theta_i)_n$ est la suite des unificateurs les plus généraux d'une SLD-réfutation de $P \cup B$.

Exemple :

Sur l'exemple de programme Prolog donné en 1.2, pour le but initial *"enfant(jeanne, Parent)"*, nous avons les deux substitutions calculées :

$$\Theta_1=\{\text{Parent} \leftarrow \text{lucien}\}$$
$$\Theta_2=\{\text{Parent} \leftarrow \text{berthe}\}$$

Maintenant nous pouvons énoncer le théorème important faisant le lien entre la notion de réponse correcte et le calcul d'une solution par une SLD-résolution.

Théorème :

Soient P un programme défini, B un but. Toute réponse calculée pour $P \cup B$ est une ***réponse correcte*** pour $P \cup B$.

Lorsque l'on pose une question à une système Prolog, celui-ci peut fournir plusieurs solutions calculées, correspondant à toutes les réponses correctes de la formule logique. La collection de toutes ces réponses calculées nous donne un ensemble de solutions à un but. Nous pouvons définir cet ensemble de la façon suivante.

Définition :

Soient P un programme défini, B un but. On appelle Es(B) ***l'ensemble des solutions*** du but B, l'ensemble de toutes les solutions correctes du but B.

En pratique, il est plus intéressant de considérer une solution comme étant les valeurs des arguments d'une formule atomique. Ainsi, soit $B = f(t_1 \ldots t_n)$, l'ensemble des solutions de B peut être défini par :

$$Es(B) = \{(a_1 \ldots a_n) \,/\, \exists\Theta \text{ tel que } (a_1 \ldots a_n) = (t_1 \ldots t_n)\Theta \text{ et } \Theta \text{ est une solution correcte pour } B\}$$

Cette définition contient toutes les réponses possibles pour une formule. Ainsi, lorsqu'il existe une solution multiple, cette solution apparaîtra autant de fois dans l'ensemble de solutions. De même, lorsqu'il n'existe aucune solution pour la formule, l'ensemble de solutions est vide et cela correspond à ce que l'on nomme un échec de la SLD-résolution.

2.6 Programmes normaux

Il est parfois intéressant d'utiliser des négations de formules dans la queue d'une clause, ce qui permet d'utiliser des connaissances négatives. Les programmes définis ne permettent pas d'en tenir compte. De plus, il faut rajouter une règle d'inférence particulière permettant d'exprimer que l'on peut déduire la négation d'une formule. Pour cela, il faut définir quelques notions élémentaires permettant d'exprimer cette règle d'inférence. Cette partie définit les programmes normaux. Nous introduirons ensuite les éléments nécessaires pour définir une règle d'inférence de la négation.

Définition :

Une ***clause normale*** est une clause de la forme $\{A \Leftarrow L_1, L_2, \ldots L_n\}$ où les L_i sont des littéraux et A une formule atomique.

Définition :

Un ***programme normal*** est un ensemble fini de clauses normales.

Définition :

Un ***but normal*** est une clause de la forme $\{\Leftarrow L_1, L_2, \ldots L_n\}$. C'est-à-dire une clause normale dont la conséquence (tête) est vide.

Définition :

La ***définition d'un symbole de prédicat*** p d'un programme normal P, est l'ensemble de toutes les clauses normales du programme P dont les têtes sont de symbole fonctionnel p.

Nous pouvons reprendre les définitions correspondant aux réponses pour les programmes normaux.

Définition :

Soient P un programme normal et B un but normal. On appelle ***réponse*** pour $P \cup \{B\}$ toute substitution des variables de B.

Définition :

Soient P un programme normal, B un but normal de la forme $\{\Leftarrow L_1, L_2, \ldots L_n\}$, Θ une réponse pour $P \cup \{B\}$. On dit que Θ est ***réponse correcte*** pour $P \cup \{B\}$ si la formule $\forall((L_1, L_2, \ldots L_n)\Theta)$ est une conséquence logique de P. Ce qui revient à dire :

$$P \models \{(L_1, L_2, \ldots L_n)\Theta\}$$

2.7 La négation

La négation par l'échec a été étudiée comme une extension intéressante de la programmation logique. En effet, le programmeur désire parfois exprimer une information négative dans ses programmes. L'utilisation de la SLD-résolution ne permet pas de pouvoir déduire de l'information négative. En d'autre termes, on ne peut prouver que la négation d'un but est une conséquence logique d'un programme car il n'existe aucune règle d'inférence exprimant une négation.

Prenons un exemple afin d'illustrer ce problème, considérons le programme :

```
élève(jean).
élève(paul).
élève(éric).
professeur(christian).
```

Au vu de cette déclaration, il est relativement simple de trouver la sémantique logique de ce programme. Ainsi, pour le programmeur, *"christian"* n'est pas un élève car c'est le professeur. Et il serait souhaitable que la résolution puisse aussi le déduire par le fait que la proposition "¬ *élève(christian)"* soit vraie dans l'interprétation sous-jacente du programme. Cependant, il faut remarquer que *"élève(christian)"* n'est pas une conséquence logique de ce programme. De même, "¬ *élève(christian)"* n'est pas non plus une conséquence logique du programme. Aucune règle ne permet de pouvoir déduire ce fait.

Ainsi, pour pouvoir effectuer une telle déduction, il est nécessaire de définir une règle d'inférence particulière. Dans la littérature, il existe deux types de règle définissant la négation. Celle de la négation par l'échec [Clark-78] et celle que j'appellerai la négation close [Reiter-78].

2.7.1 Le monde clos

Afin de pouvoir donner un sens logique à la négation, nous devons faire une hypothèse importante : *"le monde est clos"*. C'est-à-dire que l'on fait l'hypothèse que chaque déclaration de prédicat est une déclaration complète et qu'elle peut se ramener à une équivalence. Ainsi, si nous reprenons l'exemple précédent, nous pouvons avoir la déclaration complète du prédicat élève :

$$\forall X\ (\text{élève}(X) \Leftrightarrow (X=\text{jean}) \vee (X=\text{paul}) \vee (X=\text{éric}))$$

Ensuite, il suffit de rajouter les axiomes de l'égalité (=) afin d'obtenir une complétude de la définition. Enfin, nous pouvons maintenant déduire "¬ *élève(christian)*", par le fait que "*christian*" est différent de "*jean*" et de "*paul*" et de "*éric*".

Ce procédé de complétion est une autre manière de formaliser l'idée que l'information, qui n'est pas donnée par le programme, est considérée comme fausse. Alors le concept d'une réponse correcte de la résolution peut être étendu à cette propriété en disant que toute réponse est correcte si le but auquel on applique la substitution correspondant à la réponse est une conséquence logique de la complétion du programme.

Une autre conséquence de cette hypothèse est que l'ensemble des solutions d'un but doit être fini.

2.7.2 Complétion d'un programme

Dans cette partie nous définissons la transformation d'un programme normal en son programme complet. Considérons un programme normal P, et soit p un symbole de prédicat d'arité n. La déclaration du prédicat p est un ensemble de clauses dont une est de la forme :

$$p(T_1, T_2 \ldots T_n) \Leftarrow L_1, L_2 \ldots L_m.$$

où les L_i sont des littéraux quelconques. Considérons le symbole de prédicat =/2, dont l'interprétation sous-jacente est l'égalité. Nous pouvons alors transformer cette clause sous sa forme canonique correspondant à l'énoncé :

$$\forall (X_i)_n\ \{p((X_i)_n) \Leftarrow (X_i=T_i)_n, L_1, L_2 \ldots L_m\}$$

Où les X_i sont des variables nouvelles dans la "clause". Si nous appelons $(Y_i)_k$, les k variables de la clause originelle que nous avons considérée, nous obtenons l'énoncé suivant :

$$\forall (X_i)_n\ \{p((X_i)_n) \Leftarrow \exists (Y_i)_k, (X_i=T_i)_n, L_1, L_2 \ldots L_m\}$$

Si nous appelons E_i la queue de cette "clause",

$$E_i = (\exists (Y_i)_k, ((X_i=T_i)_n, L_1, L_2 \ldots L_m))$$

nous pouvons effectuer une telle transformation sur les r clauses de la définition du prédicat p, et nous obtenons la déclaration complète du prédicat p :

$$\forall (X_i)_n\ \{p((X_i)_n) \Leftarrow E_1\}$$

$$\forall (X_i)_n\ \{p((X_i)_n) \Leftarrow E_2\}$$

...

$$\forall (X_i)_n \{p((X_i)_n) \Leftarrow E_r\}$$

Enfin, la définition complète du prédicat p est l'énoncé complet suivant :

$$\forall (X_i)_n \{p((X_i)) \Leftrightarrow E_1 \vee E_2 \vee ... E_r\}$$

De plus, afin d'obtenir une définition complète du programme, il faut ajouter une déclaration implicite d'échec pour tout symbole de prédicat n'étant pas défini par le programme. Ainsi, pour tous les prédicats se trouvant dans une queue de clause et dont il n'existe pas de définition dans le programme on applique cette règle d'inférence : *"il y a échec implicite de ces prédicats"*.

Définition :

La ***complétion*** d'un programme normal P, notée complet(P), est la collection de toutes les définitions complètes des symboles de prédicats et des axiomes de la théorie de l'égalité.

Définition :

Soient P un programme et B un but normal. On dit que Θ est une ***réponse*** pour B relativement à P si c'est une substitution de l'ensemble des variables de B.

Définition :

Soient P un programme, B un but normal et Θ une réponse pour B. On dit que Θ est une ***réponse correcte*** pour B relativement à la complétion de P, si l'énoncé $\forall((B)\Theta)$ est une conséquence logique de la complétion de P.

Cette définition, qui généralise la définition précédente de réponse correcte, fournit la description déclarative d'une solution d'un but normal et d'un programme normal.

Définition :

Soient P un programme, B un but normal. Un ***arbre SLD normal*** de B relativement à complet(P) est un arbre ET/OU vérifiant les conditions suivantes :

- la racine de l'arbre est le but B,
- l'arbre est fini,
- chaque nœud de l'arbre est un but normal appelé sous-but de B,
- chaque branche ET est la conjonction de sous-buts normaux correspondant aux prémisses d'une clause de P dont la tête s'unifie avec le nœud supérieur.
- chaque branche OU correspond au choix d'un unificateur du but associé au but courant.

Définition :

Soient P un programme, et B un but normal. Une ***SLD-résolution normale*** est un parcours de l'arbre SLD-normal associé au but B et relativement au programme P.

2.7.3 La négation par l'absurde

La réalisation d'une négation correcte suppose de pouvoir démontrer que la résolution d'un but est finie. C'est-à-dire que l'arbre de résolution est de profondeur finie. Ainsi, il nous est nécessaire de pouvoir détecter une résolution non finie. Dans une partie suivante, nous présentons une méthode originale de détection de boucles en Prolog basée sur la caractérisation de boucles régulières. Ainsi, un méta-interprète muni de cette détection permet de pouvoir réaliser une négation.

Pour obtenir une arbre SLD normal il faut s'assurer que cet arbre est de profondeur fini. Or la négation par l'absurde ne le permet pas. Il faut donc faire une hypothèse sur les programmes pour le vérifier. Une méthode intéressante pour garantir cette contrainte sur l'arbre est de le vérifier lors de la résolution, c'est-à-dire réaliser une détection de boucles. Ceci est possible que pour une certaine classe de programme.

Pour réaliser une détection de boucles efficace (ce qui n'est pas le cas pour tous les programme), il faut que les dérivation soit régulières, et donc faire une hypothèse d'une base de donnée complète (l'ensemble des constantes est fini). Dans ce cas, la négation par l'absurde est compatible avec cette hypothèse, il est possible de démontrer qu'un but possède une solution ou non. Ainsi, nous possédons une négation complète par utilisation d'une négation par l'absurde et d'un détecteur de boucles (dans le cas d'une boucle, la résolution le signale et s'arrête).

2.8 La coupure

La coupure, ou coupe-choix, est pour Prolog une facilité de contrôle de la résolution largement utilisée par les programmeur. Le coupe-choix s'écrit "/" pour Prolog-2 ou "!" pour les Prolog écossais. Son effet est de modifier le déroulement séquentiel du parcours de l'arbre SLD de recherche. Nous utilisons dans cette partie et dans toute la suite de l'exposé, la syntaxe écossaise pour exprimer les clauses d'un programme Prolog [CProlog-1.5].

La sémantique du coupe-choix est de provoquer un échec lors d'un retour-arrière sur le but, appelé but parent, s'unifiant avec la tête de la clause contenant le coupe choix. Ainsi, lorsqu'un retour-arrière se produit sur un coupe-choix, le système arrête de chercher dans le sous-arbre qui à pour racine le but parent. Le coupe-choix consiste alors à élaguer tous les choix dans la sous-arborescence du but parent.

Considérons maintenant un programme contenant une clause avec un coupe-choix et pouvant se réécrire sous la forme suivante : "*but :- but*$_1$ *, ! , but*$_2$", où "*but*$_1$" est la conjonction des buts avant le coupe-choix, "*but*$_2$" le reste de la queue de clause. Nous pouvons définir la sémantique du coupe-choix de la façon suivante :

- le coupe-choix réussit immédiatement lors de l'appel.

- lors du retour arrière sur le coupe-choix, son effet est de provoquer un échec sur l'appel du but parent *but*.

Dans notre exemple, le but parent du coupe-choix est *but*, ce but échoue lors d'un retour arrière sur ce coupe-choix. Ainsi, à la sémantique du coupe-choix est associé un effet de bord non négligeable qui est le suivant : le coupe-choix permet d'éliminer les éventuelles nouvelles solutions, par des points de choix sur les deux buts but_1 et but, après le retour-arrière sur le coupe-choix. On peut comparer cet effet avec le "Go-To" de certains langages de programmation, car on modifie le déroulement séquentiel de l'interprétation d'un but.

Cet effet de bord de l'interprétation du coupe-choix peut provoquer une incomplétude de la résolution. En effet, si dans la sous-arborescence élaguée par le coupe-choix il n'existe pas de solution alors l'usage du coupe-choix est sûr propice à l'optimisation, parcours d'une branche sans succès ou avec une dérivation infinie. En revanche, si une branche contenant un succès est élaguée par le coupe-choix, l'usage est dit dangereux car il peut éliminer une solution correcte de la SLD-résolution. L'usage sûr, dit "*cut vert*", est utile pour améliorer l'efficacité sans omettre de réponses. L'usage dangereux du coupe-choix, dit "*cut rouge*", est néfaste car dans certains cas il élimine des solutions correctes. Par exemple, prenons le programme suivant :

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

il existe une SLD-réfutation du but "*p(a)*" mais un échec pour produire la substitution X=a lors de la résolution du but "*p(X)*". Ainsi, un usage non sur de la coupure fait perdre la sémantique déclarative du programme.

3. Mise au point

3.1 Les problèmes

La mise au point d'un programme quelconque consiste à déterminer la position d'une erreur dans le programme, relativement à l'interprétation que s'en fait le programmeur. Cette interprétation est toujours associée à un langage. Le débogage est un problème essentiel dans la mise au point d'un programme. Aussi, le développement d'une application sera-t-il favorisé par la disponibilité d'outils de mise au point pour son langage.

La plupart des environnements de développement en langage de haut niveau proposent des outils de mise au point plus ou moins sophistiqués. La littérature décrit des solutions particulières pour un certain nombre de langages. Zellweger définit un ensemble de méthodes pour la mise au point interactive de langages compilés et optimisés [Zellweger-84]. De même, Murray décrit la réalisation d'une méthode automatique de mise au point pour un système éducatif du langage Lisp [Murray-86]...

La réalisation d'applications en Prolog nécessite de la part du programmeur une méthodologie différente décèle classique, utilisée en programmation procédurale. Le caractère non déterministe de la résolution, un des aspects importants de la programmation logique, permet d'obtenir des ensembles de solutions possibles à un problème donné à partir de l'interprétation sous-jacente (le modèle de Herbrand dans le cas de Prolog pur).

La recherche d'une solution à un problème, ou résolution d'un but, consiste à obtenir une substitution correcte du but. Elle est fondée sur le parcours en profondeur d'un arbre *ET/OU*. Chaque branche *ET* est une conjonction de buts et chaque branche *OU* est une disjonction sur un but.

Lors de la réalisation d'une application Prolog, le programmeur construit de façon virtuelle, en utilisant une représentation sous forme de clauses de Horn, un ensemble d'arbres *ET/OU* pour chaque problème ou sous-problème à résoudre. La réunion de ces arbres constitue un programme Prolog. Dans une phase de mise au point, le programmeur vérifie les ensembles de solutions de son programme au moyen de tests caractéristiques.

Tout programmeur désire savoir si ses programmes sont corrects au sens où il l'entend. D'une façon générale nous pouvons donner les deux définitions suivantes sur les différents cas d'erreurs d'un programme Prolog.

Définition-1 :

Un programme Prolog est dit ***correct*** si tous les ensembles de solutions fournis par la résolution sont corrects selon l'interprétation attendue du programmeur. C'est-à-dire si toutes les substitutions correctes fournies par la résolution sont des réponses correctes selon l'interprétation attendue du programmeur.

Définition-2 :

Un programme Prolog est dit ***incorrect*** s'il existe un ensemble de solutions fournies par la résolution qui est non correct selon l'interprétation attendue du programmeur. C'est-à-dire s'il existe au moins une substitution correcte fournie par la résolution mais qui n'est pas une réponse correcte selon l'interprétation attendue du programmeur

L'environnement de programmation d'un langage évolué du type Prolog se doit de posséder un ensemble d'outils de mise au point permettant de caractériser un programme incorrect. Ces outils doivent fournir des renseignements lors de la résolution d'un but relativement à un programme. Il le peuvent d'un point de vue statique, en permettant d'effectuer une analyse syntaxique, éventuellement sémantique, ou bien de fournir l'ensemble des solutions d'un but Prolog. D'un point de vue dynamique, il le peuvent également par l'utilisation d'un outil de trace fournissant une représentation de diverses étapes : l'appel d'un but, l'instance des variables, les retours sur un but, ou d'autres informations, lors du parcours de l'arbre de résolution.

Lorsqu'un programmeur désire réaliser une mise au point en Prolog, il utilise généralement la trace standard définie par Byrd. Elle traduit, sous forme de messages, les principaux états de l'interprète : *appel, réussite, retour, échec* [Byrd-80]. Cependant, une telle approche pose parfois des problèmes pour trouver une cause associée à un symptôme d'erreur. L'un des plus importants est le fait que le programmeur d'une application, s'il veut détecter des erreurs dans ses programmes, doit connaître la sémantique opérationnelle de Prolog, ce qui complique parfois l'interprétation d'une résolution pour connaître les ensembles de solutions obtenus. D'autres méthodes de trace en Prolog [Boizumault-84, Eisenstadt-85, Ducassé-88] permettent de donner des informations complémentaires sur la résolution. Ainsi, Eisenstadt définit une technique de trace partielle (Zooming) permettant d'affiner la recherche d'erreurs par une technique de trace en largeur d'abord. Ducassé fournit à l'utilisateur un système modulaire, OPIUM+, permettant de réaliser différentes méthodes de mise au point fondées sur la trace de la résolution. D'autre part, Mellish ou Looi & Roos proposent des méthodes automatiques de correction de programme Prolog pour un système intelligent d'enseignement [Mellish-81, Looi-Roos-87].

Une autre technique de mise au point, fondée sur l'interprétation du programmeur, a été présentée dans [Shapiro-83]. L'utilisateur du système exprime uniquement la sémantique déclarative de son programme, la partie opérationnelle étant effectuée par le détecteur d'erreurs. D'autres auteurs ont réalisé des méthodes de mise au point similaires [Av Ron-84, Pereira-85,88, LLoyd-86]. L'ensemble de ces réalisations fournissent des méthodes déclaratives de mise au point pour les deux cas *solution fausse* et *absence de solution*. Dans l'article [Pereira-88] les auteurs définissent une méthode rationnelle de mise au point pour les programmes généraux qui n'utilisent pas de prédicats prédéfinis avec effet de bord (*assert* par exemple). Ainsi, l'utilisateur peut effectuer une mise au point de ses programmes lorsqu'il utilise une coupure et/ou une négation sûre. Ce problème est aussi étudié dans le cadre de Prolog parallèle [Huntbach-87, Shapiro-88]. Enfin, les auteurs de la méthode déductive [Dershowitz-Lee-87] s'inspirent de la méthode de Shapiro afin de réaliser un détecteur d'erreurs déductif, basé sur la vérification des contraintes associées à un programme. Le programmeur fournit des assertions que doivent vérifier les solutions des buts et spécifie des invariants sur ses clauses que le détecteur d'erreurs vérifie en cours de résolution.

3.2 Qu'est-ce-qu'une mise au point déclarative

3.2.1 Cas d'erreurs en Prolog

Lors d'une phase de mise au point, le programmeur d'une application effectue une série de tests caractéristiques permettant de déterminer si son programme est correct. Lorsque le programme est incorrect, il existe dans l'application une erreur de conception qui peut être caractérisée par un symptôme d'erreur. Compte tenu des caractéristiques de la programmation logique nous pouvons définir un ensemble de symptômes d'erreur possible :

Définition :

Soient P un programme Prolog, B un but, Θ une substitution correcte de B. La substitution Θ est dite ***solution fausse*** du but B, si c'est une substitution fausse selon l'interprétation attendue de B relativement au programme P.

Autrement dit, une solution est fausse si elle est calculée et incluse dans le complémentaire de l'ensemble des solutions attendues du programme.

Définition :

Soient P un programme Prolog, B un but, Θ une substitution correcte de B. L'arbre de preuve de la formule (B)Θ est dit ***arbre faux*** s'il existe un nœud de cet arbre dont le but associé possède une solution fausse.

Autrement dit, lorsque un arbre de résolution est un arbre faux, il existe un symptôme d'erreur permettant d'affirmer qu'un programme est incorrect selon l'interprétation attendue du programmeur.

Définition :

Soient P un programme Prolog, B un but, Θ une substitution de B correcte selon l'interprétation attendue de B relativement au programme P. La substitution Θ est dite ***solution insatisfaite*** si la résolution du but B ne permet pas de démontrer que (B)Θ est une conséquence logique du programme P.

Autrement dit, une solution est insatisfaite si elle est non calculable et incluse dans l'ensemble des solutions attendues du programme.

Définition :

Soient P un programme Prolog, B un but, Θ une substitution correcte de B. La substitution Θ est dite ***solution insuffisante*** si le nombre de réussites de la résolution de B fournissant l'upg Θ est incorrect selon l'interprétation attendue de B relativement au programme P.

Lorsque l'on désire tester un programme afin de savoir s'il contient une erreur, il faut caractériser un symptôme d'erreur. Nous venons de voir les principales erreurs produisant un ensemble de solutions incorrect dans l'interprétation attendue. À partir de ces définitions nous pouvons les réunir pour former un seul critère d'erreur.

Définition :

Soient P un programme Prolog, B un but, Θ une substitution de B. La substitution Θ est dite ***solution incorrecte*** pour le but B si c'est soit une solution fausse, soit une solution insatisfaite, soit une solution insuffisante. Dans le cas contraire, la substitution Θ est dite ***solution juste*** dans l'interprétation attendue de B relativement au programme P.

Maintenant, nous pouvons définir ce qu'est un symptôme d'erreur dans un programme Prolog. Ce symptôme d'erreur permet d'affirmer qu'un programme est incorrect relativement à l'interprétation attendue.

Définition :

Soient P un programme Prolog, B un but. La résolution du but B est dite ***résolution incorrecte*** s'il existe un nœud de l'arbre de résolution dont le but associé possède une solution incorrecte.

Autrement dit, un programme sera considéré comme incorrect s'il existe un but particulier tel que sa résolution soit une résolution incorrecte. De même, on considère comme correct tout programme dont on ne trouve pas de but possédant une résolution incorrecte.

Pour pouvoir prouver qu'un programme est correct, c'est-à-dire faire la preuve de la sémantique attendue de ce programme, il faudrait mettre en œuvre une méthode constructive de buts tests permettant de prouver que toutes les résolutions sont correctes. Cette tache est impossible car on ne peut construire tous les buts permettant de prouver cette sémantique attendue.

L'avantage de cette approche, qui associe symptôme d'erreur et résolution incorrecte, est que l'on n'a pas besoin de construire tous les buts pour tester un programme. Il suffit de définir un jeux de tests particuliers permettant de construire les arbres de résolution qui serviront pour détecter un symptôme d'erreur dans le programme.

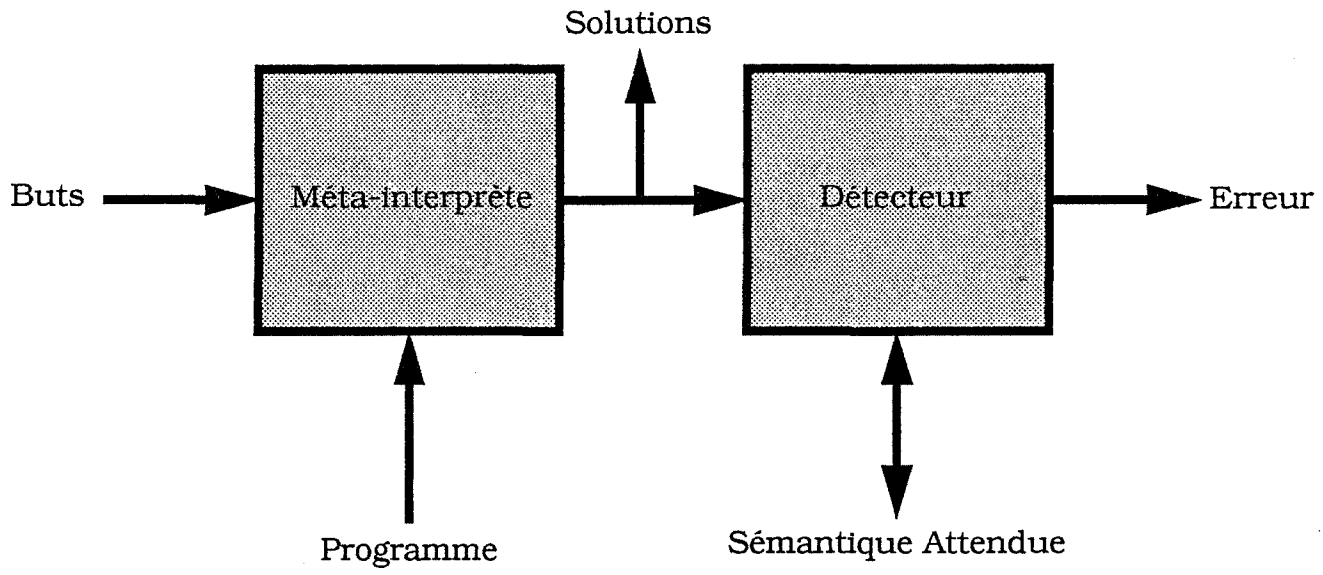
3.2.2 Techniques de recherche déclaratives

D'une façon générale, un système de mise au point pour un langage donné est constitué d'une partie centrale, le système de détection d'erreurs qui simule l'exécution du langage, mais aussi d'un analyseur fournissant une cause probable de l'erreur. À partir de la définition que nous avons donnée sur un symptôme d'erreur, nous pouvons définir l'algorithme de détection d'erreurs.

Définition :

Soient P un programme Prolog et B un but possédant une résolution incorrecte. On appelle ***débogueur Prolog*** un algorithme de parcours de l'arbre de recherche du but B associé au programme P permettant de trouver une solution incorrecte.

Ainsi, un débogueur du langage Prolog est constitué de deux composantes : un simulateur de la résolution permettant de construire et de parcourir l'arbre de recherche et un système de détection d'erreurs caractérisant une solution incorrecte. Ce système peut être décrit par le schéma :



Définition :

Soient P un programme Prolog et B un but Prolog. On appelle ***méta-interprète*** un programme permettant de simuler la résolution du but B associé au programme P.

Définition :

Soient P un programme Prolog et B un but Prolog. On appelle ***détecteur d'erreurs*** un algorithme permettant de caractériser que le but B possède une solution incorrecte relativement à la sémantique attendue du programme P.

Lorsqu'un programmeur réalise une application Prolog, il écrit un programme **P** correspondant à un domaine logique. D'une façon générale, il existe une interprétation calculée du programme, le modèle **M(P)**, qui est l'ensemble des solutions correctes que calcule la résolution et que l'on appelle la sémantique actuelle du programme P. Cependant, au monde auquel le programmeur se réfère, son domaine d'application, est associé une certaine théorie **T**, modélisée par une interprétation **M** qui fournit une valeur de vérité à toutes les formules atomiques que l'on appelle la sémantique attendue du programme.

La mise au point des programmes Prolog consiste à faire coïncider les deux modèles du même domaine d'application, autrement dit un programme Prolog est correct si les deux sémantiques sont identiques, c'est-à-dire si M(P) = M. En fait, pour le programmeur seule l'inclusion d'un sous ensemble de M(P) dans la sémantique attendue est nécessaire pour que le programme soit considéré comme correct. En effet, le programmeur con-

naît généralement le domaine d'application et donc la sémantique attendue associée, alors qu'il n'utilise qu'un sous-ensemble de cette sémantique pour réaliser son application. Mais aussi, on peut valider le programme en ne donnant qu'un certain nombre de buts, les buts tests, qui caractérise le programme.

Plus généralement, lorsque l'on réalise une application, le programmeur s'intéresse à l'intersection de deux sémantiques. Si cette intersection est suffisamment riche pour que tous les résultats obtenus aient une interprétation attendue, le programme sera considéré comme "validé". En revanche, lorsqu'une solution calculée n'est pas incluse dans la sémantique attendue associée au programme, on doit considérer le programme comme étant incorrect. Ainsi, nous pouvons énoncer les théorèmes correspondant à la notion de programme correct ou incorrect (c'est un abus de langage, on devrait dire validé) :

Théorèmes :

Soit P un programme Prolog.

Le programme P est correct si et seulement si pour tout but Prolog l'ensemble des solutions calculées est inclus dans la sémantique attendue associée au programme P.

Le programme P est incorrect si et seulement si il existe un but Prolog admettant un ensemble de solutions calculées qui n'est pas inclus dans la sémantique attendue associée au programme P.

Démonstrations

Triviales, par l'application des définitions correspondantes.

Dans ces conditions, l'objectif de tout système de mise au point en Prolog consiste à déterminer un sous-modèle d'un programme P, par la résolution d'un certain nombre de buts Prolog. Lorsque le système de détection d'erreurs trouve une solution incorrecte lors de la résolution d'un but, il signale la non inclusion de ce sous-modèle par rapport à la sémantique attendue associée au programme P. Autrement dit, lorsqu'il trouve un ensemble de solutions de M(P) non inclus dans M.

Ainsi, le support du détecteur d'erreurs Prolog est la sémantique attendue associée au programme. La technique la plus simple est de connaître cette sémantique attendue au préalable par un ensemble d'axiomes permettant d'évaluer une solution calculée (calculer sa valeur de vérité). C'est le principe de la méthode déductive [Dershowitz-Lee-87] qui peut être comparée à celle des assertions de certains langages, en particulier les assertions du langage Eiffel [Meyer-88]. Une autre méthode est l'utilisation d'une description formelle du modèle attendu du programme (c'est la sémantique attendue par le programmeur) afin de réaliser une détection d'erreurs. Cette méthode est utilisée dans [Drabent-all-88].

Réaliser une telle base de connaissances contenant toutes les solutions de la sémantique attendue pose un problème au programmeur. En fait, celui-ci doit écrire des spécifications formelles donnant la valeur de vérité des solutions d'un prédicat, sous forme d'assertions. Le système de mise au point se contente de vérifier ces assertions lors de la recherche d'erreurs. Lorsqu'il trouve une incohérence, parce qu'une valeur de vé-

rité est différente de celle calculée, il détecte un symptôme d'erreur. Mais spécifier de telles assertions ne garantit plus la complétude du système d'assertions [Gödel-30]. En effet, il faut pouvoir disposer d'un système permettant de vérifier de telles assertions, car le programmeur peut désirer savoir si elles sont correctes ou non. Nous retombons sur le problème d'un système de détection d'erreurs, mais cette fois-ci pour le système formel d'un système de détection d'erreurs sur des programmes Prolog, ce qui explique l'incomplétude de la méthode envisagée.

D'une façon générale, seul un sous-ensemble est nécessaire pour détecter des solutions incorrectes. La technique la plus simple est que le système de détection d'erreurs demande la sémantique attendue du programmeur pour déterminer si une solution calculée est ou non incluse dans cette interprétation.

Dans tous les cas, assertions évaluables et questions, l'algorithme de détection d'erreurs est alors simple :

```
Parcourir l'arbre de résolution,
sur chaque but trouvé,
    questionner le programmeur pour connaître l'interprétation attendue
    si la solution calculée est correcte (incluse)
        continuer
    sinon on vient de trouver un symptôme d'erreur.
```

De plus, pour améliorer la méthode de détection, le système peut mettre en œuvre une technique d'apprentissage des solutions correctes lorsqu'il pose les questions à l'utilisateur. Le système construit lui-même les assertions dont il a besoin à partir d'informations qu'il récolte auprès du programmeur. Ainsi, lorsque le système connaît une solution correcte, dans l'interprétation attendue, il la conserve pour pouvoir la réutiliser ultérieurement, soit pour la détection d'erreur courante, soit pour une détection d'erreur sur d'autres programmes.

Une autre technique envisageable est d'utiliser un programme que l'on sait correct. Puis d'utiliser les solutions calculées par ce programme comme spécifications formelles d'un autre programme plus astucieux mais que l'on veut tester. À ce sujet deux approches sont possibles. Dans la première, les solutions sont préalablement calculées pour être utilisées lorsque le système de détection d'erreurs le désire. Dans la seconde, les solutions sont évaluées à partir du programme correct par le système de détection d'erreurs pour vérifier une solution calculée par le programme que l'on cherche à tester.

La seconde approche nous semble préférable, car si elle est lourde en temps d'exécution, elle évite cependant de stocker toutes les solutions possibles et elle teste toutes les solutions calculées par le programme.

3.3 Conclusion

La littérature nous fournit une multitude de méthodes de mise au point en Prolog. Pour un utilisateur quelconque, son choix est difficile. Lorsque celui-ci ne dispose que de la trace standard, il l'utilise sans pouvoir obtenir les souplesses d'une méthode intelligente de mise au point.

En revanche, si nous sommes capables de proposer une interface particulière pour un système intelligent, que nous pourrons adapter à ses besoins propres (machine d'accueil, système Prolog, interface utilisateur), l'utilisateur pourra obtenir des outils performants de mise au point.

Aussi, dans la partie suivante et dans cet objectif, nous allons parcourir les principales méthodes de mise au point en Prolog. Cette étude nous permettra de mettre en place les bases d'un système de mise au point intelligent pour répondre aux divers cas d'erreurs en Prolog que nous venons de définir.

Enfin, la dernière partie sera consacrée à une réalisation particulière du système de mise au point, par l'utilisation des méthodes de détection d'erreurs. À partir de ce système, nous présentons quelques sessions de mise au point mettant en œuvre l'utilisation d'assertions obtenues par apprentissage, par des questions posées à l'utilisateur et des assertions évaluables à partir d'un programme.

Partie 2

État de l'art

4. Les traces

4.1 Principe

La disposition d'un système de trace constitue un élément important d'un environnement de programmation d'un langage interprété. En effet, la trace de l'exécution d'un programme permet de visualiser les différentes étapes de l'interprétation. Dans la plupart des langages interprétés, comme Apl ou Lisp, le modèle classique d'appel-retour sert de base aux outils de traces. De même pour les langages compilés tels Pascal et C, l'utilisation d'un système de détection d'erreurs fondé sur une technique de trace permet d'effectuer la mise au point des programmes. Cette technique de recherche d'erreurs repose sur le principe de découpage du programme pour réduire la partie du programme aux instructions qui peuvent avoir eu une influence sur les valeurs erronées des variables du programme.

Cependant, pour le langage Prolog la situation est plus complexe compte-tenu de la sémantique non déterministe de ce langage. La trace doit pouvoir permettre de suivre l'évolution des différents mécanismes de la résolution, c'est-à-dire le parcours en profondeur d'abord de l'arbre de résolution et le mécanisme d'unification. Un système de trace pour Prolog doit permettre de suivre les appels et les réussites sur un but, ainsi que les différents retour-arrière de l'interprète sur les points de choix (le choix d'une clause, la réussite d'une clause...). De même, la trace doit permettre à un programmeur de répondre à différentes questions qu'il peut se poser lors de la mise au point d'un programme. Ces questions peuvent être les suivantes :

- pourquoi y-a-t-il un échec lors de l'appel d'un but ?
- pourquoi y-a-t-il une réussite lors de l'appel d'un but ?
- quelles sont les clauses d'un paquet qui sont appelées lors de la résolution d'un but ?
- quelles sont les instances des variables d'appel lors de l'activation d'une clause ?
- quelles sont les instances des variables lors de la réussite du but ?
- lors d'une trace, à quel choix de clause se rapporte un message ?

Dans le contexte particulier d'une résolution incorrecte, ces questions correspondent à la recherche du bon découpage du programme pour déterminer un symptôme d'erreur de la résolution d'un but mais aussi la cause de l'erreur.

Les différents modèles de traces envisagés tentent de donner une aide au programmeur dans le découpage de son programme.

Afin de modéliser le principe d'une trace, nous présentons la sémantique d'une résolution Prolog. Puis nous présentons la trace classique de Prolog, celle de Byrd, ainsi que deux approches différentes, la trace de Eisenstadt et celle de Boizumault.

4.2 Sémantique de la résolution

Nous présentons ici une sémantique de la résolution d'un but Prolog. Notre dessein n'est que de fournir un modèle afin de décrire les traces que nous allons présenter et cette sémantique est proche de la sémantique de la SLD-résolution. De plus, elle fait intervenir des notions d'unification de termes Prolog [Robinson-65] d'échec, de réussite et de retour-arrière, notions utiles pour interpréter une trace.

Soient $P = \{ C_i \}$ un programme Prolog, c'est-à-dire un ensemble fini de clauses de Horn C_i, et *But* un but Prolog à résoudre.

Définition :

Soient F un symbole de prédicat et N un entier. On appelle *P(F,N) le paquet de clauses* du programme P défini comme l'ensemble des clauses de P dont la tête de clause ait F comme symbole de prédicat et soit d'arité N.

Définition :

La résolution d'un but peut être définie par l'algorithme suivant :

Si le but à résoudre est de symbole de prédicat F et d'arité N alors
Soit P(F, N) le paquet correspondant à ce but.
Si le paquet P(F, N) est vide
Alors il y a *échec* du but
Sinon
    retirer la première clause de P(F, N)
    Si la tête de cette clause ne *s'unifie* pas avec le but
    Alors aller au test du paquet vide
    Sinon résoudre la queue de la clause
        Si la queue de cette clause *échoue*
        Alors aller au test du paquet vide
        Sinon il y a *réussite* du but

La notion de retour-arrière correspond toujours à une recherche d'une nouvelle solution suite à la réussite d'un but. Ce retour peut être dû au retour sur le but initial afin de trouver une nouvelle solution, ou bien faire suite à l'échec d'un sous-but de la résolution.

Soit C une clause de Horn du programme P. Cette clause que nous écrirons sous la forme :

$$C = \{\text{Tête :- } B_1, B_2, \ldots B_n\}$$

Si la résolution d'un but conduit à la réussite de la clause C, lorsqu'il y a retour-arrière sur le but on effectue un retour-arrière sur la queue de cette clause. Ce retour-arrière consiste à faire un retour-arrière sur le dernier but de la queue de clause, le but B_n. Cette technique permet de trouver une nouvelle réussite pour la queue de la clause. qui est alors une nouvelle solution pour le but s'unifiant avec la tête de la clause.

La résolution de la queue de clause C correspond à la résolution du but "B_1, B_2,...B_n" et consiste à résoudre chacuns de ces buts l'un après l'autre. Lorsque l'un de ces buts échoue, il y a retour sur le but précédent dans l'ordre de réussite.

Définition :

On appelle résolution d'un ensemble de but {B_1, B_2,...B_n}, avec comme valeur initiale 1 pour l'indice i, l'algorithme suivant :

Si la résolution du but B_i réussit
Alors
    Si i < n
        résoudre le but B_{i+1}
    Sinon réussite de {B_1, B_2,...B_n}
Sinon /* B_i échoue */
    Si i > 1
        retour-arrière sur la résolution du but B_{i-1}
    Sinon échec de {B_1, B_2,...B_n}

Définition :

On appelle trace d'une résolution, un algorithme de résolution d'un ensemble de buts, qui matérialise l'évolution de la résolution par l'écriture de messages.

Ainsi, lorsque nous voulons une trace exhaustive sur la résolution d'un ensemble de buts, cette trace doit pouvoir indiquer l'état courant de l'interprète. Ces différents états correspondent à certains points particuliers de l'interprète lors de la résolution. Ces points peuvent être : l'appel d'un but, l'unification d'un but avec une tête de clause, la réussite d'une clause, l'échec d'une clause, le retour-arrière sur-un but.

Les différents modèles de trace que nous présentons consistent à illustrer la résolution par des messages lorsque l'algorithme de résolution passe par certains de ces états.

4.3 Trace de Byrd

La plupart des systèmes de programmation en Prolog sont équipés d'une trace définie par Byrd en 1980. Celle-ci est fondée sur un modèle de boîte noire [Byrd-80]. Ce modèle de trace repose sur l'interprétation opérationnelle de la logique des clauses de Horn. Chaque clause d'un programme peut être interprétée de la façon suivante :

- la tête de clause est une procédure possédant des paramètres,
- la queue de clause est l'appel à des sous-programmes,
- l'invocation d'un sous-programme est l'Appel, (Call),
- la réussite de l'appel de tous les sous-programmes est la Sortie, (Exit),

- l'échec d'un des sous-programmes est l'Échec, (Fail),
- le retour-arrière sur un point de choix d'un sous-programme (évaluation d'une branche OU), c'est-à-dire la recherche d'une nouvelle solution, est le Retour (Redo, Back-To).

Le modèle correspond au parcours en profondeur d'abord de l'arbre ET-OU de la SLD-résolution. Les messages *Appel* et *Sortie* correspondent au parcours-avant de l'arbre, les messages *Retour* et *Echec* correspondent à un parcours-arrière de cet arbre lors du retour-arrière de l'interprète.

4.3.1 La trace

Nous pouvons définir l'ensemble des messages de la trace de Byrd de la façon suivante :

Appel — l'appel d'un but à démontrer, le but courant. Lorsqu'un but *"p(X, Y)"* est invoqué afin d'être résolu, le contrôle active le port *"Appel"*. Ce message correspond au début de l'algorithme de résolution.

Sortie — la réussite de l'appel d'un but. Ce message signale qu'une clause s'est unifiée avec le but courant et la réussite de la queue de cette clause.

Retour — le retour-arrière afin de rechercher une nouvelle solution du but courant. Ce message correspond au retour sur le but.

Echec — l'échec final du but courant. Ce message indique que le paquet de clause est vide, soit initialement, soit après érosion successive, ou alors qu'il ne contient pas de clauses s'unifiant avec le but courant.

Prenons un exemple de programme afin d'illustrer cette trace.

```
a(A) :-
    b,
    c.

a(A) :-
    write(encore). % écriture du terme "encore"

b.

b.

c.
```

La résolution du but *"a(A)"* permet d'illustrer quelques mécanismes d'une résolution SLD. La résolution du but *"a(A)"* avec la trace de Byrd nous donne l'ensemble des messages suivants :

```
| ?- trace, a(A).    % trace et appel de la résolution
    Call: a(_351) ? % appel initial
    Call: b ?       % appel du premier sous-but
    Exit: b ?       % ah, une réussite
    Call: c ?       % le second sous-but
    Exit: c ?       % ah, une réussite
    Exit: a(_351) ? % ah, une réussite
A = _351 ;          % OK, une autre solution ?
    Redo: a(_351) ? % Retour sur le but initial
    Redo: c ?       % Retour sur le dernier but ?
    Fail: c ?       % Echec, une autre clause ?
    Redo: b ?       % Retour par échec du but frère c
    Exit: b ?       % seconde réussite après retour
    Call: c ?       % nouvel appel de c
    Exit: c ?       % on continue...
    Exit: a(_351) ?
A = _351            % nouvelle réussite...
```

Ce petit exemple de résolution d'un but associé à un programme permet de faire quelques remarques la trace de Byrd. Aucun des messages ne permet d'identifier clairement quelques mécanismes importants de la résolution comme, le choix de la clause courante, l'unification, le retour-arrière... De plus, utilisée seule, cette trace conduit à un flux important de messages. Il est donc important de réduire le nombre de ces messages afin d'obtenir une trace efficace par un mécanisme de trace partielle.

4.3.2 Extensions

Les réalisations actuelles des systèmes Prolog font évoluer cette trace afin de donner plus d'informations sur la résolution.

Par exemple, pour la non définition d'un prédicat, le système IF-Prolog permet d'arrêter la résolution par un mécanisme d'exceptions équivalent aux exceptions du langage Ada. Le système Quintus-Prolog signale cette non définition de prédicat par un message d'alerte "Warning". D'autres systèmes utilisent un message de trace supplémentaire pour différencier l'échec d'unification *"Echec"* et l'échec par non définition *"Inconnu"*. D'autres extensions ou modifications de la trace sont envisageables, comme par exemple l'extension Coda [Plummer-87] ou l'extension Opium [Ducassé-86].

De même, pour réduire le flux d'une trace, il est possible d'effectuer une trace sélective de la résolution d'un but, par l'utilisation du prédicat *"spy/1"*. Ce prédicat permet de tracer l'exécution d'un paquet de clauses quelconque, chaque message de trace étant imprimé quand le prédicat est espionné. Son utilisation est cependant difficile pour le programmeur d'une grande application car celui-ci doit *deviner* quels prédicats il doit tracer. Cet aspect rend la recherche d'erreurs difficile car si l'on veut réduire le nombre de messages pour qu'une trace soit intéressante, il faut pouvoir estimer où mettre les points de trace, ce qui est pratiquement impossible.

4.3.3 Remarques

L'exemple relativement simple de programme que nous avons choisi, ne permet pas de montrer les différents mécanismes d'une trace de la résolution Prolog. Cependant, nous pouvons effectuer quelques remarques sur les limites d'une telle trace.

Dans les trois cas d'échec suivants : l'échec après un coupe-choix, l'échec sur l'unification, l'échec sur la définition d'un prédicat, le message de trace est identique : *"Fail"* (Échec). Ce message ne permet pas par une simple lecture de les différencier.

De même, ce modèle ne permet pas de différencier les retour-arrière effectués par l'interprète. À la suite d'un échec, on cherche une nouvelle unification, à la suite d'une réussite, on cherche une nouvelle solution.

