Les types en Prolog : un système d'inférence de type et ses applications

Hamid Azzoune

THESE

présentée par

Hamid AZZOUNE

Pour obtenir le titre de **DOCTEUR**

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

(arrêté ministériel du 5 Juillet 1984)

Spécialité: INFORMATIQUE

****

LES TYPES EN PROLOG.
UN SYSTEME D'INFERENCE DE TYPE
ET SES APPLICATIONS

****

Date de soutenance: 11 Janvier 1989

Composition du Jury: *Président: L. Trilling*
D. Herman
J. Mossiere
P. Jacquet
J. Briat

Thèse préparée au sein du Laboratoire de Génie Informatique

A ma famille

Je tiens à remercier:

Monsieur Laurent Trilling, Professeur à l'Université Joseph Fourier I de Grenoble, de m'avoir fait l'honneur de juger ce travail et de présider le jury de cette thèse,

Monsieur Daniel Herman, Professeur à l'IRISA Rennes, pour avoir apporté son jugement sur cette thèse,

Monsieur Jacques Mossiere, Directeur de l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées (ENSIMAG) et Professeur à l'ENSIMAG, pour avoir bien voulu diriger ce travail,

Monsieur Paul Jacquet, Maitre de conférences à l'Ecole Nationale Superieure d'Informatique et de Mathématiques Appliquées (ENSIMAG) pour avoir apporter son jugement sur cette thèse,

Monsieur Jacques Briat, Maitre de Conférences à l'Université Joseph Fourier I de Grenoble, qui a été à l'initiative de cette thèse et qui a bien voulu me guider.

Je tiens à remercier aussi Monsieur Y. Rouzaud pour toutes les remarques et suggestions pertinentes qui m'ont beaucoup aider pour réaliser ce travail.

Je tiens à remercier tous les membres de l'équipe FLOP et toutes les personnes du Laboratoire de Génie Informatique qui m'ont été très sympathiques.

Cette thèse a été effectué avec le support d'une bourse de coopération Franco-Algerienne.

Table des matières

Titre *Page*

Introduction 1

Chapitre 1: Pourquoi typer en Prolog 5

1 Prolog 5

2 Bases Logiques et syntaxiques de PROLOG 7

2.1 Termes 7

2.1.1 Univers de Herbrand 7

2.1.2 Les Variables 8

2.2 Littéral atomique 8

2.3 Clause de Horn 8

2.3.1 Les faits 9

2.3.2 Clauses avec corps 9

2.4 Procédure 9

2.5 Programme 9

2.6 La question 10

2.7 La clause vide 10

2.8 Les entités syntaxiques et lexicographiques 10

3 Sémantique d'une clause 11

3.1 Sémantique déclarative 11

3.2 Sémantique procédurale 12

4 Principe de la résolution 13

4.1 Substitution 14

4.2 Unification 15
4.3 Arbre et/ou 17
4.4 Stratégies de résolution 19
4.4.1. Recherche en profondeur d'abord 19
4.4.2. Recherche en largeur d'abord 19
4.5 Retour en arrière 20

5 Mise en oeuvre de la résolution en PROLOG 21
5.1 Stratégie depth-first de PROLOG 21
5.2 Représentation des substitutions et déliaison 26

6 Pourquoi les types en Prolog 27

Chapitre 2: Les types en PROLOG 31

1 Introduction 31

2 Vérifier les types [Mycroft82] 33
2.1 Représentation des types 33
2.2 Typer une clause 34
2.3 Bon-typage 35
2.4 Autres travaux 38
2.4.1 Les types 38
2.4.2 Les modes 40

3 Calculer les types [Mishra84] 41
3.1 Les types 41
3.2 Représentation des types 43
3.3 Le système d'inférence 44
3.4 Vérifications 46
3.5 Exemples d'inférence de type 47
3.6 Autres travaux 49

4 Bilan 51
4.1 L'approche déclarative 51
4.2 L'approche inférentielle 52

5 Conclusion 54

Chapitre 3: Une méthode d'inférence de type 55

1 Introduction 55

2 Eléments de type 57

3 La méthode d'inférence 59
3.1 Les bases de la méthode d'inférence 59
3.2 Principe de la méthode 59
3.2.1 Equations de type 60
3.2.2 Résolution des équations de type 61
3.3 Les faits 61
3.4 Clauses avec corps typé 62
3.4.1 Corps à un but 62
3.4.1.1 Les équations de type 63
3.4.1.2 Unification de type (génération inéquations) 63
Simulation de l'unification 63
Simualtion de tous les appels possibles 66
3.4.1.3 Transformation des systèmes d'inéquations 67
Filtrage 67
Normalisation 68
Inéquations productives et inéquations contraintes 70
Vérification des inéquations contraintes 71
Opération d'union 72
Solutions maximales 72
Remplacement des variables de type 73

3.4.1.4 Résolution finale des équations de type 73
Intersection 74
Simplification 75
3.4.1.5 Exemple d'inférence 75
3.4.2 Cas général 78
Eclatement de & et de ou 78
Second filtrage 79
Exemples d'inférence 79
Bilan général 81
3.4.3 Utilisation des types déjà calculés 81
3.5 Clauses avec corps incomplètement typé 82
3.5.1 Récursion simple (ou plate) 82
3.5.1.1 Les équations de type 83
3.5.1.2 Unification de type 83
Traitement spécial des appels récursifs 83
3.5.1.3 Transformation des inéquations 84
Normalisation 84
Notion d'hypothèses 85
Vérification des inéquations contraintes 85
Notion de contraintes 87
3.5.1.4 Résolution des équations de type 88
Décomposition 89
Vérification des contraintes 90
3.5.1.5 Exemples 91
Double récursion 96
3.5.2 Récursion profonde 97
Inéquations intermédiaires 97

4. Bilan de la méthode 101
4.1 Situations d'échec 101
4.1.1 Au niveau du premier filtrage 101
4.1.2 Au niveau de la normalisation 103
4.1.3 Au niveau de la vérification des contraintes 104

4.1.4 Bilan des situations d'échec 105
4.2 Les types inférés sont corrects 105
4.3 Bilan général 106

5. Travail réalisé 107

Chapitre 4: Bilan et Perspectives 109

1 Comparaison 109

2 Variantes et améliorations 110

3 Quelques applications des types inférés 111
Eviter les exécutions inutiles 112
Bon-typage 113
Détection d'erreurs 113
Détection des branches en échec 114
Remontée des propriétés des arguments 114
Unification partielle 115
Eliminer les variables redondantes 116
Prédicats toujours vrais 117
Déterminisme 117
En compilation 119
Détermination des modes 120
Cardinalité 120
Autres applications 120

Conclusion 123
Annexe 1: Le système d'inférence de type 125

Les étapes de l'inférence 125

1 Décomposition du programme 126
2 Inférence des types d'une partie indépendante 129
2.1 Calcul des types des prédicats 129
2.1.1 Règles de calcul des types 130
2.1.2 Equations de type 131
2.2 Génération des inéquations 132
2.2.1 Règles de génération 132
2.3 Transformation des inéquations 134
2.3.1 Principe de la transformation 134
2.3.2 Algorithmes de la transformation 135
Inéquations toujours vraies 137
Inéquations toujours fausses 137
Traitement appels récursifs 137
Normalisation 137
2.3.3 Résolution des équations de type 138
Elimination des composantes de type 138
Elimination des variables d'hypothèse 138
Algorithme de résolution des équations de type 139
Algorithme de décomposition 140
Vérification des contraintes 141
Intersection 142

3 Utilisation des types calculés 143

4 Possiblité de déclaration des types 144

Annexe 2: Exemples d'inférences 145

Références 161

INTRODUCTION

La dernière décennie s'est caractérisée par le développement de produits logiciels de plus en plus complexes, à la durée de vie importante, à l'évolution permanente ainsi que par une exigence de fiabilité toujours croissante. L'industrie du logiciel a exigé des langages de programmation permettant de:

- diminuer les coûts de programmation individuelle,
- développer de façon séparée des logiciels,
- faciliter la réalisation et l'évolution de composants logiciels,
- garantir la sûreté de programmation.

Les langages classiques de simple abstraction de la machine physique se sont enrichis de concepts et opérateurs permettant toujours de contrôler finement une machine mais aussi de contrôler la qualité des programmes et de leur assemblage (types, modules).

Une autre approche a été le développement de langages fondés sur une théorie mathématique et une méthode d'évaluation. Dans cette catégorie, nous trouvons:

- Les langages fonctionnels, basés sur le lambda-calcul, avec la réduction comme méthode de calcul.
- Les langages logiques, basés sur la logique des prédicats du premier ordre, avec la résolution comme méthode de calcul.

LISP et PROLOG en sont les représentants les plus connus.

Ces langages se caractérisent par une syntaxe très simple, un minimum de symboles prédéfinis mais "fortement significatifs". Programmer dans de

tels langages produit des programmes concis par rapport aux langages classiques. On dit qu'ils ont un fort pouvoir d'expression.

Les inconvénients de cette "densité de signification" sont:

- une faible lisibilité, c'est-à-dire que les erreurs d'interprétation (par le programmeur) du sens d'une formule sont fréquentes,
- une forte sensibilité aux erreurs typographiques mineures, c'est-à-dire qu'une erreur de frappe peut fournir un programme différent de celui pensé par le programmeur.

Ces erreurs et/ou comportements non désirés ne sont détectés qu'à l'exécution du programme.

Ces fautes sont particulièrement fréquentes en PROLOG du fait de certains traits spécifiques du langage:

- la typographie (majuscule-minuscule) sert à distinguer les entités de base du langage (variable ou constante).
- la ponctuation est toujours significative et dénote une construction d'objet (terme et prédicat) ou une connexion logique (ET, OU).

Il s'ensuit qu'il est très rare qu'un programme incorrect soit rejeté au niveau de l'analyse syntaxique.

La sémantique du langage accroît ce phénomène dans la mesure où un programme PROLOG, à la différence des autres langages, ne définit pas une valeur calculable "a priori" en fonction d'autres valeurs mais un ensemble de propriétés que les valeurs des variables (solutions) doivent satisfaire. Il s'ensuit que l'échec à trouver de telles valeurs est un comportement correct de l'exécution d'un programme PROLOG. Le critère "pas de solutions" ne permet pas de distinguer un programme sans solution d'un programme erroné comme c'est le cas dans les autres langages.

L'introduction de la notion de type en PROLOG a été considérée comme un moyen de contourner cette fragilité intrinsèque au langage sans le remettre en cause. Comme dans les autres langages, les types sont un

moyen de dénoter les domaines de valeurs sur lesquels est défini un programme ou une partie du programme. Ils sont utiles pour la documentation du programme (lisibilité accrue) et pour la sécurité dès qu'ils s'accompagnent d'un "calcul" qui permet de dire qu'un programme ou une partie de programme est conforme à un certain typage. Ils pourraient être utiles pour des transformations et/ou optimisations des programmes PROLOG.

Plan de la thèse: Le plan de notre thèse est le suivant:

- Le chapitre 1 présente le langage PROLOG et les concepts clés du langage. Cette présentation est suivie d'une description des fragilités du langage qui justifie l'introduction de la notion de type.

- Le chapitre 2 présente ensuite deux vues du typage d'un programme PROLOG:
  * Une vue "déclarative" analogue aux autres langages, où il s'agit de déclarer des types et de vérifier la conformité du programme à cette déclaration.
  * Une vue "inférentielle", où il s'agit d'extraire automatiquement (inférer) les types d'un programme.

  Cette présentation s'appuie sur les travaux de MYCROFT-O'KEEFE pour l'approche déclarative et de MISHRA pour l'approche inférentielle. Elle se termine par un bilan de ces deux approches.

- Le chapitre 3 décrit notre système d'inférence de type pour PROLOG (sans *assert*, *retract*, on ne tient pas compte du !).

- Le chapitre 4 discute de l'utilisation des résultats de l'inférence de type, ainsi que de l'intérêt de la connaissance des types pour l'optimisation des programmes.

  En conclusion, nous faisons le point sur l'apport de cette thèse et le travail restant à faire.

CHAPITRE 1

POURQUOI TYPER EN PROLOG

1. PROLOG

Dans ce chapitre nous donnons les notions de base du langage PROLOG. Nous essayons de faire apparaître quelques aspects du langage qui permettent d'argumenter l'introduction de la notion de type en PROLOG.

PROLOG (pour PROgrammation LOGique) est un langage basé sur la logique du 1er ordre *[Changlee73], [Shapiro86], [Shoenfield67]* (logique mathématique).

Il a été conçu par A. Colmerauer *[Colmerauer75]* avec l'aide de Ph. Roussel en 1971 à l'université d'Aix-Marseille à la suite des travaux de A. Colmerauer sur le traitement des langues naturelles. Le premier interpréteur a été réalisé par Ph. Roussel *[Roussel75]* en Algol W. C'est en 1973 que R. Kowalski et Van Emden *[Kowalski73]* établissent la sémantique de PROLOG. On trouve un historique complet sur PROLOG dans *[Van Caneghem86]*.

Connu au début seulement des spécialistes, son audience est maintenant largement internationale *[Clocksin81], [Condillac86], [Dincbas85], [Donz84], [Giannesini85]*. Le projet Japonais *Ordinateurs de cinquième génération [Icot80]* est fondé sur le langage PROLOG, ainsi que divers projets Européens et Americains.

Par ailleurs, on trouve actuellement des interpréteurs et des compilateurs de plus en plus performants *[Bonnard86], [Clocksin85], [Colmerauer80],*

[Colmerauer82b], [Criss86], [Garby85], [Pereira84], [VanRoy84], [Warren77], [Warren83]. Cette liste de références n'est pas exhaustive.

Programmer en PROLOG consiste à décrire les connaissances nécessaires pour résoudre un problème; l'interprète recherche la (les) solution(s) à une requête posée en appliquant la résolution et l'unification *[Robinson65]*. Ceci le différencie des autres langages classiques (impératifs), où le programmeur doit expliciter pas à pas le chemin conduisant à la solution. En PROLOG, le programmeur consacre tous ses efforts à l'expression de son problème, les tâches subalternes étant prises en charge par le système PROLOG. C'est un langage dit *déclaratif*. On décrit le *QUOI* (ce que l'on veut faire, sous la forme d'un ensemble de "déclarations") mais non le *COMMENT* (sous la forme d'une suite d'instructions).

2. Bases logiques et syntaxiques de PROLOG

2.1. Termes:

Les objets que manipule PROLOG sont appelés "*Termes*". Un Terme est:

* Une variable ou
* Une constante (*identificateur, entier, réel, chaîne*) ou
* Un symbole de fonction (foncteur) n-aire appliqué à un n-uplet de termes.

Tout terme constant pris par un argument lors de toutes les exécutions d'un programme PROLOG appartient à un domaine (généralement infini) appelé *Univers de Herbrand* associé au programme.

2.1.1. Univers de Herbrand.

On appelle "*Univers de Herbrand*" *[Herbrand68], [Melter72]* d'un programme PROLOG S, l'ensemble des *termes constants (appelé aussi termes clos)* défini récursivement de la manière suivante:

* Soit U_0 l'ensemble des constantes apparaissant dans S, (si $U_0 = \varnothing$, on choisit une constante arbitraire: c).
* U_{i+1} est défini récursivement comme étant la réunion de U_0, ... U_i et de l'ensemble de tous les termes constants de la forme $f(t_1, ..., t_n)$ pour tous les foncteurs n-aires f apparaissant dans S; les t_j sont des éléments de U_i.
* $H = U_\infty$ est appelé *l'Univers de Herbrand* associé au programme.

Exemple d'Univers de Herbrand:

Supposons que S contienne les constantes {a, b} et le foncteur { f } d'arité 2, alors les U_i sont construits de la manière suivante:

U_0 = { a, b }

U_1 = { a, b, f(a, a), f(a, b), f(b, a), f(b, b) }

U_2 = { a, b, f(a, a), f(a, b), ...

U_∞ est l'Univers de Herbrand associé à S.

2.1.2. Les Variables

En plus des éléments appartenant à l'Univers de Herbrand, un terme peut contenir des variables ou être lui même une variable. Une variable en PROLOG est une entité non encore déterminée (une inconnue). Cette variable sera instanciée lorsqu'il lui sera affecté un terme t. Si ce terme t contient des variables alors cette instance représentera un ensemble de termes dont chaque élément est obtenu en substituant ces nouvelles variables par des termes. Le domaine où une variable prend ses valeurs en général est un sous-ensemble de l'Univers de Herbrand.

Le processus mis en oeuvre lors de l'association entre les variables et les valeurs qu'elles prennent est appelé "*substitution*". Ces substitutions sont élaborées au cours d'une opération appelée "*unification*". On obtient une instance constante d'un terme t en remplaçant (substituant) les variables de t par des éléments de l'Univers de Herbrand du programme associé.

2.2. Littéral atomique.

Un littéral atomique (appelé aussi formule atomique) est de la forme $p(t_1,....,t_n)$ où p est appelé "*prédicat*" et les t_i sont des termes appelés "*arguments*" du prédicat p *[Chang lee73], [Nilsson82]*. Les termes t_i prennent leurs valeurs constantes dans l'Univers de Herbrand. Le nombre n est appelé l'arité du prédicat p.

2.3. Clause de Horn

Une clause générale est une formule bien formée du premier ordre, quantifiée universellement et composée d'une disjonction de littéraux positifs et négatifs. Elle est de la forme:

$$A_1 \vee A_2 \vee .... \vee A_n \vee \neg B_1 \vee \neg B_2 \vee .... \vee \neg B_m$$

qui s'ecrit également:

$$A_1, A_2, ... A_n :- B_1, ... B_m$$

Une **clause de Horn** est une clause contenant au plus un littéral positif c'est-à-dire $0 \leq n \leq 1$. Elle est de la forme:

$$A_1 :- B_1, B_2, ..., B_m$$

2.3.1. Les faits

Si le nombre de littéraux négatifs est égal à zéro ($m = 0$) alors la clause de Horn est appelée un fait. Elle est de la forme:

$$p(t_1, ..., t_n).$$

où p est un prédicat et les t_i sont des termes.

2.3.2. Clauses avec corps

Si le nombre de littéraux négatifs est supérieur à zéro ($m > 0$) alors la clause est dite clause avec corps. Elle de la forme:

$$p_0(t_1, ..., t_n) :- p_1(...), p_2(...), ..., p_m(...)$$

où chaque p_i $i=0,1,..$ est un prédicat. $p_0(t_1,...,t_n)$ est appelé la tête de la clause, Chaque $p_i(...)$ $i=1,m$ est appelé un but. La conjonction de ces m buts forme le corps de la clause. Dans cette notation, le symbole :- représente l'implication et peut se prononcer **si**. Le symbole "," entre les buts du corps de la clause représente la conjonction.

2.4. Procédure

On définit une procédure comme étant un ensemble de clauses avec le même symbole de prédicat et la même arité dans la tête de chaque clause. Une procédure donne la définition du prédicat.

2.5. Programme

Un programme est constitué d'un ensemble de procédures.

2.6. La question

Une clause de Horn sans littéraux positifs est appelée la question, le but ou la réfutation. Le processus de résolution est généralement déclenché sur la formulation d'une question. Elle est de la forme:

$$p_1(...), ..., p_m(...).$$

2.7. La clause vide.

On définit enfin la clause vide. Elle correspond à une clause sans littéraux positifs et sans littéraux négatifs. Elle est représentée par □.

2.8. Les entités syntaxiques et lexicographiques

Les entités syntaxiques et lexicographiques de Prolog sont donc définies par (syntaxe de C-prolog *[Pereira84]*):

Terme ::= Var / Atome / entier / Foncteur(Terme[,Terme]*)
Var ::= majuscule[lettre*] / _ / _[lettre*]
Atome ::= minuscule[lettre*]
Foncteur ::= minuscule[lettre*]

Prédicat ::= minuscule[lettre*]
Littéral ::= Prédicat / Prédicat(Terme[,Terme]*)
Clause ::= Littéral :- Expression. / Littéral.
Expression ::= Littéral / Conjonction / Disjonction
Conjonction ::= Expression[,Expression]*
Disjonction ::= Expression[;Expression]*

Programme ::= Clause*

Remarque: Nous faisons remarquer qu'il n'y a pas de distinction lexicographique entre un littéral et un terme fonctionnel. Il n'y a pas de distinction syntaxique entre un prédicat et un foncteur. La notion de prédicat défini comme un paquet de clauses n'est pas une notion syntaxique.

3. Sémantique d'une clause.

Une phrase du langage PROLOG peut être interprétée de façon déclarative ou de façon opérationnelle.

3.1. Sémantique déclarative

Dans l'interprétation déclarative, on fait entièrement abstraction des notions liées à la machine ou à la stratégie de résolution.

a) Une clause de Horn

$$p \leftarrow q_1 , \ldots , q_m$$

peut être interprétée comme une clause conditionnelle:

pour toutes les variables X_1, X_2, X_3, ... apparaissant dans la clause
p est vrai **Si** q_1 est vrai
et q_2 est vrai
et .....
et q_m est vrai

b) Pour le cas où m=0, l'assertion p peut être énoncée de la manière suivante: Pour toutes valeurs prises par les variables X apparaissant dans p, p est vrai. p est une assertion inconditionnelle. Ce qui correspond à l'affirmation que le domaine de ces variables est tout l'Univers de Herbrand de l'ensemble des clauses.

c) Une question $q_1, \ldots, q_n$ est interprétée de la manière suivante: Il existe $X_1, X_2, \ldots$ (variables apparaissant dans la question), telle que la conjonction $q_1 \wedge \ldots \wedge q_n$ est vraie, c'est-à-dire est une conséquence logique du programme.

Pour considérer PROLOG comme un langage de programmation, on interprète les clauses de Horn d'une manière procédurale .

3.2. Sémantique procédurale.

Du point de vue opérationnel, une résolution de problèmes en PROLOG apparaît comme un enchaînement ordonné d'appels de procédures. Cette interprétation procédurale a été introduite par Kowalski *[Kowalski74]*. L'unification est le mécanisme gérant l'appel de procédure par la possiblité ou non de sa réalisation. Si l'unification réussit, il y a possibilité d'appel de la procédure, sinon la procédure ne peut pas être appelée, il y a incompatibilité entre paramètres d'appel et paramètres de définition. La résolution est le mécanisme gérant l'enchaînement des appels.

Une clause de Horn

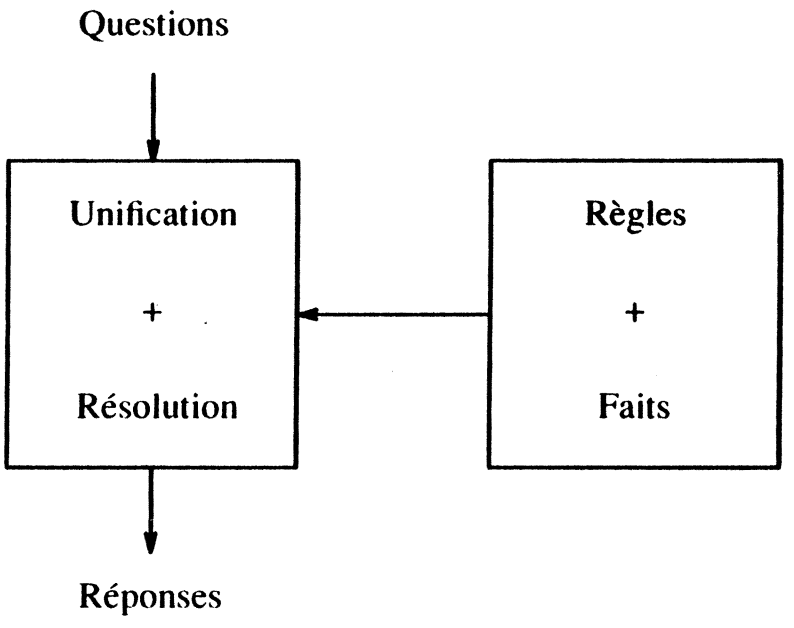
$$B_0 \leftarrow B_1, \ldots, B_n$$

est interprétée comme une déclaration de procédure "B_0" et dont le corps est formé de la conjonction des "B_i". Chaque littéral B_i $i=1,m$, qui a la forme $p(t_1,\ldots,t_n)$, est interprété comme un appel de procédure de nom p avec les arguments $(t_1, \ldots, t_n)$ où les variables sont instanciées par les substitutions rencontrées jusqu'à B_{j-1}.

Dans le cas où $n=0$, la déclaration de procédure a un corps vide et est interprétée comme une assertion. Les variables apparaissant dans de telles assertions peuvent être instanciées par tout élément de leur domaine (l'Univers de Herbrand).

4. Principe de la résolution.

PROLOG met à la disposition de l'utilisateur un formalisme lui permettant d'exprimer ses connaissances sur l'application à traiter par des faits et des règles (clauses de Horn), ainsi que les problèmes qu'il désire résoudre par des questions. Cela signifie qu'il fournit des mécanismes puissants de manipulation de connaissances et de résolution de problèmes. Ces mécanismes sont l'unification *[Champeaux84]*, *[Fages84]*, et la résolution *[Changlee73]*, *[Robinson65]*. Ils constituent le "*moteur d'inférence*" de PROLOG.



L'unification à elle seule remplace de nombreux mécanismes tels que: l'affectation, l'instanciation, le passage de paramètres dans les appels de procédure, la sélection des parties des objets structurés nécessaires à une construction ou à une destruction d'un terme, la construction de nouveaux objets structurés, etc. Elle travaille sur des termes et peut être vue comme un opérateur à deux arguments: les deux termes à unifier.

4.1. Substitution.

Définition.

Une substitution est une application qui est l'identité presque partout sauf en un nombre fini de points $\{<x_1, t_1>, ..., <x_n, t_n>\}$. Elle est définie en ces points par:

$$\sigma : X \rightarrow T$$

où X représente un ensemble de variables et T représente un ensemble de termes.

Définition.

L'instanciation t' d'un terme t est définie comme étant l'application d'une substitution σ sur le terme t. On remplace chaque variable x_i de $<x_i, t_i>$ par le terme t_i qui lui correspond. $t' = \sigma(t)$. On peut la définir par l'application suivante:

$$\bar{\sigma} : T \rightarrow T$$
$$\bar{\sigma}(f(t_1,....t_n)) \rightarrow f(\bar{\sigma}(t_1), ..., \bar{\sigma}(t_n))$$
$$\bar{\sigma}(a) = a$$
$$\bar{\sigma}(X) = \sigma(X) \quad \text{où X est une variable}$$

En termes de domaines, une substitution constante consiste à associer à la variable x_i un terme constant appartenant au domaine de la variable. Si ce terme contient des variables il représente un sous-ensemble de l'Univers de Herbrand, obtenu en substituant ces variables par de nouveaux termes.

Exemple: Soit $t = f(X,Y)$

$$\text{avec } \sigma = \{ <X, a> , <Y, Z> \}$$
$$\sigma(f(X,Y)) = f(a,Z)$$

La composition de substitutions est elle même une substitution; l'opération de composition des substitutions est associative et possède un élément neutre, la substitution vide notée $\{\}$.

4.2. Unification

L'opération de base en PROLOG est l'unification. Elle apparaît comme la construction (éventuellement impossible) d'un terme qui est l'instance commune la plus générale aux 2 termes t_1 et t_2 à unifier *[Colmerauer83]*. Les algorithmes d'unification consistent à trouver un ensemble d'éléments de substitutions qui s'appliquent aux 2 termes t_1 et t_2 pour en déduire le terme commun t. Cela revient donc à la résolution d'un système d'équations sur les sous-termes qui composent les 2 termes à unifier t_1 et t_2. Cette résolution n'est pas toujours possible; dans ce cas on dit que l'unification est impossible, il n'y a pas de terme commun aux 2 termes à unifier.

Définitions.

* 2 termes t_1 et t_2 sont unifiables s'il existe une substitution σ telle que $\sigma(t_1) = \sigma(t_2)$.

* Un ensemble de termes $S = \{t_1, ... t_n\}$ est unifiable s'il existe une substitution σ telle que $\sigma(S)$ soit un singleton, c'est-à-dire:

$$\sigma(\{ t_1, ..., t_n\}) = \{ \sigma(t_1), ..., \sigma(t_n)\} = \{\sigma(t_i)\}$$

* Un ensemble fini de termes, s'il est unifiable, possède plusieurs unificateurs (en fait une infinité), comme le montre l'exemple suivant.

Exemple:

Soit $S = \{X, Y, f(Z)\}$ où X, Y et Z sont des variables.
S admet comme unificateur $\sigma_1 = \{<X, f(a)>, <Y, f(a)>, <Z, a>\}$
et donc $\sigma_1(S) = \{f(a)\}$,
mais il admet aussi $\sigma_2 = \{<X, f(g(a))>, <Y, f(g(a))>, <Z, g(a)>\}$
et $\sigma_2(S) = \{f(g(a))\}$,

....

$\sigma_n(S)) = \{f(f(f(...(f(g(a))...)$

* Soit U(S) l'ensemble des unificateurs de S; on munit cet ensemble de la relation *"est plus général que"* notée < et définie par:

$$\sigma_1 < \sigma_2 \quad \text{s'il existe } \theta \text{ telle que la composition } \theta.\sigma_1 = \sigma_2$$

La relation < possède la propriété d'être un préordre sur l'ensemble U(S). L'équivalence de 2 éléments de U(S) est définie classiquement par:

$$\sigma \equiv \sigma' \quad \text{ssi } \sigma < \sigma' \text{ et } \sigma' < \sigma$$

Elle correspond à l'identité, à un renommage des variables près. On peut montrer que U(S)/≡ admet un élément minimal appelé *mgu* pour "*most greater unificator*" (ou "*pguc*" pour "*plus grand unificateur commun*"). Tout unificateur peut être obtenu par composition du *pguc* avec une substitution.

Pour des raisons d'efficacité, plusieurs algorithmes d'unification sont proposés dans la littérature *[Huet76], [Martelli82], [Paterson78].*

Remarque:

Dans les algorithmes d'unification utilisés dans les interpréteurs PROLOG, pour une substitution $\sigma = \{ \langle x_i, t_i \rangle \}$ on suppose que t_i ne contient pas la variable x_i. La plupart des implémentations ne font pas ce test, connu sous le nom de test d'occurrence (occur-check) qui a un coût exponentiel.

Exemple:

Les termes X et f(X) ne sont pas unifiables mais en l'absence du test d'occurrence, un algorithme d'unification produit le résultat f^{∞}, solution de l'équation $x = f(x)$.

Des études complètes des solutions infinies de telles unifications ont été effectuées. Certains interpréteurs offrent la possibilité d'activer ce test sur simple demande. Il est proposé de manière optionnelle dans l'interpréteur PROLOG II de Marseille *[Colmerauer82a], [VanCaneghem84].* Il

existe une option qui permet de choisir entre l'unification classique et l'unification sur les arbres infinis, mais au prix d'un écart de coût de 20 à 40 % *[Van Caneghem86]*, ce qui est important et permet de justifier l'option de l'unification sans arbres infinis (sans test d'occurrence), sauf quand c'est vraiment nécessaire.

4.3. Arbre et/ou.

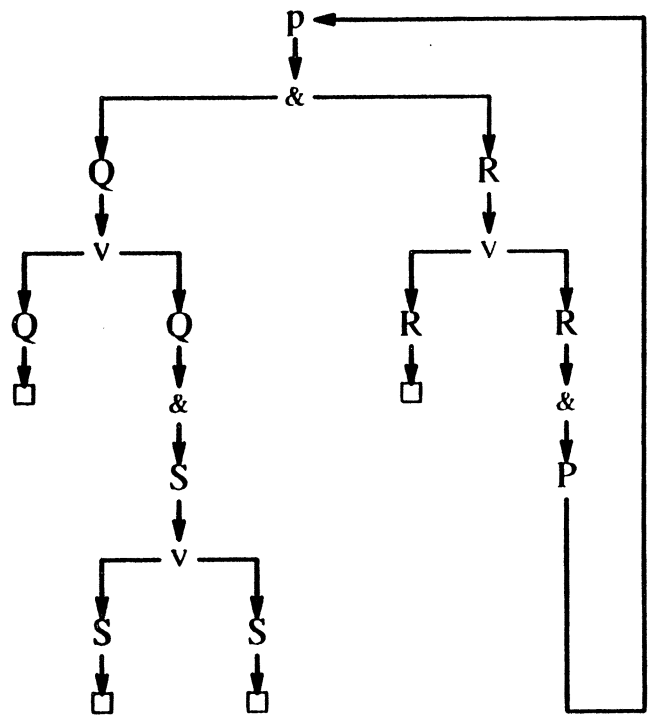
Le principe de résolution de PROLOG est une application particulière du principe de résolution de Robinson *[Robinson65]*, *[Robinson79]* dans le cadre de clauses de Horn *[Changlee73]*. Il correspond à la construction et au parcours d'un arbre et/ou éventuellement infini, défini statiquement par les clauses du programme *[Kowalski79]*.

Exemple:

Soit le programme suivant:

```
P:-Q,R.

Q.
Q:-S.

R.
R:-P.

S.
S.

? P.
```

Le graphe et/ou (correspondant à l'arbre et/ou en fusionnant les noeuds étiquettés par le même but) de cet exemple est donné par:



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

- Un noeud "et" (noté &) est considéré comme vrai si tous ses fils le sont.

- Un noeud "ou" (noté v) est considéré comme vrai si au moins l'un de ses fils l'est.

Les arcs des noeuds "v" correspondent aux substitutions réalisées pour passer d'un noeud "v" à un noeud "&".

4.4. Stratégies de résolution.

Le mécanisme de résolution effectue un parcours de cet arbre. Il y a plusieurs manières de l'explorer, ce qui correspond à plusieurs stratégies de résolution. La gestion des noeuds "et" de l'arbre caractérise le principe de résolution alors que la gestion des noeuds "ou" caractérise l'espace de recherche.

Les stratégies généralement utilisées sont le plus souvent basées sur un effacement séquentiel dans l'ordre d'écriture des sous-buts des corps des clauses dans le programme source. Les choix restants se situent au niveau des branches "ou": choisir la clause sélectionnée parmi celles possibles.

4.4.1. Recherche en profondeur d'abord:

L'arbre et/ou est parcouru en profondeur en prenant une seule clause à la fois, ce qui correspond au parcours d'une branche. Si la résolution n'aboutit pas à une solution (ou si on veut avoir une autre solution), une autre branche est parcourue en partant du point de choix le plus récent. En général, le choix des clauses s'effectue dans l'ordre de leur écriture. Une autre solution consiste à choisir en priorité "les faits" (ils sont les seuls à pouvoir effacer un littéral).

4.4.2. Recherche en largeur d'abord:

On développe en même temps toutes les branches "ou" d'un niveau de l'arbre et/ou. L'arbre est donc parcouru en largeur, un niveau de l'arbre étant parcouru avant de passer à un niveau suivant.

Le coût exponentiel de cette dernière stratégie laisse entendre que la première est la plus utilisée (depht-first), malgré l'incomplétude causée par les appels récursifs à gauche (risque de boucle lors de tels appels).

D'autres stratégies consistent à effectuer des choix en utilisant certaines informations supplémentaires permettant d'accélérer l'effacement: modes *[Oudot87]*, probabilité d'unification des buts *[Naish85]*, choix des faits en priorité.

4.5. Retour en arrière.

L'utilisation la plus générale de PROLOG (avec la stratégie de recherche en profondeur), s'intéresse à l'ensemble des solutions pour lesquelles un but d'un programme est vrai. Ceci conduit à fournir un moyen pour parcourir tout l'arbre et/ou, soit au moment d'un échec sur une branche, soit pour avoir une autre solution lors d'un succès. C'est le "*retour en arrière*". Il existe différentes approches pour effectuer ce retour en arrière, telle que le retour en arrière "*chronologique*" (retour au point de choix le plus récent) propre à PROLOG, et les retours en arrière dits *"intelligents" [Bruynooghe83], [Pereira81], [Pereira83],*

Ces retours en arrière peuvent être contrôlés par l'utilisation d'un prédicat extra-logique prédéfini noté "/" (slash) ou "!" (cut). Il est utilisé pour éliminer les points de choix restants dans un retour en arrière chronologique. En général il supprime les solutions indésirables, évite des traitements inutiles, aide à déterminer la 1ère solution seulement. A la limite, l'usage de ce prédicat permet de rendre le programme entièrement déterministe. Cet opérateur peut être avantageux car l'interprète ne détecte pas les boucles infinies; cependant son usage peut conduire facilement à des abus. D'autre part il supprime l'aspect déclaratif du langage.

5. Mise en oeuvre de la résolution en prolog

La résolution en PROLOG a été présentée dans la littérature sous différents aspects: un système de réécriture, un mécanisme d'effacement *[Colmerauer83]*, une suite d'appels de procédures Pascal (ou Algol) *[Nilsson83]*. Nous la présenterons comme une suite de restrictions sur les domaines des variables. Initialement, le domaine d'une variable est l'Univers de Herbrand dans sa totalité.

