MASTER EN SCIENCES INFORMATIQUES

Interprétation abstraite des programmes Prolog

Gerard, Jacques

Award date:
1988

Awarding institution:
Universite de Namur

Interprétation abstraite des programmes PROLOG.

Jacques GERARD.

Septembre 1988.

Remerciements.

C'est avec grand plaisir que je remercie Monsieur Baudouin Le Charlier pour son aide permanente au cours de ce mémoire. L'algorithme principal ainsi que les meilleures idées se trouvant dans ce texte sont les siennes. Je le remercie aussi pour l'amabilité avec laquelle il m'a reçu lors des multiples entretiens que nous avons eus ensembles.

I. Introduction.

L'étude des propriétés statiques des programmes écrits en langage PROLOG commence d'intéresser beaucoup de chercheurs [1][2][3][4][5][6]. Le but de cette étude est souvent une optimisation de la compilation des programmes[1][2][6]. On peut aussi, dans une certaine mesure, développer quelques-unes des idées contenues dans l'étude des propriétés statiques des programmes pour vérifier la correction de ceux-ci[5].

Tout en sachant qu'il est impossible de concevoir un algorithme capable de démontrer la correction totale des autres algorithmes, nous pouvons malgré tout opérer certaines vérifications.

En connaissant le mode de fonctionnement du moteur d'inférence PROLOG, nous pourrons construire un algorithme (conçu par Monsieur B. Le Charlier[7]) plus performant que certains autres trouvés dans la littérature. Il analysera les clauses dans le même ordre que le moteur d'inférence.

Pour une clause et une substitution données, nous appliquerons un algorithme d'étude qui "exécute" la clause dans le même ordre que l'interpréteur PROLOG. La substitution s'en trouvera modifiée à chaque étude d'un sous-but et la substitution résultat sera utilisée comme argument pour l'étude du sous-but suivant.

Après cette analyse, nous prendrons comme résultat la substitution la plus générale qui sera "l'union" des résultats obtenus pour chacune des clauses dont la tête s'unifie avec le but à étudier. En pratiquant de la sorte, nous perdrons peut-être une partie de l'information obtenue pour une clause

particulière, mais nous assurerons la correction de notre algorithme.

Pour notre étude, nous nous intéresserons plus particulièrement à l'inférence de mode, qui en PROLOG est d'une importance certaine. Nous essaierons de concevoir un algorithme correct [7] d'étude de l'inférence de mode et nous en donnerons sa transcription en langage PROLOG. Pour assurer la correction de notre algorithme, nous choisirons un ensemble fini de substitutions "abstraites". Nous choisirons néanmoins cet ensemble suffisamment significatif pour rendre notre étude intéressante.

Nous faciliterons notre travail en "normalisant" au préalable les programmes que nous étudierons. En effet, avec une forme "normale" de clause, le nombre de cas à interpréter pouvant survenir sera limité.

II. Interprétation abstraite et PROLOG.

A. Le PROLOG [8].

Les constructions de base en programmation logique, termes et formulations, sont héritées de la logique. Il y a trois formulations de base: les faits, les règles et les questions et il n'y a qu'une seule structure de données: le terme logique.

L'élément de formulation le plus simple que nous pouvons rencontrer est le fait. Les faits sont le moyen d'affirmer qu'il existe une certaine relation entre des objets. Par exemple:

father(abraham,isaac).

Ce fait dit que Abraham est le père de Isaac. On appelle aussi cette relation un prédicat.

La seconde formulation existant dans les programmes logiques est la question. Une question demande si il existe une certaine relation entre des objets. Par exemple:

father(abraham,isaac)?

Cette question demande s'il y a une relation du type "father" entre abraham et isaac. Si nous avions le fait précédemment défini, la réponse à cette question est "yes".

Syntaxiquement, la question a la même structure que le fait, mais le contexte permet de distinguer les deux formulations. Quand il y a une confusion possible, un point indique le fait tandis qu'un point d'interrogation indique la question.

Répondre à une question à l'aide d'un programme, c'est déterminer si la question est une conséquence logique du programme.

Avant de définir la troisième formulation possible, voici quelques définitions intermédiaires.

Une variable logique est donnée pour un individu non spécifié et est utilisée en accord avec celui-ci. Par exemple:

father(abraham,X).

On dira que X est une variable logique (ou variable).

Une substitution est un ensemble fini (parfois vide) de paires de la forme Xi=ti, où Xi est une variable et ti un terme, et Xi‹›Xj pour tout i‹›j, et Xi ne peut pas être présent dans tj pour tout i›j.

Un exemple de substitution pour le fait que nous avons donné est la paire {X=isaac}. Les substitutions peuvent être appliquées aux termes. Le résultat de l'application d'une substitution Θ à un terme A notée AΘ, est le terme obtenu en remplaçant toutes les occurrences de X par t pour toute paire X=t de Θ.

Le résultat de l'application de {X=isaac} au terme father(abraham,X) est le terme father(abraham,isaac).

A est une instance de B si il existe une substitution Θ telle que A = BΘ.

Par définition, nous dirons que le but father(abraham,isaac) est une instance de father(abraham,X).

Les questions contenant des variables sont appelées questions existantielles. C'est la première forme de règle de déduction que nous pourrons rencontrer. Par exemple, la question father(abraham,X)? signifie: "existe-t-il un X tel que abraham est le père de X?".

La seconde règle de déduction est une généralisation: une question existantielle P est une conséquence logique d'une de ses instantiations, PΘ, pour toute substitution Θ. Le fait father(abraham,isaac) implique qu'il existe un X tel que father(abraham,X) est vrai, c'est-à-dire X=isaac.

Les faits universels sont des faits contenant des variables. Par exemple, "likes(X,pomegranates)" signifie que toute occurrence de X aime les grenades. Par exemple, on peut déduire de ce fait likes(abraham,pomegranates). C'est la troisième forme de déduction appelée **instantiation**. On peut l'exprimer de la façon suivante:

A partir d'une formulation quantifiée universellement P, on déduit une instance ΘP de P, pour toute substitution Θ.

Après ces quelques définitions, nous pouvons définir la dernière formulation possible en programmation logique.

Une règle est une formulation de la forme A<--B1,..,Bn. où n>=0. A est la tête de la règle et les Bi forment le corps de la règle. Et A et l'ensemble des Bi sont des buts. Règles, faits et questions ainsi définies sont encore appelées clauses de Horn ou clause en abrégé. Le fait est un cas spécial de règle où n=0.

Nous définirons enfin la signification d'un programme logique.

La signification d'un programme logique P, M(P), est l'ensemble déductible de P des buts, unités dont les variables sont de base.

Pour terminer, voici une définition d'un terme.

Un terme est une constante, une variable ou un terme composé. Les constantes représentes des individus particuliers tels que des entiers ou des atomes. Les variables représentent un individu simple mais non spécifié. Les termes composés comprennent un foncteur (appelé foncteur principal) et une séquence d'un ou plusieurs termes appelés arguments. Un foncteur est caractérisé par son nom et son arité (ou nombre d'arguments). Les constantes sont considérées comme des foncteurs d'arité 0. Syntaxiquement, les termes composés sont de la forme f(t1,..,tn) où f est le nom du foncteur, n est l'arité et les ti sont les arguments. On le notera f/n.

Un terme sera dit de base s'il ne contient aucune variable; dans le cas contraire, il sera dit non de base. Les buts sont des atomes ou des termes composés et sont généralement non de base.

Nous pouvons maintenant définir un programme PROLOG, c'est un ensemble de clauses réparties en groupes dont chacune des têtes de clauses d'un groupe a un foncteur et un arité identiques à ceux de tout le groupe.

Ce qui déclenche l'exécution d'un programme PROLOG, c'est le fait de poser une question. Cette question a évidemment la forme d'un but. L'exécution consistera à rechercher une clause dont la tête s'unifie avec ce but. Si plusieurs clauses sont dans ce cas, l'interpréteur choisira la première rencontrée dans l'ordre de lecture dans la base de connaissance qui est composée de toutes les clauses mises à la disposition de l'interpréteur. Si la clause rencontrée est un fait, il y a unification, et, soit le but est vérifié, soit l'interpréteur regarde si une autre clause correspond. Si il n'en existe pas, le but échoue et si il en existe une, l'interpréteur essaie cette nouvelle clause. Dans le cas où la clause n'est pas un fait, la tête de clause réussit si tous les sous-buts contenus dans le corps de la clause réussissent. Ceux-ci sont examinés dans l'ordre d'apparition dans le corps de la clause, c'est-à-dire de gauche à droite. Dans ce cas, chaque unification rendue effective par un sous-but, le reste pour les sous-buts suivants.

Dans le cas d'un échec de l'évaluation d'un sous-but quelconque, le mécanisme du backtracking [8] permet de revoir des choix effectués pour certaines variables et donc le cas échéant, de trouver une solution à la question posée. Le dernier choix est revu, si il existe une autre possibilité de choix, elle est prise en compte et il y a réessai des sous-buts suivants avec ce nouveau choix. Si ce dernier choix était unique, on remonte au choix précédent tant que c'est le cas et tant que c'est possible. Quand ce n'est plus possible, il y a échec.

B. Principe de l'interprétation abstraite.

Soit un but p(X1,..,Xn) et une substitution Θ sur p/n. Après application du but p/n, nous aurons une substitution Θ' sur les variables {X1,..,Xn}. L'interprétation abstraite consiste, en connaissant certaines propriétés sur les variables dans Θ, à déduire des propriétés que doivent avoir les variables dans Θ'. C'est-à-dire, qu'en connaissant certaines propriétés sur les variables avant application du but, nous pourrons en déduire certaines propriétés sur les variables après application du but sans réellement "exécuter" ce but.

L'ensemble des substitutions vérifiant certaines propriétés avant interprétation du but, sera la substitution abstraite et sera noté β. Après interprétation, l'ensemble des substitutions résultâtes sera la substitution abstraite résultante et sera notée β'.

III. Normalisation.

A. Définitions.

Voici la définition, selon la notation BNF, du langage PROLOG sur lequel notre analyse portera.

‹Clause› ::= ‹Fait›‹.› / ‹Tête›‹:-›‹Corps›‹.›

‹Corps› ::= ‹Terme› { ‹,› ‹Corps› }

‹Tête› ::= ‹Fait›

‹Terme› ::= ‹Fait› / ‹Expression› / ‹Unification›

‹Fait› ::= ‹Symbole› { ‹(›‹List-arg›‹)› }

‹Symbole› ::= ‹Minuscule› { ‹Mot› }

‹Minuscule› ::= ‹a› .. ‹z›

‹Mot› ::= ‹MMC› { ‹Mot› }

‹MMC› ::= ‹Minuscule› / ‹Majuscule› / ‹Chiffre›

‹Majuscule› ::= ‹_› / ‹A› .. ‹Z›

‹Chiffre› ::= ‹0› .. ‹9›

‹List-arg› ::= ‹Argument› { ‹,›‹List-arg› }

‹Argument› ::= ‹Symbole› / ‹Variable› / ‹Nombre›

‹Nombre› ::= ‹Entier-signé›

‹Entier-signé› ::= { ‹Signe› } ‹Entier›

‹Signe› ::= ‹+› / ‹-›

‹Entier› ::= ‹Chiffre› { ‹Entier› }

‹Variable› ::= ‹Majuscule› { ‹Mot› }

‹Expression› ::= ‹Variable›‹is›‹Expr›

‹Expr› ::= ‹Argument› / ‹Expr›‹Opérateur›‹Expr› / ‹(›‹Expr›‹)›

‹Opérateur› ::= ‹+› / ‹-› / ‹*› / ‹/›

‹Unification› ::= ‹Terme›‹=›‹Terme›

Après cette définition, voici deux notions supplémentaires pour la compréhension de la suite.

Variable: Elément appartenant à l'ensemble des variables normalisées (voir normalisation).

Substitution abstraite: C'est la représentation par un couple de l'ensemble des substitutions d'une part, et l'ensemble fixé des variables d'autre part . On la représente par β où

$\beta \equiv (\{xi1,..,xin\},\Phi)$,

Φ est l'ensemble des substitutions Θ,

$\Theta = \{xi1<--t1,..,xin<--tn\}$.

B. <u>Transformations.</u>

Le but de la normalisation des programmes PROLOG avant leur interprétation, est de faciliter notre tâche lorsque nous devrons réaliser effectivement cette interprétation. En effet, après normalisation, il n'existera que peu de cas de figure différents et toutes les clauses auront une tête de clause identique. Cela facilitera beaucoup l'unification d'un but avec cette tête.

Par exemple, si nous avons la clause (1), sa normalisation sera la clause (2).

```
ami(albert,X) :- voisin(albert,X),sage(X). (1)
ami(X1,X2) :- X1=albert,X3=X1,voisin(X3,X2),sage(X2). (2)
```

Soit un ensemble infini de variables {Xi} que nous dirons normalisées, supposons en outre que le programme à normaliser ne contienne aucune des ces variables Xi.

Un programme écrit dans notre langage PROLOG est normalisé si toutes ses clauses sont normalisées.

Une clause est normalisée si sa tête est normalisée et si son corps est normalisé.

Une tête de clause est normalisée si elle est de la forme p(X1,..,Xn) où, les Xi représentent des variables normalisées distinctes.