Ainsi, lorsqu'un programmeur Prolog utilise une telle trace dans la résolution d'un but d'une application importante, rapidement celui-ci ne sait plus où en est la résolution. Nous pouvons faire les remarques suivantes :

1- les quatre messages de la trace ne permettent pas de faire une discrimination rapide des différents problèmes pouvant être rencontrés lors de la mise au point d'un programme (échec incorrect d'un but, solution fausse, retour-arrière oublié...). Ceci est dû essentiellement à l'importance des messages de la trace. L'utilisateur doit avoir une attention importante sur la trace pour trouver le message *utile*. Ainsi, un utilisateur est rapidement découragé par l'inefficacité d'une telle trace sur des applications concrètes.

2- on ne dispose d'aucune information sur l'absence d'unification d'une clause lors d'un appel. En effet, il peut y avoir plusieurs types d'erreurs d'unification : le prédicat n'existe pas, il y a une mauvaise arité, aucune clause ne s'unifie, plus d'autres unifications possibles. Tous ces types d'échec de la résolution sont caractérisés par le même message *"Échec"*, ce qui rend l'interprétation de la trace difficile et répondre à la question : *"pourquoi y-a-t-il échec ?"* est complexe pour l'utilisateur de la trace.

3- on manque d'information sur l'activation d'une clause. En effet, lorsqu'une clause est appelée, seul un message sur l'appel de son premier sous-but est donné. Ainsi, lorsque plusieurs clauses peuvent s'unifier avec un but, il n'existe aucune information sur le choix de la clause permettant de connaître simplement quelle est la clause en cours d'évaluation lors d'un appel, d'un retour... Ceci est d'autant plus gênant lorsque les clauses sont des faits, il est alors difficile de savoir lequel réussit.

4- on a aucune information sur l'instance des variables lorsqu'une clause particulière est choisie par la résolution, après unification. Seule la visualisation du but est effectuée lors d'une *"Sortie"*. Il est difficile de répondre simplement à la question : *"quelles sont les instances des variables lors de l'activation d'une clause ?"*

5- les noms des variables locales sont abscons : par exemple "*_65644*", ce qui ne permet pas une lecture rapide des liens de variables entre la tête et la queue de la clause appelée, ainsi que lors du déroulement de la trace. Le parcours partiel de l'arbre de résolution de façon visuelle est rendue complexe.

6- il faudrait un indentation permettant de visualiser les différents niveaux d'appels lors d'une résolution. Seule une utilisation complexe de numéros associés aux divers messages, permet de reconnaître un niveau d'appel. Répondre aux questions : "*à quelle clause se rapporte ce message ?*" et "*quel est la suite des appels de buts ?*" est difficile, pour un utilisateur non averti. Celui-ci ne peut construire un sous-arbre de la résolution pour l'analyser.

7- on se heurte à la difficulté de retrouver l'origine de l'appel lors d'un retour-arrière, car il faut parcourir la trace à l'envers pour déterminer le point de reprise. De même, sur l'exemple précédent, les trois messages de retour "*Redo*" ont une signification distincte, ce qui complique de façon sérieuse la lecture arrière de la trace pour retrouver sur quel but on effectue un retour.

8- il manque des informations sur l'appel des prédicats systèmes. Ceci peut être important lorsque l'on modifie le déroulement de la résolution, par exemple lors d'un coupe-choix "!". De même pour l'appel de prédicats pouvant provoquer des échecs. Cet aspect devient important lorsque l'on utilise des prédicats définis par l'utilisateur, compilés et se comportant comme des prédicats systèmes : "*pourquoi y-a-t-il échec ou retour ?*".

4.3.4 Commentaires

Nous pouvons faire quelques commentaires sur la trace de Byrd. Considérons un programmeur Prolog devant réaliser une application particulière.

Soit un programmeur devant réaliser un système expert en Prolog. La démarche de développement de son application pourrait être la suivante :

- écriture d'un moteur d'inférence.
- réalisation d'une maquette de son système expert.
- réalisation en grandeur nature de l'application (extension du moteur d'inférence, extension des connaissances...).

Le développement d'une telle application pose des problèmes de divers types. La réalisation du moteur d'inférence fait appel à des notions ne faisant pas partie de la logique du premier ordre. Sa mise au point complique sérieusement le travail du programmeur. En effet, il doit tester sur des exemples précis son moteur : il réalise une pré-maquette de son application afin de valider les choix de l'algorithme du moteur d'inférence, de la représentation des connaissances... La maquette proprement dite est là pour valider les différents modèles utilisés par le système, le moteur, les connaissances, etc.

La mise au point complète de l'application, aux diverses phases du développement, nécessite des outils de trace sophistiqués. Lorsque sur une question le système expert se comporte d'une façon imparfaite ou inattendue, est-ce le moteur d'inférences, la représentation des connaissances ou autre chose qui est en cause ? Pour le programmeur d'une telle application, quel doit être son comportement afin d'analyser la résolution Prolog ?

La trace de Byrd telle quelle est disponible, ne permet pas de réduire le flot de messages en fonction des désirs de l'utilisateur. Devant une quantité importante de messages d'une trace, le programmeur est souvent découragé et n'utilise pas la trace comme outil de mise au point. Celui-ci réalise lui même sa mise au point, par des raisonnements logiques sur son programme.

Le programmeur simule ainsi un détecteur d'erreurs intelligent, par l'utilisation de l'ensemble des connaissances qu'il possède sur son domaine d'application et sur la réussite d'un but, l'échec d'un but, les solutions d'un but... La trace ne lui sert qu'à préciser ses idées sur la mise au point ou sur une partie de la résolution. Cette trace devient un support de son raisonnement pour vérifier tel ou tel point de la résolution. Toute la démarche de mise au point en Prolog est ainsi réalisée par le programmeur.

Il est donc important de définir et fournir à un utilisateur de Prolog, un ensemble d'outils intelligents, spécifiques aux problèmes à résoudre, permettant de traiter les diverses raisons d'erreurs d'un programme Prolog. Ainsi, nous devons définir un modèle du raisonnement logique du programmeur lors de la mise au point d'un de ses programmes. Ce modèle nous permettra de simuler le comportement du programmeur sur la résolution d'un but Prolog afin d'obtenir un système de détection d'erreurs.

4.4 Trace de Boizumault

Boizumault propose dans son article [Boizumault-84] un modèle de trace proche de la sémantique de l'interprète Prolog. Ce modèle est sensiblement le même que celui de Byrd dans son principe de fonctionnement. Cette trace permet de visualiser l'appel d'un but, l'essai d'unification du but avec une clause, la réussite d'unification, ainsi que l'échec.

A chaque prédicat, défini par le programme considéré, est associé un paquet de clauses, chaque clause du paquet est scindée en deux, la tête et la queue. Les messages de contrôle sont au nombre de quatre : *"appel"* (l'appel d'un but), *"échec"* (l'échec d'un but), *"unifie"* (l'unification d'un but), *"prouve"* (la preuve d'un but), et correspondent comme la trace de Byrd à l'illustration de la résolution.

4.4.1 La trace

Nous pouvons décrire l'ensemble des différents messages de la façon suivante :

Appel — C'est l'appel initial d'une procédure, sous la forme d'un but à démontrer. Si le paquet correspondant à la procédure est non vide, le contrôle continue vers la porte *"Unifie"*. Dans le cas contraire, le contrôle se dirige vers la porte *"Échec"*.

Unifie — Ce message indique les différentes tentatives d'unification du but courant à partir de la première tête jusqu'à la dernière. Si l'unification s'effectue alors le contrôle continue sur la porte *"Prouve"* produisant l'appel de la résolution sur la queue de clause, sinon il y a passage à la tête suivante. Lorsqu'il n'y a plus de tête à unifier alors le contrôle se dirige vers la porte *"Échec"*.

Prouve — Ce message indique la demande de résolution sur une queue de clause dont la tête s'est unifiée avec le but courant. Lorsqu'un échec se produit soit sur l'appel d'un sous-but soit par retour-arrière, le contrôle retourne à l'état précédent de *"Unifie"* afin de choisir une nouvelle clause.

Échec — Ce message indique un échec du but courant après l'essai de toutes les solutions possibles d'unification de tête, ou alors après le retour-arrière sur un coupe-choix.

Sur l'exemple précédent de programme nous avons la trace suivante :

```
?- a(X).
Appel : a(_309)      % Appel initial du but
    Unifie 1 : a(_309)% la première clause
    Prouve : b , c  % OK, on appelle la queue de clause
    Appel : b       % exécution
        Unifie 1 : b
        Prouve : true
    Appel : c
        Unifie 1 : c
        Prouve : true
Yes
X = _309 ;
    Echec : c
        Unifie 2 : b
        Prouve : true
    Appel : c
        Unifie 1 : c
        Prouve : true
Yes
X = _309 ;
    Echec : c
    Echec : b
```

```
    Unifie 2 : a
        Prouve : write(encore)
encore Yes
X = _309 ;
Echec : a
```

Ce petit exemple démontre la meilleure lisibilité de cette trace. Cependant quelques problèmes persistent.

4.4.2 Commentaires

Cette trace apporte quelques informations supplémentaires, intéressantes sur le principe. Mais lors d'une utilisation en grandeur réelle sur de gros programmes, par exemple lorsqu'un prédicat possède un grand nombre de clauses de définition dont une seule s'unifie, on obtient tous les messages "*Unifie*" pour une seule réussite. Cette trace est donc difficilement utilisable sur des applications de grande importance.

4.5 Trace de Eisenstadt

Le modèle de trace de Eisenstadt est un modèle différent de trace de programme Prolog. Pour plus de détails on se référera à l'article original [Eisenstadt-85]. Eisenstadt constate le manque d'efficacité de la trace de Byrd et propose un modèle de trace beaucoup plus exhaustif. Elle repose sur un parcours en largeur d'abord de l'arbre de résolution. Cette technique, appelé *"Zooming"*, se rapproche des méthodes de mise au point utilisées pour des langages compilés tels que le langage C [Ducassé-Emden-87].

Lorsqu'un programmeur désire effectuer une détection d'erreurs en Prolog, il est intéressant de lui fournir une technique permettant d'effectuer un découpage de la résolution, afin de réduire le nombre des prédicats suspects. Un parcours en largeur d'abord permet de ne tracer que la partie qu'intéresse l'utilisateur. C'est le principe du Zooming.

L'interprétation d'un but Prolog peut être décrite en terme de satisfaction de buts, par une recherche séquentielle dans la base de données, qu'est l'ensemble des clauses, afin de trouver une clause dont la tête s'unifie avec le but courant, ensuite de satisfaire les sous-buts composant la queue de clause. Les différents cas de satisfactions pour un but sont les suivants :

- réussite du but car c'est un fait,
- réussite du but car tous ses sous-buts réussissent,
- échec du but,
- échec du but par retour-arrière sur le coupe-choix, produisant un échec sur le but père,
- aucune clause de la base de donnée ne s'unifie avec le but courant,
- l'arité du but courant n'est pas bonne,

- aucune définition d'un prédicat de même symbole que celui du but courant.

La trace de Eisenstadt utilise un ensemble de symboles permettant d'identifier et de discerner ces différents cas. Les symboles utilisés ont été sélectionnés afin d'être brefs, d'avoir un aspect mnémonique, ainsi que d'être discriminatoires. Les principaux symboles sont les suivants :

?	nouveau but à résoudre, (le but courant),
> [n]	unification du but courant avec la n-ième clause et instance des variables,
< [n]	échec de la n-ième clause,
+ [n]	réussite du but courant par réussite de la queue de clause,
+* [n]	réussite de la n-ième clause et c'est un fait,
-	échec du but courant,
^ [n]	retour-arrière sur la n-ième clause,
--!	appel du coupe-choix,
^^!	retour-arrière sur le coupe-choix,
-!	échec du but courant après retour-arrière sur le coupe-choix.

Les autres messages correspondent aux traitements des prédicats pré-définis et aux autres cas d'échecs.

4.5.1 Exemple de trace

Reprenons comme exemple le programme précédent pour illustrer la différence avec la trace de Byrd et de Boizumault. Nous obtenons la trace suivante :

```
?- a(A).
1   :   ?   a(_357)
2   :   >   a(_357) [1]
3   :       ?  b
4   :       +* b [1]
5   :       ?  c
6   :       +* c [1]
7   :   +   a(_357) [1]
Yes
A = _357 % Une réussite
8   :   ^   a(_357) [1]
9   :       ^  c [1]
```

```
10  :        -  c
11  :        ^  b [1]
12  :        +* b [2]
%...
```

Cet exemple montre l'utilisation des symboles d'appel, de sortie, de réussite, d'échec, de retour d'un but. Cette trace permet de suivre d'une façon explicite les différents mécanismes d'interprétation d'un but Prolog : appel d'un but, activation d'une clause et valeur des arguments, sortie d'un but par échec d'un sous-but ou du but lui-même, les retour-arrières sur un but afin de considérer de nouvelles clauses. Les autres symboles de trace sont employés dans le même esprit que ceux intervenant dans cet exemple.

4.5.2 Extension

Une utilisation de cette trace sur de grosses applications peut donner un flot de messages parfois important et inutilisable. Pour une utilisation rationnelle, il peut être souhaitable de disposer d'un système équivalent au *"spy"* de la trace de Byrd pour obtenir une trace partielle.

Pour réduire le flux de trace Eisenstadt fournit une trace en largeur d'abord qui ne visualise que certains niveaux d'appels en utilisant les mêmes symboles. Il définit pour cela un certain nombre de prédicats de trace partielle :

analyse/1 permet de visualiser l'appel initial d'un but ainsi que son message de sortie, réussite ou cause de l'échec,

zoom/0 permet de visualiser l'appel initial, les appels et sorties des sous-buts ainsi que la sortie du but initial,

zoom/1 permet de visualiser l'appel d'un but de numéro donné ainsi que ses sous-buts. Ce prédicat est équivalent au précédent mais pour un but quelconque de la résolution.

suspects/0 permet de déterminer les problèmes rencontrés lors de la résolution d'un sous-but (coupe-choix suivi d'un échec).

Comme nous pouvons le remarquer à partir de cette description, ces prédicats permettent d'effectuer une trace partielle en largeur d'abord. Voici une trace avec une utilisation de ces prédicats, toujours sur le même exemple de programme :

```
?- analyse(a(X)).
1   :   ?- a(_357)
7   :   +   a(_357) [1]
15  :   +   a(_357) [1]
25  :   +   a(_357) [2]
30  :   -   a(_357)
?- zoom.
1   :   ?   a(_357)
3   :       ?   b
```

```
4   :       +* b [1]
5   :       ?   c
6   :       +* c [1]
7   :   +   a(_357) [1]
```

4.5.3 Commentaires

Cette trace constitue un outil très intéressant pour permettre la mise au point d'un programme complexe. Cependant, elle génère, lorsqu'on l'applique sur de gros programmes, un flux important de données. Il faut donc d'une façon générale utiliser cette trace en réduisant au préalable sa longueur à l'aide du zooming. Ce qui constitue un énorme apport pour une mise au point efficace.

Ainsi, les prédicats de traces partielles permettent de trouver un sous-but de la résolution dont la solution est fausse ou insatisfaite. Cette détection est rendue facile par la numérotation de la trace. Cependant, si l'on effectue la trace partielle d'un sous-but qui peut s'unifier avec un grand nombre de clauses, on ne pourra pas réduire le flot de trace. Il y aura un appel à chacun des sous-buts de chaque clause d'où un nombre important de messages. Cet aspect représente la limite essentielle de cette méthode de mise au point.

De plus, l'utilisateur doit maîtriser la technique de mise au point en largeur d'abord. Lors d'une mise au point avec la trace partielle, l'utilisateur doit concentrer son attention sur les messages et interpréter ceux-ci pour déterminer le sous-but suivant de cette technique.

Enfin, une telle technique de recherche en largeur d'abord peut déconcerter l'utilisateur non expérimenté dont la programmation a pu être influencé par une résolution en profondeur d'abord.

4.6 Conclusion sur les traces

Les trois traces que nous avons décrites permettent sur des exemples simples de comprendre le mécanisme d'interprétation d'un but. Elles deviennent inefficaces, voir inadaptées sur des applications réelles. Cependant, un modèle interactif de trace permettant de parcourir dynamiquement l'arbre de résolution, est utile à la compréhension de l'interprétation d'un programme. Elle doit permettre de visualiser certaines informations : clause en cours d'évaluation, but appelé ainsi que l'ensemble des ancêtres du but courant. Cet outil s'avère important lorsqu'un programmeur désire approfondir ses connaissances sur les mécanismes de la résolution.

5. La détection d'erreurs

5.1 Principe

Comme nous l'avons vue dans la première partie, la détection d'erreurs en Prolog consiste à déterminer une solution incorrecte lorsqu'une résolution est incorrecte. Cette technique de recherche repose sur l'utilisation d'un méta-interprète de Prolog et d'un système de détection d'erreurs fondé sur les connaissances déclaratives d'une application. La littérature nous montre que la réalisation de technique de détection d'erreurs est possible pour des programmes Prolog pur, dans les deux cas de *solution fausse* et de *solution insatisfaite.*

Plaçons nous dans le cadre de Prolog pur et considérons une résolution incorrecte produisant une solution fausse. Une étude détaillée de l'arbre de résolution nous montre qu'il existe un premier sous-but produisant une solution fausse, qui est dans le pire des cas le but initial. La détection d'erreurs dans le cas de la résolution d'un but avec une solution fausse consiste à trouver un premier sous but avec une solution incorrecte.

De même, l'étude du cas de la solution insatisfaite nous montre qu'il existe dans la résolution une première solution incorrecte, qui est dans le pire des cas la solution insatisfaite. La détection d'erreurs consiste à déterminer la première solution incorrecte de la résolution de la solution insatisfaite.

Les deux méthodes principales que nous présentons permettent de répondre à ces deux cas particuliers et utilisent des techniques de recherche différentes.

5.2 *"Algorithmic Program Debugging"*

En 1982 Ehoud Shapiro [Shapiro-83] présente un système de détection d'erreurs pour des programmes Prolog pur. Cette approche complètement nouvelle apporte une automatisation de la recherche d'erreurs lors de la phase de mise au point de programmes Prolog. Ce système interactif, développé pour un utilisateur quelconque, permet de localiser une erreur produisant une solution incorrecte.

Shapiro décrit diverses réalisations d'algorithmes de mise au point pour des programmes Prolog pouvant contenir des négations closes, dans les trois cas d'erreurs suivants : *solution fausse, solution insatisfaite, bouclage d'une résolution.* Ces algorithmes permettent de localiser une erreur dans un programme en demandant à l'utilisateur du système l'interprétation correcte des résultats obtenus pour tous les buts intermédiaires d'une résolution. Nous allons décrire la réalisation de ces algorithmes.

5.2.1 Solution fausse

Shapiro présente deux approches de la détection d'erreurs pour ce cas, l'algorithme simple *"single stepping"* et l'algorithme *"divide and query"* optimal en nombre de questions.

Pour l'algorithme simple, la méthode de recherche est dans le pire des cas équivalente à une trace exhaustive de la résolution. En effet, si l'arbre de résolution possède n nœuds l'algorithme pose au plus n questions. En revanche, pour une trace classique, il y a de l'ordre de 4n+p messages (p = nombre d'échecs). Cette méthode peut être décrite de la façon suivante :

Le problème

Détecter une clause incorrecte C du programme P écrit en Prolog lorsque sur une question l'interprète retourne une solution fausse. C'est-à-dire que la résolution d'un but initial donné retourne une solution particulière, mais celle-ci est une solution fausse selon l'interprétation du programmeur.

Technique

Simuler par une pseudo-trace l'exécution de l'interprète Prolog afin de pouvoir retourner la première clause fausse, produisant une solution fausse mais dont l'arbre de preuve est juste.

Algorithme

L'algorithme *"incorrect"* que nous allons décrire est la recherche d'une clause *"A <- B"* d'un programme P dont chaque sous-buts de la queue de clause possède une solution correcte, mais dont la conclusion possède une solution fausse. Le but *"incorrect(A, X)"* retourne dans la variable X la clause fausse la plus profonde pour la solution fausse A, ou l'atome "bon" pour une solution correcte A.

Nous avons trois cas à traiter :

$A = (A_1, A_2)$

l'application de l'algorithme incorrect sur le but A_1 retourne X_1. Si "X_1 = bon" alors l'application de l'algorithme incorrect sur X_2 retourne la clause X, sinon "$X = X_1$" est la clause erronée.

A est pré défini

"X = bon", un tel but admet toujours une solution correcte.

Défaut :

Il existe une clause *"A <- B"*, l'application de l'algorithme incorrect sur le but B retourne la clause X_b. Si "$X_b \neq$ bon" alors retourner "$X = X_b$", sinon si A est juste "X = bon" sinon "X = *A <- B*".

La traduction en Prolog de cet algorithme de recherche correspond à la déclaration :

```
incorrect((A , B),X) :-
    incorrect(A, Xa),
    ( Xa == bon -> incorrect(B, X) ; X=Xa).
```

```
incorrect(A, bon) :-
    système(A),
    call(A),
    !.

incorrect(A, X) :-
    clause(A, B),
    incorrect(B, Xb),
    ( Xb \== bon -> X = Xb
    ; question_tous( A, true) -> X = bon
        ; X = (A :- B)).
```

où le prédicat "question_tous/2" est décrit dans l'annexe 4.

Prenons un exemple afin d'illustrer ce principe. Considérons le programme *trier*, tiré de [Shapiro-83] page 41. Ce programme réalise un tri croissant par insertion dans une liste de nombres, mais il est incorrect. Il y a une inversion du test sur les nombres A et B, dans la première clause de "*insérer/3*".

```
trier([A|B],C) :-
    trier(B, D),
    insérer(A, D, C).

trier([],[]).

insérer(A,[B|C],[B|D]) :-
    B > A,  % C'est B < A
    insérer(A, C, D).

insérer(A,[B|C],[A, B|C]) :-
    A =< B.

insérer(A,[],[A]).
```

Sur la résolution du but *"trier([2,1,3],X)"*, nous obtenons la solution : *"X = [2,3,1]"*. C'est une solution fausse du programme, dans l'interprétation d'un tri croissant de la liste *"[2,1,3]"*. Utilisons les traces de Byrd et de Eisenstadt sur cette solution, afin de les comparer avec l'algorithme de recherche de Shapiro.

La trace de Byrd sur le but *"trier([2,1,3],[2,3,1])"*, nous donne :

```
(1) 0 Call: trier([2,1,3],[2,3,1]) ?
(2) 1 Call: trier([1,3],_514) ?
(3) 2 Call: trier([3],_553) ?
(4) 3 Call: trier([],_592) ?
(4) 3 Exit: trier([],[]) ?
(5) 3 Call: insérer(3,[],_553) ?
(5) 3 Exit: insérer(3,[],[3]) ?
(3) 2 Exit: trier([3],[3]) ?
(6) 2 Call: insérer(1,[3],_514) ?
(7) 3 Call (BUILT-IN): 3>1 ?
```

```
(7) 3 Exit (BUILT-IN): 3>1 ?
(8) 3 Call: insérer(1,[],_670) ?
(8) 3 Exit: insérer(1,[],[1]) ?
(6) 2 Exit: insérer(1,[3],[3,1]) ?
(2) 1 Exit: trier([1,3],[3,1]) ?
(9) 1 Call: insérer(2,[3,1],[2,3,1]) ?
(10) 2 Call (BUILT-IN): 2=<3 ?
(10) 2 Exit (BUILT-IN): 2=<3 ?
(9) 1 Exit: insérer(2,[3,1],[2,3,1]) ?
(1) 0 Exit: trier([2,1,3],[2,3,1]) ?
```

Sur un exemple aussi simple, la trace de Byrd montre une relative lourdeur, par un nombre de messages important. Pour un utilisateur non averti, il est difficile de trouver l'erreur dans le programme à partir de cette trace. Sur le même exemple, avec la trace de Eisenstadt, nous obtenons les messages suivants :

```
1   :   ? trier([2,1,3],[2,3,1])
2   :   > trier([2,1,3],[2,3,1]) [1]
3   :       ? trier([1,3],_681)
4   :           > trier([1,3],_681) [1]
5   :           ? trier([3],_963)
6   :           > trier([3],_963) [1]
7   :               ? trier([],_1235)
8   :               +* trier([],[]) [2]
9   :               ? insérer(3,[],_963)
10  :               +* insérer(3,[],[3]) [3]
11  :           + trier([3],[3]) [1]
12  :           ? insérer(1,[3],_681)
13  :           > insérer(1,[3],[3|_1906]) [1]
14  :               } 3>1
15  :               ++ 3>1
16  :               ? insérer(1,[],_1906)
17  :               +* insérer(1,[],[1]) [3]
18  :           + insérer(1,[3],[3,1]) [1]
19  :       + trier([1,3],[3,1]) [1]
20  :       ? insérer(2,[3,1],[2,3,1])
21  :       > insérer(2,[3,1],[2,3,1]) [2]
22  :           } 2=<3
23  :           ++ 2=<3
24  :       + insérer(2,[3,1],[2,3,1]) [2]
25  :   + trier([2,1,3],[2,3,1]) [1]
Yes
```

L'analyse de cette trace par un utilisateur chevronné montre que le but intermédiaire *"insérer(1,[3],[3,1])"* est une première solution fausse. Cependant, il est difficile de pouvoir trouver l'origine de cette erreur. L'application de l'algorithme *"incorrect"*, sur cet exemple nous donne la trace interactive suivante :

```
| ?- incorrect(trier([2,1,3],[2,3,1]),X).
Question : trier([],[]) ? y.
Question : insérer(3,[],[3]) ? y.
Question : trier([3],[3]) ? y.
Question : insérer(1,[],[1]) ? y.
Question : insérer(1,[3],[3,1]) ? n.

X = insérer(1,[3],[3,1]):-3>1,insérer(1,[],[1])
```

Cette clause possède une queue dont chaque but a une solution correcte, mais dont la conclusion est une solution fausse. C'est donc une clause fausse du programme.

Cet exemple nous montre qu'un système de détection d'erreurs, fondé sur les algorithmes de Shapiro, permet de trouver une clause incorrecte dans un programme Prolog. Car, même si la trace de Eisenstadt nous donne une aide appréciable sur la position de l'erreur, elle ne fournie pas directement la clause incorrecte, comme le fait la méthode de Shapiro. L'utilisateur doit en plus analyser la trace pour comprendre les raisons de l'erreur. Il simule les déductions logiques d'un système de détection d'erreurs.

L'algorithme *"divide and query"* correspond à une approche équivalente mais il minimise le nombre de questions sur la résolution [Shapiro-83]. L'idée de base est de diviser l'arbre de recherche en deux sous-arbres équivalents en taille et d'en examiner un seul à chaque étape. Si lors de la division, le nœud correspondant au premier sous-arbre admet une solution fausse alors l'erreur est dans ce sous-arbre, sinon elle est dans son sous-arbre complémentaire. Ainsi, si l'arbre de recherche contient n nœuds et b branches, cet algorithme admet une complexité en *"O(b log n)"* et c'est un algorithme optimal pour le cas de la solution fausse. Malheureusement, il est relativement difficile de construire deux arbres équilibrés en taille et la réalisation de Shapiro utilise une heuristique de partage en deux.

5.2.2 Solution insatisfaite

Shapiro présente une méthode unique de détection d'erreurs pour le traitement d'une solution insatisfaite. La méthode correspond à une recherche en largeur d'abord d'un premier sous but de la résolution qui ne possède pas de solution alors que l'utilisateur en connaît au moins une. Le problème et son traitement peuvent s'énoncer de la façon suivante :

Problème

Détecter un but possédant une solution connue par le programmeur, mais ne pouvant être calculée par l'interprète Prolog. C'est-à-dire que la résolution du but initial ne permet pas de trouver une preuve d'une solution connue par le programmeur.

Technique

Simuler par une pseudo trace l'exécution de l'interprète Prolog afin de pouvoir retourner un premier but avec une solution insatisfaite qui n'admet pas dans son arbre de recherche une solution insatisfaite.

Algorithme

L'algorithme *"insatisfaite"*, que nous allons décrire, est la recherche dans l'arbre de résolution d'un but admettant une solution insatisfaite, d'un premier sous but ayant une solution insatisfaite.

La résolution du but *"insatisfaite(A, X)"*, simule l'exécution de l'interprétation du but initial A dont on connaît une solution pour retourner le premier sous but X ayant une solution insatisfaite.

L'algorithme de traitement d'une solution insatisfaite recherche, sur un but A donné, un sous-but ayant une solution insatisfaite X, de la façon suivante :

$A = (A_1 , A_2)$

Si la résolution du but A_1 retourne une solution juste alors le but avec une solution insatisfaite X est à rechercher sur le but A_2, sinon il faut le chercher sur le but A_1.

Défaut

Soit une clause "A <- B", si chaque sous-but de la queue B possède une solution satisfaite selon l'interprétation du programmeur, continuer sur le but B, sinon la solution insatisfaite est "X = A".

La traduction en Prolog de cet algorithme de recherche, correspond à la déclaration suivante :

```
insatisfaite( (A1 , A2) , X ) :-
    call(A1)
    -> insatisfaite( A2 , X )
    ; insatisfaite( A1 , X ).

insatisfaite( A , X ) :-
    clause( A , B ),
    satisfaite( B )
    -> insatisfaite( B , X )
    ; X = A.

satisfaite( (A, B) ) :-
    !,
    question(existe, A, true),
    satisfaite(B).

satisfaite( A ) :-
    question_existe( A, true).
```

où le prédicat "question_existe/2" est décrit dans l'annexe 4.

Afin d'illustrer ce principe, considérons le programme *"trier"*, tiré de [Shapiro-83] page 41. Le programme *"trier/2"* réalise un tri croissant par insertion, mais il est faux, car il manque le test de fin de l'appel récursif du prédicat *"insérer/3"*.

```
trier([X|XS],YS) :-
    trier(XS, ZS),
    insérer(X, ZS, YS).

trier([],[]).

insérer(X,[Y|YS],[Y|ZS]) :-
    X > Y,
    insérer(X, YS, ZS).

insérer(X,[Y|YS],[X, Y|YS]) :-
    X =< Y.
```

Sur le but initial *trier([3,2,1],X)*, nous n'obtenons aucune solution. L'utilisation de la trace de Byrd sur ce but, nous donne :

```
| ?- trace, trier([3,2,1],X).
(1) 0 Call: trier([3,2,1],_67) ?
(2) 1 Call: trier([2,1],_203) ?
(3) 2 Call: trier([1],_242) ?
(4) 3 Call: trier([],_281) ?
(4) 3 Exit: trier([],[]) ?
(5) 3 Call: insérer(1,[],_242) ?
(5) 3 Fail: insérer(1,[],_242) ?
(4) 3 Redo: trier([],[]) ?
(4) 3 Fail: trier([],_281) ?
(3) 2 Fail: trier([1],_242) ?
(2) 1 Fail: trier([2,1],_203) ?
(1) 0 Fail: trier([3,2,1],_67) ?
```

Sur une trace aussi courte, il est aisé pour un utilisateur non averti de trouver que l'échec du but *"insérer(1,[],_242)"*, est responsable de tous les échecs des buts ancêtres et donc de la solution insatisfaite du but initial. Avec la trace de Eisenstadt, il est plus simple de trouver l'échec incorrect par le message particulier d'échec d'unification "-~ ".

En revanche, l'application de l'algorithme de recherche pour la solution insatisfaite sur cet exemple, nous donne la trace de détection d'erreurs :

```
?- insatisfaite(trier([3,2,1],[1,2,3]),X).
question : trier([2,1],X) ? y.
avec X ? [1,2].
question : insérer(3,[1,2],[1,2,3]) ? y.
question : trier([1],X) ? y.
avec X ? [1].
question : insérer(2,[1],[1,2]) ? y.
question : trier([],X) ? y.
avec X ? [].
question : insérer(1,[],[1]) ? y.
X = insérer(1,[],[1])
```

La comparaison est alors rapide. L'utilisateur doit interpréter une trace pour deviner où se trouve l'erreur, le premier échec incorrect. En revanche, la méthode de Shapiro fournit directement la solution insatisfaite. L'utilisateur n'a plus qu'à analyser la cause réelle de l'erreur dans son programme. Cependant, un manque d'informations concernant les circonstances de l'erreur se fait sentir. Par exemple, on peut se demander dans quelle clause se produit cet échec de la résolution.

Cet exemple nous montre qu'il est possible de trouver un but avec une solution insatisfaite lorsque la résolution d'un de ses sou-buts ne retourne aucune solution alors qu'il en existe au moins une. La complexité de cette méthode est en O(b·d) pour un arbre de résolution contenant b branches et de profondeur maximale d.

5.2.3 Bouclage du programme

Shapiro remarque que si, sur l'interprétation d'un but, la branche de l'arbre ET/OU en cours d'évaluation est d'une longueur supérieure à une taille fixée alors il peut exister un bouclage. En effet, si nous considérons que pour toute résolution correcte d'un but Prolog, toute SLD-réfutation possède une longueur inférieure à une borne fixée (non garantie par la théorie), lorsque nous sommes sur une dérivation infinie la longueur de la branche d'évaluation dépasse cette borne. Il suffit ensuite d'analyser la branche afin d'y détecter un cycle de l'évaluation, sur les clauses activées ou bien sur les buts. Ainsi, sur son modèle de méta-interprète sa réalisation de la détection de boucles est la suivante :

```
boucle(A, N) :-
    évaluer(A, N, D),
    (D = débordement -> chercher_boucle(D) ; true).

évaluer(true, D, true) :-
    !.

évaluer(A,0,[débordement]) :-
    !.

évaluer((A, B), D, S) :-
    évaluer(A, D, Sa),
    ( Sa=true
        -> évaluer(B, D, S)
        ; S=Sa ).

évaluer(A, D, Sa) :-
    D1 is D-1,
    clause(A, B),
    évaluer(B, D1, SB),
    ( SB = true
        -> Sa = true
        ; ( SB = [débordement|S]
            -> Sa = [débordement,(A:-B)|S] )).
```

Lorsque la méta-interprétation d'un but donné retourne un *débordement*, ce qui signifie qu'il existe une branche de l'arbre de résolution de taille supérieure à D, on peut alors rechercher un but responsable d'une boucle. En effet, lorsqu'il y a *débordement*, le résultat S de *"évaluer(but, D, S)"*, contient toute la branche générant le débordement. Ainsi, on peut rechercher dans cette liste s'il existe un cycle dans l'évaluation des buts et conclure qu'il y a bouclage quand il existe deux buts qui sont chacun un variante de l'autre. Cette recherche, le long de la branche trop longue, s'effectue avec l'aide de l'utilisateur du système.

La méthode de Shapiro pour résoudre ce problème consiste à limiter la profondeur de l'arbre de résolution d'une façon arbitraire. Ceci peut-il être concevable lors d'une réalisation grandeur nature d'un programme ? L'arbre de résolution pouvant être très grand, il est difficile de pouvoir donner la profondeur maximale. Est-il trop court, trop long, quelle est la *bonne* profondeur ? Aucune réponse ne peut être fournie.

5.2.4 Conclusion

L'utilisation de telles méthodes de détection d'erreurs pour la mise au point de programmes Prolog pur est très utile lors du développement d'une application. Si nous étudions attentivement l'approche de Shapiro pour résoudre l'absence de solution ainsi que pour la solution fausse, nous devons reconnaître qu'elle modélise de façon parfaite le comportement logique et les réflexions que peut se faire le programmeur et dans une phase de mise au point de ses programmes.

Cependant, si nous voulons qu'une telle approche de la mise au point soit enrichissante pour le programmeur, il est important d' utiliser ces méthodes sur tout Prolog ou au moins sur un sous-ensemble de Prolog contenant obligatoirement la coupure et la négation. Ainsi, nous devons effectuer une étude des extensions Prolog afin de déterminer leurs interactions avec les algorithmes de mise au point. L'utilisateur disposerait de cette façon d'un outil performant pour la mise au point en Prolog.

Cependant, un aspect négatif de la méthode de Shapiro est d'obliger l'utilisateur à donner une valeur aux variables lorsqu'il existe une solution insatisfaite. Il nous faut revoir ce traitement de la non connaissance de la valeur de la solution que l'on connaît.

Par ailleurs, Nous devons analyser les méthodes concurrentes pour établir les fondements d'une méthode étendue de mise au point en Prolog permettant de détecter une solution incorrecte dans une résolution incorrecte. En effet, l'extension directe des méthodes de Shapiro est insuffisante pour traiter tous les cas d'erreurs en Prolog, comme nous le montrerons plus loin.

5.3 Rational Debugging

Sur le principe la méthode de Pereira est identique à celle de Shapiro. Cette méthode utilise les dépendances entre tous les termes Prolog de la dérivation d'un but par une technique de parcours en profondeur d'abord ou largeur d'abord, de l'arbre de recherche. Comme pour la méthode de Shapiro, celle-ci réalise une détection d'erreurs pour le problème de la solution fausse ainsi que celui de la solution insatisfaite.

Dans son article Pereira décrit le principe de l'algorithme de son détecteur d'erreurs qui permet d'automatiser la mise au point sous une forme rationnelle pour l'utilisateur (d'où le nom de la méthode). Son système de détection d'erreurs utilise l'aspect déclaratif et la sémantique opérationnelle des programmes Prolog. D'une façon générale, pour une solution incorrecte, il existe un terme qui est incorrect (solution incorrecte = un terme est incorrect).

En addition au code source, au code objet du programme et en complément des informations données par l'utilisateur, le détecteur d'erreurs utilise des informations entre les dépendances des termes. Ces informations résultent de la modification de l'algorithme d'unification lors de l'exécution du programme. Cet algorithme a été développé par Bruynooghe et Pereira afin de réaliser une méthode de retour-arrière intelligent, [Bruynooghe-Pereira-84]. Dans cet algorithme, un arbre de déduction est associé à chaque terme lors de l'unification. L'utilisation de ces arbres de déduction permet de trouver une erreur lors d'un résolution.

5.3.1 Solution fausse

L'idée de l'algorithme de recherche d'une solution fausse est de parcourir l'arbre de dérivation en largeur d'abord en prenant en compte les relations de dépendances entre les termes faux des solutions fausses. Plus précisément, en parcourant les arbres de déduction des termes faux associés aux solutions fausses ainsi que des informations fournies par l'utilisateur. L'algorithme s'arrête lorsqu'il trouve une clause dont chaque prémisse est vraie (sa preuve est correcte), mais dont la solution est fausse. L'algorithme peut être découpé en plusieurs parties :

A1 Un but possédant une solution fausse est présenté au débogueur. Il y résolution du but pour trouver une clause fausse produisant une solution fausse.

A2 Sur la solution fausse, il y a détermination du terme faux par l'utilisateur.

A3 Le débogueur utilise les relations entre les termes de la résolution afin de trouver le nœud le plus récent de la résolution dont dépend ce terme faux.

A4 Sur ce plus récent terme faux, le détecteur d'erreurs questionne l'utilisateur pour déterminer si cette solution est une solution juste.

A5 Cas A4-non. Si cette solution est une solution incorrecte, le débogueur recommence au point A3 avec un nouveau terme faux déterminé par l'utilisateur.

A6 Cas A4-oui. Si la solution est une solution juste, le débogueur unifie le but avec la tête de la clause correspondant au nœud courant et questionne l'utilisateur pour déterminer si le littéral résultant est *solvable*. C'est-à-dire s'il existe une solution relativement au programme.

A7 Cas A6-non. Nous pouvons avoir un des cas suivants :
La tête de clause est fausse car elle produit une conclusion incorrecte ou parce qu'elle ne s'unifie pas avec une clause.
Un certain but de la queue de clause échoue alors qu'il ne devrait pas.

A8.1 Si la queue de clause est vide, alors la clause courante est fausse et le débogueur la retourne.

A8.2 Si la queue de la clause n'est pas vide, le débogueur questionne l'utilisateur sur chaque sous-but pour déterminer s'il est : (i) incorrect, (t) vrai, (f) faux.
Cas (i) : retour vers le point A3 avec le terme incorrect.
Cas (t) : Si chaque sous-but est juste, alors la clause courante est une clause fausse.
Cas (f) : Si ce but est résolu, le débogueur questionne l'utilisateur pour savoir si on doit repartir au point A1 sur ce sous-but.

A9 Cas A6-oui. Si la clause est un fait alors on continue en A4. Sinon la queue de clause n'est pas vide. La solution fausse est produite par cette clause. Si la solution du but est correcte on continue sur A4, sinon on va en A8.2.

5.3.2 Absence de solution

L'idée de base de l'algorithme est que sur un but juste sans solution, il y a nécessairement un premier but juste sans solution. Le problème est alors de trouver un but ayant une solution insatisfaite sans sous-but ayant une solution insatisfaite. D'une façon générale le programmeur peut déterminer si un but est juste ou non, mais il ne sait pas pourquoi il n'y a pas de solution. Aussi, l'algorithme met-il le programmeur à contribution pour déterminer si un but est juste ou non et le parcours de l'arbre de résolution permet alors de trouver une erreur dans le programme. L'algorithme peut être découpé en plusieurs parties :

B1 Un but, dont la résolution échoue pour produire une solution connue, est présenté au débogueur.

B2 Le débogueur cherche à trouver une solution quelconque.

B3 Lorsqu'un but sans solution est trouvé, l'échec est légitime (l) ou non justifié (i).

B4 Si l'échec d'un but est légitime, alors il n'est pas à l'origine de l'échec du but initial. Le débogueur prend le premier ancêtre de ce but, s'il n'y en a pas on continue en B8.

B5 Si l'ancêtre trouvé possède une solution connue on continuer en B8. Sinon, selon l'échec du but faire un retour-arrière sur cet ancêtre ou pas.

B6-1 Sinon, on recherche à résoudre cet ancêtre et on continue en B7.

B6-2 Si oui, il y a recherche d'une nouvelle solution de l'ancêtre par retour-arrière et on continue en B7.

B7-1 Si la solution de l'ancêtre est prouvée, le débogueur se branche au point B3 pour trouver le premier but sans solution.

B7-2 Si la solution de l'ancêtre échoue, le débogueur questionne l'utilisateur pour connaître si cet échec est (1) inadmissible, (2) non solvable (3) solvable.
Cas (1) : On détermine le mauvais terme et on continue sur la mauvaise solution au point A3.
Cas (2) : L'échec est correct, on recherche un nouvel échec et on continue au point B4.
Cas (3) : on continue en B8.

B8 Le système analyse l'échec du but.

B8-1 Il n'existe pas de clause s'unifiant avec le but. Le débogueur informe l'utilisateur et s'arrête.

B8-2 Il existe une clause s'unifiant avec le but, mais le but échoue pour produire une solution. Le détecteur d'erreurs questionne l'utilisateur pour savoir si ce but est (1) inadmissible, (2) non solvable (3) solvable.
Cas (1) : On détermine le mauvais terme dans le but et on continue sur la mauvaise solution au point A3.
Cas (2) : L'échec est justifié, le détecteur d'erreurs continue l'exécution par un retour-arrière sur le but.
Cas (3) : L'erreur est que les clauses s'unifiant ne résolvent pas le but.

Tous les échecs dans l'exécution d'une des clauses sont légitimes. Le but est retourné par le détecteur d'erreurs et l'exécution se termine.

B8-3 Le but a réussi, mais échoue après un retour-arrière. Le détecteur d'erreurs questionne l'utilisateur pour savoir si ce but est (1) inadmissible (2) non solvable, (3) solvable.
Cas (1) : Idem que pour B8-2
Cas (2) : La solution est mauvaise et le détecteur d'erreurs continue sur le traitement de la mauvaise solution (A1).
Cas (3) : Il y a déjà eu une solution de ce but, mais toutes n'ont pas été produites. Le détecteur d'erreurs appelle l'interprète pour déterminer toutes les solutions possibles de ce but. Il questionne ensuite l'utilisateur sur cet ensemble de solutions pour déterminer s'il est complet et correct. L'utilisateur indique à partir de cet ensemble, si une solution est mauvaise, inexistante ou autre.
Si une solution est mauvaise, le détecteur d'erreurs continue sur le traitement d'une mauvaise solution (cas A)
Si une solution est inexistante, c'est-à-dire l'ensemble des solutions

est incomplet, le but est détecté comme ayant une solution insatisfaite.
Sinon, les solutions sont correctes et toutes présentes, l'échec est correct est le détecteur d'erreurs continue au point B4.

5.3.3 Remarque

L'analyse des deux algorithmes de mise au point de Pereira nous montre que ceux-ci sont voisins des méthodes de Shapiro sur l'approche logique des problèmes. Cependant, Pereira effectue un parcours des arbres de déductions, alors que Shapiro effectue un parcours des arbres de recherches. Ainsi, seule la technique de recherche diffère dans le parcours de l'arbre de résolution, car Pereira utilise les relations entre les différents termes à chaque nœud de l'arbre de résolution afin de modifier le choix du nœud suivant de recherche en fonction des relations obtenues. L'objectif de Pereira étant de réduire par cette technique le nombre de questions posées à l'utilisateur en ne parcourant que les nœuds incorrects, contenant un terme incorrect. Cependant, aucune étude sérieuse de la complexité en nombre de questions permet de montrer une efficacité plus importante. Les algorithmes sont-ils efficace ? Selon Pereira, le nombre moyen des questions est considérablement réduit.

5.4 Autres méthodes

Nous présentons dans cette partie des approches voisines sur la mise au point de Prolog. La première *"Declarative Error Diagnosis"* est du à LLoyd, la seconde *"Top Down Diagnosis"* de Av Ron, puis une adaptation théorique de la méthode de Shapiro par Ferrand, puis une extension de la méthode *"Rational Debugging"* de Pereira et Calejo, et enfin les méthodes utilisant les assertions déclaratives [Dershowitz-Lee-87, Drabent-all-88].

5.4.1 Declarative Error Diagnosis

Dans son article LLoyd présente un *"détecteur d'erreurs déclaratif"* [LLoyd-86] au sens où le programmeur n'a besoin de connaître que la sémantique déclarative de ses programmes afin d'y rechercher des erreurs par l'utilisation d'une syntaxe étendue et un contrôle avancé. Son détecteur d'erreurs est réalisé de façon déclarative et utilise un sous-ensemble de ce détecteur d'erreurs pour des programmes Prolog sans prédicats non déclaratifs. Il est interactif et s'intéresse aux cas de la solution fausse et de la solution insatisfaite. Le programmeur l'utilise s'il connaît l'interprétation d'un programme erroné.

De plus, LLoyd réalise une étude théorique de la détection d'erreurs déclarative qui aboutit à la description d'un détecteur déclaratif, dans son livre sur les fondements de la programmation logique [LLoyd-87].

5.4.2 Top-Down Diagnosis

Les méthodes de détection d'erreurs *Top-Down Diagnosis* de Av Ron [Av Ron-84] permettent de réaliser des algorithmes de détection d'erreurs pour les deux cas solution fausse et solution insatisfaite. Ces méthodes utilisent un parcours en largeur d'abord de la recherche d'erreurs et sont fondées sur les algorithmes de Shapiro. Ces méthodes sont modifiées pour permettre un parcours rapide de la résolution. De plus, Av Ron propose une méthode similaire de celle de Shapiro et utilise comme Pereira certaines relations entre les termes des solutions incorrectes pour effectuer un retour-arrière intelligent.