Dans l'interprétation procédurale, un ensemble de déclarations de procédures est un programme. L'exécution est initialisée par une conjonction de buts initiaux dont les variables ont pour domaine a priori tout l'Univers de Herbrand; elle procède à des suites d'appels en utilisant des déclarations de procédures et dérive un nouveau but en effectuant des restrictions sur les domaines des variables. Ces restrictions sont effectuées en fonction des substitutions réalisées. En unifiant par exemple une variable ayant comme domaine l'Univers de Herbrand dans sa totalité, avec un terme f(...), nous éliminons dès à présent du domaine de la variable tous les termes constants appartenant à l'Univers de Herbrand mais n'ayant pas la forme f(...) (structure de mêmes foncteur et arité).

La résolution se termine lors de la dérivation de la clause vide □ (aucun appel de procédure). A l'expression d'arrêt, la solution est formée des valeurs prises par les variables du but initial, et calculées à partir de la composition des substitutions rencontrées au cours de la résolution. En terme de domaines, les domaines des solutions sont les domaines de ces variables obtenus après les restrictions successives, effectuées au cours de la résolution.

5.1. Stratégie depth-first de PROLOG.

L'exécution d'un programme PROLOG est une réfutation (preuve par l'absurde) utilisant la résolution comme règle d'inférence. En termes de systèmes experts, la stratégie globale de PROLOG est une stratégie de chaînage arrière *[Changlee73]*, *[Nilsson82]*: on part de la question et on essaye de dériver la clause vide □. En termes de domaines, nous présentons la résolution comme une suite de restrictions sur les domaines des arguments.

Nous explicitons la résolution sur une étape: Une résolvante initiale est constituée par la question. Soit la question $A_1,....A_{n_0}$, et donc:

$$R_0 = A_1 , ... , A_{n_0}$$

Soit $\{t_1, ... t_m\}$ l'ensemble des arguments apparaissant dans la résolvante R_0. Le domaine associé à un atome est constitué de l'atome lui-même. Le domaine associé à chaque variable de l'étape initiale de la résolvante est l'Univers de Herbrand dans sa totalité que nous représentons par H. A chaque argument t_i est associé donc un domaine D_i:

$$\{ (t_1, D_1), ..., (t_m, D_m) \} \qquad (1)$$

où (t_i, D_i) signifie que D_i est le domaine de l'argument t_i. (Connaissant le domaine d'un argument (un terme) on peut déterminer les domaines de ses variables éventuelles).

Nous sélectionnons parmi les A_i un but A_k, contenant les arguments $\{t_1, ... t_n\}$ avec des domaines $\{(t_1, D_1), ... (t_n, D_n)\}$ et une clause $C_0 :- B_1, ... B_s$ du programme telle que A_k et C_0 soient unifiables. (on renomme les variables de C_0).

Désignons le domaine de définition de A_k par le n-uplet $(D_1, ..., D_n)$ et le domaine d'appel de C_0 par $(D'_1, ... D'_n)$. La possibilité d'unification implique que pour chaque argument, l'intersection entre le domaine de définition et le domaine d'appel n'est pas vide: $D_i \cap D'_i \neq \varnothing$ pour tout $i=1,n$. Lorsqu'une variable ayant comme domaine d'appel D est unifié avec un terme ayant comme domaine D', alors après unification le domaine de la variable (dans le résolvante obtenue) est donné par $D \cap D'$.

Après unification, le domaine de l'argument t_i devient $D_i \cap D'_i$, où D_i est le domaine de l'argument t_i dans la résolvante initiale et D'_i est le domaine de définition de l'argument unifié avec l'argument t_i dans la clause C_0.

Soit σ le *pguc* de $A_k$et de C_0. En appliquant la substitution σ à la résolvante R_0, la nouvelle résolvante est:

$$R_1 = \sigma(A_1, ... A_{k-1}, B_1, ... B_s, A_{k+1}, ... A_n)$$

Pour le passage de R_0 à R_1, les domaines des arguments deviennent alors:

$$\{ (t_1, D_1 \cap D'_1), (t_2, D_2 \cap D'_2), \ldots, (t_n, D_n \cap D'_n), \ldots \}$$

Nous avons effectué l'unification du but A_k avec sa 1ère définition. D'autres clauses peuvent éventuellement s'unifier avec A_k. Soit C_1:-... une clause telle C_1 et A_k soient unifiables. Désignons le domaine de définition de la tête de cette clause par $(D''_1, \ldots, D''_n)$. La possibilité d'unification implique aussi que pour chaque argument, l'intersection entre le domaine de définition et le domaine d'appel n'est pas vide: $D_i \cap D''_i \neq \emptyset$ pour tout i=1,*n*. Le domaine de l'argument t_i devient alors $D_i \cap D''_i$ pour i=1,n.

L'unification de A_k avec C_0 (ou C_1 respectivement) a restreint le domaine de l'argument t_i à $D_1 \cap D'_i$ (ou $D_i \cap D''_i$ respectivement). En prenant la réunion des deux possibilités, nous trouvons que le domaine de l'argument t_i est restreint à $D_i \cap D'_i \cup D_i \cap D''_i = D_i \cap \{ D'_i \cup D''_i \}$. Pour utiliser Prolog dans le cadre le plus général (recherche de toutes les solutions) il faut examiner toutes les unification possibles du but A_k. Le domaine de l'argument t_i (pour toutes les unifications possibles du but A_k) est donc $D_i \cap \{ D_{i0} \cup D_{i1} \cup \ldots \cup D_{il} \}$ où D_{ij} est le domaine d'appel de l'argument i dans la définition j.

Une réponse est obtenue quand la résolvante R_f (résolvante finale) est la séquence □, auquel cas la réponse est la composition des substitutions obtenues le long du parcours de R_0 à R_f. En termes de domaines, la solution appartient aux domaines des arguments $t_1, \ldots, t_m$, obtenus par les restrictions successives.

A chaque argument t_i correspond donc un domaine D_f obtenu à partir du domaine initial D_i en effectuant des restrictions sur ce dernier. Connaissant le domaine d'un argument t_i nous pouvons déterminer les domaines de ses variables éventuelles. La valeur prise par une variable de la résolvante initiale lors de son succès appartient au domaine de cette variable.

Exemple:

pere(a, b)
pere(c, d)
mere(b, e)
mere(d, f)
gparent(X, Y):-pere(X, Z), mere(Z, Y)

L'Univers de Herbrand pour cet ensemble de clause est: H={a, b, c, d, e, f}

Soit la résolvante initiale: R = gparent(A, B) avec comme domaine pour les arguments A et B:

$$D_R = \{ (A, H), (B, H) \}$$

Par application de la substitution σ = {<A, X>, <B, Y> nous obtenons la résolvante:

$$R_1 = \text{pere}(A, Z), \text{mere}(Z, B) \quad \text{avec } D_1 = \{(A, H \cap H), (B, H \cap H)\}$$
$$= \{ (A, H), (B, H) \}$$

Pour le 1er but, on a deux possibilités d'unification:

1)σ = {<A,a>, <Z,b>} donne R_{21}= mere(b, B) et $D_{21} = \{ (A, H \cap \{a\}), (B, H)\}$
$= \{ (A, \{ a \}), (B, H)\}$

2)σ = {<A,c>, <Z,d>} donne R_{22}= mere(d, B) et $D_{22} = \{ (A, H \cap \{c\}), (B, H) \}$
$= \{ (A, \{ c \}), (B, H)\}$

$$D_2 = \{ (A, \{a, c\}), (B, H) \}$$

Pour la résolvante R_{21} = mere(b, B) (respectivement R_{22} = mere(d, B)) nous n'avons qu'une seule possibilité d'unification (21) (respectivement (22)) avec la substitution <B, e> (respectivement (<B, f>):

$21)\sigma = \{<B/e>\}$ $R_{211} = \square$ avec $D_{sol1} = \{ (A, \{a\}), (B, H \cap \{e\}) \}$
$= \{ (A, \{a\}), (B, \{e\}) \}$

$22)\sigma = \{<B/f>\}$ $R_{221} = \square$ avec $D_{sol2} = \{ (A, \{c\}), (B, H \cap \{f\}) \}$
$= \{ (A, \{c\}), (B, \{f\}) \}$

d'où $D_{sol} = \{ (A, \{a, c\}), (B, \{e, f\}) \}$

qui représente le domaine des solutions.

Le domaine des solutions est un sous-ensemble de l'intersection des domaines de définition et des domaines d'appel.

Si nous considérons la résolution en terme de domaines, nous remarquons donc qu'au fur et à mesure de l'avancement de la résolution, nous faisons des restrictions sur les domaines des arguments. A l'étape initiale de la résolution, le domaine d'une variable est l'univers de Herbrand. D'une étape à une autre, cet espace est restreint, d'où notre approche qui a consisté à exprimer la résolution comme une suite de restrictions sur les domaines des arguments.

5.2. Représentation des substitutions et Déliaison.

Dans les descriptions précédentes de l'unification et de la résolution, nous avons négligé les problèmes posés par la gestion des variables logiques apparaissant dans les clauses.

Lors d'une unification réussie, il est nécessaire de propager les substitutions qu'elle définit à l'ensemble des variables apparaissant dans les autres buts.

Lors d'un retour en arrière (backtracking), il faut effectuer une déliaison des liaisons faites depuis ce point de choix.

En PROLOG on utilise la technique de copie *[Mellish82]* (ou partage *[Boyer72]*) de structure pour résoudre ces problèmes et pouvoir effectuer facilement les liaisons et les déliaisons. On utilise des piles pour stocker les informations nécessaires (environnement) pour ces liaisons et ces déliaisons: pile locale, pile globale, pile trainée.

Dans le calcul des types que nous proposerons, pour éviter de lier les variables logiques, les liaisons sont réalisées par des simulations de l'unification. Nous n'effectuons pas de déliaison car la simulation ne lie définitivement aucune valeur à une variable logique. L'unification d'un argument t_1 d'un appel avec un argument t_2 d'une définition sera simulée par une inéquation $t_1 \leq t_2$.

6. Pourquoi les types en Prolog ?.

PROLOG est désigné comme un langage à un seul type: *l'Univers de Herbrand.* Il est considéré comme un langage de très haut niveau par sa puissance d'expression, sa lecture déclarative, sa facilité de créer et de manipuler des structures de données. Cependant sa syntaxe est de faible niveau comparée aux pratiques courantes des langages classiques. Des erreurs typographiques simples comme par exemple une erreur sur le nom d'une variable, le nom d'une procédure ou d'un foncteur, l'oubli d'un argument (c'est-à-dire l'appel d'un prédicat d'arité différente), ..., ne produisent pas d'erreurs de compilation ou d'exécution. Elles donnent simplement des résultats qui n'ont pas le sens espéré. La raison est l'absence totale de redondance dans les programmes. Les définitions de procédures ne sont pas introduites par des déclarations définissant le nombre et le type de ses paramètres; les variables ne sont pas déclarées.

Du point de vue de l'exécution, les buts qui échouent toujours et qui sont mal typés ne sont pas différenciés de ceux qui échouent parce que le programme ne contient pas de données appropriées.

Les bénéfices de l'introduction des types pour la détection des erreurs en compilation et pour la vérification des programmes sont bien connus *[Bruynooghe82], [Gang86], [Mishra84], [Mycroft84], ...* En Programmation logique et en démonstarction automatique il a été dit en de nombreuses occasions *[Chassin86], [Oudot87], [Turbo86], ...* que le fait de contraindre ou de typer les variables logiques favorise l'obtention plus rapide d'un programme correct. Nous verrons au prochain chapitre qu'il y a deux approches pour introduire la notion de type en Prolog:

* L'approche déclarative, analogue aux langages classiques.
* L'approche inférentielle, où il s'agit d'extraire automatiquement les types des prédicats.

Les types peuvent avoir un rôle naturel et utile en Prolog. La sémantique procédurale d'un programme syntaxiquement correct est totalement définie. Si l'on fait abstraction des prédicats prédéfinis, il est impossible qu'une condition d'erreur arrive ou qu'une opération indéfinie se produise. Comme une valeur normale (correcte) ne peut pas être distinguée

d'une valeur erronée, les erreurs sont extrêmement difficile à détecter et à comprendre. Ceci est particulièrement le cas lorsqu'une simple erreur de transposition des arguments s'est produite.

Exemple: Le prédicat rev est supposé inverser une liste.

```
rev([], []).
rev([A|B], C) :- rev(B, D), append(D, A, C).
append([], K, K).
append([X|L], M, [X|N]) :- append(L, M, N).
```

Ce programme contient une erreur dans la seconde clause: il y a append(D,A,C) au lieu de append(D,[A],C).

La réponse à la question rev([a, b], [b, a]) échoue, alors que ce littéral appartient à la dénotation attendue du prédicat "rev". On essaye de comprendre la raison de l'échec. Vient-elle de la définition de rev? Ou de celle de append ? Ou des deux ?.

En inférant les types des prédicats append et rev, le programmeur remarque que le type inféré de append donné par ([] V r!([any|[]]+r]), any V r!(any+r), any V r!([any|any+r]) est correct alors que le type de rev donné par ([] V r!([any|[]]+r]), [] V any) ne correspond pas à celui espéré. En analysant les types du prédicat rev (voir le prédicat revbad dans l'annexe 2) il déduit que l'erreur est dans sa seconde clause. Il peut alors focaliser son attention sur cette clause au lieu des quatre.

Exemple: Ce prédicat est supposé calculer la longueur d'une liste.

```
longfaux([], 0).
longfaux([A|B], N) :- longfaux(N1, B), N is N1 + 1.
```

Ici le programmeur a fait une erreur de transposition d'arguments dans le premier but de la seconde clause (longfaux(N1,B) au lieu de longfaux(B,N1)).

Grâce au type prédéfini is(any, any), on détecte l'erreur de transposition. En analysant le type inféré donné par ([] V ∅, 0 V ∅), nous remarquons que le type inféré de la seconde clause ne correspond pas aux résultats espérés par le programmeur.

Dans un programme Prolog avec un espace de recherche très grand, une simple erreur peut mener à des boucles.

L'inférence de type détecte les clauses qui ne réussissent jamais. Elle permet aussi de déterminer si tous les cas d'un programme ont été considérés ou non.

Exemple

```
append([X|L], M, [X|N]):-append(L, M, N).
```

L'inférence du type du prédicat append donne (r![anyIr]), r!(r), r![anyIr]). Elle nous informe que cette définition est incomplète.

Du point de vue du génie logiciel, le langage Prolog manque de facilités permettant un développement structuré et controlé de gros programmes par plusieurs programmeurs. Il est nécessaire d'introduire de telles facilités sans perdre en efficacité de prototypage.

En particulier, on n'a pas la possibilité de spécifier formellement les types des arguments d'un prédicat. De telles informations sont utiles pour plusieurs raisons: Elles fournissent un bon moyen de documentation; elles peuvent, dans un système à module, être une partie de la description de l'interface d'un module. Elles permettent d'améliorer la lisibilité du programme.

Les types produisent aussi des informations utiles non seulement au programmeur mais à un outil d'analyse chargé d'inférer des caractéristiques de l'exécution à partir du texte du programme. Ces informations de type peuvent être utilisées pour le déboguage en détectant les différences entre le modéle pensé par le programmeur et les types inférés par l'outil.

Ces types inférés nous donnent aussi les contraintes de typage des variables initiales d'une question pour que la question réussisse. Les types forment donc une prédiction de la compilation des valeurs de type qu'une expression peut avoir pendant les exécutions.

Il existe une occasion où le typage des variables permet d'augmenter

l'efficacité de la résolution: la compilation. On a besoin de restreindre les possibilités d'unification des paramètres de certaines clauses. On peut indexer l'appel de certaines clauses moyennant la connaissance des domaines de ses arguments. A l'appel d'une clause à deux définitions par exemple ayant comme domaine pour le premier argument respectivement f(...) et g(...), on peut indexer l'appel de la clause suivant le terme d'appel de ce premier argument. Si l'appel se fait avec un terme de foncteur f alors on appelle la première définition, si l'appel se fait avec le foncteur g alors on appelle la seconde définition. Cette indexation nous permet de faire un choix rapide de la clause à envisager. Ce qui est le rôle des types.

En conclusion il est clair qu'un tel typage doit rester optionnel ou automatique et ne pas devenir contraignant, privant ainsi l'utilisateur de la souplesse de Prolog. Ceci peut se réaliser d'une manière purement syntaxique comme nous le verrons au second chapitre, mais la discrimination sur des critères sémantiques comme nous le verrons dans le 3ème chapitre lors de notre proposition, serait bien utile et plus intéressante également.

Nous donnerons au dernier chapitre quelques applications des types pour des vérifications et/ou des optimisations. Des exemples d'inférence par notre système que nous proposerons sont donnés dans l'annexe 2.

CHAPITRE 2

LES TYPES EN PROLOG

1. Introduction

La notion de type a différentes acceptions dans les langages de programmation *[Cardelli85]*. Une caractéristique commune est de servir à des calculs de propriétés de programmes (vérifications), d'être exploitée à la compilation (contrôles, optimisations,..) ou à l'exécution.

Cette notion de type n'existe pas a priori en PROLOG où un seul domaine est considéré: *l'Univers de Herbrand.* Plusieurs auteurs ont ressenti le besoin de caractériser plus finement les domaines des valeurs prises par les variables. Il s'agit toujours de préciser des sous-ensembles de l'Univers de Herbrand, généralement au moyen d'une grammaire de termes.

La vérification minimale de cohérence des types consiste à vérifier qu'il y a au moins une clause où les intersections des domaines de définition avec les domaines d'appel des arguments ne sont pas vides, garantissant ainsi la possibilité de l'unification; c'est ce qu'on appelle la vérification de *bon-typage.*

Une autre information qui peut être utilisée pour la vérification de la cohérence du programme est la déclaration des *modèles d'appel* (*modes*), c'est-à-dire la description de l'état d'instanciation des sous-buts au moment de leur appel. Nous ne nous intéressons pas à ce second aspect dans cette étude.

On peut diviser en deux classes les premiers travaux (à notre connaissance) effectués sur les types en PROLOG:

* L'approche déclarative *[Simonet81]*, *[Mycroft82]*, *[Bruynooghe82]*, *[Nilsson83]*, *[Azzoune85]*, *[Gang86]* où la déclaration des domaines est fournie explicitement par le programmeur. Le but poursuivi est la vérification que les variables prennent leurs valeurs dans les domaines (types) espérés par le programmeur. Nous détaillerons le travail de Mycroft et O'Keefe *[Mycroft82]*. Des résumés des autres travaux seront présentés.

* L'approche inférentielle *[Mishra84]*, *[Kanamori85]*, *[Zobel87]*, *[Kluzniak87]* où les domaines sont déduits "automatiquement" à partir des clauses du programme. Le but poursuivi est l'obtention d'informations sur les valeurs possibles des variables pour permettre d'effectuer des vérifications et/ optimisations. Nous détaillerons le système de *[Mishra84]*. Des résumés des autres travaux seront présentés. Nous proposerons aux chapitres suivants une méthode d'inférence de type ainsi que quelques applications des résultats inférés.

Dans l'approche déclarative, le programme est accompagné de déclarations explicites des types des variables, des foncteurs et des prédicats (et éventuellement les modèles d'appel (modes)) pour chaque clause. Ces déclarations expriment les contraintes imposées par le programmeur sur les valeurs possibles des arguments. La vérification de *bon-typage* est faite par calcul d'intersections des types. On déclare manuellement puis on vérifie automatiquement. Le slogan de typage est "*well typed programs do not go wrong*" *[Mycroft82]*.

Dans l'approche inférentielle, les types des variables sont déduits "automatiquement" à partir du programme, sans aucune déclaration, ou moyennant des déclarations simples. Cette déduction automatique est indépendante des intentions du programmeur; Les types expriment et décrivent les exécutions possibles du programme et leurs résultats et non les contraintes exprimées par le programmeur, comme c'est le cas dans l'approche déclarative. La notion précédente de bon-typage d'un programme est validée implicitement. Le slogan de typage est: "*ill typed programs always fails*" *[Mishra84]*.

2. Vérifier les types [Mycroft82].

Les travaux à déclarations explicites que nous avons rencontré le long de notre étude, manipulent la même notion de type. Ils exigent des déclarations explicites des types comme dans les langages classiques mais sous des formalismes différents.

Comme nous l'avons fait remarquer, la seule caractérisation du type d'une variable en PROLOG standard est que celui-ci est un sous-ensemble de l'Univers de Herbrand. Définir le type d'un argument, consiste à définir un sous-ensemble de l'Univers de Herbrand où cet argument peut prendre ses valeurs. Il s'agit donc de caractériser des restrictions sur l'Univers de Herbrand.

Inspirés des travaux de Milner sur le contrôle de type dans le langage ML *[Milner78]*, Mycroft et O'keefe *[Mycroft82]* proposent un schéma de types polymorphes défini par une grammaire, la même que celle utilisée en ML *[Gordon78]*. Les déclarations explicites des types accompagnent le programme formant ainsi un méta-langage. L'interpréteur ou le compilateur peut ignorer ces déclarations, avec la garantie qu'un programme bien-typé donnerait des résultats identiques avec ou sans contrôle de type.

2.1. Représentation des types.

Les formalismes de représentation des types permettent d'exprimer les restrictions sur l'Univers de Herbrand. Ils sont généralement dérivés de la notion de grammaire de termes.

Mycroft et O'keefe *[Mycroft82]* ont choisi une représentation définissant le polymorphisme par le biais de variables de type. Ce schéma de type est défini par la grammaire suivante:

Type ::= Tvar / Tcons(Type*)

Déclaration ::= **type** Tcons(Tvar*) = Functor(Type*)* / **pred** Pred(Type*)

où Tvar est à un ensemble de variables (variables de type), Tcons est à un ensemble de constantes et de foncteurs (constructeurs de type), Functor est un foncteur et Pred est un prédicat.

Dans cette définition, on peut avoir des types polymorphes dit *polytypes* dûs à l'existence de variables de type (définitions génériques), par exemple:

type liste(T) = nil / [T | liste(T)]

et des types dits *monotypes* dûs à l'absence de variables de type comme par exemple:

type couleur = blanc / rouge / vert

type entier = 0 / suc(entier)

Ce schéma (polytypes) forme des types génériques dont une instance est obtenue en remplaçant les variables de type par des types prédéfinis ou définis, comme par exemple liste(couleur), liste(liste(T)).

Notons que le même foncteur peut appartenir à plusieurs types. Le type des arguments d'un prédicat est déclaré par le programmeur:

pred append(liste(T), liste(T), liste(T)).

pred rev(liste(T), liste(T)).

2.2. Typer une clause

Typer un programme revient à typer chacune de ses clauses. Pour typer une clause à partir des déclarations, le système doit déterminer le type des variables, des foncteurs et des prédicats y apparaissant. Pour désigner les types associés aux composantes d'une clause, les notations suivantes sont utilisées:

* Une variable X de type τ est notée X^{τ}.

* Un foncteur f d'arité n, est noté:

$$f^{\alpha_1.\alpha_2. \ldots . \alpha_n \rightarrow \tau}$$

où les α_i sont les types des arguments de f, et τ est le type du terme $f(t_1,\ldots,t_n)$.

* Un prédicat p d'arité n, est noté:

$$p^{\sigma_1, \sigma_2, \ldots, \sigma_n}$$

où σ_i est le type de l'argument i.

Exemple: Soit la clause suivante:

append([A|L], M, [A|N]):-append(L, M, N).

où append est déclaré en: **pred** append(liste(T), liste(T), liste(T))

Le calcul des types associés aux composantes de la clause fournit:

$$\{ A^{T}, L^{\tau}, M^{\tau}, N^{\tau}, \text{append}^{(\tau,\tau,\tau)}, |^{(T, \tau) \rightarrow \tau} \}$$

où τ désigne le type liste(T) = nil / [T | liste(T)].

2.3. Bon-typage

Mycroft et O'Keefe définissent la notion de "*bon-typage*" (*well-typing*) et montrent qu'un programme PROLOG "*bien-typé*" ne peut aboutir à un résultat erroné dont la cause de l'erreur soit le type. Cette notion de "*bon-typage*" exprime la cohérence du programme avec les déclarations de type. Ce test de "*bon-typage*" est effectué en vérifiant un ensemble de conditions qu'ils ont définies *[Mycroft82]* et que nous donnons ci-dessous.

Soit p l'ensemble des variables, des foncteurs et des prédicats apparaissant dans une clause q. Le "typant" (l'environnement de type) $\bar{p}$ de p est défini comme étant l'association d'un type à chaque symbole de q (tous les éléments de p) comme celles données ci-dessus. La clause typée $\bar{q}$ est obtenue en réecrivant la clause q mais en typant chacun de ses symboles par les types donnés dans $\bar{p}$. On dit que $\bar{q}$ est un "bon-typant" de q sous $\bar{p}$, notée $\bar{p} \vdash \bar{q}$ si les conditions suivantes sont vérifiées:

(1) $\bar{p} \vdash (A \leftarrow B_1,....,B_m)$ si
$A = a(t_1{}^{\tau}1, ... t_k{}^{\tau}k)$ et $a^{\rho} \in \bar{p}$
avec $(\tau_1, ...,\tau_k) \equiv \rho$
$\bar{p} \vdash t_i{}^{\tau}i \quad (1 \leq i \leq k)$ et $\bar{p} \vdash B_i \quad (1 \leq i \leq m)$

(2) $\bar{p} \vdash A$ si A est un littéral et
$A = a(t_1{}^{\tau}1, ... t_k{}^{\tau}k)$ et $a^{\rho} \in \bar{p}$
avec $(\tau_1, ...,\tau_k) \leq \rho$ et $\bar{p} \vdash t_i{}^{\tau}i \quad (1 \leq i \leq k)$

(3) $\bar{p} \vdash u^{\sigma}$ si u est un terme et
$u = f(t_1{}^{\tau}1,....t_k{}^{\tau}k)$ et $f^{\rho} \in \bar{p}$
avec $((\tau_1,....,\tau_k) \rightarrow \sigma) \leq \rho$ et $\bar{p} \vdash t_i{}^{\tau}i \quad (1 \leq i \leq k)$.

(4) $\bar{p} \vdash X^{\sigma}$ si $X^{\sigma} \in \bar{p}$

où $u \leq v$ signifie que u est une instance de v, et $u \equiv v$ signifie que $u \leq v$ et $v \leq u$.

Appliquons ces conditions à l'exemple de append. L'ensemble p correspondant est donné par $p = \{ A, L, M, N, append, I \}$. Le typant de append est donné par $\bar{p} = \{ A^T, L^{\tau}, M^{\tau}, N^{\tau}, append^{(\tau, \tau, \tau)}, I^{(T, \tau)\rightarrow\tau} \}$. La clause typée associée est donnée par $append([A^T \mid L^{\tau}]^{\tau}, M^{\tau}, [A^T \mid N^{\tau}]^{\tau})$:- $append(L^{\tau}, M^{\tau}, N^{\tau})$. Nous vérifions si la clause typée est bien-typée, c'est-à-dire si elle vérifie les conditions précédentes.

D'après la condition (1), la clause de append est bien-typée si:
* a) $append^{\rho} \in \bar{p}$ avec $(\tau, \tau, \tau) \equiv \rho$,
* b) $\bar{p} \vdash append(L, M, N)$ et $\bar{p} \vdash \{ [A^T \mid L^{\tau}], M^{\tau}, [A^T \mid N^{\tau}] \}$.

Pour la première condition (a) le triplet (τ, τ, τ) est identique à $\rho = (\tau, \tau, \tau)$, elle est donc vérifiée.

Pour ce qui est de la seconde condition (b) nous l'examinons en deux parties:
- b1) Pour le premier cas on utilise la règle (2). Nous vérifions la condition $\bar{p} \vdash append(L, M, N)$. Cette condition est vérifiée si $append^{\rho} \in \bar{p}$ avec $(\tau, \tau, \tau) \leq \rho$ et $\bar{p} \vdash \{L^{\tau}, M^{\tau}, N^{\tau}\}$. Pour sa première condition, elle est vérifiée puisque

$(\tau, \tau, \tau) \leq (\tau, \tau, \tau)$. Pour sa seconde condition nous l'examinons avec le reste de la condition (b) initiale, ce qui nous donnent comme condition à vérifier:

$$\tilde{p} \vdash \{ [A^T \mid L^\tau], M^\tau, [A^T \mid N^\tau], L, M^\tau, N^\tau \}.$$

- b2) Nous vérifions cette condition en utilisant les règles (3) et (4). Pour la première composante de cet ensemble, il faut que (en utilisant la condition (3)): $(T, \tau) \rightarrow \tau \leq (T,\tau) \rightarrow \tau$ condition qui est vérifiée. Pour le second élément de l'ensemble, nous utilisons la règle (4). L'expression $M^\tau \in \tilde{p}$. Même chose pour les autres éléments de l'ensemble, nous utilisons les règles (3) et (4).

La déclaration du type de append donnée par:

Pred append(Liste(T), Liste(T), Liste(T))

type Liste(T) = nil / [T | Liste(T)]

vérifie ces conditions, par conséquent la clause est bien-typée.

En terme de résolution, un programme est "*bien-typé*", si lors d'une étape de la résolution, en prenant une résolvante "*bien-typée*" alors elle donnera toujours comme résultat une résolvante "*bien-typée*". Si les symboles d'une résolvante, en particulier les variables, vérifient les déclarations de type alors elle est bien-typée. La notion de bon-typage liée à l'exécution d'un programme mène au slogan "*well-typed programs do not go wrong*": il y a compatibilité dans tous les appels du programme. La notion de "do not go wrong" signifie seulement que les résolvantes satisfont les restrictions de type mais ne donne aucune information sur le succès ou l'échec de la résolution: C'est un calcul syntaxique.

Exemple

eq(X, X).

avec la déclaration eq(integer, integer). La résolvante eq(a,a) est mal-typée mais elle donne une réponse affirmative. Avec la déclaration eq(atome, atome), la résolvante eq(a, b) est bien-typée mais la réponse est négative.

2.4. Autres travaux

2.4.1. Les types

D'après *[Deirtich88]*, un des inconvénients de la méthode de Mycroft et O'keefe est que les relations entre les types ne peuvent pas être spécifiées. Un type ne peut pas être défini comme une union de types ou un sous-type d'un autre type. Dietrich et Hagl *[Deitrich88]* ont proposé une extention du système de Mycroft et O'Keefe. Cette extension consiste à permettre une nouvelle notion: *la notion de sous-type*. Le but est d'améliorer le système de Mycroft et Ok'eefe, pour une plus grande précision dans le contrôle statique des types d'un programme.

Dans le système de Mycroft et O'keefe, pour une variable ayant plusieurs occurrences dans une clause, il faut que cette variable ait un type unique (unification des types de toutes les positions). Si une variable à double occurrence par exemple a deux types non unifiables en ses deux positions, on dit qu'elle est mal-typée, le programme est alors mal-typé alors qu'il peut réussir. Pour éviter ce cas d'imprécision du système de Mycroft et O'keefe, Deitrich et Hagl ont introduit cette notion de sous-type. La condition de bon-typage d'une variable garantie que dans une clause bien-typée le sens de l'instanciation d'une variable à plusieurs occurrences ayant des types différents en ses différentes positions doit se faire d'une occurrence d'une position ayant un "sous-type" à une position ayant un "sur-type" de ce sous-type. On utilise pour cela les relations entre les types et la connaissance des modes c'est-à-dire le sens d'instanciation.

Bruynooghe *[Bruynooghe82]* utilise les modèles d'appel avec des informations redondantes sur les valeurs possibles d'un argument pour des vérifications de cohérence entre ces déclarations et le texte du programme. Ces déclarations sont données sous forme d'ensembles ou de schémas de type et ont un double rôle:

* augmenter la lisibilité du programme en rendant plus explicite les domaines des arguments des différents prédicats.
* Permettre des vérifications statiques.

Il vérifie la compatibilité des types entre les différents appels en propageant les domaines des variables utilisant la notion de mode.

Nilsson *[Nilsson83]* traduit les programmes PROLOG en programmes Pascal *[Wirth73]*. Pour chaque clause il produit autant de procédures Pascal qu'il y a de modèles d'appel de sa tête. La notion de domaine est introduite pour obtenir une division de l'Univers de Herbrand des termes constants, et pour distinguer les différents usages des données dans le programme. Cette notion est utile aussi pour imposer des contraintes statiques sur le programme et obtenir leur équivalent en Pascal. Sa définition de domaine s'inspire de la notion d'algèbre de termes des types abstraits *[Gutag78]*, *[Goguen78]*.

Simonet *[Simonet81]* a défini la notion de *D.clauses de Horn*, en attachant un domaine à chaque variable dans une clause de Horn au moyen d'une bigrammaire régulière. Les D.clauses de Horn (dont les clauses de Horn sont un cas particulier où le domaine de chaque variable est l'Univers de Herbrand de l'ensemble des clauses considérées) sont équivalentes aux W.grammaires de termes.

Nous avons présenté *[Azzoune85]* une méthode de déclaration de type par des bigrammaires réguliers. Cette méthode utilise les algorithmes d'intersection, de réunion et d'équivalence des bilangages réguliers. Ces algorithmes sont décidables et sont donnés par Pair et Quéré *[Quéré68]*.

Y. Gang et X. Zhiliang *[Gang86]* ont proposé un système de type pour PROLOG. Les objectifs sont d'améliorer l'efficacité de l'unifcation, de réduire l'espace mémoire et de simplifier l'environnement d'exécution. C'est un système à déclarations explicites. Ils exigent la déclaration de tous les types des arguments et des variables. A chaque argument correspond une déclaration d'un foncteur. Chaque variable est accompagnée de la déclaration d'un ensemble de foncteur. Chaque foncteur est accompagné d'une déclaration de ses sous-arguments éventuels. Un sous-argument peut être lui même un foncteur à déclarer ou un type primitif *string, atome, ...* Une clause est bien-typée si chacune de ses formules atomiques est bien-typée et si pour toute variable apparaissant dans la clause, cette variable a un ensemble non vide de foncteurs. Une formule atomique $p(t_1,...,t_n)$ est bien-typée si chacun de ses arguments est bien-typé. Une variable est bien-typée si elle a un ensemble non-vide de foncteurs.

Un exemple d'implémentation de PROLOG nécessitant les déclarations explicites des types est réalisé pour TURBO-PROLOG *[Turbo86]* en utilisant des types primitifs.

2.4.2. Les modes (modèles d'appel).

Certains auteurs cités précédemment *[Bruynooghe82]* parlent également de type pour caractériser les modèles d'appel des clauses. Les types sont utilisés conjointement avec les modèles d'appel. Bien qu'une clause puisse être appelée en théorie avec tout modèle d'appel, certains cas d'appels peuvent conduire à des résultats dangereux.

Le modèle d'appel caractérise le sens de passage de l'information à travers les arguments *[Debray85a], [Deransart84a], [Deransart84b], [Mellish81a]*. C'est le problème du sens de l'instanciation des arguments dans une clause. Comme pour les types, deux solutions sont possibles pour déterminer les modes des arguments. Soit déclarés explicitement par le programmeur, soit déduits automatiquement.

Plusieurs auteurs *[Bruynooghe82, Nilsson83, Deitrich88, Kluzniak87]* ont lié dans leurs études sur les types en Prolog, la notion des modes avec celle des types. Faut-il effectuer des études séparées sur ces deux notions alors que l'une peut-être complémentaire à l'autre? La connaissance des types peut fournir des informations pour aider la détermination des modes *[Mishra84]*. La connaissance des modes peut aider à une vérification plus précise des types *[Deitrich88]*. Nous pensons qu'une étude mixte sur les types et les modes peut donner des résultats interressants.

Le but de ce paragraphe n'était pas d'étudier la notion de mode, mais de citer les travaux *[Bruynooghe82], [Nilsson83]* rencontrés précédemment qui utilisent cette notion conjointement avec les types. D'autres travaux que nous n'examinons pas ici ont été effectués sur l'utilisation des modes dans l'implémentation de PROLOG *[Oudot86], [Oudot87]*.

3. Calculer les types [Mishra84].

Nous présentons maintenant le système d'inférence de type de Mishra *[Mishra84]*. Nous présenterons ensuite des résumés de travaux plus récents. Les chapitres suivants seront consacrés à nos propres travaux sur ce sujet *[Azzoune88a]*, *[Azzoune88b]* en proposant un système d'inférence et quelques applications de ses résultats.

Le but de Mishra *[Mishra84]* est d'effectuer des vérifications statiques sur un programme PROLOG. Il infére automatiquement les types d'une classe restreinte de prédicats (les prédicats "non-polymorphes") sans qu'aucune déclaration explicite ne soit nécessaire. Ces types sont représentés par des arbres réguliers *[Hopcroft69]* analogues aux expressions régulières dans les langages réguliers.