Un corps de clause est normalisé si tout sous-but du corps de clause est normalisé.

D'après notre définition du langage, il y a trois types de sous-but:(1) les termes, (2) les expressions, (3) et les unifications.

Un terme est normalisé si il est de la forme p(X1,..,Xn) où les Xi sont des variables normalisées non nécessairement distinctes.

Une expression est normalisée si elle est de la forme "Xi is p(Xj1,..,Xjn)" où Xi n'appartient pas à l'ensemble de variables {Xj1,..,Xjn} et où les variables Xi et {Xj1,..,Xjn} sont des variables normalisées.

Une unification est normalisée si elle est de la forme "Xi = p(Xj1,..,Xjn)" où Xi n'appartient pas à l'ensemble de variables {Xj1,..,Xjn} et où les variables Xi et {Xj1,..,Xjn} sont des variables normalisées.

Il existe deux transformations de base que nous utiliserons pour la normalisation d'une clause. On pourra également montrer

que ces transformations conservent un sens rigoureusement identique à la clause ainsi transformée.

(1) p(t1,..,tn):-B. ==› p(X1,..,Xn):-X1=t1,..,Xn=tn,B.

(2) A:-B,p(t1,..,tn),C. ==› A:-B,X1=t1,..,Xn=tn,p(X1,..,Xn),C.

D'un point de vue pratique, nous représenterons les variables normalisées par des termes de la forme v(i). Ceci nous permettra d'éviter des problèmes d'unification impromptues et nous permettra, d'un point de vue technique, de comparer des variables;et nous allons décomposer le problème de la manière suivante.

Soit C la clause à normaliser. Alors, normtete(C) = (C',n) sera une fonction permettant de passer d'une clause non normalisée à une clause dont le tête est normalisée.
Si C est de la forme

p(t1,..,tn) :- B.

avec p d'arité n, alors C' sera

p(v(1),..,v(n)) :- v(1)=t1,..,v(n)=tn,B'.

où B' est B tel que toute occurrence de ti est remplacée par v(i) si ti est une variable (transformation (1)).

Soit C une clause dont la tête de clause d'arité n est normalisée. Alors, la fonction "normc(C,n) = C' où C' est une clause normalisée", permettra de conclure la normalisation d'une clause.
Si C est de la forme

```
p(v(1),..,v(n)) :- B.
```

alors C' sera

```
p(v(1),..,v(n)) :- normcorps(B,n).
```

Soit B est un ensemble de sous-buts, alors la fonction normcorps(B,n)= B' où B' est un ensemble de sous-buts normalisés. Si B est de la forme

```
(q(t1,..,tn),B"),
```

alors B' sera

```
(normterme(q(t1,..,tn),n),normcorps(B",m)).
```

où m indiquera la première variable libre de l'ensemble des variables normalisées après application de la fonction normterme.

Soit B un terme et les n variables normalisées existantes, alors normterme(B,n) = B' où B' est l'ensemble des sous-buts résultants de la normalisation de B et contenant le terme B normalisé et l'ensemble des unifications dues à la normalisation de B. Un terme est normalisé si il est une variable normalisée (v(i)), ou si il est une unification (v(i)=f(v(i1),..,v(in))) où v(ij) sont tous différents de v(i), ou si le terme est de la forme f(v(i1),..,v(in)).

C. Spécifications de la procédure de normalisation.

Nous savons à l'avance que toute les procédures que nous allons définir sont conçues pour n'être utilisées qu'une seule fois et qu'elles sont par ailleurs presque toutes déterministes. De plus elle seront souvent utilisées, seulement dans une seule

directionalité. C'est pourquoi nous pouvons donner les conventions suivantes, que nous utiliserons pour définir les spécifications de toutes les procédures.

Toute procédure sera définie par une précondition appelée pré et une postcondition appelée post.

La précondition donnera toutes les variables (ainsi que leur domaine de valeurs le cas écheant) qui doivent être de base avant application de la procédure. Les variables citées dans la précondition seront considérées comme des arguments de la procédure et devrons nécessairement être des termes de base. On tiendra pour des variables-résultats celles qui ne seront pas citées dans cette précondition.

La post condition donnera les propriétés que devront vérifier les variables-résultats après application de la procédure.

Dans le cas d'une procédure conçue pour avoir la double directionalité, nous donnerons deux spécifications pré-post, une pour chaque directionalité.

norme(SF,L).

Pré: SF un fichier contenant un programme PROLOG à normaliser.

Post: L est une liste qui contient toutes les clauses normalisées du programme qui se trouvait dans le fichier SF.

Pour réaliser cela, cette procédure doit ouvrir le fichier SF en lecture. La procédure 'normfile/1' se charge de la normalisation proprement dite. Après cette normalisation, il faut fermer le fichier SF.

normfile(L).

Pré: Un fichier est ouvert en lecture et contient un programme PROLOG.

Post: L est une liste qui contient toutes les clauses normalisées du programme contenu dans le fichier ouvert en lecture.

Cette procédure lit le fichier source clause par clause, et applique à chacune d'elles la procédure 'normclause/2' qui transforme une clause T en une clause normalisée R et l'écrit dans le fichier destination. Elle se termine quand toutes les clauses du fichier source ont été lues.

Avant de donner la spécification de la procédure normclause, nous allons définir quelques petites procédures utiles par la suite.

append(L1,L2,L3).

Pré: L1 et L2 deux listes.

Post: L3 est la concaténation de L1 et L2.

dans((X,T),L).

Pré: L une liste de couples de termes de base.

Post: si X est le premier élément d'un couple contenu dans la liste L, T est identifié au second élément du couple. La procédure échoue si ce n'est pas le cas.

in(X,L).

Pré: L une liste d'éléments.

Post: La procédure réussit si X ∈ L, sinon elle échoue.

retr(X,L1,L2).

Pré: Soit X un élément et une liste L1 et X ∈ L1.

Post: La liste L2 est la liste L1 où l'élément X a été enlevé.

constr(B,L).

Pré: L est de base et est une liste de sous-buts.

Post: B est l'ensemble des sous-buts contenus dans L.

Pré: B est de base et constitué d'un ensemble de sous-buts.

Post: L est la liste des sous-buts contenus dans B.

reconstr(P,L,Q).

Pré: P est un but et L une liste de sous-buts.

Post: Si L est une liste vide, alors Q a pour valeur P, sinon Q est unifié à (P:-B) où B est l'ensemble des sous-buts correspondants à la liste de sous-buts L.

Grâce à la procédure 'constr/2', il sera facile de transformer une liste de sous-buts en un ensemble de sous-buts.

Nous pouvons maintenant entrer dans le vif du sujet et donner une spécification pour toutes les procédures de normalisation proprement dites.

normclause(C,T).

Pré: C une clause à normaliser.

Post: Si C est le signal de fin de fichier 'end_of_file', alors la procédure échoue. Sinon, T est la clause C normalisée.

Cette normalisation sera réalisée en appliquant les procédures de normalisation de la tête de clause 'normtete/5' et le cas échéant, la procédure de normalisation de corps de clause 'normcorps/4'. Ensuite, il suffira de reconstruire la clause normalisée grâce à la procédure 'reconstr/3'.

normtete(H,P,LB,LV,M).

Pré: H une tête de clause.

Post: P est la tête de clause H normalisée, LB est la liste des unifications normalisées dues à la normalisation de H, LV est la liste des variables utilisées dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg après normalisation.

Cette procédure revient à utiliser la procédure 'normtete/7'

normtete(A,AN,LB,LVI,LVF,M,N).

Pré: A est une liste de termes, LVI est une liste de variables dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: AN est la liste des variables normalisées v(I) contenant autant de membres que la liste A ne contient de termes et dans cette liste AN, les variables v(I) sont successives et commencent à v(1). LB est la liste des unifications dues à la

normalisation des termes, LVF est la liste des variables utilisées dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg après normalisation.

normlblt(LB,LBN,LV,LVN,N,M).

Pré: LB une liste d'unifications, LV est une liste des variables utilisées dont il existe une variable normalisée correspondante et N désigne la première variable libre de l'ensemble Vg.

Post: LBN est la liste des unifications LB normalisées, LVN est l'ensemble LV augmenté des nouvelles correspondances et M désigne la première variable libre de l'ensemble Vg après application de la procédure.

normcorps(LC,LCN,LV,M).

Pré: LC est une liste de sous-buts, LV est la liste des variables dont il existe un variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: LCN est la liste des sous-buts normalisés des sous-buts contenus dans LC.

normcorps(LC,LCN,LV,M,N).

Pré: LC est une liste de sous-buts, LV est la liste des variables dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: LCN est la liste des sous-buts normalisés des sous-buts contenus dans LC et N désigne la première variable libre de l'ensemble Vg après application de la procédure.

normterme(X,Y,LB,LVI,LVF,M,N).

Pré: X est un terme, LVI est la liste des variables dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: Y est le terme X normalisé, LB est la liste des unifications engendrées par la normalisation du terme X et N désigne la première variable libre de l'ensemble Vg après application de la procédure.

normbltin((X=Y),Z,LB,LVI,LVF,M,N).

Pré: (X=Y) est une unification, LVI est la liste des variables dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: Z est l'unification (X=Y) normalisée, LB est la liste des unifications engendrées par la normalisation de l'unification (X=Y) et N désigne la première variable libre de l'ensemble Vg après application de la procédure.

nbltin(X,Y,Z,LB,LVI,LVF,M,N).

Pré: X est une variable, Y est un terme, LVI est la liste des variables dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: Z est l'unification (X=Y) normalisée, LB est la liste des unifications engendrées par la normalisation du terme Y et N désigne la première variable libre de l'ensemble Vg après application de la procédure.

ntvar(X,v(I),LVI,LVF,M,N).

Pré: X est une variable, LVI est la liste des variables dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: Si il existe une variable normalisée correspondante à X, I est unifié à la valeur du second membre du couple correspondant dans la liste LVI. Dans ce cas LVF = LVI et N = M. Sinon, I prend la valeur M, N vaut M+1 et LVF = [(X,M)¦LVI].

nter(LA,LAN,LB,LVI,LVF,M,N).

Pré: LA est une liste de termes, LVI est la liste des variables dont il existe une variable normalisée correspondante et M désigne la première variable libre de l'ensemble Vg.

Post: LAN est la liste des termes normalisés de LA, LB est la liste des unifications engendrées par cette normalisation,LVF est l'ensemble LVI augmenté des couples (variable,I) pour toute nouvelle variable rencontrée lors de la normalisation des termes de la liste LA où I est la valeur attribuée à chacune de ces nouvelles variables comme première valeur de variable libre de Vg au moment de l'attribution. Toute attribution est unique. N désigne la première variable libre de l'ensemble Vg après application de la procédure.

IV. Interprétation abstraite des programmes normalisés.

A. Problème de la procédure.

1. Cadre général.

Supposons que l'on ait un but p(X1,..,Xn) et une substitution de départ Θ sur {X1,..,Xn}. On veut connaître l'ensemble des substitutions Φ après résolution du but p par application de la procédure dont la tête de clause (notée par la suite p/n) s'unifie avec notre but.
On notera ceci Θ p(X1,..,Xn) Φ.

Si Φinit est l'ensemble des substitutions Θ vérifiant certaines propriétés, on veut connaître l'ensemble des substitutions Φres correspondant. Pour ce faire, on appliquera:

Φinit p(X1,..,n) Φres.

On se donnera une classe βinit de substitutions abstraites de Φinit, à laquelle on va s'intéresser et on calculera βres, la meilleure substitution abstraite possible telle que Φres soit inclus dans βres. Nous chercherons évidemment la substitution βres la plus restrictive possible et donc, qui se rapprochera le plus de Φres.

2. Substitution de départ.

Au départ, supposons que l'on ait les clauses C1,..,Cm, dont la tête est le but p(X1,..,Xn). Toutes ces têtes de clauses s'unifieront avec notre but à étudier p(Xi1,..Xin). Il faut exécuter C1,C2,..,Cm. On peut leur associer les substitutions suivantes:

βinit C1 βres,

β2init C2 β2res,

...

βminit Cm βmres.

Le problème sera de déterminer ce que valent les βiinit et les βires. Dans un premier temps, nous pourrons déjà dire que tous les βiinit sont les mêmes. Notre premier sous-problème sera donc de passer de βinit à βiinit. Ceci sera réalisé par la fonction chvarg qui aura pour but de changer les variables de gauche d'une substitution β.

chvarg ({Xj1,..,Xjk},({Xl1,..,Xlk},Φ)) = ({Xj1,..,Xjk},Φ')

où

Φ' = {Θ' : Θ' = {Xj1‹--t1,..,Xjk‹--tk} et il existe Θ ∈ Φ tel que Θ = {Xl1‹--t1,..,Xlk‹--tk}}

Grâce à cette fonction, nous pourrons passer aisément de βinit à βiinit. Il suffira d'effectuer l'opération suivante:

βiinit = chvarg ({X1,..,Xn},βinit).

3. Utilisation des résultats.

Le second problème que l'on rencontre lors de l'application d'une procédure est de pouvoir déduire βres de tous les βires obtenus par application des clauses C1,..,Cm.