5.4.3 Error Diagnosis in Logic Programming

La méthode de détection d'erreurs de Ferrand est une adaptation de celle de Shapiro [Ferrand-85]. Dans cet article, Ferrand présente en détail une analyse théorique des algorithmes de Shapiro. L'intérêt principal de ce travail est l'approche théorique qu'il a de la détection d'erreurs. Elle permet à l'auteur de réaliser un système de déclaratif plus général, mais s'appuyant sur des techniques de détections d'erreurs similaires.

5.4.4 Select And Query

Cette méthode est une extension de la détection d'erreurs basée sur les principes des méthodes de Pereira. Cette technique est fondée sur un algorithme de sélections et de questions sur les termes d'un littéral [Pereira-Calejo-88]. Il permet, selon les auteurs, le traitement de la coupure, de la négation mais aussi des prédicats méta-logiques qui n'ont pas d'effet de bord sur la résolution. Les principes de l'algorithme sont les suivants :

- Fournir un environnement de mise au point pour Prolog.
- Proposer un traitement uniforme pour la solution fausse, la solution insatisfaite et la solution incorrecte sur le type d'un terme.
- Doter la procédure de détection d'une heuristique pour qu'elle minimise le nombre de questions posées.

L'algorithme de recherche d'une erreur est fondé sur la de construction de structures permettant de méta-interpréter la résolution Prolog. Ces structures sont construites pour permettre l'utilisation d'une coupure étendue pour simuler le coupe-choix, mais aussi pour gérer les ensembles de buts suspects en cours d'évaluation.

Comme nous le verrons plus loin, dans le chapitre sur le méta-interprétation, l'utilisation d'une coupure étendue ne permet pas de réaliser un système de mise au point *déclaratif*, car il utilise des techniques de programmation du type *langages structurés*.

De plus, une telle méthode se situe à la limite entre la détection d'erreurs déclarative et la trace interactive. Elle nécessite une connaissance opérationnelle de la sémantique du programme, car le système demandera parfois des informations qui ne concernent que la sémantique actuelle, c'est-à-dire le calcul de la résolution.

5.4.5 Méthodes Déductives

L'originalité de l'approche déductive repose sur l'utilisation d'une connaissance "a priori" de la sémantique attendue du programme.

La sémantique attendue se présente sous la forme d'une description formelle. Deux approche peuvent être envisagées :

- La description complète de la sémantique attendue est donnée au détecteur d'erreurs. Le système vérifie chaque solution obtenue à l'aide de cette description. C'est la méthode déductive [Dershowitz-Lee-87].

- La description est approximative et ce n'est qu'une vue partielle de la sémantique attendue, c'est une vue partielle de cette sémantique. Lorsque le système désire une connaissance de la sémantique attendue, il peut poser des questions à l'utilisateur pour en obtenir la description formelle. C'est la méthode de l'utilisation des assertions dans la détection d'erreurs [Drabent-all-88]

Ces descriptions formelles sont écrite en Prolog et le système de détection d'erreur les utilisent pour vérifier la sémantique obtenue en la calculant. L'intérêt essentiel de ces approches est le fait qu'elles permettent de réduire voire d'annuler l'interaction entre l'utilisateur et le système.

Dans le cas de l'utilisation des assertions [Drabent-all-88] l'utilisateur écrit des descriptions formelles pour les deux cas de recherche d'erreurs sous forme de programmes Prolog que le système utilisera pour vérifier une solution calculée. Cette technique utilise une forme particulière des méthode de Shapiro pour les deux cas de la solution fausse et de la solution insatisfaite. Le système de détection questionne l'utilisateur lorsqu'il n'existe pas de connaissances formelles sur la solution obtenue, sous forme d'assertions. L'utilisateur peut ajouter des assertions lors de ces questions ou répondre par oui ou non s'il le désire.

Ces approches sont intéressantes car elles mettent en oeuvre un mécanisme évaluable pour vérifier la solution d'un but.

5.5 Conclusion sur la détection d'erreurs

L'utilisation de telles méthodes pour réaliser la détection d'erreurs de programmes Prolog contenant des négations, des coupures et des prédicats prédéfinis, suppose une étude du comportement de ces extensions par rapport à Prolog pur et par rapport à la résolution. Plus particulièrement, nous devons étudier les algorithmes de recherche pour fournir un environnement unifié de mise au point offrant au programmeur une grande souplesse d'utilisation. Notre étude peut se diriger dans trois grandes directions : les algorithmes de détection d'erreurs, la simulation de la résolution, l'ergonomie de l'interface de dialogue d'un système de mise au point. Ces trois directions correspondent aux composantes d'un système de mise au point en Prolog.

Cependant, nous devons réaliser un outil susceptible de simuler la résolution d'un but. De plus, il nous semble que l'ensemble des méthodes présentées dans cette partie sont équivalente sur l'approche logique et sur la modélisation de la mise au point en Prolog. En revanche, seule une méthode utilisant les principes de Shapiro mérite toute notre attention afin d'être étendue à tout Prolog. En effet, l'utilisation d'un méta-interprète comme support logique de ces algorithmes rend ces méthodes simples, logiques et les plus ouvertes aux extensions que nous souhaiterons réaliser.

Notre étude va don porter sur la définition d'un méta-interprète de Prolog permettant d'effectuer une interprétation des programmes Prolog pouvant contenir des coupures, des négations et des prédicats méta-logiques compilés.

Cependant, afin de mieux adapter la technique de détection d'erreurs aux principales extensions, la coupure et la négation, nous allons étudier les extensions directes des méthodes de mise au point de Shapiro. Cette étude aboutira à la définition d'un outil plus général de mise au point, que nous intégrerons ensuite dans le méta-interprète.

6. Extensions

L'utilisation des algorithmes de détection d'erreurs de Shapiro permet de réaliser une mise au point de programmes Prolog de façon plus intelligente. Cependant, il serait intéressant de pouvoir disposer d'un tel outil pour effectuer une détection d'erreurs pour des programmes quelconques.

Si nous voulons fournir, à un utilisateur non averti, un système de détection d'erreurs pour un système Prolog particulier, nous devons étudier l'ensemble de ses extensions à Prolog pur. Nous pouvons découper l'ensemble des extensions de la façon suivante : le coupe-choix, la négation par l'absurde, les prédicats logiques et méta-logiques, les constructions du type "repeat/fail", les entrées/sorties, les prédicats de modification de la base de règle (assert/retract), ainsi que d'autres prédicats extra-logiques.

Dans une première partie, pour tenir compte de la coupure, nous allons exposer les principes des extensions des méthodes de Shapiro, sans décrire les réalisations qui peuvent cependant être obtenues à partir du méta-interprète et du détecteur d'erreurs présenté au chapitre 9. Puis nous étudierons les divers types d'extensions afin de pouvoir déterminer les limites d'une telle méthode. Cette étude nous permettra de pouvoir définir une approche différente de la détection d'erreurs pour réaliser une mise au point plus générale sur le domaine d'application et plus fiable sur les extensions visées.

6.1 La solution fausse

Considérons le problème d'une solution fausse dans le cadre de Prolog pur étendu par le coupe-choix et sans négation. Sur la résolution d'un but, nous obtenons une solution ne se trouvant pas dans l'ensemble des solutions correctes selon l'interprétation sous-jacente du programmeur.

L'objectif que nous devons atteindre est de pouvoir localiser un nœud de la dérivation, produisant une mauvaise solution causée par une clause fausse. Nous pouvons remarquer qu'il existe une suite de nœuds incorrects le long de l'arbre de résolution. Ces nœuds correspondent à une solution fausse. La méthode de Pereira utilise cette suite de solutions fausses comme structure de base de l'algorithme de recherche.

De plus, l'utilisateur connaît pour chaque but associé à un nœud, l'ensemble des solutions correctes dans l'interprétation du programme. Un parcours en largeur d'abord de l'arbre de recherche permet de suivre cette suite de solutions fausses jusqu'au dernier nœud incorrect alors qu'un parcours en profondeur d'abord permet de remonter directement jusqu'au dernier nœud au moyen des informations données par l'utilisateur.

6.1.1 Analyse du problème

Considérons un programme écrit en Prolog pur étendu par le coupe choix et la résolution d'un but quelconque sur lequel nous obtenons une solution particulière. Supposons cette solution du but initial comme étant une solution fausse dans l'interprétation sous-jacente du programmeur. Le problème de la détection d'erreurs est de trouver une clause fausse du programme Prolog responsable de cette solution fausse.

Une analyse du comportement de la résolution sur une solution fausse, nous conduit à énoncer le théorème suivant :

Théorème :

L'arbre de preuve de la solution fausse contient un nœud (sous-but) incorrect dont les solutions de chaque sous-nœud sont des solutions correctes.

Démonstration :

Effectuons une démonstration par l'absurde.

Supposons que pour chaque but de l'arbre de preuve,il existe au moins un sous-but avec une solution incorrecte. Nous devons remarquer que ceci est impossible, car l'arbre de preuve de la solution fausse est fini, or chaque feuille admet une solution correcte car c'est le prédicat toujours vrai *true*.

Ainsi, dans le pire des cas, la solution fausse cherchée est la solution initiale.

Ce théorème nous permet de caractériser une telle solution fausse de l'arbre de recherche. Nous pouvons la définir de la manière suivante :

Définition :

On appelle *nœud incorrect*, un nœud appartenant à l'arbre de preuve d'une solution fausse dont tous les sous-nœuds possèdent une solution correcte.

De ce théorème et de cette définition, nous pouvons déduire qu'il existe au moins un tel nœud. Montrons qu'il existe un nœud incorrect associé à une clause fausse du programme :

Proposition :

Soit une résolution incorrecte produisant une solution fausse. Il existe dans l'arbre de preuve de cette solution une première solution incorrecte associée à une clause fausse du programme.

Démonstration :

Appelons C cette clause fausse du programme. Considérons la sous-arborescence du nœud de l'arbre de preuve associé à cette clause et à la solution fausse. Dans le cas où cette sous-arborescence ne contient pas de coupe-choix, nous sommes dans les conditions de Prolog pur, la clause C est bien une clause fausse car sa conclusion est incorrecte (c'est une solution fausse), et chaque sous-but de celle-ci possède une solution correcte.

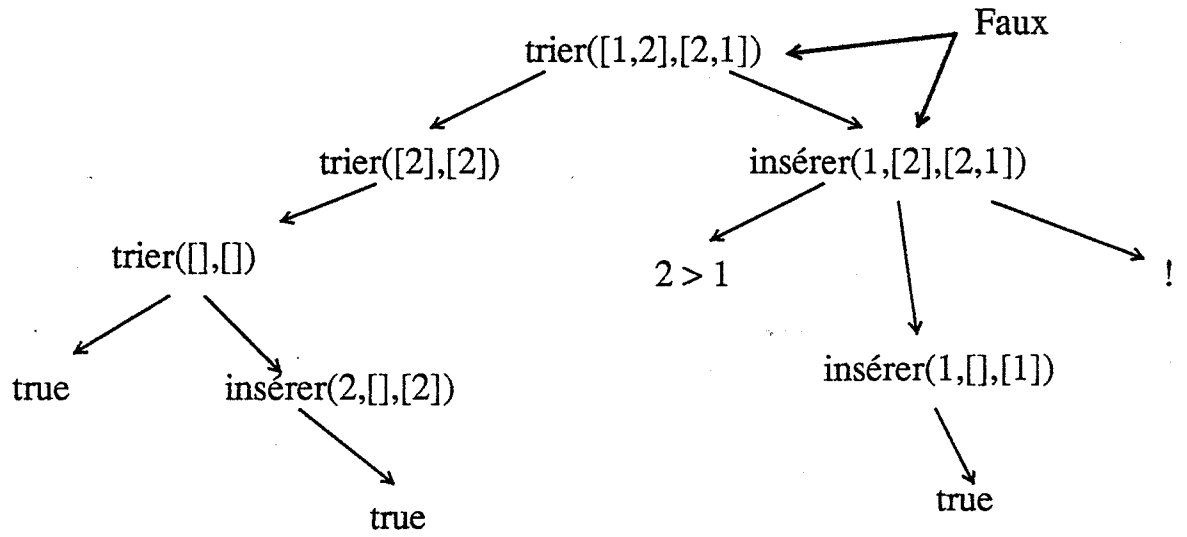
Sinon, soit C' la clause associé au nœud de la sous-arborescence contenant le coupe-choix, on a "C' = A :- (A_1), !, (A_2)". Si la clause C' n'est pas la clause trouvée, C ≠ C', nous pouvons affirmer que la conclusion A est correcte, c'est à dire que la solution calculée est une solution correcte. De même que pour les sous-buts A_1 et A_2, les solutions sont correctes. Dans le cas contraire, la clause C est identique à C' (c'est bien une clause fausse du programme) car tous les sous-buts ont une solution juste mais la conclusion est une solution fausse.

Illustrons ce théorème sur le programme, du tri par ordre croissant :

```
trier([X|XS], YS) :-
    trier(XS, ZS),
    insérer(X, ZS, YS).

trier([],[]).

insérer(X,[Y|YS],[Y|ZS]) :-
    Y > X,  /* en fait : 'X > Y' */
    insérer(X, YS, ZS),
    !.      /* coupe les autres cas */

insérer(X,[Y|YS],[X, Y|YS]) :-
    X =< Y.

insérer(X,[],[X]).
```

Sur le but initial : *"trier([1,2],X)"*, nous obtenons une solution calculée, *"X = [2,1]"*, qui est une solution fausse dans l'interprétation sous-jacente correspondant au tri croissant par insertion de la première liste.

Nous pouvons construire l'arbre de preuve de cette solution en donnant l'interprétation de la solution, lorsqu'elle est fausse. Nous obtenons :



Au vu d'une telle construction de l'arbre de preuve, il est clair que l'on peut trouver la clause incorrecte du prédicat *"insérer/3"*. La preuve de la solution fausse est correcte : chaque sous-but est juste, mais la solution obtenue, *"insérer([1],[2],[2,1])"*, est une solution incorrecte. Dans ces conditions, si nous effectuons un parcours de l'arbre de recherche d'une solution fausse, nous pouvons déterminer un sous-but produisant une solution fausse et la clause incorrecte qui la produit.

6.1.2 Algorithme de recherche

Dans la partie précédente nous avons caractérisé le problème de la solution fausse. Ceci nous permet de pouvoir réaliser un algorithme de recherche d'une clause fausse d'un programme Prolog. En effet, si nous effectuons un parcours en profondeur d'abord de l'arbre de preuve d'une solution incorrecte, le théorème précédent caractérise une clause fausse du programme.

Définition :

Soit B un but Prolog à résoudre. L'algorithme de recherche, que nous appellerons *"résoudre(B)"*, peut être énoncé comme suit :

Soit un but et son nœud de l'arbre de résolution.
Si chaque sous-nœud de ce nœud est juste,
Alors
    Si la solution retournée pour ce but est juste
        Alors on continue {la solution et sa preuve sont justes}
        Sinon la clause courante est une clause fausse.
Sinon on a déjà trouvé la clause fausse dans la sous-arborescence

La recherche en profondeur d'abord s'effectue par l'application récursive de cet algorithme. La détermination d'une solution juste est effectuée par l'utilisateur qui répond aux questions posées par le système. Pour illustrer cette méthode considérons l'exemple précédent et appliquons cet algorithme :

```
?- résoudre(trier([1,2],X)).
    Diagnostic, Recherche d'une Clause fausse...
Question, sur le but: trier([],[])
La solution est correcte ? y.

Question, sur le but: insérer(2,[],[2])
La solution est correcte ? y.

Question, sur le but: trier([2],[2])
La solution est correcte ? y.

Question, sur le but: insérer(1,[],[1])
La solution est correcte ? y.

Question, sur le but: insérer(1,[2],[2,1])
La solution est correcte ? n.
```

```
Sur le but : insérer(1,[2],[2,1]).
Il existe une clause fausse :
Paquet "insérer/3", clause 1

    insérer(X,[Y|Ys],[Y|Zs]):-
        Y>X,
        insérer(X, Ys, Zs),
        !.
```

Nous déterminons ainsi d'une façon sûre la clause fausse associée à la première solution fausse. Cependant, une telle approche suppose de parcourir toute la résolution par une trace interactive. En effet, cet algorithme va tracer tous les sous-buts dont les conclusions sont justes avant de trouver la clause fausse. Dans le cas où le nombre de branches est important, nous obtenons un grand nombre de messages de traces demandant confirmation. Cette approche est, comme la méthode *"single stepping"* de Shapiro, linéaire sur le nombre de nœuds d'une résolution incorrecte.

Av Ron montre qu'il est possible de trouver une clause fausse d'un programme, lorsqu'il existe une solution fausse, par l'application d'une recherche en largeur d'abord [Av Ron-84]. Ainsi, nous pouvons modifier l'algorithme de recherche en ne parcourant que les branches fausses, par une technique de parcours en largeur d'abord. Nous pouvons procéder de la façon suivante :

L'appel d'un But retourne une solution quelconque (X).
Si cette solution (X) est juste
Alors on continue car ce but n'est pas celui cherché.
Sinon
    Si l'interprétation de ce but est juste
    Alors on est sur une clause fausse.

Ainsi, lorsque l'algorithme retourne une clause fausse, le but associé à cette clause possède une solution fausse et chaque sous-but de cette clause possède une solution juste. Nous sommes bien sur une clause fausse du programme. L'application de cet algorithme, sur l'exemple précédent, donne la trace interactive de détection d'erreurs suivante :

```
| ?- résoudre(trier([1,2],X)).
Yes : trier([1,2],[2,1])
Solution juste (o/n) ? n.
    Diagnostic, Recherche d'une Clause fausse...

Question, sur le but: trier([1,2],[2,1])
La solution est correcte ? n.
Question, sur le but: trier([2],[2])
La solution est correcte ? y.

Question, sur le but: insérer(1,[2],[2,1])
La solution est correcte ? n.
```

```
Question, sur le but: insérer(1,[],[1])
La solution est correcte ? y.

Sur le but : insérer(1,[2],[2,1])
    Clause fausse : Paquet "insérer/3", clause 1
        insérer(X,[Y|Ys],[Y|Zs]):-
            Y>X,
            insérer(X, Ys, Zs),
            !.
```

Dans cette étude, nous devons remarquer que l'utilisation de la coupure dans un programme ne modifie pas d'une façon importante l'approche logique de l'algorithme de recherche. L'effet du coupe-choix permet de supprimer certaines branches de l'arbre de résolution. Lorsque la résolution retourne une mauvaise solution, cet effet peut être ignoré lors de l'application de l'algorithme de mise au point. Tenir compte de la coupure, dans le cas d'une solution fausse, revient à construire de façon correcte l'arbre de recherche de la solution fausse et donc son arbre de preuve comme restriction de l'arbre de recherche.

6.2 La solution insatisfaite

Le problème de l'absence de solution, ou solution insatisfaite, correspond au fait que la résolution d'un but échoue pour retourner une solution particulière alors qu'il en existe au moins une dans l'interprétation sous-jacente. En d'autres termes, l'ensemble des solutions d'un but est vide et on suppose que cette absence de solution est incorrecte, c'est-à-dire que l'ensemble des solutions du but n'est pas vide dans l'interprétation du programmeur. Lorsque nous effectuons une analyse simple de ce comportement, il est clair qu'il existe un échec d'un sous-but de la résolution produisant l'échec final de la résolution. Ainsi, par une méthode de trace interactive sur la résolution on peut trouver un premier sous-but dont l'échec est incorrect. Nous présentons deux algorithmes de recherche dans ce cas, le premier en largeur d'abord et le second en profondeur d'abord.

6.2.1 Principe

Soit P un programme Prolog. Considérons un but quelconque dont on cherche à déterminer une solution particulière. On suppose connu pour ce but initial une solution mais En revanche la résolution échoue pour en produire une. Notre problème est de trouver un sous-but de l'arbre de résolution tel que son arbre de résolution soit sans solution incorrecte. C'est-à-dire que toute la sous-arborescence de résolution ne possède pas de but juste dont la résolution échoue. Ce sous-but sera par définition un but avec une solution insatisfaite.

Pour caractériser un tel but, nous devons élaborer un algorithme de recherche simulant l'interprète Prolog et questionnant le programmeur sur chaque but rencontré.

Supposons l'existence d'un oracle permettant de donner une solution à un but quelconque. En fait cet oracle permet sur un but initial donné, d'obtenir l'arbre de preuve d'une solution de ce but. Nous pouvons procéder de la façon suivante :

Soit un but particulier possédant une solution dans l'interprétation sous-jacente du programmeur. Supposons que la résolution de ce but ne fournisse aucune solution.

Considérons un sous-but quelconque de l'arbre de recherche de ce but avec une solution insatisfaite.

Lorsque ce sous but admet une solution dans l'interprétation sous-jacente et qu'il existe un preuve de cette solution, la solution est calculable et il faut continuer car la solution insatisfaite cherchée est dans le reste de l'arbre de recherche.

S'il n'existe pas de preuve pour cette solution, la solution n'est pas calculable. Une recherche dans l'arbre de résolution de ce sous-but permet de fournir une solution insatisfaite. S'il n'en existe pas, c'est alors le sous-but courant qui possède une solution insatisfaite

Cette détection d'erreurs, dans le cas d'une solution insatisfaite, est une recherche en largeur d'abord que l'on peut comparer à celle de la méthode de Shapiro.

Son étude nous montre un problème d'analyse de la recherche d'erreurs. Lorsque le sous-but courant admet une solution calculable, rien ne garantit que l'arbre de preuve de cette solution soit un arbre correct, c'est à dire sans solution incorrecte. Cette méthode ne permet pas de le vérifier. Il est cependant clair qu'il existe une solution insatisfaite dans le reste de l'arbre de résolution.

Ainsi, nous venons de mettre en évidence un problème sous-jacent que nous pouvons énoncer par la phrase suivante :

L'arbre de preuve d'une solution est-il juste ?

Nous devons analyser ce problème, qu'aucun auteur de la littérature n'expose. La réponse à ce problème est l'un des objectifs de la méthode de détection d'erreurs que nous présentons dans la partie III. C'est pourquoi nous la qualifions de méthode de vérification d'une résolution.

6.2.2 Réalisation

L'algorithme de recherche est basé sur le principe précédent et utilise un oracle pour déterminer les informations dont il a besoin pour détecter l'erreur. Cet oracle est un prédicat qui permet de questionner l'utilisateur du système pour connaître si un but, dont on ne trouve pas de solution calculée, admet une solution particulière. S'il existe une solution, l'oracle la demande à l'utilisateur. Ensuite, le système vérifie si elle est une conséquence logique du programme. Sur ce principe nous pouvons définir l'algorithme de recherche de la façon suivante :

Si le but courant possède une solution
et cette solution admet une preuve
Alors continuer l'interprétation
Sinon
    Si la méta-interprétation de ce but échoue
    Alors arrêter,car c'est le but recherché
    Sinon continuer l'interprétation

Afin d'illustrer le principe que nous venons de décrire, prenons le programme Prolog *"trier"*. Ce programme réalise un tri croissant par insertion au moyen d'un algorithme récursif. Ce programme est faux car il manque le test d'arrêt de fin de l'appel récursif pour le prédicat *"insérer/3"*, sa déclaration est la suivante :

```
trier([X|XS],YS) :-
    trier(XS, ZS),
    insérer(X, ZS, YS).

trier([],[]).

insérer(X,[Y|YS],[Y|ZS]) :-
    X > Y,
    insérer(X, YS, ZS),
    !.

insérer(X,[Y|YS],[X, Y|YS]).
```

Sur le but *"trier([2,1],X)"*, nous n'obtenons aucune solution. Nous avons dans ce cas une solution insatisfaite *"X=[1,2]"*. La trace de Eisenstadt de ce but nous donne :

```
| ?- résoudre(trier([2,1],X)).
1   :   ?   trier([2,1],_59)
2   :   >   trier([2,1],_59) [1]
3   :       ?   trier([1],_357)
4   :       >   trier([1],_357) [1]
5   :           ?   trier([],_639)
6   :           +*  trier([],[]) [2]
7   :           ?   insérer(1,[],_357)
8   :           --  insérer(1,[],_357)
9   :           ^*  trier([],[]) [2]
10  :           -   trier([],_639)
11  :       <   trier([1],_357) [1]
12  :       -   trier([1],_357)
13  :   <   trier([2,1],_59) [1]
14  :   -   trier([2,1],_59)
```

Cette trace permet de trouver une absence d'unification pour le but *"insérer(1,[],_357)"* par le message 7. L'utilisateur de cette trace doit ensuite interpréter quelle doit être la substitution pour la variable *"_357"*, et si l'échec d'unification est incorrect. Cependant, l'application de l'algorithme de recherche dans ce cas, nous donne la trace interactive suivante :

```
| ?- résoudre(trier([2,1],X)).

Question : Sur le but: trier([2,1],_361)
Une solution ? oui.
Avec : _361 ? [2,1].

Question : Sur le but: trier([1],_652)
Une solution ? oui.
Avec : _652 ? [1].

Question : Sur le but: trier([],_853)
Une solution ? oui.
Avec : _853 ? [].

Question : Sur le but: insérer(1,[],[1])
Une solution ? oui.
=====> Solution insatisfaite : insérer(1,[],[1])
```

Nous obtenons une information plus précise sur l'échec de l'appel récursif, il n'est pas possible de prouver que l'insertion de "1" dans la liste vide donne la liste "[1]". La différence n'est pas très importante, mais l'utilisateur possède une information plus précise dans ce cas, il est impossible de prouver la solution fournie.

6.2.3 Une autre approche

La méthode que nous venons de présenter est directement inspirée de celle de Shapiro pour le cas absence de solutions (solution insatisfaite). Nous présentons dans cette partie un algorithme différent de recherche d'un but ayant une solution insatisfaite. Il permet de parcourir en profondeur d'abord l'arbre de recherche de la solution insatisfaite. Cette méthode vérifie que tout sous-but de l'arbre de recherche du but avec une solution insatisfaite est correct.

Soit un but ayant une solution insatisfaite. Si, dans la résolution, il existe un sous-but ayant une solution et tel qu'aucune clause ne permette de trouver une solution, ce sous-but possède une solution insatisfaite. Considérons l'algorithme de recherche suivant :

Soit un but à résoudre,
Si la méta-résolution du but fourni comme solution la substitution Θ
    Alors le but a une solution insatisfaite pour Θ
    Sinon
        S'il existe une solution connue Θ
            Alors la substitution Θ est insatisfaite
            Sinon continuer, l'échec est valide

Ainsi, l'application de cet algorithme sur la résolution d'un but nous permet de retourner le premier but avec une solution insatisfaite. Sur l'exemple de programme précédent, nous obtenons la trace interactive suivante :

```
| ?- résoudre(trier([2,1],X)).
Question : Sur le but insérer(1,[],_389)
Une solution ? oui.

=====> Solution insatisfaite : insérer(1,[],_389)
```

Nous trouvons ici directement le premier sous-but ayant une solution insatisfaite. Cette méthode en profondeur d'abord recherche la première cause de l'échec final. La recherche est plus rapide lorsque , nombre de questions plus réduit. En revanche la complexité est linéaire par rapport au nombre de nœuds de l'arbre de recherche. Cette méthode de recherche est similaire à celle de Pereira dans le cas de la solution insatisfaite, mais en parcourant l'arbre de recherche et non pas l'arbre de déduction, elles sont identiques en complexité.

6.2.4 L'extension du coupe-choix

Dans toute l'étude que nous avons menée sur la réalisation de la méthode de recherche pour la solution incorrecte, nous n'avons pas abordé le problème de l'utilisation du coupe-choix dans les programmes. Plus précisément, nous possédons deux techniques de recherche, l'une en largeur d'abord, l'autre en profondeur d'abord. Quelle est la méthode fiable dans le cas d'une solution insatisfaite ?

Il faut bien être conscient que lors de la réalisation d'une application, l'utilisateur, et par la même occasion le système de détection d'erreurs, peuvent être confrontés à des cas particuliers d'utilisation de la coupure. En d'autre termes, si le programmeur utilise un "*cut vert*", la présence de solutions incorrectes peut rendre ce coupe-choix comme "*rouge*". De même, lorsque le programmeur utilise un "*cut rouge*", il faut adapter les méthodes de mise au point au problème de l'utilisation non sûre du coupe-choix.

Dans le cas particulier d'une solution insatisfaite produit par un programme contenant un coupe-choix, le symptome d'erreur doit conduire le programmeur à une réflexion sur l'utilisation qu'il fait de ce coupe-choix. En effet, il existe deux types d'utilisation du coupe-choix. La première qui vise à rendre optimal la résolution par l'élagage des branches ne produisant pas d'autre solution, l'usage de la coupure est dite sûre. L'autre visant à éliminer des solutions possibles, par l'élagage de branches produisant une autre solution, la coupure est dite dangereuse. L'usage sûr de la coupure permet d'améliorer l'efficacité sans omettre de solution. En revanche, l'usage dangereux du coupe-choix est néfaste à la déclarativité des programmes.

Pour illustrer une mauvaise utilisation de coupure ou une mauvaise écriture de coupure, prenons le programme suivant :

```
go(X) :-
    a(X),
    !,
    b(X).

go(X) :-
    X = 1.
```

```
a(1).

a(2).

b(2).
```

Sur le but *"go(1)"* nous n'obtenons pas de solutions. En effet, *"a(1)"* réussit, puis le coupe-choix, mais *"b(1)"* n'admet pas de solution. Deux cas peuvent se produire :

b(1) est juste
: c'est une solution insatisfaite et l'algorithme de recherche le trouvera, pour les deux approches.

b(1) est faux
: ce n'est pas un *théorème* dans l'interprétation du programmeur. Le but *"go(1)"* s'il a une solution correcte, ne peut pas être démontré par la résolution : *"go(1)"* a une solution insatisfaite.

Dans ce dernier cas nous avons un programme incorrect soit dans l'utilisation du coupe choix, soit dans la déclaration de *"go(1)"*.

Ainsi, il convient de bien analyser le comportement de la résolution d'un but ayant une solution insatisfaite lorsque l'on utilise le coupe-choix. De part la complexité de l'analyse des raisons d'un échec dans ce cas précis, un détecteur d'erreurs intelligent devrait pouvoir indiquer l'activation du coupe-choix lors d'une recherche et de donner des indications sur l'effet de son appel. Nous pourrions indiquer sur l'exemple précédent :

dans la clause *"go(X) :- a(X), !, b(X)"*, l'échec du but *"b(1)"* survenant après un coupe choix, ne permet pas de démontrer le but *"go(1)"*

De plus, il convient de remarquer que sur l'appel du but *"go(X)"* l'utilisation de la coupure est dangereuse. En effet, le but *"go(X)"* n'admet pas de solutions calculées, En revanche *"go(2)"* est une solution calculée. Ce comportement doit toujours être considéré comme incorrect. C'est-à-dire que le programme sera toujours un programme incorrect lorsqu'il se produit un tel effet. Cet aspect est à rapprocher d'une mauvaise utilisation de la négation par l'échec, appelée négation dangereuse. Ce problème complique de façon sérieuse la méthode que nous devons employer pour traiter correctement le problème de la solution insatisfaite.

Reprenons notre étude de la mauvaise utilisation de la coupure produisant une effet de bord non désiré, avec en tête le souci de faire une mise au point efficace. Deux cas peuvent se produire selon la technique de recherche d'erreurs sur le but *"go(X)"*, but possédant une solution insatisfaite :

Largeur d'abord :
: *"go(X)"* admet-il une solution ? oui. *"X = 2"*
: *"go(2)"* est-il correct ? oui.
: il n'y a pas de réponse de la détection d'erreurs

Profondeur d'abord :

"a(X)" donne la solution *"X = 1"*,
"b(1)" admet-il une solution ? non. (échec correct)
"go(X)" admet-il une solution ? oui.
alors *"go(X)"* possède une solution incorrecte

Ainsi l'usage d'une coupure non sûre dans un programme Prolog produit une forme d'incomplétude de la technique de recherche en largeur d'abord. Dans ce cas, la résolution ne garantit pas de pouvoir fournir toutes les solutions *calculables* d'un but. La détection d'erreurs devient incomplète car le système peut répondre qu'il n'y a pas d'erreur alors qu'il devrait répondre qu'il en existe au moins une (voir le cas de *"go(X)"* et la solution *"X=2"*). Quelle que soit l'interprétation de ce résultat, nous devons le considérer comme incorrect afin de conserver la sémantique déclarative des programmes Prolog. Cet objectif est essentiel pour un système de mise au point intelligent.

De plus, d'autres effets sont à regretter pour une interprétation correcte de la recherche d'erreurs de l'abusive de la coupure. Tel est le cas lorsque le programmeur utilise les effets de bord de la coupure pour écrire des programmes qui ne sont pas corrects sur le plan de la sémantique déclarative. Par exemple, si nous considérons le programme :

```
maximum(X, Y, Y) :-
    X < Y,
    !.

maximum(X, Y, X).
```

où l'interprétation du prédicat *"maximum(X, Y, Z)"*,est vraie si et seulement si *"Z est le maximum de X et de Y"*. Bien que l'on puisse cependant conserver une déclaration correcte de la seconde clause *"maximum(X, Y, X) :- X >= Y"* qui permettrait d'obtenir une sémantique déclarative et le coupe-choix pour une optimisation. Dans la programmation du prédicat *maximum/3* profite du coupe-choix pour éliminer le test sur la valeur des variables X et Y au moment de l'exécution. Un tel programme, même s'il possède un sens opérationnel, n'est plus déclaratif et l'interprétation d'une solution incorrecte due à une mauvaise programmation devient difficile pour un utilisateur non averti.

En revanche, une écriture déclarative du prédicat *maximum/3* peut être faite à l'aide d'un contrôle de haut niveau, le *si-alors-sinon*, de la façon suivante :

```
maximum(X, Y, Z) :-
    X < Y
    -> Z = Y
    ; Z = X.
```

Ainsi, il faut, pour rendre la sémantique de ses programmes la plus déclarative, orienter le programmeur vers une utilisation de constructions déclaratives de haut niveau, telles que : le *si-alors-sinon*, le *différent*, la *coupure sûre*, la *négation sûre*, la *négation complète*. Il faut donc que l'utilisateur acquiert une méthodologie de programmation qui ne pourra qu'améliorer et rendre plus efficace la phase de mise au point. Cette méthodologie ne peut être obtenue que lors de l'apprentissage ou lors de séminaires sur le langage Prolog.

Enfin, les divers types d'utilisation de la coupure doivent être examinés. Le lecteur pourra se reporter, à ce sujet, aux nombreuses références existantes [Lloyd-87, Shapiro-Sterling-86, Emden-82]

Notre conclusion, sur ce problème du coupe-choix, est que pour utiliser de façon correcte un système de détection d'erreurs dans le cas d'une solution insatisfaite, ce système doit effectuer une recherche en profondeur d'abord. Nous obtenons ainsi une contrainte à respecter pour réaliser une système intelligent de mise au point pour Prolog.

6.3 La négation

Le problème de l'utilisation correcte de la négation est complexe. En effet, comme nous l'avons expliqué, la négation par l'échec n'est pas une négation logique.

Par exemple, nous avons un échec de résolution du but : "$\neg(X = 1), X = 2$", lorsque X est une variable libre. En revanche, une négation logique nous donnerait une réussite de ce but avec comme solution "$X = 2$", de même que la résolution du but "$X = 2, \neg(X = 1)$".

Ainsi, la sémantique opérationnelle de la négation, rend celle-ci non logique. Pour éviter de tels problèmes, certains systèmes utilisent une mise en attente des buts [NU-Prolog].

Dans le cas qui nous préoccupe, celui de la mise au point et plus particulièrement de la détection intelligente d'erreurs, nous connaissons cette sémantique de la négation par l'échec. Il convient donc d'analyser de façon correcte les diverses particularités de cette négation. Nous avons vu que nous pouvons nous placer dans un contexte particulier : le monde clos. Dans ce contexte le monde fini et nous pouvons contrôler de façon fine l'évolution de l'interprétation. Une conséquence particulière de cette hypothèse est qu'un ensemble de solutions doit être fini car c'est un sous ensemble d'un monde fini. De même, toute dérivation doit être finie.

Une étude sommaire de l'échec de la négation d'un but, nous montre que ce but admet une solution calculée. Deux cas peuvent se produire : soit cet échec de la négation est correct, soit il est incorrect.

Échec correct :

> le programmeur est capable de dire qu'il existe une solution pour le but. Doit-on vérifier dans ce cas que la solution calculée est bien celle que connaît le programmeur ? La bonne question serait plutôt : "*La solution du but est elle une solution incorrecte ?*" Nous pensons que la so-

lution de la négation doit être donnée à l'utilisateur. S'il estime que cette solution est fausse, alors nous devons réaliser une mise au point sur ce but. Si cette solution est correct, la négation sera considérée comme étant correcte.

Il est cependant nécessaire de remarquer que le problème initial de la mise au point subsiste. L'algorithme de recherche doit se poursuivre, car la négation du but doit réussir selon le programmeur et comme elle admet une preuve, les raisons de l'erreur sur le but initial subsistent. Nous avons donc un sous-problème à résoudre : *la preuve de la négation*. De cette façon, si la résolution d'un but est une résolution correcte selon l'interprétation minimale du programmeur, la négation de ce but doit être considérée comme correcte. Nous devons, pour résoudre ce problème, mettre au point une technique de preuve de la résolution.

Échec incorrect :

la résolution du but réussit alors qu'elle devrait échouer. Pour ce but il existe une solution fausse. Nous devons alors effectuer une détection d'erreurs sur une solution fausse à partir du but nié. Ceci peut être réalisé par une méthode de vérification de la résolution permettant de détecter une solution incorrecte. Lorsque la vérification de la résolution retourne une solution fausse, la négation est une négation incorrecte et la détection s'est arrêtée sur la position de l'erreur. En revanche, lorsque cette vérification est correcte, c'est-à-dire qu'il n'existe pas une solution incorrecte dans l'arbre de recherche de la négation, on se trouve dans le premier cas car la solution est une solution correcte et l'échec de la négation serait correct.

De même, lorsque la négation d'un but réussit, la résolution du but ne produit aucune solution. Nous avons deux cas pour cette réussite :

Réussite correcte :

Le programmeur sait qu'il n'existe pas de solution pour la résolution de ce but. Pour les mêmes raisons que précédemment, rien ne garantit que l'arbre de résolution soit correct. Lorsque la résolution de ce but est correcte, une vérification de cette résolution ne retourne aucune solution incorrecte.

Réussite incorrecte :

Le programmeur connaît une solution insatisfaite. La résolution du but est une résolution incorrecte et une vérification de cette résolution permet de trouver une solution incorrecte dans le programme.

Cependant, il est parfois utile d'analyser l'utilisation d'une négation. D'une façon générale le programmeur doit connaître les problèmes que pose la négation par l'échec. La détection d'erreurs, dans ce cas, pourrait être une aide à l'utilisation de la négation d'une façon sûre. Pour illustrer les difficultés d'une telle entreprise, considérons le petit exemple suivant :

```
% humain(X) <==> X est un humain
humain( jean ).
humain( paul ).
humain( pierre ).

% père(X,Y) <==> X est le père de Y
père( jean, paul ).
père( pierre, paul ).

% sans_enfant(X) <==> X ne possède pas d'enfant
sans_enfant(X) :-
    not père(_, X).
```

Nous sommes dans un cas d'utilisation d'une négation non sûre. En effet, si nous posons la question *"sans_enfant(X)"*, il n'existe pas de réponse car le but *"père(_,X)"* réussit au moins une fois avec pour solution "*X = paul*" et donc il y a échec du but *"sans_enfant(X)"*. En revanche, les questions *"sans_enfant(jean)"* et *"sans_enfant(pierre)"* réussissent. Dans un cas tel que celui-ci, l'utilisation de la négation pourrait être vue comme étant une contrainte sur la variable X, la déclaration faite sur le prédicat *"sans_enfant/1"* ne permet pas de résoudre une telle contrainte car il n'y a pas de définition de l'ensemble auquel appartient X, X peut être n'importe quoi. Dans l'hypothèse du monde clos, il doit être possible de donner l'ensemble des valeurs possible de la variable X.

Nous devons analyser ce problème d'un point de vue logique. Il faut pour cela reprendre la déclaration du prédicat *sans_enfant/1* pour obtenir une version plus proche de la réalité et qui sera plus correctement interprétée. Nous pouvons exprimer qu'une personne possède un enfant de la façon :

Quel que soit l'humain X,
    X possède un enfant s'il existe Y
    tel que Y est un humain et X est le père de Y.

L'incohérence de la précédente déclaration vient de l'utilisation d'une variable muette à l'intérieur de la négation. La relation de lien de parenté entre deux individus X et Y peut se traduire par la déclaration :

```
a_un_enfant(X) :-
    humain(X),
    père(X, Y),
    humain(Y).
```

Maintenant, exprimons qu'un individu ne possède pas d'enfant par la phrase :

quel que soit X, X ne possède pas d'enfant
    si X est un humain et X n'a pas d'enfant.

qui se traduit par la déclaration :

```
pas_de_enfant(X) :-
    humain(X),
    ¬ a_un_enfant(X).
```

sans_enfant/1 n'exprime pas la même relation sur le terme X que ce prédicat. Cette réalisation utilise une négation sûre sur le but *"a_un_enfant(X)"* par ajout d'une contrainte sur la variable X.

Cet exemple nous montre que l'utilisateur doit toujours programmer son application de façon déclarative afin d'obtenir une négation close, permettant d'approcher une négation logique. Bien souvent, le programmeur utilise une négation non close, parce qu'il omet les déclarations apparentées à des contraintes qui rendrait cette négation close. Une négation est close si lors de la réussite du but aucune variable libre n'a pas été substituée par un terme.

Ainsi, l'utilisation d'une négation dans des programmes Prolog ne simplifie pas la mise au point. En effet, seule une analyse de la résolution du but de la négation est possible et le système de détection d'erreurs ne peut pas connaître la logique de la négation qu'utilise le programmeur. De plus, nous devons étudier en détail les problèmes de vérification d'une résolution Prolog.

En effet, supposons que nous disposions d'un système permettant de fournir la preuve d'une résolution par une absence d'erreurs. Lorsque nous vérifions l'utilisation d'une négation, si la résolution du but nié est une résolution correcte, nous considèrons l'utilisation de la négation comme correcte lorsque la négation logique du but. Il faut comprendre par négation correcte : la réussite ou l'échec de la négation est correct.

Dans le cas où la résolution du but est incorrecte, une vérification de cette résolution fournit une solution incorrecte.

Le sujet central de cette thèse est de pouvoir répondre à la question énoncée par la phrase :

> Pouvons nous définir une méthode de vérification d'une résolution normale et permettant de détecter une solution incorrecte ?

Les réalisations actuelles de détection d'erreurs sont relativement impuissantes sur le traitement de la négation et plus particulièrement sur la preuve de la résolution. De plus, nous devons remarquer que les méthodes de recherche en largeur d'abord sont imparfaites sur le traitement de la négation.

Considérons un programme utilisant au moins une négation et qui posséde une résolution incorrecte. Considérons un but ayant une solution fausse (respectivement une solution insatisfaite) et contenant un sous-but nié, de la forme "¬ *f(X)*". Supposons que la résolution du but nié f(X)soit incorrecte. Dans le cas où cette négation échoue, nous pouvons affirmer qu'il existe une solution pour le but "¬ *f(X)*", par définition de la négation par l'échec. Si cette solution est fausse, la résolution est bien incorrecte, mais par

contre l'échec est correct (le programmeur connaît une solution particulière de ce sous-but). Dans ces conditions particulières, l'algorithme de recherche de la solution fausse ne questionnera pas l'utilisateur pour savoir si l'échec est correct (respectivement pour la solution insatisfaite, ce n'est pas le but recherché car l'échec est valide).

Ainsi, ces deux méthodes de détection d'erreurs, dans leur réalisation en largeur d'abord, sont inefficaces sur des programmes contenant la négation d'un but ayant une solution incorrecte.

En conclusion, pour effectuer une mise au point efficace sur des programmes utilisant la négation ou la coupure, seule une méthode en profondeur d'abord est compatible avec ces deux extensions. Une détection d'erreurs nous oblige à fournir une méthode de recherche en profondeur d'abord fondée sur la vérification de la résolution qui permettra de découvrir une solution fausse ou une solution insatisfaite dans toutes les circonstances.

6.4 Extensions

Dans cette partie nous étudions les extensions des méthodes de détection d'erreurs en tenant compte des prédicats prédéfinis. Le but n'est pas de donner une solution au traitement de chaque prédicat mais seulement de déterminer le domaine d'application des méthodes de détection d'erreurs que nous avons décrites. Ces extensions peuvent être découpées en trois parties : les prédicats logiques pouvant être écrits en Prolog pur, les prédicats extra-logiques ou du second ordre, les autres prédicats pouvant être écrits en Prolog étendu par la négation et la coupure.

6.4.1 Importance

L'extension des méthodes de débogage est rendue nécessaire par le fait que tout programmeur Prolog utilise de façon courante les prédicats prédéfinis. La diversité de ces prédicats rend l'extension difficile. On peut cependant répartir ces prédicats en sous-ensembles distincts. Nous utilisons pour cela les références [Cprolog-1.5, Sbpl-86] comme base de définition des prédicats prédéfinis et nous étudions pour chaque sous-ensemble les possibilités de débogage.

6.4.2 Prédicats logiques et méta-logiques

Ces prédicats possèdent un comportement *logique*. Quelles que soient leurs circonstances d'appel l'interprétation de l'un de ces prédicats est unique. Si nous prenons par exemple le prédicat prédéfini *"Var(X)"*, son interprétation est indépendante de l'environnement d'exécution (programme, fichiers ouverts...). Le but *"Var(X)"* réussit si et seulement si X est une variable.

De plus, généralement les prédicats prédéfinis possèdent des spécifications d'appel. Si elles ne sont pas respectées, il peut se produire un arrêt brutal de la résolution. Ce cas d'erreur correspond à l'appel incorrect d'un prédicat prédéfini. Dans le cas de la mise au point de Prolog, ce problème ne se produit que rarement. Cependant, lorsqu'il existe

une erreur de ce type, il est intéressant de connaître le déroulement de la résolution jusqu'à l'appel incorrect du prédicat prédéfini. Pour des cas particuliers tels que ceux-ci, il est important de posséder un outil de trace intelligent pour suivre pas à pas l'interprète Prolog.

Les méthodes de détection d'erreurs sont possibles sur des programmes Prolog contenant des prédicats logiques ou méta-logiques. En effet, l'application des méthodes de détection d'erreurs sur ces prédicats repose sur l'indépendance de l'interprétation des prédicats méta-logiques par rapport à la résolution. Les méthodes de mise au point réalisent un parcours partiel de l'arbre de recherche de la solution incorrecte. Ceci aspect évite d'éventuels retour-arrière de la résolution Prolog, ou bien dans le cas de la méthode en largeur d'abord, les appels multiples d'un but et le parcours partiel de l'arbre de recherche. Le comportement logique de ces prédicats méta-logiques ne modifie pas le raisonnement logique du détecteur d'erreurs Prolog.