En appliquant un ensemble de règles sur les clauses, il dérive un système d'inéquations dont la résolution donne la description de tous les termes constants pris par les variables pour lesquels le prédicat peut réussir: son "type". Cette résolution est possible grâce à la décidabilité des algorithmes d'intersection, de réunion, d'égalité *[Solomon78]*, *[Kral73]* sur les langages réguliers.

3.1. Les types

Les termes constants représentant les types et manipulés dans le système d'inférence de Mishra *[Mishra84]* sont représentés par des arbres réguliers constants. La sémantique de sa définition de type est qu'un prédicat p a un type T_p, si p(t) échoue pour tout terme constant t n'appartenant pas au type T_p.

Exemple: Soit le programme suivant:

```
graphe(a,b).
graphe(c,d).
```

Le type du prédicat graphe est donné par:

$$T(\text{graphe}) = a + c \# b + d$$

où l'opérateur "+" est pris dans le sens du "ou" (l'union ensembliste) et # est l'opérateur de construction de n-uplets. L'interprétation de a + c (respectivement b + d) contient tous les éléments que peut prendre le premier (respectivement le second) argument du prédicat graphe pour lesquels il peut réussir.

Mishra parle aussi de dénotation d'un prédicat. Cette dénotation consiste en une représentation de la sémantique de toutes les solutions possibles. La dénotation d'un prédicat p n-aire d'un ensemble de clause A, est l'ensemble de tous les n-uplets $(t_1,....t_n)$ où chaque t_i appartient à l'Univers de Herbrand et pour lesquels $p(t_1,....t_n)$ est déduit à partir de A. Celle-ci est notée par:

$$D(p) = \{(t_1,....t_n) \text{ tels que } A \rightarrow p(t_1,....t_n)\}$$

Un cas particulier est constitué par l'ensemble des solutions obtenues en exécutant la procédure, lorsqu'il est fini. Dans le cas particulier de l'exemple précédent, les solutions sont: (a, b) et (c, d). Poser une question ?p(t) consiste à voir si t appartient à D(p). Si t appartient à D(p), la résolution réussit, sinon elle échoue.

Or le **type** d'un prédicat décrit tous les termes pour lesquels le prédicat peut réussir; pour tout terme non décrit par le type, le prédicat ne peut pas réussir. Le type d'un prédicat est donc un sur-ensemble de sa dénotation et on a la propriété:

$$D(p) \subseteq T(p)$$

Dans l'exemple précédent, T(graphe) = {(a, b), (a, d), (c, b), (c, d)} et D(graphe) = {(a, b), (c, d)}. Il existe des éléments de T(graphe) qui n'appartiennent pas à D(graphe). L'ensemble différence T(p) - D(p) peut être important comme nous le verrons à la fin du chapitre. Cet ensemble caractérise la finesse du système d'inférence.

3.2. Représentation des types.

Comme nous l'avons souligné précédemment, les types sont représentés classiquement à l'aide d'une grammaire de termes.

Mishra *[Mishra84]* représente les types par des arbres réguliers (équivalents aux expressions régulières dans les langages réguliers) dont l'interprétation est un ensemble de termes constants.

Exemples:

1) L'ensemble "*couleur*" est représenté par l'arbre régulier:

couleur = rouge + blanc + vert

où le signe + est pris dans le sens du "ou" logique.

2) Pour une "*liste*" d'éléments appartenant à un domaine D, il a proposé la représentation suivante:

liste(D) = z! nil + cons(D,z)

où le symbole z ! exprime la récursivité sur z: on remplace "z" par nil ou par cons(D, z) à chaque fois que c'est nécessaire. L'interprétation de liste(D) est un ensemble de termes constants qui contient nil et contient cons(D, z) à chaque fois qu'il contient z. Le domaine D doit être défini car ce système n'autorise pas le polymorphisme. L'équivalent de cette représentation en notation grammaticale est donné par:

Liste_D ::= nil / cons(D, Liste_D)).

Les termes constants manipulés dans ce système, représentés par des arbres réguliers constants, sont définis par:

* L'ensemble vide ∅ est un arbre régulier.
* Un atome est un arbre régulier.
* Un n-uplet $(t_1, ..., t_n)$ d'arbres réguliers est un arbre régulier.
* Une structure ft est un arbre régulier où f est un foncteur et t un arbre régulier.
* Une union $t_1 + t_2$ est un arbre régulier, où chaque t_i est un arbre régulier. Le symbole "+" est pris dans le sens du "ou" logique.
* Si g(z) est un arbre régulier contenant une variable de récursion z alors z ! g(z) est un arbre régulier.

Les interprétations de ces définitions sont des ensembles de termes constants. L'interprétation de z ! g(z) par exemple consiste en le plus petit ensemble de termes X tel que X = { g(x) / x ∈ X }, c'est-à-dire l'ensemble vide.

Cette représentation permet de décrire tous les termes constants manipulés en PROLOG. Le symbole "!" de récursion permet de représenter des ensembles infinis de termes constants, comme par exemple l'ensemble des entiers naturels:

$$N = z!\ 0 + s(z) = \{\ 0, s(0), s(s(0)), ...\}$$

3.3. Le système d'inférence.

Une observation faite par *[Solomon78]*, *[Kral73]* sur les définitions des types paramétrés en Pascal est à la base de ses résultats. Solomon *[Solomon78]* identifie le graphe d'une définition de type paramétré (essentiellement une forme restreinte d'un arbre régulier) avec un automate fini non-déterministe (AFND). Il conclut que l'équivalence d'une telle définition peut être calculée utilisant les algorithmes standards pour le calcul de l'équivalence des automates finis déterministes (AFD) *[Hopcroft69]*. Une définition de type peut donc être transformée en une représentation minimale utilisant l'algorithme de minimisation d'un AFD. En ayant un tel automate (langage), il utilise des algorithmes décidables sur les langages réguliers.

L'inférence se fait en deux étapes:

1ère étape:

La 1ère étape consiste à déterminer les types des têtes des clauses. Ces types contiennent éventuellement des variables qu'il faut instancier. Le type d'un prédicat n-aire p tête d'une clause est pris comme étant le n-uplet formé des types des arguments du prédicat. Le type d'un argument i d'un prédicat p est la réunion formée des types des arguments i de toutes les définitions de p. Le type d'un argument i est l'argument lui même, où les variables seront instanciées après la seconde étape.

2ème étape:

La 2ème étape consiste en la génération et la résolution des inéquations. Cette génération se fait par application sur le programme d'un ensemble de règles *[Mishra84]* exprimant les conditions de bon typage du programme. Pour chaque appel $p(t_1, ..., t_n)$ (dans un corps d'une clause) d'un prédicat p de type $p(\xi_1, ..., \xi_n)$, on produit l'inéquation $t_i \leq \xi_i$ pour i=1,n. Pour chaque clause avec corps on obtient un système d'inéquations d'arité égale à la somme des arités des buts formant le corps de la clause.

Mishra identifie l'ensemble des solutions à un langage régulier, ce qui lui permet d'appliquer les algorithmes classiques d'égalité, d'intersection, d'inclusion et de l'union dans les langages réguliers pour la détermination effective de la solution (types des variables). Pour effectuer cette identification, les membres droits des inéquations sont transformés en des équations appelées "*équations linéaires de feuilles*". Un ensemble d'équations linéaires de feuilles sur un ensemble de variables V est un ensemble d'équations de la forme:

$X = t_1 + ... + t_n$ où

$t_i = c / Y / (Y_1, ..., Y_n) / cY$ où $Y \in V$ et c un atome.

Il fait correspondre à ces équations linéaires de feuilles des automates appelés "*automates de racine à feuille (RAF)*", ce qui permet d'appliquer un théorème de Solomon *[Solomon78]* qui garantit que le langage accepté par cet automate est régulier.

Pour chaque équation $X = t_1 + ... + t_n$, sont produites les règles suivantes:

$X \rightarrow_c F$ **si** $c \in \{t_i\}$

$X \rightarrow_{vide} Y$ **si** $Y \in \{t_i\}$

$X \rightarrow_c Y$ **si** $cY \in \{t_i\}$

$X \rightarrow (n,j)Y_j$ **si** $(Y_1,...,Y_n) \in \{t_i\}$ $\quad j=1,n$

où F est un état final.

Les algorithmes d'intersection, de réunion et d'équivalence sont donc ceux des automates. L'algorithme de fermeture appliqué à deux arbres régulières s(0) et s(s(0)) par exemple donne s(0 + s(0)). Cet algorithme est analogue à celui de la transformation des automates finis non-déterministes en des automates finis déterministes.

3.4. Vérifications

Il montre la cohérence de son système, en vérifiant qu'un programme "*mal-typé*" ne peut pas réussir. En terme d'inférence de type, la notion de "*mal-typé*" est liée à l'impossibilité de résoudre le système d'inéquations, ce qui correspond à une impossibilité dans l'unification. Cette propriété est donnée par le théorème suivant:

Théorème: Soient $r_i \leq s_i$ un ensemble d'inéquations simultanées engendrées à partir d'une clause. S'il n'existe pas de solutions pour cet ensemble d'inéquations, alors cette clause ne réussissant jamais est inutile.

La démonstration de ce théorème est donnée dans *[Mishra84]*.

Remarque: Le système de Mishra est bien fondé dans la mesure où le type inféré contient la dénotation, mais il n'est pas très précis. Les ensembles différence T(p) - D(p) des prédicats sont très grands.

Il a proposé en outre une étude générale sur la vérification des types pour les langages applicatifs (HOPE, ML, SASL,..) dans *[Mishra85]*.

3.5. Exemples d'inférence de type:

Exemple (1): Soit le programme suivant:

```
pere(a, b).
pere(b, c).
pere(c, d).
g_pere(X, Y):-pere(X, Z), pere(Z, Y).
```

Calculer les types T_{pere} et T_{g_pere} de pere et g_pere. On pose:

$T_{pere} = \xi_1 \# \xi_2$
$T_{g\_pere} = \xi_3 \# \xi_4$ avec

$\xi_1 = a + b + c$
$\xi_2 = b + c + d$
$\xi_3 = X$
$\xi_4 = Y$

et avec les inéquations:

$X \leq \xi_1 \rightarrow X \leq a + b + c$ **qui est solution**
$Z \leq \xi_2 \rightarrow Z \leq b + c + d$
$Z \leq \xi_1 \rightarrow Z \leq a + b + c$
$Y \leq \xi_2 \rightarrow Y \leq b + c + d$ **qui est solution**

L'intersection des 2^{eme} et 3^{eme} inéquations donne:

$$Z \leq b + c$$

On montre donc que le système précise le domaine de la variable Z, mais comme Z n'est pas utilisée directement dans les types recherchés, il suffit de vérifier que l'intersection n'est pas vide et déduire que le système d'inéquations est solvable.

On obtient:

$$T_{pere} = \xi_1 \# \xi_2 = a + b + c \# b + c + d$$
$$T_{g\_pere} = \xi_3 \# \xi_4 = X \# Y = a + b + c \# b + c + d$$

Les ensembles différences T(pere) - D(pere) (respectivement T(g_pere) - D(g_pere)) sont donnés par:

{(a, c), (a, d), (b, b), (b, d), (c, b), (c,c) }

(respectivement { (a, b), (a, d), (b, b), (b, c), (c, b), (c, c), (c,d) })

qui ont des cardinalités très importantes.

Exemple (2): Soit le programme suivant:

```
sub_un(s(0),0).
sub_un(s(s(X)),s(Y)):-sub_un(s(X),Y).
```

On pose le type du prédicat sub_un: $T_{sub\_un} = \xi_1 \# \xi_2$ avec:

$$\xi_1 = s(0) + s(s(X))$$
$$\xi_2 = 0 + s(Y)$$

et comme inéquations:

(1) $s(X) \leq \xi_1$

En remplaçant ξ_1 par son expression, on obtient $s(X) \leq s(0) + s(s(X))$.

Par fermeture, on obtient: $s(X) \leq s(0+s(X)) \rightarrow X \leq 0 + s(X)$

Par l'algorithme d'inclusion: $X = z!\ 0 + s(z)$ (3).

(2) $Y \leq \xi_2$

En remplaçant ξ_2 par son expression, on obtient $Y \leq 0 + s(Y)$.

Par l'algorithme d'inclusion $Y = z!\ 0 + s(z)$ (4).

En remplaçant les variables X et Y par leurs expressions (3), (4):

$$T_{sub\_un} = s(0) + s(s(X)) \# 0 + s(Y)$$
$$= z!\ s(0) + s(z) \# z!\ 0 + s(z)$$

3.6. Autres travaux.

Kanamori et Horiuchi *[Kanamori85]* ont proposé une méthode d'inférence qui nécessite, à l'encontre de celle de Mishra, la déclaration de tous les constructeurs de type sous une forme clausale. Par contre ils autorisent les prédicats polymorphes. L'idée est de décrire un sur-ensemble de l'ensemble de succès en associant une substitution de type (affectations d'ensembles de termes constants aux variables) à chaque tête de clause. Le sur-ensemble est calculé par une approximation séquentielle qui est équivalente à la résolution d'inéquations de Mishra. Cette approximation séquentielle se fait par des transformations inspirées des transformations de Herbrand *[Van Emdem82]*, *[Lloyd84a]*. Une autre approche d'inférence par une interprétation abstraite a été évoquée par les deux auteurs *[Horiuchi87]*.

J. Zobel *[Zobel87]* a étudié le problème d'une dérivation des types polymorphes pour les programmes PROLOG. Un programme est transformé de telle sorte que les corps des clauses soient des conjonctions de littéraux *[Lloyd84b]*. Il a proposé un système d'inférence sans déclaration. Les types sont représentés par des règles de réécriture. Les récursions éventuelles ne sont pas explorées. Il a défini un algorithme d'unification de type, qui est utilisé pour dériver le type sous forme de règles de réécriture de chaque variable. Cet algorithme correspond à l'algorithme d'intersection de Mishra. Le résultat de l'unification de deux expressions de type (intersection des 2 expressions) est formé par la composition de substitutions, d'une manière analogue à celle de la composition d'unificateurs dans l'unification classique *[Lloyd84a]*, avec néanmoins certaines différences. En particulier, l'unification classique utilise le concept de substitution qui est le *pguc* d'un ensemble de termes à unifier, alors que l'unification des expressions de type utilise des règles de fermeture *[Zobel87]* (telles que $a \wedge b \rightarrow \varnothing$, $a \wedge a \rightarrow a$,...). Si l'algorithme ne peut pas dériver le type d'une variable (type vide), le corps de la clause considerée ne peut pas réussir.

F. Kluzniak *[Kluzniak87]* a proposé une technique pour analyser le flot des données dans les programmes pour une forme restreinte de PROLOG. Cette technique est utilisée pour une dérivation d'information du type à partir du programme. La forme restreinte de PROLOG, est *un PROLOG constant [Bruynooghe86a], [Bruynooghe86b], [Bruynooghe86c]*. Le système exige la déclaration de tous les modes. Le programme doit remplir les conditions suivantes: Les paramètres d'entrée d'un appel doivent être constants (clos). Les paramètres de sortie d'un appel doivent être des variables libres. La dernière condition est que le programme doit être accompagné de la question à poser.

Il existe d'autres travaux que nous n'avons pas exposé ici et qui utilisent la notion de type en PROLOG ou un de ses dérivés *[Goguen84], [Ait-kaci84], [Ait-kaci85], [Naish87]*.

4. Bilan

Dans ce chapitre nous avons présenté quelques travaux rencontrés au cours de notre étude. Nous allons présenter un bilan des deux approches précédentes. Nous exposons quelques limitations de ces deux approches prise l'une indépendamment de l'autre.

4.1. L'approche déclarative.

Dans l'approche déclarative, le programmeur doit déclarer explicitement les types des arguments sous une forme ou une autre. Ceci n'a d'utilité que si des règles de "*bon-typage*" sont énoncées et vérifiées. Cette notion de "*bon-typage*" n'est pas très nette en PROLOG dans la mesure où l'échec d'un programme peut être le comportement souhaité. De plus, un des principes de la programmation logique est que le programmeur ne se préoccupe que des relations logiques qu'il décrit, mais pas des valeurs que pourraient avoir les variables.

L'obligation de déclarer systématiquement tous les types a les inconvénients suivants:

* On n'a pas de relations avec la notion d'échec et de réussite et donc on ne fait qu'un contrôle syntaxique.
* Diminution de la liberté d'expression du langage en obligeant le programmeur à tout typer.
* Difficulté de se souvenir de toutes les valeurs possibles qu'un argument peut prendre.
* Une définition incorrecte d'un type provoque des erreurs à l'exécution sans rapport évident avec l'origine de l'erreur, (si c'est une erreur typographique ou de programmation).
* Une modification du programme peut obliger le programmeur à changer les types de certains prédicats.

Nous pensons qu'une meilleure solution serait d'avoir une approche souple: Les déclarations des types exprimées doivent être statiquement vérifiées (intersection du domaine d'appel et du domaine de définition non vide) mais il n'est pas obligatoire de tout typer. Le programmeur peut

exprimer ses pensées et présenter la sémantique du programme à travers les résultats qu'il désire obtenir sous forme de déclarations de type de quelques prédicats.

Cette approche peut être doublée de l'utilisation en parallèle d'un système d'inférence; le programmeur aide alors le système d'inférence en donnant l'information qu'il connaît sans être obligé de tout déclarer. Un système de vérification englobant un système d'inférence et pouvant utiliser les connaissances éventuelles fournies par le programmeur peut être efficace.

4.2. L'approche inférentielle.

La 2ème approche consiste en des travaux d'inférence (déduction automatique) des types des prédicats. La limitation majeure de l'approche inférentielle est que l'on calcule toujours un sur-ensemble de l'ensemble de solutions. Tous les systèmes existants actuellement essayent de rapprocher au mieux l'ensemble de type de l'ensemble de solutions. D'un point de vue pratique aucun système existant actuellement n'a donné des résultats complètement satisfaisants: le problème est très délicat pour le langage PROLOG. Certains auteurs *[Kluzniak87], [Mishra84]* ont choisi une voie progressive. Kluzniak essaye de résoudre le problème pour une forme restreinte du langage: PROLOG constant *[Bruynooghe86a, 86b, 86c]*. Même pour cette forme restreinte les résultats ne sont pas très satisfaisants vu que l'on calcule toujours un sur-ensemble de l'ensemble de solutions. Mishra a suivi ce même chemin en inférant les types des prédicats non-polymorphes. Des limitations de ce système seront présentées ci-dessous. S'ajoute à tout ceci le problème de l'efficacité qui est un facteur important et les informations nécessaires pour de telles inférences car il ne faut pas l'oublier, certaines méthodes (Kluzniak par exemple) exigent des informations de la part du programmeur (Kluzniak exige les modes).

Nous exposerons quelques points de "faiblesse" qui nous ont paru importants à souligner dans le système de Mishra, pour essayer de les éviter dans le système que nous proposerons.

Le système de Mishra ne traite que les prédicats non-polymorphes.

Or, une grande partie des prédicats d'un programme sont polymorphes. D'autre part, le système tel qu'il est, appliqué aux prédicats non-polymorphes, produit des ensembles de type très largement supérieurs à l'ensemble représentant la dénotation. Le système a une forte imprécision; l'ensemble différence T(p) - D(p) est très important. Il engendre beaucoup d'éléments pour les ensembles de type qui ne sont pas solution, et dont la présence est inutile.

Exemple:

Soit l'exemple (1) précédent. Les possibilités d'appel du type inféré du prédicat g_pere par ce système sont:

{(a,b), (a,c), (a,d), (b,b), (b,c), (b,d), (c,b), (c,c), (c,d)}

7 appels parmi les 9 possibles conduisent à l'échec. Les seules possibilités de succès sont:

{(a,c), (b,d)}

Ce système produit beaucoup d'éléments de type appartenant à T(p) mais n'appartenant pas à D(p), alors que la précision d'un système d'inférence se mesure par la différence T(p) - D(p).

5. Conclusion

Déclarer tous les types est fortement contraignant et diminue la liberté d'expression du langage. Il n'est pas facile de se souvenir de l'ensemble de termes que pourrait prendre chaque argument à son écriture. Par contre nous pensons que l'information ainsi associée à chaque prédicat constitue un enrichissement du langage; elle exprime la pensée de l'auteur, aide à cerner la sémantique du programme et permet des vérifications statiques de cohérence. Nous pensons aussi qu'il peut être judicieux de donner la possibilité au programmeur de donner les informations dont il dispose sans lui exiger de tout typer.

L'idéal serait d'avoir un système d'inférence de type qui:

* Soit applicable à tous les prédicats polymorphes ou non.
* N'exige aucune déclaration ou intervention du programmeur pour être complètement automatisé.
* Qui puisse utiliser les connaissances fournies par le programmeur sur la structure de certains arguments ou de prédicats.
* Rapproche au mieux l'ensemble de type inféré de l'ensemble de dénotation (c'est-à-dire minimiser l'ensemble différence T(p) - D(p)).
* Donne des résultats applicables pour des optimisations et/ou vérifications en compilation et/ou en interprétation.

Dans le prochain chapitre, nous proposerons une méthode d'inférence de type. Cette méthode est basée sur une simulation de l'unification et une simulation de tous les appels possibles des buts des corps des clauses avec un traitement particulier sur les appels récursifs *[Azzoune88a]*, *[Azzoune88b]*.

CHAPITRE 3

UNE METHODE D'INFERENCE DE TYPE

1. Introduction.

Nous exposons dans ce chapitre les notions de base de notre méthode d'inférence d'une manière progressive. La première étape dans une étude d'inférence de type est la définition de la notion de type des arguments et des prédicats. Nous introduisons une notion *d'éléments de type*. Ces éléments de type décrivent les objets que manipulent un programme Prolog.

Quelle est la définition du type d'un prédicat ? Notre réponse est un sur-ensemble de son ensemble de succès. Le slogan de notre typage est qu'un programme mal-typé ne peut pas réussir. Si un prédicat n-aire p a un type T_p alors pour tout n-uplet constant t n'appartenant pas à T_p, p(t) échoue.

Pour déterminer le type d'un prédicat p, il est donc nécessaire de calculer l'ensemble des valeurs prises par les variables apparaissant dans les têtes des clauses de définition de p. Idéalement, on voudrait que cet ensemble coincide avec l'ensemble de succès mais le problème de la détermination de toutes les instanciations possibles d'une variable est en général indécidable. Notre but sera de rapprocher au mieux cet ensemble de valeurs (le type) de l'ensemble de succès.

La sémantique déclarative nous informe que pour toute valeur prise par une variable n'apparaissant que dans la tête de la clause, la clause réussit (si les autres arguments permettent la reussite). La seule condition qu'on ait dans de telles situations est que si une telle variable est répétée la même valeur lui est assignée.

Pour une variable apparaissant dans la tête et dans le corps de la clause, les valeurs prises par cette variable dans la tête dépendent de celles prises dans le corps (et vice versa). Nous appellerons de telles variables: *les variables de type.*

Pour déterminer les ensembles de valeurs prises par les variables de type il faut examiner les corps des clauses. Ces ensembles sont déterminés suivant les unifications de ces variables de type. Comme nous ne pouvons pas réaliser toutes les unifications (cas récursif), nous simulerons une unification d'un terme d'appel t_1 avec un terme de définition t_2 par l'inéquation $t_1 \leq t_2$. Le terme t_2 peut contenir de nouvelles variables de type qu'il faut remplacer par leurs types.

Pour déterminer un sur-ensemble de l'ensemble de valeurs prises par une variable de type apparaissant dans un but p d'un corps d'une clause, il faut examiner toutes les unifications possibles de cette variable et donc examiner tous les choix possibles des unifications du but p avec ses définitions; nous simulerons tous les appels possibles du but p de ses définitions. Pour que cette simulation ne boucle pas (cas récursif) nous traiterons de manière particulière les appels récursifs.

2. Eléments de type.

Nous appelons éléments de type (ou valeur de type) une partie de l'Univers de Herbrand, ensemble de termes constants associés à un programme PROLOG. Un élément de type est décrit par un formalisme d'expression de type auquel nous donnerons une interprétation ensembliste.

Un élément de type est:

* Un *atome* (*exemple: 1, b, 2, nil, ...*).
* Un symbole "any_i" i=0,1,2,.... Un symbole any_i dénote le type indéterminé. Il est utilisé pour représenter une partie quelconque de l'Univers de Herbrand.
* Une structure "$f(t_1, ..., t_n)$" où f est un symbole de fonction (foncteur) et les t_i des éléments de type (*exemple: f([], any_1), s(0), ...*).
* Une union "$t_1 + ... + t_n$" d'éléments de type $t_1, ..., t_n$. (*exemple: a + b + any_1*).
* $\varnothing$ représentant l'ensemble vide (aucun élément possible): nous l'appelons ensemble *contradiction*. Il correspond à une situation d'échec que l'on a pu détecter: appel d'un prédicat non défini, utilisation d'une variable dans plusieurs occurences dont les domaines sont disjoints, des unifications impossibles.
* L'expression "$r\ !\ (t_1 +...+ f(...r...) +...+ t_n)$". Son interprétation définit le plus petit ensemble qui contient l'interprétation de t_i, $i=1,n$ et contient l'interprétation de $f(..r..)$ à chaque fois qu'il contient celle de r. $f(..r..)$ est un élément de type contenant un symbole de récursion r. Les t_i sont des éléments de type. Le symbole r peut être remplacé par

  $(t_1 + ... + f(..r..) + ... + t_n)$ (avec renommage des any_i éventuellement) à chaque fois que c'est nécessaire. Cette notation nous permet de représenter des termes constants ayant la même structure. Pour raison de simplicité, nous avons repris la syntaxe de Mishra au lieu d'en proposer une autre. (*exemple: Liste = r ! ([] + [any_i / r]).*

Nous donnons maintenant l'interprétation I des éléments de type définis précédemment comme des ensembles de termes constants.

I : Eléments de type → Univers de Herbrand

Définition de l'interprétation *I:*

* $I(\varnothing) = \{ \}$
* $I(atome) = \{ atome \}$
* $I(f(t_1, ..., t_n)) = \{ f(t_1', ..., t_n') \ / \ t'_i \in I(t_i), i=1,n \}$.
* $I(any_i) =$ *partie quelconque de l'Univers de Herbrand*
* $I(t_1 + ... + t_n) = I(t_1) \cup .... \cup I(t_n)$
* $I(r\ !\ (t_1 + ... + f(r) + ... + t_n))$ = le plus petit ensemble S de termes constants tel que $S = I(t_1) \cup ... \cup I(f(x)) \cup ... \cup I(t_n) \ / \ x \in S$. La plus petite solution pour l'interprétation de $r\ !\ f(r)$ existe toujours *[Mishra84]* et est donnée par:

$$S = \bigcup_{i=0}^{\infty} \{ f^i(\varnothing) \} = \varnothing$$

Exemples:

$I(a + b + \{c + d\} + e) = \{ a, b, c, d, e \}$

$I(r\ !\ (a + f(r)) = \{ a, f(a), f(f(a)), f(f(f(a))), ... \}$.

$I(f(a + b\ .\ c + d\ )) = \{ f(a, c), f(a, d), f(b, c), f(b, d) \}$

$I(r\ !\ (any_0 + r)\ ) = I(any_0)$

$I(r\ !\ (f(any_1) + f(r)) = I(f(any_1))$

3. La méthode d'inférence.

3.1. Les bases de la méthode d'inférence.

Comme nous l'avons souligné auparavant, le problème de la détermination de toutes les instanciations possibles d'une variable est en général indécidable. Néanmoins nous pouvons déterminer un ensemble qui contient cet ensemble d'instanciations. Cette détermination est possible grâce à la possibilité de faire des restrictions sur l'Univers de Herbrand. Ces restrictions se font de la manière suivante: la structure des arguments nous permet d'affirmer qu'ils ne peuvent s'unifier qu'avec des termes ayant une certaine forme. Nous pouvons affirmer par exemple que le terme $f(t_1, ..., t_n)$ ne peut s'unifier avec aucun terme constant appartenant à l'Univers de Herbrand et ayant la forme $g(t'_1, ..., t'_m)$ avec $f \neq g$ ou $n \neq m$, ou un t_i ne s'unifiant pas avec t'_i. En termes ensemblistes, l'ensemble des termes constants obtenus en instanciant les variables de $g(t'_1,....,t'_n)$ ne peut contenir aucune instance du terme $f(t_1,....t_n)$. Nous obtenons ainsi un sous-ensemble de l'Univers de Herbrand comme type d'un argument.

Définition: Nous définissons le type T_p d'un prédicat p n-aire à m définitions comme étant un n-uplet $<T_1, ..., T_n>$. T_i est le type de l'argument i.

Définition: Nous définissons le type T_i de l'argument i comme étant un ensemble ordonné $T_{i1} \vee ... \vee T_{im}$ de m éléments. Chaque composante de type T_{ij} est le type de l'argument i dans la clause j, $1 \leq i \leq n$, et $1 \leq j \leq m$. Par là même nous définissons le type de la $j^{ème}$ clause de p comme étant le n-uplet $< T_{1j}, ..., T_{nj} >$.

3.2. Principe de la méthode

Le type d'un prédicat p dépend du type des ses clauses. Inférer le type d'une clause revient à déterminer aussi finement que possible les types des variables apparaissant dans sa tête. Le calcul du type d'une variable diffère suivant que la variable apparaît dans la tête seulement ou dans la tête et dans le corps de la clause. Une variable n'apparaissant que dans la tête peut représenter tout terme de l'Univers de Herbrand. Si une

variable apparaît dans la tête et dans le corps de la clause alors les valeurs qu'elle prend dans la tête dépendent de celles qu'elle prend dans le corps et vice versa. Dans ce cas, l'inférence du type de la variable revient à déterminer ses types aux différentes positions dans le corps. Nous calculons ensuite les intersections éventuelles des interprétations de ces types en ses différentes positions.

Exemple: Soit la clause p(X, Y):-q(Y), r(Y). avec T_q le type de q et T_r celui de r (q et r ont une seule définition chacun). La variable X n'apparaissant que dans la tête de la clause, peut représenter tout terme de l'Univers de Herbrand, son type est any_0. La variable Y apparaît dans la tête et dans le corps de la clause, son type est donc $I(T_q) \cap I(T_r)$.

3.2.1. Equations de type.

Nous avons défini le type T_{p_j} d'un prédicat p n-aire à sa définition j comme étant le n-uplet $< T_{1j}, ..., T_{nj} >$ où la composante de type T_{ij} correspond au type de l'argument i dans la définition j.

Soit le terme t, i^{eme} argument dans la j^{eme} définition d'une tête de clause. Si t est un atome alors son type est cet atome lui-même. Si t est une variable n'apparaissant que dans la tête de la clause alors son type est any_i (nous renommons les any_i en changeant l'indice i. Ce renommage correspond à celui des variables en Prolog). Si une variable apparaît dans la tête et dans le corps de la clause alors elle forme **une variable de type**. Cette variable de type sera remplacée par son type obtenu après traitement du corps de la clause. Si t est de la forme $f(a_1, ..., a_m)$ alors son type est $f(a'_1, ..., a'_m)$ où a'_i est le type de a_i i=1,m.

Nous formons une équation de type comme suit: $T_{ij} = t$ où les variables n'apparaissant que dans la tête de la clause sont remplacées par des any_i (t est le type de l'argument ayant T_{ij} comme composante de type). Leur forme générale est donc actuellement:

T_{ij} = Exp
Exp = atome / any_k / variable de type / $f(Exp^*)$

Exemple: Soit la clause append([X|L], M, [X|N]) :- append(L, M, N).

Les équations de type pour cette clause sont données par:

$$T_{11} = [any_0|L]$$
$$T_{21} = M$$
$$T_{31} = [any_0|N]$$

où L, M et N sont des variables de type à remplacer par leurs types obtenus après traitement du corps de la clause.

3.2.2. Résolution des équations de type.

Définition: Une équation de type est dite "résolue" lorsque sa partie droite est un élément de type, c'est-à-dire sans variable de type.

Il faut donc déterminer les types des variables de type. Nous allons présenter cette résolution progressivement suivant la nature de la clause (avec ou sans corps) et suivant que nous avons des parties récursives ou non dans le programme. Nous commençons par le cas où la clause est un fait (pas de corps). Nous présenterons ensuite le cas où la clause a un corps. Nous commençons par celui où nous n'avons pas d'appel récursifs, puis nous compliquons avec l'introduction des appels récursifs.

3.3. Les faits.

Si une clause est un fait de la forme $p(t_1, ..., t_n)$. alors les variables de cette clause n'apparaissent évidemment que dans sa tête. Une équation de type pour cette clause est de la forme $T_{ij} = t'_i$ où t'_i est égal à t_i avec un remplacement des variables par des éléments de type any_i. ($t'_i = t_i\,\theta$ avec $\theta = (V_i/any_i$ pour i=1,2,...)). Les membres droits de ces équations de type sont des éléments de type, et donc ces équations sont résolues. Une équation de type est donc de la forme $T_{ij} = t'_i$ avec:

$$t'_i = atome \,/\, any_k \,/\, f(t'_1, ..., t'_m).$$

Exemples: 1) Soit le fait: append([], K, K). Les équations de type sont:

$$T_{11} = []$$
$$T_{21} = any_0$$
$$T_{31} = any_0$$

2) Soient les deux faits:

pere(a, b).

pere(b, c).

Les équations de type sont données par:

$$T_{11} = a$$
$$T_{21} = b$$
$$T_{12} = b$$
$$T_{22} = c$$

3) Soit le fait: equal(X, X). Les équations de type sont données par:

$$T_{11} = any_0$$
$$T_{21} = any_0$$

Tous les membres droits de ces équations sont des éléments de type et donc toutes ces équations sont résolues.

3.4. Clauses avec corps typé.

Connaissant les types des buts du corps d'une clause, nous inférons le type de sa tête. Ce type calculé peut être à son tour utilisé pour d'autres inférences des types des prédicats appelant cette clause. Pour "faciliter" un peu la compréhension de la méthode, nous allons étudier cette catégorie de clauses progressivement suivant le nombre de buts dans les corps. Nous commençons par le cas où le corps d'une clause est constitué d'un seul but, et nous généraliserons ensuite à n buts.

3.4.1. Corps à un but.

Supposons que le corps de la clause soit formé d'un seul but q à k définitions. La clause est donc de la forme $p(a_1, \ldots, a_n) :- q(t_1, \ldots, t_m)$. Supposons que le type de q soit donné par:

$$q_1(t'_{11}, \ldots, t'_{m1}).$$
ou
$$\ldots \quad \ldots$$
ou
$$q_k(t'_{1k}, \ldots, t'_{mk}).$$

où t'_{ij} est un élément de type de la forme:

$$t_i = \text{atome} / any_l / t_1 + \ldots + t_l / r\,!\,(t_1 + \ldots + t_l) \;/\; f(t_1, \ldots, t_l).$$

Exemple: Nous développons au fur et à mesure l'exemple suivant:

```
p(f(V,X)) :- equal(f(a),X).
q(Y):-pere(Y,b).
```

avec les types de equal et pere calculés précédemment:

$equal(any_0, any_0)$

pere(a, b)

ou

pere(b, c).

3.4.1.1. Les équations de type

Dans de telles clauses, les équations de type sont de la forme $T_{ij} = a_i$ $(i=1,n)$ où les variables de a_i n'apparaissant que dans la tête de la clause sont remplacées par des any_l. Les variables restantes (apparaissant dans la tête et dans le corps simultanément) forment les **variables de type**. Nous déterminons les valeurs de ces variables de type en examinant le corps de la clause.

Exemple: Pour les exemples précédents, les équations de type sont:

Pour p: $T_{11} = f(any_1, X)$

Pour q: $T_{11} = Y$

où X et Y sont des variables de type qu'il faut remplacer par leurs types obtenus après traitement des corps des clauses.

Pour examiner un corps d'une clause, nous allons suivre les étapes suivantes:

3.4.1.2. Unification de type

Simulation de l'unification

L'unification d'un argument d'appel t_i (appartenant au corps de la clause) avec un argument de définition ayant le type t'_{ij} (appartenant à la tête de la clause avec laquelle nous unifions) est simulée par une inéquation: $t_i \leq t'_{ij}$. Cette inéquation a la sémantique suivante: l'ensemble

des valeurs prises par t_i est inclu dans l'interprétation de t'_{ij}.

Soit une inéquation $t_i \leq t'_i$, produite lors de l'unification d'un but p_j d'une clause P:-... avec la tête d'une clause C:-.... Nous discutons cette simulation suivant la nature des termes t_i et t'_i.

a) *t_i est un atome c_l.*

Si t'_i est un atome c_2 alors pour que l'unification réussisse il faut que $c_1=c_2$ ce qui implique $c_1 \leq \{c_2\}$ d'où l'inéquation $t_i \leq t'_i$.