Pour résoudre ce problème, nous utiliserons la fonction suivante:

lub (β,β') = β U β'.

où β et β' sont définies sur le même domaine D de variables.

Cette fonction sera utilisable à condition que l'ensemble des variables de β soit identique à l'ensemble des variables de β'.

Nous pourrons aussi définir les cas limites d'utilisation de cette fonction:

lub (β) = β et lub () = ⊥.

⊥ représente l'ensemble vide de substitutions et nous pourrons écrire ⊥{Xi1,..,Xin} = ({Xi1,..,Xin},Ø).

Dans notre cas, si nous avons les substitutions résultats βi de l'application des différentes clauses dont la tête s'unifie avec le but que nous étudions, nous aurons la substitution résultat de la procédure donnée par:

β* = lub(β1res,β2res,..,βmres),

βres = chvarg ({Xi1,..,Xin},β*)

Etant donné notre définition de lub pour le cas de la substitution vide, nous pourrons même définir β^* de manière plus efficace pour la suite :

$$\beta^* = lub(\bot, \beta 1res, \beta 2res, .., \beta mres).$$

4. Cas de l'unification.

Dans ce cas, nous ne travaillerons pas de la même manière, nous associerons à cette unification une unification abstraite:

$$\beta res = fp(\beta init)$$

où fp sera la fonction associée à l'unification p.

Nous verrons plus tard que, pratiquement, nous pouvons relier cette notion à d'autres pour en faire un cas particulier d'un problème que nous aurons résolu par ailleurs.

B. Problème de la clause.

1. Cadre général.

Le problème de la clause peut être posé en ces termes. Soit une substitution de départ $\beta init$ et une clause C définie de la manière suivante:

C = p(X1,..,Xn) :- A1,..Am.

Quelle est alors la substitution βres correspondante à βinit après application de la clause C ?

Nous devrons résoudre tous les sous-buts les uns après les autres et utiliser chaque fois le résultat d'un sous-but comme argument du sous-but suivant de cette manière:

β0 A1 β1,..,βi-1 Ai βi,..,βm-1 Am βm.

Nous nous préoccuperons donc en premier lieu de trouver β0 en fonction de βinit. Ensuite, nous devrons pouvoir calculer βres à partir de βm et finalement, si nous pouvons calculer βi en fonction de βi-1, nous obtiendrons une solution à notre problème.

Remarque: Il se peut que le résultat d'une des opérations élémentaires que nous allons définir ne soit pas dans l'ensemble fini que nous avons choisi, mais de fait, nous nous servirons d'approximations, et dans ce cas, cela ne dérange pas. Nous essaierons d'avoir l'approximation la meilleure possible. Nous utiliserons les notions d'ensembles infinis pour développer ces fonctions pour des raisons évidentes de facilité de compréhension.

2. La substitution du premier sous-but.

Le premier problème est par conséquent de calculer β0 en fonction de βinit. En effet, un sous-but particulier peut utiliser des variables différentes de celles de la tête de la clause. Il faudra donc compléter la substitution en y ajoutant toutes ces variables itermédiaires. Nous utiliserons la fonction suivante:

ext1({X1,..,Xn,Xn+1,..,Xn+m},({X1,..,Xn},Φ) = Φ'

où

Φ' = { Θ' tq Θ' = {X1<--t1,..,Xn<--tn,Xn+1<--Y1,..,Xn+m<--Ym} et
il existe Θ ∈ Φ tq Θ = {X1<--t1,..,Xn<--tn} et
Y1,..,Ym ∈ Vd et
Y1,..,Ym ne figurent pas dans les termes t1,..,Tn et
Y1,..,Ym sont différents deux à deux.}

Si var(C) = l'ensemble des variables figurant dans C, alors nous pourrons dire que:

β0 = ext1 (var(C),βinit).

3. Trouver la substitution résultat.

Ayant la dernière substitution βm, il faudra la restreindre aux seules variables de la tête de la clause pour obtenir la substitution résultat βres.

Pour ce faire, nous définirons une nouvelle fonction qui aura pour but de "projeter" le résultat βm sur les variables de la tête de la clause pour obtenir la substitution souhaitée.

proj({Xi1,..,Xik},({X1,..,Xl},Φ)) = ({Xi1,..,Xik}, Φ')

où

Φ' = { Θ' tq Θ' = {Xi1<--ti1,..,Xik<--tik} et
il existe Θ ∈ Φ tq Θ = {X1<--t1,..,Xl<--tl}

Pour pouvoir appliquer cette fonction, il faudra évidemment que l'ensemble {Xi1,..,Xik} soit inclus dans {X1,..,Xl} c'est-à-dire que tout ij soit compris entre 1 et l.

En appliquant cette fonction à notre problème, nous aurons:

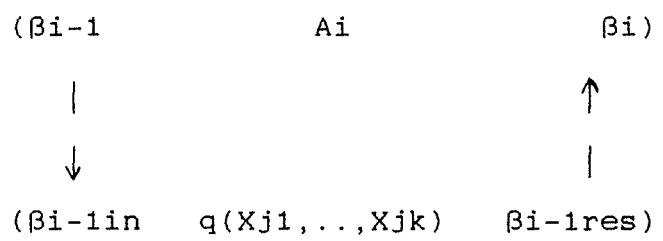
βres = proj ({X1,..,Xn},βm).

4. Substitutions transitoires.

Le dernier problème auquel nous serons confrontés lors de l'application d'une clause est de pouvoir donner la substitution utile pour l'application d'un sous-but et de pouvoir transmettre

l'information glanée par l'application du sous-but à la substitution abstraite résultat.

Nous pourrons formaliser ce problème de la manière suivante, en supposant que les Ai sont de la forme q(Xj1,..Xjk):



Il faudra donc pouvoir passer de βi-1 à βi-1in qui ne tient compte que des variables {Xj1,..,Xjk}. L'opération de "projection que nous avons déjà définie plus haut réalise parfaitement ceci:

βi-1in = proj ({Xj1,..,Xjk},βi-1).

Notre deuxième sous-problème est la déduction de βi à partir de βi-1 et de βi-1res. En effet, si nous ne voulons pas perdre toute l'information récoltée antérieurement, nous devons disséminer les nouvelles informations contenues dans βi-1res à l'intérieur de βi-1 pour obtenir βi. Cette opération complexe sera effectuée grâce à la fonction suivante:

ext(({X1,..,Xl},Φ),({Xj1,..,Xjk},Φ')) = ({X1,..,Xl},Φ") où

Φ" = { Θ" tq il existe Θ ∈ Φ et il existe Θ' ∈ Φ' tq
Θ = {X1<--t1,..,Xl<--tl} et Θ' = {Xj1<--tj1',..,Xjk<--tjk'}
et σ = mgu ((tj1,tj1'),..,(tjk,tjk')) et Θ" = Θσ }.

Cela revient donc à appliquer σ à la substitution Θ. En appliquant cette fonction dans notre cas, nous obtenons:

$\beta i = ext (\beta i-1, \beta i-1res)$.

C. Algorithme principal.

1. Principe.

Si nous appliquons brutalement le principe ébauché lors de la définition des fonctions de base, nous allons construire un arbre de résolution du problème de plus en plus grand et notre algorithme ne se terminera pas.

Pour assurer cette terminaison, nous allons utiliser l'astuce suivante. Si on rencontre un $\beta i-1in$, on supposera que l'on a le $\beta i-1res$ résolu correspondant (que l'on initialisera à $\perp$); si en descendant dans l'arbre nous rencontrons à nouveau une substitution $\beta i-1in$ équivalente, nous trouverons une solution pour la substitution $\beta i-1res$ initiale que nous supposerons meilleure. Nous remplacerons l'ancienne solution par celle-ci et on recommence le calcul, jusqu'à ce que la nouvelle solution n'évolue plus. Nous considérerons alors avoir trouvé la solution au problème.

Nous devrons construire deux algorithmes pour résoudre notre problème. Le premier réalisera l'interprétation abstraite d'un sous-but et le second l'interprétation abstraite d'une clause.

2. Algorithmes proprement dits.

a. Interprétation abstraite d'un sous-but.

Considérons l'ensemble de triplets déjà résolus (β,q/n,β') où β représente la substitution de départ, q/n le but à étudier et β', la substitution-résultat. Appelons cet ensemble ETA (ensemble des triplets abstraits). Nous pouvons donner l'algorithme IASB de résolution des sous-buts abstraits.

ALGORITHME I.

IASB (in: β,p/n ; out: β' ; var: ETA).

Cas 1: p/n est une unification.

==> β' := fp(β);

où fp(β) est la fonction associée à p/n

stop.

Cas 2: p/n n'est pas une unification.

==> Si il existe un triplet (β,p/n,β*) ∈ ETA alors,

β' := β*;

stop.

==> Sinon

a)ETA := ETA U { (β,p/n,inst(β)) };

b)β' := ⊥;

c)Pour C1,..,Cp les clauses dont la tête s'unifie à p/n faire:

|IAC(β,Ci,β*,ETA);

|β' := lub(β',β*);

c)Soit β* tq (β,p/n,β*) ∈ ETA,

Si (β' = β*) alors,

stop.

sinon

ETA := (ETA \ {(β,p/n,β*)}) U {(β,p/n,β')};

ETA := adj((β,p/n,β*),ETA);

aller en b);

==› ETA := ETA \ {(β,p/n,β')};

Dans cet algorithme se trouvent deux fonctions dont nous n'avons pas encore parlé. La première est la fonction inst(β) qui sert à initialiser la substitution de départ que l'on introduit dans ETA. La seconde est nécessaire pour maintenir la cohérence de notre ensemble ETA. Nous verrons plus tard un exemple qui montre cette utilité.

b. Interprétation abstraite d'une clause.

Supposons que toute clause est de la forme:

p(X1,..,Xn) :- A1,..,Am.

où chaque Ai est de la forme:

qi(Xi1,..,Xili).

Nous pourrons alors pour l'interprétation abstraite d'une clause écrire l'algorithme suivant:

ALGORITHME_II.

IAC (in: β,C ; out: β' ; var: ETA).

1) βext := ext1(var(C),β);
2) Pour i:=1 jusque m,
   faire:
   | βrestr := proj({Xi1,..,Xili},βext);
   | βrestr := chvarg({X1,..,Xli},βrestr);
   | IASB (βrestr,qi/li,βsucc,ETA);
   | βsucc := chvarg({Xi1,..,Xili},βsucc);
   | βext := ext(βext,βsucc);
3) β' := proj({X1,..,Xn},βext);
4) Stop.

3. Interprétation_abstraite_normale.

Pour une procédure p/n à étudier, nous pourrons utiliser l'algorithme IASB qui nous donnera la substitution-résultat pourvu qu'on lui fournisse la substitution bien choisie comme

argument (ce choix aliénera la sémantique de la substitution-résultat).

Pour ce faire, nous donnerons les arguments suivants qui forment une utilisation normale de notre algorithme:

ETA := Ø

IASB(βin,p/n,βout,ETA)

Après exécution, βout sera la substitution recherchée. Cet algorithme est général et fonctionne quelle que soit la forme des substitutions abstraites qu'on lui donne en entrée. L'interprétation que l'on pourra en tirer est évidemment liée à celle-ci.

Dans notre cas, et comme nous l'avons souligné précédemment, nous nous intéresserons plus particulièrement à l'étude des modes des variables. Dans ce but, nous devrons particulariser toutes les fonctions intermédiaires à cette étude de mode tandis que l'algorithme principal, lui, restera le même quelle que soit l'interprétation que nous pourrions donner.

V. Application à l'étude des modes.

A. Principes appliqués.

Soit β un triplet défini de la manière suivante sur un ensemble de variables {X1,..,Xn}:

β = (δ , Sv , Ps). où

δ sera l'ensemble des couples (Xi,Mi) où Mi est défini pour toutes les variables,

Sv sera une partition de l'ensemble des variables,

Ps sera une partition d'une partie de l'ensemble des variables.

Toutes les fonctions définies aux points B et C du chapitre III vont être spécialement adaptées à l'étude des modes. Nous écrirons toute substitution abstraite β = (δ,Sv,Ps) qui se composera comme suit:

δ sera composé des couples (variable,mode), la partition de Sv sera définie par les unifications respectives rencontrées au cours de l'étude (une partition se composera donc de variables qui ont nécessairement le même mode), et la partition de Ps se fera sur l'ensemble des variables {X1,.;,Xn} qui ne sont pas de base (non g) sera définie par définie par les relations rencontrées entre les différentes variables (une partition se composera de variables qui ont d'autres variables en commun, qui sont liées fonctionnellement entre elles).

Nous aurons par exemple pour un ensemble de variables {X1,X2,X3,X4} une substitution β de la forme:

$$\beta = (\ \{(X1,g),(X2,f),(X3,f),(X4,a)\}\ ,\ \{\{X1\},\{X2,X3\},\{X4\}\},\ \{\{X2,X4\},\{X3\}\}\ )$$

Nous écrirons aussi Xi ∈ Sv(Xj) (ou Xi ∈ Ps(Xj)) pour signifier que Xi est dans le même sous-ensemble de la partition Sv (ou Ps) que Xj.

Les différents modes auxquels considérés seront:

g : (ground) pour une variable qui a pris une valeur,

f : (free) pour une variable libre,

a : (any) pour toute variable qui pourra être quelconque.