En vertu de ce principe, si l'interprétation d'un prédicat prédéfini est indépendante de son environnement, les méthodes de mise au point restent applicables à des programmes en contenant. Il suffit de considérer ces prédicats comme corrects. S'il existe une solution son interprétation est qu'elle est correcte sinon il n'y a pas de solution.

6.4.3 Prédicats extra-logiques

Dans cet ensemble, il faut inclure tous les prédicats dont l'interprétation est *aléatoire*. L'interprétation dépend de l'environnement d'exécution et plus particulièrement de la sémantique opérationnelle de la résolution. Dans ce cas, les méthodes de détection d'erreurs ne peuvent s'appliquer directement sur ces prédicats car leur comportement n'est pas logique. En fait, il n'est pas répétitif, si p admet une solution, rien ne garantit que cette solution sera produite par un autre appel de p.

Lorsque l'on réalise un méta-interprète de Prolog, on doit tenir compte de ces prédicats pour effectuer une simulation complète du langage et tenir compte des exeptions d'appels. De plus, dans certains cas, à cause de leurs effets de bord, l'application stricte des méthodes de détection d'erreurs, nous donne des résultats incorrects. Pour l'illustrer considérons l'exemple suivant :

```
test :-
    p(X),
    essai.

p(0).
p(1).

b(3).

essai :-
    appel.

appel :-
    compter(X),
    b(X).
```

```
compter(X) :-
    retract(s(X)),
    !,
    Y is X + 1,
    asserta(s(Y)).

compter(0) :-
    asserta(s(0)).
```

Il est clair que ce programme ne possède pas une sémantique opérationnelle identique à sa sémantique déclarative. L'appel du prédicat *test* ne donnant aucune solution, supposons qu'il y en ait une pour laquelle *"compter(3)"* serait une solution correcte. L'erreur se situerait à la valeur par défaut du prédicat *"compter/1"* et il faudrait une solution à *"compter(1)"*. La trace interactive de la détection d'erreurs est :

```
| ?- lire(exemple),résoudre(test).
Chargement du fichier : exemple.pl
Pas de solution sur le but test.
OK ? n.
Diagnostic, Recherche d'une solution insatisfaite

Question : Sur le but : test
Une solution ? oui.

Question : Sur le but: p(_600)
Une solution ? oui.
Avec : _600 ? 1.

Question : Sur le but : essai
Une solution ? oui.

Question : Sur le but : appel
Une solution ? oui.

Question : Sur le but : compter(_964)
Une solution ? oui.
Avec : _964 ? 3.

Question Sur le but: b(3)
Une solution ? oui.
Yes

| ?-
```

Nous n'observons aucune réponse du détecteur d'erreurs sur cette résolution Prolog. Le système ne trouve aucune incohérence dans la résolution du but initial. Cet effet de bord est dû à l'algorithme de détection d'erreurs en largeur d'abord pour l'absence de solution.

Ces méthodes ne peuvent donc pas être utilisées sur ces prédicats extra-logiques. De plus, si on réalise une détection d'erreurs par une méthode en profondeur d'abord, quelle diagnostic doit-on donner sur la solution d'un prédicat extra-logique : la réussite est-elle correcte, l'échec est-il correct ? Les spécifications de ces prédicats méta-logiques ne permettent pas de répondre de façon précise à ces questions car c'est à la fois correct et incorrect. Nous reviendrons sur ce problème dans la troisième partie.

Comme nous l'avons vu dans la partie précédente, les algorithmes de mise au point classiques parcourent en partie l'arbre de résolution d'un but. Comme, l'utilisation des prédicats extra-logiques peut modifier l'arbre de recherche en fonction du contexte. Le raisonnement de l'algorithme de détection d'erreurs se trouve modifié au fur et à mesure que se transforme l'arbre recherche du but initial.

6.5 Conclusion sur les extensions

L'étude que nous venons de faire sur la mise au point de programmes utilisant des négations, des coupures ou des prédicats méta-logiques, basées sur les méthodes de Shapiro, nous permet de mieux cerner les problèmes à résoudre. Elle nous conduit également à étudier une méthode générale de détection d'erreurs.

Cette méthode correspond à la vérification de la résolution par une recherche en profondeur d'abord des deux cas d'erreurs, la solution fausse et la solution insatisfaite. Elle permet la mise au point, plus précisément la preuve de résolution de programmes Prolog méta-logiques.

Enfin, nous avons mis en évidence la nécessité de définir une trace mieux structurée par les informations qu'elle fournit, permettant de suivre la résolution d'un programme extra-logique.

La troisième partie aborde le problème d'une méthodologie de mise au point pour des programmes Prolog. En effet, l'utilisateur final devra effectuer un choix de l'outil qu'il appliquera à ses programmes. Nous étudierons et décrirons ensuite un système de mise au point de programmes méta-logiques regroupant deux fonctions essentielles : la détection de boucles et la vérification de la résolution.

La quatrième partie décrit un prototype utilisant de tels concepts de mise au point. Ce prototype a été doté d'un interface particulièrement adaptée à la tache de mise au point dans un contexte aussi particulier que celui de Prolog.

Partie 3

La mise au point

Page blanche

7. Méthodologie

Dans la partie précédente, nous avons étudié les problèmes de la mise au point en Prolog en utilisant une approche voisine de celle de Shapiro. L'algorithme de recherche est fondé sur une connaissance du comportement que doit avoir un débogueur intelligent dans la résolution d'un but incorrect.

Comme de telles méthodes de mise au point sont inutilisables sur des programmes Prolog quelconques, les algorithmes sont restreints à Prolog pur étendu par la coupure, la négation sûre et enfin les prédicats logiques ou méta-logiques. S'il utilise des prédicats extra-logiques, le programmeur souhaitera disposer d'un outil de mise au point intelligent voisin de celui-ci. Il convient donc d'examiner une approche méthodologique de la mise au point en Prolog afin de répondre à cette attente. Cette approche de la mise au point, nous pouvons la faire en utilisant les raisonnements des algorithmes de recherches.

Pour situer les limites d'un détecteur d'erreurs intelligent, nous allons décrire un exemple de programme extra-logique, un système expert sur la reconnaissance des animaux. Puis nous étudierons une approche déclarative de la mise au point en utilisant la trace de Eisenstadt et en simulant les techniques de détection d'erreurs au moyen d'une trace partielle de la résolution (Zooming).

7.1 Quelle mise au point ?

L'utilisateur qui déceloppe une application Prolog peut s'interroger sur le mode d'emploi d'outils sophistiqués (traces intelligentes, détecteurs d'erreurs interactifs...). Il est évident que, dans certains cas, un outil de type trace de la résolution constitue une aide à la mise au point. C'est le cas par exemple lorsque le programmeur ne sait pas comment il doit interpréter une solution ou une absence de solution.

L'utilisation des méthodes de mise au point de Shapiro suppose une connaissance parfaite de l'interprétation d'un but. L'utilisateur doit répondre sans ambiguïté aux questions du détecteur d'erreurs. Cependant, il peut exister des programmes dont justement on ne connaît pas le comportement. De plus, en général le programmeur utilise des prédicats extra-logiques, qui provoque des effets de bords de l'interprète.

Pour les différentes méthodes de mise au point il faut analyser les avantages et les inconvénients de l'un par rapport à l'autre. Quand l'un est-il plus performant, plus adapté que l'autre ? Pour un problème particulier de mise au point quelle est la méthode adaptée, quels sont les raisonnements logiques que nous devons effectuer ?

7.2 Un exemple

L'utilisation de prédicats extra-logiques dans une application est relativement courante : primitives de lecture ou écriture, prédicats du second ordre, etc. Le cas se produit par exemple pour réaliser un système expert. La littérature nous montre que la programmation logique est particulièrement puissante et bien adapté pour la réalisation

de systèmes experts [Sterling-84,86, Clark-Cabe-82]. Cette propriété est due au fait qu'un système expert est la réunion d'une base de connaissances et d'un moteur d'inférence, dont la réalisation est facilité par l'utilisation de l'évaluation partielle ou de la méta-programmation

Si, de plus, on l'augmente par des extensions objets le langage Prolog permet de mixer d'une façon efficace le moteur d'inférences et les représentations de connaissance [LOO-89].

Prenons pour exemple le programme *"résout(T est V)"*, dont la définition complète est donnée en annexe A, qui cherche à démontrer que le terme T est de type V. Les connaissances sont exprimées de la façon suivante :

si T_1 est T_2
et T_3 est T_4
et ....
alors T_P est T_Q.

Par exemple, pour un carnivore, nous aurons la règle :

si classe est mammifères
et mange_viande est oui
alors ordre est carnivore.

Dans cet exemple d'application, le moteur d'inférence détermine le type d'un animal. Par exemple, pour le lion, on a la règle :

si espèces est 200
et terrestre est oui
et en_afrique est oui
alors type_animal est lion.

Si aucune règle ne permet de déduire de nouvelles informations, le système peut poser des questions à l'utilisateur. Ces questions sont représentées sous la forme de faits. Par exemple :

```
question(mange_viande,
    'Votre animal mange-t-il de la viande rouge ?').
```

Toutes les déductions que le système obtient sont conservées sous la formes de faits qui décrivent comment elles ont été obtenues. Leur forme générale est :

```
% "F est X" est déduit par une question
fait(F est X, question).

% "F est X" est déduit par la règle N
fait(F est X, règle(N)).
```

Enfin, si la valeur du fait F n'a pu être déduite, il y a ajout d'un fait *"échec(F)"*. Pour démontrer un fait, le moteur d'inférence utilise la stratégie de déduction reposant sur quatres tentatives hiérarchisées :

- le fait est-il déjà démontré ?
- le fait est-il faux (échec) ?
- s'il existe une règle la concluant sur ce fait alors on cherche à démontrer les prémisses de cette règle,
- s'il existe une question prévue pour ce fait, on interroge l'utilisateur du système expert,
- on inscrit l'échec si tous les quatres essais ne marche pas.

L'utilisation de ce moteur sur un exemple simple nous donne comme résolution :

```
| ?- résout(type_animal est lion).
Votre animal a-t-il une colonne vertébrale ? oui.
Votre animal a-t-il le sang chaud ? oui.
La femelle de votre animal nourrit-elle son petit avec
du lait ? oui.
Votre animal mange-t-il de la viande rouge ? oui.
Votre animal a-t-il des doigts ailes ? non.
Votre animal a-t-il un pouce opposable ? non.
Votre animal adulte pèse-t-il plus de 200 Kg ? oui.
Votre animal est-il terrestre ? oui.
Votre animal vit-il en Afrique ? oui.
Yes
```

Nous laissons au lecteur le soin de tracer cette résolution du but *"résout(type_animal est lion)"*. La trace d'Eisenstadt sur cet exemple, donne de l'ordre de 150 messages avant que la première question soit posée et environ 500 messages pour toute la résolution soit une dizaine de pages de texte.

Ainsi, une trace ne permet pas de réaliser aisément la mise au point du système expert. Si l'on veut comprendre la résolution de l'interprète Prolog ou si l'on modifie des connaissances avec pour effet de produire une résolution incorrecte, quelle méthodologie de mise au point doit-on adopter ? En effet, nous avons vu dans la partie précédente qu'une trace ne peut d'être d'aucun secours et que sur de tels programmes les extensions des méthodes de mise au point sont inutilisables.

L'analyse globale des problèmes de la mise au point des programmes extra-logiques nous montre qu'il faut une trace dynamique, comme par exemple par l'utilisation des méthodes de Shapiro [Ducassé-88], ou statique fondée sur la trace d'Eisenstadt, et qui permet de visualiser les points fort de la résolution : but en cours d'évaluation, clause activée, instances de ses arguments, but suivant à appeler, ensemble des buts ancêtres du but courant...

7.3 Une mise au point statique ?

Dans les parties précédentes nous avons étudié les différentes techniques de mise au point sans les comparer entres elles. Pourquoi ne pourrait-on pas réaliser une mise au point intelligente en utilisant une trace comme support du raisonnement logique. Nous étudierons cet aspect avec la trace de Eisenstadt. En effet, les possibilités de la trace de Byrd sont bien trop pauvres pour réaliser une trace partielle comme le Zooming.

Cependant, il est possible de simuler des traces partielles en espionnant les prédicats (*spy/1*), en réitérant l'appel d'un sous-but, en simulant un échec... L'utilisation de telles méthodes sur des programmes extra-logiques, avec la trace de Byrd, produit des comportements anormaux de la résolution, similaires à une technique de détection d'erreurs en largeur d'abord. De plus, l'espionnage ne permet pas de réduire de manière satisfaisante la trace. Dans l'exemple précédent, si l'on fait un espionnage du prédicat "résout" presque toute la totalité de la résolution est tracée.

Reprenons l'exemple de Shapiro [Shapiro-83] de la page 41, modifié pour tenir compte du coupe choix, et effectuons un parcours partiel de la résolution pour produire une trace partielle par une technique de Zooming. Nous pouvons utiliser une technique "Top-Down", de la façon suivante :

```
| ?- analyse(trier([2,1,3],X)).
1   :   ?-  trier([2,1,3],_433)
33  :   +   trier([2,1,3],[3,2,1]) [1]
34  :   ^   trier([2,1,3],[3,2,1]) [1]
52  :   -   trier([2,1,3],_433)
non

| ?- zoom.
1   :   ?-  trier([2,1,3],_399)
3   :       ?-  trier([1,3],_481)
20  :       +   trier([1,3],[3,1]) [1]
21  :       ?-  insérer(2,[3,1],_482)
32  :       +   insérer(2,[3,1],[3,2,1]) [1]
33  :   +   trier([2,1,3],[3,2,1]) [1]
Yes

| ?- zoom(3).
3   :   ?-  trier([1,3],_418)
5   :       ?-  trier([3],_494)
11  :       +   trier([3],[3]) [1]
12  :       ?-  insérer(1,[3],_495)
19  :       +   insérer(1,[3],[3,1]) [1]
20  :   +   trier([1,3],[3,1]) [1]
Yes

| ?- zoom(12).
12  :   ?-  insérer(1,[3],_421)
14  :       }   3>1
15  :       ++  3>1
```

```
16  :       ?-  insérer(1,[],_495)
17  :       +*  insérer(1,[],[1]) [3]
18  :   ---------------- !
19  :   +   insérer(1,[3],[3,1]) [1]
Yes

| ?-
```

Sur le premier zoom, le premier sous-but possède une solution fausse (ligne 20); on effectue une trace partielle de son appel, zoom(3) nous donne le premier sous-but incorrect (ligne 19) qui est tracé par son appel zoom(12), dans ce cas nous avons un but ayant une solution fausse (ligne 19) dont tous les sous-buts sont justes. Nous sommes donc sur une clause fausse que nous identifions à partir de la ligne 19 comme étant la clause 1 du paquet *insérer/3*.

Comme nous le voyons sur cet exemple de mise au point utilisant la trace de Eisenstadt, les raisonnements logiques d'un détecteur d'erreurs intelligent comme celui de Shapiro, peuvent être simulés par des outils de trace. Cependant, il convient de mesurer la portée d'une telle entreprise. Même si l'on peut simuler une méthode de détection d'erreurs en largeur d'abord, celle-ci doit être appliquée par l'utilisateur et il est à remarquer que la trace partielle peut contenir un nombre importants de messages à interpréter.

Ainsi, l'application stricte de cette technique nécessite une connaissance parfaite de la trace et des principes de mise au point en Prolog. En effet, l'utilisateur doit être vigilant pour pouvoir repérer les différents niveaux sur lesquels il devra réaliser une trace partielle et connaître les différents raisonnements logiques qu'il devra effectuer sur tous les sous-buts, pour déterminer l'action suivante : "*zooming* sur quel but ?"

En revanche, si cette méthode est directement réalisée par un système de détection d'erreurs intelligent, seule l'attention de l'utilisateur est portée sur la connaissance des réponses à donner, c'est-à-dire la connaissance de l'interprétation sous-jacente de son application. Ainsi, l'utilisation d'un détecteur d'erreurs sera plus efficace dans la mise au point de gros programmes méta-logiques.

7.4 Une mise au point dynamique ?

L'étude précédente nous montre qu'il est possible de concevoir une méthode de mise au point basée sur le parcours de l'arbre de résolution, comme le font les extensions de la trace d'Eisenstadt. Une façon de procéder consiste à résoudre le but et à chaque pas de la résolution à mémoriser des informations pour construire l'arbre de résolution. Le système de détection d'erreurs parcourt cet arbre en appliquant directement dessus les raisonnements logiques des méthodes de Shapiro.

En fait, cette méthode pose des problèmes de connaissances.Dans l'ensemble, les méthodes de Shapiro pré-supposent une connaissance parfaite du comportement de l'interprétation d'un but : admet-il une solution ? est-il juste ? est-il faux ? Il faudra répondre d'une façon précise à ces questions.

Ainsi, dans le cas d'applications comme une base de donnée relationnelle ou un système expert, cette connaissance est parfois imparfaite. Le developpement d'un système expert demande la réunion d'au moins deux entités : l'expert et le cogniticien [Vial-88]. Lors de la phase de mise au point, le système de détection d'erreurs demandera des informations sur l'interprétation d'un but : lequel possède la connaissance pour répondre ? Le cogniticien s'il sait pourquoi on lui pose une question sur un but ne sait pas forcément comment y répondre car il ne possède pas toute l'expertise nécessaire pour cela. Quant à l'expert, s'il sait pourquoi telle connaissance est vraie ou fausse, s'il existe une solution ou non, il ne sait pas comment interpréter les messages du détecteur d'erreurs, par exemple ceux qui relèvent de l'exécution du moteur d'inférence...

Un autre problème limitant les possibilités des algorithmes, est la détection d'erreurs sur des programmes utilisant des prédicats extra-logiques. Par exemple, le système expert présenté à l'anexe A, n'a pas un comportement logique en fonction des informations fournies par l'utilisateur. En effet, lors de la résolution du but *"résout(type_animal est Quoi)"*, les réponses aux questions posées par le système modifient l'arbre de résolution et donc le comportement du système expert. Comme l'algorithme de détection d'erreurs travaille sur l'arbre de résolution (parcours partiel de la résolution, connaissance des solutions d'un but), les raisonnements qu'il effectue à un instant donné ne seront plus valables l'instant suivant. Par exemple, un but qui a échoué au premier appel, réussit à l'appel suivant. Aussi, les conclusions du système de détection d'erreurs, quand elles existent, n'ont aucune signification.

De plus, même si le programmeur sait comment interpréter les résultats de la résolution d'un but, les modifications s'effectuant en cours de résolution ne lui permettent pas de pouvoir répondre à une question du détecteur d'erreurs. Par exemple, il sait qu'un but selon les circonstances admet ou n'admet pas de solution et que le système lui pose la question : *"ce but admet-il une solution ?"*, que doit-il fournir au système de détection ? Comment doit se comporter le système de détection d'erreurs en fonction des réponses ?

Il est donc clair que si conceptuellement nous pouvons effectuer la mise au point, celle-ci bute contre l'insuffisance des réponses qui peuvent être fournies. La réalisation d'un système de détection d'erreurs n'est donc pas possible sur des programmes extra-logiques. Il faut pour ces cas-là définir un outil de trace interactif et présentant suffisamment d'informations sur la résolution.

Un autre de problème survient quand on se limite à de telles méthodes de mise au point. Il est illustré sur le tri croissant rapide :

```
trier([],[]).

trier([X|Xs],Es) :-
    séparer(Xs, X, Lo, Hi),
    trier(Lo, S_Lo),
    trier(Hi, S_Hi),
    append(S_Lo, S_Hi, Es).
```

```
séparer([X|Xs], Cr, [X|Lo], Hi) :-
    X < Cr,
    séparer(Xs, Cr, Lo, Hi).

séparer([X|Xs], Cr, [X|Lo], Hi) :-
    Cr > X,
    séparer(Xs, Cr, Lo, Hi).

séparer([],Cr,[],[]).
```

Le but initial *"trier([2,1,3],R)"* ne nous fournit pas de solution. Nous sommes en présence d'une solution insatisfaite, *"R=[1,2,3]"*.

L'algorithme de recherche en profondeur d'abord sur ce but nous retourne l'échec du sous-but *"séparer([3], 2, _1, _2)"*. Dans ce cas, l'utilisation complémentaire de la trace d'Eisenstadt nous donne beaucoup plus de renseignements sur l'échec :

```
1   :   ?-  séparer([3],2,_1,_2)
2   :   >   séparer([3],2,[3|_3],_2) [1]
3   :       }   3 < 2
4   :       --  3 < 2
5   :   <   séparer([3],2,[3|_3],_2) [1]
6   :   >   séparer([3],2,_1,[3|_4]) [2]
7   :       }   2 > 3
8   :       --  2 > 3
9   :   <   séparer([3],2,_1,[3|_4]) [2]
10  :   -   séparer([3],2,_1,_2)
```

Ainsi, l'analyse de cette trace nous a permis de corriger rapidement l'erreur de ce programme (Cr ≤ X dans la seconde clause du prédicat *séparer/4*).

Cet exemple de mise au point de programme incorrect nous situe les limites d'un système de détection d'erreurs. En complément d'un système de détection d'erreurs, il est intéressant de fournir à l'utilisateur une trace permettant d'affinner l'analyse d'une cause d'erreur.

En effet, lorsque l'utilisateur débute en programmation logique, il doit acquérir sur des programmes simples, une méthodologie de mise au point. Un détecteur d'erreurs lui en fournit une partie. Mais lorsque le système lui retourne une erreur, l'utilisateur doit analyser correctement les circonstances de l'appel.

Ainsi, il faut fournir en complément d'une solution incorrecte les circonstances dans lesquelles elle apparaît. Les techniques actuelles ne donne que la clause incorrecte associée à une solution fausse. Dans le cas d'une solution insatisfaite que peut-on fournir ? Un certain nombre d'informations seulement. Par exemple, lorsqu'une solution insatisfaite est détectée, on peut indiquer s'il existe des clauses dont les têtes s'unifient. Mais plus généralement, doit-on fournir des informations sur l'échec des prédicats prédéfinis ? Doit-on analyser les clauses dont les têtes s'unifie ?

Les réponses à de telles questions ne sont pas possibles. Il faut toujours, pour un tel logiciel, donner un minimum d'informations judicieuses, sans tout donner. Le système de mise au point peut fournir un ensemble d'outils spécifiques pour les traitements de cas particuliers, comme la trace d'Eisenstadt peut le faire par une sorte d'évaluation partielle des buts à partir des solutions connues (analyse statique de la résolution).

7.5 Conclusion sur la méthodologie

Le programmeur expérimenté, grace à trace intelligente peut vérifier de façon précise le contrôle de la résolution. Pour le programmeur débutant, le système de détection d'erreurs permet d'exercer ses raisonnements logiques, de valider des modélisations de son application, pour obtenir une sémantique déclarative correcte de ses programmes. De plus, la trace permet à un débutant de mieux comprendre la résolution d'un but. Ces aspects correspondent aux objectifs essentiels d'un système intelligent de développement pour le langage Prolog.

Une système intelligent de détection d'erreurs couplé à une trace intelligente est l'outil parfait pour une langage de programmation méta-logique (Prolog pur avec la négation, la oupure et les prédicats méta-logiques).Une trace intelligente fournissant un minimum d'informations sur la résolution, permettrai de pallier les inconvénients d'une trace classique.

Les deux exemples que nous avons utilisés, nous montrent que la combinaison judicieuse des deux outils de mise au point, trace intelligente et détection d'erreurs, est productive pour le programmeur d'application Prolog. Celui-ci programme de façon déclarative et modulaire e tl'emploie des sous-programmes méta-logiques. Quelle que soit la méthodologie de mise au point, le programmeur doit séparer dans son programme les résolutions extra-logiques du reste de l'application.

Cette méthodologie peut se situer à deux niveaux.

Le premier correspond à la sémantique déclarative des programmes Prolog. En effet, il est relativement courant que les programmeurs utilisent les primitives extra-logiques et une mauvaise connaissance du comportement de ces prédicats produit des résultats anormaux.

Le second concerne de la mise au point et le suivi d'une application. En effet, le programmeur doit conserver à l'esprit, lors de la réalisation de son application, qu'il aura une mise au point à faire. Cette méthodologie de programmation doit le conduire à découper son application en sous-modules méta-logiques dont chaque programme est la réalisation d'un prédicat méta-logique. Ensuite, il veillera à encapsuler l'utilisation de prédicats méta-logiques afin d'éviter leurs effets de bord.

Ensuite, lors de la phase de mise au point de son application, s'il a au préalable encapsulé tous les prédicats corrects par une compilation, le programmeur peut détecter de façon *rationnelle* les problèmes de son application, à l'aide des techniques de détection d'erreurs comme celle que nous avons présentées, ou bien de méthodes de trace partielle de certains prédicats. Par exemple, il peut ; faire une trace partielle (Zooming)

afin de trouver la position de l'erreur. Déterminer s'il sagit d'un prédicat méta-logique ou d'un prédicat extra-logique ? Puis effectuer la mise au point proprement dite, par l'utilisation de la bonne méthode suivant le cas : détection d'erreurs pour les prédicats méta-logiques, trace intelligente pour les prédicats extra-logiques.

8. La détection d'erreurs

L'étude précédente a mis en évidence la nécessité de disposer d'un outil de mise au point tenant compte de la coupure et de la négation. Elle nous a permis également de cerner les problèmes à résoudre : la détection d'une récursion infinie et la vérification d'une résolution.

Dans cette partie, nous allons étudier les problèmes ds dérivations infinies, mais aussi définir les principes d'une vérification de la résolution. Nous pourrons ainsi définir des algorithmes de détection de boucles régulières et de détection de solutions incorrectes dans une résolution.

8.1 Le bouclage

L'ensemble des interprètes Prolog n'ont pas la possibilité de détecter une dérivation infinie de l'arbre de résolution. Ce phénomène est généralement appelé bouclage et est provoqué par une récursivité infinie. La seule méthode de détection connue repose sur une limitation de la profondeur de l'arbre de dérivation. On peut trouver dans la littérature quelques études de ce problème de la dérivation infinie dans le cas de Prolog pur [Covington-85a-b, Nute-85, Poole-Goebel-85, Shapiro-83]

Cependant Gold [Gold-65] étudie le problème théorique de la limitation de la récursion. De même, Van Gelder [Gelder-87] utilise une méthode intéressante de détection d'une récursion infinie, basée sur un algorithmede parcours d'arbres dû à Knuth. Enfin, Pelhat [Pelhat-87] effectue une étude des différents cas de bouclages en Prolog pur et propose un environnement adapté à la détection de ce type de problème : type du bouclage, création de cycles...

L'aspect déclaratif de la sémantique du langage Prolog peut donner des effets non souhaités lors de la résolution d'un but. Une conséquence directe est la génération d'une dérivation infinie de l'arbre de résolution, due à une récursion infinie.

Dans cette partie, nous proposons une méthode de détection d'une récursion infinie fondée sur la caractérisation des divers cas de bouclages réguliers. Préalablement, nous définissons une sémantique de l'évaluation d'un but par un programme Prolog. Ceci nous permettra de pouvoir gérer l'ensemble des ancêtres d'un but. De cette connaissance, nous caractérisons les divers cas de récursion infinie.

8.1.1 Évaluation d'un but

Le langage Prolog est fondé sur un sous-ensemble de la logique des clauses de Horn et sur une stratégie de l'évaluation, la recherche en profondeur d'abord et l'évaluation de gauche à droite. Il est complété par ensemble de primitives extra-logique de contrôle. Parmi celles-ci figurent la coupure, l'accès à la base de règles (ajout, retrait, consultation...). La stratégie d'évaluation de l'interprète peut être modélisée par l'utilisation d'une pile de buts et d'une liste de buts.

La pile de buts contient l'ensemble des buts en cours d'évaluation, alors que la liste de buts contient l'ensemble des buts en attente d'évaluation. L'évaluation d'un but s'effectue de la façon suivante :

- Le but courant est le premier but de la liste des buts en attente.
- On prend une clause du programme dont la tête s'unifie avec le but courant.
- On empile le but courant dans la pile de buts, avec une référence sur la clause choisit.
- On insère la queue de la clause choisit en tête de la liste de buts.
- Si la liste de buts est vide, il y a réussite de la résolution (le but initial réussit).
- Sinon on réitère l'évaluation.
- S'il n'y a pas d'unification possible, on effectue un retour-arrière sur la dernière évaluation (le but précédent dans la pile) et on réïtère.

Cette modélisation de l'évaluation d'une but est voisine des technique utilisées par la plupart des interprètes Prolog [Boizumault-88].

8.1.2 Le problème de la dérivation infinie

Le problème de la dérivation infinie est relativement courant lors de la mise au point d'un programme Prolog. En effet, la formulation récursive d'une déclaration Prolog peut engendrer une dérivation infinie de l'arbre de résolution. Sur le modèle que nous venons de présenter, elle se traduit par le fait que la liste de buts n'est jamais vide. De plus, nous pouvons remarquer qu'il existe un cycle de la résolution Prolog se manifestant par un cycle dans la pile de buts. En effet, dans l'hypothèse du monde clos, correspondant au fait que les ensembles de solutions et les clauses sont de cardinal fini, le nombre de combinaisons possibles pour obtenir un but est finit, il existe alors dans une branche infinie, deux sous-buts similaires provoquant un cycle dans l'évaluation.

Pour l'illustrer, prenons le programme Prolog suivant :

Exemple 1.

```
R1 : f(a)
R2 : f(X) :- f(X), g(X).
R3 : g(a).
```

Soit le but initial *"f(b)"*. Nous pouvons d'ores et déjà remarquer que l'arrêt de la récursivité, prévu par la clause R1, ne se produira jamais. Nous avons au bout de trois itérations l'état courant suivant :

```
        f(b) , R2
        f(b) , R2
        f(b) , R2
        ---pile---

liste = [ f(b), g(b), g(b), g(b) ].
```

Le cycle se traduit dans les deux structures : le cycle "f(b) -R2→ f(b) -R2→ f(b)" dans la pile, et le cycle sur "g(b)" dans la liste des buts.

Supposons que le programme Prolog considéré engendre une boucle régulière sur l'appel initial d'un but donné. L'interprétation de but initial permet de détecter une boucle. En effet, lorsqu'il existe une récursion infinie de l'évaluation des buts, il existe nécessairement un cycle dans celle-c,produit par un appel récursif. Plus précisément, il existe parmi les ancêtres du but courant, deux buts similaires sur l'instance de leurs arguments, ayant déclenché la même clause. Sur ce principe nous pouvons déterminer une méthode de détection de la récursivité infinie.

8.1.3 Détection d'une récursivité

La méthode de détection de la récursivité infinie est basée sur la comparaison du but courant avec ses ancêtres. Une réalisation naïve de cette méthode consisterait à comparer chaque ancêtre avec le but courant. Il faudrait déterminer si la résolution du but courant déclenche une clause déjà en cours d'évaluation. Il suffirait ensuite de déterminer si les deux buts ne sont pas "similaires", par le fait que leurs arguments sont voisins au sens d'un critère qui reste à déterminer.

Cependant, l'utilisation d'une telle méthode est pénalisante en temps car elle nécessite de réaliser autant de comparaisons qu'il y a d'ancêtres à chaque pas de la résolution. En effet, si la taille de la pile est n, nous avons déjà effectué de l'ordre de n^2 comparaisons avec tous les ancêtres. Gelder [Gelder-87] montre qu'il est possible de pouvoir détecter une récursivité infinie en n'effectuant qu'une seule comparaison par évaluation. Pour cela, il utilise la structure de la pile de buts.

Ecrivons la taille de la pile des buts sous la forme $n = 2p + r$, avec $r=0$ ou $r=1$. La comparaison du but à la position p et du but de sommet de la pile à la position n, permet de pouvoir détecter une récursivité infinie lorsque les deux buts sont similaires. En effet, si nos supposons l'existence d'une récursivité infinie de longueur k (longueur du cycle), il existe alors une taille de la pile n telle que la distance entre les deux buts, n-p, soit proportionnelle à k et que les deux buts soient similaires. Cette propriété est due au fait que n-p croît de un chaque fois que n est impair, c'est à dire une fois sur deux.

L'utilisation directe de cet algorithme de détection est inefficace dans certains cas particuliers. En effet l'utilisation de la pile de buts comme structure de contrôle pose des problèmes car elle contient d'autres buts que tous les ancêtres du but courant. Ainsi, il est possible de détecter une récursivité sans qu'il y ait bouclage. Prenons l'exemple suivant :

```
R1 p :- q, r.
R2 q :- f.
R3 r :- s.
R4 s :- f.
R5 f.
```

Si nous considérons le but initial "p". À la sixième évaluation, nous avons l'état courant :

```
f , R5 <- n
s , R4
r , R3
f , R5 <- p
q , R2
p , R1
---pile--
```

Ainsi la comparaison s'effectue sur les deux buts "f" et "f". Ces deux buts sont similaires alors que les ancêtres du but "f" au sommet de la pile sont ["p", "r", "s"] et il n'y a pas de bouclage.

Pour d'éviter ce type de problème, nous pouvons stocker dans cette pile tous les ancêtres du but courant. Nous procédons de la façon suivante :

- Empiler le but courant lorsqu'une unification réussit, avec une référence sur la clause activée.
- Dépiler le but lorsque tous ses sous-buts réussissent (réussite du but).
- Empiler le but lors d'un retour-arrière sur la clause.
- Dépiler le but lors de l'échec final de la clause.

Cette méthode permet donc de gérer dans la pile de buts tous les ancêtres du but courant. Sur l'exemple précédent, à la troisième itération nous aurons l'état suivant :

```
f , R5  <- n
q , R2
p , R1  <- p
---pile---

Liste = [ r ].
```

le but f réussit car c'est un fait et on le dépile. Puis q réussit car f réussit et on le dépile. On obtient :

```
p , R1  <- n, p
---pile---

Liste = [ r ].
```

Puis après trois nouvelles itérations :

```
        f , R5  <- n
        s , R5
        r , R5  <- p
        p , R5
        ---pile---

    Liste = [].
```

le but f qui à réussit est dépilé et ainsi de suite pour les buts s, r et p. La pile et la liste sont alors vides, donc le but initial réussit. Dans ce modèle, le retour-arrière consiste à empiler de nouveau le dernier but ayant réussit afin de chercher une nouvelle clause s'unifiant.

Ainsi par cette gestion de la pile de buts nous pouvons adapter la méthode précédente pour détecter une récursivité infinie de la résolution Prolog. Nous allons préciser ce qui caractérise deux buts similaires.

8.1.4 La comparaison de deux buts

La détection d'une récursivité infinie est basé sur le fait qu'en cas de bouclage, nous pouvons trouver un état du système tel que :

Si n est la taille de la pile et p calculé par n = 2 * p + r, alors les buts à la position n et p dans la pile activent une même clause et sont similaires.

Le fait d'activer la même clause impose des contraintes sur les deux buts :

- même symbole fonctionel.
- même nombre d'arguments (arité).

Cette remarque permet de simplifier le test de similitude de deux buts, qui peut être effectué itérativement (récursivement) sur chaque argument. Ainsi, nous pourrons définir les différents cas de similitude lorsque le nombre d'arguments est égal à un et généraliser ensuite. En effet, si les arguments des deux buts de même symbole fonctionnel sont similaires un à un, alors les deux butsont similaires. Nous supposons donc les deux buts d'arité un et de même symbole fonctionnel f.

8.1.5 La similitude de deux buts

Soient :

- $f(X_1)$ le but à la position p dans la pile et ayant déclenché la clause C_k,
- $f(X_2)$ le but à la position n dans la pile (au sommet de la pile), ayant aussi déclenché la clause C_k.

Ces hypothèses correspondent à la dérivation "[f(X_1),C_k] → [f(X_2),C_k]".

Un cas particulier de bouclage est le cas où X_1 et X_2 sont identiques. Considérons chaque cas particulier en fonction de la nature des arguments X_1 et X_2 :

- X_1 est atomique, c'est-à-dire une constante atomique comme "a",
  - Si X_2 == X_1, il y a bouclage.
    c'est le cas : "f(a) → f(a)"
  - Si X_2 est une variable :
    c'est le cas : "f(a) → f(X_2)". On peut arbitrairement considérer qu'il n'y a pas récursivité. Dans le cas contraire, on aurait une dérivation du type f(a) → f(X_2) → f(X') et si X' == a ou si X' est un variable, on pourra par d'autres cas détecter la récursivité.
  - Si X_2 est une structure.
    c'est le cas : "f(a) → f(g(Args))". S'il existe une récursivité infinie de ce type, alors nécessairement nous aurons un autre cas (soit "g(Args1) → g(Args2)", soit "a → a") et on considère qu'ils ne sont pas similaires. .

- X_1 est une variable non instancié :
  - Si X_2 est une constante.
    c'est le cas : "f(X) → f(a)". Arbitrairement il n'y a pas de bouclage pour les mêmes raisons que le cas X_1 est un atome, X_2 est une variable
  - Si X_2 est une variable (X_1 == X_2) :
    c'est le cas : "f(X) → f(X)" et il y a bouclage.
  - Si X_2 est une structure :
    c'est le cas : "f(X) → f(g(Args))". S'il existe une récursivité infinie de ce type, nous aurons un autre cas (soit "g(Args1) → g(Args2)", soit "X → X") et on considère qu'ils ne sont pas similaires.

- X_1 est une structure de la forme "g(Args1)" :
  - Si X_2 est une constante a :
    c'est le cas : "f(g(Args1)) → f(a)". Il n'y a pas bouclage.
  - Si X_2 est une variable :
    c'est le cas : "f(g(Args1)) → f(X_2)". Il n'y a pas bouclage.
  - Si X_2 est une structure de la forme "f(h(Args2))"
    c'est le cas : "f(g(Args1)) → f(h(Args2))". Le symbole fonctionnel g est différent du symbole fonctionnel h, il n'y a pas de bouclage.

- Si X_2 est une structure de la forme "f(g(Args2))".
  On est sur le cas "f(g(Args1)) → f(g(Args2))".
  Si Args1 et Args2 sont similaires, il existe une récursivité infinie.

  Si "g(Args1)" est inclus dans "g(Args2)", il existe une récursivité infinie.
- Dans les autres cas, nous considérons arbitrairement qu'il n'y a pas de récursivité infinie.

La méthode de détection de récursion infinie que nous venons de spécifier, permettra de signaler au programmeur qu'il existe une boucle régulière dans la résolution et que de lui préciser quel est son type de bouclage.

8.2 *Vérification d'une résolution*

8.2.1 *Pourquoi une autre approche ?*

Les méthodes que nous avons décrites dans la deuxième partie ne permettent pas de résoudre de façon unique les deux problèmes principaux que posent la solution fausse et la solution insatisfaite. Elles ne peuvent que rechercher une erreur dans l'un ou l'autre cas et éventuellement passer de l'un à l'autre, notamment pour traiter la négation [Pereira-Calejo-88].

Cependant, sur le plan de l'approche logique les algorithmes de détection sont similaires. De plus, il peut exister dans certaines résolutions erronées des solutions incorrectes simultanées, parce qu'une solution fausse provoque une solution inexistante, ou réciproquement.

La question que nous nous posons est alors la suivante :

"Peut-on définir une approche commune du système de mise au point pour traiter les deux causes d'erreurs simultanément ?"

Un exemple typique est la résolution d'un but sans négation ne donnant pas de réponse alors qu'il en existe au moins une. L'application de l'algorithme de détection de la solution insatisfaite nous retourne le premier but de la résolution ayant une solution insatisfaite. Mais il est possible que l'échec de ce but soit dû à la solution fausse d'un sous-but et qu'il n'existe pas d'autre solution correcte pour ce sous-but. Dans ce cas, nous sommes en droit d'estimer que nous obtenons un échec de la méthode de mise au point. En effet, la méthode de détection d'erreurs ne permet pas de retourner l'erreur correcte pour la résolution du but initial. En revanche, dans ce cas, une méthode commune de détection d'erreurs permettrait de découvrir que la solution incorrecte est une solution fausse.

De même, dans le cas d'une résolution sans négation retournant une solution fausse, l'application de l'algorithme de recherche nous retourne une clause fausse s'il en existe une. Il est possible qu'il existe un sous-but de cette résolution admettant une solution insatisfaite dont l'effet serait justement de provoquer une solution fausse sous la forme d'une solution parasite engendrée par le retour-arrière suite à cet échec inattendu. Dans ce cas, la méthode de la solution fausse ne permet pas de trouver l'erreur produisant une mauvaise solution et la méthode de mise au point est mise en échec.

Enfin, il est souhaitable de répondre de façon sûre à d'autres questions que peut se poser un programmeur confronté aux résultats de la résolution. Ces questions sont :

"j'ai une solution inconnue, est-elle correcte ?"

"je n'ai pas de solution, est-ce correct ?"

La résolution peut également retourner une solution correcte et le programmeur veut obtenir la preuve de cette résolution. En effet, il est possible que plusieurs erreurs de programmation se compensent mutuellement.

Si nous voulons offrir un outil déclaratif de diagnostic d'erreurs aux utilisateurs des systèmes de programmation logique, c'est-à-dire un outil interactif utilisable sans connaissance préalable requises sur la sémantique opérationnelle de Prolog, pouvons nous utiliser des méthodes dont on connaît des cas qui les mettent en défaut ? La réponse est évidemment non, car nous devons être certains que les algorithmes traitent correctement tous les cas d'erreurs.

Ces petites imperfections de la détection classique d'erreurs en Prolog nous conduisent à réfléchir sur une méthode originale. De même, nous avons montré que si nous voulons traiter les programmes normaux, nous devons choisir une méthode de parcours en profondeur d'abord. De plus, certains cas pathologiques nous impose de faire une détection simultanée des deux causes d'erreurs : *la solution fausse* et *la solution insatisfaite.*

Pour résoudre ces divers problèmes, nous allons donc examiner la vérification de la résolution d'un but. En effet, sil nous sommes capable de vérifier que dans la résolution d'un but on ne découvre pas un but ayant soit une solution insatisfaite, soit une solution fausse, nous pourrons considérer le programme comme correct, bien que l'on ne puisse pas trouver de méthode pour prouver qu'un programme soit correct.

8.2.2 Principe

Considérons le problème de la vérification d'une résolution, que nous pouvons énoncer de la façon suivante :

> Soient un programme Prolog normal et un but dont on recherche une solution.
> La résolution de ce but retourne une solution quelconque ou échoue pour en produire une. Le programmeur désire savoir si cette résolution est correcte, c'est-à-dire, s'il existe dans l'arbre de résolution un sous-but ayant soit une solution fausse, soit une solution inexistante.

Ce problème pose de façon claire les objectifs d'un système unique de détection d'erreurs en Prolog :

"Permettre la validation d'une résolution quelconque".

Autrement dit, l'utilisateur d'un tel système disposera d'une méthodologie simple de preuve d'une résolution. Ainsi, en phase de mise au point, si aucune des vérifications de toutes les résolutions du jeu de tests caractéristiques ne fournit de solution incorrecte, l'utilisateur pourra considérer son programme comme suffisamment correct.

Analysons maintenant la résolution d'un but quelconque. Supposons l'existence dans cette résolution d'un premier sous-but ayant une solution incorrecte. Deux cas peuvent se produire :

solution fausse
: L'application d'une méthode ascendante permet de détecter cette solution fausse. En effet, par hypothèse, ce sous-but est le premier qui possède une solution incorrecte dans l'arbre de résolution ; chaque sous-nœud est correct et donc le nœud associé à ce sous-but est un nœud faux. Nous tenons une clause fausse.

solution insatisfaite
: L'application d'une méthode ascendante permet de détecter cette solution insatisfaite. Le premier sous-but incorrect échoue et ce but n'a pas de solution car il n'existe pas dans la sous-arborescence de cette résolution de sous-but ayant une résolution incorrecte.

Ainsi, ces deux techniques de détection utilisent le même type de parcours de l'arbre de recherche du but initial : la profondeur d'abord. La différence essentielle réside dans le test de détection d'erreurs. Pour réaliser la vérification d'une résolution, nous devons détecter simultanément la solution fausse et la solution insatisfaite. La détection repose sur la caractérisation de la présence ou de l'absence d'une solution. L'algorithme de la vérification est alors le suivant :

```
Simuler la résolution du but initial.
Soit un sous-but de cette résolution.
Si l'algorithme de résolution retourne une solution à ce sous-but
    Si le programmeur déclare cette solution "fausse"
        La clause courante est fausse ; arrêter
    Sinon continuer, le sous-but réussit
Sinon
    Sin le programmeur connait une solution
        Cette solution est insatisfaite ; arrêter
    Sinon continuer, le sous-but échoue.
```

Cet algorithme correspond à un parcours en profondeur d'abord de l'arbre de résolution pour le but initial. Il permet d'effectuer la *preuve* de la résolution d'un but. En effet, cette méthode permet de vérifier que l'interprétation de chaque sous-but de cette résolution est correcte ou incorrecte selon le programmeur (c'est-à-dire selon le modèle du programme). Dans le cas où il n'existe pas d'erreur détectable par cet algorithme, nous considèrons la résolution comme correcte. Nous avons alors une preuve déclarative de la résolution.

Nous devons justifier la *complétude* de cette méthode en termes d'objectif s qu'elle doit résoudre : détecter une solution insatisfaite ou une solution fausse sur des programmes méta-logiques.

8.2.3 Les prédicats méta-logiques

Les caractéristiques des prédicats méta-logiques permettent d'effectuer une mise au point sur des programmes les contenant. Nous l'avons vu sur l'étude des extensions de la détection d'erreurs.

En effet, un programme contenant ces prédicats conserve une sémantique déclarative si on lui ajoute les théorèmes ou axiomes nécessaires à ces prédicats méta-logiques.

Dans le cas particulier de prédicats extra-logiques, il n'existe pas de théorie du premier ordre permettant de pouvoir les spécifier au niveau des programmes. C'est la raison principale de l'impossibilité de mise au point de ces prédicats.

8.2.4 La coupure

Examinons maintenant le problème de la coupure dans le cadre de la vérification de la résolution d'un but. Énonçons la proposition suivante :

Proposition :

Soient un programme P, et un but B. Considérons l'application de l'algorithme de détection d'erreurs sur le but B et le programme P. Si l'algorithme retourne une clause C produisant une solution déclarée fausse par le programmeur, alors cette clause est fausse.

Démonstration :

Montrons que chaque sous-but du but associé à la clause fournie par l'algorithme possède une solution correcte et que cette clause est une clause fausse associé à une solution fausse. Considérons la sous-arborescence du nœud de l'arbre de résolution associé à cette clause fausse.