Si t'_i est une variable n'apparaissant que dans la tête de la clause (dont le type est any_i) alors l'unification réussit en donnant à cette variable la valeur c_1. Une telle variable peut représenter tout terme de l'Univers de Herbrand. L'atome c_1 appartient à l'Univers de Herbrand d'où $t_i \leq t'_i$.

Si t'_i est une variable X apparaissant dans la tête et dans le corps de la clause C (variable de type) alors lors de l'unification elle prendra la valeur c_1. Cette valeur est propagée dans les autres positions de la variable dans la clause C. Pour que ses unifications en ses différentes positions réussissent il faut qu'il existe au moins une possibilité d'appel telles que les unifications en ces différentes positions soient possibles avec l'atome c_1. L'atome c_1 doit donc apparaître dans tous les domaines en ces autres positions de X. D'où $X \leq \{c_1\}$ en ces différentes positions, et donc son apparition dans l'intersection des interprétations des domaines en ces différentes positions. Par conséquent $X \leq \{c_1\}$ d'où $t_i \leq t'_i$.

Si t'_2 est de un terme $f(a_1, \ldots, a_n)$, l'unification échoue. Une inéquation de la forme $c_1 \leq f(a_1, \ldots, a_n)$ est aussi impossible (système à éliminer).

b) *t_i est une variable X.*

Si t'_i est un atome c_1 alors leur unification donne à X la valeur c_1. La substitution $< X/c_1 >$ est propagée aux autres positions de la variable X dans la clause P. Pour que les unifications en ces autres occurrences de la variable réussissent, il faut qu'elles soient possibles avec cet atome c_1 et donc nous avons $X \leq \{c_1\}$ en ces positions. Par intersection des interprétations

des domaines en ces différentes positions, nous obtenons un ensemble qui contient C_1 et donc $X \leq \{C_1\}$ d'où $t_i \leq t'_i$.

Si t'_i est une variable n'apparaissant que dans la tête de la clause dont le type est any_i alors cette variable peut représenter tout terme de l'Univers de Herbrand. Lors de l'unification des deux arguments, ce terme de l'Univers de Herbrand sera pris par la variable X d'où $X \leq I(any_i)$ et donc $t_i \leq t'_i$.

Si t'_i est une variable Y, alors la valeur prise par Y est aussi prise par X et donc $X \leq Y$ d'où $t_i \leq t'_i$.

Si t'_i est de terme $f(a_1, ..., a_n)$ alors l'unification de t_i avec t'_i donne une forme $f(b_1, ..., b_n)$ à t_i et nous effectuons les unifications des arguments. Si ces unifications réussissent alors nous aurons les inéquations $b_j \leq a_j$ et donc $f(b_1, ..., b_n) \leq f(a_1, ..., a_n)$ d'où $t_i \leq t'_i$.

c)*t_i est de un terme $f(a_1,...,a_n)$.*

Si t'_i est un atome C_1 alors l'unification échoue. Un système contenant une inéquation $f(a_1, ..., a_n) \leq C_1$ est aussi impossible.

Si t'_i est une variable n'apparaissant que dans la tête de la clause (le type de cette variable est any_i) alors pour toute instance des variables de $f(a_1, ..., a_n)$, cette instance appartient toujours à l'Univers de Herbrand, d'où $f(a_1, ..., a_n) \leq I(any_i)$ et donc $t_i \leq t'_i$.

Si t'_i est une variable Y alors elle prendra la nouvelle forme $f(b_1,...,b_n)$. Nous unifions les arguments a_i et b_i entre eux. Si ces unifications réussissent alors nous obtenons les inéquations $a_i \leq b_i$ d'où l'inéquation $f(a_1,...,a_n) \leq f(b_1,...,b_n)$ et donc $t_i \leq t'_i$.

Si t'_i est de un terme $g(b_1,....b_m)$ alors pour que l'unification réussisse il faut que $f=g$ et $n=m$, condition exigée aussi lors de la "normalisation" des inéquations. Si cette condition est vérifiée alors nous unifions les arguments. Si ces unifications réussissent alors nous obtenons $a_i \leq b_i$ et donc $f(a_i,...,a_n) \leq g(b_1,....b_m)$ d'où $t_i \leq t'_i$. Si $f \neq g$ ou $n \neq m$ ou un a_i ne s'unifie pas

avec b_i alors l'unification est impossible. Un système contenant une telle inéquation est impossible.

Si t'_i est un type déjà calculé c'est-à-dire de la forme $t''_1 + ... + t''_n$ ou $r!(t''_1 + .. + f(r) + .. + t''_n)$ alors pour toutes les unifications des exécutions réussies nous avons $t_i \leq t'_i$, puisque l'interprétation du type de ces éléments de type contient l'ensemble de solutions de cet argument lors de toutes les exécutions réussies.

Bilan: En simulant l'unification par des inéquations de type, nous ne perdons pas de solution.

A chaque appel de la $j^{ème}$ définition de q correspond donc le système d'inéquations:

$$s_i = (t_1 \leq t'_{1j} \,\&\, t_2 \leq t'_{2j} \,\&\, ... \,\&\, t_m \leq t'_{mj}).$$

le signe & est pris dans le sens de "et".

Simulation de tous les appels possibles:

Le prédicat q a k définitions et donc le but $q(t_1, ..., t_m)$ peut s'unifier avec ces k définitions. En traitant toutes ces possibilités d'unification nous obtenons une disjonction de k systèmes d'inéquations. Tous ces systèmes ont les mêmes membres gauches d'inéquations (correspondant aux arguments t_i du but $q(t_1,....t_m)$).

En simulant tous les appels possibles, nous obtenons les systèmes d'inéquations (expression de contraintes d'unification):

$$\underset{j=1}{\overset{k}{ou}} (t_1 \leq t'_{1j} \,\&\, t_2 \leq t'_{2j} \,\&\, ... \,\&\, t_m \leq t'_{mj}) = s_1 \text{ ou } .... \text{ ou } s_k$$

Exemple: Pour nos exemples, les simulations de l'unification et de tous les appels possibles produisent les systèmes d'inéquations suivants:

Pour p: $f(a) \leq any_0 \,\&\, X \leq any_0$ (1)

Pour q: $Y \leq a \,\&\, b \leq b$ (2)

ou

$Y \leq b \,\&\, b \leq c$ (3)

3.4.1.3. Transformation des systèmes d'inéquations

Filtrage:

Lors de la précédente étape nous avons produit k systèmes d'inéquations. Parmi ces systèmes nous avons éventuellement des systèmes impossibles correspondant à des unifications impossibles. Tout système d'inéquations contenant une (ou des) inéquation(s) de la forme suivante est un système impossible:

1) $a \leq f(t_1, ..., t_n)$ avec a un atome.

2) $a \leq b$ avec a et b des atomes différents.

3) $t_0 \leq t_1 + ... + t_n$ (ou r ! $(t_1+...+t_n)$) et $t_0 \notin I(t_1+...+t_n)$ ($\notin I(r!(...))$ t_0 est sans variable.

4) $X \leq t_1$ & ... & $X \leq t_n$ et $I(t_1) \cap ... \cap I(t_n) = \varnothing$.

5) $f(t_1,....t_n) \leq a$ et a est un atome.

6) $f(t_1,....t_n) \leq g(t'_1,....t'_m)$ avec $f \neq g$, $n \neq m$ ou une inéquation $t_i \leq t'_i$ est impossible.

7) L'appel d'un prédicat non défini.

Toutes ces situations correspondent à des situations d'impossiblité d'unification. Dans la 3ème forme par exemple, une telle inéquation est impossible puisque le membre droit de l'inéquation correspond au type d'un argument qui est sur-ensemble de l'ensemble de succès de cet argument. Si le membre gauche n'appartient pas à ce sur-ensemble, il n'appartient pas à l'ensemble de succès et donc l'unification serait inutile. Pour la 4ème forme, si l'intersection est vide alors la variable X ne peut pas avoir d'unification, pour une exécution réussie, avec un terme constant en ses différentes occurrences.

Remarque: Ce filtrage n'élimine pas tous les systèmes impossibles.

Exemple:

Pour les exemples que nous déroulons, nous remarquons que le système (3) ($Y \leq b$ & $b \leq c$) est impossible. Il contient l'inéquation $b \leq c$ de la catégorie (2).

Normalisation:

Le terme t_i d'une inéquation $t_i \leq t'_i$ d'un système s_j peut être un atome, une variable ou une structure $f(b_1, ... b_n)$.

Soit par exemple le système $a \leq any_1$ & $X \leq any_1$. L'élément de type any_1 représente une partie quelconque de l'Univers de Herbrand. (L'Univers de Herbrand contient a). Supposons que l'interprétation de cette partie ne contienne pas l'atome a. L'inéquation $a \leq any_1$ est donc impossible, et par conséquent le système est à éliminer. Si cette interprétation contienne l'atome a alors l'inéquation $a \leq any_1$ est possible. La seule condition que cette interprétation doit vérifier est qu'elle contienne l'atome a. La partie de l'Univers de Herbrand formée seulement de la constante peut être solution, c'est la solution optimale. Pour avoir donc une plus grande précision dans de tels systèmes nous remplaçons l'élément de type any_1 par a, puisque seule la constante a peut être solution. Nous obtenons alors le système $a \leq a$ & $X \leq a$. Nous remarquons que le membre droit de la seconde inéquation est plus précis dans ce dernier système que dans le précédent.

Le membre gauche d'une inéquation peut être une structure contenant des variables. Soit par exemple l'inéquation $f(a, X) \leq f(a, b)$. Pour obtenir le domaine de la variable X, nous décomposons puis nous normalisons ce système. Nous obtenons alors le système $a \leq a$ & $X \leq b$. Supposons que le membre droit de l'inéquation soit any_1. Cet élément de type représente une partie quelconque de l'Univers de Herbrand. Cette partie peut être séparée en deux classes: les termes constants ayant la forme $f(any_2, any_3)$ et les termes constants n'ayant pas cette forme c'est-à-dire ayant un foncteur et/ou une arité différent(s). Tous les termes constants appartenant à la seconde classe ne peuvent pas être solutions puisque leur interprétation ne peut en aucun cas contenir des termes constants ayant comme foncteur f et comme arité 2. Nous remplaçons donc any_1 par $f(any_2, any_3)$.

Exemples: 1) Normaliser les deux systèmes suivants:

$[] \leq any_0$ & $X \leq any_0 \rightarrow [] \leq []$ & $X \leq []$

$f(a,Y) \leq any_0 \& X \leq any_0 \rightarrow a \leq any_1 \& Y \leq any_2 \& X \leq f(any_1, any_2) \rightarrow a \leq a \& Y \leq any_2 \& X \leq f(a, any_2)$

2) Normalisons les systèmes d'inéquations des exemples que nous déroulons. Pour le premier système, nous avons une inéquation ayant

comme membre gauche une structure. La normalisation du premier système donne le nouveau système.

$f(a) \leq f(any_2)$ & $X \leq f(any_2)$ puis $a \leq any_2$ & $X \leq f(any_2)$ puis $a \leq a$ & $X \leq f(a)$

Le système (2) reste inchangé c'est-à-dire $Y \leq a$ & $b \leq b$.

Pour obtenir donc une plus grande précision des types inférés lorsque t_i est un atome et obtenir les types des variables apparaissant dans t_i lorsque t_i est une structure, nous normalisons les systèmes d'inéquations, en éliminant toujours les systèmes impossibles. Pour normaliser un système d'inéquations nous utilisons les règles suivantes (en interprétation ensembliste):

Soit une inéquation $t_i \leq t'_i$.

* **Si** t_i est une variable **alors** l'inéquation $t_i \leq t'_i$ reste inchangée.
* **Si** $t'_i = t'_{i1} + ... + t'_{ik}$ **alors** normaliser $t_i \leq t'_{ij}$ pour j=1,k. (†)
  Appliquer l'opération d'union + aux inéquations résultats.
  Si une inéquation a un membre droit = $\emptyset$ **alors** $t_i \leq \emptyset$.
* **Si** $t'_i = r!\ (t'_{i1}+...+t'_{ik})$ **alors** normaliser $t_i \leq t'_{ij}$ pour j=1,k. En remplaçant r par (††)
  $t'_{i1}+...+t'_{ik}$ chaque fois que c'est nécessaire en renommant les any_i.
  Appliquer l'opération d'union + aux inéquations résultats.
  Si une inéquation a un membre droit = $\emptyset$ **alors** $t_i \leq \emptyset$.
* **Si** t_i est un atome a **alors**:
  Si $t'_i = any_j$ **alors** nous remplacons toutes les occurrences de any_j par l'atome a (*)
  Sinon Si $t_i \in I(t'_i)$ **alors** l'inéquation reste inchangée.
  Sinon impossible. L'inéquation devient $t_i \leq \emptyset$.
* **Si** t_i est une structure $f(b_1, ..., b_n)$ **alors**:
  Si $t'_i = f(a_1, ..., a_n)$ **alors** normaliser les inéquations $b_i \leq a_i$ i=1,n.
  Si une inéquation est $b_i \leq \emptyset$ **alors** $t_i \leq \emptyset$.
  Si t'_i est any_j **alors** remplacer any_j par $f(any_1,....,any_n)$ dans ce système. (**)
  normaliser $b_k \leq any_k$ k=1,n.
  Sinon impossible. L'inéquation devient $t_i \leq \emptyset$

Remarque: Si dans un système, une inéquation a la forme $t_i \leq \emptyset$ alors ce système est impossible.

Dans la forme (†), la normalisation est appliquée aux éléments de la disjonction dont chacun correspond à une combinaison d'appel possible.

Dans la forme (††), la normalisation est appliquée aussi à chaque élément de la disjonction en remplaçant le symbole de récursion r par toute l'expression (éventuellement après renommage des any_i). Le seul cas où le renommage d'un any_i n'est pas effectué (nous gardons le même nom du any_i) est celui où il n'apparaît dans la partie droite d'une inéquation que sous la forme de r ! (... any_i .. + r + ...). Dans ce cas la récursion assure la liaison des mêmes noms de variables.

Dans la règle (*), le remplacement de any_i par l'atome est correct puisque seule la partie de l'Univers de Herbrand formée de l'atome t_i peut être solution. Tous les autres termes constants sont des situations d'échec.

Dans la règle (**) le remplacement de any_j par $f(any_1, ..., any_n)$ est correct puisque seuls les termes constants de la forme $f(\alpha_1, ... \alpha_n)$ peuvent être solution. Les interprétations de tous les autres éléments de type n'ayant pas la forme de $f(a_1, ..., a_n)$, ne peuvent en aucun contenir des termes constants qui sont instances de $f(b_1, ..., b_n)$.

Un remplacement d'un any_j par t dans un élément de type récursif r ! (t_1 + .. any_j .. + t_n) donne t_1+..t..+t_n + r!(t_1+..any_k..+t_n) sauf dans le cas où any_j n'apparaît dans le membre droit d'une inéquation que sous la forme r ! (...any_j..+ r + ...) auquel cas le remplacement donne r ! (...t.. + r + ...).

Inéquations productives et inéquations contraintes:

Après normalisation, le terme t_i d'une inéquation $t_i \leq t'_i$ est un atome ou une variable (variable de type ou variable n'apparaissant que dans le corps de la clause). Les t_i sont les termes syntaxiques du corps de la clause. Seules les variables de type sont utilisées directement dans le calcul du type de la tête. Les variables n'apparaissant que dans le corps de la clause sont utilisées éventuellement pour augmenter la précision des types inférés.

C'est pourquoi nous séparons les inéquations de chaque système en deux classes: Inéquations productives et Inéquations contraintes.
- La classe des inéquations productives est formée des inéquations dont les membres gauches sont des variables de type (variables apparaissant dans la tête et dans le corps de la clause).
- La classe des inéquations contraintes est formée du reste des inéquations (le membre gauche est un atome ou une variable n'apparaissant que dans le corps de la clause).

Exemple: Pour les exemples que nous déroulons, nous obtenons:

	Inéquations productives	inéquations contraintes
Pour p:	$X \leq f(a)$	$a \leq a$
Pour q:	$Y \leq a$	$b \leq b$

Vérification de la classe d'inéquations contraintes

Nous vérifions pour chaque système la possibilité ou non de sa classe d'inéquations contraintes. Cette classe peut être formée de deux catégories d'inéquations:

- Inéquations de la forme $a \leq t'_i$. Nous vérifions si l'atome $a \in I(t'_i)$.

- Inéquations ayant une même variable dans leurs membres gauches. Si nous avons n inéquations $X \leq t_i$ $(i=1,n)$ avec $n > 1$, alors nous vérifions si l'intersection $I(t_1) \cap ... \cap I(t_n)$ est vide ou non. Si pour une variable Y, $n=1$ alors nous éliminons cette inéquation (Y est une variable indéfinie).

Si une classe d'inéquations contraintes est possible alors nous prenons la classe d'inéquations productives correspondante comme solution, sinon nous éliminons le système correspondant. La classe d'inéquations contraintes est impossible si elle contient une inéquation de la forme:

- $a \leq t_j$ et $a \notin I(t_j)$. L'interprétation de t_j ne peut pas contenir l'atome a.
- $X \leq t_1$ & ... & $X \leq t_n$ et $I(t_1) \cap ... \cap I(t_n) = \emptyset$ avec $n>1$. Ceci correspond à l'impossiblité de trouver au moins un terme constant en commun aux interprétations des types de la variable dans ses différentes occurrences.

Exemple:

Pour l'exemple que nous déroulons, nous remarquons que les deux classes des deux systèmes sont possibles et donc les deux classes des inéquations productives sont des solutions.

Opération d'union et solutions maximales:

Après la séparation des inéquations en deux classes et élimination des systèmes impossibles, nous obtenons m classes d'inéquations productives à n inéquations chacune. Les membres gauches des inéquations sont les mêmes pour toutes ces classes et correspondent aux variables de type des termes syntaxiques du corps de la clause. Nous appliquons l'opération d'union aux $j^{èmes}$ inéquations de ces m classes pour j=1,n. Nous prenons ensuite les solutions maximales.

$$\begin{array}{cccccccc} X_1 \le t_{11} & & X_1 \le t_{21} & & \dots & & X_1 \le t_{m1} & X_1 \le t_{11} + t_{21} + \dots + t_{m1} \\ \dots & + & \dots & + & \dots & + & & \rightarrow \\ X_n \le t_{1n} & & X_n \le t_{2n} & & \dots & & X_n \le t_{mn} & X_n \le t_{1n} + t_{2n} + \dots + t_{mn} \end{array}$$

Cette opération d'union est correcte car si la valeur prise par X_i appartient à $I(t_{ij})$ alors cette valeur appartient toujours à $I(t_{i1} + \dots + t_{im})$ (voir définition de l'interprétation), et grâce aussi à la propriété de l'intersection d'ensemble suivante:

Propriété: L'ensemble $I(t_{11} + \dots + t_{m1}) \cap \dots \cap I(t_{1k} + \dots + t_{mk})$ contient toujours l'ensemble $I(t_{11}) \cap \dots \cap I(t_{1k}) \cup \dots \cup I(t_{1k}) \cap \dots \cap I(t_{mk})$.

La solution maximale pour chaque inéquation est prise comme étant

$$X_i = t_{1i} + \dots + t_{mi}. \quad i=1,n.$$

Nous obtenons m solutions maximales. Dans le cas où une variable de type apparaît dans plusieurs occurrences dans le corps de la clause alors elle a des solutions maximales multiples.

Exemple: Pour les exemples que nous déroulons, l'application de l'opération d'union donne comme solutions maximales:

Pour p: $X = f(a)$

Pour q: $Y = a$

Remplacement des variables de type.

Nous remplaçons les variables de type des membres droits des équations de type par leurs solutions maximales. Soit l'équation de type $T_{kl} = \psi(X_i)$ contenant une variable de type X_i dans son membre droit. Supposons que cette variable ait n solutions maximales $X_i = t_{ij}$ pour j=1,n. Nous produisons alors n équations de type

$$T_{kl} = \psi(t_{ij}) \text{ pour } j=1,n.$$

Nous appliquons ce remplacement pour toutes les variables de type apparaissant dans le membre droit d'une équation de type. A la fin de ces remplacements nous aurons k équations de type résolues pour une composante de type T_{ij}.

Exemple: Pour les exemples que nous déroulons, nous obtenons:

Pour p: $T_{11} = f(any_1, f(a))$

Pour q: $T_{11} = a.$

3.4.1.4. Résolution finale des équations de type

Une équation de type obtenue après le remplacement des variables de type est de la forme:

$$T_{ij} = Exp$$

$$Exp = atome \,/\, any_i \,/\, t_1 + ... + t_n \,/\, r\,!\,(t_1 + ... + t_n) \,/\, f(Exp^*)$$

Nous avons dit au paragraphe 3.2.2 qu'une équation de type est résolue lorsque son membre droit est un élément de type. C'est le cas des équations de type obtenues, donc toutes ces équations de type sont résolues. Pour obtenir l'équation de type unique d'une composante de type lorsque celle-ci a des solutions multiples, nous calculons les intersections des membres droits de ses équations. Ce qui revient à faire l'intersection des éléments de type.

Intersection:

L'intersection des interprétations de deux éléments de type est calculée utilisant les règles commutatives suivantes (en interprétation ensembliste):

* $any_i \cap any_j = any_k$.
* $t \cap any_i = t$ pour tout t terme constant.
* $t \cap t_1 + ... + t_n = t \cap t_1 + t \cap t_2 + ... + t \cap t_n$
* $f(t_1, ..., t_n) \cap g(t'_1, ..., t'_m)$ = **si** f=g **et** n=m **alors** répéter (i=1,n) $tt_i = t_i \cap t'_i$
  si un $tt_j = \emptyset$ **alors** $\emptyset$
  sinon $f(tt_1, ..., tt_n)$
  sinon $\emptyset$
* $atome_1 \cap atome_2$ = { **si** $atome_1 = atome_2$ **alors** $atome_1$ **sinon** $\emptyset$ }.

Dans le cas où $t = r\ !\ (t_1 + ... + f(r) + ... + t_n)$ et $t' \neq r\ !\ (...)$, l'intersection de $I(t)$ avec $I(t')$ revient à faire $I(t_1) \cap t' + I(t_2) \cap I(t') + ... + I(t_n) \cap I(t') + I(t') \cap I(f(r))$, en remplaçant le symbole r par l'expression $t_1 + ... + f(r) + ... + t_n$ à chaque fois que c'est nécessaire et en mémorisant à chaque fois le couple (t, r). Pour éviter éventuellement un remplacement infini de r par son expression, nous vérifions à chaque remplacement lors du calcul de l'intersection $t \cap r$, si le couple (t, r) est mémorisé ou non. S'il est mémorisé alors $r \cap t = \emptyset$. Sinon nous mémorisons le couple (t, r) et nous remplaçons le symbole r par son expression.

Exemples:

$$a \cap r\ !\ (b + r) = \emptyset$$

$$s(s(0)) \cap r\ !\ (0 + s(r)) = s(s(0)).$$

Dans le cas où les deux éléments de type sont récursifs de la forme $r_1\ !\ (t_1 + ... + f(r_1) + ... + t_n)$ et $r_2\ !\ (t'_1 + ... g(r_2) + ... + t'_n)$, alors l'intersection est calculée de la manière suivante: Nous mémorisons l'ensemble $\{r_1, r_2\}$. Nous calculons les intersections $t_i \cap t'_j$ pour i=1,n et j=1,m en utilisant les règles commutatives précédentes. Il reste le cas $f(r_1) \cap (t'_1 + ... + g(r_2) + ... + t'_m)$ (et $g(r_2) \cap (t_1 + ... + f(r_1) + ... + t_n)$ respectivement). Les intersections des interprétations de $f(r_1)$ ($g(r_2)$ respectivement) avec t'_i (t_i respectivement) pour i=1,m (i=1,n respectivement) sont calculées de la même manière que le cas précédent. Il nous reste le cas $I(f(r_1)) \cap I(g(r_2))$.

Nous calculons cette intersection en utilisant les règles commutatives précédentes, mais avant de remplacer r_1 et r_2 simultanément par leurs expressions respectives, nous vérifions si l'ensemble $\{r_1, r_2\}$ est mémorisé ou non. S'il est mémorisé alors nous ne remplaçons pas r_1 et r_2 par leurs expressions, nous avons dans ce cas une récursion r_{12} dans l'intersection résultat de r_1 ! (...) $\cap$ r_2 ! (...). S'il n'est pas mémorisé alors nous remplaçons ces deux symboles par leurs expressions respectives, nous mémorisons l'ensemble $\{r_1, r_2\}$ et nous continuons le calcul de l'intersection.

Cette mémorisation est effectuée pour éviter au calcul d'intersection de boucler. Il se peut que l'ensemble intersection des interprétations des deux éléments de type récursifs, soit un ensemble infini.

Exemple: $I(r_1 ! (0 + s(r_1))) \cap I(r_2 ! (s(0) + s(s(r_2)))) = r_{12} ! (s(0) + s(s(r_{12})))$.

Pour les exemples que nous déroulons, chaque composante de type n'a qu'une seule équation et donc nous ne calculons pas d'intersection.

Simplification:

Chaque composante de type a une équation dont le membre droit est le type de l'argument correspondant. Nous appliquons des simplifications élémentaires à ces résultats. Les simplifications effectuées sont les suivantes:

* $t_1 + .. + any_i + ... t_n = any_i$
* $t_1 + ... + t_i + ... + t_{j-1} + t_i + t_{j+1} + ... + t_n = t_1 + ... + t_i + ... + t_{j-1} + t_{j+1} + ... + t_n$.

3.4.1.5. Exemple d'inférence de type

Soit la clause à un seul but à 3 définitions
p(g(V,a), X, f(X,Y)):-q(a, X, b, Z1, f(X,Y), Z1). Supposons que le type de q soit donné par:

q(any_1, any_1, b, f(any_3,any_4), any_2, any_2) ou
q(any_5, a, any_5, c, f(e,f), c) ou
q(a, b, b, c, f(b,d), c)

* Nous produisons les équations de type pour la tête de la clause:

$$T_{11} = g(any_0,a)$$
$$T_{21} = X$$
$$T_{31} = f(X,Y)$$

où X, Y sont les variables de type.

* Les simulations de l'unification et de tous les appels possibles donnent:

$a \le any_1$ & $X \le any_1$ & $b \le b$ & $Z1 \le f(any_3,any_4)$ & $f(X,Y) \le any_2$ & $Z1 \le any_2$ (1) ou

$a \le any_5$ & $X \le a$ & $b \le any_5$ & $Z1 \le c$ & $f(X,Y) \le f(e,f)$ & $Z1 \le c$ (2) ou

$a \le a$ & $X \le b$ & $b \le b$ & $Z1 \le c$ & $f(X,Y) \le f(b,d)$ & $Z1 \le c$ (3)

* Le premier filtrage n'élimine aucun système.

* Nous normalisons ces systèmes d'inéquations et nous obtenons:

$a \le a$ & $X \le a$ & $b \le b$ & $Z1 \le f(any_3,any_4)$ & $X \le any_7$ & $Y \le any_8$ & $Z1 \le f(any_7,any_8)$ (1) ou

$a \le a$ & $X \le a$ & $b \le a$ & $Z1 \le c$ & $X \le e$ & $Y \le f$ & $Z1 \le c$ (2) ou

$a \le a$ & $X \le b$ & $b \le b$ & $Z1 \le c$ & $X \le b$ & $Y \le d$ & $Z1 \le c$ (3)

Le système (2) est impossible car il contient l'inéquation $b \le a$.

* Nous séparons les inéquations de chaque système (1) et (3) en classe d'inéquations productives et classe d'inéquations contraintes (X et Y sont les variables de type):

Inéquations productives	Inéquations contraintes	
$X \le a$ & $X \le any_7$ & $Y \le any_8$	$a \le a$ & $b \le b$ & $Z1 \le f(any_3,any_4)$ & $Z1 \le f(any_7,any_8)$	(1)
$X \le b$ & $X \le b$ & $Y \le d$	$a \le a$ & $b \le b$ & $Z1 \le c$ & $Z1 \le c$	(3)

* En vérifiant les classes d'inéquations contraintes, les deux classes des deux systèmes (1) et (3) sont possibles.

* Nous appliquons l'opération d'union + aux classes d'inéquations productives des systèmes (1) et (3) et nous obtenons comme solutions maximales:

$$X = a + b$$
$$X = any_7 + b$$
$$Y = any_8 + d$$

* Nous remplaçons ces solutions maximales dans les équations de type. La variable X a deux solutions et donc nous produisons deux équations pour chaque équation de type contenant cette variable dans son membre droit. Nous produisons deux équations pour chacune des équations de type $T_{21} = X$ et $T_{31} = f(X, Y)$.

$$T_{11} = g(any_0, a)$$
$$T_{21} = a + b$$
$$T_{21} = any_7 + b$$
$$T_{31} = f(a+b, any_8+d)$$
$$T_{31} = f(any_7+b, any_8+d)$$

* Les composantes de type T_{21} et T_{31} ont des équations multiples. Nous calculons les intersections des interprétations de leurs membres droits:

$$T_{21} = a + b + b$$
$$T_{31} = f(a+b+b, any_9+d+d+d)$$

* et donc (après simplification)

$$T_{11} = g(any_0, a)$$
$$T_{21} = a + b$$
$$T_{31} = f(a+b, any_9+d) = f(a+b, any_{10})$$

3.4.2. Cas général.

Nous passons maintenant au cas général où le nombre de buts du corps de la clause est $n > 1$. Nous déterminons les équations de type de la tête de la clause, nous produisons les inéquations pour chaque but, nous effectuons un filtrage et nous normalisons les systèmes d'inéquations pour chaque but comme au cas précédent. Pour chaque but p_i ayant n_i systèmes, nous obtenons donc l'expression: $SS_i = s_1$ ou s_2 ou ... ou s_{n_i}.
Pour les m buts du corps de la clause, nous obtenons l'expression:

$$SS_1 \ \& \ SS_2 \ \& \ ... \ \& \ SS_m$$

Eclatement de ou et de &

Nous effectuons l'éclatement de ou et de &. Si nous avons une disjonction de k_i systèmes normalisés s_i^j $(j=1.k_i)$ pour le but p_i alors l'éclatement donne Πk_i $i=1.n$ (produit des k_i) nouveaux systèmes d'inéquations. Tous ces systèmes ont les mêmes membres gauches de leurs inéquations correspondant aux termes syntaxiques du corps de la clause. L'arité d'un système issu de l'éclatement est égale à la somme des arités des systèmes des p_i. L'éclatement se fait de la manière suivante:

$$\text{ou } s_1^{j_1} \ \& \ \text{ou } s_2^{j_2} \ \& \ ... \ \& \ \text{ou } s_n^{j_n} \quad \text{pour } j_i = 1. \ k_i \text{ et } i=1.n$$

$$= (s_1^1 \text{ ou } ... \text{ ou } s_1^{k_1}) \ \& \ .... \ \& \ (s_n^1 \text{ ou } s_n^2 \text{ ou } ... \text{ ou } s_n^{k_n}) =$$

$$(s_1^1 \ \& \ s_2^1 \ \& \ ... \ \& \ s_n^1) \text{ ou } ... \text{ ou } (s_1^{j_1} \ \& \ s_2^{j_2} \ \& \ ... \ \& \ s_n^{j_n}) \text{ ou } ... \text{ ou } (s_1^{k_1} \ \& \ s_2^{k_2} \ \& \ ... \ \& \ s_n^{k_n}$$

en renommant les any_i à chaque passage d'un système $s_i^{j_i}$ à un autre. Ce renommage correspond au renommage des variables en Prolog à chaque passage d'une clause à une autre.

En Prolog nous renommons toutes les variables (apparaissant dans la tête seulement, dans le corps seulement ou dans la tête et dans le corps simultanément). Seules les variables apparaissant dans la tête de la clause seulement ayant le type any_i, apparaissent dans les éléments de type inférés. Une variable de type est remplacée par son type ce qui revient à renommer les any_i de ce type. Les variables n'apparaissant que dans le corps de la

clause ne sont utilisées que pour le calcul des intersections des interprétations des membres droits de leurs inéquations. A chaque passage d'une clause à une autre il faut donc renommer les any_i.

Second filtrage:

Nous effectuons le second filtrage sur les systèmes ss_i obtenus. Ce second filtrage est effectué pour détecter les situations d'échec produites lors de la présence d'une même variable dans deux buts.

Nous séparons les inéquations des système ss_i obtenus en classe d'inéquations productives et classe d'inéquations contraintes, nous vérifions les classes d'inéquations contraintes en eliminant les systèmes éventuellement impossibles, nous appliquons l'opération d'union aux classes d'inéquations productives, nous prenons les solutions maximales, nous remplaçons dans les équations de type les variables de type par ces solutions maximales, nous calculons les intersections puis nous simplifions, comme au paragraphe précédent. Nous obtenons ainsi le type du prédicat p.

Bilan: Pour généraliser nous avons introduit la notion *d'éclatement de ou et de &*, puis *l'application du filtrage une seconde fois.*

Exemple d'inférence de type

Soit la clause p(X,Y,f(V,X)):-q(X, Z, b), r(Z, Y, c). Supposons que les types de q et r soient donnés par:

q_1(a+f, d, any_1 + e) ou
q_2(e+f, any_2 + g, any_2 + f) ou
q_3(f, any_4, c).

r_1(b, any_5, any_5). ou
r_2(b, d, c)

* Les équations de type sont:

$T_{11} = X$
$T_{21} = Y$
$T_{31} = f(any_0, X)$

* Après production , élimination des systèmes éventuellement impossibles et normalisation des inéquations comme au paragraphe précédent, nous obtenons les systèmes d'inéquations suivants:

pour q $(X \leq a+f \;\&\; Z \leq d \;\&\; b \leq b+e \text{ ou } X \leq e+f \;\&\; Z \leq b+g \;\&\; b \leq b+f)$

pour r $(Z \leq b \;\&\; Y \leq c \;\&\; c \leq c \text{ ou } Z \leq b \;\&\; Y \leq d \;\&\; c \leq c)$

* Nous éclatons les deux opérations + et &, et nous obtenons:

$(X \leq a+f \;\&\; Z \leq d \;\&\; b \leq b+e \;\&\; Z \leq b \;\&\; Y \leq c \;\&\; c \leq c)$ (1)

$(X \leq a+f \;\&\; Z \leq d \;\&\; b \leq b+e \;\&\; Z \leq b \;\&\; Y \leq d \;\&\; c \leq c)$ (2)

$(X \leq e+f \;\&\; Z \leq b+g \;\&\; b \leq b+f \;\&\; Z \leq b \;\&\; Y \leq c \;\&\; c \leq c)$ (3)

$(X \leq e+f \;\&\; Z \leq b+g \;\&\; b \leq b+f \;\&\; Z \leq b \;\&\; Y \leq d \;\&\; c \leq c)$ (4)

* L'application du filtrage une seconde fois entraîne que les systèmes (1) et (2) sont impossibles. $(I(d) \cap I(b) = \varnothing)$ dans $Z \leq d \;\&\; Z \leq b$.

* Nous séparons les inéquations des systèmes restants en classe d'inéquations productives et classe d'inéquations contraintes.

Inéquations productives	Inéquations contraintes	
$X \leq e+f \;\&\; Y \leq c$	$Z \leq b+g \;\&\; b \leq b+f \;\&\; Z \leq b \;\&\; c \leq c$	(3)
$X \leq e+f \;\&\; Y \leq d$	$Z \leq b+g \;\&\; b \leq b+f \;\&\; Z \leq b \;\&\; c < c$	(4)

* Nous vérifions que les classes d'inéquations contraintes sont possibles. Les classes d'inéquations productives des systèmes (3) et (4) sont donc solutions.

* Nous appliquons l'opération d'union + à ces deux classes, nous prenons les solutions maximales et nous obtenons:

$$X = e+f+e+f$$

$$Y = c + d$$

* Nous remplaçons ces variables de type dans les équations de type puis nous simplifions (pas de calcul d'intersection):

$$T_{11} = X = e + f + e + f = e + f$$

$$T_{21} = Y = c + d$$

$$T_{31} = f(any_0, e+f+e+f) = f(any_0, e+f)$$

Bilan général:

Connaissant le type des buts du corps d'une clause p:-..., nous avons déterminé le type de la tête p. Ce type peut être à son tour utilisé pour calculer le type d'un prédicat qui appelle p (exemple q :- ...p..).