Pour nos opérations sur les partitions, nous définirons quelques opérateurs:

P « P' ssi pour tout S ∈ P,

il existe S' ∈ P' : S est inclus dans S'.

On dira que la partition P est plus fine que la partition P'.

$P \sqcap P' = \{\ S \cap S' : S \in P \text{ et } S' \in P' \text{ et } S \cap S' \neq \emptyset\ \}$.

On dira que la partition résultante éclate la partition P et la partition P'.

$\beta \uplus \beta' \equiv \sqcap \{\ P'' : P \ll P'' \text{ et } P' \ll P''\}$

On dira que la partition résultante regroupe la partition P et la partition P'.

On pourra aussi en déduire les propriétés suivante:

$P \sqcap P' \ll P \ , \ P' \ll P \sqcup P'$

pour tout P" : P" « P' et P" « P' ==› P" « $P \sqcap P'$.

B. Les_fonctions_appliquées_aux_modes.

1. chvarg

chvarg ({Xj1,..,Xjn} , {Xs1,..,Xsn} , (δ,Sv,Ps)) = (δ',Sv',Ps')

où δ' = {(Xs1,M'1),..,(Xdk,M'k)} et pour tout i : Mi = Mi'.
Sv' est tel que pour tout Xjk ∈ Sv(Xji), Xlk ∈ Sv'(Xli).
Ps' est tel que pour tout Xjk ∈ Ps(Xji), Xlk ∈ Ps'(Xli).

2. lub

lub((δ,Sv,Ps) , (δ',Sv',Ps')) = (δ",Sv",Ps")

où δ" = {(Xi,Mi")} tel que
pour tout Xi : Mi" = (Mi Ω Mi').
avec l'opérateur Ω défini de la façon suivante:

Mi	Ω	g	f	a
	g	g	a	a
Mi'	f	a	f	a
	a	a	a	a

$Sv'' = Sv \sqcap Sv'$,

$Ps'' = Ps \sqcup Ps'$.

Remarque: La table de l'opérateur Ω est définie de telle façon que, si d'un côté, nous avons comme résultat pour une variable de la première substitution le même mode que dans le second cas, le mode résultant de l'union des deux résultats est ce mode. Si ces deux modes sont différents, nous ne pouvons que conclure, pour le mode résultant, par un a qui indique justement le fait que le mode est quelconque.

3. ext1

ext1({X1,..,Xn,Xn+1,..,Xn+m},{X1,..,Xn},(δ,Sv,Ps))= (δ',Sv',Ps')

où δ' = δ U {(Xn+1,f),..,(Xn+m,f)}

Sv' = Sv U {{Xn+1},..,{Xn+m}}

Ps' = Ps U {{Xn+1},..,{Xn+m}}

4. proj

proj({Xi1,..,Xik},{X1,..,Xl},(δ,Sv,Ps)) = (δ',Sv',Ps')

où δ' = δ \ { (Xi,Mi) : Xi ∈ {{X1,..,Xl}\{Xi1,..,Xik}} }

Sv' = { Sv'(Xi) = Sv(Xi) \ { {Xj} tq
Xj ∈ {{X1,..,Xl}\{Xi1,..,Xik}} } }

Ps' = { Ps'(Xi) = Ps(Xi) \ { {Xj} tq
Xj ∈ {{X1,..,Xl}\{Xi1,..,Xik}} } }

5. ext

ext(β , β') = β"

où β = (δ,Sv,Ps),
β' = (δ',Sv',Ps') et
β" = (δ",Sv",Ps") et
les domaines D' de β' et D de β sont tels que:
D = {X1,..,Xn} et D' = {Xi1,..,Xin} ==› D' est inclus dans D.

Définissons Dg comme l'ensemble des variables du domaine D qui ont le mode g; alors, on a:

$Sv'' = Sv \sqcap Sv'$

$Ps'' = Ps^* \sqcup Ps'$

où $Ps^* = \{ S\backslash Dg'' : S \in Ps \text{ et } S\backslash Dg'' \neq \emptyset \}$ et

$Dg'' = Dg \cup Dg' \Longrightarrow$ pour tout $Xi \in D$, Si $Mi=g$ alors $Xi \in Dg''$ et

$\Longrightarrow$ pour tout $Xi \in D'$, Si $Mi' =g$ alors $Xi \in Dg''$.

δ'' est construit sur D de la façon suivante:

$\delta'' = \{(Xi,Mi'')\}$ tq

| Si $Xi \in D' \Longrightarrow Mi'' = Mi'$.

| Si $Xi \in (D \setminus D') \Longrightarrow$

Si $(Sv''(Xi) \cap D' \neq \emptyset) \Longrightarrow Mi'' = Mj'$ où $Xj \in (Sv''(Xi) \cap D')$

Si $(Sv''(Xi) \cap D' = \emptyset) \Longrightarrow$

| $Mi'' = g$ si $Mi = g$

| $Mi'' = f$ si $Mi \neq g$ et pour tout $Xj \in Ps''(Xi)$: $Mj = f$

| $Mi'' = a$ si $Mi \neq g$ et il existe $Xj \in Ps''(Xi)$: $Mj = a$

<u>Remarque:</u> Si une des deux substitutions β ou β' est la substitution $\perp$ (qui représente la substitution où aucun des trois membres du triplet (δ,Sv,Ps) n'est défini et qui représente l'ensemble vide de substitutions), alors $ext(\beta, \beta') = \perp$.

6. <u>inst</u>

$inst(\beta, p/n, ETA) = \beta'$

où β' = lub({ β* : (β",p/n,β*) ∈ ETA et β" « β })
et l'opérateur « est défini de la manière suivante:

β « β' ssi pour toute variable X des substitutions β et β', les modes M de X dans β et M' de X dans β' sont tels que M « M'.

où M « M' est vrai si M=M' ou si M' = a. Dans le cas contraire, M et M' seront dits non comparables et β et β' ne seront pas comparables non plus.

7. adj

ETA := adj (β , p/n , β* , ETA)

signifie que ETA est remplacé par

ETA \ {(β',p/n,β*) : β « β'} U {(β',p/n,lub(β",β*)) : β « β'}

Ceci est nécessaire pour conserver la monotonie de l'ensemble ETA.

Remarque: L'ensemble ETA est dit monotone si, quelles que soient deux triplets (βi,p/n,βi') et (βj,p/n,βj'),si βi « βj, alors βi' « βj'.

C. Cas de l'unification.

1. Idée.

Avec un peu de perspicacité, nous pouvons remarquer qu 'il est facile de calculer la substitution résultante d'une unification grâce à la fonction d'extension ext que nous avons définie. En effet, seuls deux cas d'unification sont possibles pour un programme normalisé.

Dans le premier cas, nous aurons une unification de la forme Xi = Xj et donc, les deux modes seront égaux; les deux variables ainsi que les variables appartenant à la même classe de Sv et Ps ne formeront plus qu'une seule classe de Sv et une classe de Ps.

Dans le second cas, nous aurons une unification de la forme Xi = f(Xj1,..,Xjn) et nous pourrons déduire le mode de Xi en fonction des modes de l'ensemble {Xj1,..,Xjn} ou l'inverse et nous pourrons définir une classification pour Sv et Ps.

2. Réalisation.

Si notre unification est du type Xi = Xj, le résultat est le suivant:

$\delta^* = \{(Xi,(Mi@Mj)),(Xj,(Mi@Mj))\}$

$Sv^* = \{\{Xi,Xj\}\}$

| Si Mi = g ==› Ps* = {}

| Si Mi ‹› g ==› Ps* = {{Xi,Xj}}

où l'opérateur @ est défini de cette manière:

	Mi			
	@	g	f	a
	g	g	g	g
Mj	f	g	f	a
	a	g	a	a

Le résultat final sera calculé de la manière suivante:

$\beta' = ext (\beta , (\delta^*, Sv^*, Ps^*))$.

Si l'unification est du type Xi = f(Xj1,..,Xjn), et pour autant que Xi ne soit pas une des variables de l'ensemble {Xj1,..,Xjn}, nous pourrons donner la substitution β' résultante de cette unification:

$\delta^* = \{(Xi, Mi^*), (Xj1, Mj1^*), .., (Xjn, Mjn^*)\}$ où

Si Mi = g,

alors | Mi* = g et

| tous les Mjk* = g

Si Mi ‹› g et tous les Mjk = g,

alors | Mi* = g et

| tous les Mjk* = g

Si Mi ‹› g et il existe un Mjk ‹› g,

alors | Mi* = a et

| pour tout k, Mjk* = Mjk.

$Sv^* = Sv$

$Ps^* = \{ D \setminus Dg \}$ où

$D = \{Xi, Xj1, .., Xjn\}$ et $Dg = D \setminus \{ Xs : Xs = g \}$.

Le résultat final sera calculé de la même manière que dans le premier cas, c'est-à-dire $\beta' = ext (\beta , (\delta^*, Sv^*, Ps^*))$.

Remarques:

Si Xi est une des variables appartenant à une classe de Sv contenant une des variables de {Xj1,..,Xjn}, alors il est évident que l'unification ne peut qu'échouer et nous pourrons conclure plus rapidement avec $\beta' = \perp$. C'est un cas très intéressant qui permettra dans d'autres cas de donner plus de renseignements sur la substitution-résultat (voir chapitre VII).

VI. Spécifications de l'algorithme.

A. Les fonctions de base.

1. chvarg

Au cours des spécifications, nous rencontrerons des propriétés qui se répéteront. Nous les formaliserons donc quand cela sera nécessaire.

Propriété (1).

Soit L la liste [v(i1),..,v(ik)] et soit le triplet (D,SV,PS). Alors le triplet (D,SV,PS) vérifie la propriété (1) si D est une liste de k couples [(v(i1),Mi1),..,(v(in),Min)] où Mil vaut soit f, soit g, soit a et SV est une liste de sous-listes d'éléments de L1 où tout v(il) est représenté de manière unique, et PS est une liste de sous-listes d'éléments de L1 où tout v(il) est représenté de manière unique.

Propriété (2).

Soit L la liste [v(i1),..,v(ik)]. Alors la liste D vérifie la propriété (2) si D est une liste de k couples [(v(i1),Mi1),..,(v(in),Min)] où Mil vaut soit f, soit g, soit a.

Propriété (3).

Soit L la liste [v(i1),..,v(ik)]. Alors la liste PSV vérifie la propriété (3) si PSV est une liste de sous-listes d'éléments de L1 où tout v(il) est représenté de manière unique.

chvarg(L1,L2,(D1,SV1,PS1),(D2,SV2,PS2)).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. Le triplet (D1,SV1,PS1) vérifie (1) pour L1. L2 est une liste de n variables [v(j1),..,v(jn)].

Post: le triplet (D2,SV2,PS2) est le triplet (D1,SV1,PS1) où toute occurrence de v(ik) est remplacée par v(jk) et qui vérifie (1) pour L2.

chvargd(D1,D2,L1,L2).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. D1 vérifie (2) pour L1. L2 est une liste de n variables [v(j1),..,v(jn)].

Post: D2 est la liste de couples [(v(j1,Mi1),..,(v(jn),Min)] qui vérifie (2) pour L2.

inlst(X,L1,Y,L2).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. X est une variable appartenant à L1. L2 est une liste de n variables [v(j1),..,v(jn)].

Post: Y est la variable de L2 qui occupe la même position d'ordre que X dans L1.

chvargl(PSV1,PSV2,L1,L2).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. PSV1 vérifie (3) pour L1. L2 est une liste de n variables [v(j1),..,v(jn)].

Post: PSV2 est la liste PSV1 où toute occurrence de v(ik) est remplacée par v(jk) et qui vérifie (3) pour L2.

chvarglx(X,Y,L1,L2).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. X est une sous-liste de L1. L2 est une liste de n variables [v(j1),..,v(jn)].

Post: Y est la liste X où toute occurrence de v(ik) est remplacée par v(jk).

2. lub

lub((D1,SV1,PS1),(D2,SV2,PS2),(D3,SV3,PS3)).

Pré: Le triplets (D1,SV1,PS1) et (D2,SV2,PS2) vérifient la propriété (1).

Post: D3 est une liste de couples [(v(i1),Mi1"),.., (v(in),Min")] telle que Mik" = Mik Ω Mik' où Mik et Mik' sont les modes correspondants dans D1 et D2.
SV3 = SV1 $\sqcap$ SV2 et PS3 = PS1 $\sqcup$ PS2.

lubd(D1,D2,D3).

Pré: D1 et D2 vérifient la propriété (2).

Post: D3 est une liste de couples [(v(i1),Mi1"),.., (v(in),Min")] telle que Mik" = Mik Ω Mik' où Mik et Mik' sont les modes correspondants dans D1 et D2.

commun(D1,D2,D12,D11,D22).

Pré: D1 et D2 vérifient la propriété (2).

Post: D12 = D1 ∩ D2 et D11 = D1 \ D12 et D2 = D2 \D12.

lubop(D1,D2).

Pré: D1 est une liste de triplets (v(ik),Mik,Mik').

Post: D2 est une liste de couples (v(ik),Mik") telle que pour tout triplet de D1 il existe un couple correspondant de D2 où Mik" = Mik Ω Mik'.