Si cette sous-arborescence ne contient pas de coupe-choix, nous sommes dans les conditions de Prolog pur et la clause est bien une clause fausse car la solution est incorrecte et chaque solution d'un sous-but prémisse de la clause est une solution correcte.

Sinon, considérons C' une clause quelconque de la sous-arborescence contenant le coupe-choix. Nous pouvons écrire cette clause sous la forme

suivante "C'={A:-(A_1), !, (A_2)}". Pour chaque clause C' différente de la clause C, par hypothèse sur la clause C, la solution du but A est une solution correcte, sinon C' serait la clause C, de même que pour les solutions des deux sous-buts A_1 et A_2. Dans le cas où la clause C contient un coupe-choix, c'est bien une clause incorrecte du programme, car tous les sous-buts sont corrects mais la solution calculée est une solution fausse.

Ainsi, les conclusions de l'algorithme de recherche pour le cas de la solution fausse sont correctes.

Dans le cas où nous obtenons une solution insatisfaite, nous invitons le programmeur à analyser de façon précise les causes de cet échec. Dans le cas particulier où l'utilisation du coupe-choix ne modifie pas la sémantique déclarative du programme alors les conclusions de l'algorithme de mise au point sont correctes. En effet, il suffit de considérer le programme logique équivalent obtenu par suppression de la coupure et d'y appliquer ce même algorithme de mise au point. Les conclusions sont alors identiques, car il retourne le même résultat.

En revanche, lorsque l'utilisation de la coupure n'est pas déclarative, sa suppression risque de donner des résultats différents. Notre problème est de savoir quelle est la portée de la coupure sur les conclusions que nous obtenons de l'algorithme de mise au point.

Cependant, nous pouvons proposer une analyse de l'échec afin de trouver une clause "C={A:-(A_1), !, (A_2)}" tel que le but avec une solution insatisfaite s'unifie avec l'atome A et le but "(A_1)" admet une solution (qui est correcte dans l'interprétation attendue). Dans ces conditions, s'il n'existe pas de clause vérifiant cette contrainte nous pouvons affirmer que la réponse du système est correct, la solution est bien une solution insatisfaite.

En revanche, lorsqu'il existe une telle clause, comme il y a un échec du but fournit, il y a un échec du sous-but "(A_2)" et la coupure peut être une coupure rouge. Mais nous ne pouvons pas l'affirmer car il se peut que la solution obtenue pour le sous-but "(A_1)" ne soit pas celle attendue comme première solution. Or cet aspect ne peut être connue que du programmeur car c'est une notion opérationnelle.

Ainsi, il est parfois utile de proposer au programmeur une autre approche de mise au point, fondée sur la trace d'un but ayant une solution insatisfaite. Seule une trace intelligente permet au programmeur de connaître la sémantique opérationnelle d'un de ses programmes, afin de la vérifier. Il est donc indispensable de munir le système de mise au point d'une de trace intelligente.

8.2.5 La négation par l'absurde

La réalisation d'une négation correcte suppose de démontrer que la résolution d'un but est finie, c'est-à-dire que son arbre de résolution est de profondeur finie. Il faut donc détecter une résolution non finie. Comme nous l'avons souligné dans la partie correspondant à la détection de boucle, notre méthode permet de réaliser une négation complète.

Comme la négation par l'absurde est compatible avec l'hypothèse de la base de donnée complète et que nous pouvons démontrer qu'un but possède une solution ou non, nous traitons la négation complète.

Considérons maintenant un programme Prolog normal et simulons la résolution d'un but par la méthode de vérification de l'arbre de recherche.

Supposons que dans la résolution de ce but initial nous trouvions la négation d'un sous-but. Si la résolution de ce sous-but est une résolution incorrecte, elle contient un nœud incorrect et l'algorithme ne retournerait pas une réussite ou un échec sur la négation du sous-but et serait interrompu.

Dans le cas contraire, où l'arbre de recherche de ce sous-but nié ne contient pas de nœud incorrect, la négation du but est correcte. Ainsi, si le sous-but nié échoue il y a une réussite correcte de la négation. Dans le cas contraire l'échec de la négation est considéré comme correct.

Il est à remarquer que lorsque la négation échoue, il existe une solution du but. Cette méthode nous garantit que la solution du but trouvée est correcte dans l'esprit du programmeur. En revanche, nous ne pouvons pas prouver que cette solution correcte est bien la première qui doit être trouvée. Nous avons là une limitation (partielle) de cette méthode pour la négation et par la même occasion de la preuve de la résolution.

8.2.6 Conclusion

Cet algorithme de recherche permet simultanément de rechercher une cause d'erreur dans la résolution d'un but. De plus, il permet de vérifier l'arbre de résolution d'un but, ce qui correspond à une preuve de la résolution. Il permet de résoudre simultanément le problème de la recherche d'une clause fausse sur un but ayant une mauvaise solution et celui de la recherche d'une erreur sur un but ayant une solution insatisfaite. Mais aussi, cet algorithme permet de valider une solution inattendue mais correcte, ou bien de trouver une erreur sur une solution correcte avec un arbre de résolution incorrect.

9. Réalisations

Ce chapitre présente les principes d'une réalisation particulière des algorithmes que nous venons de définir. Nous présenterons ensuite une réalisation opérationnelle d'une interface utilisateur de mise au point en Prolog mettant en oeuvre ces algorithmes de détections d'erreurs ainsi qu'une trace interactive.

9.1 Un méta-interprète

L'utilisation d'un méta-interprète est relativement courante en programmation logique. On peut définir un méta-interprète comme un programme simulant l'exécution de tout programme écrit dans un langage de programmation quelconque. Cependant, il existe d'autres applications d'un méta-interprète, par exemple, l'analyse syntaxique.

Un cas particulier de méta-interprète est la simulation du langage dans lequel il est écrit, ce qui correspond à l'auto-définition de ce langage. Ainsi, on peut réaliser un méta-interprète écrit dans le langage Prolog et simulant l'exécution de l'interprète Prolog. Les applications d'un tel programme sont multiples. Le plus célèbre exemple est le programme *demo* de Bowen et Kowalski [Bowen-Kowalski-82]. Le prédicat *demo* possède deux arguments, le programme et le but à résoudre. Il permet de pouvoir mélanger des objets et du méta-langage. Bundy et Welham utilisent une telle approche dans le domaine de la démonstration automatique [Bundy-Welham-81]. De même, Venken réalise une évaluation partielle de programmes basée sur un méta-interprète pour optimiser l'interrogation de bases de données [Venken-85]. Barklund utilise une telle notion afin d'optimiser l'évaluation d'un programme [Barklund-87]. Enfin, Safra & Shapiro [Safra-Shapiro-86] de même que Bowen & Weinsberg [Bowen-Weinsberg-85a 85b] étendent les spécifications de Prolog en lui ajoutant des fonctionnalités sous une forme méta-interprétée. On peut de même réaliser des outils de mise au point sur la base de méta-interprètes, pour la trace [Byrd-80, Boizumault-84, Eisenstadt-85] ou bien pour la détection d'erreurs [Shapiro-82,83, Eisenstadt-85, Eisenstadt-Brayshow-87, Pereira-83,88]...

9.1.1 Un Méta-Interprète de Prolog pur

A partir de la sémantique opérationnelle que nous avons présentée, nous pouvons définir la résolution que fait l'interprète Prolog de la façon suivante :

- true est toujours vrai,
- (A, B) est vraie si A est vrai et si B est vrai,
- A est vrai s'il existe une clause (A :- B) et B est vrai.

Ainsi, un méta-interprète de Prolog Pur s'écrit très facilement :

```
méta_call( true ) :-
    !.
```

```
méta_call( ( But , Reste ) ) :-
    !,
    méta_call( But ),
    méta_call( Reste ).

méta_call( But ) :-
    clause( But , Queue ),
    méta_call( Queue ).
```

Ce méta-interprète permet de simuler une résolution Prolog. Nous devons le généraliser pour tenir compte des diverses extensions du langage Prolog.

9.1.2 Généralisation

Le langage Prolog est basé sur un sous-ensemble de la logique des clauses de Horn et sur la stratégie de l'évaluation de l'interprète (interprétation de gauche à droite et recherche en profondeur) et un ensemble de primitives de contrôle extra-logique. Parmi les plus importantes de cet ensemble figurent la coupure, pour modifier le contrôle de l'exécution de l'interprète, ainsi que les primitives *"assert"*, *"retract"* et *"clause"* pour manipuler les clauses. Il est à noter que la négation est une primitive obtenue à l'aide de la coupure. On pourra se rapporter à l'article *"Cut and Paste"* [Moss-86] concernant la sémantique extra-logique de ces primitives.

La généralisation des méta-interprètes a fait l'objet de nombreuses études. La littérature nous montre qu'elle est basée essentiellement sur l'utilisation d'une coupure étendue, appelée "*ancestor cut*". Un exemple de son principe d'utilisation est donné dans le livre *"Art of Prolog"* [Shapiro-Sterling-86]. Ce prédicat, que nous nommerons cut/1 est prédéfini dans un certain nombre de système du commerce, tels que SB-Prolog, Xilog de ACT [Sbpl-86, Xilog]... Il réalise une coupure sur un nœud de l'arbre de résolution. Nous devons remarquer que l'utilisation de cette extension de la coupure se rapproche de la notion du Go-To de certains langages de programmation tel que *Pascal, C, Fortran*... De plus, cette extension est généralement utilisée en association avec un prédicat de marquage d'un nœud de l'arbre [Venken-85] *"mark/1"*. Ce prédicat marque le nœud courant de l'arbre de résolution afin de déterminer la portée de la coupure étendue. Par exemple, le programme suivant :

```
p(X) :- a(X),b(X).
b(1) :- !.
b(X) :- write(X).
```

peut être transformé par une technique d'évaluation partielle en son interprétation opérationnelle :

```
p(X) :-
    a(X),
    mark(1),
    (X = 1,cut(1); write(X)).
```

Où *"mark(1)"* détermine la portée de la coupure *"cut(1)"*. Ce type de programmation en Prolog nous parait contraire à l'esprit de la Programmation Logique et surtout peut conduire à des écritures de programmes non logiques [O'Keefe-85]. Nous pouvons écrire d'une façon simple un méta-interprète pour Prolog l'utilisant cette extension de la coupure mais dont la sémantique n'est pas déclarative :

```
méta_call(true, _) :- !.
méta_call(!, CutTo) :-
    !,
    cut(CutTo).
méta_call( (But, Reste), CutTo) :-
    !,
    méta_call(But, CutTo),
    méta_call(Reste, CutTo).
méta_call(But, _) :-
    mark(CutHere),
    clause( But, Queue),
    méta_call( Queue, CutHere).
```

Lorsqu'il n'y a pas de coupe-choix dans la queue d'une clause, la sémantique pure est conservée. Dans le cas contraire, la marque faite avant la recherche de cette queue de clause détermine la portée de cette coupure. Il est clair qu'un tel programme est de nature à détériorer l'image du langage Prolog

Dans son article, O'Keefe nous présente un méta-interprète du langage Prolog qu'il nomme *méta-circulaire* et qui traite le coupe-choix, la conjonction de buts et la disjonction de buts. Nous présentons ici la partie contenant uniquement le coupe-choix, car les autres cas s'en déduisent de façon évidente :

```
do_goal(true):-!.
do_goal(Goal) :-
    clause(Goal, Body),
    do_body(Body, AfterCut, HadCut),
    ( HadCut = yes, !, do_body(AfterCut); HadCut = no).

do_body(Body) :-
    do_body(Body, AfterCut, HadCut),
    ( HadCut = yes -> do_body(AfterCut) ; HadCut = no ).

do_body((!, AfterCut), AfterCut, yes):-!.
do_body((Goal, Body), AfterCut, HadCut):-
    do_goal(Goal),
    do_body(Body, AfterCut, HadCut).
do_body((!), true, yes).
do_body(Goal, true, no):-
    do_goal(Goal).
```

Cette analyse très correcte de la coupure permet de réaliser un méta-interprétation efficace. Le prédicat *"do_body/3"* permet de simuler l'effet du coupe-choix et d'en calculer la portée : les choix sur les buts précédant la coupure et les choix sur les prédicats suivants dans le paquet dont provient la clause.

Il nous est possible d'utiliser cette technique de méta-interprétation pour réaliser un système de mise au point. Cependant, les caractéristiques particulières de certains interprètes Prolog disponibles sur le marché, nous ont conduits à rechercher une technique différente de la méta-interprétation. Les problèmes sont essentiellement sur la sémantique de la coupure et de la recherche d'une clause. Nous avons choisi le système Quintus-Prolog qui possède une restriction sur le prédicat *"clause/2"* qui ne permet de consulter que les clauses dynamiques. La réalisation que nous avons effectuée vise à ne pas utiliser ce prédicat, coûteux en temps d'exécution, mais plutôt d'utiliser la coupure propre de l'interprète Prolog par une évaluation partielle du programme.

Dans cette partie nous étudions la sémantique opérationnelle de la coupure afin de réaliser une évaluation partiellequi permettra de simuler la résolution. Le programme résultant de cette évaluation est l'interprétation sémantique du programme à méta-interpréter. Ainsi, la transformation des programmes Prolog est fondée sur une technique d'évaluation partielle. Cette transformation nous permet de contrôler le déroulement de la résolution et de fournir des informations sur la résolution. Enfin, elle permet de répondre à quelques questions que l'on peut se poser en cours de résolution :

- Quelle est la clause en cours d'évaluation ?
- Quels sont les noms des variables ?
- Quelle est la listek des ancêtres du but courant ?
- Quel est le but qui va être évalué dans la clause ?

9.1.3 Le coupe-choix

Nous allons modifier le méta-interprète défini en 9.1.1 pour qu'il tienne compte de la coupure, le *fameux "cut"* noté ! dans les interprètes en syntaxe C-Prolog, noté / en syntaxe Prolog II. Nous nous plaçons dans le cas de la syntaxe et de la sémantique de C-Prolog.

Pour illustrer de façon précise la sémantique opérationnelle de la coupure, considérons le programme :

```
test :- a , ! .
test.
a :- b.
a.
b :- c.
b.
c.
c.
```

et le but initial "*test*". Après la réussite du coupe-choix, il y a réussite du but initial *test*. Il existe de nouvelles solutions possibles, par des points de reprise sur le but c, sur le but b, sur le but a, et sur le but test. Lors de la recherche d'une nouvelle solution sur le but *test*, on effectue un retour-arrière sur la queue de la première clause du prédicat *test*. Ce retour provoque un retour-arrière les coupe-choix de la clause. Son effet est alors de supprimer toutes les solutions en attente et le but *test* n'a pas d'autre solution.

Une réalisation naïve de l'interprétation du coupe-choix serait d'ajouter la déclaration suivante : "*méta_call(!):-!*". Cependant, nous n'obtiendrions pas l'effet désiré lors d'un retour-arrière sur une coupure, car il ne se passerait rien. En revanche, l'interprétation correcte consiste à transformer toute clause "*but :- but1 , ! , but2*" par son interprétation sémantique : pour résoudre le but *but* il faut résoudre le but *but1*, puis interpréter un coupe-choix et enfin résoudre le but *but2*. Nous pouvons alors traduire cette phrase par la déclaration suivante :

```
méta_call( but ) :-
    méta_call( but1 ),
    !,
    méta_call( but2 ).
```

Lors de la simulation du but, l'action du coupe-choix est bien interprétée, puisque c'est celle de l'interprète Prolog. Lors de la méta-interprétation d'un but, l'appel de "*méta_call(but)*" provoque l'appel de cette clause, éventuellement avec des points de choix, la réussite de la méta-interprétation du but *but1* est suivie du coupe-choix. Lors du retour-arrière sur ce coupe-choix, son effet est de provoquer un échec sur l'appel du but "*méta_call(but)*", ce qui correspond à l'effet désiré du coupe-choix sur le but *but*.

Donc, si nous modifions toutes les clauses d'un programme P par une telle méthode, nous réalisons une pseudo-interprétation du programme fondée sur une méthode d'évaluation partielle de chacune de ses clauses. Cependant, cette façon d'aborder les choses ne correspond pas une simulation de l'interprète Prolog (nous effectuons un réécriture du programme sous une forme équivalente du point de vue sémantique, qui correspond aussi au déroulement du programme "*do_goal/1*"). De plus, pour des objectifs de contrôle de la résolution, nous décidons de modifier d'une façon particulière les clauses :

```
but( Args ) :- Queue.
```

sous la forme sémantique équivalente :

```
méta( but , [ Args ] ) :- Partial_éval.
```

où *Partial_éval* est l'interprétation sémantique de la queue de clause *Queue* obtenue par l'évaluation partielle et où *Args* est la liste des arguments de la tête de la clause. Ce choix de structure absolument arbitraire est simplement fait pour regrouper les buts dans un même paquet. Pour illustrer ce principe, considérons un programme et son évaluation partielle :

```
    Programme          Transformation
a(X) :-                méta( a, [X] ) :-
    b(X),                  méta_call( b(X) ),
    !,                     !,
    c(X).                  méta_call( c(X) ).
a(X) :-                méta( a, [X] ) :-
    d(X).                  méta_call( d(X) ).
a(atome).              méta( a, [atome] ).
```

Une telle technique de réécriture de programmes rend évidente l'écriture du méta-interprète :

```
méta_call( But ) :-
    But =.. [Func|Args],
    clause( méta( Func , Args ) , Queue ),
    call( Queue ).
```

Cependant, comme là encore, la sémantique de la primitive "*call*" lorsqu'elle s'applique sur un coupe-choix varie d'un interprète Prolog à un autre (on pourra se rapporter aux études faites par Moss ou Lepape & Sellami [Moss-86, Lepape-Sellami-87]), nous devons légèrement modifier cette déclaration. Il suffit pour cela de remarquer que la conjonction des deux buts *"clause(méta(Func, Args), Queue)"* et *"call(Queue)"* est absolument identique à l'appel direct de la clause *"méta(Func, Args)"*. Dans ces conditions, nous obtenons la déclaration :

```
méta_call( But ) :-
    But =.. [Func|Args],
    méta( Func , Args ).
```

Ainsi, nous observons que le méta-interprète est fondé sur l'évaluation partielle de la queue de la clause : *Queue --evpar--> Partial_éval.* Cette transformation s'effectue de la façon suivante :

```
méta_évaluer( ( B1 , B2 ) , ( M1 , M2 ) ) :-
    !,
    méta_évaluer( B1 , M1 ),
    méta_évaluer( B2 , M2 ).
méta_évaluer( ( B1 ; B2 ) , ( M1 ; M2 ) ) :-
    !,
    méta_évaluer( B1 , M1 ),
    méta_évaluer( B2 , M2 ).
méta_évaluer( true , true ) :- !.
méta_évaluer( ! , ! ) :- !.
méta_évaluer( But , méta_call( But ) ).
```

Comparons les deux méta-interprètes présentés :

- dans le premier, pour résoudre un but composé il faut résoudre le premier but puis le reste des buts,

- dans le deuxième, il faut résoudre le premier but puis le deuxième but puis le troisième but...

D'où l'idée de la transformation initiale des queues de chaque clause du programme qui est obtenue par le prédicat *"méta_évaluer/2"*.

9.1.4 La négation

La coupure permet d'implanter une certaine forme de négation en Prolog, appelée négation par l'échec. La réalisation standard de cette négation est :

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

Sa sémantique opérationnelle est :

"la négation d'un but réussit si et seulement si la résolution de ce but échoue"

Cette réalisation utilise l'appel d'une méta-variable et la conjonction cut/fail. Lors de la résolution de la négation d'un but *not(G)* leffet de la première clause est d'exécuter le but *G*. Lorsque la résolution du but *G* réussit, le coupe-choix est appelé puis suivi par l'appel de l'échec et la résolution de la négation du but, *not(G)*, échoue. Dans ce cas la résolution du but *G* ne retourne que la première solution. Dans le cas contraire, i.e. lorsque la résolution du but *G* ne retourne aucune solution, la négation *not(G)* réussit par la seconde clause. Ainsi, la sémantique opérationnelle de cette négation est : *not(G)* réussit si le but *G* n'admet pas de solution, *not(G)* échoue si le but *G* admet au moins une solution dont seule la première est calculée.

Dans ces conditions, la méta-interprétation de la négation s'effectue d'une façon simple. Si nous considérons le but *méta* comme étant l'interprétation sémantique du but *But*, alors *not(But)* réussit si et seulement si *not(méta)* réussit. Pour réaliser une évaluation partielle de la négation, nous pouvons effectuer la déclaration :

```
méta_évaluer( not But , not Méta ) :-
    !,
    méta_évaluer( But , Méta ).
```

Il est à remarquer que d'autres approches sont possibles pour réaliser la négation en Prolog. Ainsi, Naish [Naish-85] utilise les ensembles de solutions comme support de divers prédicats prédéfinis, tels que : Var/1, Not/1... Sa proposition de sémantique de la négation est la suivante :

```
not(X) :- solutions(_,X,[]).
```

Ce qui revient à dire que si l'ensemble des solutions du but X est vide alors la négation du but X réussit. Cependant, comme le font remarquer Shapiro et Sterling et plus particulièrement Clark, si la négation par l'échec réussit alors l'arbre de recherche de la résolution du but X est finie [Shapiro-Sterling-86, Clark-78]. En revanche, lorsque l'arbre de résolution du but X est infini, sa négation peut être finie car seule la première solution est importante. Dans la réalisation de Naish, si l'arbre de recherche est infini, la négation n'est pas finie.

Dans toute la suite de notre étude, nous considérerons que la négation est une négation par l'échec.

9.1.5 Le but initial

Dans les parties précédentes, nous nous sommes surtout intéressés à l'interprétation du but courant. Nous devons donc maintenant tenir compte de l'appel du but initial, que nous appellerons *Top_goal*. Ce but initial est de la forme : "$B_1, B_2 \ldots B_n$", où $n > 0$.

Afin de simplifier le problème, nous pouvons considérer qu'il existe une clause correspondant à la déclaration du but initial et qui peut s'écrire sous la forme uniquew : *"top_goal(Args) :- Top_goal"*. Dans ces conditions, nous pouvons appliquer sur cette clause l'évaluation partielle pour obtenir une clause dont la sémantique est équivalente au but initial. Nous transformons cette clause par la déclaration : *"méta(top_goal, Args) :- Méta_goal"*.

En fait, il suffit simplement d'évaluer l'interprétation du méta-but *Méta_goal* pour effectuer l'interprétation du but initial. Nous pouvons ainsi définir le prédicat *"solve(Top_goal)"* comme étant l'appel initial de la résolution du but *Top_goal*, et nous obtenons ainsi la déclaration :

```
solve( Top_goal ) :-
    méta_évaluer( Top_goal , Méta_goal ),
    call( Méta_goal ).
```

Ce prédicat permet de *lancer* l'exécution de la méta-interprétation du but initial.

9.1.6 Méta Prolog : le méta-interprète

Les extensions possibles de ce méta-interprète consisteraient à tenir compte des buts prédéfinis et des méta-variables. Pour les buts prédéfinis une façon simple d'en tenir compte est de rajouter une déclaration de la forme :

```
méta_évaluer(But, But) :-
    predef(But),
    !.
```

Où le prédicat *"predef/1"* réussit si et seulement si son argument est un prédicat prédéfini. Cependant, comme l'appel d'un but prédéfini peut causer l'arrêt de la résolution, il est alors nécessaire de réaliser une méta-interprétation spécifique pour chacon d'eux.

En revanche, pour le cas d'un but contenant une méta-variable, lors de l'évaluation partielle du programme, la méta-variable n'est pas instanciée. Ce n'est que lorsque l'on appellera ce méta-but que l'on pourra réaliser l'évaluation partielle. Aussi, devons-nous traiter ce cas dans le programme *"méta_évaluer/2"* de la façon suivante :

```
méta_évaluer( call(X),
    (méta_évaluer(call(X),Méta),Méta)) :-
    var( X ),
    !.
```

LCette réalisation pose un autre problème : lors de l'exécution de la méta-interprétation du méta-but X, si la méta-variable X n'est pas instanciée,il y aura un appel récursif de *méta_évaluer* sur une variable non instanciée qui provoquera un bouclage du méta-interprète.

Nous pouvons donc donner une déclaration (partielle) du méta-interprète Prolog écrit en Prolog. Cependant, pour des raisons de place nous ne décrirons pas l'évaluation partielle d'un programme Prolog obtenu par le prédicat *"méta_évaluer/2"*.

```
solve( Top_goal ) :-
    méta_évaluer( Top_goal , Méta_goal ),
    call( Méta_goal ).

méta_call( But ) :-
    But =.. [Func|Args],
    méta( Func , Args ).

méta_évaluer( call(X),(méta_évaluer(call(X),Méta),Mé-
ta)) :-
    var( X ),
    !.

méta_évaluer( ( B1 , B2 ) , ( M1 , M2 ) ) :-
    !,
    méta_évaluer( B1 , M1 ),
    méta_évaluer( B2 , M2 ).

méta_évaluer( ( B1 ; B2 ) , ( M1 ; M2 ) ) :-
    !,
    méta_évaluer( B1 , M1 ),
    méta_évaluer( B2 , M2 ).

méta_évaluer( true , true ) :- !.

méta_évaluer( ! , ! ) :- !.

méta_évaluer(But, But) :-
    predef(But),
    !.
```

```
méta_évaluer( not But , not Méta_goal ) :-
    !,
    méta_évaluer( But , Méta_goal ).

méta_évaluer( But , méta_call( But ) ).
```

9.1.7 Application

9.1.7.1 La trace de BYRD

Avant de réaliser la trace de BYRD, rappelons brièvement son principe. La trace produit quatre messages :

Appel	appel de la résolution d'un but,
Sortie	réussite d'un but,
Retour	rappel d'un but par retour-arrière,
Echec	échec d'un but.

Oon peut associer ces quatre messagesdeux à deux :

Appel / Echec
Sortie / Retour

En effet, pour le couple Appel/Echec, le message d'appel est émis juste avant l'appel effectif du but et le message d'échec après l'échec du but. On peut traduire cette propriété sous la forme d'un but :

"(appel ; échec),call(But)"

L'appel de ce but, déclanche d'abord la résolution du but *"appel"* qui envoie le message "Appel", puis la résolution du but But. Enfin lors du retour-arrière sur le point de choix ";", la résolution du but *"échec"* envoie le message "Echec" puis échoue afin de simuler l'échec du but. Pour le second couple l'idée est identique et nous pouvons procéder de la façon suivante :

- déclarer un prédicat d'appel ayant deux points de choix, le premier faisant l'appel, le second l'échec.
- déclarer un prédicat de sortie ayant lui aussi deux points de choix, le premier signalant une sortie du but, le second signalant un retour-arrière sur le but.

```
appel( But ) :-
    message( But , 'Appel : ').
appel( But ) :-
    message( But , 'Echec : '),
    fail.
```

```
sortie( But ) :-
    message( But , 'Sortie : ').
sortie( But ) :-
    message( But , 'Retour : '),
    fail.
message( But , Mess ) :-
    nl,
    write( Mess ),
    write( But ).
```

Enfin, pour tracer un but il suffit de les insérer avant et après l'appel effectif du but à tracer dans le méta-interprète :

```
méta_call( But ) :-
    appel( But ),
    But =.. [Func|Args],
    méta( Func , Args ),
    sortie( But ).
```

Considérons par exemple, le programme "test" et sa méta-transformation :

```
test :-          méta(test,[]) :-
    a,               méta_call(a),
    !,               !,
    b.               méta_call(b).
a :-             méta(a,[]) :-
    c.               méta_call(c).
a.               méta(a,[]).
b :-             méta(b,[]) :-
    c.               méta_call(c).
b :-             méta(b,[]) :-
    c.               méta_call(c).
c.               méta(c,[]).
```

La résolution du but initial *test*, avec le méta-interprète contenant les prédicats de traces, nous donne la trace :

```
?- solve(test).
Appel :     test
Appel :     a
Appel :     c
Sortie :    c
Sortie :    a
Appel :     b
Appel :     c
Sortie :    c
Sortie :    b
Sortie :    test
```

Cet exemple illustre la simplicité d'incorporation d'une trace dans un méta-interprète de Prolog.

9.1.7.2 La trace de Boizumault

Le principe de la trace de Boizumault est sensiblement le même au niveau du couple *Appel/Echec*.ous pouvons garder le prédicat d'appel d'une clause *appel(but)*. En revanche, pour les deux autres messages de trace, il faut premièrement rechercher toutes les clauses du paquetcorrespondant au but courant, puis tester l'unification de chacune d'elles avec le but courant. Lorsqu'il y a unification, il faut afficher le message *prouve* ; dans le cas contraire passer à la clause suivante. Il y a échec lorsqu'il n'y a plus de clauses pouvant s'unifier.

Nous devons donc légèrement modifier l'évaluation partielle du programme pour réaliser cette trace. Si nous testons l'unification dans la queue de chaque clause nous simulerons l'appel de chacune d'elles. Ainsi, nous pouvons transformer chaque clause de la forme *"f(T1,T2,...TN) :- Body"* par sa forme canonique équivalente : *"f(X1,X2,...Xn) :- [X1,X2,..Xn] = [T1,T2,...Tn], Body"*. Puis, par une évaluation partielle, nous lui ajoutons les prédicats de trace d'*unification* et de *preuve* :

```
méta(Func, Arité, N, Args) :-
    message( N , ' Unifie '),
    write( f(T1, T2 ... Tn) ),
    Args = [T1, T2 ... Tn],
    message( Queue , ' Prouve : '),
    write( Body ),
    Méta.
```

Enfin, nous incluons dans le méta-interprète la déclaration concernant la trace d'appel/échec :

```
méta_call( But ) :-
    appel( But ),
    functor(But, Func, Arité),
    But =.. [Func|Args],
    méta(Func, Arité, N, Args).

appel( But ) :-
    message( But,' Appel : ').
appel( But ) :-
    message( But,' Echec : '),
    fail.
```

Sur le programme précédent, nous avons la trace suivante :

```
?- méta_call(test).
    Appel : test
    Unifie 1 test
    Prouve : a , ! , b
    Appel : a
```

```
        Unifie 1 a
        Prouve : c
        Appel : c
        Unifie 1 c
        Prouve : true
        Appel : b
        Unifie 1 b
        Prouve : c
        Appel : c
        Unifie 1 c
        Prouve : true
yes
        Echec : c
        Unifie 2 b
        Prouve : c
        Appel : c
        Unifie 1 c
        Prouve : true
yes
        Echec : c
        Echec : b
        Echec : test
```

Nous observons que le coupe-choix a bien fait son effet sur la première clause du prédicat *a/0*. Il n'y a pas eu d'essai d'unification de la deuxième clause, ni de message d'échec de "a", car le but réussit mais on ne recherche pas d'autre solution. Il reste à positionner les messages pour les indenter.

9.1.7.3 La trace de Eisenstadt

Pour la réalisation de la trace d'Eisenstadt, il nous modifier la méthode d'évaluation partielle. Nous pouvons procéder à la transformation :

```
méta(Func, Arité, Nb, [Args]) :-
    méta_call_appel(Func(Args), Nb),
    Partial_éval,
    méta_call_réussit(Func(Args), Nb).
```

où *Func* est le symbole du prédicat considéré, *Arité* est le nombre d'arguments du prédicat, Nb est le numéro d'apparition de la clause dans le paquet *Func/Arité*, *Args* est la liste des arguments du prédicat et *Partial_éval* est l' évaluation partielle obtenue à partir de la queue de la clause.

Le prédicat *méta_call_appel/2* signale l'activation d'une clause après une réussite d'unification du but courant. Il permet d'imprimer les messages correspondant aux deux *labels* ">" et "<". Pour réaliser l'indentation successive lors du déroulement de la résolution, nous utilisons un fait particulier, *"position(X, Y)"*, où X est la position du but parent, Y la position du but courant. La déclaration du prédicat *méta_call_appel/2*, peut être réalisée de la façon suivante :

```
méta_call_appel(But ,N_CL) :-
    message(But,'>  ', N_CL),
    retract(position(_, Y)),
    NY is Y + 3,
    asserta(position(Y, NY)).
méta_call_appel(But, N_CL) :-
    retract(position(X, _)),
    NX is X - 3,
    asserta(position(NX, X)),
    message(But,'<  ', N_CL),
    fail.
```

De même, le prédicat *méta_call_réussit/2* signale une réussite du but courant et le retour-arrière sur la queue de clause. Il permet d'imprimer les messages correspondant aux deux labels "+ " et "^ ". La réalisation, utilise le fait *position/2* pour gérer correctement l'indentation :

```
méta_call_réussit(But, N_CL) :-
    retract(position(X,_)),
    Y is X - 3,
    asserta(position(Y, X)),
    message(But,'+  ', N_CL).
méta_call_réussit(But, N_CL) :-
    message(But,'^  ', N_CL),
    retract(position(_, X)),
    Y is X + 3,
    asserta(position(X, Y)),
    fail.
```

Nous devons généralement traiter le cas particulier des faits lors de l'évaluation partielle. Dans ce cas, nous procédons de la façon suivante :

```
méta(Func, Arité, Nb, [Args]) :-
    méta_call_fait(Func(Args), Nb).
```

où le prédicat *méta_call_fait/2* permet de signaler la réussite d'un fait par le message "+* " ainsi que le retour-arrière sur ce fait "^* ". Nous avons volontairement ajouté ce message pour différencier le retour-arrière de l'appel d'un fait.

```
méta_call_fait(Tête, N_CL) :-
    message(Tête, '+* ', N_CL).
méta_call_fait(Tête, N_CL) :-
    message(Tête, '^* ', N_CL),
    fail.
```

Afin de signaler l'activation du coupe-choix, nous déclarons le prédicat *méta_cut/0*. Il permet à son appel d'envoyer le message "-- !", ainsi que lors du retour-arrière sur la position de la coupure d'envoyer le message "^^ !". De plus, nous ajoutons un fait *méta_cut_fail*, nous permettant d'indiquer un retour-arrière sur le coupe-choix.

```
méta_cut :-
    position(X,_),
    écrire(' ! ',----------------,X).
méta_cut :-
    retract(position(X,_)),
    NX is X - 3,
    asserta(position(NX, X)),
    écrire(' ! ',^^^^^^^^^^^^^^^^,X),
    asserta(méta_cut_fail),
% Retour-arrière sur un coupe choix
    fail.
```

Enfin, le prédicat *"méta_call(But, Func, Arité, Args)"* indique l'appel initial d'un but *But* par le message "? " et effectue la résolution du but *"méta(Func, Arité, _, Args)"*. Lors d'un retour-arrière sur ce prédicat, lorsqu'il n'y a plus de solution ou pas de solution, ce prédicat envoie, selon le cas, l'un des différents messages d'échec. Voici la déclaration des trois premières clauses correspondant aux trois principaux messages, les autres cas s'en déduisent simplement.

```
méta_call(But, Func, Arité, Args) :-
    message(But,'?  '),
    méta( Func , Arité , _ , Args ).
méta_call(But, Func, Arité, Args) :-
    retract(méta_cut_fail),
    !,
    message(But,'-! ' ),
    fail.
méta_call(But, Func, Arité, Args) :-
    not (not clause(méta(Func, Arité,_, Args),_)),
    !,
    message(But,'-  '),
    fail.
```

Si nous reprenons l'exemple précédent, l'évaluation partielle du programme nous donne l'ensemble de clauses :

```
?- listing(méta).
méta(test,0,1,[]) :-
    méta_call_appel(test,1),
    méta_call(a,a,0,[]),
    !,
    méta_cut,
    méta_call(b,b,0,[]),
    méta_call_réussit(test,1).
méta(a,0,1,[]) :-
    méta_call_appel(a,1),
    méta_call(c,c,0,[]),
    méta_call_réussit(a,1).
méta(a,0,2,[]) :-
    méta_call_fait(a,2).
```

```
méta(b,0,1,[]) :-
    méta_call_appel(b,1),
    méta_call(c,c,0,[]),
    méta_call_réussit(b,1).
méta(b,0,2,[]) :-
    méta_call_appel(b,2),
    méta_call(c,c,0,[]),
    méta_call_réussit(b,2).
méta(c,0,1,[]) :-
    méta_call_fait(c,1).
```

Sa trace d'Eisenstadt est :

?- solve(test).

```
1   :   ?   test
2   :   >   test [1]
3   :       ?   a
4   :       >   a [1]
5   :           ?   c
6   :           +*  c [1]
7   :       +   a [1]
8   :   ---------------- !
9   :       ?   b
10  :       >   b [1]
11  :           ?   c
12  :           +*  c [1]
13  :       +   b [1]
14  :   +   test [1]
Yes
```

9.1.8 Intérêt

Les trois réalisations de trace avec le méta-interprète montrent la facilité d'utilisation de la méta-programmation pour réaliser des outils de simulation de l'interprète Prolog.

La simplicité de ce méta-interprète va nous permettre d'adapter les méthodes de détection d'erreurs basées sur le principe de Shapiro ainsi que d'étudier les extensions de la mise au point en Prolog.

La puissance de cette réalisation de méta-interprète avec la trace d'Eisenstadt, nous a permis à de nombreuses reprises de tester des algorithmes de détection d'erreurs en Prolog. Nous montrerons dans les parties suivantes l'intérêt d'une trace sophistiquée donnant des informations sur la dérivation.

9.2 La détection des boucles

Nous allons intégrer dans le méta-interprète proposé l'algorithme de détection de récursion infinie. Pour cela nous devons contrôler la dérivation.

La modification du méta-interprète se situe à trois niveaux : La structure de la pile de buts contenant les ancêtres d'un but, la gestion de la pile de buts et la similitude de deux buts.

9.2.1 La gestion des buts ancêtres

La pile de buts, structure de base du contrôle de la détection d'une récursivité infinie, peut être réalisée à l'aide d'une liste. Pour des raisons de simplicité, nous la réaliserons par ajout et retrait dans la base sous la forme d'un prédicat particulier *"méta_pile(Func, Arité, Nb, Args, X)"*, où *Func* est le symbole de prédicat du but courant, *Arité* le nombre d'arguments du but, *Nb* le numéro de la clause dans le paquet *Func/Arité*, *Args* la liste des arguments, X la position dans la pile du but. Ainsi, sur l'exemple :

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

Nous avons après trois itérations l'état courant :

```
méta_pile(f,0,1,[],3).
méta_pile(q,0,1,[],2).
méta_pile(p,0,1,[],1).
```

Dans ces conditions nous pouvons connaître par l'appel du prédicat *"méta_pile(Func, Arité, Nb, Args, X)"* le but à la position X dans la pile de buts.

Nous devons connaître la taille de la pile à un instant donné. Cette taille doit être modifiée lors de l'évolution de la méta-interprétation et, on utilise pour cela le prédicat *"taille_pile_num(X)"*, où X est la taille de la pile, l'état initial étant "*X = 0*". Nous réalisons la gestion de pile de la façon suivante :

```
empiler_but(Func, Arité, N_CL, Args, Y) :-
    retract(taille_pile_num(X)),
    Y is X + 1,
    asserta(taille_pile_num(Y)),
    asserta(méta_pile(Func, Arité, N_CL, Args, Y)).

dépiler_but(Func, Arité, N_CL, Args, X) :-
    retract(taille_pile_num(X)),
    Y is X - 1,
    asserta(taille_pile_num(Y)),
    retract(méta_pile(Func, Arité, N_CL, Args, X)).
```

Par cet ajout dans le méta-interprète nous connaîtrons l'ensemble des ancêtres du but courant et de les accéder. Il nous reste à réaliser le test de similitude entre un but ancêtre et le but courant.

9.2.2 *La similitude de deux buts*

Nous avons décrit dans son principe la similitude de deux buts. Nous réalisons ce test de la façon suivante :

```
méta_test_boucle(Func,Arité,N,A2,X) :-
    Y is X // 2,
    méta_pile(Func,Arité,N,A1,Y),% l'ancêtre du but
    similaire(A1,A2),% buts similaires
    !,
    But =.. [Func|A1],% le message
    message(But,'@  ',N),% de récursion
    nl,
    abort.          % arrêt (ou échec) de la résolution
méta_test_boucle(_,_,_,_,_).

similaire([],[]).
similaire([A1|L1],[A2|L2]) :-
    test_argument(A1,A2),% test de similitude
    similaire(L1,L2).% itération sur le reste

test_argument(A1,A2) :-
    atomic(A1),     % X1 est une constante
    !,
    A1 == A2.

test_argument(A1,A2) :-
    var(A1),        % X1 est une variable
    !,
    var(A2).

test_argument(A1,A2) :-
    nonvar(A2),     % X1 est une structure-
    functor(A1,F,Arité),functor(A2,F,Arité),
    structure_test(A1,A2).

structure_test(A1,A2) :-% comparaisons
    A1 =..[_|Args1],
    A2 =..[_|Args2],
    similaire(Args1,Args2),% sont-elles similaires
    !.
structure_test(A1,A2) :-% sous arborescence
    struct_member(A1,A2).

struct_member(A1,A2) :-
    numbervars(A1,0,N),% on renomme
    numbervars(A2,0,N),
    A1 = A2,        % A1 et A2 identiques ?
    !.
```

```
struct_member(A1,A2) :-
    functor(A2,F,Arité),
    A2 =..[F|Args],
    member(Z, Args),
    nonvar(Z),      % Z == sous-arborescence
    functor(Z, F, Arité),
    struct_member(A1, Z).
```

9.2.3 *Le programme*

Pour réaliser la détection d'une récursivité infinie dans le méta-interprète, nous devons modifier l'évaluation partielle pour tenir compte de la similitude. Nous pouvons procéder par une transformation de chaque clause par évaluation partielle :

```
méta(Func, Arité, Nb, Args) :-
    méta_call_appel(Func, Arité, Nb, Args),
    Partial_éval,
    méta_call_réussit(Func, Arité, Nb, Args).
```

Les prédicats *méta_call_appel*/4 et *méta_call_réussit*/4 contrôlent : l'appel et la réussite du but *"Func(Args)"* et réalisent la gestion de la pile de buts et le test de récursivité infinie. Nous avons ajouté quelques messages de la trace d'Eisenstadt. Nous les déclarons de la façon suivante :

```
méta_call_appel(Func, Arité, Nb, Args) :-
    But =.. [Func|Args],
    message(But,'>  ',Nb),
    empiler_but(Func, Arité, Nb, Args, Taille),
    méta_test_boucle(Func, Arité, N_CL, Args, Taille).

méta_call_appel(Func, Arité, Nb, Args) :-
    But =.. [Func|Args],
    message(But,'<  ', Nb),
    dépiler_but(Func, Arité, Nb, Args,_),
    !,
    fail.

méta_call_réussit(Func, Arité, Nb, Args) :-
    But =.. [Func|Args],
    message(But,'+  ', Nb),
    dépiler_but(Func, Arité, Nb, Args1, Taille),
    (
        true
        ;
        message(But,'^  ',Nb),
        empiler_but(Func, Arité, N_CL, Args1, Taille),
        !,
        fail
    ).
```

9.2.4 Exemples

Nous allons illustrer le test de détection de récursivité infinie sur quelques programmes engendrant des boucles.

Exemple 1.

```
p(X,Y) :- q(X,U), p(U,Y).
p(X,Y) :- a(X,Y).
r(X,Y) :- p(X,U),b(U,Y).
q(X,Y) :- r(X,Y).
a(0,1).
a(2,3).
a(4,5).
b(1,2).
b(3,4).
b(5,6).
```

Sur le but initial "r(X,6)", nous obtenons :

```
?- solve(r(X,6)).
    >  r(_341,6) [1]
    >  p(_341,_631) [1]
    >  q(_341,_895) [1]
    >  r(_341,_895) [1]
    >  p(_341,_1385) [1]
    @  p(_1799,_1801) [5,2]
[ execution aborted ]
| ?- listing(méta_pile).

méta_pile(p,2,1,[A,B],5). <- n
méta_pile(r,2,1,[A,B],4).
méta_pile(q,2,1,[A,B],3).
méta_pile(p,2,1,[A,B],2). <- p
méta_pile(r,2,1,[A,6],1).
```

Les messages de la forme '> But [N]' indiquent l'appel de la N-ième clause du paquet But. La détection d'un récursivité est signalée par le message '@ '. Il indique sur quel but de la pile la récursion est détectée. Dans cet exemple, c'est sur la clause 1 du paquet *p/2* entre les deux buts à la position 5 et 2 dans la pile que nous détectons une récursion infinie.

Exemple 2.

```
p(A,B) :- a(A,C), p(C,B).
p(A,B) :- a(A,B).
a(1,2).
a(2,1).
r(A,B) :- p(A,C), b(C,B).
```

Sur le but initial "r(X,5)", nous obtenons :

```
| ?- solve(r(X,5)).
    >  r(_39,5) [1]
    >  p(_39,_329) [1]
    >  a(1,2) [1]
    +  a(1,2) [1]
    >  p(2,_329) [1]
    >  a(2,1) [2]
    +  a(2,1) [2]
    >  p(1,_329) [1]
    >  a(1,2) [1]
    +  a(1,2) [1]
    >  p(2,_329) [1]
    >  a(2,1) [2]
    +  a(2,1) [2]
    >  p(1,_329) [1]
    >  a(1,2) [1]
    +  a(1,2) [1]
    >  p(2,_329) [1]
    @  p(2,_2750) [7,3]
[ execution aborted ]
| ?- listing(méta_pile).

méta_pile(p,2,1,[2,A],7). <- n
méta_pile(p,2,1,[1,A],6).
méta_pile(p,2,1,[2,A],5).
méta_pile(p,2,1,[1,A],4).
méta_pile(p,2,1,[2,A],3). <- p
méta_pile(p,2,1,[A,B],2).
méta_pile(r,2,1,[A,5],1).
```

Le message '+ But [N]' indique la réussite du but, par activation de la clause N. La récursivité est détectée entre les deux buts à la position 3 et 7 de la pile, sur la clause 1 du paquet *p/2*.

Exemple 3.

```
f(0).
f(A) :- f(s(A)).
```

Sur le but initial "f(X)", nous obtenons :

```
| ?- solve(f(X)).
    >  f(_40) [1]
    +  f(0) [1]
Yes
    ^  f(0) [1]
    >  f(_40) [2]
    >  f(s(_40)) [2]
```

```
    >  f(s(s(_40))) [2]
    >  f(s(s(s(_40)))) [2]
    @  f(s(_287)) [4,2]
[ execution aborted ]
| ?- listing(méta_pile).
méta_pile(f,1,2,[s(s(s(A)))],4).  <- n
méta_pile(f,1,2,[s(s(A))],3).
méta_pile(f,1,2,[s(A)],2).        <- p
méta_pile(f,1,2,[A],1).
```

Le message "^ But [N]" indique un retour-arrière sur la clause N.

Exemple 4.

```
g(0).
g(s(A)) :- g(A).
```

Sur le but initial "g(X)", nous obtenons :

```
| ?-  solve(g(X)).
    >  g(0) [1]
    +  g(0) [1]
Yes
    ^  g(0) [1]
    >  g(s(_278)) [2]
    +  g(0) [1]
    +  g(s(0)) [2]
Yes
    ^  g(s(0)) [2]
    ^  g(0) [1]
    >  g(s(_528)) [2]
    @  g(s(_715)) [2,1]
[ execution aborted ]
```

Dans cet exemple, l'ensemble des solutions de "g(X)" est infini. On gênère une récursion infinie sur le but "g(s(X))" par l'appel de la seconde clause.