3.4.3. Utilisation des types déjà calculés

Si une composante de type T_{ij} d'un argument, avant le calcul de l'intersection des interprétations des membres droits de ses équations de type, a des équations multiples $T_{ij}=t_k$ pour $k=1,n$ contenant des any_i alors lors de la génération des inéquations (à l'appel de p), nous produisons l'inéquation (pour cet argument):

$$X \leq t_1 \ \& \ ... \ \& \ t_n.$$

Lors du filtrage et de la normalisation cette inéquation est équivalente à:

$$X \leq t_1 \ \& \ X \leq t_2 \ ... \ \& \ X \leq t_n.$$

Lors de l'application de l'opération d'union + aux classes d'inéquations productives, si une inéquation de la $i^{ème}$ classe d'inéquations productives parmi les $X \leq t_i$ est donnée par

$$X \leq d_1 \ \& \ ... \ \& \ d_n$$

alors nous obtenons n solutions maximales, données par

$$X = t_1 + ... + t_{i-1} + d_j + t_{i+1} + ... + t_n \quad \text{pour } j=1,n.$$

Nous exploitons combinatoirement les résultats.

Si les membres droits des équations de type avant l'intersection ne contiennent pas de any_i alors c'est le résultat S du calcul de l'intersection simplifié qui est utilisé comme type. Nous produisons alors $X \leq S$.

Cette distinction est faite afin de garantir la propagation des nouvelles formes éventuelles des any_i.

Nous pouvons maintenant déterminer le type des prédicats dans le cas où nous n'avons pas d'appels récursifs. Nous présenterons dans le prochain paragraphe le cas où nous avons des appels récursifs.

3.5. Clauses à corps incomplètement typé.

Nous passons maintenant au cas où les buts du corps d'une clause sont incomplètement typés. Soit l'exemple suivant:

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

Pour calculer le type de p il faut déterminer le type de q. Pour déterminer le type de q il faut déterminer celui de r. Pour déterminer le type de r il faut déterminer celui de p que nous sommes en train de calculer, et qui est inconnu. Ceci correspond au cas où nous avons au moins un appel récursif.

Lors de l'inférence du type du prédicat p et au moment de l'unification du but du corps de la dernière clause, nous supposons que le type de p est connu et est donné par les composantes de type de la seconde clause. Nous pouvons alors produire le système d'inéquation correspondant à l'unification du but p de la dernière clause avec la tête de la seconde clause. Nous arrêtons ainsi l'itération. *C'est le traitement spécial des appels récursifs*. Nous effectuons le même traitement que précédemment en utilisant toutes les notions introduites dans le cas précédent. Les opérations telles que normalisation, vérification de la classe d'inéquations contraintes, résolution des équations de type etc, sont éventuellement élargies au fur et à mesure de nos besoins.

Nous introduisons de nouvelles notions telles que: traitement d'appels récursifs, hypothèses, contraintes, décomposition.

Nous commençons par le cas simple de récursivité (directe, ou plate). Nous passerons ensuite au cas où la récursion est indirecte.

3.5.1. Récursion simple (ou plate).

Nous présentons d'abord le cas où la récursion ne se fait qu'une seule fois dans le corps de la clause. Nous généralisons ensuite au cas où l'appel récursif se répète dans le corps de la clause.

3.5.1.1. Les équations de type.

La récursion en général dans ce cas se fait de la manière suivante: $p(a_1,....,a_n)$:-...., $p(t_1,....,t_n)$,..... Supposons que les composantes de type de cette clause soient données par $< T_{1j}, ..., T_{nj} >$. Nous produisons les équations de type de la tête de la clause comme aux paragraphes précédents. Elles sont de la forme $T_{ij} = a_i$ j=1,m où les variables n'apparaissant que dans la tête de la clause sont remplacées par des any_k.

3.5.1.2. Unification de type

Exemple: Soit l'exemple suivant

entier(0).

entier(s(X)):-entier(X).

Les composantes de type sont produites de la même manière qu'aux paragraphes précédents, et sont données par:

$$T_{11} = 0$$

$$T_{12} = s(X)$$

où X est une variable de type qu'il faut remplacer par son type obtenu après traitement du corps de la seconde clause. La production des inéquations se fait de la manière suivante:

* Unification de entier(X) avec sa première clause donne: $X \leq 0$.
* Unification de entier(X) avec sa seconde clause donne: $X \leq s(X')$

où X' est une variable de type; nous sommes en train de déterminer le type de son prédicat ($s(X) \equiv s(X')$ à un renommage des variables près). Or, que constatons-nous? le membre droit de l'inéquation, s(X'), peut être remplacé par T_{12}, puisque l'on a justement $T_{12} = s(X)$. La composante de type du membre droit de l'inéquation sera remplacé lors de la résolution des éaqutions de type par un symbole de récursion. Nous renommons les any_i à chaque remplacement d'un symbole de récusion par son expression, ce qui correspond au renommage des variables de s(X) pour avoir s(X'). D'où **le traitement spécial des appels récursifs**: lors de la production des inéquations d'une clause j, remplacer $X \leq a_i$ par $X \leq T_{ij}$.

Cette nouvelle inéquation a la même sémantique que la précédente: la valeur prise par t_i appartient à l'interprétation de T_{ij}.

Dans de telles clauses, nous supposons que le type de ce but est connu c'est-à-dire qu'il aurait pu être calculé précédemment. Les types inférés seront sous une forme récursive. Le renommage des variables de a_i (pour avoir a'_i) correspond au renommage des any_i s'effectuant lors du remplacement du symbole de récursion r par $r\,!\,(t_1 + ... + f(r) + ... + t_n)$ dans f(r).

En traitant de manière particulière les appels récursifs, nous obtenons des inéquations de la forme $t_i \leq t'_i$ avec:

$$t'_i = atome \,/\, any_k \,/\, t'_1 + ... + t'_n \,/\, r\,!\,(\,...\,) \,/\, T_{kj} \,/\, f(t'_1, ..., t'_r)$$

Bilan 1:

Nous avons introduit une nouvelle notion qui est *le traitement spécial des appels récursifs*. Cette introduction implique la possibilité d'avoir des composantes de type dans les membres droits des inéquations.

3.5.1.3. Transformations des inéquations

Nous effectuons ensuite un filtrage comme aux cas précédents, sauf que dans une inéquation $t_i \leq t'_i$ contenant une composante de type dans le membre droit, nous ne pouvons rien affirmer actuellement sur l'impossibilité ou non du système.

Normalisation:

Une fois les systèmes d'inéquations produits et filtrés, nous les normalisons en utilisant la technique de normalisation précédemment décrite. Le seul cas nouveau est celui où nous avons une inéquation de la forme $f(b_1, ...,b_n) \leq T_{ij}$ ou $a \leq T_{ij}$.

Exemple (1): Soit l'exemple suivant:

```
p(K).
p(X) :- p(f(X)).
```

Les équations de type sont produites de la même manière que précédemment et sont données par:

$$T_{11} = any_0$$
$$T_{12} = X$$

Les inéquations produites pour la seconde clause sont données par (après application du traitement special des appels récursifs):

$$f(X) \leq any_1 \quad \text{ou} \quad f(X) \leq T_{12}$$

Comment normaliser le second système alors que le domaine de T_{12} est inconnu actuellement?

Dans ce cas nous introduisons une nouvelle notion: *la notion d'hypothèse*. Une hypothèse est de la forme $T'_k = \{ t / f(...., t, ...) \in I(T_{ij}) \}$. En utilisant cette notion d'hypothèse nous pouvons alors normaliser cette catégorie d'inéquations.

Pour notre exemple précédent, nous trouvons:

$$X \leq f(any_2) \quad \text{ou} \quad X \leq T'_1 \text{ avec } T'_1 = \{ t / f(t) \in I(T_{12}) \}$$

En ayant à normaliser $f(b_1, ... b_n) \leq T_{kj}$ nous normalisons les inéquations $b_i \leq T'_i$ avec $T'_i = \{ t_i / f(\_, ..., \_, t_i, \_, ..., \_) \in I(T_{kj}) \}$ comme variable d'hypothèse pour i=1,n. Si de nouveau nous avons à normaliser une inéquation de la forme $f(b_1, ..., b_n) \leq T'_i$ (où T'_i est une variable d'hypothèse) alors nous normalisons les inéquations $b_j \leq T'_j$ avec $T'_j = \{ t / f(\_, ..., \_, t, \_, ..., \_) \in I(T'_i) \}$.

Ces hypothèses seront prises en compte dès que c'est nécessaire, en utilisant une nouvelle notion: *la décomposition* que nous allons introduire plus loin.

Bilan 2: Nous avons élargie la fonction de Normalisation. Nous avons introduit la notion *d'hypothèse* et la notion de décomposition. Cette dernière sera présentée en détail plus loin.

La séparation des inéquations en classe d'inéquations productives et classe d'inéquations contraintes s'effectue comme dans le paragraphe précédent.

Vérification des classes d'inéquations contraintes:

Exemple (2): Soit l'exemple suivant:

```
p(K, K).
p(X, Y) :- q(X, Z), p(Z, Y).
q(a,b)
```

Les équations de type pour les deux premières clauses sont produites de la même manière que précédemment donnant:

$T_{11} = any_0$
$T_{21} = any_0$
$T_{12} = X$
$T_{22} = Y$ où X, Y sont des variables de type.

Les inéquations produites sont (après traitement des appels récursifs):

$X \leq a \;\&\; Z \leq b \;\&\; Z \leq any_1 \;\&\; Y \leq any_1$ ou $X \leq a \;\&\; Z \leq b \;\&\; Z \leq T_{12} \;\&\; Y \leq T_{22}$

Après filtrage, normalisation et séparation des inéquations en classe d'inéquations productives et classe d'inéquations contraintes, nous obtenons:

équations productives	Inéquations contraintes
$X \leq a \;\&\; Y \leq any_1$	$Z \leq b \;\&\; Z \leq any_1$
$X \leq a \;\&\; Y \leq T_{22}$	$Z \leq b \;\&\; Z \leq T_{12}$

Dans la vérification des classes d'inéquations contraintes, la classe du premier système est vérifiée de la même manière que précédemment. Dans la vérification de la classe du second système, nous ne pouvons pas calculer l'intersection $\{b\} \cap I(T_{12})$ puisque l'interprétation de la composante de type T_{12} n'est pas connue actuellement.

Nous vérifions donc les classes d'inéquations contraintes comme dans les cas précédents, sauf que les cas suivants peuvent se présenter:

- Cas où nous avons une inéquation de la forme $a \leq t$ où le membre droit t contient une composante de type T_{ij} et/ou une variable d'hypothèse T'_i. Dans ce cas $I(t)$ n'est pas connu actuellement, par conséquent nous ne pouvons pas décider éventuellement si l'atome a appartient à cette interprétation ou non.

- Cas où nous avons des inéquations de la forme $X \leq t_1 \;\&\; \ldots \;\&\; X \leq t_n$ où des membres droits t_j $j=1,2,\ldots$ contiennent des composantes de type T_{kl} et/ou des variables d'hypothèse T'_r. Dans ce cas nous ne pouvons pas calculer les intersections des interprétations des membres droits de ces inéquations.

Dans de telles situations, nous vérifions les inéquations de la classe d'inéquations contraintes qui ne sont pas des deux catégories précédentes et nous introduisons une nouvelle notion: *la notion de contrainte*. Nous disons que les inéquations restantes de la classe d'inéquations contraintes forment une contrainte C_j. Celle-ci sera vérifiée après la détermination des expressions de type et/ou des variables d'hypothèse.

Les inéquations de la classe d'inéquations productives sont donc solutions sous la contrainte C_j. Si cette contrainte est vraie alors les inéquations de la classe d'inéquations productives sont solutions. Une inéquation $X \leq t$ de la classe d'inéquations productives devient alors $X \leq t \backslash C_j$ (t sous la contrainte C_j).

Bilan 3: Nous avons introduit la notion de *contrainte* lors de la vérification de la classe d'inéquations contraintes.

Les variables membres gauches des $j^{ème}$ inéquations $j=1,n$ sont les mêmes respectivement dans toutes les classes d'inéquations productives. Elles correspondent aux variables de type des termes syntaxiques du but d'appel. Nous appliquons l'opération d'union + aux $j^{èmes}$ inéquations $X \leq t_j$ des n classes d'inéquations productives et nous obtenons $X \leq t_1 + ... + t_n$. Cette application est faite comme au paragraphe précédent sauf qu'un t_j peut être sous une contrainte C_k.

Nous prenons les solutions maximales et nous obtenons $X = t_1 + ... + t_n$. Pour une même variable nous pouvons avoir une définition multiple.

Nous avons des équations de type de la forme $T_{ij} = \psi(X)$ contenant des variables de type X à remplacer par leurs solutions maximales. Nous remplaçons ces variables de type par ces solutions maximales comme au cas précédent. Nous produisons autant d'équations de type $T_{ij} = \psi(t_i)$ qu'il y a de solutions maximales $X = t_i$ pour la variable de type X appartenant au membre droit de l'équation $T_{ij} = \psi(X)$. Nous appliquons ce traitement à toutes les variables de type de $\psi(X)$.

Pour chaque composante de type nous obtenons k équations $T_{ij} = t_r$ pour $r=1,k$ contenant des composantes de type T_{ij} et/ou des variables d'hypothèses

T'_i dans les membres droits.

$$t_r = \text{atome} / \text{any}_i / t_1 + \ldots + t_n / r\,!\,(\ldots) / T_{kl} / T'_i / f(t_i^*)$$

Ces équations de type contiennent des composantes de type et/ou des variables d'hypothèse dans leurs membres droits et ne sont donc pas encore résolues.

3.5.1.4. Résolution des équations de type

Une équation de type a la forme:

$T_{ij} = t_k$ avec:

$t_k = \text{atome} / \text{any}_r / t_1 + \ldots + t_m / r\,!\,(\ldots) / T_{rl} / T'_r / f(t_1, \ldots, t_m)$

Remarque: L'élément t_k peut être éventuellement sous une contrainte C_j, auquel cas le résultat de la résolution de t_i sera sous cette contrainte C_j.

Le but de la résolution est maintenant d'éliminer les composantes de type et/ou les variables d'hypothèse des membres droits des équations de type.

C'est au niveau de cette résolution qu'apparaissent les récursions dans les éléments de type. En ayant une équation de la forme $T_{ij} = \psi(T_{ij})$, nous remplaçons la composante de type de droite T_{ij} par le symbole de récursion r. Nous obtenons un élément de type récursif de la forme $r\,!\,\psi(r)$ comme membre droit de cette équation de type.

Si une composante de type T_{kl} apparaît dans le membre droit de l'équation de type $T_{ij} = \psi(T_{kl})$ alors nous vérifions si nous sommes en train de calculer son expression. Si oui nous avons alors une récursion à ce niveau (nous remplaçons T_{kl} par le symbole de recursion r), sinon nous renommons les any_i de T_{kl} et nous résolvons ses équations de type. En ayant m solutions S_r r=1,m pour cette composante de type T_{kl}, nous produisons m équations $T_{ij} = \psi(S_r)$ r=1,m pour l'équation de type $T_{ij} = \psi(T_{kl})$.

Si le membre droit d'une équation de type $T_{ij} = \psi(T'_r)$ contient une variable d'hypothèse T'_r alors nous cherchons ses domaines par décomposition. Si la décomposition donne k solutions t_r alors nous produisons k équations $T_{ij} = \psi(t_r)$ pour r=1,k.

A la fin de la résolution des équations de type, leurs membres droits sont des éléments de type et ces équations sont donc résolues.

Bilan 4: Nous avons élargi *la notion de résolution des équations de type.* Nous obtenons des éléments de type récursifs.

Décomposition:

Soit une variable d'hypothèse $T'_k = \{ t / f(..., t, ...) \in I(T_{ij}) \}$. Pour avoir le domaine de cette variable d'hypothèse T'_k nous décomposons chaque membre droit de l'équation de type $T_{ij} = t_i$ (en interpétation ensembliste) avec f(..., t, ...). Ce qui revient à calculer les éléments t. Pour que cette détermination ne boucle pas (dans le cas des expressions récursives), nous ajoutons un test de circularité indiquant si une décomposition a déjà été rencontrée auparavant. Si c'est le cas, nous prenons la solution maximale any_j. Une plus grande précision peut être obtenue en explorant à fond cette décomposition, ceci fera l'objet d'une amélioration envisageable du système.

Cette décomposition se fait suivant la nature de t_i.

- **Si** $t_i = a_1 + ... + a_n$ **alors** décomposer f(..., t, ...) avec a_i pour i=1,n.
- **Si** $t_i = r!(a_1 + ... + a_n)$ **alors** décomposer f(..., t, ...) avec a_i en remplaçant r par
  $r ! (a_1 + ... + a_n)$ chaque fois que c'est nécéssaire
  en renommant les any_i.
- **Si** t_i = atome **alors** ∅.
- **Si** $t_i = any_i$ **alors** $t = any_j$.
- **Si** $t_i = g(..., t', ...)$ **alors si** même foncteur **et** même arité de f et g **alors** t=t'.
  sinon ∅.
- **Si** $t_i = T_{kl}$ **alors si** nous sommes passé par cette décomposition
  alors $t=any_j$ (circularité → solution maximale).
  sinon mémorisation de cette décomposition et
  renommer les any_i des $T_{ij} = t \rightarrow T_{ij} = t'$.
  décomposition de f(..., t, ...) avec t'
- **Si** $t_i = T'_j$ variable d'hypothèse **alors** mémoriser cette hypothèse,
  la décomposer pour avoir ses domaines,
  Décomposer ses résultats avec f(..., t, ...).

Remarque: Un élément de t_i peut être sous une contrainte c_j auquel cas les résultats de la décomposition seront pris sous cette contrainte.

Exemple: Dans l'exemple (1) précédent, l'application de l'opération d'union + donne comme solution maximale:

$$X = f(any_2) + T'_1 \text{ avec } T'_1 = \{ t / f(t) \in I(T_{12}) \}$$

Le remplacement de X dans la seconde équation de type donne:

$$T_{12} = f(any_2) + T'_1$$

La résolution de cette équation de type utilise l'algorithme de décomposition pour avoir le domaine de la variable d'hypothèse T'_1. L'application de l'algorithme de décomposition donne:

$$T'_1 = any_3 + any_4$$

où any_3 provient de la décomposition de any_2 avec f(t) et any_4 est la solution maximale prise lors du succès du test de circularité de l'algorithme.

Bilan 5: Nous avons introduit la notion de *décomposition* pour obtenir les domaines (éventuellement multiples) d'une variable d'hypothèse.

Vérification des contraintes:

Lors de la séparation des inéquations d'un système en classe d'inéquations productives et classe d'inéquations contraintes, nous avons introduit la notion de contrainte. Une contrainte correspond à une classe d'inéquations contraintes dont nous ne pouvions pas vérifier ses inéquations car elles contenaient des composantes de type et/ou des variables d'hypothèse dans leurs membres droits. Maintenant que ces composantes de type sont connues, nous pouvons vérifier ces contraintes. Les éléments de type, membres droits des équations de type résolues sont sous ces contraintes c_j. Nous avons deux catégories d'inéquations dans ces contraintes:

- $a \leq \psi(T_{ij}, T'_k)$. Nous vérifions les contraintes de $\psi(T_{ij}, T'_k)$ puis nous vérifions si $a \in I(\psi(T_{ij}, T'_k))$.

- Des inéquations de la forme $X \leq t_1$ & ... & $X \leq t_n$ et les t_i contiennent des

contraintes. Nous vérifions les contraintes des membres droits de ces inéquations puis nous calculons les intersections de leurs interprétations.

Pour que cette vérification ne boucle pas, nous ajoutons un test de circularité indiquant si une contrainte à démontrer a déjà été rencontrée auparavant. Si c'est la cas, alors nous prenons la solution maximale (qui correspond à prendre l'élément de type sous cette dernière contrainte), sinon nous la mémorisons et nous la vérifions.

Exemple

Dans l'exemple (2) précédent, nous avions une classe d'inéquations contraintes donnée par: $Z \leq b \ \& \ Z \leq T_{22}$ et les deux équations de type $T_{22} = any_1 + T_{22}(C_2)$ et $T_{12} = a + a(C_2)$ avec $C_2 = Z \leq b \ \& \ Z \leq T_{12}$. L'application de l'algorithme de vérification des classes d'inéquations contraintes entraîne que C_2 est vraie (solution maximale prise lors du succès du test de circularité de l'algorihtme)

Bilan 6: Nous avons introduit la notion de *vérification de contraintes* dans le cas où les membres droits des inéquations de la classe d'inéquations contraintes contiennent des composantes de type et/ou des variables d'hypothèse.

L'intersection des interprétations des membres droits de deux équations de type résolues $T_{ij} = t$ et $T_{ij} = t'$ revient à faire l'intersection de $I(t)$ et $I(t')$. Cette intersection est calculée de la même manière qu'aux paragraphes précédents.

La simplification est effectuée de la même manière qu'aux paragraphes précédents.

3.5.1.5. Exemples

Exemple(1): Soit le programme suivant.

```
append([], K, K).
append([X|L], M, [X|N]) :- append(L, M, N).
```

* Les équations de type du prédicat append sont:

$$T_{11} = []$$
$$T_{21} = any_0$$
$$T_{31} = any_0$$
$$T_{12} = [any_1|L].$$
$$T_{22} = M$$
$$T_{32} = [any_1|N]$$

où L, M, N sont des variables de type qu'il faut remplacer par leurs types obtenus après traitement du corps de la seconde clause.

* Par application des simulations de l'unification et de tous les appels possibles avec traitement spécial des appels récursifs, nous obtenons les systèmes d'inéquations suivants:

$$\begin{array}{lll} L \leq [] & & L \leq T_{12} \\ M \leq any_2 & ou & M \leq T_{22} \\ N \leq any_2 & & N \leq T_{32} \end{array}$$

* En normalisant les deux systèmes, les inéquations restent inchangées.

* En séparant les inéquations en classe d'inéquations productives et classe d'inéquations contraintes, nous remarquons que les classes d'inéquations contraintes correspondant aux deux systèmes sont vides (les trois variables sont des variables de type). Les classes d'inéquations productives sont donc sans contraintes.

* Nous appliquons l'opération d'union + aux classes d'inéquations productives (qui sont en fait les deux systèmes) et nous prenons les solutions maximales, nous obtenons:

$$L = [] + T_{12}$$
$$M = any_2 + T_{22}$$
$$N = any_2 + T_{32}$$

* Nous remplaçons les variables de type par leurs solutions maximales dans les équations de type. Chaque variable de type n'a qu'une seule solution maximale:

$T_{11} = []$ qui est résolue.
$T_{21} = any_0$ qui est résolue.
$T_{31} = any_0$ qui est résolue.
$T_{12} = [any_1 | [] + T_{12}]$ à résoudre.
$T_{22} = any_2 + T_{22}$ à résoudre.
$T_{32} = [any_1 | any_2 + T_{32}]$ à résoudre.

* Après résolution des équations de type non résolues, nous trouvons:

$T_{12} = r\ !\ ([\ any_1 | [] + r\ ])$
$T_{22} = r\ !\ (\ any_2 + r\ )$
$T_{32} = r\ !\ ([\ any_1 | any_2 + r\ ]) = [any_1 | any_2]$

* Nous n'avons pas de contraintes correspondant aux classes d'inéquations contraintes (car ces dernières sont vides).

* Nous ne calculons pas les intersections des interprétations des composantes de type puisque chaque composante n'a qu'une seule équation.

Exemple (2). inférer le type du prédicat rev donné par le programme suivant (utilisant le prédicat append):

```
rev([], []).
rev([A|B], C) :- rev(B, D), append(D, [A], C).
```

* Les équations de type sont données par:

$T_{11} = []$
$T_{21} = []$
$T_{12} = [A|B]$
$T_{22} = C$

où A, B, C sont des variables de type.

* Chaque but du corps de la seconde clause a deux définitions. Nous obtenons 2 systèmes d'inéquations pour chaque but qui sont (en appliquant le traitement spécial des appels récursifs pour le but rev(B, D)):

Pour rev $(B \leq [] \ \& \ D \leq [])$ ou $(B \leq T_{12} \ \& \ D \leq T_{22})$

Pour append $(D \leq [] \ \& \ [A] \leq any_0 \ \& \ C \leq any_0)$ ou
$(D \leq r\,!\,([\,any_1 \mid [] + r\,]) \ \& \ [A] \leq r\,!\,(any_2+r) \ \& \ C \leq r\,!\,([\,any_1 \mid any_2 + r\,])$

* Nous normalisons ces systèmes d'inéquations et nous obtenons:

$(B \leq [] \ \& \ D \leq [])$ ou $(B \leq T_{12} \ \& \ D \leq T_{22})$

$(D \leq [] \ \& \ A \leq any_3 \ \& \ [] \leq [] \ \& \ C \leq [\,any_3\,])$ ou
$(D \leq r\,!\,([\,any_1 \mid [] + r\,]) \ \& \ A \leq r\,!\,(any_4+r) \ \& \ [] \leq [] \ \& \ C \leq r\,!\,([\,any_1 \mid [any_4] + r])$

Nous avons propagé la nouvelle forme $[any_3]$ (respectivement $[any_4]$) de any_1 (respectivement de any_2).

* L'éclatement ou et & (en renommant les any_i) donne:

(1) $(B\leq[] \ \& \ D\leq[] \ \ \& \ \ D\leq[] \ \& \ A\leq any_3 \ \& \ [] \leq [] \ \& \ C\leq[any_3])$

(2) $(B\leq[] \ \& \ D\leq[] \ \ \& \ \ D\leq r!([any_1|[]+r]) \& A\leq r!(any_4+r) \ \& \ []\leq[] \ \& \ C\leq r!([any_1|[any_4]+r]))$

(3) $(B\leq T_{12} \ \& \ D\leq T_{22} \ \ \& \ \ D\leq[] \ \& \ A\leq any_5 \ \& \ []\leq[] \ \& \ C\leq[any_5])$

(4) $(B\leq T_{12} \ \& \ D\leq T_{22} \ \& \ D\leq r!([any_7|[]+r]) \ \& \ A\leq r!(any_6+r) \ \& \ []\leq[] \ \& \ C\leq r!([any_7|[any_6]+r]))$

* En appliquant le filtrage une seconde fois, nous remarquons que le syet'me (2) est impossible ($I([]) \cap I(r!([any_1|[]+r])) = \varnothing$). La variables D apparait dans deux buts. Il ne reste que les systèmes 1, 3 et 4.

* Nous séparons les inéquations en classe d'inéquations productives et classe d'inéquations contraintes, (B, A, et C sont les variables de type).

Inéquations productives	Inéquations contraintes	
$B \leq [] \ \& \ A \leq any_3 \ \& \ C \leq [any_3]$	$D \leq [] \ \& \ D \leq [] \ \& \ [] \leq []$	(1)
$B \leq T_{12} \ \& \ A \leq any_5 \ \& \ C \leq [any_5]$	$D \leq T_{22} \ \& \ D \leq [] \ \& \ [] \leq []$	(3)
$B\leq T_{12} \ \& \ A\leq r!(any_6+r) \ \& \ C\leq r![any_7\|[any_6]+r]))$	$D\leq T_{22} \ \& \ D\leq r!([any_7\|[]+r] \ \& \ []\leq[]$	(4)

La classe d'inéquations contraintes du premier système est possible. La première classe d'inéquations productives est donc solution sans contrainte.

Dans la classe d'inéquations contraintes du système (3) l'inéquation $[] < []$ est possible. Nous avons aussi les inéquations $D \leq [] \ \& \ D \leq T_{22}$ dont l'intersection des interprétations des membres droits ne peut pas être

calculée actuellement. Ces deux inéquations forment une contrainte C_1. La classe d'inéquations productives correspondante devient alors:

$$B \leq T_{12} \backslash C_1 \;\&\; A \leq any_5 \backslash C_1 \;\&\; C \leq \lfloor any_5 \rfloor \backslash C_1.$$

Dans la classe d'inéquations contraintes du système (4), la 3ème inéquation $[] \leq []$ est possible. L'intersection des interprétations des membres droits des deux inéquations restantes ne peut pas être calculée actuellement. Elles forment une contrainte C_2 sur la classe d'inéquations productives. La classe d'inéquations productives devient donc:

$$B \leq T_{12} \backslash C_2 \;\&\; A \leq r!(any_6 + r) \backslash C_2 \;\&\; C \leq r!(\lfloor any_7 \mid \lfloor any_6 \rfloor + r \rfloor) \backslash C_2.$$

* Nous appliquons l'opération d'union "+" aux 3 classes d'inéquations productives, nous prenons les solutions maximales et nous obtenons:

$$\begin{array}{ll} & B = [] + T_{12} \backslash C_1 + T_{12} \backslash C_2 \\ (1) + (2) + (3) & A = any_3 + any_5 \backslash C_1 + r!(any_6 + r) \backslash C_2 \\ & C = \lfloor any_3 \rfloor + \lfloor any_5 \rfloor \backslash C_1 + r!(\lfloor any_7 \mid \lfloor any_6 \rfloor + r \rfloor) \backslash C_2 \end{array}$$

* Nous remplaçons les variables de type A, B et C par leurs solutions maximales dans les équations de type. Dans ces solutions toutes les variables n'ont qu'une solution chacune:

$$T_{11} = []$$
$$T_{12} = [A|B] = [\, any_3 + any_5 \backslash C_1 + r!(any_6 + r) \backslash C_2 \mid [] + T_{12} \backslash C_1 + T_{12} \backslash C_2 \,]$$
$$T_{21} = []$$
$$T_{22} = C = \lfloor any_3 \rfloor + \lfloor any_5 \rfloor \backslash C_1 + r!(\lfloor any_7 \mid \lfloor any_6 \rfloor + r \rfloor) \backslash C_2$$

* Nous résolvons ces équations de type et nous obtenons:

$$T_{11} = [].$$
$$T_{12} = r\,!\,([\, any_3 + any_5 \backslash C_1 + r!(any_6 + r) \backslash C_2 \mid [] + r \backslash C_1 + r \backslash C_2 \,]$$
$$T_{21} = []$$
$$T_{22} = \lfloor any_3 \rfloor + \lfloor any_5 \rfloor \backslash C_1 + r!(\lfloor any_7 \mid \lfloor any_6 \rfloor + r \rfloor) \backslash C_2$$

* Nous vérifions les deux contraintes C_1 et C_2 introduites:
$C_1 = (D \leq T_{22} \;\&\; D \leq [])$ et $C_2 = (D \leq T_{22} \;\&\; D \leq r!([any_7 \mid [] + r]))$. En vérifiant la première contrainte, nous trouvons qu'elle est fausse (intersection vide des inteprétations des membres droits des deux inéquations) et donc les éléments de type sous cette contrainte sont à éliminer. En vérifiant la contrainte C_2, nous trouvons que l'intersection des deux interprétations des

membres droits des inéquations n'est pas vide et donc les éléments de type sous cette contrainte sont solution. Nous obtenons donc (et après simplification):

$$T_{11} = []$$
$$T_{12} = r\ !\ [\ any_3 + r!(any_6+r)\ |\ []\ +\ r\ ] = r\ !\ [any_8\ |\ []\ +r]$$
$$T_{21} = []$$
$$T_{22} = [any_3] + r\ !\ ([any_7\ |\ [any_6] + r) = r\ !\ [any_9\ |\ []\ +\ r]$$

2ème cas: double appel récursif.

Cette catégorie d'appel récursif apparaît dans une clause de la manière suivante p:-...p,....p,... Dans le cas où nous avons une récursion plate double du même prédicat, nous traitons chaque appel récursif de la même manière que le cas précédent. Nous utilisons ensuite la notion d'éclatement de ou et de & introduite précédemment. Le reste du traitement se fait comme précédemment.

3.5.2. Récursion profonde

Nous passons maintenant au cas plus compliqué où la récursion est indirecte. Soit une clause A_0 :- A_1,....,A_n. Nous produisons les équations de type de la même manière que précédemment. Nous déterminons les domaines des variables de type de cette clause en examinant son corps.

Nous choisissons un but A_i du corps (nous prenons i=1). Nous produisons tous les systèmes d'inéquations comme dans les cas précédents en traitant de manière particulière les appels récursifs. Le but A_i appelle une de ses définitions B_j:-B_1,...B_m (nous prenons la première clause unifiable avec A_i). On peut avoir les cas suivants:

1)Si la clause B_j:-... est un fait (m=0) ou si son type est déjà calculé alors la génération des inéquations se fait comme dans les cas précédents.

2) L'appel A_i forme un appel récursif et dans ce cas nous appliquons le traitement spécial des appels récursifs (donné précédemment).

3) La clause B_j:-B_1,....,B_m est avec corps, le type de B_0 n'est pas calculé et l'appel ne forme pas un appel récursif. Dans ce cas nous produisons les inéquations correspondant à l'unification du but A_i avec la tête B_j de la clause. Nous aurons des inéquations de la forme suivante $t_i \leq t'_i$. Ce sont des *inéquations intermédiaires*. t'_i contient de nouvelles variables de type correspondant aux variables de type de la clause B_j:-B_1,....,B_m. Nous mémorisons le prédicat B_j (pour détecter les appels récursifs) et nous produisons les équations de type de cette clause. Il faut maintenant déterminer les types de ces nouvelles variables de type. Pour le faire nous traitons le corps de la clause B_j:-B_1,....,B_m de la même manière que le corps de la clause A_0:-A_1, ... A_n, en supposant que les buts B_i s'unifient avec des clauses C_j:-C_1,....,C_k.

Nous effectuons cet enchaînement d'appels (production d'inéquations intermédiaires) jusqu'à aboutir à l'unification d'un but avec un fait, à un appel récursif ou à l'appel d'un prédicat dont le type est connu. Nous obtenons des systèmes d'inéquations dont une inéquation a la forme suivante $t_i \leq t'_i$ où:

t'_i = atome / any_j / t'_1 + ... + t'_n / r ! (...) / T_{kj} / T'_k / $f(t'_1, ... t'_n)$ et
t_i = atome / variable de type / $f(t_1,....,t_m)$

Nous normalisons ce système comme aux cas précédents et nous obtenons des inéquations de la forme $t_i \leq t'_i$ avec:

t'_i = atome / any_j / t'_1 + ... + t'_n / r ! (...) / T_{kj} / T'_r / $f(t'_1,....,t'_m)$ et
t_i = atome / variable de type.

En ayant traité l'appel de B_i de sa clause C_1, nous passons à une autre définition C_2 avec laquelle le but B_i peut s'unifier. Nous lui effectuons le même traitement que précédemment.

Nous répétons ce même traitement jusqu'à épuisement de toutes les définitions pouvant s'unifier avec un B_i. Nous obtenons une disjonction de systèmes d'inéquations:

$ss_1 = s_1$ ou s_2 ou ... ou s_n avec
$s_i = t_1 \leq t'_1$ & ... & $t_m \leq t'_m$ i=1,n avec
t_i = atome / variable i=1,m
t'_i = atome / any_j / t'_1 + ... + t'_n / r ! (...) / T_{kl} / T'_r / $f(t'_1,....t'_m)$

Nous passons ensuite au second but B_2 et nous lui effectuons le même traitement qu'à B_1. Nous obtenons une autre disjonction ss_2 de systèmes d'inéquations. Nous appliquons ce même traitement à tous les buts B_i en obtenant à chaque fois une expression ss_i.

Pour le corps d'une clause B_j:-B_1,....,B_n nous avons obtenu l'expression de systèmes d'inéquations:

$$ss_1 \ \& \ ss_2 \ \& \ ... \ \& \ ss_n$$

Nous éclatons les opérateurs ou et & comme aux cas précédents en renommant les any_i des membres droits des inéquations à chaque passage d'un système s_i à un autre (ce renommage correspond au renommage des variables à chaque passage d'une clause à une autre). Nous obtenons des systèmes d'inéquations ayant les mêmes membres gauches qui sont les termes syntaxiques du corps de la clause. Nous effectuons le second filtrage comme précédemment.

Nous séparons les inéquations de chaque système obtenu en classes d'inéquations productives et classes d'inéquations contraintes. Nous vérifions les inéquations qui sont possibles de chaque classe d'inéquations contraintes en éliminant toujours les systèmes impossibles. Les inéquations restantes forment une contrainte C_k sur la classe d'inéquations productives. Une inéquation productive $t_i \leq t'_i$ devient alors $t_i \leq t'_i \backslash C_k$.

Nous appliquons l'opération d'union + aux classes d'inéquations productives en explorant combinatoirement les solutions des variables de type. Si la j^{eme} inéquation est:

$$X \leq t_1 + \ldots + t_{i-1} + d_1 \,\&\, d_2 \,\&\, \ldots \,\&\, d_m + t_{i+1} + \ldots + t_n$$

alors nous produisons m inéquations pour cette variable X données par:

$$X \leq t_1 + \ldots + t_{i-1} + d_j + t_{i+1} + \ldots + t_n \quad j=1,m$$

qui sont équivalentes à:

$$\&_j (X \leq t_1 + \ldots + d_j + \ldots + t_n) \equiv X \leq \&_j(t_1 + \ldots + d_j + \ldots + t_n).$$

Nous prenons les solutions maximales des résultats de l'application de l'opération d'union + et nous obtenons les domaines des variables de type de la clause B_j:-B_1,....,B_m. Une variable de type X peut avoir éventuellement n solutions maximales $X = t_i$ $i=1,n$ (qui seront utilisées combinatoirement).