3. ext1

ext1(L1,L2,(D1,SV1,PS1),(D2,SV2,PS2)).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. Le triplet (D1,SV1,PS1) vérifie (1) pour L1. L2 est une liste de k variables [v(j1),..,v(jk)] et la liste L2 est incluse dans L1.

Post: D2 = D1 U D où D est la liste des couples (v(il),f) telle que v(il) ∈ (L1 \ L2). SV2 = SV1 U SV où SV est la liste des listes singletons [v(il)] telle que v(il) ∈ (L1 \ L2). PS2 = PS1 U PS où PS est la liste des listes singletons [v(il)] telle que v(il) ∈ (L1 \ L2).

ext1d(L1,D1,D2).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. D1 vérifie la propriété (2).

Post: D2 = D1 U D où D est la liste de tous les couples (v(il),f) telle que v(il) ∈ L1 et n'est pas le premier élément d'un couple de D1.

extil(L1,L2,PSV1,PSV2).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. PSV1 est une liste qui vérifie la propriété (3). L2 est une liste de k variables [v(j1),..,v(jk)] et la liste L2 est incluse dans L1.

Post: PSV2 = PSV1 U PSV où PSV est la liste des listes singletons [v(il)] telle que v(il) ∈ (L1 \ L2).

4. proj

proj(L1,L2,(D1,SV1,PS1),(D2,SV2,PS2)).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. Le triplet (D1,SV1,PS1) vérifie (1) pour L1. L2 est une liste de k variables [v(j1),..,v(jk)] et la liste L1 est incluse dans L2.

Post: D2 est la liste des couples (v(il),Mil) telle que (v(il),Mil) ∈ D1 et v(il) ∈ L1. SV2 est la liste de listes SV1 tout v(il) ∈ (L2 \ L1) a été retiré. PS2 est la liste de listes PS1 où tout v(il) ∈ (L2 \ L1) a été retiré.

projd(L1,D1,D2).

Pré: L1 est une liste de n variables [v(i1),..,v(in)]. D1 vérifie la propriété (2).

Post: D2 est la liste des couples (v(il),Mil) telle que (v(il),Mil) ∈ D1 et v(il) ∈ L1.

diff(L1,L2,L3).

Pré: L1 et L2 sont des listes d'éléments.

Post: L3 est telle que l'addition des listes L2 et L3 donne L1 où L1 est sans répétition.

projl(L1,PSV1,PSV2).

Pré: L1 est une liste de k variables [v(i1),..,v(ik)]. PSV1 est une liste qui vérifie la propriété (3).

Post: PSV2 est la liste des listes PSV1 où tout X ∈ L1 a été retiré.

ret(X,L1,L2).

Pré: Soit X un élément et L1 une liste de listes d'éléments.

Post: L2 est la liste L1 où l'élément X a été enlevé de l'élément liste de L1 qui le contenait.

inin(X,L).

Pré: Soit X un élément et L une liste de listes d'éléments.

Post: La procédure réussit si X est un élément d'une liste élément de L. Elle échoue dans le cas contraire.

5. ext

ext((D1,SV1,PS1),(D2,SV2,PS2),(D3,SV3,PS3)).

Pré: Le triplets (D1,SV1,PS1) et (D2,SV2,PS2) vérifient la propriété (1) pour des listes différentes.

Post: (D3,SV3,PS3) est un triplet de listes qui vérifie en terme de listes les propriétés définies pour la substitution β" au paragraphe 5 du sous-chapitre B du chapitre II.

dg(D,L).

Pré: D est une liste qui vérifie la propriété (2).

Post: L est la liste de toutes les variables v(ik) des couples [(v(i1),Mi1),.., (v(in),Min)] telles que Mik=g.

union(L1,L2,L3).

Pré: L1 et L2 sont des listes d'éléments.

Post: L3 est la concaténation de L1 et de L2 où toute occurrence multiple d'un élément a été remplacée par une occurrence unique.

extd(D1,D2,D3).

Pré: D1 et D2 vérifient la propriété (2) pour des listes différentes.

Post: D3 est une liste qui vérifie en terme de listes les propriétés définies pour l'élément δ de la substitution β" au paragraphe 5 du sous-chapitre B du chapitre II.

choix((X,M1),D1,D2,SV,PS,M).

Pré: Soit (X,M1) un couple (variable,mode) appartenant au moins à D1. D1 et D2 vérifient la propriété (2) pour des listes différentes. SV et PS sont des listes de listes d'éléments et le mode de tout élément d'une liste élément de PS est différent de g.

Post: Si M1 = g, alors M = g. Sinon, Si X appartient à une liste de SV dont un élément Y est le premier élément d'un couple (Y,MM) appartenant à D2, alors M = MM. Sinon, Si tout élément de la liste contenant X de PS est de mode f dans la liste D1, alors M = f et sinon, M= a.

dd(D,L).

Pré: D une liste vérifiant la propriété (2).

Post: L est la liste de toutes les variables v(ik) des couples [(v(i1),Mi1),..., (v(in),Min)].

sv(X,SV,L).

Pré: Soit X un élément contenu dans une liste élément d'une liste SV.

Post: L est la liste élément de SV contenant X.

inter(L1,L2,L3).

Pré: Soit L1 et L2 deux listes d'éléments.

Post: L3 est la liste de tous les éléments appartenant à L1 et à L2.

mode(LSV,LPS,D1,D2,M).

Pré: D1 et D2 vérifient la propriété (2) pour des listes différentes. LSV et LPS sont des listes de listes d'éléments dont le mode de tout élément d'une liste élément de la liste LPS est différent de g.

Post: Si X appartient à une liste de LSV dont un élément Y est le premier élément d'un couple (Y,MM) appartenant à D2, alors M = MM. Sinon, Si tout élément de la liste élément contenant X de LPS est de mode f dans la liste D1, alors M = f et sinon, M= a.

mode(LPS,D1,M).

Pré: D1 vérifie la propriété (2). LPS est une liste de listes d'éléments dont le mode de tout élément d'une liste élément de la liste LPS est différent de g.

Post: Si tout élément de la liste élément contenant X de LPS est de mode f dans la liste D1, alors M = f, sinon M= a.

6. ouop

ouop(L1,L2,L3).

Pré: L1 et L2 sont deux listes qui vérifient la propriété (3).

Post: L3 est une liste telle que L3 = L1 ⊔ L2.

listou(L1,L2,L3).

Pré: Soit L1 une liste d'éléments et L2 une liste de listes d'éléments.

Post: L3 est la liste des listes éléments de L2 qui ont des éléments communs avec la liste L1.

regroupe(L1,L2).

Pré: Soit L1 une liste de listes d'éléments.

Post: L2 est une liste contenant tous les éléments de toutes les listes éléments de L1.

7. etop

etop(L1,L2,L3).

Pré: L1 et L2 sont 2 listes qui vérifient la propriété (3).

Post: L3 est une liste telle que L3 = L1 $\sqcap$ L2.

separe(L1,X,L2).

Pré: L1 est une liste de listes d'éléments et X est une liste d'éléments dont certains sont contenus dans les listes éléments de L1.

Post: L2 est la liste de toutes les intersections entre les listes éléments de L1 et la liste X à laquelle on ajoute l'élément liste composé de tous les éléments de X qui ne sont pas éléments d'éléments listes de L1.

separe(L1,X,LX,XX).

Pré: L1 est une liste de listes d'éléments et X est une liste d'éléments dont certains sont contenus dans les listes éléments de L1.

Post: LX est la liste de toutes les intersections entre les listes éléments de L1 et X et XX est une liste composée de tous les éléments de X qui ne sont pas éléments d'éléments listes de L1.

sep(L1,X,L2).

Pré: L1 est une liste de listes d'éléments et X est une liste d'éléments qui sont tous contenus dans les listes éléments de L1.

Post: L2 est la liste de toutes les intersections entre les listes éléments de L1 et X.

lappend(L1,L2,L3).

Pré: Soient L1 et L2 deux listes de listes d'éléments.

Post: L3 est la concaténation des listes de listes L1 et L2 où tout élément liste vide a été retiré.

8. inst

inst(B1,PN,B2).

Pré: Soit B1 un triplet vérifiant la propriété (1) et PN un but. Soit aussi la base de faits ETA contenant peut-être des faits de la forme p(Bi,PN,Bj).

Post: B2 = lub({ Bj : p(Bi,PN,Bj) ∈ ETA et Bi « B1 })

trie(L1,B,L2).

Pré: Soit L1 une liste de faits de la forme p(B1,PN,B2) où B1 et B2 sont des triplets qui vérifient la propriété (1) et PN est un but.

Post: L2 est la liste de tous les éléments p(Bi,PN,Bj) de L1 tels que Bi « B.

pluspetit(B1,B2).

Pré: Soit B1 et B2 deux triplets de la forme (D1,SV1,PS1) et (D2,SV2,PS2) vérifiant la propriété (1).

Post: La procédure réussit si, pour tout X premier élément d'un couple de D1 et Y l'élément correspondant dans D2, X « Y. La procédure échoue si ce n'est pas le cas.

pltit(D1,D2).

Pré: Soit D1 et D2 deux listes de couples vérifiant la propriété (2).

Post: La procédure réussit si pour tout X premier élément d'un couple de D1 et Y l'élément correspondant dans D2, X « Y. La procédure échoue si ce n'est pas le cas.

opcmp(M1,M2).

Pré: Soit M1 et M2 deux modes.

Post: La procédure réussit si M1 « M2,sinon elle échoue.

lubb(L,B2).

Pré: L est une liste de triplets [B1,..,Bn] vérifiant la propriété (1).

Post: Si L est la liste vide, alors B2 = $\perp$, dans le cas contraire, B2 = lub({ Bi : Bi ∈ L }).

9. adj

adj(B1,PN,B2).

Pré: Soit B1 et B2 deux triplets vérifiant la propriété (1) et PN un but. Soit aussi la base de faits ETA contenant peut-être des faits de la forme p(Bi,PN,Bj).

Post: Tout fait p(Bi,PN,Bj) ∈ ETA tel que B1 « Bi est remplacé par le fait p(Bi,PN,lub(Bj,B2)).

trieadj(L1,B,L2).

Pré: Soit L1 une liste de faits de la forme p(Bi,PN,Bj) où Bi et Bj sont des triplets qui vérifient la propriété (1) et PN est un but. Et soit B un triplet vérifiant la propriété (1).

Post: L2 est la liste de tous les faits p(Bi,PN,Bj) de L1 tels que B1 « Bi.

retrac(L).

Pré: Soit L une liste de faits de la forme p(_,PN,_) appartenant à la base de faits ETA.

Post: Il n'existe pas de fait de la forme p(_,PN,_) dans la base de faits.

asser(L,B).

Pré: Soit L une liste de faits de la forme p(Bi,PN,Bj) appartenant à la base de faits ETA et B un triplet vérifiant la propriété (1).

Post: La base de fait ETA contient tous les faits p(Bi,PN,lub(Bj,B)) correspondant à chaque fait contenu dans L.

B. L'algorithme_principal.

1. iasb

iasb(B1,PN,B2).

Pré: Soit B1 un triplet vérifiant la propriété (1) et PN un but. Soit aussi la base de faits ETA contenant peut-être des faits de la forme p(Bi,PN,Bj) et un ensemble de clauses dont la tête de clause s'unifie avec PN.

Post: Si la base de faits ETA contient le fait p(B1,PN,B), alors B2 = B. Sinon, B2 est le résultat de l'interprétation abstraite du but PN définie par l'algorithme I page 34.

executeclause(B1,PN,B2).

Pré: Soit B1 un triplet vérifiant la propriété (1) et PN un but. Soit aussi la base de faits ETA ne contenant pas de fait de la forme p(B1,PN,_) et un ensemble de clauses dont la tête de clause s'unifie avec PN.

Post: B2 est le résultat de l'interprétation abstraite du but PN définie par l'algorithme I page 34.

decision(B1,PN,B2,B3).

Pré: Soit B1 un triplet vérifiant la propriété (1) et PN un but. Soit aussi la base de faits ETA contenant le fait p(B1,PN,B) et un ensemble de clauses dont la tête de clause s'unifie avec PN.

Post: Si B2 = B, alors B3 = B. Sinon, B3 est le résultat de l'interprétation abstraite du but PN définie par l'algorithme I page 34.

execlause(B1,PN,B2).

Pré: Soit B1 un triplet vérifiant la propriété (1) et PN un but. Soit aussi la base de faits ETA ne contenant pas de fait de la forme p(B1,PN,_) et un ensemble de clauses dont la tête de clause s'unifie avec PN.

Post: B2 est le résultat de l'interprétation abstraite du but PN.

2. iasbuilt

iasbuilt(B1,X,B2).

Pré: Soit B1 un triplet de la forme (D1,SV1,PS1) vérifiant la propriété (1) et X une unification.

Post: B2 un triplet de la forme (D2,SV2,PS2) vérifiant la propriété (1) où D2 est le même que D1 sauf peut-être pour les variables impliquées dans l'unification X dont le mode est identifié en respect des algorithmes du paragraphe C du chapitre IV; SV2 est SV1 où la les listes contenant les variables impliquées dans l'unification X sont regroupées en une seule liste si l'unification est du type v(i)=v(j) sinon SV2 = SV1; et PS2 est PS1 où les listes contenant les variables impliquées dans l'unification X sont regroupées en une seule liste.

constitue(L1,L2).

