Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

UNIVERSITÉ DE NAMUR

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Optimisation des programmes Prolog
une approche transformationnelle

Mairesse, Richard

Award date:
1988

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 26. Apr. 2024

Facultés Universitaires N.D. De La Paix
Namur
Institut d'informatique

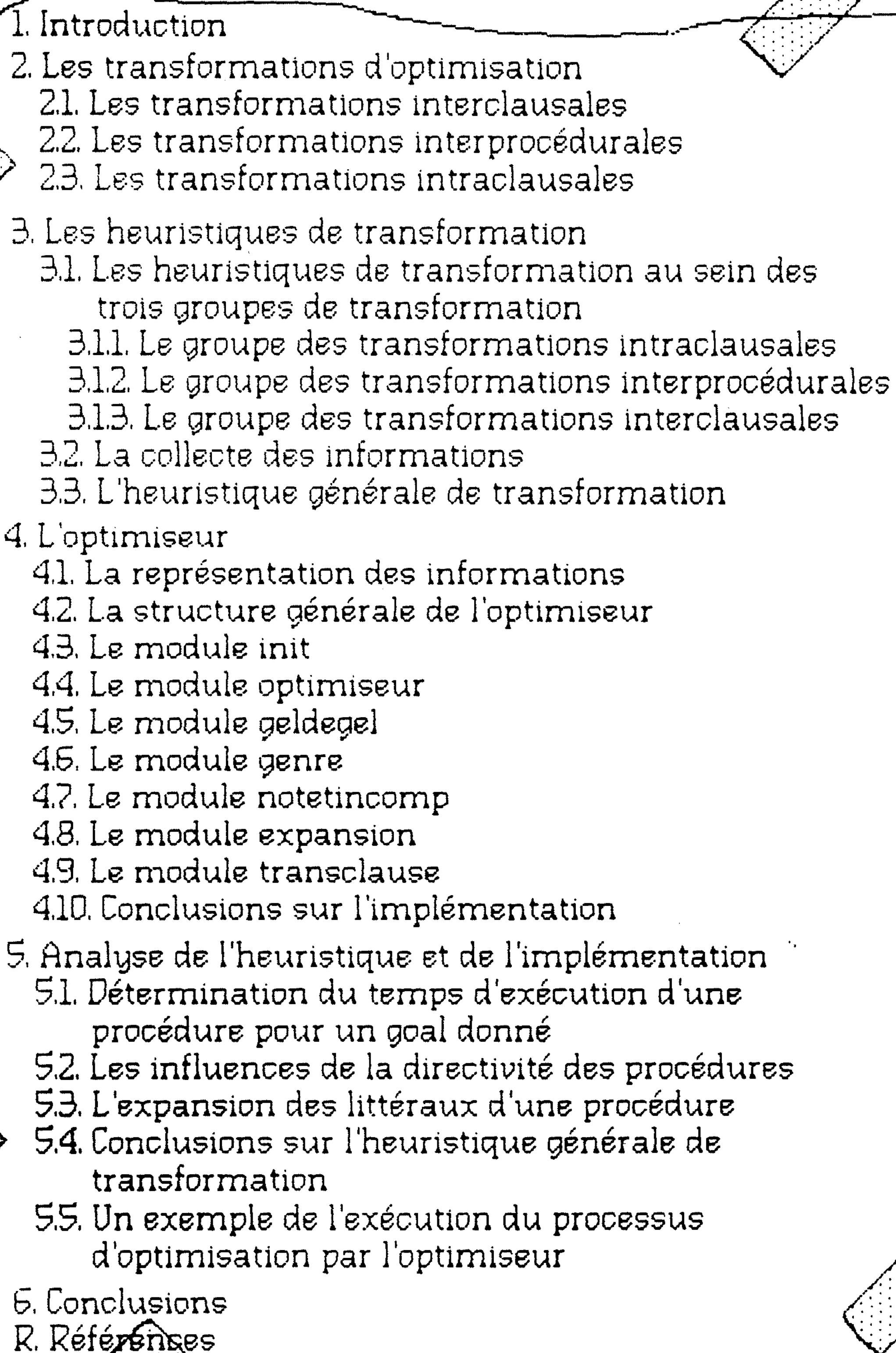
Optimisation des programmes Prolog: Une approche transformationnelle

Promoteur: monsieur B. Le Charlier

Année académique: 1987-1988

Mémoire présenté pour
l'obtention du grade de
licencié et maître
en sciences
avec option informatique
par