Les types des nouvelles variables de type (de l'unification de A_i avec B_0) sont maintenant connus (éventuellement sous contraintes). Nous pouvons remplacer les variables de type de t'_i des inéquations intermédiaires $t_i \leq t'_i$ (les variables de type des équations de type $T_{ij} = t'_i$ respectivement) par leurs solutions. Si $t'_i(X)$ contient une variable de type X ayant n solutions maximales d_j pour $j=1,n$ alors nous produisons combinatoirement l'inéquation $t_i \leq t'_i(d_1)$ & ... & $t'_i(d_n)$ qui est équivalente à la conjonction $t_i \leq t'_1(d_1)$ & ... & $t_i \leq t'_n(d_n)$ ($T_{ij} = t'_i(d_1)$, ..., $T_{ij} = t'_i(d_n)$ respectivement).

Nous effectuons le même traitement que précédemment aux autres buts A_i. Une fois que les inéquations de chaque unification du but A_i ne contiennent plus de variables de type dans leurs membres droits, nous leur appliquons le même traitement de séparation des inéquations en deux classes, vérification des classes d'inéquations contraintes, etc, en explorant les résultats combinatoirement.

Nous déterminons ainsi les domaines des variables de type de la première clause A_0:-A_1,.....A_m.

Nous remplaçons les variables de type dans les équations de type comme aux cas précédents (combinatoirement). Une équation de type peut avoir éventuellement des équations multiples.

Nous résolvons les équations de type comme dans les cas précédents.

Nous vérifions les contraintes C_i introduites lors de toutes les séparations des inéquations en les deux classes d'inéquations productives et d'inéquations contraintes comme dans les cas précédents.

Nous calculons les intersections éventuelles des interprétations des membres droits des équations de type.

Bilan: Pour calculer le type de tels prédicats, nous avons généré des *inéquations intermédiaires* entre la clause principale et les faits, les prédicats de type connu ou les appels récursifs; nous avons utilisé combinatoirement les types des nouvelles variables de type.

Soit une clause A_0:-A_1,.....A_n et une inéquation intermédiaire $t_i \leq \psi(X)$ contenant une variable de type X dans son membre droit. Cette inéquation est produite lors d'un appel d'un but A_i de l'une de ses définitions B_j:-B_1,...,B_m où $m \neq 0$ et l'appel ne forme pas un appel récursif. Supposons que la variable X ait k solutions maximales $X=d_j$ j=1,k obtenues en traitant le corps de cette dernière clause. En remplaçant la variable de type X dans l'inéquation intermediaire $t_i \leq \psi(X)$ précédente, nous produisons $t_i \leq \psi(d_1)$ & ... & $\psi(d_k)$ (qui est équivalente à $t_i \leq \psi(d_1)$ & ... & $t_i \leq \psi(d_k)$). Lors de la prise de la solution maximale de l'expression:

$$Y_i \leq t'_1 + ... + Y \leq \psi(d_1) \& ... \& \psi(d_k) + ... + Y \leq t'_n,$$

nous produisons k solutions maximales pour la variable Y, données par:

$$Y = t'_1 + ... + t_{i-1} + \psi(d_j) + t'_{i+1} + ... + t'_n \text{ pour } j=1,k$$

Cette exploration combinatoire est possible grâce à la propriété de l'intersection donnée précédemment.

4. Bilan de la méthode

Dans ce paragraphe nous allons justifier de manière informelle que les résultats inférés par notre système sont bien fondés, dans la mesure où l'interprétation du type inféré est un sur-ensemble de l'ensemble de succès. Nous justifierons cette propriété en 2 étapes.

- Dans la première étape nous notons que l'impossibilité d'une expression d'inéquations d'une possibilité d'appels correspond à une impossibilité de l'unification de cette suite d'appels. Ce qui revient à dire que tous les filtrages effectués le long de l'inférence correspondent à des échecs de l'unification. Alors nous pouvons affirmer que la possibilité d'un système d'inéquations est un "faible" test de l'unification.

- Nous notons ensuite que l'ensemble de solutions est inclus dans l'ensemble interprétation du type inféré. Nous prenons un n-uplet t de termes constants n'appartenant pas à l'ensemble de type d'un prédicat p et nous vérifions que l'appel $p(t)$ échoue.

4.1. Situations d'échec

Nous allons reprendre les 3 niveaux de filtrage et nous vérifions qu'ils correspondent à des impossibilités d'unification.

4.1.1. Au niveau du 1er filtrage.

Nous allons examiner les 7 situations:

* Une expression de systèmes d'inéquations contenant une inéquation de la forme $a \leq f(b_1, \ldots, b_n)$ est impossible. Cette inéquation correspond à l'unification de a avec $f(b_1, \ldots, b_n)$ qui est impossible.

* Une expression contenant une inéquation de la forme $a \leq b$ avec $a \neq b$ est impossible. Cette inéquation correspond à l'unification de a avec b qui est impossible.

* Le cas suivant d'impossibilité est celui où nous avons une inéquation de la forme $t_0 \leq t_1 + ... + t_n$ (ou r ! (...)) avec a $\not\leq t_i$ pour i=1,n. Le membre droit de cette inéquation est obtenu lors de l'application de l'opération d'union + aux classes d'inéquations productives des systèmes du corps d'une clause. Chaque t_i i=1,n correspond à une suite d'appel du corps d'une clause qui est appelée lors de la génération de cette inéquation. Une inéquation $t_i \leq t'_i$ est impossible signifie que t_i ne peut pas s'unifier avec la suite d'appel ayant produit l'inéquation $t_i \leq t'_i$. Si toutes les n suites d'appel sont impossibles alors l'unification avec cette clause est impossible.

* Dans le 4ème cas, supposons que dans un corps d'une clause, une variable apparaît en plusieurs positions avec $I(t_1) \cap ... \cap I(t_n) = \emptyset$. Supposons que l'unification de cette variable dans cette clause lors d'un appel avec succès réussisse. L'unification réussie avec un terme t implique que la variable X prend la même valeur t dans toutes ses autres positions (propagation de la substitution <X/t> dans ses autres positions). Nous avons donc $X \leq t$ dans toutes ces positions de X. Par conséquent l'intersection des interprétations des membres droits des inéquations $X \leq t$ en ces différentes positions contient au moins t et donc elle n'est pas vide; contradiction avec l'hypothèse $\cap I(t_i) = \emptyset$ i=1,n et donc l'unification échoue.

* Une inéquation de la forme $f(b_1, ..., b_n) \leq a$ correspond à l'unification de $f(b_1, ..., b_n)$ avec "a", qui est impossible.

* Dans le 6ème cas, nous avons l'inéquation $f(a_1, ..., a_n) \leq g(b_1, ..., b_m)$. Si $f \neq g$ ou $n \neq m$ alors l'unification des 2 termes $f(a_1,...,a_n)$ et $g(b_1,...,b_m)$ ne peut pas réussir. Soit le cas où f=g et n=m avec une inéquation $a_i \not\leq b_i$. Cela signifie que l'unification des 2 arguments a_i et b_i est impossible ce qui implique que l'unification des 2 termes $f(a_1,...a_n)$ et $g(b_1,...b_n)$ ne peut pas réussir.

* Si un but appelle un prédicat non-défini, l'unification échoue.

Bilan 1: Au niveau du 1er filtrage, toutes les situations d'impossibilités de systèmes d'inéquations correspondent à des situations d'échec de l'unification.

4.1.2. Au niveau de la normalisation

Un autre filtrage est effectué au niveau de la normalisation. Les cas d'impossibilité dans cette normalisation (non détectés par le premier filtrage) sont obtenus lorsque nous avons des inéquations de la forme :

- $a \leq b$ où le membre droit de l'inéquation est obtenu lors de la propagation de la nouvelle forme d'un élément de type any_i.
- $f(a_1, ..., a_n) \leq g(b_1, ..., b_m)$ avec $f \neq g$, $n \neq m$ ou $a_i \not\leq b_i$. Le membre droit de l'inéquation est obtenu après une propagation d'une nouvelle forme d'un élément de type any_i.

Tous ces cas correspondent en fait à des impossibilités d'unification.

* Soit une inéquation $a \leq any_i$ qui est remplacée par $a \leq a$ lors de la normalisation. Nous effectuons la propagation any_i/a, dans ce système. Cette inéquation correspond à l'unification de "a" avec une variable libre V ayant le type any_i. La substitution V/a est propagée aux autres positions de la variable dans la tête de la clause avec laquelle nous unifions. Soit une autre inéquation dans le même système ayant la forme $b \leq any_i$. Elle correspond à la simulation de l'unification de b avec V. En unifiant V avec "a", la substitution V/a est propagée aux autres positions et donc l'unification de V avec une autre constante $b \neq a$ ne peut pas réussir.

* L'autre situation est le cas où nous avons propagé une nouvelle forme d'un any_i (correspondant au type d'une variable libre V) dans une inéquation $f(t_1,...,t_n) \leq any_i$. Ceci correspond à l'unification de $f(t_1,...,t_n)$ avec V, par conséquent V est remplacée dans ses autres positions par $f(t_1,...,t_n)$. Soit une autre inéquation $g(t_1,...,t_n) \leq any_i$. Cette dernière inéquation correspond à l'unification de $g(t_1,...,t_n)$ avec V où cette variable a la nouvelle forme $f(t_1,...,t_n)$, et donc essayer d'unifier $f(t_1,..., t_n)$ avec $g(t'_1,...,t'_m)$; si $n \neq m$ ou $f \neq g$ alors l'unification échoue et donc l'unification est impossible. Si $f=g$ et $n=m$, mais nous avons $t_i \not\leq t'_i$, qui correspond à une impossibilité d'unification des $i^{èmes}$ arguments, l'unification de $f(t_1, ..., t_n)$ et $g(t'_1, ..., t'_n)$ est impossible.

Bilan 2: Les deux situations d'échec du second filtrage (de la normalisation) sont des situations d'échec de l'unification.

4.1.3. Au niveau de la vérification des contraintes

Après normalisation des systèmes d'inéquations pour une configuration d'appel, nous avons séparé les inéquations en classe d'inéquations productives et classe d'inéquations contraintes. La seconde classe forme une contrainte C_i sur la possibilité ou non de la classe d'inéquations productives. La classe d'inéquations contraintes peut contenir deux catégories d'inéquations:

- Dans la première, les inéquations sont de la forme $a \leq \psi(T_{ij}, T'_i)$ avec "a" un atome et $\psi(T_{ij}, T'_i)$ un élément de type contenant éventuellement des composantes de type T_{ij} et/ou des variables d'hypothèse T'_k.

- Dans la seconde catégorie, les inéquations sont de la forme $X \leq t_1$ & ... & $X \leq t_n$ où les t_i contiennent des composantes de type T_{ij} et/ou des variables d'hypothèses T'_k.

Une contrainte C_i est fausse si au moins l'une de ces deux catégories est impossible.

* Dans la 1ère catégorie d'inéquation $a \leq \psi(T_{ij}, T'_k)$, nous vérifions les contraintes C_j du membre droit de l'inéquation, nous obtenons une expression S sans contraintes et nous vérifions si l'atome "a" appartient à cette expression. Si a n'appartient pas à I(S) alors 2 cas peuvent se présenter:
  - Soit $a \backslash C_i \in I(\psi(T_{ij}, T'_k))$ mais C_i est fausse alors le système ayant cette contrainte n'est pas solution et donc "a" n'appartient pas à $I(\psi(T_{ij}, T'_k))$.
  - Soit "a" n'apparaît pas dans $\psi(T_{ij}, T'_k)$ et dans ce cas ce système ne peut pas être solution.

* L'autre catégorie d'inéquations de la classe d'inéquations contraintes est de la forme $X \leq t_1$ & ... & $X \leq t_n$. Pour vérifier cette partie de la contrainte, nous vérifions éventuellement les contraintes des t_i, nous obtenons les inéquations $X \leq t'_1$ & ... & $X \leq t'_n$ où t'_i correspond à t_i après la vérification de ses contraintes. Si l'intersection des interprétations de t'_i est vide alors il n'existe aucune substitution possible pour la variable X dans cette suite d'appel, et donc échec de l'unification.

Bilan 3: Lors de la vérification des contraintes, si une classe d'inéquations contrainte est fausse alors le système correspond à une situation d'échec.

4.1.4. Bilan

Dans les trois filtrages, un système d'inéquation impossible correspond à une unification impossible. Il est équivalent de dire que pour une unification possible alors l'expression de système d'inéquations est possible.

4.2. Les types inférés sont corrects

Nous allons vérifier maintenant que les types inférés sont corrects, c'est-à-dire que $T(p) \geq D(p)$. Nous affirmons que toutes les solutions d'un prédicat appartiennent à son ensemble de type. Admettons en effet l'existence d'un n-uplet $x = \langle x_1, \ldots, x_n \rangle$ n'appartenant pas à l'ensemble $T(p)$ et montrons que $p(x)$ échoue. Soit un prédicat défini par: $p_i(t)$:-... avec $i=1,m$ et $t = \langle t_1, \ldots, t_n \rangle$. Trois cas peuvent se présenter:

1) Si $p(x)$ ne s'unifie avec aucun $p_i(t)$ alors échec de l'appel de p et alors x n'appartient pas à $D(p)$.

2) Nous avons $x \not\in t$ pour $i=1,n$ alors échec d'après le bilan précédent.

3) Soit un $p(t)$:-A_1,....,A_n unifiable avec $p(x)$ par θ, alors nous obtenons la résolvante $(A_1, \ldots A_n)\theta$. Il faut montrer qu'au moins un $A_i\theta$ ne réussit pas. Nous le montrons en 3 cas:

a) Supposons qu'il n'existe pas de variable partagées entre t et les A_j pour $j=1,n$ (pas de variables de type).

a1) Si t_i contient des variables, alors leur type est any_j. L'unification de x avec t est possible par θ signifie que la simulation de cette unification est donnée par $x \leq t$, où les variables des membres droits sont remplacées par des any_j. Comme $T(p_i) = t$ où les variables sont remplacées par des any_j alors $x \leq T(p)$. Contradiction avec $x \not\leq T(p)$ et donc l'unification est impossible.

a2) Cas où t ne contient pas de variable et l'unification de x avec t est possible par θ. Cette unification est simulée par $x \leq t$. Comme $T(p) = t$ (t est sans variables) alors $x \leq T(p)$, contradiction avec $x \not\leq T(p)$ et donc l'unification est impossible.

Dans ce 1er cas, nous supposons que dans le corps de la clause, il y a au moins une combinaison d'appel possible pour laquelle la clause réussit. Si toutes ces combinaisons sont impossibles alors tous les systèmes sont impossibles et donc la clause est une clause d'échec. Les éléments type donné par sa tête ne figurent pas dans le type de la clause.

b) Supposons qu'il existe des variables de type partagées entre t et les A_i. Soient $\{A_{j_1} ..... A_{j_k}\}$ l'ensemble des A_i contenant des variables de type. Soient $\{t_{j_1}, ..., t_{j_k}\}$ l'ensemble des termes appartenant à t et contenant des variables de type.

Supposons que pour chaque terme où apparaissent des variables de type, toutes les inéquations $t_{j_k} \leq A_{j_k}$ sont vraies. Comme nous avons pris les solutions maximales des inéquations alors nous obtenons $t_{j_k} = A_{j_k}$. Le terme $x = \langle x_1, ..., x_n \rangle$ est unifiable avec $t = \langle t_1, ..., t_n \rangle$ par θ. L'unification est possible signifie que les inéquations $x_i \leq t_i$ sont possibles. Or, toutes les autres variables n'apparaissant que dans la tête de la clause ont le type any_i et t est formé à partir de $\{t_{j_1}, ..., t_{j_k}\}$, nous obtenons donc $x_j \leq t_{j_k}$. Mais comme $t_{j_k} = A_{j_k}$ alors $x_i \leq A_{j_k}$. Ceci implique que $x \leq A_0$, alors $x \leq T(p)$ (puisque toutes les inéquations $t_{j_k} \leq A_{j_k}$ sont vraies), contradiction avec $x \not\leq T(p)$ et par conséquent il existe au moins une inéquation $t_{j_k} \leq A_{j_k}$ qui n'est pas possible.

4.3. Bilan général

Le type inféré d'un prédicat est correct. L'ensemble de type contient l'ensemble de solutions. Nous pouvons affirmer alors que pour tout terme constant n'appartenant pas à T_p alors p(t) échoue (le type est un sur-ensemble de l'ensemble de succès).

Le but de ce paragraphe n'est pas une prétention à une preuve rigoureuse mais seulement une augmentation tentant à persuader de la validité du travail.

5. Travail réalisé

Nous avons réalisé en Prolog (C-Prolog *[Pereira84]*) le système d'inférence décrit dans ce chapitre. Une description des algorithmes est donnée dans l'annexe 1. Des exemples d'inférence traités par ce système sont données dans l'annexe 2. En l'état actuel du système, un espace mémoire et un temps d'exécution "énormes" sont nécessaires. Ces deux paramètres varient de façon exponentielle avec le nombre de définitions des prédicats et avec le nombre de buts dans les corps des clauses.

Ces lourdeurs sont liées d'une part au fait que nous effectuons une inférence à partir de rien, et que nous sommes donc obligés de créer tous les éléments de type. D'autre part, cette création nous oblige à effectuer des renommages des symboles de récursion et des any_i au niveau des manipulations des éléments de type récursifs. D'un autre côté, puisque nous produisons toutes les combinaisons d'appels possibles, l'espace d'inférence est énorme, mais ceci au gain d'une meilleure précision. Nous pensons qu'une déclaration des constructeurs de type à la manière de Kanamori peut améliorer le temps de réponse mais ceci au prix d'exiger de déclarer les constructeurs de type. Nous pensons aussi, que plus de déclarations explicites (à la manière de l'approche déclarative), donnant les types des prédicats souvent utilisés, peut améliorer le temps de réponse et surtout prendre en compte les pensées du programmeur.

Nous présenterons dans le prochain chapitre quelques améliorations envisageables permettant d'améliorer la précision et le temps de réponse du système, en utilisant les notions de PROLOG (unification et résolution). A l'état actuel et après les tests que nous avons effectués, nous avons constaté que le système est mieux adapté (donne des résultats plus satisfaisants) pour les prédicats non-polymorqhes que ceux polymorphes, et dans cette catégorie des prédicats non-polymorphes le système est mieux adapté aux prédicats non-récursifs que ceux récursifs. Des améliorations citées dans le prochain chapitre permettent de reduire ces différences.

CHAPITRE 4

Bilan et Perspectives

Nous présentons dans ce chapitre une comparaison de notre système avec un système de la même famille (sans déclarations), celui de Mishra *[Mishra84]*. Nous exposons ensuite quelques améliorations envisageables pour augmenter l'efficacité du système d'inférence. Nous terminons par une description de quelques applications possibles des types inférés par le système dans un but d'optimisation et/ou de vérification.

1. Comparaison.

Nous avons présenté au chapitre 2 le système d'inférence de Mishra ainsi que des résumés des autres systèmes existants. Nous allons comparer les types inférés par notre système avec ceux de Mishra.

Notre système est plus précis et plus général que celui de Mishra dans la mesure où notre système est applicable aux prédicats polymores et non-polymorphes. Le système de Mishra, n'est applicable qu'aux prédicats non-polymorphiques. Même pour les prédicats non-polymorphiques notre système est plus précis que celui de Mishra dans la mesure où l'ensemble T(p) inféré par notre système est plus restreint que celui inféré par Mishra, et dans le pire des cas égal.

Exemple:

```
pere(a, b).
pere(b, c).
pere(c, d).

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

Par Mishra: $I(T_{pere}) = \{(a, b), (a, c), (a, d), (b, b), (b, c), (b, d), (c, b), (c, c), (c, d)\}$

$I(T_{g\_pere}) = I(T_{pere})$

Par notre système: $I(T_{pere}) = \{ (a, b), (b, c), (c, d)\}$

$I(T_{g\_pere}) = \{(a, c), (a, d), (b, c), (b, d)\}$

Cette différence de précision est dûe au fait que notre système élimine plus de systèmes d'échec que celui de Mishra.

2. Variantes et améliorations.

Une variante dans la définition des éléments de type est la définition des types primitifs tel que *entier* pour désigner le type constante entière par exemple. En pratique plusieurs choix de type primitifs ont été proposés *[Turbo86]*. Ce choix revient à l'utilisateur suivant les applications qu'il envisage de faire.

Une amélioration peut être faite en explorant à fond la décomposition dans son algorithme lorsqu'elle celle-ci est récursive. Dans cet algorithme, dès que nous traitions un couple (T'_k, T_{ij}) déjà empilé (succès du test de circularité) nous arrêtions l'itération en donnant la solution maximale any_l à la variable d'hypothèse T'_k. Cette amélioration permet de donner l'élément $\varnothing$ à certains appels qui bouclent.

Une autre amélioration peut être apportée en utilisant la notion de dépliage à un niveau pour détecter les branches d'échec plus rapidement. Cependant si les expressions de type contiennent des éléments de type récursifs d'un type déjà calculé de la forme $r\ !\ (t_1 + ... + f(r) + ... + t_n)$, il n'est pas possible de l'appliquer.

Une autre amélioration peut être obtenue en simplifiant au maximum les expressions de type résultats. Dans la version actuelle du système, une expression de la forme $r1\ !\ (0 + s(r1)) + r2\ !\ (s(0 + r2))$ par exemple n'est pas simplifiée à la forme $r\ !\ (0 + s(r))$.

Une autre voie reste à explorer pour garantir un même renommage des any_k lors des appels récursifs. Pour cela il faut indicer les r^i pour chaque appel récursif. En remplaçant un r^i par son expression, nous remplaçons tous les autres symboles r^j correspondant à cet appel récursif. Nous garantissons ainsi un même renommage des any_i au niveau de tous l'appel

récursif. Actuellement le seul cas où nous gardons le même nom d'un any_i lors des récursions est celui où il n'apparaît dans le membre droit d'une même inéquation que sous la forme r ! (... any_i ... + r + ...). Cette amélioration premet d'augmenter la précision du système.

Une autre amélioration pourait être effectuée pour passer de la simulation de l'unification à l'unification. Ceci nous permet de faire toutes les liaisons des variables lors de la production des inéquations. A l'état actuel du système et dans la conjonction $X \leq t_1$ & $X \leq t_2$, nous calculons l'intersection des interprétations de t_1 et de t_2. Il serait interessant de lier t_1 et t_2. Dans ce cas nous n'appliquons pas l'opération d'union + mais nous exploitons tous les systèmes combinatoirement (parcourir toutes les branches de l'arbre).

Une autre possibilité d'amélioration est de permettre au programmeur de donner des déclarations de type à la Mycroft-o'Keefe *[Mycroft82]* ou des déclarations des constructeurs de type à la Kanamori *[Kanamori84]*. Ces déclarations permettent d'améliorer les temps de réponse du système.

3. Quelques applications des types inférés

Le but de ce paragraphe est de présenter quelques applications des types inférés, en supposant que ceux-ci soient suffisamment précis pour permettre leurs utilisations. Bien-que les types inférés par notre système peuvent être utilisés pour ces applications, l'idéal serait d'avoir un système aussi précis que possible permettant d'utiliser ses résultats et en bénéficier de ces applications. A notre avis, prendre en compte les améliorations citées précédemment favorise cette idée.

Il existe actuellement divers axes de recherches pour les applications des types en PROLOG. Une application possible de l'inférence de type est la compilation. La connaissance des types peut être utilisée pour améliorer l'efficacité du programme produit, pour déterminer des situations dangereuses à la compilation ou la mise en oeuvre de règles de bonne programmation, etc.

Eviter les exécutions inutiles.

Actuellement PROLOG peut être considéré comme un langage à un seul type: *l'Univers de Herbrand*, ce qui fournit une grande liberté pour l'écriture de programmes. Cependant, pour l'écriture de logiciels importants, l'absence de vérifications statiques est un handicap certain. Une simple erreur typographique peut conduire à un programme qui s'exécute mais dont les résultats sont erronés. Ces erreurs sont actuellement très difficiles à détecter dans des programmes écrits en PROLOG pur. On n'a pas le moyen de distinguer une bonne réponse d'une réponse affirmative fausse.

D'un autre côté beaucoup de prédicats n'ont un sens qu'avec des types d'arguments bien précis; utilisés avec des valeurs appartenant à d'autres types, ils peuvent créer un état d'erreur dont la détection ne peut se faire que pendant l'exécution. Ce qui est plus grave, ils peuvent fournir des réponses affirmatives avec des résultats erronés sans que l'on puisse s'en rendre compte.

Ce type d'erreur est dû en partie au fait qu'un programme n'est accompagné d'aucune déclaration du type des variables, ni de systèmes automatiques de vérifications de compatibilité de type des paramètres d'appel et de définition d'une clause. La seule condition (vérification) faite est que l'unification réussisse.

L'idée d'introduire la notion de type en PROLOG répond en partie à ce problème de garantir une certaine cohérence dans la structure et le type des valeurs manipulées.

Règle: Soit le type d'un prédicat p n-aire donné par:

$$T_p = < T_{11} \vee \ldots \vee T_{1m}, \ldots, T_{n1} \vee \ldots \vee T_{nm} >$$

En posant la question $?p(t_1, \ldots t_n)$ de type $p(t'_1, \ldots t'_n)$, il faut qu'il existe une définition j de p telle que:

$$T_{ij} \cap t'_i \neq \varnothing \quad \text{pour } i=1,n$$

Ceci est dû au fait que l'interprétation S du type d'un argument est un sur-ensemble de son ensemble de succès S'. Si l'intersection de l'interprétation d'un t'_i avec S est vide, alors l'intersection entre cet ensemble interprétation de t'_i et tous sous-ensemble de S (entre autre S') est toujours vide.

Bon-typage.

Au cours de l'inférence de type, nous calculons les domaines des variables des corps des clauses. Nous vérifions implicitement que les intersections entre les domaines d'appel et les domaines de définition des arguments ne sont pas vides.

Définition: Un programme est bien-typé (sauf si l'échec est une solution souhaitée), si pour tout prédicat p il a au moins une définition j tels que tous les $T_{ij} \neq \emptyset$ i=1,n où T_{ij} sont les composantes de type de la $j^{ème}$ clause de p.

Ceci est dû aussi au fait que l'inteprétation du type d'un argument est un sur-ensemble de son ensemble de solutions. Si cette interprétation est vide alors l'ensemble de solutions est vide.

Détection d'erreurs

Nous avons dit qu'en PROLOG, les erreurs typographiques échappent à la détection. Comme une solution exacte ne peut pas être distinguée d'une solution fausse, il est très difficile de détecter certaines erreurs et de les comprendre. Une partie de ces erreurs peut être détectée en analysant les types inférés. En comparant le type inféré avec le modèle de solution voulue, nous pouvons constater que telle partie d'un programme n'a pas le sens espéré. Voir exemples du paragraphe "Pourquoi les types en Prolog" du chapitre 1.

Détection des branches en échec.

La stratégie de résolution de PROLOG consiste à explorer l'arbre et/ou en profondeur d'abord. Son parcours est exhaustif, il est nécessaire de pratiquer des retours en arrière pour parcourir des branches de l'arbre restant lors d'un choix. Ce retour en arrière est chronologique: on revient "*aveuglement*" toujours au point de choix le plus récent soit en cas d'échec dans une branche, soit lors de la recherche d'une autre solution.

Cependant lors du parcours de l'arbre, il y a des branches qui conduisent à des échecs. Il serait intéressant de les déterminer statiquement, ce qui est en partie possible au cours de l'inférence de type.

Lors du calcul du type, chaque système d'inéquations d'une combinaison d'appels possible (d'un prédicat tête d'une clause), correspond à une branche dans l'arbre de recherche au niveau du noeud du prédicat. Si un système a une inéquation $r \leq \varnothing$, la branche correspondante est une branche d'échec.

Dans certains cas, ceci permet une spécialisation du prédicat contenant la clause pour son contexte d'appel. Cette spécialisation consiste à fabriquer un prédicat spécifique dérivé par retrait du prédicat initial de la clause en échec.

Remontée des propriétés des arguments.

Nous pouvons voir l'optimisation d'un programme PROLOG comme une transformation de ce dernier en un autre programme PROLOG plus efficace. Une transformation est ce que nous appelons *la remontée des propriétés*. Connaissant la structure d'un argument dans un programme au niveau terminal d'une liaison de référence, nous pouvons faire remonter la structure de l'argument aux niveaux superieurs.

Exemple: Considérons le problème générique suivant:

t_1 est t_2, t_2 est t_3, ..., t_{n-1} est t_n, t_1 a une propriété t.

représenté en PROLOG par:

$t_n(X)$:- $t_{n-1}(X)$

$t_{n-1}(X)$:- $t_{n-2}(X)$

... ...

$t_2(X)$:- $t_1(X)$

t_1(propriete(t)).

Soit la question $t_n(X)$?. La liaison "X = propriété(t)" se fait après *n* unifications. On fait un parcours de t_n jusqu'a t_1 pour avoir X unifié à "propriété(t)".

Le calcul de type de ce programme affecte l'ensemble { propriété(t) } à tous les prédicats tête des clauses. Nous remplaçons ces variables par l'élément de type "propriété(t)", ce qui donne comme nouveau programme:

t_n(propriété(t)).

.....

t_1(propriété(t)).

Unification partielle

La procédure de base d'un interpréteur PROLOG est la procédure d'unification qui se ramène toujours à une résolution de systèmes d'équations. Le nombre de systèmes à résoudre au moment de l'exécution correspond au nombre d'unifications à réaliser. Une optimisation possible est de réaliser éventuellement une partie de ces unifications statiquement à la compilation. Ceci nous amène à minimiser le nombre d'appels à la procédure d'unification, et à résoudre une partie des équations à la compilation. C'est ce que nous appelons "l'unification partielle".

Une réalisation de ces unifications statiques est faite en utilisant la connaissance des types. En effet lors de l'inférence du type, nous calculons un sur-ensemble de l'ensemble de tous les termes éventuellement constants (si ce n'est toutes les valeurs possibles) que peut prendre une variable du programme lors de toutes ses exécutions réussies.

Si nous connaissons une partie de la structure d'un argument lors du calcul de son type ou nous déterminons sa valeur unique durant toutes les exécutions du programme, nous pouvons réaliser l'unification à la compilation de la partie de la structure connue.

Ceci nous ramène aussi à une dépolymorphisation d'un prédicat. Si dans un contexte d'appel, un prédicat polymorphique est appelé par un type non-polymorphique, nous pouvons le transformer en un prédicat non-polymorphique et donc passer de l'unification générale sur l'Univers de Herbrand à une unification sur le domaine du prédicat.

Règle: Si tous les éléments de type d'une variable X d'un prédicat ont une même structure sans récursion (par exemple $f(g(t_1), f(t_2)) = f(\{g(t_1, t_2\}))$, alors nous réalisons la partie commune de l'unification à la compilation (unification de f). Remplacer les occurences de la variable X par f(X'). Le type de la variable X' est $\{g(t_1), t_2\}$. . Une clause p(X):-...r(X)... peut être transformée en p_f(X'):-...r(A').

Eliminer les variables redondantes

Nous présentons maintenant le cas où nous pouvons supprimer les variables redondantes. Ce sont des variables qui prennent une valeur unique dans un programme lors de toutes les exécutions réussies. C'est un cas particulier du paragraphe précédent où le type d'une variable X est un singleton défini (pas de symbole any_i ni de récursion). Dans ce cas nous utilisons la règle suivante:

Règle: Si le type inféré d'une variable dans un programme est un singleton défini (ne contenant pas de symbole "any_i" ni de récursion), nous remplaçons cette variable par l'élément de type du singleton.

Ce remplacement est correct puisque l'ensemble interprétation du type est un sur-ensemble de l'ensemble solution.

Prédicats toujours vrais

En écrivant des programmes PROLOG, on exprime les relations dans leur cadre le plus général. En écrivant par exemple la relation de g_parent, on utilise des variables pour exprimer la relation de g_parent dans son cadre le plus général. Cette relation peut être restreinte à un contexte de programme.

Exemple: Soit le programme suivant:

```
pere(a, b) .
pere(b, c).
pere(b, g).
gpere(X, Y):-pere(X, Z),pere(Z, Y). buts_en _fonction (X, Z et Y)
```

Après calcul du type, les domaines affectés aux variables X, Z et Y sont respectivement { a } , { b } et { c, g }. Nous pouvons remplacer les occurences des variables X, Z et Y par leur valeurs possibles et nous obtenons alors:

```
pere(a,b).
pere(b,c).
pere(b,g).
gpere(a,c):-buts_en_fonction (a, b et c)
gpere(a,g):-buts_en_fonction (a, b et g)
```

Déterminisme

Une autre application des types peut être possible pour déterminer le déterminisme des prédicats *[Sawamura84]*, *[Mellish85]*, *[Harel85]*, *[Sawamura86]*. On n'utilise pas toujours la souplesse offerte par la programmation logique tel que le non-déterminisme. De larges segments des programmes PROLOG sont déterministes et directionnels et ne fournissent qu'une solution. PROLOG implémente le non-déterminisme par une stratégie de recherche avec un retour en arrière chronologique; l'inefficacité d'une telle stratégie est connue *[Mackworth77]*, ce qui a conduit beaucoup d'auteurs à utiliser des stratégies de recherche intelligentes *[Pereira83]*, *[Bruynooghe83]* qui sont largement déterministes *[Mellish81b]*, *[Milne83]*.

Généralement il n'existe pas d'algorithme de décision pour tout prédicat d'un programme s'il est déterministe ou non *[Sawamura86]*. Néamoins nous pouvons déterminer le déterminisme de certains prédicats moyennant l'utilisation des types et des modes conjointement.

Dans un appel d'un prédicat p 1-aire ayant le mode Entrée et une constante "a " comme argument d'appel, pour que cet appel réussisse il faut qu'il y ait une définition au moins parmi celles de p dont l'argument puisse s'unifier avec "a". L'ensemble de solutions de cet argument doit donc contenir l'élément constant "a". Pour que cet appel puisse réussir une seconde fois (lors du backtrack), il faut que ce même argument d'une seconde définition de p soit "a". Si le domaine de solution du même argument d'une autre définition de p contient le terme "a", on peut dire que le prédicat p peut être non-déterministe. Si aucune autre définition de p ne contienne "a", aucun nouveau appel ne peut réussir et par conséquent le prédicat p est donc déterministe.

Mais comme on ne peut pas calculer l'ensemble de solution, nous ne pouvons pas décider si un prédicat est non-déterministe. La connaissance dont nous disposons est un sur-ensemble de l'ensemble de succès (le type).

Propriété: Soit S'' l'ensemble de solutions. Soit S' un sur-ensemble de S''. Si l'intersection d'un ensemble S avec S' est vide alors on peut décider que $S \cap S''$ est vide aussi.

Utilisant la connaissance du sur-ensemble de l'ensemble de solutions d'un prédicat (son type), des modes et l'utilisation de cette propriété de l'intersection, nous pouvons décider si un prédicat est déterministe.

En élargissant cette propriété à tous les arguments ayant un mode d'Entrée, on peut décider du déterminisme de certains prédicats en utilisant la règle suivante (Nous supposons connus les modes clos (Entrée et Sortie) des appels):

Règle: Soit le type d'un prédicat p donné par:

$$T_p = < T_{11} \vee \ldots \vee T_{1m}, \ldots, T_{n1} \vee \ldots \vee T_{nm} >$$

Soient $k_1, k_2, \ldots, k_l$ les indices des arguments ayant le mode Entrée. Si

toutes les définitions du prédicat p vérifient la propriété suivante, alors le prédicat p est déterministe.

Propriété: Soient $T_i = < T_{1i}, \ldots, T_{ni} >$ et $T_j = < T_{1j}, \ldots, T_{nj} >$ les types des définitions i et j. L'ensemble:

$$T_{k_1 i} \cap T_{k_1 j} \cup T_{k_2 i} \cap T_{k_2 j} \cup \ldots \cup T_{k_l i} \cap T_{k_l j} = \varnothing.$$

Cette propriété définit le déterminisme d'un prédicat à deux définitions et dont les arguments k_j (j=1,l) ont un mode d'Entrée. Pour l'appliquer à un prédicat ayant plus de deux définitions, nous l'appliquons à toutes les définitions deux à deux. Si un prédicat vérifie cette propriété pour toutes ses définitions alors il est (absolument) déterministe.

En compilation.

L'un des principaux obstacles à une utilisation plus large de PROLOG est l'inefficacité des programmes interprétés. La compilation de PROLOG a permi de gagner un facteur d'efficacité de l'ordre de 20. Les compilateurs existants sont basés sur la définition d'une machine abstraite PROLOG et générent des programmes objets exécutables par ces machines abstraites.

Deux idées principales sont à la base de la compilation de PROLOG:

* accélerer l'unification: le compilateur traduit les têtes des clauses en instructions spécialisant les unifications, guider l'appel des prédicats du corps de la clause, *typage des données facilitant ainsi l'appariement des arguments tout en accélerant la détection des incompatibilités.*

* *Accélerer le choix de la clause candidate à la résolution du but* courant. Des instructions d'aiguillage permettent un choix rapide de la clause unifiable en fonction du type des arguments du but courant.

Ainsi la connaissance de la structure des variables peut être utilisée pour améliorer les deux points précédents.

Détermination des modes

Il existe peu de programmes complètement multidirectionnels. La majorité des prédicats pré-définis sont directionnels et cette directionnalité est héritée par leur procédure d'appel directement ou indirectement. Ce mythe de directionnalité est discuté par Mc Dermott *[Mcdermot80]*. Il serait très intéressant d'avoir une stratégie pour PROLOG, permettant au programme d'avoir un répertoire de procédures multidirectionnelles, et de construire les programmes avec une directionnalité spécifique. Pour cela Warren *[Warren77]* a introduit la notion de mode pour préciser les directions dans lesquelles un prédicat peut être utilisé.

P. Mishra affirme *[Mishra84]* que si le système de type est utilisé conjointement avec un système d'inférence de mode, on pourrait améliorer l'efficacité de ce dernier système. En effet la génération des systèmes d'inéquations dans les systèmes d'inférence de type, décrit les unifications possibles durant toutes les exécutions réussies.

Cardinalité.

Le type d'un prédicat est un sur-ensemble de l'ensemble de succès. Une faible cardinalité de ce sur-ensemble donne une idée sur celle de l'ensemble de solution. Connaissant la cardinalité des prédicats, nous pouvons guider le choix du but à effacer *[Naish85]*.

Autres applications

* *Transformations de programmes.* Plusieurs transformations pour les programmes PROLOG ont été proposées dans la littérature *[Conrad86], [Debray85b], [Kanamori84], [Landais86], [Reddy84], [Stepenkova84], [Takenchi85], [Tamaki82], [Vasak83]*, en vue d'optimisation. Utiliser ces transformations conjointement avec la connaissance des structures des arguments pour les améliorer.

* *Modularité.* Vérification des types des modules *[Bron85]* (pour la compilation séparée ou décomposition des gros programmes en modules).

* *détermination des termes temporaires.* Dans une clause p_0:-p_1,...p_n, On dit qu'un terme appartenant au corps est temporaire, si sa représentation en fin de l'activation de la clause n'a pas de partage de structure avec la clause appelée p_0. Supposons qu'un terme temporaire ne partage pas de structure de données avec les termes du point du backtrack dans l'arbre de recherche enraciné en p_0, alors l'espace mémoire de ce terme peut êre récupéré à la fin de l'activation de la clause.

Il existe deux méthodes pour la détermination des termes temporaires, la méthode directe et la méthode indirecte. La méthode directe de ramasse miettes a été dévelopée par Bruynooghe *[Bruynooghe84]* pour l'implémentation de PROLOG. L'autre classe de méthodes est celle des méthodes directes avec deux méthodes (dynamique et statique). On trouve une méthode statique de détermination des termes temporaires moyennant la connaissance des structures des variables (type) dans *[Vataja84]*.

* *détection de boucles*: Le problème de boucles en Prolog a été étudié par plusieurs auteurs *[Convington85a]*, *[Covington85b]*, *[Nute85]*, *[Poole85]*. Dans notre méthode d'inférence nous avons traité de manière particulière les appels récursifs. Les éléments de type de tels prédicats ont une forme récursive. Nous pouvons analyser ces résultats pour détecter éventuellement certaines boucles.

* *Déboguage*: Connaissant les types des prédicats, on peut comparer les modèles inférés avec ceux de l'exécution pendant le déboguage. (Eventuellement réaliser un système de déboguage dynamique, utilisant les types des variables).

* Introduire des prédicats contraintes sur la structure des arguments d'une manière analogue à celle des prédicats "geler et wait" sur les variables.

* Inférence des types du point de vue interprétation abstraite *[Bruynooghe87a]*, *[Bruynooghe87b]*, *[Horiuchi87]*, *[Mellish86]*.

Conclusion

Le concept de type est bien connu pour les langages de programmation classiques. La plupart des systèmes de programmation logiques manquent de propriétés qui sont essentielles au développement de grand programmes. Spécialement en Prolog, il n'existe pas de moyen de spécifier formellement les types des arguments d'un prédicat. Il existe différentes tentatives de l'introduire dans la programmtion logique. Ainsi de nombreuses approches apparaissent dans la littérature dans un but d'étendre Prolog pour la prise en compte des types.

De telles informations peuvent être utilisées pour différentes raisons. Elle fournissent un bon moyen de documentation, dans un système modulaire elles peuvent être une partie de la description d'interface d'un module, elles peuvent fournir des informations utiles au programmeur, elles peuvent être un moyen d'optimisation et de vérification des programmes, elles peuvent fournir des informations permettant de produire un code plus efficace en compilation.

Deux approches existent actuellement pour étendre Prolog à prendre en compte de telles informations: l'approche déclarative et l'approche inférentielle.

Nous pensons qu'il serait intéressant d'avoir une approche souple; les déclarations de type exprimées doivent être statiquement vérifiées, mais il n'est pas obligatoire de tout typer. Le programmeur peut exprimer ses pensées et les connaissances dont il dispose et présente la sémantique du programme à travers des résultats qu'il désire obtenir sous forme de déclarations de type. Cette approche peut être utilisée conjointement avec un système d'inférence de type pour améliorer l'efficacité au niveau temps d'exécution et précision.

Nous avons présenté une étude sur la notion de type dans le langage de programmation PROLOG. A travers des exemples, nous avons montré l'intérêt d'introduire cette notion dans un but d'une meilleure sécurité de programmation. Deux travaux représentant l'approche déclarative et l'approche inférentielle ont été exposés. Nous avons montré l'intérêt de l'approche inférentielle sur l'approche déclarative, ceci afin de garder la simplicité de Prolog qui est un atout important. Cette extension doit donc respecter cette simplicité.

Nous avons proposé un système d'inférence de type. Ce système n'exige aucune information du programmeur. Il est basé sur une simulation de l'unification et une simulation de tous les appels possibles avec un traitement particulier des appels récursifs. La sémantique que nous avons donné à la notion de type d'un prédicat est un sur-ensemble de son ensemble de succès. Nous avons présenté quelques améliorations envisageables afin d'améliorer la précision du système. Nous avons présenté quelques applications des résultats de l'inférence dans un but de vérification et/ou d'optimisation en interprétation et/ou en compilation.

Perspectives

Pour ce qui est des perspectives, plusieurs directions de recherches sont envisagées actuellement pour l'utilisation de la connaissance des types dans un but d'une meilleure implémentation du langage.

D'un point de vue pratique, le chemin de l'utilisation de la connaissance des types dans les implémentations peut être envisagé. En effet, nous avons cité dans cette étude quelques applications possibles des résultats de l'inférence des types à la compilation et/ou à l'interprétation. Nous pensons que la réalisation d'un compilateur pour un PROLOG typé prenant en compte les optimisations citées précédemment serait une voie intéressante.

D'un autre point de vue, il faudrait rattacher notre travail à l'interprétation abstraite des programmes PROLOG. Des travaux ont été effectués récemment sur ce domaine *[Kanamori87], [Bruynooghe87a, 87b].*

Annexe1: Le système d'inférence de type

Dans cette annexe, notre méthode d'inférence est présentée par des règles et des algorithmes. Nous faisons apparaître ses aspects opératoires et algorithmiques. Elle se résume à la détermination des types des prédicats contenant éventuellement des *variables de type* et la génération d'inéquations pour avoir les types de ces variables de type. La détermination des types des prédicats et la génération d'inéquations se font respectivement par application de deux ensembles de règles. Les inéquations sont générées par une simulation de l'unification et une simulation de tous les appels possibles des corps des clauses. Leurs transformations, pour obtenir les types des variables de type, sont effectuées par une exécution simulant la stratégie de résolution avec un traitement particulier pour les appels récursifs.

Si nous analysons les expressions d'inéquations générées lors de la simulation de l'unification, simulation de tous les appels possibles et éclatement des opérateurs ou et &, nous remarquons qu'elles expriment l'arbre et/ou avec un traitement particulier des appels récursifs.

Les étapes de l'inférence

L'inférence des types des prédicats d'un programme se fait en deux passes:

* Dans la première passe le programme est décomposé en parties indépendantes.

* Dans la seconde passe les types des prédicats de chaque partie indépendante sont inférés, utilisant éventuellement les types des prédicats d'autres parties indépendantes.

1. 1ère passe: décomposition du programme.

- Soit par exemple le programme "reverse" composé des prédicats "rev" et "append":

```
rev([],[]).
rev([A|B], C) :- rev(B, D), append(D, [A], C).
append([], K, K).
append([X|L], M, [X|N]) :- append(L, M, N).
```

L'inférence du type de "rev" utilise celui de "append" (rev appelle append) mais le type de "append" n'utilise pas de celui de "rev", (append n'appelle pas rev). Pour inférer le type de "rev" il est nécessaire d'inférer celui de "append" une première fois. Ce même type de "append" (à un renommage des any_i près) est inféré une seconde fois lors du traitement du prédicat "append". Cette répétition ne se produirait pas si nous inférions le type de "append", puis si nous utilisions ces résultats pour inférer celui de "rev".

- Soit l'exemple suivant composé de trois prédicats p, q et r:

```
p(a).
p(f(X)) :- q(X).
q(g(Y)) :- p(Y), r(Y).
r(b).
r(Z) :- r(f(Z))
```

Pour inférer le type de p nous devons inférer le type de q (p appelle q). Pour inférer le type de q nous devons inférer les types de p et de r. Les clauses de r n'appellent aucun autre prédicat (autre que r). Le calcul du type de r ne nécessite donc aucun type d'un autre prédicat. Nous commençons par inférer le type de r. Nous inférons ensuite les types de p et q (qui s'appellent entre eux dans les deux sens) en même temps, utilisant le type de r.

Pour avoir l'ordre d'inférence des types des prédicats d'un programme, nous décomposons ce programme en parties indépendantes. Pour décomposer un programme, nous dressons le graphe d'appel lui correspondant. Dans un graphe d'appel les noeuds sont les prédicats du

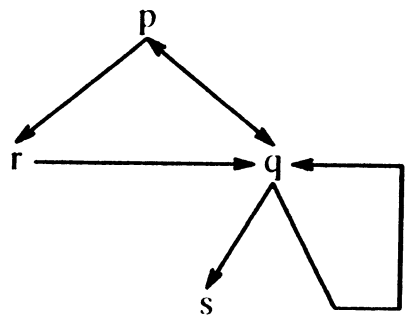
programme, et les arcs entre ces noeuds sont formés de la manière suivante: nous avons un arc du noeud p au noeud q si une clause de tête p appelle le prédicat q. Nous cherchons les composantes fortement connexes du graphe. Nous remplaçons ensuite chaque composante fortement connexe par un noeud. Nous obtenons un graphe de dépendance (sans cycles). Nous commençons par inférer les types des prédicats des feuilles, puis nous remontons jusqu'aux racines en transmettant les résultats (types inférés) des noeuds fils aux noeuds péres.

Une partie indépendante (composante fortement connexe) du graphe d'appels contient tous les prédicats têtes de clauses qui s'appellent entre eux dans les 2 sens d'une manière directe ou indirecte. On trouve un algorithme pour déterminer les composantes fortement connexes d'un graphe dans *[Tarjan72]*.

Exemple. Soit le programme suivant:

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

donnant comme graphe d'appel:

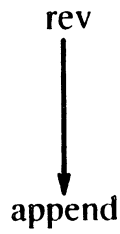


Nous avons deux composantes fortement connexes ((p, q, r) et (s)). Le graphe de dépendance est donc:

Nous inférons le type de s puis ceux de p, q et r en même temps en utilisant le type de s.

Nous avons réalisé un programme écrit en Prolog pour décomposer un programme en parties indépendantes. Le même algorithme a été utilisé pour un même traitement que précédemment pour les modes *[Oudot87]*.

Le graphe de dépendance du 1er exemple précédent est:



Le graphe de dépendance de l'exemple (2) précédent est:

2. $2^{ème}$ passe: Inférence des types d'une partie indépendante.

L'inférence des types des prédicats têtes de clauses d'une partie indépendante se fait en 3 étapes:

(i) Calcul des types des prédicats à partir des têtes des clauses. Ces types contiennent éventuellement des variables de type qu'il faut remplacer par leurs types. Nous formons les équations de type $T_{ij} = t$ où t (contenant des variables de type éventuellement) est le type de l'argument ayant T_{ij} comme composante de type.

(ii) Génération des inéquations par simulation de l'unification et simulation de tous les appels possibles des corps des clauses. (Parcours d'un niveau de l'arbre et/ou en largeur au niveau de chaque clause avec corps).

(iii) Transformation des inéquations par une exécution simulant la stratégie de résolution en traitant d'une manière particulière les appels récursifs. (Parcours de l'arbre et/ou en profondeur en traitant de manière particulière les appels récursifs). Nous résolvons ensuite les équations de type.

2.1. Calcul des types des prédicats.

Nous rappelons que le type T_p d'un prédicat p n-aire tête de m clauses est un n-uplet $< T_1, ..., T_n >$. La i^{eme} composante T_i est la réunion ordonnée $< T_{i1} \vee ... \vee T_{im} >$ de tous les types T_{ij} des $i^{èmes}$ arguments $(i=1,n)$ des définitions j $(j=1,m)$ du prédicat p.

Une équation de type est de la forme $T_{ij} = t$ où t est le type de l'argument ayant T_{ij} comme composante de type.

2.1.1. Règles de calcul des types.

Soit la définition générale d'un prédicat p:

$$\begin{array}{l} p_1 :- e_1. \\ p_2 :- e_2. \\ \quad \ldots\ldots \\ p_n :- e_n. \end{array} \qquad \text{avec } e_i = \begin{cases} \text{conjonction de formules atomiques} \\ \text{ou} \\ \varnothing \end{cases}$$

En appliquant les règles suivantes à une partie indépendante d'un programme PROLOG, nous obtenons les types des prédicats têtes des clauses contenant éventuellement des variables de type.

1) Si un argument est un atome alors son type est cet atome.

2) Si un argument est une variable qui n'apparaît que dans la tête d'une clause alors son type est "any_i". Changer l'indice de any_i à chaque passage d'une variable à une autre.

3) Si une variable X apparaît dans la tête et dans le corps de la clause, nous lui faisons correspondre **une variable de type** X qu'il faut remplacer par son type. (Renommer les variables à chaque passage d'une clause à une autre)

4) Si l'argument est un terme fonctionnel $f(a_1, \ldots, a_n)$, son type est $f(t_1, \ldots, t_n)$ où t_i est le type de a_i $(i=1,n)$.

5) Soit la $j^{ème}$ définition d'une formule atomique $p(a_{1j}, \ldots, a_{nj})$ tête d'une clause où chaque a_{ij} a un type T_{ij} $(i=1,n)$, le type de p_j est: $T_{p_j} = < T_{1j}, \ldots, T_{nj} >$.

6) Soient les m définitions p_i $(i=1,m)$ d'un prédicat p.

$$\begin{array}{l} p_1 t_1 :- e_1. \quad \text{avec } T_{p_1} = T_1 \\ p_2 t_2 :- e_2. \quad \text{avec } T_{p_2} = T_2 \\ \quad \ldots\ldots \qquad \ldots\ldots \\ p_m t_m :- e_m. \quad \text{avec } T_{p_m} = T_m \end{array}$$

Type de p

$$T_p = T_1 \vee T_2 \vee \ldots \vee T_m$$

Soient T_i les types du prédicat p avec m définitions, i=1,m.

$$T_1 = <T_{11}, \ldots, T_{n1}>$$
$$T_2 = <T_{12}, \ldots, T_{n2}>$$
$$\ldots\ldots \qquad \ldots\ldots$$
$$T_m = <T_{1m}, \ldots, T_{nm}>$$

disjoncion v ----------

$$T_p = T_1 \vee \ldots \vee T_m = < T_{11} \vee \ldots \vee T_{1m} \ , \ \ldots \ , \ T_{n1} \vee \ldots \vee t_{nm} >$$

Si une variable apparaît uniquement dans le corps de la clause en plus d'une position alors son type est déterminé pour pouvoir décider de la possibilité ou non de la suite d'appels envisagée. Nous calculons l'intersection des interprétations des types de cette variable en ces différentes positions. Si cette intersection est vide alors la suite d'appels envisagée est une suite impossible.

Si une variable n'apparaît que dans une seule position dans le corps de la clause alors son type ne fournit aucune information sur la possibilité ou non de l'unification. C'est une variable libre qu'on représente syntaxiquement (C-Prolog *[Pereira84]*) par le symbole indéfini "_".

2.1.2. Equations de type:

Nous produisons les équations de type $T_{ij} = t$ où t est le type de l'argument ayant T_{ij} comme composante de type.

$t = atome / any_i / X / f(t^*)$ où X est une variable de type.

Dans cette 1ère étape, nous n'avions traité que les têtes des clauses. Les types obtenus (membres droits des équations de type) contiennent éventuellement des variables de type (variables X de la règle (3)) qu'il faut remplacer par leurs types, obtenus en examinant les corps des clauses.

2.2. Génération des inéquations.

Si une clause n'a pas de corps (un fait), alors les variables apparaissant dans cette clause ont des types any_i. Par conséquent il n'y a pas de variables de type dans les membres droits des équations de type. Nous ne produisons pas d'inéquations pour cette clause.

Si la clause a un corps alors les équations de type contiennent des variables de type dans leurs membres droits. Ces variables correspondent à la variable X de la règle (3) de l'ensemble de règles précédent. Elles seront remplacées par leurs types obtenus après la transformation des inéquations générées au cours de cette étape. Cette génération est faite par simulation de l'unification: (un argument t_1 d'un appel unifié avec un argument ayant le type t_2 d'une définition est simulé par $t_1 \leq t_2$), et par simulation de toutes les possibilités d'appels des corps des clauses.

2.2.1. Règles de génération des inéquations.

En appliquant les règles suivantes à une partie indépendante d'un programme PROLOG, nous obtenons une expression d'inéquations simulant tous les appels possibles des corps des clauses. Nous parcourons un niveau de l'arbre et/ou en largeur au niveau de chaque clause avec corps.

a) L'appel de la i^{eme} clause de p. $p_i(t'_1,....t'_n)$:-$corps_i$ par $p(t_1,....t_n)$, produit le système d'inéquations:

$$" s_i = p(t_1 \leq t'_1 \ \& \ ..... \ \& \ t_n \leq t'_n) \quad corps_i"$$

Le nombre d'inéquations de s_i est égal à l'arité de p. Les variables des t'_i n'apparaissant que dans la tête de la clause sont remplacées par des any_i. Les variables restantes sont des variables de type. Si p appelle un predicat dont le type est déjà calculé alors $corps_i = \emptyset$.

b) D'une suite d'appel $p_1, ... p_n$ (une possibilité d'appel pour chaque but p_i), nous produisons le système d'inéquations formé par la conjonction de tous les systèmes s_i générés par chaque appel p_i de la règle **(a)**:

$$S_j = s_1 \,\&\, s_2 \,\&\, \ldots \,\&\, s_n$$

Le nombre d'inéquations du système S_j est égal à la somme des arités des buts p_i pour i=1,n.

c) Soit une clause $p :- p_1, \ldots, p_n$. Supposons que nous avons m_i définitions p_i^j pour le but p_i (j=1,m_i et i=1,n). Nous obtenons $\Pi\, m_i$ (produit des m_i i=i,n) suites d'appel possibles. Ces Πm_i suites d'appels sont produites de la manière suivante:

$$SSp = \overset{k=\Pi m_i}{\underset{j=1}{ou}} S_j = \overset{m_1}{\underset{j=1}{ou}} appel(p_1\ p_1^{\,j})) \& (\overset{m_2}{\underset{k=1}{ou}} appel(p_2\ p_2^{\,k})) \& \ldots \& (\overset{m_i}{\underset{l=1}{ou}} appel(p_i\ p_i^{\,l})) \& \ldots \& (\overset{m_n}{\underset{r=1}{ou}} appel(p_n\ p_n^{\,r}))$$

où le signe "ou" est pris dans le sens de "ou" et le signe "&" est pris dans le sens de "et". $Appel(p_i\ p_i^{\,j})$ signifie que p_i est unifié avec (appelle) sa $j^{ème}$ définition. Nous renommons les variables et les any_i à chaque passage d'un appel à un autre. Nous obtenons la disjonction de systèmes suivante:

$$SS_p = S_1 \text{ ou } S_2 \text{ ou } \ldots \text{ ou } S_k \qquad \text{avec } k = \Pi\, m_j \text{ pour } j=1,n.$$

où chaque système S_j correspond à la j^{eme} combinaison parmi les $\Pi\, m_i$ possibles (j = 1,Πm_i et i=1,n) produite par **(b)**. Toutes les expressions S_i de SS_p ont les mêmes membres gauches des inéquations. Ils correspondent aux arguments des buts du corps de la clause.

d) Pour toutes les r définitions d'un prédicat p (une procédure), nous prenons la disjonction **v** des SS_{p_i}. La $i^{ème}$ expression SS_{p_i} correspond à la $i^{ème}$ clause du prédicat p produite par la règle **(c)**. Nous obtenons:

$$I_p = p(SS_{p_1} \text{ v } SS_{p_2} \text{ v } \ldots \text{ v } SS_{p_r})$$

e) Pour obtenir l'expression d'inéquations d'une partie indépendante contenant les prédicats p, q, r,..... s têtes de clauses, nous faisons la conjonction de toutes les expressions $I_p, I_q, I_r, \ldots I_s$ de ces prédicats produites chacune par **(d)**. Nous aurons l'expression finale:

$$I = I_p \,\&\, I_q \,\&\, I_r \,\&\, \ldots \,\&\, I_s$$

2.3. Transformation des inéquations

Lors de la précédente étape de génération des inéquations, nous avons simulé toutes les combinaisons possibles d'appels. Nous effectuons un premier filtrage de chaque expression d'inéquations de (**b**) pour éliminer éventuellement des systèmes impossibles correspondant à des unifications impossibles. Les règles de filtrage ont été présentées dans le chapitre 3. Les systèmes restants seront transformés comme ci-dessous.

2.3.1. Principe de la transformation.

Dans l'inférence des types, la plus importante étape est la transformation de l'expression d'inéquations générée lors de la phase précédente. Cette transformation est basée sur une simulation du parcours de l'arbre et/ou en profondeur d'abord. Nous simulons ce parcours, en effectuant un traitement particulier sur les appels récursifs. En traitant ces appels récursifs de cette manière, le parcours de l'arbre s'arrête toujours, il en est de même pour la transformation des inéquations. En termes d'appel de prédicats, l'enchaînement d'appels s'arrête toujours.

Pour déterminer le type d'un prédicat p_i tête de la $i^{ème}$ clause, nous déterminons les types de ses variables de type; nous examinons toutes les configurations d'appels possibles du corps de la clause; en termes d'inéquations nous transformons la i^{eme} expression de I_p de (**d**) simulant toutes les configurations d'appel de la $i^{ème}$ clause du prédicat p.

Pour transformer une configuration d'appel représentée par l'expression d'inéquations s_1 & ... & s_n, nous transformons tous les systèmes s_i pour i=1,n. Chaque s_i est de la forme $p(t_1 \leq t'_1$ & ... & $t_n \leq t'_n)$ $corps_i$.

Un système $p(t_1 \leq t'_1$ & ... & $t_n \leq t'_n)$ $corps_i$ est "*sans appel*" (ou transformé) lorsqu'il ne contient plus de variables de type dans les parties droites de ses inéquations; ce qui correspond à l'unification de son prédicat avec un fait (c'est-à-dire $corps_i = \varnothing$), ou avec un prédicat dont le type est déjà calculé.

Il devient "*sans appel*" aussi lorsqu'il forme un appel récursif. Nous lui appliquons le traitement spécial des appels récursifs.

Le cas restant est celui où "$corps_i$" n'est pas vide et p_i ne forme pas un appel récursif. Dans ce cas, nous mémorisons le prédicat p_i et nous déterminons le type de chaque but de $corps_i$. La mémorisation des prédicats traités permet de détecter les appels récursifs. Une fois les domaines des variables de type des buts "$corps_i$" déterminés, nous remplaçons les variables de type des t'_i des inéquations $t_i \leq t'_i$ par leurs solutions maximales en explorant les résultats combinatoirement. Le système d'inéquations devient alors *"sans appels"*.

2.3.2. Algorithmes de transformation

Dans ce paragraphe nous présentons les algorithmes de transformation des systèmes d'inéquations que nous avons développé. Nous avons montrer comment certains algorithmes évitent de boucler en mémorisant certaines informations.

Entrée: Pour calculer le type des prédicats, nous avons en entrée, les types des prédicats sous la forme:

$$T_p = < T_{11} \vee \ldots \vee T_{1m} , \ldots . , T_{n1} \vee \ldots . \vee T_{nm} >$$

et les équations de type $T_{ij}=t$ correspondantes aux composantes de type. Nous avons aussi l'expression d'inéquations suivante:

$$I = I_p \,\&\, I_q \,\&\, \ldots . \,\&\, I_s \qquad (1)$$

où chaque expression I_p est donnée par:

$$I_p = p(SS_{p_1} \vee SS_{p_2} \vee \ldots . \vee SS_{p_m}) \qquad (2)$$

où une expression SS_{p_i} correspond à la $i^{ème}$ clause de p:

$$SS_{p_i} = p(S_1 \text{ ou } S_2 \text{ ou } \ldots . \text{ ou } S_k) \quad \text{où } k \leq \Pi m_j \qquad (3)$$

où chaque expression S_j correspond à une combinaison d'appel possible des buts du corps de la clause. Chaque expression S_j est donnée par:

$$S_j = s_1 \,\&\, s_2 \,\&\, \ldots \,\&\, s_m \qquad (4)$$

où s_i correspond à un système d'inéquations correspondant à un appel de prédicat, donné par: $s_i = p(t_1 \leq t'_1 \,\&\, \ldots . \,\&\, t_n \leq t'_n)\, corps_i$ (5)

L'Algorithme: La transformation des inéquations se fait comme suit:

(a)* Pour calculer le type d'un prédicat p têtes de clauses nous calculons le type de ses clauses par (b), (nous transformons toutes les expressions de (2)).

(b)* Pour calculer le type d'un prédicat p_i tête d'une clause, mémoriser le prédicat p_i, transformer toutes les expressions S_i de (3) par (c). Aller à (g).

(c)* Pour une suite d'appels, transformer les systèmes de (4) par (d). Aller à (f).

(d)* - Si le but est unifié avec un fait alors le système est *sans appels*. Aller à (e).
- Si le but forme un appel récursif (prédicat mémorisé) alors lui appliquer le traitement spécial des appels récursifs, il devient *sans appels*. Aller à (e).
- Si le système a un corps alors calculer les domaines des nouvelles variables de type des buts par (b) après renommage des any_i.
Calculer les intersections éventuelles et possibles(†).
Remplacer les variables de type par leurs solutions maximales (combinatoirement).
Le système devient alors *sans appels*. Aller à (e).

(e)* Normaliser le système. Aller à (c).

(f)* Séparer les inéquations d'une expression (4) de systèmes sans appels en:
-(i) Inéquations productives (les membres gauches sont des variables de type).
-(ii) Inéquations contraintes formée par le reste des inéquations.
. Eliminer les inéquations possibles et toujours vraies de cette classe (ii). (††)
Si une classe (ii) est impossible alors la classe (i) correspondante est impossible.
Les inéquations restantes forment une contrainte C_j sur la classe (i).

(g) * Appliquer l'opérateur "+" aux classes d'inéquations productives de (3).

* Prendre les solutions maximales des inéquations de (g) (combinatoirement).
Calculer les intersections éventuelles de ces solutions maximales (†).
Produire autant d'équations T_{ij} = t qu'il y a de domaines pour X apparaissant dans t.

(†) Tous les termes sont définis (pas de any_i, ni d'appels récursifs).
(††) Pas de composantes de type, ni de variables d'hypothèse ni de contraintes.

Inéquations toujours vraies:

Les inéquations suivantes sont toujours vraies.

- $a \leq t'$ et $a \in I(t')$ avec a un atome.
- $X \leq t_1$ & ... & $X \leq t_n$ et $I(t_1) \cap ... \cap I(t_n) \neq \emptyset$
- Une inéquation $Y \leq t$ et Y n'apparaît que dans cette position comme membre gauche.

Inéquations toujours fausses:

Les inéquations suivantes sont toujours fausses.

- $a \leq t'$ et $a \notin I(t')$ avec a un atome.
- $X \leq t_1$ & ... & $X \leq t_n$ et $I(t_1) \cap ... \cap I(t_n) = \emptyset$.

Traitement des appels récursifs.

Le traitement particulier des appels récursifs s'effectue de la même manière que dans le chapitre 3. Si au cours du calcul du type d'un prédicat p, nous rencontrons une unification d'un but q_j avec sa définition *q_j-corps* par où nous sommes déjà passé (q_j est mémorisé-nous sommes entrain d'inférer le type de q_j), nous prenons alors les j^{emes} composantes de type T_{ij} correspondant à la j^{eme} clause du prédicat q. Le système d'inéquations "$t_1 \leq t'_1$ & ....& $t_n \leq t'_n$ $corps_i$" de cet appel devient alors " $t_1 \leq T_{1j}$ & ....& $t_n \leq T_{nj}$ pas de corps". Les membres droits des inéquations sont remplacés par les composantes de type T_{ij} i=1.n du type de q. Le système est ainsi devenu sans appels.

Normalisation.

Nous normalisons les systèmes sans appels, en éliminant toujours les systèmes impossibles. La normalisation appliquée à un système d'inéquations s_i sans appels ($corps_i=\emptyset$), donne un autre système d'inéquations s'_i (éventuellement s_i) où ses membres gauches sont des termes simples (atomes ou variables) correspondant aux termes syntaxiques du but, plus éventuellement un ensemble d'hypothèses. Les règles de normalisation ont été présentées dans le chapitre 3.

Sortie de la transformation:

En sortie de cet algorithme de transformation, nous obtenons les équations de type suivantes (une composante de type peut avoir plusieurs équations).

$T_{ij} = YC_k$

$YC_k = Y \ / \ Y\backslash C_i$ (Y avec une contrainte C_i)

$Y = atome \ / \ any_i \ / \ YC_1 + ... + YC_n \ / \ r \ ! \ (t_1 + ... + f(r) + ... + t_n) \ / \ T_{kl} \ / \ T'_i \ / \ f(YC_1, ..., YC_n)$

où t_i est un élément de type , T_{kl} est une composante de type, T'_i est une variable d'hypothèse et C_i une contrainte.

2.3.3. Résolution des équations de type.

Entrée: L'entrée de cette étape est la sortie de l'étape précédente.

Elimination des composantes de type et des variables d'hypothèses.

Nous avons dit dans le chapitre 3 qu'une équation de type est résolue si son membre droit est un élément de type. Le but de cette résolution est donc d'éliminer les composantes de type T_{kl}, les variables d'hypothèses T'_i et les contraintes C_i des membres droits des équations de type. Nous ne passons par cette résolution que si la partie indépendante contient un appel récursif. C'est à ce niveau donc qu'apparaissent les éléments de type récursifs.

Si la partie indépendante ne contient pas d'appels récursifs alors les membres droits des équations de type ne contiennent ni composantes de type T_{ij}, ni variables d'hypothèse T'_i ni contraintes C_i (pas de traitement spécial des appels récursifs). Les équations de type sont donc résolues et nous passons directement au calcul de l'intersection.

Algorithme de résolution des équations de type:

(a) * Empiler la composante de type T_{ij}.

* Sol = ∅. Traiter toute la partie droite de l'équation de type $T_{ij} = Y$.

(b) * **Si** Y est:

- un atome ou any_i **alors** ajouter Y à Sol.

- Une union $Y_1 + ... + Y_n$ **alors**:
  - Résoudre chaque Y_i par (b).
  - Calculer les intersections éventuelles et possibles(†)
  - Produire autant de Y qu'il y a de solutions Y_i.

- Un élément de type $r!(t_1 + ... f(r) + ... + t_n)$ **alors** l'ajouter à Sol.

- Une composante de type T_{kl} **alors**:
  - **Si** $T_{kl} \in$ Pile **alors** ajouter r_{kl} (récursion au niveau de T_{kl})
    Sinon Renommer les any_i de $T_{kl} = t \rightarrow T_{kl} = t'$
    - Résoudre les équations de type $T_{kl} = t'$ par (a),
    - Calculer les intersections éventuelles et possibles, (†)
    - Produire autant de Y qu'il y a de solutions.

- $f(YC_1, ..., YC_n)$ **alors** traiter ses arguments par (b),
  - Calculer les intersections éventuelles et possibles (†) pour YC_i,
  - Produire autant de Y qu'il y a de solutions.

- Une variable d'hypothèse T'_k **alors**:
  - Décomposer l'hypothèse correspondante par l'algorithme ci-dessous,
  - Traiter les résultats de la décomposition par (b),
  - Calculer les intersections éventuelles et possibles (†),
  - Produire autant de Y qu'il y a de solutions.

- **Si** récursion au niveau de T_{ij} **alors** Solution = r_{ij} ! Sol **sinon** Solution = Sol.

(†) Tous les termes sont définis (pas de any_i, ni de contraintes).

Sortie de la résolution: Nous obtenons en sortie de l'algorithme, des équations de type de la forme (un T_{ij} peut avoir des équations multiples):

T_{ij} = YC

YC = élément de type / élément de type(C_i) (avec contrainte)

Algorithme de décomposition. Pour déterminer les valeurs prises par une variable d'hypothèse $T'_i = \{ t / f(...t...) \in I(T_{ij}) \}$, nous décomposons T_{ij} avec f(t) par application de l'algorithme suivant en interprétation ensembliste: (tous les éléments de T_{ij} peuvent être sous une contrainte C_k, auquel cas la solution sera aussi sous cette contrainte).

Entrée - Variable d'hypothèse $T'_k = \{ t / f(..., t, ...) \in I(T_{ij}) \}$.

- Equations de type $T_{ij} = t$.
- Hypothèses générées par la normalisation.

L'Algorithme.

* Pile = ∅

* empiler (T'_k, T_{ij})

(a) * **Si** t = atome **alors** ∅.

* **Si** $t = f(t_1,...,t_n)$ **alors** $T'_i = t_i$.

* **Si** $t = any_i$ **alors** $T'_i = any_j$

* **Si** $t = t_1 + ... + t_n$ **alors** décomposer T'_i avec t_i pour i=1,n.

* **Si** $t = r!(t_1 + ... + t_n)$ **alors** décomposer T'_k avec les t_i en remplaçant r par $r!(t_1 + ... + f(r) + ... + t_n)$ chaque fois que c'est nécessaire en renommant les any_i.

* **Si** $t = T_{kh}$ **alors si** $(T'_k, T_{kh}) \in$ Pile
alors $T'_i = any_j$ (solution maximale) (*)
sinon empiler (T'_k, T_{kh}), aller en (a).

* **Si** $t = T'_j$ variable d'hypothèse sur T_{rs} **alors**

- empiler (T'_j, T_{rs}), aller en (a)
- Soient S les résultats de la décomposition.
- Calculer les intersections possibles et éventuelles.(†)
- Décomposer T'_k avec S.

(†)Tous les termes sont définis (pas de any_i ni de contraintes).

Sortie: Domaine de la variable d'hypothèse T'_k (éventuellement multiple).

Remarque: Dans cet algorithme nous faisons remarquer qu'au niveau de l'étape (*), nous ne continuons pas la décomposition de T'_i avec le résultat de la décomposition de T_{kh}. Nous donnons à T'_i la solution maximale any_j. Une décomposition à fond est une éventualité permettant de préciser mieux le domaine de la variable d'hypothèse.

Vérification des contraintes.

Pour vérifier la possibilité ou non d'une expression de systèmes normalisés, nous avons séparé les inéquations en classe d'inéquations productives et classe d'inéquations contraintes. Nous avons vérifié les inéquations possibles (cas de non récursivité). Les inéquations restantes formaient une contrainte sur la classe d'inéquations productives. Nous vérifions maintenant ces contraintes.

Entrée: En entrée de cette vérification, nous avons les expressions:

- C_k à vérifier.
- T_{ij} = YC.
- YC = élement de type / élément de type (C_l) (avec contrainte(s)).

Nous appliquons la technique de vérification des contraintes introduite dans le chapitre 3.

Sortie:

Nous obtenons en Sortie la liste des contraintes fausses, la liste des contraintes vraies, ainsi que les équations de type sans contraintes:

T_{ij} = élément de type

Une composante de type T_{ij} peut avoir des équations de type multiples.

Intersection.

Entrée: L'entrée de l'algorithme d'intersection est:

T_{ij} = élément de type

Une composante de type T_{ij} peut avoir plusieurs équations.

Nous calculons l'intersection des membres droits des équations de type ayant le même T_{ij} comme membre gauche. Ceci revient à faire l'intersection des interprétations des éléments de type correspondants. Nous utilisons les règles commutatives données dans le chapitre 3.

Sortie:

En sortie, nous obtenons un élément de type pour chaque composante de type T_{ij} donnée par:

T_{ij} = élément de type

Chaque composante de type T_{ij} a une équation unique dont le membre droit est le type de l'argument ayant T_{ij} comme composante de type.

Nous simplifions les résultats comme précédemment.

3. Utilisation des types calculés.

Dans la $1^{ère}$ passe nous avons décomposé le programme en parties indépendantes. Pour calculer le type des prédicats d'une partie indépendante nous utilisons éventuellement des types déjà calculés.

Si les membres droits des équations de type d'une composante de type T_{ij} de l'entrée de l'algorithme d'intersection ne contiennent pas de any_i alors le type S simplifié obtenu après le calcul de l'intersection éventuelle sera utilisé pour les inférences ultérieures. Lors de la simulation de l'unification d'un terme t_i avec cet argument ayant T_{ij} comme composante de type, nous produisons alors l'inéquation $t_i \leq S$.

Par contre si ces membres droits contiennent un any_i, alors ceux sont ces expressions multiples qui seront utilisées combinatoirement pour les inférences ultérieures. Ceci afin de garantir la propagation des nouvelles formes éventuelles des any_i. Nous produisons alors l'inéquation

$$t_i \leq t'_1 \,\&\, \ldots \,\&\, t'_n.$$

Au niveau du filtrage, de la normalisation et de la prise de la solution maximale cette inéquation est équivalente à la conjonction:

$$t_i \leq t'_1 \,\&\, t_i \leq t'_2 \,\&\, \ldots \,\&\, t_i \leq t'_n$$

4. Possibilité de déclaration des types.

Les déclarations des types des prédicats connus (ou imposés) par le programmeur peuvent être prises en compte par le système d'inférence pour améliorer son temps de réponse. Ces déclaration se font pour les prédicats souvent utilisés dans les programmes PROLOG, ou les faits. Elles sont données de la même manière que les types des prédicats prédéfinis.

Les déclarations se font de la manière suivante:

prédéfini(Prédicat, Arité, [p(T_{11},....,T_{n1}),....,p(T_{1m},....,T_{nm})]).

Exemple.

Déclarer le type de equal(X, X). par prédéfini(equal, 2, [equal(any_1, any_1)]).

ANNEXE: Exemples d'inference.