Pré: Soit L1 une liste.

Post: L2 = [] si L1 = [] sinon, L2 = [L1].

phi(M1,M2,M3).

Pré: Soient M1 et M2 deux modes (a,f ou g).

Post: M3 est le mode calculé par M3 = M1 @ M2.

transps(M,L1,L2).

Pré: Soit M un mode (a,f ou g) et L1 une liste.

Post: Si M = g, alors L2 = [] sinon, L2 = L1.

phiplus(D1,X,Z,D).

Pré: Soit D1 une liste vérifiant la propriété (2) et X une variable appartenant à un couple contenu dans D1 et Z une liste de variables dont toute variable appartient à un couple contenu dans D1.

Post: Si X a le mode g dans D1 ou si toutes les variables de la liste Z ont le mode g dans la liste D1, D est la liste de tous les couples (variable,mode) constituée de la variable X de mode g et de toutes les variables de la liste Z auxquelles on donne le mode g. Sinon, la liste D est constituée de tous les couples (variable,mode) des variables de Z avec leur mode trouvé dans D1 et du couple (X,a).

transzg(Z,L).

Pré: Soit Z une liste de variables.

Post: L est la liste vérifiant la propriété (2) de tous les couples (variable,g) de toutes les variables contenues dans la liste Z.

tousg(Z,D).

Pré: Soit Z une liste de variables et D une liste vérifiant la propriété (2) pour une liste contenant Z.

Post: La procédure réussit si le mode de toutes les variables de la liste Z dans D1 est g. Dans le cas contraire, la procédure échoue.

transmode(Z,D,L).

Pré: Soit Z une liste de variables et D une liste vérifiant la propriété (2) pour une liste contenant Z.

Post: L est la liste vérifiant la propriété (2) de tous les couples (variable,Mi) de toutes les variables Xi contenues dans la liste Z où Mi est le mode correspondant à Xi dans D.

3. iac

iac(B1,C,B2).

Pré: Soit B1 un triplet vérifiant la propriété (1) et C une clause dont la tête de clause est PN. Soit aussi la base de faits ETA contenant peut-être des faits de la forme p(Bi,PN,Bj) et un ensemble fini de clauses.

Post: B2 est le résultat de l'interprétation abstraite de la clause C définie par l'algorithme II page 36.

execiac(B1,LB,B2,L).

Pré: Soit B1 un triplet vérifiant la propriété (1) et LB une liste de sous-buts. Soit aussi la base de faits ETA et L la liste de toutes les variables utilisées dans LB.

Post: B2 est le résultat de l'interpretation abstraite de la liste des sous-buts LB définie par l'algorithme II page 36.

varb(L1,L2).

Pré: Soit L1 une liste de sous-buts.

Post: L2 est la liste de toutes les variables utilisées dans tous les sous-buts de la liste L1.

varbx(X,L).

Pré: Soit X un but.

Post: L est la liste de toutes les variables utilisées dans le but X.

newvar(L1,L2).

Pré: Soit L1 une liste de n variables.

Post: Si L1 est une liste vide, alors L2 est une liste vide. Sinon L2 une liste de n variables successives v(i) dont la première est v(1).

newvar(L1,L2,M).

Pré: Soit L1 une liste de n variables avec n > 0.

Post: L2 une liste de n variables successives v(i) dont la première est v(1) et M = n.

4. verifie

verifie.

Pré: -.

Post: Le nom du fichier contenant la procédure et la procédure à étudier avec ses arguments sont lus. L'interprétation abstraite de cette procédure est réalisée et le résultat de cette interprétation abstraite est affiché à l'écran. Ce résultat d'interprétation abstraite est définie par l'algorithme I page 33.

lirebeta(PN,B,PNORM,Q).

Pré: PN est un but.

Post: B est le triplet vérifiant la propriété (1) déduit du but PN et PNORM est le but PN où tout argument du but a été remplacé par une variable et les variables de PNORM sont successives et la première est v(1). Q est la liste des arguments de PN.

tested(L,D,N).

Pré: L est une liste d'arguments et N, un entier.

Post: D est la liste des couples vérifiant la propriété (2) telle qu'à tout argument i de L correspond le couple (v(i),Mi) dans D où Mi est le mode de l'argument i.

moded(X,M).

Pré: X est un terme.

Post: M est le mode du terme X.

ground(X).

Pré: X est un terme simple ou composé.

Post: La procédure réussit si le terme X est un terme de base et échoue sinon.

grnd(L).

Pré: L est une liste de termes simples ou composés.

Post: La procédure réussit si tout terme de la liste L est un terme de base et échoue dans le cas contraire .

testesv(L,SV).

Pré: L est une liste de variables.

Post: SV est la liste de toutes les listes singletons où tout élément singleton d'un élément liste de SV est un élément de la liste L.

testeps(L,PS).

Pré: L est une liste.

Post: Si L = [], alors PS = []; sinon PS = [L].

normepn(P,PN).

Pré: P est un but.

Post: PN est le but P où tout argument du but a été remplacé par une variable et les variables de PN se succèdent en commençant par la variable v(1).

replace(Q,D1,D2).

Pré: Q est une liste de n arguments et D1 est une liste de n couples vérifiant la propriété (2).

Post: D2 est une liste de n couples dont le premier élément du ième couple a été remplacé par le ième argument de la liste Q.

insere(L).

Pré: L est une liste de n clauses normalisées.

Post: La base de faits contient n faits du type clause(T,N,M,C) où pour toute clause de la liste L, T est la tête de la clause C, N est son arité, et M est le numéro d'ordre de la clause dans sa procédure.

assertclause(T,N,C).

Pré: T est la tête de la clause C et N son arité.

Post: La base de fait contient le fait du type clause(T,N,M,C) où M est le numéro d'ordre de la clause dans sa procédure.

retire.

Pré: Soit la base de faits contenant peut-être des faits du type clause/4.

Post: La base de faits ne contient plus de faits du type clause/4.

VII. Exemples.

Voici quelques exemples révélateurs de la capacité d'interprétation abstraite des programmes PROLOG par notre programme. Le premier de ces exemples est la fonction "concat" qui concatène deux listes pour en donner une troisième. Cette fonction a été choisie pour sa simplicité.

Le texte de la fonction concat est le suivant:

```
concat([],L,L).
concat([X|L1],L2,[X|L3]) :-  concat(L1,L2,L3).
```

Ce programme est normalisé avant son interprétation; en voici le texte:

```
concat(v(1),v(2),v(3)) :- v(1)=[],v(2)=v(3).
concat(v(1),v(2),v(3)) :- v(1)=[v(4)|v(5)],v(3)=[v(4)|v(6)],
                                        concat(v(5),v(2),v(6)).
```

La première procédure interprétée était la procédure:

```
concat([a],[b],L).
```

Ce qui donne la substitution de départ:

```
{([a],g),([b],g),(L,f)}.
```

Le résultat obtenu est le suivant:

{([a],g),([b],g),(L,a)}.

Nous pouvons observer que le résultat obtenu n'est pas optimal. En effet, le mode de la liste L résultat de la concaténation de [a] et de [b] est la liste [a,b] dont le mode est g. Ceci est dû au fait que nous interprétons les unifications avant l'application du but donnant naissance à ces unifications. Un moyen très simple de palier ce défaut serait, pendant la normalisation, de rajouter ces unifications avant et après le but concerné au lieu de les écrire simplement avant le but. De cette manière, ces unifications seront exécutées deux fois, mais elles permettront d'affiner l'interprétation.

Pour cette même fonction concat(L1,L2,L3), nous avons essayé quelques autres possibilités logiques d'arguments. Voici un tableau qui résume les résultats obtenus:

	avant			après		
Mode de	L1	L2	L3	L1	L2	L3
	g	g	g	g	g	g
	g	g	f	g	g	a
	g	f	g	g	a	g
	f	g	g	a	g	g
	f	f	g	a	a	g
	f	f	f	a	a	a
	g	f	g	g	a	g
	f	g	g	a	g	g

L'exemple suivant montre que notre interprétation peut donner malgré tout un excellent résultat. Il est tiré de l'article [2] et a été décrit pour la première fois par Debray [9].

```
p(X,Y) :- q(X,Y),r(X),s(Y).
q(Z,Z).
r(a).
s(W).
```

Voici sa normalisation:

```
p(v(1),v(2)) :- q(v(1),v(2)),r(v(1)),s(v(2)).
q(v(1),v(2)) :- v(1)=v(2).
r(v(1)) :- v(1)=a.
s(v(1)).
```

L'exemple significatif, est d'essayer l'interprétation avec les modes {(X,f),(Y,f)} comme arguments de p. Nous obtenons alors {(X,g),(Y,g)}.

C'est bien le résultat auquel nous serions en droit de nous attendre. L'exécution de notre algorithme dans ce cas précis est plus que satisfaisant.

Alors que nous ne l'avions pas encore fait, l'exemple suivant dû à Monsieur B Le Charlier montre l'utilité de la fonction "adj":

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

Normalisé, cela donne:

```
p(v(1)) :- v(1)=f(v(2)).
p(v(1)) :- q(v(1)),p(v(2)).
q(v(1)).
q(v(1)) :- q(v(1)),p(v(1)).
```

Si nous essayons cet exemple avec la substitution {(X,a)} comme argument de la procédure p, nous obtenons le résultat {(X,a)}. Si nous n'avions pas, au moment opportun, effectué la transformation de l'ensemble ETA grâce à la fonction "adj", pendant l'interprétation de cette procédure, l'algorithme aurait bouclé. Car à un certain moment dans l'arbre de résolution, la substitution résultat aurait été une fois sur deux {(X,f)} et à l'exécution suivante {(X,a)}, de telle sorte que la comparaison effectuée après une interprétation complète avec le résultat de l'interprétation précédemment existante aurait toujours été négative.

La cause de ce bouclage vient de l'ensemble ETA. Si la fonction "adj" n'est pas appliquée après une résolution d'un but, il pourrait exister une rupture dans la monotonie de l'ensemble ETA. Cet ensemble n'aurait plus alors la même signification.

VIII. Conclusions.

Nous sommes parvenu à réaliser un programme d'interprétation abstraite des programmes écrits en PROLOG. Il permet, pour un programme donné et pour une substitution donnée, de calculer la substitution résultante de l'application de la procédure étudiée. Cette interprétation a été accomplie pour les modes des variables et donne des résultats significatifs.

Les théories exposées au cours de ce mémoire forment une base solide pour une étude plus approfondie des propriétés statiques des programmes PROLOG.

L'algorithme général est écrit de telle manière qu'il restera inchangé si on veut réaliser une étude autre qu'une étude de modes. En effet il suffira de récrire les procédures intermédiaires (chvarg, lub,...).

Il est possible de trouver dans certains cas une interprétation plus signifiante que celle que nous avons obtenue. Il est possible d'augmenter la vitesse d'exécution de notre programme, mais tel n'était pas notre propos. Il s'agissait pour nous d'écrire un programme qui donne des résultats corrects et une interprétation la plus complète possible.

Lors de la normalisation, en permettant une redondance de certaines unifications, nous pourrions affiner encore notre interprétation.

Références.

[1] COUSOT P., COUSOT R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Conf. Rec. $th ACM Symp. on Princ. of Programming Languages, 1977, pp 238-252.

[2] BRUYNOOGHE M. and al.: Abstract Interpretation: Towards the Optimisation of PROLOG Programs. Proc. of the 1987 Symp. on Logic Programming, San Fransisco 1987, pp 192-204.

[3] MISHRA P., KELLER M.: Static Inference of Properties of Applicative Programs. ACM Symp. on Principles of Programming Languages, Salt Lake City, January 1984, pp 235-244.

[4] MELLISH C.S.: Abstract Interpretation of PROLOG Programs. Proc. Third Int Conf on Logic Programming, London, July 1986, LNCS 225, Springer Verlag 1986, pp 463-474.

[5] DRABENT W., MALUSZYNSKI J.: Inductive assertion Method for Logic Programs. Proc. TAPSOFT 87, Vol 2, LNCS 250, Springer Verlag 1987, pp 167-181.

[6] SONDERGAARD H.: An application of Abstract Interpretation of Logic Programs: Occur check Reduction. Proc. European Symp. on Programming, Saarbrucken, March 1986, LNCS 213, Springer Verlag 1986, pp 327-338.

[7] LECHARLIER B.: A paraître.

[8] SHAPIRO E., STERLING L.: The art of PROLOG: Advanced Programming Techniques. The MIT Press, 1986.

[9] DEBRAY S.K. and WARREN D.S.: Automatic mode inferencing for PROLOG Programs. Proc. 1986 Logic Programming Symposium, Salt Lake City, Sept. 1986. IEEE Society Press, 1986, pp 355-359.