9.2.5 Conclusion

Les exemples que nous avons pris sont relativement simples pour illustrer les possibilités de la détection d'une récursion infinie. Cependant, nous aimerions connaître les limites d'un tel système de détection. L'exemple suivant illustre les limites de la détection de boucles.

Exemple 5.

```
v(f(g(a))).
v(X):-v(f(X)).
v(X):-v(g(X)).
```

Effectuons une analyse de la résolution pour le but *v(a)*, avec échec de la résolution lors de la détection d'une boucle. Nous obtenons alors la trace partielle suivante :

```
| ?- analyse(v(a)).
@=> v(f(a)) [4,2]
@=> v(f(g(f(f(f(a)))))) [12,6]
@=> v(f(g(f(f(f(f(f(f(f(g(f(f(f(a))))))))))))))) [28,14]
....
```

Que se passe-t'il sur le but *"v(a)"* ? Pour, nous devons étudier l'arbre de recherche de ce but. La première remarque est que l'arbre de recherche possède une branche donnant une réussite. Mais, il existe un nombre infini de dérivations infinies à gauche de la solution. L'algorithme de détection les trouve une après l'autres. Ainsi, après avoir trouvé une boucle régulière, l'algorithme se met à parcourir des branches infinies après le retour-arrière. L'algorithme détectera un nombre infini de branches infinies.

Un détecteur de boucles constitue un apport important dans la mise au point d'un programme Prolog. Cependant, seule son intégration dans un environnement complet de mise au point permet d'en tirer les meilleur parti. En effet, les raisons d'une boucle peuvent être diverses et la disponibilité d'autres outils de mise au point intelligents est indispensable.

De plus, une étude complète des effets de la détection de boucle doit être envisagée, car le langage Prolog actuel permet d'écrire des programmes dont la sémantique déclarative est infiniment récursive, mais dont la sémantique opérationnelle est finie. C'est le cas lorsqu'on utilise des tests d'arrêts conditionnés par des événements extérieurs (lecture des termes ou caractères). Une amélioration simple de cette méthode est de fixer un seuil minimal à partir duquel on commence le test de récursion. Par exemple, lorsque la taille de la pile des ancêtres est supérieure à une certaine valeur fixée au préalable.

Enfin, il convient de remarquer que nous possédons un test performant de boucles régulières. Nous pouvons utiliser une négation de type *négation du monde clos* [Lloyd-87] et plus particulièrement celle de [Reiter-78]. Cette négation fait appel à la règle du monde clos, appelée règle CWA (*Closed Word Assumption*) :

> Si un atome clos A n'est pas une conséquence logique d'un programme alors déduire non A.

Cette règle du monde clos n'est pas une règle monotone. L'ajout d'une nouvelle règle à la base de connaissance peut modifier le comportement de cette négation.

D'une façon générale, pour utiliser la règle CWA, il faut montrer que le but A n'est pas une conséquence logique du programme P. La SLD-résolution n'est pas capable de le prouver car la logique du premier ordre n'est pas décidable, elle est seulement semi décidable : Si A n'est pas une conséquence logique du programme P l'arbre de recherche de la SLD-résolution peut être infinie. Ainsi, en utilisant notre méta-interprète avec le test de similarité d'un but, nous pouvons démontrer qu'un but n'est pas une conséquence logique d'un programme, par suite de l'échec de la résolution lorsqu'il y a une récursion infinie

Enfin, cette approche de la détection d'une dérivation infinie permet de vérifier qu'un but possède un arbre de résolution soit de profondeur finie, soit de largeur finie, Ce qui permet de vérifier l'utilisation sure de la négation, appelée négation close, qui permet d'obtenir une négation répondant à la règle CWA.

9.3 *La vérification d'une résolution*

9.3.1 *L'algorithme*

Nous pouvons appliquer directement les principes de la vérification par une simulation de la résolution pour détecter une erreur dans un programme. Cette simulation est réalisée par le méta-interprète. L'analyse de l'algorithme de la vérification d'une résolution permet de le réaliser de la façon suivante :

```
méta_call(But, Func, Arité, Args) :-
    copy_term(But,Goal),
    numbervars(Goal,0,_),
    méta(Func, Arité, Nb, Args,_),
    % une solution,
    % Args est la solution,
    % Nb la référence sur la clause
    % on mémorise une réussite pour le débug
    (clause(est_calculable(Goal),_)
        -> true
        ; asserta(est_calculable(Goal))),
    question_existe(But, Réponse)
    ( Réponse = true % juste on continue
        -> true
        % c'est la clause fausse, on s'arrête
        ; clause_fausse(But, Func, Arité, Nb)
    ).

méta_call(But, Func, Args, Args) :-
    % y a t'il une solution calculable
    \+ clause(est_calculable(But),_), % oui, échec
    % non, en existe-t-il une ?
    question_une_solution(But, Réponse),
    Réponse = true,
    insatisfaite(But).
```

Pour éviter des effets de bord sur les assertions évaluables dû au retour-arrière il ne faut pas tester l'échec terminal. Ainsi, il est nécessaire de mémoriser la réussite d'un but sous la forme du fait "*est_calculable(But)*". Cette réalisation utilise une approche similaire aux techniques des systèmes experts pour déduire et conserver des informations lors de la résolution. Ainsi, le prédicat *"question_existe/2"* permet de stocker toutes les réussites correctes d'un but. On construit de façon dynamique, par un apprentissage d'assertions, l'ensemble des solutions d'un but. Le prédicat *"question_existe(But, Réponse)"* répond aux spécifications suivantes :

- le système retourne la valeur "*Réponse = X*" s'il existe une assertion de valeur X concernant l'interprétation de cette solution.

- le système retourne "*Réponse = false*" s'il existe une assertion concernant l'interprétation d'un échec correct de cette solution.

- Il n'y a aucune assertion sur la solution. L'utilisateur détermine si la solution du but est correcte. Le système retourne "*Réponse = true*" lorsqu'elle est correcte. Dans le cas contraire, le système retourne "*Réponse = false*". Le système conserve la réponse de l'utilisateur sous la forme d'une assertion.

Ainsi, la réponse "*Réponse = true*" correspond toujours à une solution fausse, "*Réponse = false*" dans le cas contraire.

De même, le prédicat "*question_une_solution/2*" permet de stocker tous les échecs corrects de la résolution. Le prédicat "*question_une_solution(But, Réponse)*" répond aux spécifications suivantes :

- le système retourne la valeur "*Réponse = X*" s'il existe une assertion de valeur X concernant l'interprétation de l'échec de cette solution.

- le système retourne la valeur "*Réponse = true*" s'il existe une assertion concernant l'existence d'une solution correcte pour le but.

- Il n'y a aucune assertion sur la solution. L'utilisateur détermine si l'échec du but est correct. Le système retourne "*Réponse = true*" lorsque la solution est insatisfaite. Dans le cas contraire, le système retourne "*Réponse = false*". Le système conserve la réponse de l'utilisateur sous la forme d'une assertion.

De cette manière, la réponse "*Réponse = true*" correspond toujours à une solution insatisfaite, "*Réponse = false*" dans le cas contraire.

Une réalisation particulière de ces prédicats peut être :

```
question_existe(But, Rep) :-
    copy_term(But, Goal),
    clause(assertions(Goal),Eval),% assertion évaluable
    !,
    ( Eval -> Rep = true ; Rep = false ).

question_existe(But, Rep) :-
    copy_term(But, Goal),
    numbervars(Goal,0,_),
    clause(une_solution(Goal, Reponse),_),
    !,
    Rep = Reponse.
```

```
question_existe(But, false) :-
    copy_term(But, Goal),
    numbervars(Goal,0,_),
    clause(sans_solution(Goal,true),_),
    !.

question_existe(But, Rep) :-
    send(['(Question, sur le but :)',
        'la solution est-elle correcte ?'],[But]),
    oui_non(Reponse),
    copy_term(But,Goal),
    numbervars(Goal,0,_),
    asserta(une_solution(Goal, Reponse)),
    Rep = Reponse.

question_une_solution(But, Rep) :-
    copy_term(But, Goal),
    % assertion évaluable
    clause(assertions(Goal), Eval),
    !,
    ( Eval -> Rep = true ; Rep = false ).

question_une_solution(But, Rep) :-
    copy_term(But, Goal),
    numbervars(Goal, 0, _),
    clause(sans_solution(Goal, Reponse),_),
    !,
    Rep = Reponse.

question_une_solution(But, Rep) :-
    copy_term(But, Goal),
    calculable(Goal),
    clause(une_solution(Goal, true),_),
    !,
    Rep = true.

question_une_solution(But, Rep) :-
    send(['Question, sur le but :',
        'existe-t-il une solution ?'],[But]),
    oui_non(Reponse),
    copy_term(But, Goal),
    numbervars(Goal, 0 , _),
    asserta(sans_solution(Goal, Reponse)),
    Rep = Reponse.
```

Il faut remarquer dans la déclaration du prédicat "*question_une_solution/2*", un test particulier pour le cas où l'on connaît déjà une solution. En effet, la création du but "*Goal*" comme variante du but "*But*" dont on cherche l'interprétation, sert à de déterminer s'il existe une solution calculable et correcte pour le but "*But*". Le prédicat "*calculable(Goal)*" détermine s'il existe une solution calculable par une recherche en largeur d'abord des solutions de ses sous-buts.

On résoud ainsi le problème de la sémantique perturbée par l'effet d'un coupe-choix, comme le montrait l'exemple du prédicat "*go(X)*" et de la solution calculée "*X = 2*". Ainsi, si l'on connaît une solution correcte d'un but, on considère comme incorrect le fait d'obtenir un échec de la résolution de ce but.

L'assertion évaluable "*assertions(But) :- Evaluation.*" permet d'évaluer une solution du but. Ainsi, l'utilisateur peut spécifier des assertions sur une partie des programmes qu'il veut vérifier et diminuer le nombre de questions posées par le système. On se référera à la quatrième partie et au programme de l'annexe 3 pour exemple d'utilisation de ces assertions.

Enfin, les prédicats "clause_fausse" et "insatisfaite" permettent de présenter les causes probables de l'erreur trouvée. Dans le cas d'une solution insatisfaite on peut mettre en oeuvre différentes techniques d'analyse permettant de rechercher une cause. Par exemple, fournir un message particulier lorsque : aucune clause ne s'unifie avec le but incorret, le prédicat n'est pas correctement défini (mauvais nombre d'arguments), le prédicat n'existe pas. De plus, une analyse en largeur d'abord fournira une clause possèdant une coupure rouge (cf analyse de la coupure).

9.3.2 Exemples

Considérons le programme "*trier/2*", qui est incorrect car il manque le test de fin de récursion du prédicat "*insérer/3*" :

```
trier([X|XS],YS) :-
    trier(XS, ZS),
    insérer(X, ZS, YS).
trier([],[]).

insérer(X,[Y|YS],[Y|ZS]) :-
    X > Y,
    insérer(X, YS, ZS),
    !.
insérer(X,[Y|YS],[X, Y|YS]) :-
    X =< Y.
```

Sur le "*trier([2,3,1],X)*", nous obtenons la trace :

```
?- trier([2,3,1],X).
Question, sur le but :
trier([],[])
la solution est-elle correcte ? oui.
Question, sur le but :
insérer(1,[],_4497)
existe-t-il une solution ? oui.
dans l'exécution de la clause 1 du prédicat trier/2 :
        trier([1],YS) :-
            trier([],[]),
            insérer(1,[],YS).
```

```
====> Solution insatisfaite : "insérer(1,[],YS)"

Aucune clause du prédicat insérer/3 ne s'unifie.
```

Nous le remarquons dans cet exemple que le détecteur d'erreurs nous pose les deux types de questions de la mise au point :

- "une solution est-elle juste ?"
- *"y-a-t-il une solution ?"*

De plus, les déductions logiques de l'algorithme nous permettent de localiser la position de l'erreur dans le programme.

Dans l'exemple précédent, il n'y a pas d'effet de bord de la coupure sur la résolution. Lors de la résolution du but *"trier([2,3,1],X)"*, la première clause du prédicat *"insérer/3"* n'est pas exécutée et donc aucun effet de bord dû au coupe-choix ne se produit. Dans cet exemple, l'utilisation de la coupure est déclarative et la suppression de celle-ci du programme ne modifierait pas la sémantique déclarative du programme.

9.3.3 Conclusion

La méthode originale de mise au point que nous venons de présenter permet donc la vérification dynamique d'un programme Prolog possédant des prédicats méta-logiques, des coupures et des négations.

Nous disposons ainsi d'un outil puissant fondé sur la vérification d'assertions en cours de résolution. Il faut maintenant intégrer dans un outil interactif cette mise au point, une détection de boucle infinie et enfin une trace de la résolution.

9.4 Conclusion sur la réalisation

Grace à ces techniques de recherche d'erreurs, l'utilisateur possède un système performant de mise au point de programmes *méta-logiques* : détection d'une boucle infinie, détection d'une solution incorrecte.

La seule limite d'un tel système est l'impossibilité de vérifier un ensemble de solutions. Lorsque l'on applique l'algorithme de vérification à une résolution correcte, chaque solution calculée est une solution juste. Il n'existe pas dans une résolution vérifiée, une solution incorrecte.

Ainsi, lorsque le programmeur réalise la mise au point d'un programme, il effectue une série de tests caractéristiques pour détecter une solution incorrecte. Quand les erreurs ont été corrigés, toutes les solutions calculées pour les résolutions des buts tests sont des solutions justes. À cet état du développement, le programmeur souhaitera savoir si les ensembles de solutions sont corrects selon son interprétation.

Lorsque le programme est écrit en Prolog pur, il est possible de vérifier les ensembles de solutions théoriques en les calculant. En revanche, lorsqu'on utilise la coupure, l'effet est de réduire ces ensembles de solutions. Dans le cas de la coupure sûre, comme il n'y a pas de restriction des ensembles de solutions et la méthode de vérification de la résolution dans ce cas consisterait à déterminer l'ensemble incorrectes. Si le système est capable de fournir les ensembles de solutions de la résolution d'un but, le programmeur détectera celui qui est faux.

Ajouter une telle méthode de recherche d'erreurs, pour traiter le cas de l'ensemble de solutions incorrectes, alourdirait d'une façon importante la technique de mise au point. En revanche, fournir les ensembles de solutions lorsque la résolution est correcte, serait une aide appréciable pour le programmeur.

Lorsqu'une résolution est correcte, le programmeur est capable d'analyser les ensembles de solutions pour détecter une incohérence par rapport à son modèle. En revanche, modéliser un tel raisonnement serait complexe.

Partie 4

Un système Interactif

10. Le matériel

10.1 Interface graphique

Les choix d'une interface pour la réalisation d'un logiciel quelconque reposent bien souvent sur les possibilités de la machine d'accueil. Nous avons développé notre application sur des stations de travail SUN/3, de SUN Microsystems.

L'interface utilisateur fondée sur un système de fenêtre rend le développement des applications plus aisé. Pour réaliser un système de mise au point pour le langage Prolog, nous avons choisi de concevoir une interface tirant parti du système multifenêtre.

En effet, il est très intéressant pour l'utilisateur de disposer du maximum d'informations sur son écran. Le multifenêtrage permet d'obtenir une quantité d'informations importante et un très bon confort d'utilisation.

Sur les machines SUN, les systèmes de multifenêtrage disponibles sont :

- SunView qui est l'interface standard de ces stations de travail [SunView-85].
- X11, le système de multifenêtrage du MIT fondé sur des concepts d'un serveur graphique, utilisant un protocole de communication sur des réseaux pour l'interaction entre l'utilisateur et une application [X11R3].
- NeWS, le système de multifenêtrage développé par SUN et fondé sur les mêmes concepts que le serveur X11. Ce système utilise un langage de communication qui est un sur-ensemble du langage PostScript [NeWS-88].

Le choix d'une interface particulière est relativement difficile. Nous avons délibérément écarté le système standard SunView des machines SUN, car ce système est fermé, ce qui n'est pas le cas des deux autres.

L'originalité des systèmes de multifenêtrages fondés sur des communications entre une application et le serveur graphique que l'on peut faire s'exécuter sur des machines distinctes reliées par un réseaux.

L'interface graphique que nous avons choisie, est le serveur NeWS. Pour utiliser d'une façon efficace le graphique X11, l'utilisateurdoit mettre en œuvre des procédures particulières pour gérer la souris, l'écran, le clavier dont la mise au point demande un temps de développement important. L'originalité du serveur graphique NeWS est justement d'offrir une grande souplesse dans l'écriture des interfaces graphiques, grace à un langage interprété de la famille des langages à objets.

NeWS est un serveur graphique utilisant une extension objet du langage PostScript. PostScript est un langage interprété de haut niveau qui offre des primitives graphiques très puissantes [PostScript-85]. L'extension offerte par le serveur NeWS permet de gérer de façon intéressante l'interactivité, mais surtout une flexibilité du serveur inégalable par les autres systèmes de multifenêtrage. NeWS offre une plate-forme, indépendante de la machine d'accueil et du système d'exploitation, pour réaliser des applications utilisant des fenêtres ou bien des interfaces d'un haut niveau.

10.1.1Architecture de NeWS

Le serveur graphique NeWS utilise la puissance d'une station de travail pour allouer des ressources dans un environnement distribué. NeWS tourne sur une machine qui dispose d'un environnement de haute définition graphique. Une application, appelée *client*, envoie des messages au serveur NeWS pour afficher des images sur un écran. Ce client peut résider quelque part sur un réseau de communication accessible par le serveur.

Tous les clients NeWS envoient, à travers le réseau, des programmes que le serveur interprète. Dans ces conditions, il est très facile, pour un client quelconque, de réaliser des extensions par la programmation de nouvelles fonctionnalités pour le serveur NeWS. Le langage PostScript est utilisé pour dessiner des images sur l'écran, l'extension objet est utilisée pour gérer l'interaction entre les différents objet NeWS et les clients NeWS.

Pour le serveur NeWS il existe trois types de communications :

- Le client envoie des programmes PostScript qui seront interprétés par le serveur.
- Le Serveur reçoit des données du clavier ou de la souris et gère l'écran.
- Les objets internes émettent des messages vers les clients ou les autres objets.

Le concept de base pour la communication interne est l'événement. Ce mécanisme permet de faire communiquer des clients entre eux, ou faire communiquer les périphériques avec des objets NeWS.

La première tache d'un client est d'établir une connection avec le serveur NeWS. Lors de son exécution, le client construit des messages NeWS et les envoie au serveur. Le serveur, lorsqu'il reçoit ces messages, les interprète de plusieurs façons, dont les principales sont :

- une modification de l'écran,
- une définition d'une procédure pouvant être appelée plus tard,
- une émission d'un message vers le client.

Pour un client, l'application existe en deux morceaux. La première est incluse dans le client, la seconde est incluse dans le serveur NeWS sous la forme de procédures définies par le client, ou chargées automatiquement par le serveur lors de l'initialisation de celui-ci, ou bien chargées par un autre client.

Le serveur NeWS, lors de son initialisation, charge un système de gestion de fenêtre. Ce système est utilisable par un client pour gérer les interactions entre la souris et les objets crées par les clients. Ainsi par défaut, l'utilisateur dispose d'une interface graphique qu'il peut moduler à volonter selon ses applications.

10.1.2Programmer avec NeWS

Un programme client existe en deux parties : la première est écrite en PostScript et réside dans le serveur NeWS, la seconde est incluse dans le client et dialogue avec le serveur NeWS à travers une connection du réseau. L'interface d'une application peut être vue à plusieurs niveaux.

- Le programmeur écrit avec le langage NeWS, PostScript et les objets, des programmes d'interfaces. Les menus et les fenêtres sont des exemples de ces interfaces, réalisés à partir de classes initiales. NeWS offre une convention pour définir une interface orientée objet pour les fenêtres, les menus, les sélections...

- Le programmeur écrit des clients dans un certain langage. Ces clients envoyent des programmes NeWS au serveur et reçoivent du serveur des événements. Par défaut, NeWS offre un pré-processeur C permettant d'utiliser l'interface NeWS à partir d'un programme écrit en C.

Dans ces conditions, la réalisation d'une application utilisant le serveur NeWS se découpe en deux parties essentielles :

- la création de l'interface par l'utilisation du langage NeWS : création de nouvelles classes, utilisation des classes prédéfinies, création d'objets, interaction entre les objets, interaction avec le client...

- l'écriture du client par l'utilisation des procédures NeWS ou des méthodes des objets crées pour l'interface.

La création d'une interface graphique à partir des classes prédéfinies de NeWS est très facile. Le système permet de créer de façon dynamique tous les objets que l'on veut par un dialogue direct avec le serveur en langage NeWS. Pour cela on utilise un programme spécial, *"psh(1)"*, qui se connecte au serveur NeWS et permet la communication entre l'utilisateur et le serveur dans le cadre d'une session de travail NeWS. En voici un exemple :

```
moon: /ia/debarb> psh

executive              % mode interactif
Welcome to NeWS Version 1.1 % réponse du serveur
4 5 add =              % une addition
```

```
9                    % le résultat
/spot {              % Une procédure 'X Y spot'
    20 0 360 arc fill
} def
800 300 spot         % l'execution du tracé d'un cercle
quit
```

Cet exemple montre la facilité du dialogue avec le serveur NeWS. Un langage de communication permet d'exprimer directement les actions que le serveur doit effectuer. De même, l'utilisateur peut construire une interface graphique par cette méthode sans écrire une seule ligne de code du client.

10.1.3L'interface utilisateur

L' utilisateur va créer l'interface de son application sans nécessairement écrire le client. Lorsque l'interface sera suffisamment complet, le client n'aura plus qu'à charger les descriptions avant de s'en servir.

Pour la création de l'interface, l'utilisateur dispose d'une série de classes prédéfinies pouvant être combinées entres elles pour créer des objets NeWS. Les principales classes disponibles sont : les fenêtres, les menus, les ascenseurs, les boutons...

Le langage NeWS utilise une programmation orientée objet similaire à Smalltalk [Smalltalk-80]. Une définition de classe dans le langage NeWS à l'aalure suivante :

```
/nom_de_classe SUPER_CLASSE
    dictbegin
        instance de super variables
    dictend
    classbegin
        variable de la classe
        méthodes de la classe
    classend
def
```

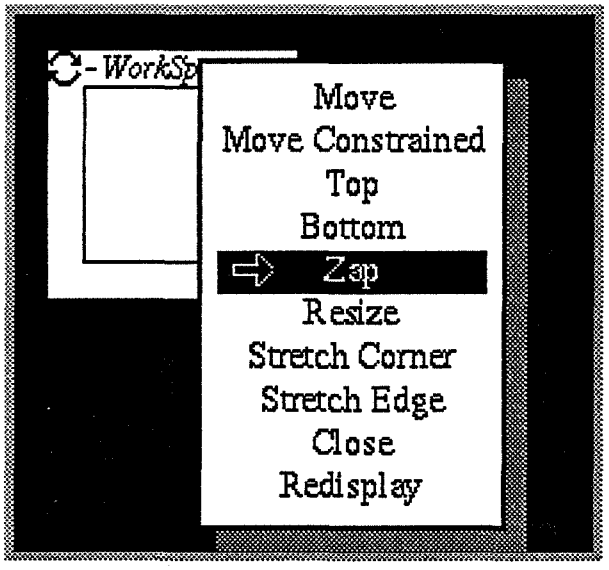
À partir de la classe prédéfinie "DefaultWindow", qui définit des objets de type fenêtre, pour pouvons créer une autre classe, de la façon suivante :

```
/Ma_Fenêtre DefaultWindow
    dictbegin
        /FrameLabel ( WorkSpace ) def
    dictend
    classbegin
        /PaintClient { .5 fillcanvas } def
        /PaintIcon { .5 fillcanvas 0 strokecanvas } def
    classend def
```

De cette nouvelle classe, on pourra créer une instance. Cette création se fait toujours par la méthode "new", les dimensions et la position de la fenêtre sont obtenus par la méthode "reshape" et la visualisation de cette instance s'obtiens par la méthode "map" ; nous obtenons ainsi le programme NeWS suivant :

```
/fenêtre
    % la création d'une instance
    framebuffer /new Ma_Fenêtre send
def
% dimensions
100 100 100 100 /reshape fenêtre send
% affichage
/map fenêtre send
```

L'exécutions de ce programme NeWS sur un serveur donne la fenêtre représentée ci-dessous. Cette fenêtre contient par défaut un menu permettant de la déplacer, la dé-



truire, la modifier, la fermer.... Le menu par défaut, hérté de la super classe et redéfinissable au niveau de la classe de l'objet, est affiché au moyen de la souris.

Cette téchnique pour créer une avec NeWS permet à un programme Prolog d'utiliser un système graphique de haut niveau sans devoir définir en Prolog toutes les procédures pour gérer des fenêtres, des menus, des textes, des ascenseurs, des boutons... La seule contrainte particulière est que l'interprète Prolog doit lire et écrire des messages NeWS par l'intermédiaire de connections TCP/IP. La réalisation complète de l'interface NeWS de notre système de mise au point, par l'utilisation des classes disponibles, n'a duré qu'une journée.

10.2 Le système Prolog

Le choix du système Prolog a été fait afin de garantir la portabilité du système que nous voulions réaliser. Nous avons porté notre choix sur une syntaxe proche de celle de C-Prolog et sur la sémantique de certains prédicats prédéfinis : le coupe choix, la négation, le "call". Les performances du système n'étaient pas primordiales, nous avions décidé de choisir un système disposant d'un compilateur Prolog efficace, dont la sémantique du code compilé était commune à celle du code interprété. Notre choix s'est porté sur le système Quintus-Prolog [Quintus-89], qui nous a donné pleine satisfaction.

Ce système dispose d'une interface TCP/IP permettant la communication à travers un réseau, entre le système Prolog et une application quelconque. Son utilisation est très simple et nous a rendu de grands services pour le développement du dialogue dans l'interface sous NeWS.

Les prédicats que nous avons définis sont les suivants :

connect/0
: réalise la connexion au serveur NeWS et initialise l'interface de l'utilisateur,

connect_to/2
: réalise une connexion à la machine du réseau sur laquelle se trouve le serveur NeWS,

send/2
: permet d'émettre un message NeWS, constitué d'un format et de paramètres,

get_char/1
: permet la lecture de caractères depuis la connexion TCP/IP ou depuis un fichier,

méta_read/1
: permet la lecture de termes en utilisant le prédicat get_char/1,

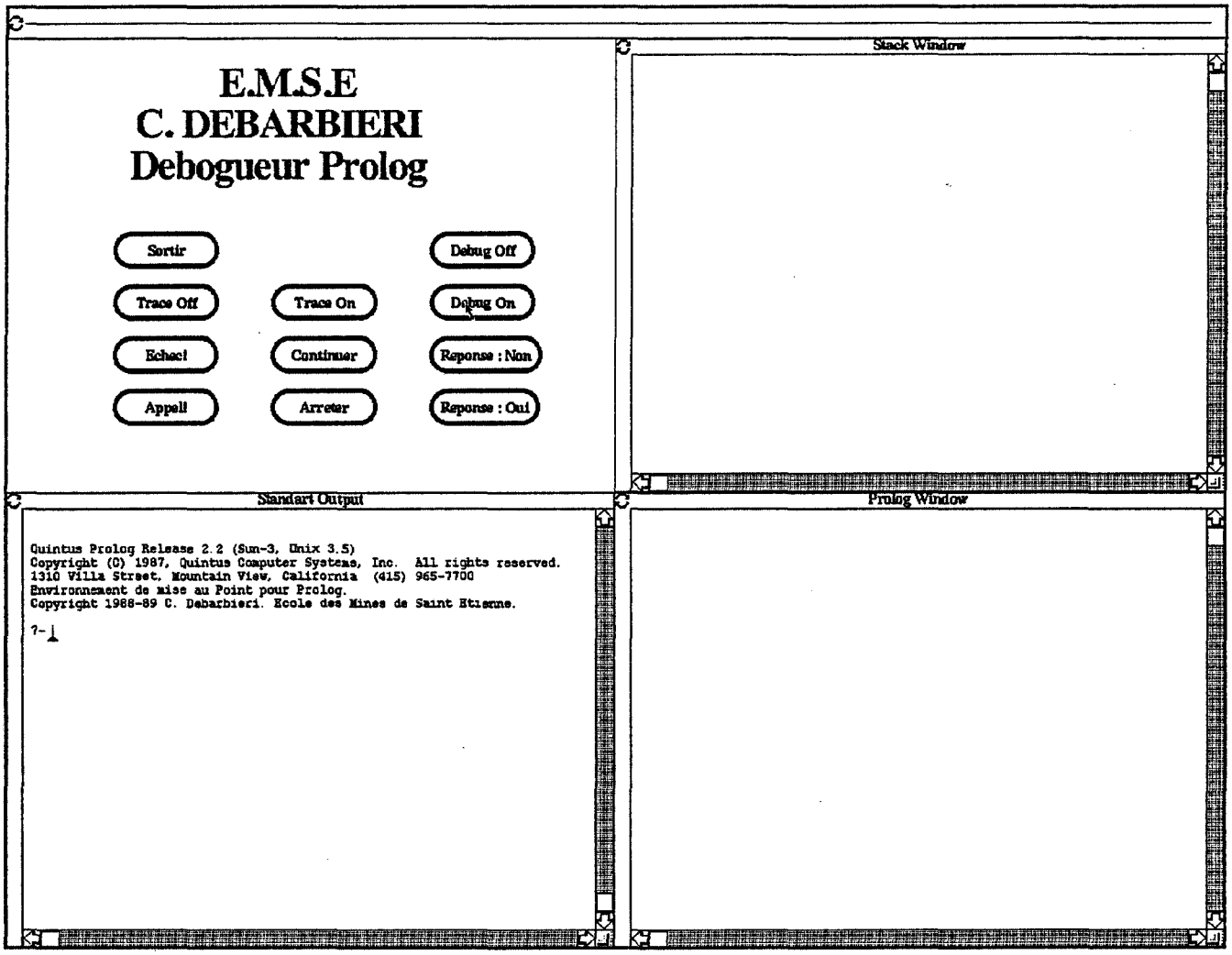
shutdown/0
: arrête la session NeWS.

Voici une utilisation de ces prédicats réalisant un dialogue entre le serveur NeWS et le système Prolog :

```
?- connect_to(2000,moon),
send('executive\n',[]),
send('(@\n) print\n',[]),
send('(hello_boy(X,Y).\n) print\n',[]]),
repeat, get_char(X),(X=64,!;put(X),fail).
Welcome to NeWS Version 1.1
```

```
?- méta_read(X),shutdown.
X = hello_boy(_495,_639)
Yes
?-connect.
```

La connexion avec le serveur NeWS et la création des objets utilisés par l'interface donne l'écran suivant :

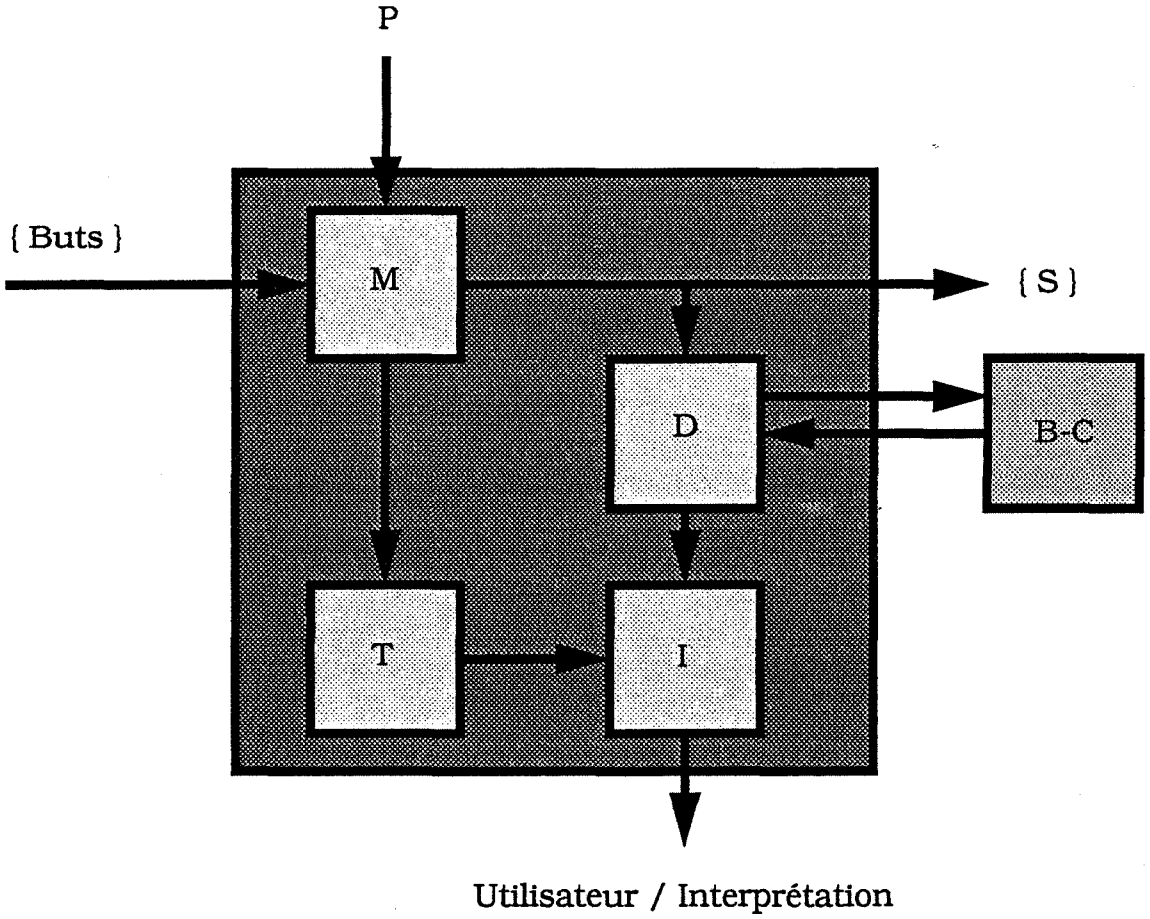


Nous reviendrons dans la partie suivante sur la description des différents fonctionnalités de cette interface.

11. L'application

Ce chapitre présente une description de la réalisation de notre système de mise au point qui applique les principes de mise au point en Prolog : la détection d'une solution incorrecte ou d'une boucle régulière.

L'architecture du système que nous avons réalisé peut être décrit par le schéma :



où :

P	Le programme Prolog que l'on veut tester et valider.
{ Buts }	L'ensemble des buts servant au test.
{ S }	L'ensemble des solutions de chaque but test, c'est la sémantique actuelle du programme.
M	Le méta-interprète décrit dans le chapitre 9.
T	Le système de trace de la résolution.
D	Le système de détection d'erreurs.

I L'interface NeWS de dialogue avec l'utilisateur

B-C La base de connaissance associè à P, formalisant la sémantique attendue.

11.1 L'interface utilisateur

L'interface du système de mise au point présente à l'écran trois informations importantes lors du déroulement de la détection d'erreurs dans des fenêtres texte :

- Une fenêtre, nommée "Standart Output", pour les entrées et sorties standard, SO en abrégé.
- Une fenêtre, nommée "Prolog Window", pour la clause en cours d'évaluation, PW en abrégé.
- Une fenêtre, nommée "Stack Window", pour la pile des ancêtres du but courant, SW en abrégé.

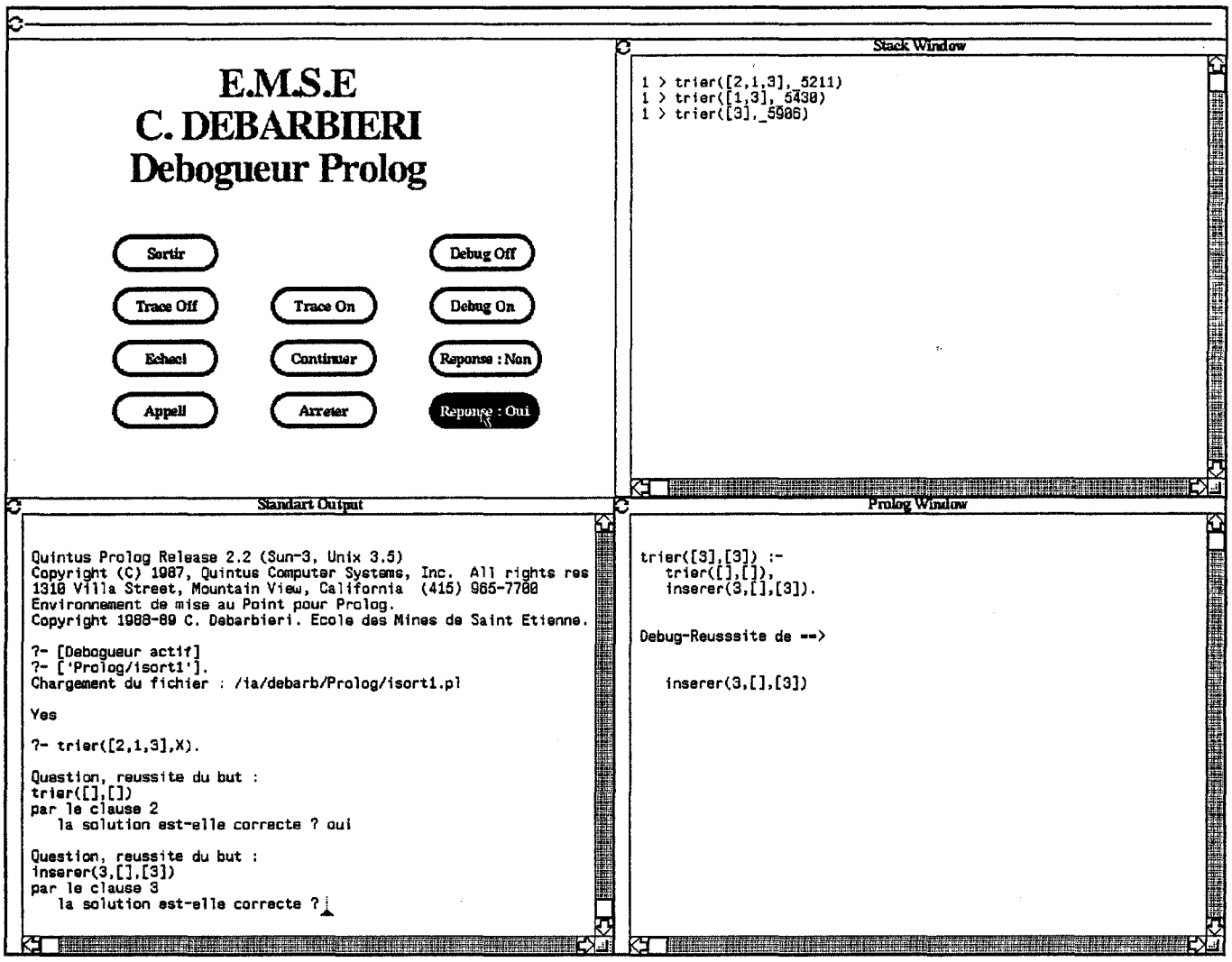
La fenêtre SO constitue pour l'utilisateur un écran virtuel servant aux entrées et aux sorties de l'interprète Prolog. L'utilisateur dialogue dans cette fenêtre avec le système Prolog pour la demande de résolution, la consultation d'un programme, la modification des paramètres de l'interface...

La fenêtre PW visualise la position de l'appel d'un but d'une clause lors de la résolution. L'utilisateur peut désirer connaître la clause en cours d'évaluation. De même, lorsque l'on visualise cette clause, il est très facile de donner la valeur des instances de ses variables. L'utilisateur dispose ainsi d'une facilité plus grande pour déterminer une erreur dans les liens des variables. Nous avons accru cette facilité en visualisant les noms originaux des variables telles qu'ils figurent dans les programmes.

La fenêtre SW contient le chemin dans l'arbre de recherche du but initial. Lors de la réalisation du détecteur de boucles, nous avons remarqué que cette information sur les ancêtres du but en cours d'évaluation était importante. Cette information devient essentielle lorsque l'on réalise une trace intelligente.

De plus, pour un dialogue plus efficace et rapide entre l'utilisateur et le système de mise au point, un ensemble de boutons sont disposés sur l'écran. Ces boutons permettent le dialogue avec le système de trace ou le système de détection d'erreurs.

L'utilisateur dispose de toutes ces informations sur l'écran à tout moment d'une session de mise au point. Voici l'écran d'une session de mise au point :



Dans la fenêtre SW, la partie à gauche de signe ">" est l'ordre d'apparition de la clause activant le but ancêtre à droite de ce signe. Nous reviendrons un peu plus loin sur l'utilisation des divers boutons de dialogue et des messages fournis par l'interface.

11.2 Le système de détection d'erreurs

Le système de détection d'erreurs utilise les deux algorithmes que nous avons présentés dans la partie précédente. Pour améliorer l'interaction entre l'utilisateur et le système de mise au point, le méta-interprète a été modifié à plusieurs niveaux. Pour réaliser un système intelligent nous avons complété les messages du détecteur d'erreurs pour les rendre plus précis.

Les informations fournis par l'interface lors de la vérification d'une résolution sont : les ancêtres du but courant, la clause en cours d'évaluation, les messages du détecteur... Un certain nombre de prédicats permettent de fournir ces informations au travers de l'interface à l'utilisateur. L'algorithme de vérification d'erreurs est réalisé par la déclaration :

```
méta_call(But, Func, Ar, Arg, Clause) :-
    copy_term(But, Goal),
    numbervars(Goal, 0 ,_),
    méta(Func, Ar, Num, Arg, _, _),
    (clause(calculable(Goal), _)
        -> true
        ; asserta(calculable(Goal))),
    question_existe(Func, Ar, Arg, Num, Rep, Clause),
    ( Rep = true
        -> true
        ; clause_fausse(But, Func, Ar, Num))
   ;
    \+ clause(calculable(But), _),
    question_une_solution(Func, Ar, Arg, Rep, Clause),
    Rep = true,
    insatisfaite(But).
```

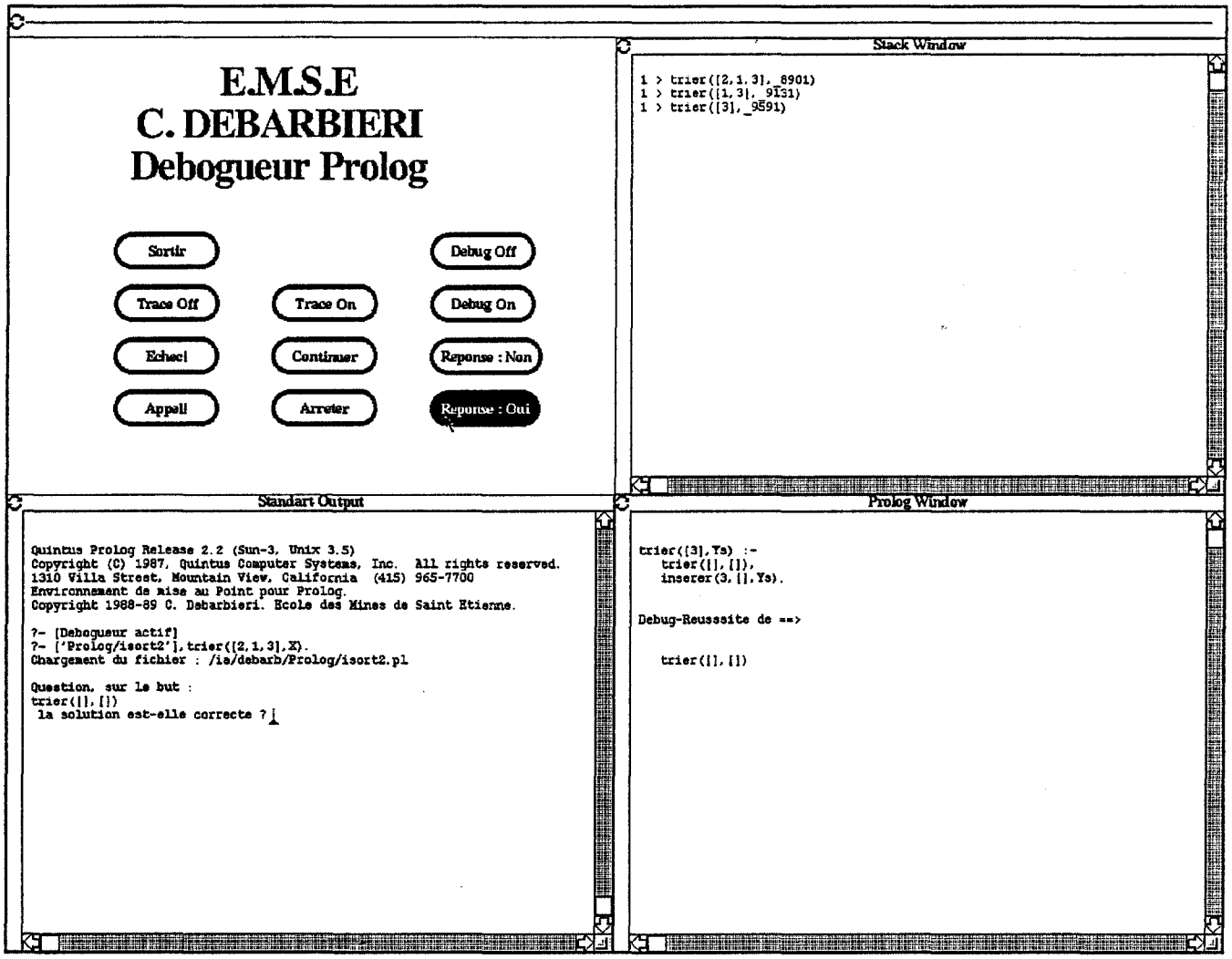
Cet algorithme correspond à un parcours en profondeur d'abord de l'arbre de résolution avec une vérification pour chaque solution calculée. L'argument "*Clause*" est la clause en cours d'évaluation dans laquelle est appelé le but courant "*But*". Les prédicats de questions sur les solutions permettent de visualiser, dans la fenêtre PW, cette clause avec les instances de ses arguments, les noms des variables libres, un message sur le but en cours d'évaluation. Dans la fenêtre SO, ils affichent les messages sur la solution courante et lisent les réponses de l'utilisateur.

De plus, les prédicats de gestion de la pile des buts, servant au test de récursion, sont modifiés pour afficher, dans la fenêtre SW, la liste des ancêtres du but courant.

11.3 Exemple

Considérons l'exemple du programme incorrect du chapitre 10. Ce programme "*trier*" permet de montrer les deux messages de la détection d'erreurs. L'activation et la désactivation du mode de détection d'erreurs se fait par l'utilisation des deux boutons de dialogue : "Debug On" et "Debug Off". Initialement, le système de mise au point n'est pas en mode de détection d'erreurs, il faut donc l'activer.