```
append([],K,K).
append([X|L],M,[X|N]) :- append(L,M,N).


equal(L,L).


p(f(a)).
p(A) :- p(f(A)),p(g(A)).


pp(a).
pp(f(X)) :- qq(X).

qq(b).
qq(g(Y)) :- rr(Y).

rr(c).
rr(t(Z)) :- pp(Z).


pere(a,b).
pere(b,c).

gpere(X,Y) :- pere(X,Z),pere(Z,Y).


graphe(a,b).
graphe(b,c).
graphe(X,Y) :- graphe(X,Z),graphe(Z,Y).


long([]).
long([_]).
long([A,B|C]) :- A<B,long([B|C]).


pop(a).
pop(f(A,B)) :- pop(A),pop(B).


pfg([]).
pfg(f(A)) :- pfg(A).
pfg(g(B)) :- pfg(B).
```

```
substract(s(0),0).
substract(s(s(X)),s(Y)) :- substract(s(X),Y).


liste([]).
liste([X|L]) :- liste(L).


appendliste([],[L],[L]).
appendliste([X|L],[M],[X|N]) :- appendliste(L,[M],N).


tri([]).
tri([_]).
tri([A,B|C]) :- A<B,tri([B|C]).


longueur([],0).
longueur([A|B],N) :- longueur(B,M),+(M,s(0),N).


longueurfaux([],0).
longueurfaux([A|B],N) :- longueurfaux(N1,B),+(N1,N,s(0)).


even([]).
even([_|L]) :- odd(L).

odd([_]).
odd([_,_|L]) :- even([_|L]).


pequal(X) :- equal(X,f(X)).


frere(a,d).


cousin(A,B) :- pere(AA,A),pere(BB,B),frere(AA,BB).


p3(a).
p3(b).
p3(op(A,B)) :- p3(A),p3(B).


p4(_).
p4(X) :- q4(f(X)).
q4(Y) :- p4(g(Y)).
```

```
att(a).


vat(X,Y) :- vat1(a,X,Y).

vat1(b,c,f).
vat1(a,_,_).
vat1(a,a,a).


integer(0).
integer(s(X)) :- integer(X).


p11(A,B):-q11(B,A).

q11(X,Y):-eq(X,Y),eq(X,Y).

f11(A):-p11(A,f(A)).


q13(a).
q13(f(X)) :- q131(X).
q13(g(X)) :- q132(X).

q131(a).
q131(f(X)) :- q131(X).

q132(a).
q132(g(X)) :- q132(X).

p14(f(f(f(f(a))))).
p14(X) :- p14(f(X)).


q15(f(a)).
q15(X) :- q15(f(f(X))).



p16(f(a)).
p16(g(a)).
p16(A):-q16(A),p16(f(A)).

q16(B):-p16(g(B)).
```

```
p00(A,B,C):-p01(f(A),g(B),C).

p01(F,F,G):-p002(F,G).

p002(H,I):-q00(H,I).


q00(K,K).
q00(J,L):-q00(J,L).


p20(X,Y):-q20(X,Y).

q20(K,K).
q20(A,f(B)):-p20(s(A),B).


p21(f(f(0))).
p21(f(X)):-p21(f(f(X))).
```

```
< T11  V  T12  V  ...  V  T1m ,
  T21  V  T22  V       V  T2m ,
    ...       ...
  Tn1  V  Tn2  V       V  Tnm >
```

Tij correspond au type de l'argument i dans la definition j.

****

traitement de la partie independante: [p21]

```
p21(f(f(0))).
p21(f(X)):-p21(f(f(X))).
```

```
< f(f(0))   V   f(0 + any) >
```

traitement de la partie independante: [q00]

```
q00(K,K).
q00(J,L):-q00(J,L).
```

```
< any   V   r ! (any + r)   ,   any   V   r ! (any + r) >
```

traitement de la partie independante: [q15]

```
q15(f(a)).
q15(X):-q15(f(f(X))).
```

Dans la clause: q15(2), les suites d'appels commencant par: [q15(1)], sont des branches d'echec.

```
< f(a)   V   any >
```

traitement de la partie independante: [p14]

```
p14(f(f(f(f(a))))).
p14(X):-p14(f(X)).
```

```
< f(f(f(f(a))))   V   f(f(f(a))) + f(f(a)) + any >
```

traitement de la partie independante: [q132]

```
q132(a).
q132(g(X)):-q132(X).
```

< a V r ! (g(a + r)) >

traitement de la partie independante: [q131]

```
q131(a).
q131(f(a)):-q131(X).
```

< a V r ! (f(a + r)) >

traitement de la partie independante: [integer]

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

< 0 V r ! (s(0 + r)) >

traitement de la partie independante: [vat1]

```
vat1(b,c,f).
vat1(a,_,_).
vat1(a,a,a).
```

< b V a V a , c V any V a , f V any V a >

traitement de la partie independante: [att]

```
att(a).
```

< a >

traitement de la partie independante: [p3]

```
p3(a).
p3(b).
p3(op(A,B)):-p3(A),p3(B).
```

```
< a  V  b   V   r ! (op(a + b + r , a + b + r)) >
```

traitement de la partie independante: [frere]

```
fere(a,d).
```

```
< a ,  d >
```

traitement de la partie independante: [longueurfaux]

```
longueurfaux([],0).
longueurfaux([A|B],N):-longueurfaux(N1,B),+(N,N1,s(0)).
```

Dans la clause: longueurfaux(2), la suite d'appels: longueurfaux(1),+(0,0,0), est une branche d'echec

```
< []   V  { }  ,   0    V   { } >
```

traitement de la partie independante: [longueur]

```
longueur([],0).
longueur([A|B],N):-longueur(B,N1),+(N,s(0),N1)
```

```
< [] V  r ! ([any|[] + r])  ,   0  V  r ! (0 + s(r)) >
```

traitement de la partie independante: [tri]

```
tri([]).
tri([_]).
tri([A,B|C]):-A<B,tri([B|C]).
```

Dans la clause: tri(3), la suite d'appels: <(any,any),tri(1), est une branche d'echec

< [] V [any] V [any , any|[] + r ! ([any|[] + r])] >

traitement de la partie independante: [appendliste]

```
appendliste([],[K],[K]).
appendliste([A|B],[C],[A|D]):-appendliste(B,[C],D).
```

< [] V r ! ([any|[] + r]) , [any] V [any + r ! (any + r))] , [any] V r ! ([any|[any] + r]) >

traitement de la partie independante: [liste]

```
liste([]).
liste([A|B]):-liste(B).
```

< [] V r ! ([any|[] + r]) >

traitement de la partie independante: [substract]

```
substract(s(0),0).
substract(s(s(X)),s(Y)):-substract(s(X),Y).
```

< s(0) V s(s(0 + r ! (s(0 + r)))) , 0 V r ! (s(0 + r)) >

traitement de la partie independante: [pfg]

```
pfg([]).
pfg(f(A)):-pfg(A).
pfg(f(B)):-pfg(B).
```

traitement de la partie independante: [pop]

```
pop(a).
pop(f(A,B)):-pop(A),pop(B).
```

```
   <  a    V   r ! (f(a + r, a + r)) >
```

traitement de la partie independante: [long]

```
long([]).
long([_]).
long([A,B|C]):-A<B,long([B|C]).
```

```
  Dans la clause: long(3), la suite d'appels: <(any,any),long(1), est une branche d'echec
   <[]    V    [any]    V    [any, any|[] + r ! ([any|[] + r])] >
```

traitement de la partie independante: [graphe]

```
graphe(a,b).
graphe(b,c).
graphe(X,Y):-graphe(X,Z),graphe(Z,Y).
```

```
  Dans la clause: graphe(3), la suite d'appels: graphe(1),graphe(1), est une branche d'echec
  Dans la clause: graphe(3), la suite d'appels: graphe(2),graphe(1), est une branche d'echec
  Dans la clause: graphe(3), la suite d'appels: graphe(2),graphe(2), est une branche d'echec
```

```
   < a    V    b    V    a    ,    b    V    c    V    r ! (c + r) >
```

traitement de la partie independante: [pere]

```
pere(a,b).
pere(b,c).
```

```
   < a V  b    ,    b V  c >
```

traitement de la partie independante: [p]

```
p(f(a)).
p(A):-p(f(A)),p(g(A)).
```

Dans la clause: p(2), les suites d'appels commencant par: [p(1),p(1)], sont des branches d'echec
Dans la clause: p(2), les suites d'appels commencant par: [p(1),p(2)], sont des branches d'echec

< f(a) V a + any >

traitement de la partie independante: [equal]

```
equal(L,L).
```

< any , any >

traitement de la partie independante: [append]

```
append([],L,L).
append([X|L],M,[X|N]):-append(L,M,N).
```

< [] V r ! ([any|[] + r]) , any V r ! (any + r) , any V r ! ([any|any + r]) >

traitement de la partie independante: [p20,q20]

```
p20(X,Y):-q20(X,Y).

q20(K,K).
q20(A,f(B)):-p20(s(A),B).
```

q20 < any V any , any V r ! (f(s(any) + r)) >

p20 < any , r ! (any + f(r)) >

```
    <  any + r ! (any + r)    ,    any + r ! (any + r) >
```

traitement de la partie independante: [p16,q16]

```
p16(f(a)).
p16(g(a)).
p16(A):-q16(A),p16(f(A)).

q16(B):-p16(g(B)).
```

```
  Dans la clause: p16(1), les suites d'appels commencant par: [p16(2),q16(1)], sont des branches d'echec
  Dans la clause: q16(1), les suites d'appels commencant par: [p16(1)], sont des branches d'echec
```

```
q16 <  a + any >

p16 <  f(a)   V  g(a)   V  a + a + a + a + a + any >
```

traitement de la partie independante: [q11]

```
q11(X,Y):-equal(X,Y),equal(X,Y).

    <  any    ,    any >
```

traitement de la partie independante: [vat]

```
vat(X,Y):-vat1(a,X,Y).
```

Dans la clause vat(1), la suite d'appels: vat1(1), est une branche d'echec
< any + a , any + a >

traitement de la partie independante: [at]

```
at([]).
at([X|L]):-att(X),at(L).
```

< [] V r ! ([a|[] + r]) >

traitement de la partie independante: [p4,q4]

```
p4(_).
p4(X):-q4(f(X)).
q4(Y):-p4(g(X)).
```

q4 < any >

p4 < any V any >

traitement de la partie independante: [pequal]

```
pequal(X):-equal(X,f(X)).
```

< f(any) >

traitement de la partie independante: [even,odd]

```
even([]).
even([_|L]):-odd(L).

odd([_]).
```

```
odd < [any]  V    r ! ([any, any|[any] + r]) >

even < []   V   [any|[any] + [any, any|r ! ([any] + [any, any|r])]] >
```

traitement de la partie independante: [gpere]

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

```
  Dans la clause: gpere(1), la suite d'appels: pere(1),pere(1), est une branche d'echec
  Dans la clause: gpere(1), la suite d'appels: pere(2),pere(1), est une branche d'echec
  Dans la clause: gpere(1), la suite d'appels: pere(2),pere(2), est une branche d'echec

   < a , c >
```

traitement de la partie independante: [p01]

```
p01(F,F,G):-p002(F,G).

   < any + r ! (any + r)   ,   any + r ! (any + r)   ,   any + r ! (any + r) >
```

traitement de la partie independante: [q13]

```
q13(a).
q13(f(X)):-q131(X).
q13(g(X)):-q132(X).

   < a   V   f(a + r ! (f(a + r)))   V  g(a + r ! (g(a + r))) >
```

traitement de la partie independante: [p11]

```
p11(A,B):-q11(B,A).

   < any   ,   any >
```

traitement de la partie independante: [cousin]

```
cousin(A,B):-pere(AA,A),pere(BB,B),frere(AA,BB).
```

Dans la clause: cousin(1), la suite d'appels: pere(1),pere(1),frere(1), est une branche d'echec
Dans la clause: cousin(1), la suite d'appels: pere(1),pere(2),frere(1), est une branche d'echec
Dans la clause: cousin(1), la suite d'appels: pere(2),pere(1),frere(1), est une branche d'echec
Dans la clause: cousin(1), la suite d'appels: pere(2),pere(2),frere(1), est une branche d'echec

< { } , { } >

traitement de la partie independante: [qq,pp,rr]

```
pp(a).
pp(f(X)):-qq(X).

qq(b).
qq(g(Y)):-rr(Y).

rr(c).
rr(t(Z)):-pp(Z).
```

rr < c V r ! (t(a + f(b + g(c + r)))) >

pp < a V r ! (f(b + g(c + t(a + r)))) >

qq < b V r ! (g(c + t(a + f(b + r)))) >

traitement de la partie independante: [p00]

```
p00(A,B,C):-p01(f(A),g(B),C).
```

Dans la clause: p00(1), les suites d'appels commencant par: [p01(1)], sont des branches d'echec

< { } , { } , { } >

traitement de la partie independante: [f11]

< f(any) >

* En analysant les types inferes, on remarque que le pre'dicat "longueurfaux" est mal de'fini

* Le de'finition de "appendliste" est mieux adapter que celle de "append" pour manipuler des listes.

* On remarque que la clause definissant le predicat "cousin" est mal-type (si l'echec n'est pas solution souhaitee)

* De meme pour le predicat "p00", il est mal-type.

REFERENCES

[Ait-Kaci84] AIT-KACI H.

"A Lattice theoretic approach to computation based on a calculus of partially ordered type structures"
Phd Thesis, Computer and Information Science, University of Pensylvannia, Philadelphia U.S.A., 1984.

[Ait-Kaci85] AIT-KACI H. and NASR R.

"LOGIN A logic programming language with built-in inheritances"
MCC TR, No AI-68-85, Micro-electronical and computer technology Corporation, Austin USA, July 1985.

[Azzoune85] AZZOUNE H.

"Introduction des types en Prolog"
Rapport DEA, Ensimag-INPG France, Juin 1985.

[Azzoune88a] AZZOUNE H.

"Inférence des types et leurs applications en Prolog"
Colloque sur les micro-ordinateurs et systèmes, HCR, Arzew Algerie, 1-3 Fevrier 1988.

[Azzoune88b] AZZOUNE H.

"A type inference system in PROLOG"
CADE-9, 9th International Conference on Automated Deduction, Argonne Chicago Illinois USA, May 23-26 1988.
Lecture Notes in Computer Science, Springer Verlag N^{o} 310, 1988.
pp 258 - 277

[Bonnard86] BONNARD D., BROYER P., HUBERT E. et DIBERDER F.
"PROLOG de Delphia: un système ouvert de programmation logique"
Séminaire de Progr. en logique, Trégastel, Mai 1986.

[Boyer72] BOYER R.S. and MOORE J.S.
"The sharing of structure in theorem proving programs"
Machine Intelligence N° 7,
Edinburgh 1972, pp101 - 116

[Bron85] BRON C. and DIJKSTRA E.J.
"A Note on the checking of interfaces between separatly compiled modules"
ACM Sigplan Notices, Volume 20, No 8, Aout 1985, pp60 - 63

[Bruynooghe82] BRYUNOOGHE M.
"Adding redundancy to obtain more reliable and more readable PROLOG Programs"
Proceedings of the First International Logic Programming Conference
Marseille France, 1982, p 129-133.

[Bruynooghe83] BRUYNOOGHE M. and PEREIRA L.M.
Deduction revision by intelligent backtrack"
R.R. No 10, University de Lisbone Portugal, July 1983.

[Bruynooghe84] BRUYNOOGHE M..
"Garbage collection in PROLOG interpreters"
J. Campbell (ed)., Implementing of PROLOG, Ellis Horwood, 1984,

[Bruynooghe86a] BRUYNOOGHE M.
"Is logic programming 'real' programming?"
Proceedings AIMSA 86, Varna.

[Bruynooghe86b] BRUYNOOGHE M.
"Compile time garbage collection or how to transform programs into code with assignments"
IFIP TC 2, Working Conference on Program Specification and Transformation, April 15-17 1986, Bad Tolz, RFA.

[Bruynooghe86c] BRUYNOOGHE M. and WEEMEEUW P.
" Towards more efficiency of PROLOG on conventionnel hardware"
Repport CW 45, Dept. of Computer wetenshapen, Université catholique de Leuven Belgique, 1986.

[Bruynooghe87a] BRUYNOOGHE M.,JANSEENS G.,CALLEBAUT A., DEMEON B.
"Abstract interpretation: towards the global optimisation of PROLOG programs"
Procedings 1987 Symposium on Logic Prog. Aout-Sep 1987
San Francisco USA

[Bruynooghe87b] BRUYNOOGHE M.
"A framework for the abstract interpretation of logic programs"
repport CW 62, Katholieke Universiteit Leuven, Oct 87.

[Cardelli85] CARDELLI L. and WEGNER P.
"On understanding Types, Data Abstraction, and Polymorphism"
Computer Surveys, Vol 17, No 4 December 1985, pp 471 - 522

[Champeaux84] CHAMPEAUX D.
"About the Paterson-Wegman linear unification algorithm"
Research Report, Departement of Computer Science,
Tulane University, Avril 1984.

[Changlee73] CHANG LEE.
"Symbolic logic and mechanical proving"
Academic Press, New-York, 1973.

[Chassin86] J. CHASSIN DE KERGOMMEAUX
"Machines abstraites pour l'implantation de PROLOG"
Rapport de Recherche IMAG No 589 France,
Fevrier 1986.

[Clocksin81] CLOCKSIN W.F. and MELLISH C.S.
"Programming in PROLOG"
Springer-Verlag, New-York, 1981.

[Clocksin85] CLOCKSIN W.F.
"Design and simulation of a sequential PROLOG machine X-PROLOG"
New Generation Computing 3, 1985, p 101 - 120

[Colmerauer75] COLMERAUER A., KANOUI H., PASERA R. et ROUSSEL P.
"Un système de communication homme machine en français"
Rapport de Recherche, Groupe Intelligence Artificielle,
Université de Marseille France, Juin 1975.

[Colmerauer80] COLMERAUER A. et Equipe d'I.A.
"PROLOG I, II, III"
R.R. équipe d'I.A., Université de Marseille France, 1970 à 1988

[Colmerauer82a] COLMERAUER. A.
"Prolog and infinite trees"
Logic Prog., K.L. Clark and J.A. Tarnlund ed. Academic Press, 1982.
pp 231 - 251

[Colmerauer82b] COLMERAUER A.
"Prolog II, manuel de référence et modèle théorique"
Groupe d'I.A. Université de Marseille France, Mars 1982.

[Colmerauer83] COLMERAUER A. , KANOUI H. VAN CANEGHEM.
" PROLOG, bases théoriques et développements"
T.S.I., Volume 2, N^{O} 4, p 271 Juillet 1983, pp 271 - 311

[Condillac86] CONDILLAC M.
"PROLOG, fondements et applications"
Editions Dunod, 1986.

[Conrad86] CONRAD T.
"Réarrangement de clauses PROLOG"
Séminaire Programmation Logique, Trégastel CNET LANNION
21-22 mai 1986 p 345 - 355

[Covington85a] COVINGTON M.A.
"A further notes on looping in PROLOG"
ACM, Sigplan Notices, Volume 20, N^{O} 8, Aout 1985, pp 28 - 31.

[Covington85b] COVINGTON M.A.
"Eliminating unwanted loops in PROLOG"
ACM Sigplan Notices, Volume 20, N^{O} 8, Aout 1985, pp 20 - 26.

[Criss86] CRISS
"PROLOG CRISS une extension du langage PROLOG"
Université Grenoble II, Version 4.1, Fevrier 1986.

[Debray85a] DEBRAY S.K.
"Automatic mode inference for Prolog programs"
T.R. #85/19, Dept of Comp. Science, State University of
New-York at Stony-Brook, Juin 1985

[Debray85b] DEBRAY S.K.
"Detection and optimisation of functionnal compuation in PROLOG"
TR 85/020 Dept. of Comp. Sc, State Univ of New-York,
at Stony Brook, Aout 1985.

[Deransart84a] DERANSART P. and MALUSZINSKY J.
"Modelling data dependencies in logic programs by attribute schemata"
Rapport de Recherche N^{O} 323, INRIA,
Juillet 1984.

[Deransart84b] DERANSART P. and MALUSZINSKY J.
"Relating logic programs and attribute grammars"
Research Report, Departement of Computer and Information Science
Linkoping University, S-581 83, Sweden
Octobre 1984.

[Deitrich88] R. DEITRICH and HAGL F.
"A polymorphic Type system with subtypes for Prolog"
LNCS 300 1988, pp 79 - 92.

[Dincbas85] DINCBAS M., BOURGAULT S. et LEPAPE J.P.
"Notes de cours PROLOG"
Cours INRIA, Méthodes et Langages de l'I.A.,
Novembre 1985.

[Donz84] DONZ P. and HURTADO R.
"Le Langage D-PROLOG, initiation au langage de la 5eme génération"
Editions Tests, 1984.

[Fages84] FAGES F.
"Associative-commutative unification"
Rapport de Recherche N^{O} 287, INRIA France,
Avril 1984.

[Gang86] GANG Y. and ZHILIANG X.
"An efficient type system for PROLOG"
Information Processing IFIP 86, editions H.J. Kugler,
North Holland, 1986 page 355 - 359

[Garby85] GARBY D.M.
"N-PROLOG an extension of PROLOG with hypothetical Implication"
Journal of Logic Programming 4, 1985, p 251 - 283.

[Giannesini85] GIANNESINI F., KANOUI H., PASERO R. et VAN CANEGHEM M.
"PROLOG"
InterEditions, 1985.

[Goguen78] GOGUEN J.A., PHAPCHER J.W.
"An initial algebra approach to the specification, correctness and the implementation of abstract data types"
in R. Yeh (ed.), current trends in Programming Methodology IV: Data structuring 1978 p 80.

[Goguen84] GOGUEN J.A. and MESEGUER J.
"Equality, types, modules and generics for logic programs"
Journal of Logic Programming, N^{O} 2, 1984, p 179-210.

[Gordon78] GORDON M.J.C., MILNER A., MORRIS L. and Newly M. and Wadsworth C.
"A Metalanguage for interactive proof in LCF".
Proc. 5th ACM Symp. Principles of Programming Languages, Tucson, AZ, 1978.

[Gutag78] GUTAG J.V., HORNING J.J.
"The algebric specification of abstract data types"
acta informatica N0 10 , 1978, pp 27-52

[Harel] HAREL D. and PRATT V.
"Non-determinism in logic programs"
Conference Record of the 5th Annual ACM Symposium on Principles of Programmation Language, pp 203 - 213.

[Herbrand68] HERBRAND J.
"Recherches sur la théorie des démonstrations"
Thèse 1930, Ecrits Logiques de J. Herbrand, Pub. 1968.

[Hopcroft69] HOPCROFT J. and ULLMAN J.D.
"Formal languages and their relation to automata"
Addison-Wesley, 1969.

[Horiuchi87] HORIUCHI K. and KANAMORI T.
"Polymorphic type inference in PROLOG by abstract interpretation"
ICOT TR 263 June 1987

[Huet76] HUET G.
"Résolution d'équations dans les langages d'ordre 1,2..,w"
Thèse d'état, Mathématiques, Univ. Paris VII 1976.

[Icot80] I.C.O.T.
"Institute for new generation computer technology"
Technical Repports 1,2,3,... ,
Tokyo Japan.

[Kanamori84] KANAMORI T. and SEKI H.
"Verifications of PROLOG programs using an extension of execution" TR 96, ICOT Japan, December 1984.

[Kanamori85] KANAMORI T. and HORIUCHI K.
"Type inference in PROLOG and its applications"
ICOT, TR 095, December 1984 and IJCAI 85 (Proc. of the Ninth International Joint Conference on Artificial Intelligence,
11-13 Aout California USA, p 704.

[Kluzniak87] KLUZNIAK F.
"Type synthesis for ground PROLOG"
Proc. 4th Int. Conf. Logic Progr., Melbourne Australie, May 1987.
pp 788 - 816.

[Kowalksi73] KOWALSKI R and VAN EMDEN M.
"the semantic of predicat logic as programming language"
DCL memo 73, Univ of Edinbourgh.
Journal of the A.C.M. Volume 23 No 4 oct 76 pp 733 - 742.

[Kowalski74] KOWALSKI R.
"Predicate logic as programming language"
I.F.I.P congres 74, North Holland 1974, p 569 - 574.

[Kowalski79] KOWALSKI R.
"Logic for problem solving"
Elsevier Science Publishing, 1979.

[Kral73] KRAL J.

"The Equivalence of modes and the equivalence of finite automata "
Algol Buletin 35, Mars 1973, p 34-35.

[Landais86] LANDAIS G.

"Transformations de programmes PROLOG en vue de la compilation"
Séminaire Programmation Logique, Trégastel CNET,
LANNION France, 21-23 Mai 1986, p133 - 157.

[Lloyd84a] LOYD J.W.

"Foundations of logic programming"
Springer-Verlag, New-York 1984

[Lloyd84b] LOYd J.W. and TOPOR R.W.

"Making PROLOG more expressive"
Journal of Logic Programming, Volume 1, N^{o} 3, octobre 1984.
pp 225 - 240

[Macworth77] MACKWORTH A.

"Consistency in networks of relations"
Artificial Intelligence N^{o}8, 1977, p 99-118.

[Martelli82] MARTELLI A.

"An efficient unification algorithm"
ACM Transactions on Programming Language and Systems,
Volume 4, N^{o} 2, April 1982, pp 258 - 282.

[Mcdermott80] McDERMOTT D.

"The PROLOG phenomenon"
Sigart Newsletter 73, 16-20, Jully 1980.

[Mellish81a] MELLISH C.S.
"Automatic generation of mode declaration for PROLOG Programs" Dept of A.I., R.R. 163, Univ Edinburgh, Aug 81.

[Mellish81b] MELLISH C.S.
"Coping with uncertainly: noun phrase interpretation and early semantic analysis" PhD thesis, Univ. of Edinburgh, 1981.

[Mellish82] MELLISH C.S.
"An alternative to structure sharing in the implementation of a Prolog interpreter" Logic Prog., Clark et Tarnuld, Academic Press London 1982

[Mellish85] MELLISH C.S.
"Some global optimisations for a PROLOG compiler"
Journal of Logic Programming 2, 1, Apr 1985, p 43-66.

[Mellish86] MELLISH C.
"Abstract interpretation of PROLOG programs "
Proc. 3th Int. Conf. on Logic Programming, London July 1986, Lect. Notes in Comp. Sc. N^{O} 225, Springer-Verlag p 463 - 474.

[Metler72] METLER B. and MICHIE D.
"Machine intelligence"
N^{OS} 1 to 8, Edinburgh at University Press, 1972.

[Milner78] MILNER R.
"A Theory of polymorphism in programming"
J. Comp. Syst. Sc, V 17, N^{O} 30, 1978, p348.

[Milne83] MILNE R.
"Resolving lexical ambiguity in a determinism parse"
Phd thesis, University of Edinburgh, 1983

[Mishra84] MISHRA P.
"Towards a theory of types in PROLOG"
Proc. 1984 inter. Symp. Logic Prog. Atlantic city,
New Jersey, p 289-298.

[Mishra85] MISHRA P.
"Declaration-free type checking"
Conference Record of the 12th ACM Symposium on Principal
of Progr. Lang., New Orleans Louisiana, January 1985, p 7-21.

[Mycroft82] MYCROFT A. and O'KEEFE R.A.
"A Polymorphic type system for PROLOG"
Logic Progr. Workshop 83, Univ. Nova de Lisboa, 1982, p 107.
A.I. N0 23, 1984, pp 295 - 307.

[Naish85] NAISH L.
"Automating control for logic programs"
Journal of Logic Programming 3, 1985, p 167 - 183.

[Naish87] NAISH L.
"Specification = program + types"
Proceedings of the 7th Conf. on Foundations of Software Technology
and Theorical Computer Science, Pune India, December 1987.

[Nilsson82] NILSSON N.J.
"Principles of artificial intelligence "
Springer-Verlag, New-York, 1982.

[Nilsson83] NILSSON N.J.
"On the compilation of a domain-based PROLOG"
Information Processing, North Holland, IFIP 1983, p 293-298.

[Nute85] NUTE D.
"A Programming solution to certain problems with loops in PROLOG"
ACM Sigplan Notices, Volume 20, N° 8, Aout 1985, pp 32 - 37.

[Oudot86] OUDOT O.
"Utilisation des modes stricts dans la résolution, principes et implémentation"
Rapport de Recherche No 596, IMAG, Fevrier 1986.

[Oudot87] OUDOT O.
"Utilisation des modes directionnels dans la résolution"
Thése INPG Grenoble, Novembre 1987.

[Paterson78] PATERSON M.S. and WEGMAN M.N.
"Linear unification"
J. Comp. syst. Sc. 16, 2 April 1978 pp 158.

[Pereira81] PEREIRA L.M. and PORTO A.
"Selective backtracking "
U.N.L./F.C.T. 11/81, University Norva de Lisboa, Portugal.

[Pereira83] PEREIRA L.M.
"An interprete of logic programs using selective backtracking"
Repports 2-3, Dept of A.I., University of Lisbone Portugal, 1983.

[Pereira84] PEREIRA L.M.
"C-PROLOG user's manual"
Version 1.5, University Edinburgh, Feb 1984.

[Poole85] POOLE D. and GOEBEL R.
"On eliminating loops in PROLOG"
ACM Sigplan Notices, Volume 20. N° 8, Aout 1985, pp 38 - 40.

[Quéré68] QUERE A. et PAIR C.
"Definitions et études des bilangages"
Information and Control 13, 1968, p 565-593.

[Reddy84] REDDY U.S.
"Transformation of logic programs into functionnal programs"
Int. Symp. on Logic Prog. Atlantic city, New-Jersey Fev 1984.
pp 187 - 196.

[Robinson65] ROBINSON J.A.
"A Machine-oriented logic based on the resolution principle"
Journal of the ACM, volume 12, N^{o} 1, December 1965.
pp 23 - 41.

[Robinson79] ROBINSON J.A.
"Logic form and function"
Edinburgh University Press and Elsevier North-Holland, 1979.

[Roussel75] ROUSSEL Ph.
"PROLOG, manuel de référence et d'utilisation"
Groupe d'I.A., Université de Marseille Luminy, 1975.

[Sawamura84] SAWAMURA H. and TAKESHIMA T.
"Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to PROLOG optimisations"
ICOT, TR 88, Tokyo Japan, Octobre 1984.

[Sawamura85] SAWAMURA H., TAKESHIMA T. and KATO A..
"Source-level optimisation techniques for PROLOG"
ICOT, TM 091, Tokyo, Japan, January 1985.

[Sawamura86] SAWAMURA H.
"Undecidability of determinacy, two decidables cases of determinacy and their applications to source-to-source transformations of PROLOG programs"
ICOT, TR 216, Tokyo Japan, Novembre 1986.

[Shapiro86] SHAPIRO E.Y. and STERLING L.
"The art of PROLOG"
MIT Press Series in Logic Programming, E. Shapiro editor, 1986.

[Shoenfield67] SHOENFIELD J.
"Mathematical logic"
Thèse, Addison-Wesley, 1967.

[Simonet81] SIMONET M.
"W.Grammaires et logique du premier ordre pour la définition et l'implantation des langages"
Thèse d'état USMG Grenoble, Juillet 1981.

[Solomon78] SOLOMON M.
"Type definitions with parameters"
Conf. Record of 5th Annual ACM Sympo. on Principles of Prog. Lang., Jan 1978, pp 31 - 38.

[Stepenkova84] STEPANKOVA O. and STEPANEK P.
"Transformation of logic programs"
Journal of Logic Programming 4, 1984, p 305 - 318.

[Takenchi85] TAKENCHI A. and FURUKAWA K.
"Partial evaluation of PROLOG programs and its applications to meta-programming"
T.R. 126, ICOT, Tokyo Japan 1985.

[Tamaki82] TAMAKI H.
"A Transformation system for logic programms which preserves equivalence"
T.R. 18, ICOT, Tokyo Japan November 1982.

[Tarjan72] TARJAN R.E.
"Depht first search an linear graph algorithms"
SIAM J. Computing, 1:2 1972 pp 146 - 160.

[Turbo86] TURBO-PROLOG
"the natural language of A.I."
Manuel d'utilisation de TURBO-PROLOG, 1986.

[Vanemdem82] Van Emdem K. and M.H.
"Contribution of theory of logic programming"
J. ACM, Volume 29, N^{o} 3, 1982, p 841.

[Vancaneghem84] VAN CANEGHEM M.
"L'Anatomie de PROLOG II"
Thèse de Doctorat d'état, Université de Marseille II, France
Octobre 1984.

[Vancaneghem86] VAN CANEGHEM M.
"Anatomie de PROLOG"
Inter-Edition, 1986.

[Vanroy84] VAN ROY P.
"A PROLOG compiler for the PLM"
Report No UCB/CSD 84/203, University of California USA
November 1984.

[Vasak83] VASAK T. and POTTER J.
"Metalogical control for logic programs"
Journal for Logic Programming N^{o} 3, 1983, p 203.

[Vataja84] VATAJA P. and UKKONEN E.
"Finding temporary terms in PROLOG programs"
Proc. of the Inter. Conf. on Fifth Generation Comp. Syst., 1984.
Eds ICOT, 1984, pp 275 - 282.

[Warren77] WARREN D.H.D.
"Implementing PROLOG"
Repports 39-40, Dept of A.I., University of Edinburgh, 1977.

[Warren83] WARREN D.H.D.
"An abstract PROLOG instructions set"
Technical note N^{o} 309, SRI International, Menlo Park
October 1983.

[Wirth73] WIRTH N.
" The programming language Pascal (revised report)"
R.R. Eidgenossische Techniosche Hochschule Zurich, Jully 1973.

[Zobel87] ZOBEL J.
"Derivation of polymorphic types for PROLOG programs"
J. Proceedings of the Fourth Inter. Conf. on Logic Prog.
Melbourne Australia, May 1987, pp 817 - 838.

département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de

Monsieur L. TRILLING, Professeur

Monsieur D. HERMAN, Professeur

Monsieur AZZOUNE Hamid

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "INFORMATIQUE".

Fait à Grenoble, le 3 Janvier 1989

Pour le Président de l'I.N.P.G.
et par délégation,
le Vice-Président
P. VENNEREAU