Annexe.

```
/*****************************************************************/
/***                        Utilitaires.                       ***/
/*****************************************************************/

append([],L,L):- !.
append([X|L1],L2,[X|L3]) :-
        append(L1,L2,L3),
        !.

dans((X,XX),[(Y,XX)|_]) :-
        X==Y,
        !.
dans((X,XX),[(Y,_)|L]) :-
        dans((X,XX),L),
        !.

in(X,[Y|L]) :-
        X==Y,
        !.
in(X,[Y|L]) :-
        in(X,L),
        !.

retr(X,[Y|L],L) :-
        X==Y,
        !.
retr(X,[Y|L],[Y|LL]) :-
        retr(X,L,LL),
        !.

reconstr(P,[],P) :- !.
reconstr(P,L,(P:-Q)) :-
        constr(Q,L),
        !.

constr((X,B),[X|L]) :-
        constr(B,L),
        !.
constr(X,[X]) :- !.

/*****************************************************************/
/***                        Normalisation.                     ***/
/*****************************************************************/

norme(SF,L) :-
        see(SF),
        normfile(L),
        seen,
        !.

normfile([R|L]) :-
        read(T),
        normclause(T,R),
        normfile(L),
        !.
normfile([]) :- !.
```

```
normclause('end_of_file',_) :-
        !,
        fail.
normclause((H:-G),T) :-
        normtete(H,P,LB,LVAR,M),
        constr(G,L1),
        normcorps(L1,LN,LVAR,M),
        append(LB,LN,L),
        reconstr(P,L,T),
        !.
normclause(H,T) :-
        normtete(H,P,LB,LVAR,M),
        reconstr(P,LB,T),
        !.

/*****************************************************************/
/***                        Normtete ...                       ***/
/*****************************************************************/

normtete(H,P,LB,LV,M) :-
        H=..[T|A],
        normtete(A,AN,LBB,[],LV1,1,MM),
        normlblt(LBB,LB,LV1,LV,MM,M),
        P=..[T|AN],
        !.

normtete([],[],[],LV,LV,N,N) :- !.
normtete([X|A],[v(I)|AN],[(v(I)=v(J))|LB],LV,LV1,I,M) :-
        var(X),
        dans((X,J),LV),
        I1 is I+1,
        normtete(A,AN,LB,LV,LV1,I1,M),
        !.
normtete([X|A],[v(I)|AN],LB,LV,LV1,I,M) :-
        var(X),
        I1 is I+1,
        LVX = [(X,I)|LV],
        normtete(A,AN,LB,LVX,LV1,I1,M),
        !.
normtete([X|A],[v(I)|AN],[(v(I)=X)|LB],LV,LV1,I,M) :-
        I1 is I+1,
        normtete(A,AN,LB,LV,LV1,I1,M),
        !.

normlblt([],[],L,L,M,M) :- !.
normlblt([(v(I)=v(J))|L],[(v(I)=v(J))|LB],LV,LVN,N,M) :-
        normlblt(L,LB,LV,LVN,N,M),
        !.
normlblt([(X=Y)|L],LB,LV,LVN,N,M) :-
        normterme(Y,Z,LBN,LV,LVI,N,NN),
        append(LBN,[(X=Z)],LL),
        normlblt(L,LLB,LVI,LVN,NN,M),
        append(LL,LLB,LB),
        !.
```

```
/*****************************************************************/
/***                          Normcorps ...                    ***/
/*****************************************************************/

normcorps(LC,LCN,LV,M) :-
        normcorps(LC,LCN,LV,M,_),
        !.

normcorps([],[],_,N,N) :- !.
normcorps([(X=Y)|LC],LCN,LV,M,N) :-
        normbltin((X=Y),Z,LB,LV,LV1,M,M1),
        normlblt(LB,LBN,LV1,LV2,M1,M2),
        append(LBN,[Z],LN),
        normcorps(LC,LCCN,LV2,M2,N),
        append(LN,LCCN,LCN),
        !.
normcorps([X|LC],LCN,LV,M,N) :-
        normterme(X,Z,LB,LV,LV1,M,M1),
        normlblt(LB,LBN,LV1,LV2,M1,M2),
        append(LBN,[Z],LN),
        normcorps(LC,LCCN,LV2,M2,N),
        append(LN,LCCN,LCN),
        !.

normterme(X,Y,[],LV,LVN,M,M1) :-
        var(X),
        ntvar(X,Y,LV,LVN,M,M1),
        !.
normterme(X,Y,LB,LV,LVN,M,M1) :-
        X=..[T|LA],
        nter(LA,LAN,LB,LV,LVN,M,M1),
        Y=..[T|LAN],
        !.

normbltin((X=Y),Z,LB,LV,LV1,M,M1) :-
        var(X),
        nbltin(X,Y,Z,LB,LV,LV1,M,M1),
        !.
normbltin((X=Y),Z,LB,LV,LV1,M,M1) :-
        var(Y),
        nbltin(Y,X,Z,LB,LV,LV1,M,M1),
        !.
normbltin((X=Y),(v(M)=XX),[(v(M)=Y)|LBX],LV,LV1,M,I) :-
        functor(X,F1,N1),
        functor(Y,F2,N2),
        F1==F2,
        N1==N2,
        M1 is M+1,
        normterm(X,XX,LBX,LV,LV1,M1,I),
        !.
normbltin(_,fail,LV,LV,M,M) :- !.

nbltin(X,Y,(v(I)=Z),LB,LV,LV1,M,M1) :-
        dans((X,I),LV),
        normterme(Y,Z,LB,LV,LV1,M,M1),
        !.
```

```
nbltin(X,Y,(v(M)=Z),LB,LV,LV1,M,N) :-
        M1 is M+1,
        normterme(Y,Z,LB,[(X,M)|LV],LV1,M1,N),
        !.

ntvar(X,v(I),LV,LV,M,M) :-
        dans((X,I),LV),
        !.
ntvar(X,v(M),LV,[(X,M)|LV],M,M1) :-
        M1 is M+1,
        !.

nter([],[],[],LV,LV,M,M) :- !.
nter([X|LA],[Y|LAN],LB,LV,LVN,M,N) :-
        var(X),
        ntvar(X,Y,LV,LVI,M,MM),
        nter(LA,LAN,LB,LVI,LVN,MM,N),
        !.
nter([X|LA],[v(M)|LAN],[(v(M)=X)|LB],LV,LVN,M,N) :-
        M1 is M+1,
        nter(LA,LAN,LB,LV,LVN,M1,N),
        !.

/*****************************************************************/
/***                          CHVARG                           ***/
/*****************************************************************/

chvarg(L1,L2,(D1,SV1,PS1),(D2,SV2,PS2)) :-
        chvargd(D1,D2,L1,L2),
        chvargl(SV1,SV2,L1,L2),
        chvargl(PS1,PS2,L1,L2),
        !.

chvargd([],[],L1,L2) :- !.
chvargd([(X1,M1)|LL1],[(Y1,M1)|LL2],L1,L2) :-
        !,
        inlst(X1,L1,Y1,L2),
        chvargd(LL1,LL2,L1,L2),
        !.

inlst(X1,[X],Y1,[Y1]) :-
        X1==X,
        !.
inlst(X1,[X|_],Y1,[Y1|_]) :-
        X1==X,
        !.
inlst(X1,[X|L1],Y1,[_|L2]) :-
        X1\==X,
        inlst(X1,L1,Y1,L2),
        !.

chvargl([],[],L1,L2) :- !.
chvargl([X|LL1],[Y|LL2],L1,L2) :-
        !,
        chvarglx(X,Y,L1,L2),
        chvargl(LL1,LL2,L1,L2),
```

```
        !.

chvarglx([],[],_,_) :- !.
chvarglx([X|LL1],[Y|LL2],L1,L2) :-
        !,
        inlst(X,L1,Y,L2),
        chvarglx(LL1,LL2,L1,L2),
        !.

/*****************************************************************/
/***                        REUNION LUB                        ***/
/*****************************************************************/

lub((D1,SV1,PS1),(D2,SV2,PS2),(D3,SV3,PS3)) :-
        lubd(D1,D2,D3),
        etop(SV1,SV2,SV3),
        ouop(PS1,PS2,PS3),
        !.

lubd(D1,D2,D3) :-
        commun(D1,D2,D12,D11,D22),
        append(D11,D22,D33),
        lubop(D12,D),
        append(D,D33,D3),
        !.

commun([],D,[],[],D) :- !.
commun([(X,M)|D1],D2,[(X,M,M2)|D12],D11,D22) :-
        dans((X,M2),D2),
        retr((X,M2),D2,DZ),
        commun(D1,DZ,D12,D11,D22),
        !.
commun([(X,M)|D1],D2,D12,[(X,M)|D11],D22) :-
        commun(D1,D2,D12,D11,D22),
        !.

lubop([],[]) :- !.
lubop([(X,M,M)|D12],[(X,M)|D]) :-
        lubop(D12,D),
        !.
lubop([(X,M1,M2)|D12],[(X,a)|D]) :-
        lubop(D12,D),
        !.

/*****************************************************************/
/***                      EXTENSION SIMPLE                     ***/
/*****************************************************************/

ext1(L1,L2,(D1,SV1,PS1),(D2,SV2,PS2)) :-
        ext1d(L1,D1,D2),
        ext11(L1,L2,SV1,SV2),
        ext11(L1,L2,PS1,PS2),
        !.

ext1d([],L,[]) :- !.
ext1d([X|LL1],D1,[(X,Z)|D2]) :-
```

```
        dans((X,Z),D1),
        extld(LL1,D1,D2),
        !.
extld([X|LL1],D1,[(X,f)|D2]) :-
        extld(LL1,D1,D2),
        !.

extll([],L,L1,L1) :- !.
extll([X|LL1],L2,LS1,LS2) :-
        in(X,L2),
        extll(LL1,L2,LS1,LS2),
        !.
extll([X|LL1],L2,LS1,[[X]|LS2]) :-
        extll(LL1,L2,LS1,LS2),
        !.

/*****************************************************************/
/***                        PROJECTION                         ***/
/*****************************************************************/

proj(L1,L2,(D1,SV1,PS1),(D2,SV2,PS2)) :-
        projd(L1,D1,D2),
        diff(L2,L1,L),
        projl(L,SV1,SV2),
        projl(L,PS1,PS2),
        !.

projd([],D1,[]) :- !.
projd([X|LL1],D1,[(X,Z)|D2]) :-
        dans((X,Z),D1),
        projd(LL1,D1,D2),
        !.
projd([X|LL1],D1,D2) :-
        projd(LL1,D1,D2),
        !.

projl([],L,L) :- !.
projl([X|L],L1,L2) :-
        inin(X,L1),
        ret(X,L1,LZ),
        projl(L,LZ,L2),
        !.
projl([X|L],L1,L2) :-
        projl(L,L1,L2),
        !.

diff([],L,[]) :- !.
diff([X|L1],L2,L3) :-
        in(X,L2),
        diff(L1,L2,L3),
        !.
diff([X|L1],L2,[X|L3]) :-
        diff(L1,L2,L3),
        !.

ret(X,[[Y]|L],L) :-
```

```
        X==Y,
        !.
ret(X,[[Y|L1]|L],[_1|L]) :-
        X==Y,
        !.
ret(X,[[Y|L1]|L],[[Y|L2]|L]) :-
        retr(X,L1,L2),
        !.
ret(X,[Y|L1],[Y|L2]) :-
        ret(X,L1,L2),
        !.

inin(X,[Y]) :-
        in(X,Y),
        !.
inin(X,[Y|_]) :-
        in(X,Y),
        !.
inin(X,[_|L]) :-
        inin(X,L),
        !.

/*****************************************************************/
/***                        EXTENSION                          ***/
/*****************************************************************/

ext((D1,SV1,PS1),(D2,SV2,PS2),(D3,SV3,PS3)) :-
        ouop(SV1,SV2,SV3),
        dg(D1,DG1),
        dg(D2,DG2),
        union(DG1,DG2,DG3),
        proj1(DG3,PS1,PSTAR),
        ouop(PSTAR,PS2,PS3),
        extd(D1,D2,D3,SV3,PS3),
        !.

dg([],[]) :- !.
dg([(X,g)|D],[X|DG]) :-
        dg(D,DG),
        !.
dg([(X,M)|D],DG) :-
        dg(D,DG),
        !.

union(L1,L2,L3) :-
        append(L1,L2,L),
        sort(L,L3),
        !.

extd([],_,[],_,_) :- !.
extd([(X,M1)|D1],D2,[(X,M)|D3],SV3,PS3) :-
        dans((X,M),D2),
        extd(D1,D2,D3,SV3,PS3),
        !.
extd([(X,M1)|D1],D2,[(X,M2)|D3],SV3,PS3) :-
        choix((X,M1),D1,D2,SV3,PS3,M2),
```

```
        extd(D1,D2,D3,SV3,PS3),
        !.

choix((X,g),_,_,_,_,g) :- !.
choix((X,M1),D1,D2,SV3,PS3,M2) :-
        dd(D2,D),
        sv(X,SV3,L),
        inter(D,L,LSV),
        sv(X,PS3,LPS),
        mode(LSV,LPS,D1,D2,M2),
        !.

dd([],[]) :- !.
dd([(X,M)|D2],[X|D]) :-
        dd(D2,D),
        !.

sv(X,[L|_],L) :-
        in(X,L),
        !.
sv(X,[L1|L],LL) :-
        sv(X,L,LL),
        !.

inter([],_,[]) :- !.
inter([X|L1],L2,[X|L3]) :-
        in(X,L2),
        inter(L1,L2,L3),
        !.
inter([X|L1],L2,L3) :-
        inter(L1,L2,L3),
        !.

mode([Y|_],_,_,D2,M2) :-