Pour ce programme, le but initial "*trier([2,3,1],X)*" possède une solution insatisfaite "*X=[1,2,3]*", interprétation d'un tri croissant de la liste "*[2,3,1]*". La vérification de la résolution de ce but nous donne l'écran initial suivant :



Le système signale l'activation de la vérification d'une résolution par un message sur la fenêtre SO : "[Débogueur actif]".

Lors de la vérification de la résolution d'un but avec une solution incorrecte, et après le chargement du fichier source, le système affiche diverses informations sur l'écran :

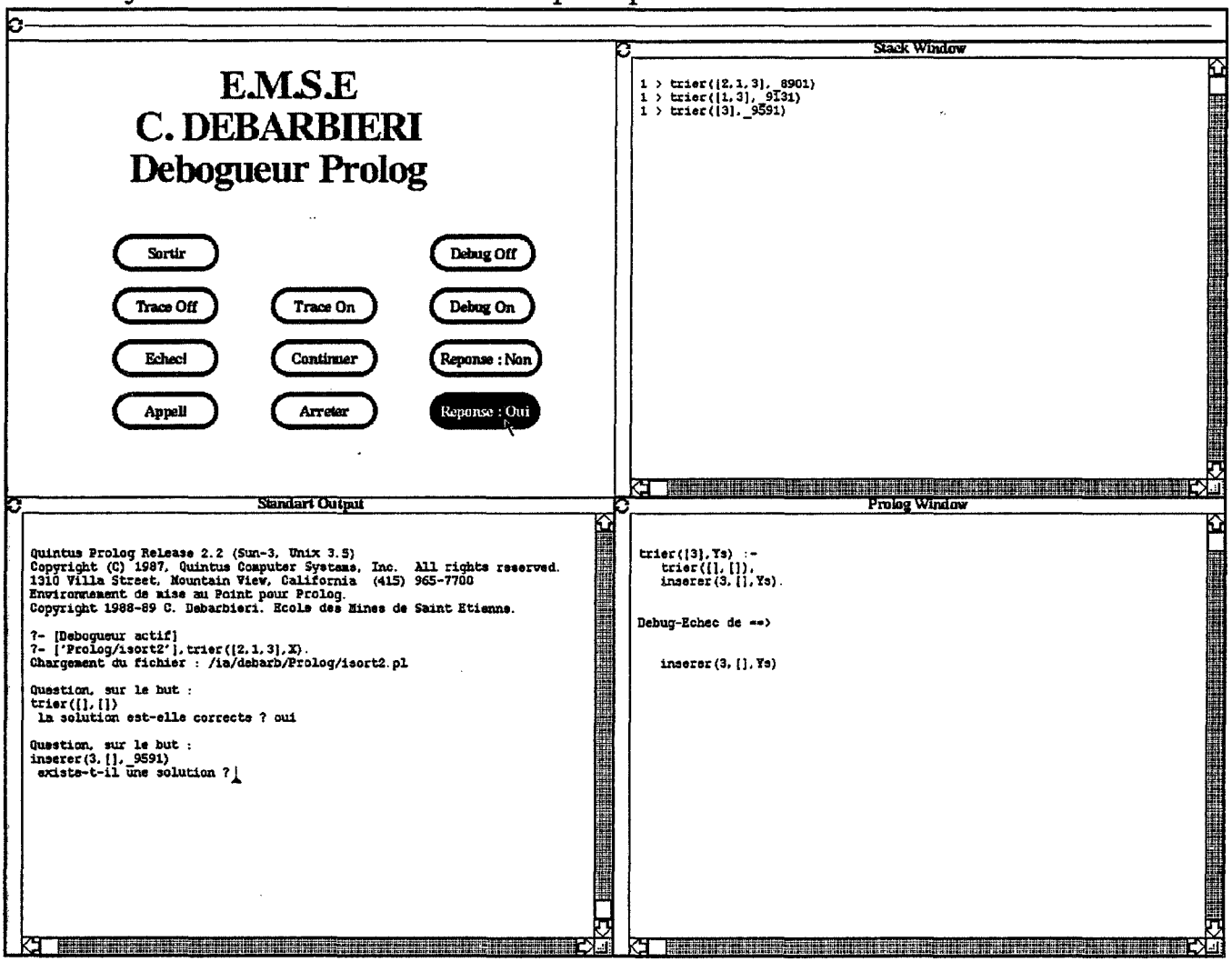
- dans la fenêtre PW : la clause en cours d'évaluation avec l'instance des arguments, les noms des variables libres, l'information sur l'échec ou la réussite du but courant,
- dans la fenêtre SW : la suite des buts ancêtres avec leurs arguments d'appel, les clauses actives associées aux ancêtres,
- dans la fenêtre SO : le message du détecteur d'erreurs sur le but courant, correspondant à la détection d'une solution fausse ou d'une solution insatisfaite

Dans cet exemple, le détecteur d'erreurs est le suivant :

"une solution est trouvée pour le but *"trier([],[])"*, est elle correcte ?".

Une pression sur le bouton "Oui" correspond à une réponse affirmative, sur le bouton "Non" à une réponse négative. L'utilisateur peut donner par l'intermédiaire la fenêtre SO, l'interprétation de la solution, si elle est correcte ou si elle est incorrecte.

Le système continu la vérification pour produire l'écran suivant :

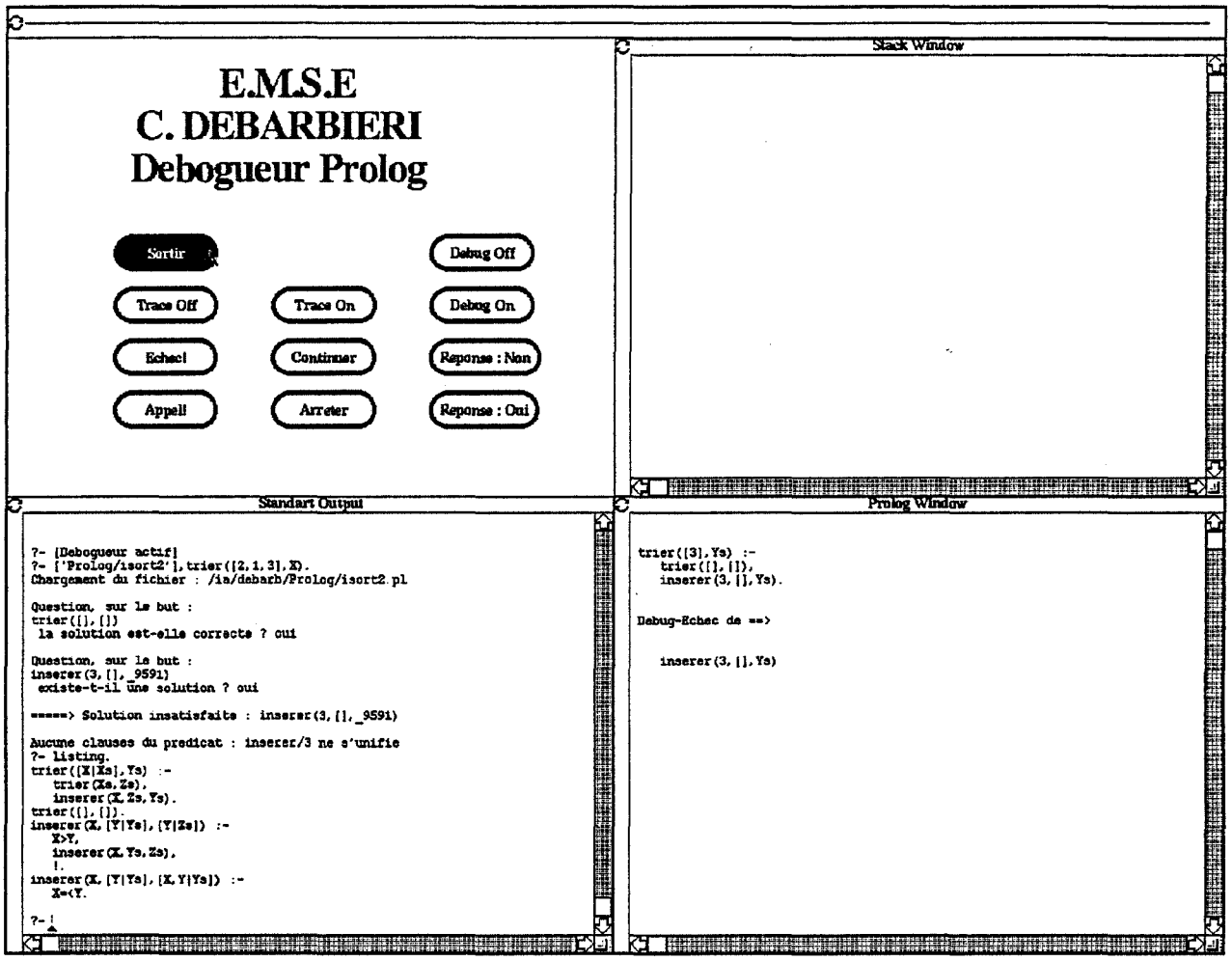


Le système signale l'échec d'un but par un nouveau message :

"échec du but insérer(1,[],Ys), existe-t-il une solution ?".

Dans ce cas, il existe une solution insatisfaite *"Ys=[1]"* dans l'interprétation attendue du prédicat *"insérer/3"*. L'action sur le bouton *"Oui"* signalera qu'il existe une solution pour ce but.

À partir de ces informations, le système de détection d'erreurs signale qu'il vient de trouver une solution insatisfaite et produit l'écran :



Le message donné par le système de détection d'erreurs est :

"J'ai trouvé une solution insatisfaite,
aucune unification possible du but"

Le système analyse la solution insatisfaite pour donner des renseignements complémentaires à l'utilisateur. Celui-ci dispose d'informations sur les circonstances de l'arrêt sur une solution insatisfaite.

Ce dernier écran montre l'intérêt de disposer de ces informations suite à une solution incorrecte lors d'une résolution. Dans cet exemple, un affichage du programme complète l'analyse de la solution insatisfaite. La visualisation simultanée de la clause contenant le but avec une solution insatisfaite et du programme, permet de trouver la déclaration manquante du prédicat *"insérer/3"*.

Pour une mise au point plus efficace, nous avons intégré au système de mise au point un mécanisme de mémorisation des solutions correctes. En effet, lorsqu'un utilisateur réalise une vérification d'une résolution, il fournit un ensemble de solutions correctes que le système conserve comme assertions pour d'autres vérifications.

Ainsi, si nous conservons ces informations dans un fichier particulier, consulté automatiquement à l'initialisation du système, l'utilisateur dispose d'un moyen pour ne pas recommencer toute la phase du dialogue jusqu'au point où précédemment le système avait trouvé une erreur. Le système va vérifier la résolution en utilisant les informations déjà connues.

Cette technique est une vérification de résolution avec apprentissage. Ainsi, l'utilisateur dispose d'un moyen efficace d'acquisition de connaissances sur son domaine, soit lors d'une génération de tests, soit lors d'une vérification. La génération de tests permet d'obtenir, à partir de programmes que l'on sait corrects, une base de connaissances pour la vérification de nouveaux programmes.

Pour l'illustrer, prenons le programme "*trier/2*" des parties précédentes qui contient deux erreurs de programmation : une clause fausse et un test d'arrêt oublié.

La vérification de la résolution du but "*trier([2,1,3],X)*" nous donne les messages :

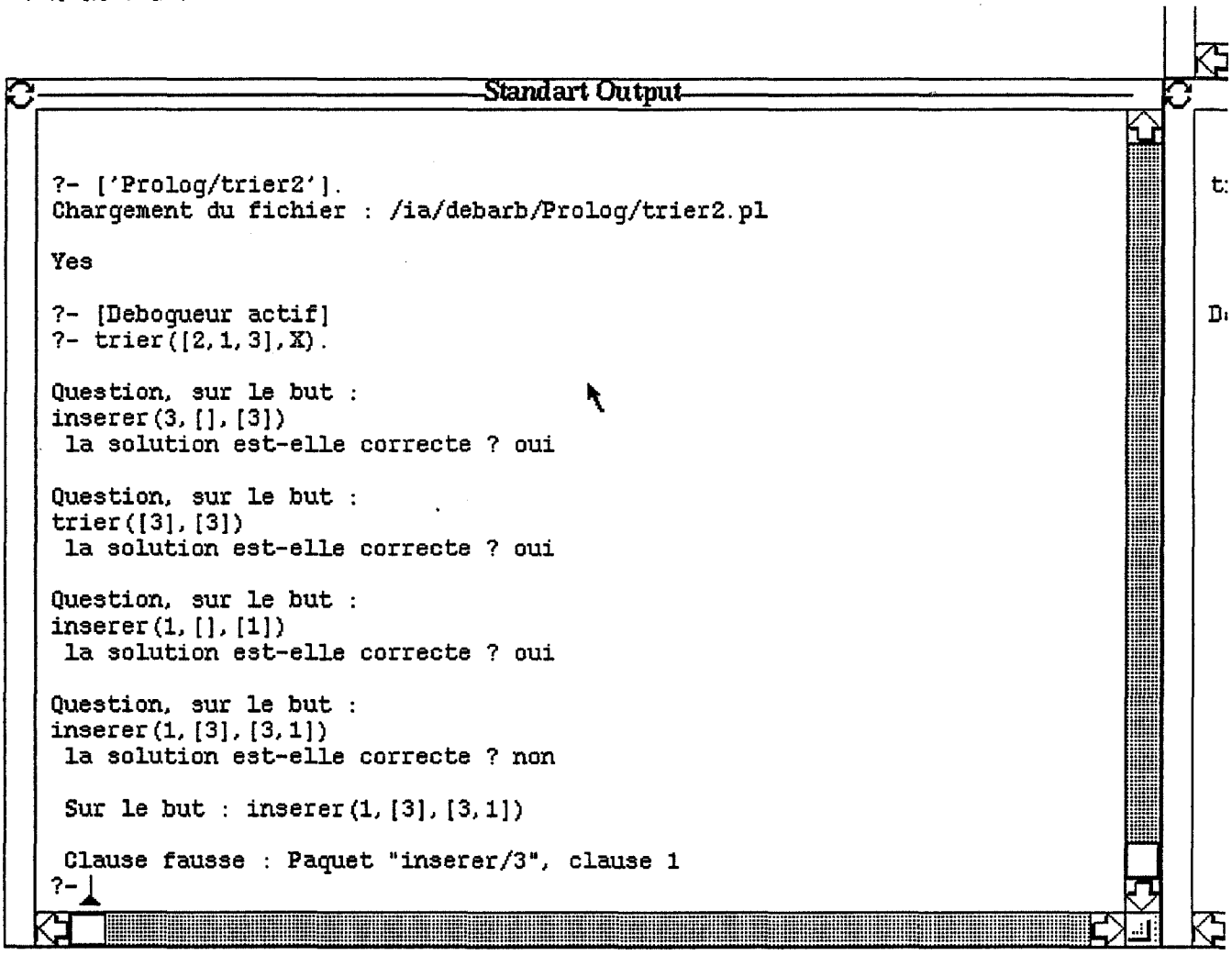
Standart Output

```
Yes

?- [Debogueur actif]
?- trier([2,1,3],X).

Question, sur le but :
trier([],[])
 la solution est-elle correcte ? oui

Question, sur le but :
inserer(3,[],_5684)
 existe-t-il une solution ? oui

=====> Solution insatisfaite : inserer(3,[],_5684)

Aucune clauses du predicat : inserer/3 ne s'unifie
?- listing.
trier([X|Xs],Ys) :-
   trier(Xs,Zs),
   inserer(X,Zs,Ys).
trier([],[]).
inserer(X,[Y|Ys],[Y|Zs]) :-
   Y>X,
   inserer(X,Ys,Zs),
   !.
inserer(X,[Y|Ys],[X,Y|Ys]).

?-
```

La correction de cette erreur, l'ajout de la déclaration manquante "*insérer(X,[],[X]).*", nous permet de poursuivre la mise au point.

Sans l'apprentissage, l'utilisateur serait obligé de reprendre toute l'interaction précédente. En revanche, avec l'apprentissage, la détection d'erreurs débute directement sur la solution obtenue pour le but "*insérer(3,[],X)*", soit la solution "*X=[3]*", qui est correcte et la vérification continue jusqu'à détecter l'erreur suivante.

Lorsque le système détecte la seconde erreur, une solution incorrecte "*insérer(1,[3],[1,3])*", nous obtenons l'écran :



Cette fonctionnalité du système de vérification permet d'aborder la mise au point comme une opération à long terme qui ne peut qu'améliorer la maintenace d'une application.

En effet, lorsqu'un programmeur réalise un logiciel, il le développe et le met au point pas à pas. Ainsi le système de vérification mémorise des assertions statiques qui seront réutilisable tout au long de la vie du logiciel.

11.4 Une trace interactive

La réalisation d'une trace *statique* requiert des spécifications formelles des messages dont elle est faite [Byrd-80, Boizumault-84]. Dans certains cas, la disposition de facilités de traces partielles étendent les possibilités, notamment le Spy ou le Zooming [Eisenstadt-84, Ducassé-88]. Nous avons vu les limites de cette approche pour fournir une mise au point adaptée à des utilisateurs non avertis.

Une technique intéressante de trace graphique [Eisenstadt-Brayshow-87a] consiste à représenter graphiquement une partie de l'arbre de résolution. Cette trace visualise : les clauses activées, les instances des arguments, les ancêtres du but courant, les noms des variables... Cependant l'utilisateur doit éviter d'afficher un trop grand nombre d'informations afin d'obtenir une mise au point efficace. Son attention est centrée sur cette activité au lieu de l'être uniquement sur l'analyse des symptômes erreurs. Pour pallier à cetl inconvénient, une extension permet d'obtenir une vue superficielle de l'arbre de résolution.

La trace que nous présentons répond au maximum des objectifs pour atteindre une trace intelligente, utilisable intuitivement. Ces objectifs permettent de disposer d'un maximum d'informations sur l'écran. Des choix ont cependant été faits pour limiter cette information à l'écran :

- la clause en cours d'évaluation,
- les noms des variables libres dans la clause courante,
- l'instances des arguments d'appel ou de réussite d'un sous-but,
- les buts ancêtres,
- le but courant.

L'interface peut les fournir de la façon suivant :

- La fenêtre SW visualise la suite des buts ancêtres avec une référence sur la clause active du prédicat. Sa manipulation est effectuée par les prédicats de gestion de la pile des buts.
- La fenêtre PW contient les informations sur la clause en cours d'évaluation pour l'appel ou la réussite d'un de ses sous-buts en prémisses. Sa manipulation est réalisée par le prédicat *"écrire_état/4"*. Ce prédicat obtient une réponse de l'utilisateur sur l'action à effectuer.
- La fenêtre SO contient les messages particuliers de la résolution : l'absence d'unification, les entrées/sorties...

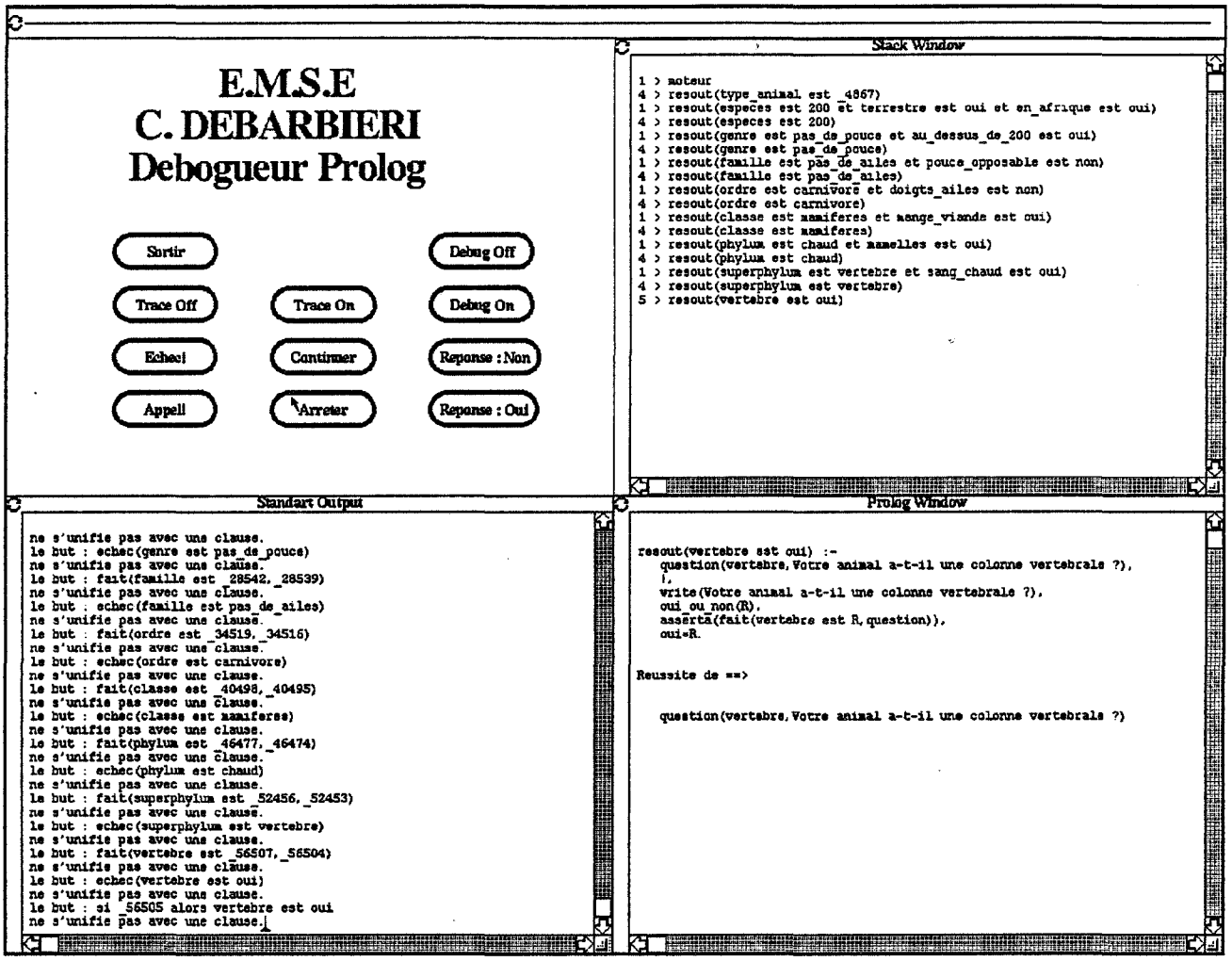
Un dialogue entre l'utilisateur et le système de trace permet d'obtenir un parcours partiel de la résolution. En fait, c'est uniquement l'interaction qui est partielle, car le méta-interprète réalise la résolution du but mais ne fournit aucun message. Ce dialogue est obtenu par l'intermédiaire des boutons :

Appel	effectue une résolution sans trace des sous-buts, sur l'appel d'un but. Ce bouton est inactif dans les autres cas.
Échec	Simule un échec de l'appel d'un but. Ce bouton ne doit être utilisé que lorsque l'on désire simuler une échec du but courant, et que celui-ci est un prédicat méta-logique.
Continuer	Permet de continuer la résolution avec la trace.
Arrêter	Permet d'arrêter la résolution d'un but.
Sortir	Permet d'arrêter une session de mise au point et de se déconnecter du serveur NeWS. Ce bouton est aussi actif au "Top Level". Il simule le prédicat de service "*halt/0*".

Les boutons "Trace On/Trace Off" active ou désactive la trace lors d'une interaction entre l'interface et le système de mise au point. Cependant, lors de la trace d'un but, nous pouvons activer la détection d'erreurs en cliquant sur le bouton "Debug On". Le mode mise au point étant prioritaire sur la trace, sa désactivation n'est possible qu'au "Top Level" de Prolog.

On peut ainsi débuter la résolution d'un but avec une trace, puis lorsqu'on a trouvé le sous-but à partir duquel on souhaite effectuer une détection d'erreurs, activer la détection d'erreurs par une pression du bouton "Debug On".

Voici l'écran d'une session de trace sur le programme présenté en annexe A, dont le but initial est "moteur" :



Pour bien comprendre l'intérêt d'une telle trace, il faut la comparer à une trace classique, par exemple celle de Eisenstadt. Lorsque l'utilisateur désire analyser une trace classique, il doit construire certaines informations concernant la résolution à partir des messages de la trace. Ces informations permettent de comparer les sémantiques actuelles et arrendue du programme. L'utilisation de notre interface simplifie l'analyse du programmeur. En effet, l'utilisateur de la trace ne doit porter son attention que sur l'analyse de la sémantique qu'il obtient par les messages donnés par l'interface. L'utilisateur focalise son attention sur la sémantique de son programme.

Comme nous le voyons sur cet exemple, l'utilisateur dispose d'un système de trace lui permettant d'effectuer une mise au point sur des programmes extra-logiques.

11.5 Une session de mise au point

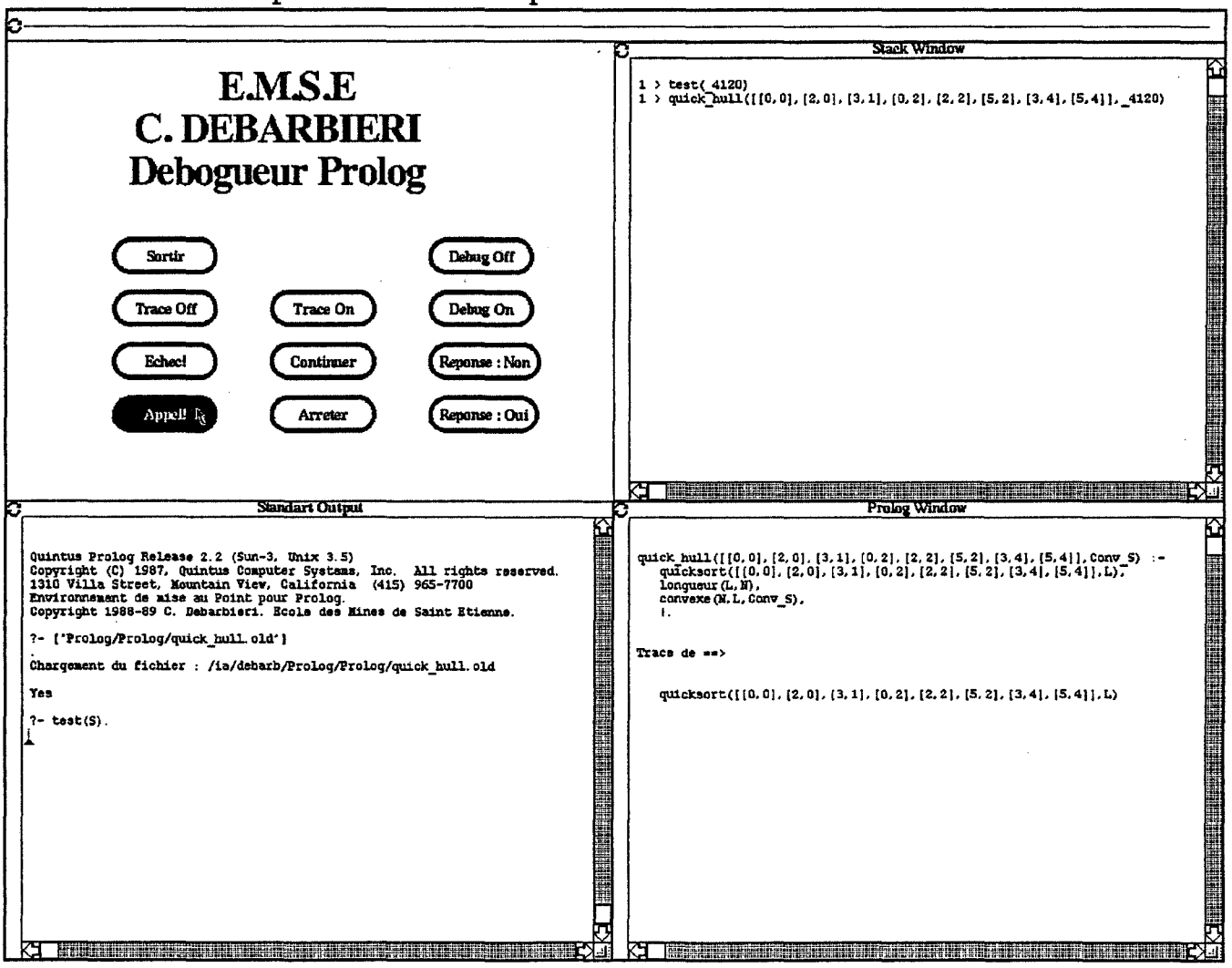
Pour illustrer l'intérêt d'une telle interface de mise au point, nous devons considérer un programme incorrect, mais dont le programmeur sait que certaines parties sont correctes.

Prenons un programme de la géométrie algorithmique, la détermination d'une enveloppe convexe d'une ensemble de points du plan. Nou représentons tout point "X, Y" du plan par la liste à deux éléments : "[X, Y]". Le programme complet de cette application est donné en annexe 3. Ce programme est incorrect car la solution obtenue estfausse.

En revanche, certaines parties de ce programme sont correctes. C'est le cas par exemple du calcul de la liste des points du polygone trié par ordre croissant sur les X. Si nous effectuons une détection d'erreurs sur le but initial "*test(C)*", le système procède à une vérification de la résolution des buts "*quicksort(P, C)*" et "*longueur(C, N)*". Ces vérifications ne sont pas forcément souhaitables car l'utilisateur sait qu'elles sont correctes. Mais il désire réaliser la détection d'erreurs sur le but "*convexe(N, C, Convexe)*" qui est incorrect.

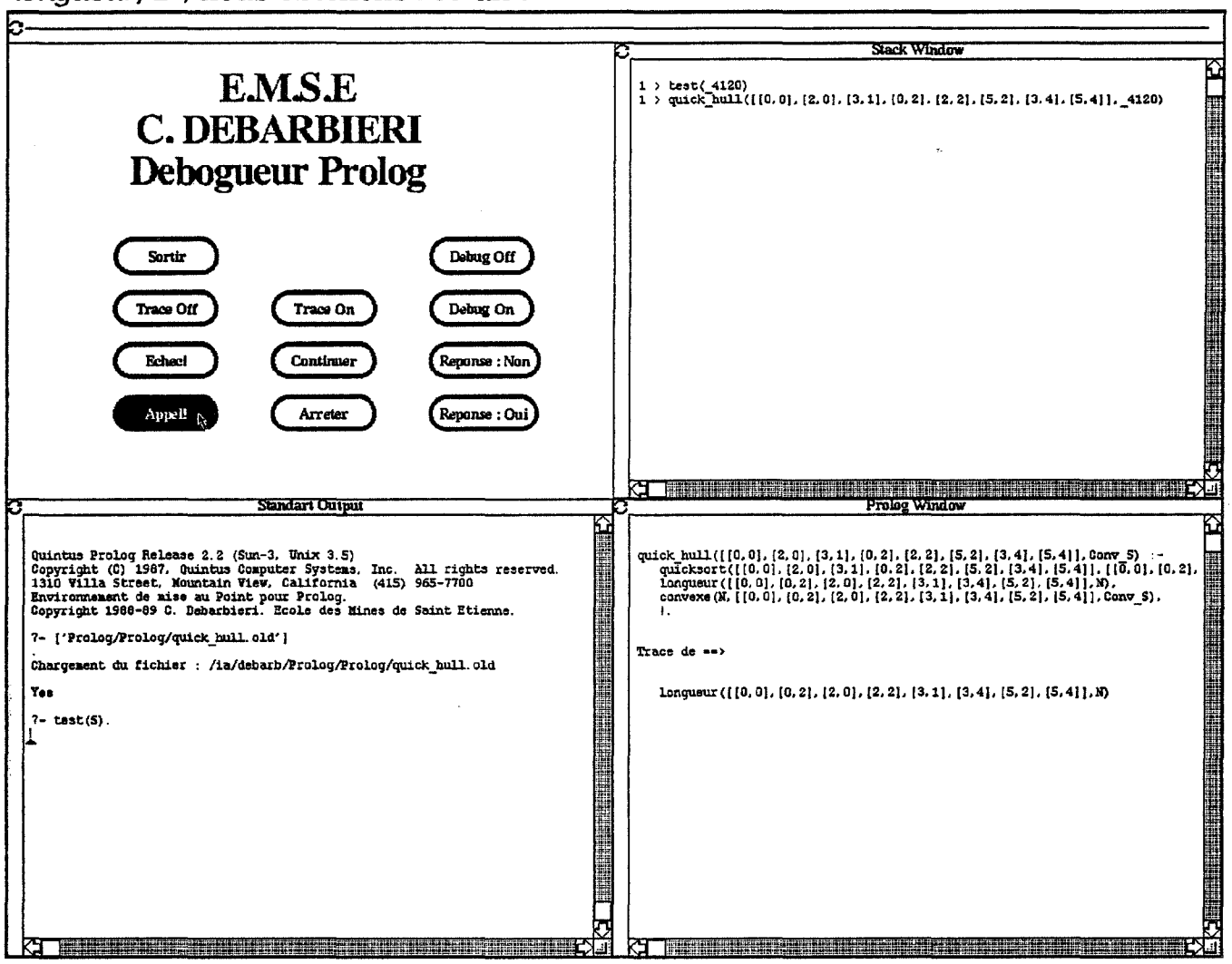
L'utilisateur peut réaliser la vérification de la résolution de ce but, mais il doit au préalable *copier* les arguments de celui-ci. Par exemple, à partir d'une trace en largeur d'abord l'utilisateur peut connaître les arguments de ce but dans la résolution du but "*test(C)*". Cette tache est assez pénible (très grande liste de points). En revanche, nous pouvons éffectuer une trace du but initial, appeller chaque but correct, jusqu'à obtenir le sous-but à vérifier.

Cette mise au point est illustrée par la session :



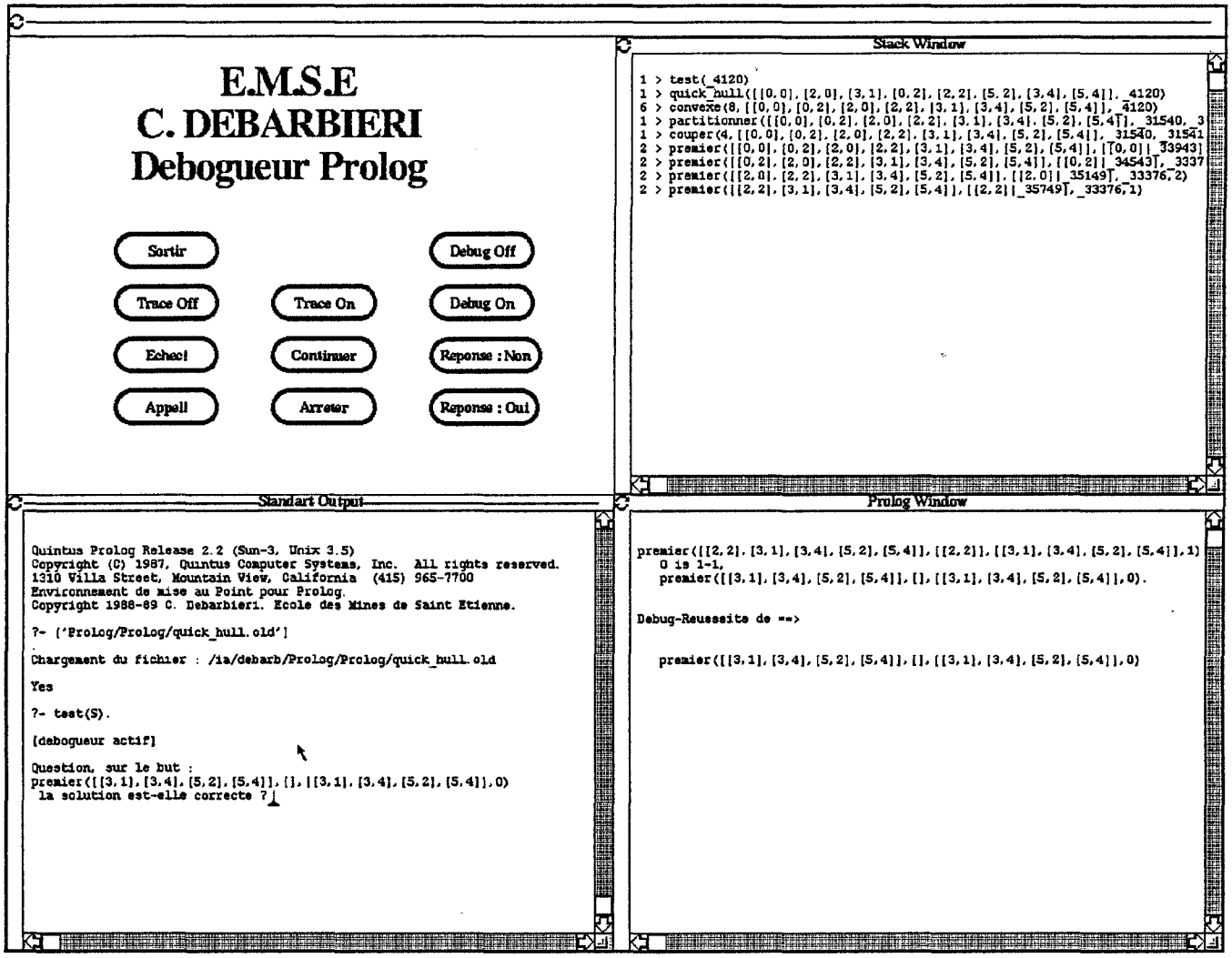
Cet écran présente les diverses informations de la trace du but initial. Nous réalisons une trace en profondeur d'abord du but initial "*test(C)*". Nous somme arrêté sur l'appel du but "*quick_sort/2*". Une action sur le bouton "Appel" évite une trace de toute la résolution de ce sous-but, que l'utilisateur sait correcte. Cet appel limite l'interaction sur toute sa résolution.

Le système signale la réussite du sous-but, il passe à la trace du sous-but suivant "*longueur/2*", nous obtenons l'écran :



De la même façon que pour le but précédent, l'utilisateur n'effectue pas la trace et le signale par l'action sur le bouton "Appel". Le système passe à la trace du but suivant, c'est-à-dire le but qui nous intéresse. Pour signaler au système que nous désirons effectuer une vérification de la résolution de ce but, lors de l'interaction de la trace, nous activons le mode détection d'erreurs par une action sur le bouton "Debug On".

Le système signale l'activation du mode de détection d'erreurs, et nous obtenons l'écran :



11.6 Conclusion sur l'application

Nous venons de montrer sur des exemples simples, la facilité d'utilisation des méthodes de mise au point en Prolog fondées sur la vérification d'assertions. Dans le cadre du développement de logiciels en Prolog, les assertions permettent de vérifier à tout moment la cohérence d'une résolution. Cependant, comme nous l'avons montré, utiliser une telle technique de mise au point pour des programmes extra-logiques est impossible. Ainsi, la disposition d'un outil de trace est important pour la mise en oeuvre du génie logiciel en Prolog et plus fondamentalement il faut disposer des deux méthodes de mise au point, comme nous le montre le dernier exemple.

La réutilisabilité des assertions, obtenues sur des programmes corrects, pour la mise au point d'autres programmes est un apport important dans la vie d'applications au sein d'une équipe de développement. En effet, la plupart des logiciels réalisés en Prolog réutilisent certaines composantes logicielles d'autres applications.

Partie 5

Conclusion

Page blanche

Dans cette thèse ont été présentées une analyse critique de l'état de l'art et la réalisation d'un système de mise au point intelligent pour Prolog utilisant la sémantique attendue du programme afin de détecter des erreurs de programmation. De plus, l'utilisation d'un interface multi-fenêtres nous a permis de réaliser un environnement de développement efficace. Les divers concepts mis en oeuvre dans ce prototype valide la faisabilité d'un produit industriel de grande diffusion constituant un système de vérification et de validation des applications Prolog.

L'originalité du travail que nous venons de décrire, repose dans l'approche nouvelle de la mise au point en Prolog et dans l'utilisation de techniques propes à l'Intelligence Artificielle pour réaliser le système de détection d'erreurs. Ces notions sont la méta-programmation, l'évaluation partielle, l'apprentissage, la preuve de programmes, l'utilisation des méta-connaissances... De plus, l'approche unifiée utilisée pour la mise au point nous permet de réaliser une détection d'erreurs efficace sur des programmes utilisant des négations et des coupures.

Nous avons réalisé un noyau unique sur lequel peut s'articuler toute sorte de raffinements de la détection d'erreurs. Par exemple, on peut effectuer une analyse spécifique d'un échec incorrect pour détecter une coupure rouge. Ou bien, par l'intégration d'un traitement de la négation on peut analyser une utilisation incorrecte de celle-ci, ou encore intégrer des traitements sur les ensembles de solutions pour analyser le nombre de solutions d'un but.

Bien que nous ayons totalement atteint les différents objectifs que nous nous étions fixés, il nous semble que certains thèmes de recherche peuvent êtres envisagés afin de généraliser l'approche que nous avons suivie.

La première évolution que nous pouvons envisager est la réalisation d'un outil de vérification et de validation comparable à ceux disponible dans les ateliers de génie logiciel. La plupart des réalisations utilisant la programmation logique sont des systèmes à base de connaissances, notés SBC. Lorsque l'on développe industriellement un SBC, on est confronté au problème de la vérification et la validation de celui-ci. Notre approche consiste à effectuer une vérification de la sémantique attendue du programme, en l'occurrence c'est un SBC. Nous possédons ainsi d'un système de validation, fondé sur la connaissance, contrôlant la sémantique attendue du programme.

Afin de disposer d'un système de vérification et de validation utilisable industriellement, il est important de rajouter des traitements spécifiques à l'analyse statique des programmes, comme l'étude du graphe de dépendance ou de mesure de la "qualité" d'un programme (taille d'une clause, nombre de clauses d'un prédicat...). Dans ces conditions, le chef de projet d'une application dispose d'un outil pour mesurer la qualité du produit et ses propriétés de réutilisabilité et de maintenabilité, qui sont les principaux apports de cette thèse dans le domaine de la validation.

Mais aussi cette approche de la validation de logiciels fondée sur les connaissances peut être utilisée pour la validation des systèmes experts. Cependant, la diversité des moteurs d'inférences fournis dans les générateurs de systèmes experts complique la méthodologie de vérification de l'arbre de déduction et l'approche que nous devons avoir. Les cas les plus simples sont les moteurs en chaînage arrière, le parcours de l'arbre de

déduction est similaire à l'algorithme de résolution de Prolog et une vérification de cette arbre de déduction à partir de la sémantique attendue peut être réalisée. En revanche, lorsque le moteur d'inférence utilise le chaînage avant dans ses déductions, comparable aux techniques de recherche en largeur d'abord, on peut envisager de construire l'arbre de déduction dans son ensemble et le vérifier en chaînage arrière à partir de la sémantique attendue.

Ainsi, une étude de la faisabilité de telles méthodes de vérification et de validation dynamiques est un axe de recherche intéressant afin de fournir aux utilisateurs de générateurs de systèmes experts des outils de génie logiciel.

La seconde évolution concerne l'étude de la faisabilité d'une mise au point déclarative pour des langages Prolog avec contraintes.

En effet, l'évolution récente dans ce domaine, avec l'apparition des premièrs interprètes Prolog avec contraintes, nous pose un défi : "Est-on capable de programmer efficacement avec les Prolog à contraintes". On peut aussi se demander quelle est la relation entre le système de contraintes et la sémantique attendue du programme, suite à l'ajout aux réponses calculées d'un système minimal de contraintes obtenues lors de la résolution.

Cette extension peut poser des ambiguïtés sur l'interprétation d'un résultat : "une solution particulière, selon l'interprétation attendue, satisfait-elle le système général attendu ?".

De même, lors d'un échec d'un but, la prise en compte des contraintes dans l'algorithme de résolution rend l'analyse de l' échec très difficile lorsque le système de contraintes n'admet pas de solution. Est-ce le système de contraintes qui est incorrect ou, est-ce une mauvaise définition d'un prédicat ? Ou bien, l'analyse d'une solution fausse peut être réduite à rien lorsque cette solution n'est calculée qu'à partir du système complet de contraintes.

D'autres questions se posent pour réaliser un système déclaratif : "doit-on vérifier qu'une solution attendue satisfait le système minimal fourni ?", "doit-on donner le système minimal ?", "comment informer le programmeur sur l'état du solveur lors de la production d'une solution d'un but intermédiaire ?". Aucune réponse précise ne peut être donnée. Mais une étude de la faisabilité d'un système "déclaratif" de mise au point est une voie de recherche intéressante pour l'aide au développement d'applications avec des langages aussi puissants.

Une autre évolution que nous pouvons envisagée est l'étude des méthodes de mise au point de programmes extra-logiques. Ce problème programmes repose essentiellement sur l'absence de fondements théoriques. Une technique que l'on peut mettre en oeuvre est celle de spécification formelles du fonctionnement temporel de ces programmes. Il faut donc mettre au point un modèle dynamique de ces programmes.

Une technique envisageable est d'utiliser une modélisation d'un système parfait, Prolog pur avec la coupure et la négation, auquel on ajoute des perturbations aléatoires correspondant aux effets des prédicats extra-logiques.

Un fois qu'une telle modélisation est effectuée, nous pouvons réaliser une étude de méthode de mise au point pseudo-déclaratives.

Une autre technique, que nous pouvons utilisée dans le système d'assertions, est l'interconnexion avec une base de données déductives de la sémantique attendue du programme. Ainsi, une convention sur le nommage des symboles utilisés permet à un groupe de programmeurs de partager des informations utiles, assertions statiques et assertions évaluables, introduites par l'un des membres du groupe. Cette technique peut être utilisée pour stocker les informations nécessaires pour la maintenance logicielle des applications produites par une autre équipe de développement.

Annexe 1

Bibliographie

[Av Ron-84]
E. Av Ron. *Top down diagnosis of Prolog programs.* Msc Thesis, Weizmann Institute. ISRAEL 1984.

[Barklund-87]
J. Barklund. *Efficient Interpretation of Prolog Programs.* ACM-87.

[Boizumault-84]
P. Boizumault. *Un modèle de trace pour Prolog.* Actes du Séminaire sur la Programmation Logique. CNET, Lannion, mai 1985.

[Boizumault-88]
P. Boizumault. *Prolog, l'implémentation.* Etudes et recherches en informatique. Ed Massonm 1988.

[Boolos-Jeffrey-80]
G.S. Boolos and R.C. Jeffrey. Computability and logic. Camb university Press. 1980, 2 ed.

[Bowen-85]
K. Bowen. *Meta-level programming and Knowledge Representation.* New Generation Computing, 3 Ohmsha Ltd and Springer-Verlag. 1985.

[Bowen-Kowalski-82]
K. Bowen and R. Kowalski. *Amalgamating Language and metalanguage in logic programming.* In Logic Programming, (Clark & Tarnlund Ed) Academic Press, 1982.

[Bowen-Weinsberg-85]
K. Bowen & T. Weinsberg. *A Meta-level extention of Prolog.* Proceedings of the 1985 Symposium on logic programming, (Cohen & Conery Ed.) IEEE Comp society Press, Washington 1985.

[Brough-Hogger-86]
D.R Brough and C.J Hogger. *The Treatment of loops in Logic Programming.* Doc Rep. Dep of computing and control. Imperial college London, 1986.

[Bruynooghe-Pereira-84]
M. Bruynooghe and L.M. Pereira. *Deduction revision trhough intelligent backtracking.* In issues in Prolog implementation. J Campbell, Ellis Howord ltd. 1984.

[Bundy-Welham-81]
A Bundy & B Welham. *Using Meta-level Inference for selective Application of multiple Rewrite-Rules.* In Algebraic Manipulation, Artificial Intelligence, V10-1981.

[Byrd-80]
L. Byrd. *Understanding the control flow of prolog programs.* In S-A Tarnlund (Ed.). Proceedings of the Logic Programming Workshop, Debrecen, Hungary 1980.

[CProlog-1.5]
C-Prolog. Manuel de référence. version 1.5 Juin 1986. Ed F. Pereira, SRI International, Menlo Park, California.

[Church-56]
A. Church. *Introduction to mathematical logic.* Vol 1, Princeton University Press. 1956.

[Clark-78]
K. L. Clark. *Negation as failure*. Logic and databases, Gallaire & Minker (Eds). Plenum, New York, 1978, pp293-322.

[Clark-Cabe-82]
K. L. Clark & F. G. Mc Cabe. *PROLOG : a language for implenting expert systems*. In machine intelligence 10, Ellis Howord 1982.

[Colmerauer-73]
A. Colmerauer, H. Kanoui, P. Roussel, R. Pasero. *Un système de communication Homme-Machine en Français*, Groupe de recherche en intelligence artificielle, Univérsité d'Aix-Marseille, 1973.

[Covington-85a]
Michael A. Covington. *Eliminating unwanted loops in Prolog*. SIGPLAN Notices, V20 #1, Janvier 1985.

[Covington-85b]
Michael A. Covington. *A further note on looping in Prolog*. SIGPLAN Notices, V20 #8, Aout 1985.

[Davis-Putnam-60]
M. Davis and H.Putnam. *A computing procedure for quantification theory*. JACM Mars 1960 no 7.

[Debarbieri-89]
Christian Debarbieri. *Détection de boucles en Prolog*. Rapport de recherche 1989, Ecole des mines de saint-étienne.

[Debarbieri-89]
Christian Debarbieri. *A meta-logical diagnoser*. Rapport de recherche 1989, Ecole des mines de saint-étienne.

[Debarbieri-89]
Christian Debarbieri. *Mise au point en Prolog, Une vérification de la résolution*. Congrès AFCET-RFIA'89, Paris 1989.

[Delahaye-86]
J.P. Delahaye. *Outils logiques pour l'intelligence artificielle*. Editions Eyrolles. 1986.

[Deransart&all-85]
P. Deransart, G.Richard, C. Moss. *Spécifications formelles de Prolog Standard*. Programmation Logique, actes du séminaire 1985, CNET.

[Deransart-Ferrand-87]
Pierre Deransart & Gérard Ferrand. *An operational formal definition of Prolog*. Rapport de Recherche No 763, INRIA-Roquencourt, Décembre 1987.

[Deransart-Ferrand-89a]
Pierre Deransart & Gérard Ferrand. *A methodological view of logic programming with negation*. Rapport de Recherche No 1011, INRIA-Roquencourt, Avril 1989.

[Deransart-Ferrand-89b]
Pierre Deransart & Gérard Ferrand. *Proofs Methods and Declarative Diagnosis in Logic Programming*. ICLP'89, June 19-23 Lisbonne.

[Dershowitz-Lee-87]
N. Dershowitz & Y.J. Lee. *Deductive Debugging*. Proceedings 1987, Symposium On Logic Programming. The Computer Society of the IEEE. 1987 San Francisco, California.

[Drabent-all-88]
Wlodek Drabent, Simin Nadjm-Tehrani, Jan Maluszynski. The use of Assertions in Algorithmic Debugging. Proceedings of the internationnal conference on Fifth generation computer systems, 1988. Ed ICOT.

[Ducassé-85]
Mireille Ducassé. *Analisys of some Prolog Debugging Tools*. Technical Report LP-5, ECRC, Mai 1985

[Ducassé-86]
Mireille Ducassé. *A Sophisticated Tracing Tool for Prolog*. Programmation Logique, actes du séminaire 1986. TREGASTEL, 21-23 mai 1986.

[Ducassé-Emde-87]
Mireille Ducassé et Anna Maria Emde. *State of the Art in automated program debugging*.Technical Report LP-25, ECRC, Sept 1987.

[Ducassé-88]
Mireille Ducassé. *OPIUM$^+$, a meta debugger for Prolog*. ECAI'88.

[Eisenstadt-85b]
M. Eisenstadt. *Retrospective Zooming : A Knowledge based tracing and debugging methodology for Logic Programming*. IJCAI 85, Los Angeles, 717-719.

[Eisenstadt-85a]
M. Eisenstadt. *A powerful prolog trace package*. In Advances in Artificial Intelligence, T. O'Shea (Ed.). Elsevier Science Publishers B.V. (North Holland) ECCAI, 1985.

[Eisenstadt-Brayshaw-87a]
M. Eisenstadt and M. Brayshaw. *Graphical Debugging with the transparent Prolog Machine (TPM)*. Proceedings of the tenth international joint conference on artificial intelligence. IJCAI 87, Milan Aout 1987.

[Eisenstadt-Brayshaw-87b]
M. Eisenstadt and M. Brayshaw. *AORTA Diagrams as an aid to visualising the execution of Prolog programs*. Technical report No 29 Jan 1986. Human Cognition Research Laboratory, Milton Keynes MK7 6AA, U.K.

[Eisenstadt-Brayshaw-88]
M. Eisenstadt and M. Brayshaw. *Adding data and procedure abstraction to the transparent prolog machine(TPM)*. Logic Programming, Proceedings of the fifth international conference and symposium, Seattle. Edited by Kowalsky & Bowen. MIT press 1988.

[Emden-Kowalski-76]
M.H Van Emden and R. Kowalski. *The semantics of predicate logic as a programming language*. J-ACM, 23, 4. oct 1976.

[Emden-82]
M Van Emden. *Red cut and green cuts*. Logic Programming newsletter, Decembre 1982.

[Emden-85]
Alan Van Emden. *Waren doctrine of the Slash*. Logic Programming newsletter, Decembre 1985.

[Ferrand-88]
Gérard Ferrand. *Error diagnosis in Logic Programming, An adaptation of E.Y Shapiro's method*. Journal og Logic Programming 1988.

[Gelder-86]
Allen Van Gelder. *Negation as failure Using Tight Derivations for general logic Programs*. 1986-IEEE.

[Gelder-87]
Allen Van Gelder. *Efficient Loop detection in Prolog using the "Tortoise-and-Hare" technique*. The journal of Logic Programming, 1987-4.

[Gelfond-86]
M Gelfond & H Przymusinska. *Negation as failure·: careful closure procedure*. Journal of Artificial Intelligence. Vol 30, 1986.

[Geyres-89]
Stéphane Geyres. Une approche industrielle de la validation et de la vérification des systèmes à base de connaissances. Génie Logiciel et Système Expert, n 16. Ed EC2, septembre 1989.

[Girardot-89]
Jean-Jacques Girardot. APL-90, Thèse d'état, à paraitre 1989. Ecole des mines de Saint Etienne.

[Godel-67]
Godel. *Researches in the Theory of demonstration*, in "From frege to Gödel : A source book in mathematical logic", Harvard University Press, Cambridge, Mass, 1967

[Gold-65]
E. Mark Gold. *Limiting Recursion*. The journal of symbolic Logic. 1965.

[Harmelen-Bundy-88]
Franck Van Harmelen and Alan Bundy. *Explanation-Based Generalisation = Partial Evaluation*. Artificial Intelligence, - vol 36, 1988 401-412.

[Herbrand-31]
J. Herbrand. *Sur le problème fondamental de la logique mathématique*. Comptes rendues des séances de la société des sciences et des lettres de varsovie. Cl III, vol 24, 1931.

[Herbrand-67]
J. Herbrand. *Researches in the Theory of demonstration*, in "From frege to Gödel : A source book in mathematical logic", Harvard University Press, Cambridge, Mass, 1967

[Hofstadter-85]
Douglas Hofstadter. *Gödel Escher Bach, les brins d'une guirlande éternelle*. InterEdition, version française, 1985.

[Huntbach-87]
M.H Huntbach. *Algorithmic Parlog debugging*. Proceedings 1987, Symposium On Logic Programming. The Computer Society of the IEEE. 1987 San Francisco, California.

[Kowalski-74]
R. Kowalski. *Predicate logic as a programming language*. Information Processing 74, Stokholm, North Holland, New York, 1974.

[Kowalski-79]
R. Kowalski. "*Algorithm = Logic + Control*", Comm ACM, 22 7, Jul 1979.

[Kowalski-79b]
R. Kowalski. *Logic for problem solving*. North Holland, New York, 1979.

[Kunen-87]
Kenneth Kunen. *Negation in logic Programming*. The journal of Logic Programming, vol 4, n4. 1987.

[Lepape-Sellami-87]
Brice Lepape et Mokhtar Sellami. *Banc d'essai pour les principales versions de Prolog. Réalisation et mesure de performances*. Séminaire de Programmation Logique. Lanion 1987.

[LLoyd-86]
J. W. LLoyd. *Declarative Error Diagnosis*. Technical Report 86/3. Department of computer science, University of Melbourne, Prakville, Victoria 3052 Australia.

[LLoyd-87]
LLoyd J. W. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984 et révisé en 1987. Traduction française Ed Eyrolles.

[LOO-89]
G. Masini et All. *Les langages à objets*. Collection iia, ed InterEdition, 1989.

[Looi-Ross-87a]
C.K. Looi & P. Ross. *Automatic Program Debugging for a prolog intelligent tutoring system*. DAI RESEARCH PAPER No 307 University of Edinburgh.

[Looi-Ross-87b]
C.K. Looi & P. Ross. *Debugging Prolog programs in an intelligent tutoring system.* DAI RESEARCH PAPER No 308 University of Edinburgh.

[Loveland-78]
D.W. Loveland. *Automated theorem proving : logical basis.* Fundamental studies in computer science. Vol 6 North Holland publishing company 1978.

[Mellish-81]
C.S Mellish. *The automatic generation of mode declaration for Prolog Programs.* DAI RESEARCH PAPER No 163 University of Edinburgh, 1981.

[Minker-82]
J Minker. *On indefinite data-bases and the closed world assumption* Proc sixth conference on automated deduction. lecture note in computer science 138. Springer Berlin 1982.

[Meyer-88]
Bertrand Meyer. *Object Oriented Software Construction.* Ed Prentice Hall Internationnal, 1988.

[Moss-86]
C. Moss. *Cut and Paste - defining the impure primitives of prolog.* Third international conference on logic-programming, Jul-86.

[Murray-86]
W.R Murray. *Automatic Program Debugging for intelligent Tutoring System.* AI TR86-27, June 1986. Artificial Intelligence Laboratory, The University of Texas at Austin.

[Naish-85]
Lee Naish. *All solutions predicates in Prolog.* Symposium on logic Programming, july 1985. Boston, Massachusetts.

[Naish-85]
Lee Naish. *Negation and Control in Prolog.* Technical report 1985-12 department of Computer Science University of Melbourne.

[NeWS-88]
NeWS 1.1 Manual. Sun microsystems. Jan 1988.

[Nute-85]
Donald Nute. *A programming solution to certain problems with loops in Prolog.* SIGPLAN Notices, V20 #8, August 1985.

[O'keefe-85]
Richard O'Keefe. *On the traitment of cuts in prolog source-level tools.* Symposium on logic Programming, july 1985. Boston, Massachusetts.

[Pabion-76]
J.F. Pabion. Logique mathématique. Coll méthode, édition Hermann, 1976.

[Pelhat-87a]
Sophie Pelhat. *Analyse des inférences récursives en Prolog : système d'aide à la détection et au contrôle de boucles.* Thèse de troisième cycle en Informatique, Université de Paris-Sud, Centre d'Orsay. Octobre 1987.

[Pelhat-87b]
Sophie Pelhat. *Etude des inférences récursives en Prolog.* Reconnaissance de formes et Intelligence Artificielle. (Ed Dunod informatique). 6ème congrès, Antibes, Novembre 1987.

[Pereira-85]
L.M. Pereira. *Rational debugging in logic programming.* Departamento de informatica. Universidade Nova de Lisboa. 2825 Monte da Caparica. Draft 1985.

[Pereira-88]
L.M. Pereira & M. Calejo. *Framework for Prolog Debugging*. Logic Programming, Proceedings of the fifth international conference and symposium, Seattle. Edited by Kowalsky & Bowen. MIT press 1988.

[Plaisted-84]
D A Plaisted. *An efficient bug location algorithm*. Second international Conference on logic programming. Uppsala. 1984.

[Plummer-87]
D Plummer. *CODA : An extended Debugger for Prolog*. Artificial Intelligence Laboratory. The University of Texas at Austin. AI TR87-54, Avril 1987.

[Plummer-88]
D Plummer. *CODA : An extended Debugger for Prolog*. Logic Programming, Proceedings of the fifth international conference and symposium, Seattle. Edited by Kowalsky & Bowen. MIT press 1988.

[Poole-Goebel-85]
D Poole & R Goebel. *On eliminating loops in Prolog*. SIGPLAN Notices, V20 #8, August 1985.

[PostScript-85]
Adobe Systems. *PostScript language reference manual*. Addison Wesley, July 1985.

[Reiter-78]
R Reiter. *On closed world date bases* in Logic and Data Bases, Gallaire H & Minker J. Plenum Press, New York 1978.

[Robinson-63]
J.A. Robison. *Theorem proving on the computer*. JACM no 10, Av 1963.

[Robinson-65]
J. Robinson. *A machine oriented logic based on the resolution principle*. J ACM 12, 1. Jan 1965.

[Safra-Shapiro-86]
S. Safra and E. Shapiro. *Meta-Interpreters for Real*. In Information Processing 86, H.J. Kugler (Ed.) Elsevier Science Publishers B.V. (North Holland). IFIP 86.

[Sbpl-86]
The SB-Prolog System, version 2.2. Manuel de référence. Department of Computer Science. University of Arizona, Tucson, AZ 85721. March 1987.

[Shapiro-81]
E.Y Shapiro. *An algorithm that infer theories from facts*. Proc seventh IJCAI 1981, Vancover.

[Shapiro-82]
E.Y Shapiro. *Algorithmic Program Debugging*. ACM 1982.

[Shapiro-83a]
E.Y Shapiro. *Algorithmic Program Debugging*. The MIT Press Cambridge, Massachusetts 1983.

[Shapiro-83b]
E.Y Shapiro. *Logic programs with uncertainties : A tool for implementing Rules-based systems*. Proc IJCAI 83.

[Shapiro-Sterling-86]
L. Sterling and E. Shapiro. *The Art of Prolog. Advanced programming techniques*. The MIT Press, Cambridge, Massachusetts, 1986.

[Shepherdson-84]
J.C. Shepherdson. *Negation as Failure*. J. Logic Programming 1984, 1, 51-79.

[Shepherdson-85]
J.C. Shepherdson. *Negation as Failure II*. J. Logic Programming 1985, 3, 185-202.

[Smalltalk-80]
Adelle Goldberg and David Robson. *Smalltalk 80: The language and its implemetation.* Addison Wesley May 1980.

[Sterling-84]
L. Sterling. *Expert System = Knowledge + Meta-Interpreters.* Technical Report No CS84-17. Weizmann institute of Science, 76100 Rehovot, Israel.

[Sterling-86]
L. Sterling. *Incremental Flavor-mixing of Meta-interpreters for Expert system construction.* Proceedings of Symposium on logic programming, Salk lake city, Utah, 20-27, 1986.

[Sterling-Lakhotia-87]
L. Sterling & A. Lakhotia. *Composing Prolog Meta-interpreters.* Case Western Reserve University, Cleveland Ohio 44106 USA. 1987.

[SunView-85]
Programmer's reference manual for SunVindows. Sun microsystems, April 1985.

[Takeuchi-Furukawa-86]
A. Takeuchi and F. Furukawa. *Partial evaluation of Prolog programs and its application to meta-programming.* In Information Processing 86, H.J. Kugler (Ed.) Elsevier Science Publishers B.V. (North Holland). IFIP 86.

[Thayse-89]
A. Thayse . Approche logique de l'Intelligence Artificielle, 1 de la logique à la programmation logique. DUNOD Informatique, 1988.

[Venken-85]
R. Venken. *A Prolog Meta-interpreter for partial evaluation, and its application to source to source transformation.* In Advances in Artificial Intelligence, T. O'Shea (Ed.). Elsevier Science Publishers B.V. (North Holland) ECCAI, 1985.

[Vial-88]
P. Vial. *Un système expert pour installer UNIX.* Thèse de troisième cycle. Ecole des Mines de Saint Etienne, Decembre 1988.

[Warren-77]
D.H. Warren. *Implementing Prolog : compiling predicates logic programs.* D.A.I research report n 39/40, university of Edinbough, 1977.

[Warren-83]
D.H. Warren. *An abstract Prolog instruction set.* Technical note 309, SRI international, Menlo Park, 1983

[X11R3]
X Window system 11 version 3. *Reference manual*, MIT, 1988.

[Xilog]
Xilog V2, manuel de référence. BULL Sciamma, Act Informatique.

[Zellweger-84]
P.T. Zellweger. *Interactive Source-Level Debugging of Optimized Program.* Xerox Corporation, Palo Alto Center 3333 Coyote Hill Road, Palo Halto, California 94304.

Annexe 2

Un Système Expert

```
:- op(145,xfx,alors).
:- op(135,xfy,et).
:- op(150,fx,si)
:- op(40,xfx,est).
:- dynamic fait/2, echec/1.

oui_non(R) :-
    repeat,
    read(R),
    (R =oui;R=non),
    !.

/* clause 1 */
resout( F1 et F2 ) :-
    !,
    resout(F1),
    resout(F2).

/* clause 2 */
resout(F est V) :-
    fait(F est W,_),
    !,
    V = W.

/* clause 3 */
resout(F est V) :-
    echec(F est V),
    !,
    fail.

/* clause 4 */
resout(F est V) :-
    si Premisse alors F est V,
    resout(Premisse),
    asserta(fait(F,regle(si Premisse alors F est V))),
    !.

/* clause 5 */
resout(F est V) :-
    question(F,Q),
    !,
```

```
        write(T),
        oui_non(R),
        asserta(fait(F est R,question)).
        V = R.

/* clause 6 */
resout(F est V) :-
        asserta(echec(F est V)),
        fail.

% le moteur
moteur :-
        abolish(fait,2),
        abolish(echec,1),
        resout(type_animal est V),
        !,
        write(type_animal est V),
        nl,
        fail.

si especes est 200
        et terrestre est oui
        et en_afrique est oui
        alors type_animal est lion.

si genre est pas_de_pouce
        et au_dessus_de_200 est oui
        alors especes est 200.

si famille est pas_de_ailes
        et pouce_opposable est non
        alors genre est pas_de_pouce.
si ordre est carnivore
        et doigts_ailes est non
        alors famille est pas_de_ailes.

si classe est mamiferes
        et mange_viande est oui
        alors ordre est carnivore.

si phylum est chaud et mamelles est oui
        alors classe est mamiferes.

si superphylum est vertebre
        et sang_chaud est oui
        alors phylum est chaud.

si vertebre est oui
        alors superphylum est vertebre.
```

```
question(vertebre,
    'Votre animal a-t-il une colonne vertebrale ?').

question(sang_chaud,
    'Votre animal a-t-il le sang chaud ?').

question(mamelles,
    'La femelle de votre animal nourrit-elle son petit
avec du lait ?').

question(mange_viande,
    'Votre animal mange-t-il de la viande rouge ?').

question(doigts_ailes,
    'Votre animal a-t-il des doigts ailes ?').

question(pouce_opposable,
    'Votre animal a-t-il un pouce opposable ?').

question(au_dessus_de_200,
    'Votre animal adulte pese-t-il plus de 200 Kg ?').

question(terrestre,'Votre animal est-il terrestre ?').

question(en_afrique,'Votre animal vit-il en Afrique ?').

?- resout(type_animal est lion).
Votre animal a-t-il une colonne vertebrale ? oui.
Votre animal a-t-il le sang chaud ? oui.
La femelle de votre animal nourrit-elle son petit
avec du lait ? oui.
Votre animal mange-t-il de la viande rouge ? oui.
Votre animal a-t-il des doigts ailes ? non.
Votre animal a-t-il un pouce opposable ? non.
Votre animal adulte pese-t-il plus de 200 Kg ? oui.
Votre animal est-il terrestre ? oui.
Votre animal vit-il en Afrique ? oui.
yes

?-
```

Annexe 3

Un exemple

```
%
% quick_hull(P, S),
% determine d'eveloppe convexe S du polygone P
%
quick_hull( S , Conv_S ) :-
    quicksort( S , L ),
    longueur( L , N ),
    convexe( N , L , Conv_S ),
    !.

% le calcul d'un convexe
% Cas triviaux
% un point
convexe( 1 , X , X ).

% deux points
convexe( 2 , X , X ).

% trois points
convexe( 3 , [X,Y,Z] , [X,Z] ):-
    angle(X,Y,Z,no_turn).

convexe( 3 , [X,Y,Z] , [X,Y,Z] ):-
    angle(X,Y,Z,right_turn).

convexe( 3 , [X,Y,Z] , [X,Z,Y] ):-
    angle(X,Y,Z,left_turn).

% cas general
convexe( N , S , Conv ) :-
    N > 3,
    partitionner( S , S1 , S2 , N , N1 , N2 ),
    convexe( N1 , S1 , C1 ),
    convexe( N2 , S2 , C2 ),
    union_convexe( C1 , C2 , Conv ).

% l'union de deux convexes donne un convexe
union_convexe( C1 , C2 , Conv ) :-
    merger( C1 , C2 , S ), % S est presque convexe
    graham( S , Conv ). % on le rend convexe
```

```
% reunion de deux convexes donne un polygone
% presque convexe
merger( C1 , C2 , S ) :-
    C1 = [ P | L ],
% On cherche l'appuis de gauche
    merger_gauche( P , C2 , Mc2 ),
    concat_list( Mc2 , C2 , N_2 ),
% On cherche l'appuis de droite
    merger_droite( P , N_2 , [ U | LD ] , V ),
% On cherche l'appuis de droite suivant
    merger_droite( U , C1 , L1 , T ),
    concat_list( L1 , NC1 , C1 ),
    concat_list( NC1 , L1 , N_C1 ),
% On cherche l'appuis de gauche suivant
    merger_gauche( V , N_C1 , L_1 ),
% on reunis les listes obtenues
    concat_list( LG1 , [ P | LG2 ] , L_1 ),
    concat_list( [P|LG2], [U|LD] , Conv1 ),
    concat_list( Conv1 , LG1 , S ).

% recherche d'un appuis droit
merger_droite( P , [ P1 ] , [ P1 ] , P1 ) :- !.

merger_droite( P , [ P1 , P2 | C2 ] , [ P1 ] , P1 ) :-
    angle( P , P1 , P2 , left_turn ),
    !.

merger_droite( P, [ P1, P2 | C2 ], [ P1 | Mc2 ] , V) :-
    merger_droite( P, [ P2 | C2 ], Mc2, V ).

% recherche d'un appuis gauche
merger_gauche( P , [ P1 ] , [ P1 ] ) :-!.

merger_gauche( P, [ P1, P2 | C2 ] , [ P1, P2| C2 ] ) :-
    angle( P , P1 , P2 , right_turn ),
    !.

merger_gauche( P , [ P1 , P2 | C2 ] , Mc2 ) :-
    merger_gauche( P , [ P2 | C2 ] , Mc2 ).

% Algorithme de Graham pour l'enveloppe convexe,
% C'est lineaire... sur la longueur du polygone
graham( [ Start | Queue ] , Conv ) :-
    graham_boucle( Start , [] , [Start|Queue] , Conv ).

graham_boucle( Start , Conv , [] , Conv) :- ! .

graham_boucle( Start, Tete, Queue, Conv ) :-
    modifier_liste( Start, Tete, N_tete, Queue, N_q),
    graham_boucle( Start , N_tete , N_q , Conv ).
```

```
% Gestion de cas particuliers pour l'algorithme
% cas d'un tourne à gauche
modifier_liste( Start , T , New_t , Q , New_q ) :-
    concat_list( Q , T , [ V , S_V , S_S_V | _ ] ),
    Q = [ V | V_q ],
    angle( V , S_V , S_S_V , right_turn ),
    concat_list( T, [ V ], New_t ),
    New_q = V_q,
    !.

% cas de linearite de V, S_V, S_S_V,
% on detruit le suivant de V, ie S_V
modifier_liste( Start , T , T , Q , New_q ) :-
    concat_list( Q , T , [ V , S_V , S_S_V | _ ] ),
    Q = [ V | V_q ],
    angle( V , S_V , S_S_V , no_turn ),
    V_q = [_|N_V_q],
    New_q = [V|N_V_q],
    !.

% cas de tourne a droite, on recule
% sauf si on est sur le point Start
modifier_liste( S, [S], [S], [_| N_q], N_q ) :-
% il y a un bug, c'est normalement
% modifier_liste( S, [S], [S], [T,_|N_q], [T|N_q] ) :-
    !.

% on n'est pas sur Start
modifier_liste( Start , T , New_t , Q , New_q ) :-
    concat_list( New_t, [X], T ),
% il y a un bug, c'est normalement
%       Q = [ Y, _ | V_q ],
%       New_q = [ X, Y | V_q ].
    Q = [ _ | V_q ],
    New_q = [ X | V_q ].

angle( [X1,Y1] , [X2,Y2] , [X3,Y3] , Angle ) :-
    X is X1 * (Y2-Y3) - X2 * (Y1-Y3) + X3 * (Y1-Y2),
    type_angle( X , Angle ).

type_angle( X , left_turn )  :- X < 0.
type_angle( 0 , no_turn ).
type_angle( X , right_turn ) :- X > 0.

longueur( [] , 0 ).
longueur( [ X | L ] , N ) :-
    longueur( L , N1 ),
    N is N1 + 1.
```

```
partitionner( S , S1 , S2 , N , N1 , N2 ) :-
    NN is N // 2,
    couper( NN , S , S1 , S2 ),
    longueur( S1 , N1 ),
    longueur( S2 , N2 ).

couper( N , S , S1 , S2 ) :-
    premier( S , _s1 , _s2 , N ),
    traiter( _s1 , _s2 , S1 , S2 ).

% traitement de cas particulier pour le partitionnement
% en deux listes presque égales
traiter( _s1 , [ [X,_] , [X,Y] | _s2 ] , S1 , S2 ) :-
    concat_list( _ , [ [X,_] ] ,  _s1 ),
    !,
    concat_list( _s1 , [ [X,Y] ] , _new_s1 ),
    traiter( _new_s1 , _s2 , S1 , S2 ).

traiter( _s1 , [ [X,Y] | S2 ] , S1 , S2 ) :-
    concat_list( _ , [ [X,_] ] , _s1 ),
    !,
    concat_list( _s1 , [ [X,Y] ] , S1 ).

traiter( _s1 , _s2 , _s1 , _s2 ).

premier( S , [] , S , 0 ).

premier( [ X | S ] , [ X | S1 ] , S2 , N ) :-
    NN is N - 1,
    premier( S , S1 , S2 , NN ).

%   declaration de quick_sort des points d'un polygone
quicksort([],[]).

quicksort([X|Xs],Ys):-
        partager(Xs,X,Littles,Bigs),
        quicksort(Littles,Ls),
        quicksort(Bigs,Bs),
        concat_list(Ls,[X|Bs],Ys).

partager([X|Xs],Y,[X|Ls],Bs) :-
        inferieur( X , Y ),
        partager(Xs,Y,Ls,Bs).

partager([X|Xs],Y,Ls,[X|Bs]) :-
        partager(Xs,Y,Ls,Bs).

partager([],Y,[],[]).
```

```
inferieur( [X1,Y1] , [X2,Y2] ) :-
    X1 < X2,
    !.

inferieur(  [X1,Y1] , [X2,Y2] ) :-
    X1 = X2,
    Y1 < Y2.

% l'append de deux listes...
concat_list([],L,L).

concat_list([T|Q],L,[T|V]):-
        concat_list(Q,L,V).

/* un test de convexe */
test( S ) :-
    L =[
        [0,0], [2,0],
        [3,1], [0,2],
        [2,2], [5,2],
        [3,4], [5,4]
       ],
    quick_hull( L , S ).

% exemple d'utilisation d'assertions evaluables
% l'évaluation se fait par eux même

% concat_list/3 ie append, c'est correct
assertions(concat_list(X, Y, Z)) :-
        concat_list(X, Y, Z).

% longueur d'une liste
assertions(longueur(X, Y)) :-
        longueur(X, Y).

% séparation d'un liste en deux listes
% longueur(S1) = NN
assertions(premier(S, S1, S2, NN)) :-
        premier(S, S1, S2, NN).

% calcul d'une valeur d'un angle
assertions(angle(X, Y, Z, T)) :-
        angle(X, Y, Z, T).

% type de l'angle
assertions(type_angle( X, Angle )) :-
        type_angle( X, Angle ).
```

Annexe 4

Extensions

```
%   Debugging
%
%   Réalisation des extensions de la méthode de Shapiro
%   pour la coupure, la négation, la détection de boucles
%
:- unknown(_,fail).
% Echec sur l'inconnu
:- compile('../Prolog/buildin.pl').

% solve(But) méta-interprétation du but
solve(But) :-
    initialisation,
    meta_eval(But,Meta),
    time(_),
    meta_wait(But,Meta).
solve(_) :-
    time(T),
    ecrire('~ntemps de resolution : ~3d sec.~n',T),
    fail.

% Dialogue de fin de résolution
% Il y a une solution.
meta_wait(But,Meta) :-
    Meta,
    ecrire('~nYes : ~w~n',[But]),
    ecrire('~n Solution juste ? ',[]),
    oui_non(false),
    ecrire('~n.... Diagnostic Recherche',[]),
    ecrire('~n.... d''une Clause fausse...~n',[]),
    asserta(false_solution),
    Meta.

%   Il n'y a pas de solution
meta_wait(But,Meta) :-
    ecrire('~n Pas de solution sur le but',[]),
    ecrire('~n ~w.~n Ok ? ',[But]),
    oui_non(false),
    asserta(no_solution),
    ecrire('~n.... Diagnostic Recherche',[]),
    ecrire('~n.... d''un but insatisfiable~n',[]),
    Meta.
%
```

```
%   predicats de tranformation des clauses
%
%   transformation func( Arg ) -->  func , arite , Arg
%
meta_func(Tete,Func,Arite,Arg) :-
    Tete =.. [Func|Arg],
    functor(Tete,Func,Arite).
%
%   transformation Queue --> Meta.
%   Appel d'une meta_variable
%
meta_eval(X,(nonvar(X),meta_eval(X,M_X),M_X)) :-
    var(X),
    !.
%
%   cas du Call
%
meta_eval(call(X),(nonvar(X),meta_eval(X,M),M)):-
    !.

meta_eval(asserta(X),meta_asserta(X)):-
    !.

meta_eval(assertz(X),meta_assertz(X)):-
    !.

meta_eval(retract(X),meta_retract(X)):-
    !.

meta_eval(clause(T,Q),meta_clause(T,Q)):-
    !.

meta_eval(ancestors(X),meta_ancetre(X)):-
    !.

meta_eval(abolish(F,Ar),meta_abolish(F,Ar)) :-
    !.

%   conjonction
%
meta_eval((A1 , A2),(Meta1 , Meta2)) :-
    !,
    meta_eval(A1,Meta1),
    meta_eval(A2,Meta2).

%   cas du () -> () ; ().
%
meta_eval((A->B;C),(M_A->M_B;M_C)) :-
    !,
    meta_eval(A,M_A),
```

```
        meta_eval(B,M_B),
        meta_eval(C,M_C).

%   cas du () -> () .
%
meta_eval((A->B),(M_A->M_B)) :-
        !,
        meta_eval(A,M_A),
        meta_eval(B,M_B).

%   disjonction
%
meta_eval((A1 ; A2),(Meta1 ; Meta2)) :-
        !,
        meta_eval(A1,Meta1),
        meta_eval(A2,Meta2).

%   cas du cut
%
meta_eval(! , ( ! , meta_cut ) ) :- !.

%   cas de la negation
%
meta_eval(\+ But,\+ Meta_but) :-
        !,
        meta_eval(But,Meta_but).

%   on est sur une des primitives du meta
%
meta_eval(But,But) :-
        meta_definition(But),
        !.

%   cas des fonctions systemes
%
meta_eval(A,A) :-
        predef_but(A),
        !.

%   defaut
%
meta_eval(But,meta_call(But,Func,Arite,Arg)) :-
        meta_func(But,Func,Arite,Arg).

%   predicats de modification de la base
%
meta_abolish(F,Ar) :-
        repeat,
        \+ retract((meta(F,Ar,_,_,_):-_)),
        !.
```

```
meta_asserta(X) :-
    traiter(X,Tete,Queue),
    meta_func(Tete,Func,Ar,Arg),
    meta_eval(Queue,meta_eval),
    asserta(
        (
        meta(Func,Ar,0,Arg,Queue) :-
            meta_call_appel(Func,Ar,0,Arg),
            meta_eval,
            meta_call_reussit(Func,Ar,0)
        )).

meta_assertz(X) :-
    traiter(X,Tete,Queue),
    meta_func(Tete,Func,Ar,Arg),
    meta_eval(Queue,meta_eval),
    assertz(
        (
        meta(Func,Ar,0,Arg,Queue) :-
            meta_call_appel(Func,Ar,0,Arg),
            meta_eval,
            meta_call_reussit(Func,Ar,0)
        )).

meta_retract(X) :-
    traiter(X,Tete,Queue),
    meta_func(Tete,Func,Ar,Arg),
    retract((meta(Func,Ar,_,Arg,Queue) :- _ )).

meta_clause(Tete,Queue) :-
    meta_func(Tete,Func,Ar,Arg),
    clause(meta(Func,Ar,Arg,_,Queue),_).

clause_num(F,Ar,Num) :-
    retract(deja_vue(F,Ar,N)),
    Num is N + 1,
    asserta(deja_vue(F,Ar,Num)),
    !.
clause_num(F,Ar,1) :-
    meta_abolish(F,Ar),
    asserta(deja_vue(F,Ar,1)).

%   predicats d'appel d'un But.
%
meta_call(But,Func,Ar,Arg) :-
    clause(false_solution,true),
    !,
    But,
    (
```

```
        question_tous(But,true)
        -> true
        ;
            meta(Func,Ar,Num,Arg,_),
            clause_fausse(But,Func,Ar,Num)
    ).

%   realisation 1 ...
%   En profondeur d'abord
%meta_call(But,Func,Ar,Arg) :-
%   clause(no_solution,true),
%   !,
%   question_existe(But,true),
%   meta(Func,Ar,_,Arg,_),
%   but_faux(But,Func,Ar,Arg).
%
%   en largeur d'abord
meta_call(But,Func,Ar,Arg) :-
    clause(no_solution,true),
    !,
    question_existe(But,true),
    ( meta(Func,Ar,_,Arg,_)
        -> true
        ;  ecrire('~nBut insatisfiable : ~w~n',But),
           abort
    ).

%   Cas général, pas de debug.
meta_call(_,Func,Ar,Arg) :-
    meta(Func,Ar,_,Arg,_).

%   On est sur une clause fausse
clause_fausse(But,Func,Ar,Num) :-
    clause(meta(Func,Ar,Num,Arg,Queue),_),
    N_but =.. [Func|Arg],
    numbervars((N_but:-Queue),0,_),
    ecrire('~n Sur le but : ~w',But),
    ecrire('~n~n Clause fausse :',[]),
    ecrire('~nPaquet "~w", clause ~d',[(Func/Ar),Num]),
    ecrire('~n~n                ~w~n',[(N_but:-Queue)]),
    abort.

% On est sur un but faux
but_faux(But,_,_,_) :-
    % il existe une solution
    But,
    % elle est demontrable
    !.
but_faux(_,Func,Ar,Arg) :-
    % on continue sur ce but
```

```
    meta(Func,Ar,_,Arg,_).

but_faux(But,_,_,_) :-
    % la solution n'est pas d'emontrable
    ecrire('~n=====> But insatisfiable : ~w~n',But),
    abort.

meta_call_appel(Func,Ar,Num_cl,Arg) :-
    empiler_but(Func,Ar,Num_cl,Arg,Taille),
    meta_test_boucle(Func,Ar,Num_cl,Arg,Taille).

meta_call_appel(Func,Ar,Num_cl,Arg) :-
    depiler_but(Func,Ar,Num_cl,Arg,_),
    fail.

meta_call_reussit(Func,Ar,Num_cl) :-
    depiler_but(Func,Ar,Num_cl,Arg,Taille),
    (
        true
    ;
        empiler_but(Func,Ar,Num_cl,Arg,Taille),
        fail
    ).

empiler_but(Func,Ar,Num_cl,Arg,Y) :-
    retract(taille_pile_num(X)),
    Y is X + 1,
    asserta(taille_pile_num(Y)),
    asserta(meta_pile(Func,Ar,Num_cl,Arg,Y)).

depiler_but(Func,Ar,Num_cl,Arg,X) :-
    retract(taille_pile_num(X)),
    Y is X - 1,
    asserta(taille_pile_num(Y)),
    retract(meta_pile(Func,Ar,Num_cl,Arg,X)).

meta_test_boucle(Func,Ar,Num_cl,Arg_2,X) :-
    Y is X // 2,
    meta_pile(Func,Ar,Num_cl,Arg_1,Y),
    similaire(Arg_1,Arg_2),
    % si les buts sont similaires
    !,
    But_1 =.. [Func|Arg_1],
    numbervars(But_1,0,_),
    message(But_1,'@  ',(X,Y)),
    nl,
    liste_buts(L),
    reverse(L,Rev_L),
    chercher_loop(Rev_L,Loop_segment),
    ecrire_loop(Loop_segment),
```

```
    abort.
meta_test_boucle(_,_,_,_,_).

liste_buts([(Func,Ar,Num,Arg)|L]) :-
    retract(meta_pile(Func,Ar,Num,Arg,_)),
    !,
    liste_buts(L).
liste_buts([]).

chercher_loop([But|Sous_buts],[But|Sous_liste]) :-
    looping_liste(But,Sous_buts,Sous_liste),
    !.
chercher_loop([_|Sous_buts],Sous_liste) :-
    chercher_loop(Sous_buts,Sous_liste).

looping_liste((F,Ar,N,Arg_1),
            [(F,Ar,N,Arg_2)|_],[(F,Ar,N,Arg_2)]) :-
    similaire(Arg_1,Arg_2),
    But =.. [F|Arg_1],
    numbervars(But,0,_),
    ecrire('~n Looping : ~w ~n~n',But),
    !.
looping_liste(But,
            [Sous_but|Liste],[Sous_but|Sous_liste]) :-
    looping_liste(But,Liste,Sous_liste).

ecrire_loop([]).
ecrire_loop([(F,_,Num,Arg)|L]) :-
    But =.. [F|Arg],
    numbervars(But,0,_),
    ecrire('
    But : ~w , Clause : ~d~n',[But,Num]),
    ecrire_loop(L).

%   test_argument : List1 , List2
%   teste si List1 et List2 sont "similaires"
similaire([],[]).
similaire([A1|L1],[A2|L2]) :-
    test_argument(A1,A2),
    similaire(L1,L2).

%   test_argument : Arg1 , Arg2
%   teste si Arg1 et Arg2 sont "similaires"
test_argument(A1,A2) :-
    atomic(A1),
    !,
    A1 == A2.
test_argument(A1,A2) :-
    var(A1),
    !,
```

```
    var(A2).
test_argument(A1,A2) :-
    nonvar(A2),
    functor(A1,F,Ar),
    functor(A2,F,Ar),
    structure_test(A1,A2).

%   Structure_test : Arg1 , Arg2
%   Teste si Arg1 et Arg2 sont identiques
%   ou si Arg1 est inclus dans Arg2
structure_test(A1,A2) :-
    A1 =..[_|Arg1],
    A2 =..[_|Arg2],
    similaire(Arg1,Arg2),
    !.
structure_test(A1,A2) :-
    struct_member(A1,A2).

%   Struct_member : Arg1 , Arg2
%   Teste si Arg1 est inclus dans Arg2
struct_member(A1,A2) :-
    numbervars(A1,0,N),
    numbervars(A2,0,N),
    A1 = A2,
    !.
struct_member(A1,A2) :-
    functor(A2,F,Ar),
    A2 =..[F|Arg],
    member(Z,Arg),
    nonvar(Z),
    functor(Z,F,Ar),
    struct_member(A1,Z).
meta_pile(But) :-
    meta_pile(Func,_,_,Arg,_),
    But=..[Func|Arg].

%
%   predicats d'appel d'un predicat systeme
%
predef_but(But) :-
    functor( But , Func , Ar ),
    meta_build_in( Func , Ar ),
    !.

predef_but(But) :-
    current_predicate( _ , But ),
    \+ predicate_property( But , interpreted ).

%   predicats d'appel du cut
meta_cut.
```

```
meta_cut :-
    depiler_but(_,_,_,_,_),
    fail.

%   predicats d'envoi d'un message
%   meta_trace.
%
message(But,Label,Num_cl) :-
    ecrire('~w~w [~w]',[Label,But,Num_cl]).

ecrire(Form,Arg) :-
    format(Form,Arg).

%   predicats de question "Oracle".
ground( P ) :-
    numbervars(P,0,0).

var_list(X,L) :-
    var_list(X,L,[]).
var_list(X,[X|L],L) :-
    var(X),
    !.
var_list(T,L1,L) :-
    T =.. [_|Arg],
    !,
    varlist1(Arg,L1,L).

varlist1([T|A],L0,L) :-
    var_list(T,L0,L1),
    varlist1(A,L1,L).
varlist1([],L,L).

list_to_and([],true) :-
    !.
list_to_and([X],X) :-
    !.
list_to_and([X|Xs],(X,Ys)) :-
    list_to_and(Xs,Ys).

liste_var(P,P_vars) :-
    var_list(P,List),
    list_to_and(List,P_vars).

une_instance(P,Q) :-
    copy_term(P,X),
    copy_term(Q,Y),
    numbervars(X,0,_),
    X = Y.

bag_of(X,P,S) :-
```

```
    bagof(X,P,S),
    !.
bag_of(_,_,[]).

set_of(X,P,S) :-
    setof(X,P,S),
    !.
set_of(_,_,[]).

fact(P,V) :-
    clause(solutions(P,S),true),
    (member(P,S) -> V=true; V= false).

demander_solution(P,S) :-
    bag_of(P,demander_solution(P),S),
    asserta(solutions(P,S)).

demander_solution(P) :-
    ecrire('~nQuestion~n Sur le but: ~w ~n Ok ? ',[P]),
    oui_non(true),
    questionner(P).

questionner(P) :-
    numbervars(P,0,0),
    !.
questionner(P) :-
    liste_var(P,P_vars),
    repondre(P_vars).

repondre(X) :-
    var(X),
    !,
    ecrire('~nAvec : ~w ? ',[X]),
    read(X).
repondre((X,Y)) :-
    repondre(X),
    repondre(Y).

question_existe(P,V) :-
    copy_term(P,Q),
    clause(solutions(Q,S),true),
    une_instance(P,Q),
    !,
    (member(P,S) -> V = true; V = false).

question_existe(P,true) :-
    fact(P,true),
    !.

question_existe(P,V) :-
```

```
    demander_solution(P,S),
    ( S=[] -> V=false ; (S=[P],V=true) ).

question_tous( P , V ) :-
    ground( P ),
    question_existe( P , V ).

oui_non(Rep) :-
    read(X),
    oui_non(X,Y),
    !,
    Rep = Y.
oui_non(X) :-
    ecrire('Repondre par oui ou non',[]),
    oui_non(X).

oui_non(o,true).
oui_non(oui,true).
oui_non(y,true).
oui_non(yes,true).
oui_non('O',true).
oui_non('Oui',true).
oui_non('Y',true).
oui_non('Yes',true).
oui_non(n,false).
oui_non(no,false).
oui_non('N',false).
oui_non('No',false).
oui_non('Non',false).

%   predicats speciaux
initialisation :-
    abolish(false_solution,0),
    abolish(no_solution,0),
    abolish(solution,0),
    asserta(solution),
    abolish(taille_pile_num,1),
    asserta(taille_pile_num(0)),
    abolish(meta_pile,5).

clear :-
    abolish(solutions,2).

init(File) :-
    time(_),
    abolish(solutions,2),
    abolish(deja_vue,3),
    get_clause(Tete,Queue,File),
    meta_func(Tete,Func,Arite,Arg),
    \+ definition(Func),
```

```
        clause_num(Func,Arite,Num),
        meta_eval(Queue,Meta),
        assertz((Tete:-Queue)),
        assertz((
            meta(Func,Arite,Num,Arg,Queue) :-
                meta_call_appel(Func,Arite,Num,Arg),
                Meta,
                meta_call_reussit(Func,Arite,Num)
        )),
        fail.
    init(_) :-
        time(T),
        ecrire('temps de traitement : ~3d sec.~n',T).

    init :-
        tell('getclc'),
        listing,
        told,
        init('getclc'),
        tell('getclc'),
        told.

    load( (X,F) ) :-
        !,
        load(X),
        load(F).
    load(F) :-
        init(F).

    get_clause( Tete , Queue , File ) :-
        seeing(Input),
        meta_open(File),
        repeat,
        read(Terme),
        (
            Terme = end_of_file,
            seen,
            see(Input),
            !,
            fail
        ;
            traiter(Terme, Tete, Queue)
        ).

    meta_open(F) :-
        absolute_file_name(F,Chemin),
        see(Chemin),
        format('~n Chargement du fichier : ~w~n',Chemin).

    traiter(:-( But ) , _ , _ ) :-
```

```
    !,
    But,
    fail.
traiter(:-( Tete , Queue ), Tete, Queue) :-
    !.
traiter(Tete,Tete,true).

meta_definition(But) :-
    functor(But,Func,_),
    definition(Func),
    !.

time( T ) :-
    statistics(runtime,[_|T]).

member(X,[X|_]).
member(X,[_|L]) :-
    member(X,L).

append([],L,L).
append([X|Xs],L,[X|Res]) :-
    append(Xs,L,Res).

reverse(L1,L2) :-
    reverse(L1,[],L2).

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