        dans((Y,M2),D2),
        !.
mode([],LPS,D1,_,M2) :-
        mode(LPS,D1,M2),
        !.

mode([],D1,f) :- !.
mode([X|LPS],D1,a) :-
        dans((X,a),D1),
        !.
mode([X|LPS],D1,M) :-
        mode(LPS,D1,M),
        !.

/*****************************************************************/
/***                          OUOP                             ***/
/*****************************************************************/

ouop([],L,L) :- !.
ouop([X|L1],L2,L3) :-
        listou(X,L2,L),
        L==[],
```

```
        oubp(L1,L2,LZ),
        append([X],LZ,L3),
        !.
oubp([X|L1],L2,L3) :-
        listou(X,L2,L),
        diff(L2,L,LL2),
        regroupe(L,LL),
        union(X,LL,LX),
        oubp(L1,[LX|LL2],L3),
        !.

listou(L1,[],[]) :- !.
listou(L1,[X|L2],L3) :-
        inter(L1,X,L),
        L==[],
        listou(L1,L2,L3),
        !.
listou(L1,[X|L2],[X|L3]) :-
        listou(L1,L2,L3),
        !.

regroupe([],[]) :- !.
regroupe([X],X) :- !.
regroupe([X,Y|L],L1) :-
        union(X,Y,Z),
        regroupe([Z|L],L1),
        !.

/*****************************************************************/
/***                        ETOP                               ***/
/*****************************************************************/

etop([],L,L) :- !.
etop([X|L1],L2,L3) :-
        listou(X,L2,L),
        L==[],
        etop(L1,L2,LZ),
        lappend([X],LZ,L3),
        !.
etop([X|L1],L2,L3) :-
        listou(X,L2,L),
        diff(L2,L,LL2),
        separe(L,X,LX),
        append(LX,LL2,LL),
        etop(L1,LL,L3),
        !.

separe(L,X,LL) :-
        separe(L,X,LX,XX),
        lappend([XX],LX,LL),
        !.

separe(L,X,LX,XX) :-
        regroupe(L,LL),
        inter(LL,X,XY),
        diff(X,XY,XX),
```

```
        sep(L,XY,LX),
        !.

sep([],_,[]) :- !.
sep([X|L],Z,LZ) :-
        inter(X,Z,_1),
        diff(X,L1,L2),
        sep(L,Z,LK),
        lappend([L1,L2],LK,LZ),
        !.

lappend([],L,L) :- !.
lappend([[]|L1],L2,L3) :-
        lappend(L1,L2,L3),
        !.
lappend([X|L1],L2,[X|L3]) :-
        lappend(L1,L2,L3),
        !.

/*****************************************************************/
/***                          IASB                             ***/
/*****************************************************************/

iasb(B,PN,BPRIM) :-
        p(B,PN,BPRIM),
        !.
iasb(B,PN,BPRIM) :-
        inst(B,PN,INSTB),
        asserta(p(B,PN,INSTB)),
        executeclause(B,PN,BPRIM),
        retract(p(B,PN,_)),
        !.

executeclause(B,PN,BPRIM) :-
        execlause(B,PN,BPR),
        decision(B,PN,BPR,BPRIM),
        !.

decision(B,PN,BPRIM,BPRIM) :-
        p(B,PN,BSTAR),
        egale(BPRIM,BSTAR),
        !.
decision(B,PN,BPR,BPRIM) :-
        retract(p(B,PN,_)),
        asserta(p(B,PN,BPR)),
        adj(B,PN,BPR),
        executeclause(B,PN,BPRIM),
        !.

execlause(B,PN,BPRIM) :-
        execc(B,PN,([],[],[]),BPRIM,1),
        !.

execc(B,PN,BPRM,BPRIM,I) :-
        functor(PN,P,N),
        clause(P,N,I,C),
```

```
        iac(B,C,BSTAR),
        p(B,PN,BPR),
        lub(BPRM,BSTAR,BPR2),
        I1 is I+1,
        execc(B,PN,BPR2,BPRIM,I1),
        !.
execc(_,_,BPRIM,BPRIM,_) :- !.

egale((D1,SV1,PS1),(D1,SV2,PS2)) :-
        sort(SV1,SV),
        sort(SV2,SV),
        sort(PS1,PS),
        sort(PS2,PS),
        !.

adj(B,PN,BPRIM) :-
        bagof(p(B1,PN,B2),p(B1,PN,B2),L),
        trieadj(L,B,LA),
        retrac(LA),
        asser(LA,BPRIM),
        !.

trieadj([],_,[]) :- !.
trieadj([p(B1,PN,B2)|L1],B,[p(B1,PN,B2)|LA]) :-
        pluspetit(B,B1),
        trieadj(L1,B,LA),
        !.
trieadj([_|L1],B,LA) :-
        trieadj(L1,B,LA),
        !.

retrac([]) :- !.
retrac([X|L]) :-
        retract(X),
        retrac(L),
        !.

asser([],_) :- !.
asser([p(B1,PN,B2)|L],B) :-
        lub(B2,B,BB),
        asserta(p(B1,PN,BB)),
        asser(L,B),
        !.

/*****************************************************************/
/***                          IASBUILT                         ***/
/*****************************************************************/

iasbuilt((D1,SV1,PS1),(v(I)=v(J)),B) :-
        dans((v(I),MI),D1),
        dans((v(J),MJ),D1),
        phi(MI,MJ,M),
        D=[(v(I),M),(v(J),M)],
        sv(v(I),SV1,Z1),
        sv(v(J),SV1,Z2),
        retr(Z1,SV1,SVA),
```

```
        retr(Z2,SVA,SVB),
        union(Z1,Z2,Z),
        transps(M,[[v(I),v(J)]],PS),
        lappend([Z],SVB,SV),
        ext((D1,SV1,PS1),(D,SV,PS),B),
        !.
iasbuilt(_,(v(I)=F),([],[],[])) :-
        F=..[X|Z],
        in(v(I),Z),
        !.
iasbuilt((D1,SV1,PS1),(v(I)=F),B) :-
        F=..[X|Z],
        phiplus(D1,v(I),Z,D),
        dd(D1,DD),
        dg(D,DG),
        diff(DD,DG,DA),
        constitue(DA,PS),
        ext((D1,SV1,PS1),(D,SV,PS),B),
        !.

constitue([],[]) :- !.
constitue(L,[L]) :- !.

phi(M,M,M) :- !.
phi(g,_,g) :- !.
phi(_,g,g) :- !.
phi(_,_,a) :- !.

transps(g,_,[]) :- !.
transps(_,L,L) :- !.

phiplus(D1,X,Z,[(X,g)|ZZ]) :-
        dans((X,g),D1),
        transzg(Z,ZZ),
        !.
phiplus(D1,X,Z,[(X,g)|ZZ]) :-
        tousg(Z,D1),
        transzg(Z,ZZ),
        !.
phiplus(D1,X,Z,[(X,a)|ZZ]) :-
        transmode(Z,D1,ZZ),
        !.

transzg([],[]) :- !.
transzg([X|L],[(X,g)|LL]) :-
        transzg(L,LL),
        !.

tousg([],_) :- !.
tousg([X|L],D1) :-
        dans((X,g),D1),
        tousg(L,D1),
        !.

transmode([],_,[]) :- !.
transmode([X|L],D1,[(X,M)|LL]) :-
```

```
        dans((X,M),D1),
        transmode(L,D1,LL),
        !.

/*****************************************************************/
/***                          IAC                              ***/
/*****************************************************************/

iac((D,SV,PS),(H:-G),BPRIM) :-
        constr(G,LB),
        varb(LB,LB1),
        H=..[T|L2],
        union(LB1,L2,L1),
        dd(D,LD),
        ext1(L1,LD,(D,SV,PS),BEXT),
        execiac(BEXT,LB,BRES,L1),
        proj(L2,L1,BRES,BPRIM),
        !.
iac(B,(H),B,_) :- !.

execiac(B,[],B,_) :- !.
execiac(BEXT,[(X=Y)|L],BRES,L1) :-
        varb([(X=Y)],LX),
        proj(LX,L1,BEXT,BR),
        iasbuilt(BR,(X=Y),BSUC),
        ext(BEXT,BSUC,BEX),
        execiac(BEX,L,BRES,L1),
        !.
execiac(BEXT,[X|L],BRES,L1) :-
        varb([X],LX),
        proj(LX,L1,BEXT,BR),
        newvar(LX,LN),
        chvarg(LX,LN,BR,BRN),
        iasb(BRN,X,BSU),
        chvarg(LN,LX,BSU,BSUC),
        ext(BEXT,BSUC,BEX),
        execiac(BEX,L,BRES,L1),
        !.

varb([],[]) :- !.
varb([X|L],LV) :-
        varbx(X,L1),
        varb(L,LL),
        union(LL,L1,LV),
        !.

varbx((X=Y),[X|L]) :-
        varb([Y],L),
        !.
varbx(v(I),[v(I)]) :- !.
varbx(T,L) :-
        T=..[_|L],
        !.

newvar([],[]) :- !.
newvar(LX,LN) :-
```

```
        newvar(LX,LN,N),
        !.

newvar([X],[v(1)],1) :- !.
newvar([X|L],LL,I1) :-
        newvar(L,LV,I),
        I1 is I+1,
        append(LV,[v(I1)],LL),
        !.

/*****************************************************************/
/***                          INST                             ***/
/*****************************************************************/

inst(B,PN,BPRIM) :-
        bagof((X,Y),p(X,PN,Y),L),
        trie(L,B,LB),
        lubb(LB,BPRIM),
        !.
inst(_,_,([],[],[])) :- !.

trie([],_,[]) :- !.
trie([(B1,B2)|LB],B,[B1|L2]) :-
        pluspetit(B1,B),
        trie(LB,B,L2),
        !.
trie([_|LB],B,L2) :-
        trie(LB,B,L2),
        !.

pluspetit((D,_,_),(D1,_,_)) :-
        pltit(D,D1),
        !.

pltit([],[]) :- !.
pltit([(X,M1)|L1],[(X,M2)|L2]) :-
        opcmp(M1,M2),
        pltit(L1,L2),
        !.

opcmp(_,a) :- !.
opcmp(M1,M2) :-
        M1==M2,
        !.

lubb([],([],[],[])) :- !.
lubb([BPRIM],BPRIM) :- !.
lubb([B|LB],BPRIM) :-
        lubb(LB,BSTAR),
        lub(B,BSTAR,BPRIM),
        !.

/*****************************************************************/
/***                         VERIFIE                           ***/
/*****************************************************************/
```

```
verifie :-
        write('Source File ? : '),
        read(FILE),
        nl,
        exists(FILE),
        write('Procedure ? : '),
        read(PN),
        lirebeta(PN,B,PNORM,Q),
        norme(FILE,L),
        insere(L),
        iasb(B,PNORM,(D,SV,PS)),
        retire,
        nl,
        write('Les modes en sortie sont : '),
        replace(Q,D,DD),
        write(DD),
        nl,
        !.

lirebeta(PN,(D,SV,PS),PNORM,Q) :-
        PN=..[T|Q],
        nl,
        tested(Q,D,1),
        dd(D,QQ),
        PNORM=..[T|QQ],
        testesv(QQ,SV),
        dg(D,DG),
        diff(QQ,DG,QPS),
        testeps(QPS,PS),
        !.

tested([],[],_) :- !.
tested([X|Q],[(v(I),M)|D],I) :-
        moded(X,M),
        I1 is I+1,
        tested(Q,D,I1),
        !.

moded(X,f) :-
        var(X),
        !.
moded(X,g) :-
        nonvar(X),
        ground(X),
        !.
moded(X,a) :- !.

ground(X) :-
        atomic(X).
ground(X) :-
        X=..[F|L],
        grnd(L).

grnd([]) :- !.
grnd([X|L]) :-
        ground(X),
```

```
        grnd(L),
        !.

testesv([],[]) :- !.
testesv([X|Q],[[X]|SV]) :-
        testesv(Q,SV),
        !.

testeps([],[]) :- !.
testeps(L,[L]) :- !.

normepn(P,PN) :-
        P=..[X|Q],
        nvar(Q,V,1),
        PN=..[X|V],
        !.

nvar([],[],_) :- !.
nvar([X|Q],[v(I)|V],I) :-
        I1 is I+1,
        nvar(Q,V,I1),
        !.

replace([],[],[]) :- !.
replace([X|L],[(_,Z)|D],[(X,Z)|DD]) :-
        replace(L,D,DD),
        !.

insere([]) :- !.
insere([(H:-G)|L]) :-
        functor(H,T,N),
        assertclause(T,N,(H:-G)),
        insere(L),
        !.
insere([X|L]) :-
        functor(X,T,N),
        assertclause(T,N,X),
        insere(L),
        !.

assertclause(T,N,C) :-
        clause(T,N,M,_),
        M1 is M+1,
        asserta(clause(T,N,M1,C)),
        !.
assertclause(T,N,C) :-
        asserta(clause(T,N,1,C)),
        !.

retire :-
        retract(clause(_,_,_,_)),
        retire,
        !.
retire :- !.

/*****************************************************************/
```