Mairesse Richard



1. Introduction
2. Les transformations d'optimisation
   2.1. Les transformations interclausales
   2.2. Les transformations interprocédurales
   2.3. Les transformations intraclausales
3. Les heuristiques de transformation
   3.1. Les heuristiques de transformation au sein des trois groupes de transformation
      3.1.1. Le groupe des transformations intraclausales
      3.1.2. Le groupe des transformations interprocédurales
      3.1.3. Le groupe des transformations interclausales
   3.2. La collecte des informations
   3.3. L'heuristique générale de transformation
4. L'optimiseur
   4.1. La représentation des informations
   4.2. La structure générale de l'optimiseur
   4.3. Le module init
   4.4. Le module optimiseur
   4.5. Le module geldegel
   4.6. Le module genre
   4.7. Le module notetincomp
   4.8. Le module expansion
   4.9. Le module transclause
   4.10. Conclusions sur l'implémentation
5. Analyse de l'heuristique et de l'implémentation
   5.1. Détermination du temps d'exécution d'une procédure pour un goal donné
   5.2. Les influences de la directivité des procédures
   5.3. L'expansion des littéraux d'une procédure
   5.4. Conclusions sur l'heuristique générale de transformation
   5.5. Un exemple de l'exécution du processus d'optimisation par l'optimiseur
6. Conclusions
R. Références

0
Avant-propos

Je remercie monsieur B. Le Charlier pour avoir accepté d'être mon promoteur de mémoire, pour son intérêt et sa disponibilité. Ce mémoire m'aura appris un tas de problèmes concernant la programmation logique et donné une bonne formation en programmation Prolog.

Je remercie monsieur V. Deville pour avoir dirigé mon mémoire et pour m'avoir éclairé de ses précieux conseils. La partie de sa thèse de doctorat ((1)) concernant le sujet du mémoire sera abondamment exploitée.

1
Introduction

Dans la démarche de conception d'un programme en Prolog, on s'intéresse d'abord à la logique du problème. Ensuite on traduit en Prolog la logique obtenue puis on optimise le code.

L'efficacité dépend d'abord de l'interpréteur Prolog. Quelles sont les stratégies auxiliaires à la résolution que l'interpréteur peut fournir ? Cette question pourrait peut-être dégager un interpréteur meilleur que tous les autres.

L'efficacité dépend bien entendu du programme lui-même. Existe-t-il un programme équivalent qui soit plus efficace ? L'efficacité est un espace à deux dimensions où les axes sont le temps d'exécution t et l'espace mémoire E d'un programme. L'espace mémoire reprend tout ce dont a besoin l'interpréteur pour exécuter le programme. Le mémoire propose une ballade dans cet espace dont le but est de réduire la coordonnée t du programme.

C'est une optimisation au niveau source d'un programme. Sur base de transformations d'optimisation, l'objectif est de fournir un programme équivalent dont le temps d'exécution est moins grand. L'optimisation consiste principalement à éliminer le plus de redondances possibles au sein du programme à analyser.

Le chapitre 2 du mémoire propose une bonne trentaine de transformations d'optimisation. Elles reprennent certaines habitudes des programmeurs Prolog ou des cas de figure moins évidents. Elles sont rangées dans trois classes suivant qu'elles s'appliquent sur une procédure, une clause ou sur une procédure mais mettant en jeu d'autres procédures du programme.

La complexité du problème d'optimisation réside dans le fait qu'une transformation peut inhiber d'autres transformations d'optimisation. La principale contribution du mémoire est de proposer, dans le chapitre 3, une heuristique d'application de ces transformations assurant qu'à partir d'un programme, un autre est généré, le plus optimisé qu'on puisse trouver à partir des transformations d'optimisation proposées. Dans le chapitre 3, nous montrons également la difficulté de collecter toutes les informations nécessaires au processus d'optimisation.

Le chapitre 4 propose un optimiseur, écrit en C-Prolog, qui est le reflet de l'heuristique de transformation trouvée au chapitre 3.

Ses performances, et donc celles de l'heuristique, figurent dans le chapitre 5. Elles montrent que l'optimisation au niveau source d'un programme Prolog est une réelle nécessité.

Nous pourrions imaginer ce processus d'optimisation complètement transparent au programme si l'optimiseur avait des outils à sa disposition pour effectuer la collecte des informations. C'est dans cet esprit qu'a été construit l'optimiseur. Il est capable de recevoir bien d'autres transformations d'optimisation et ses besoins en informations sont uniformisés au maximum.

2
Les transformations d'optimisation

Dans les transformations d'optimisation présentées dans ce chapitre, nous emploierons les différentes notions de déterminisme, d'incompatibilité... En voici les définitions précises tirées de ((1)).

Un littéral $p(t_1,...,t_n)$ est **déterministe** ssi la séquence des substitutions résultantes pour ce littéral a au plus une substitution résultante.

Un littéral $p(t_1,...,t_n)$ est **entièrement déterministe** ssi la séquence des substitutions résultantes pour ce littéral a une et une seule substitution résultante.

Un littéral $p(t_1,...,t_n)$ est **infini** ssi la séquence des substitutions pour ce littéral est infinie.

Un littéral $p(t_1,...,t_n)$ est **incompatible** avec le littéral $q(s_1,...,s_m)$ ssi la séquence des substitutions résultantes pour $q(s_1,...,s_m)$ est vide quand la séquence des substitutions pour $p(t_1,...,t_n)$ n'est pas vide et vice versa.

Nous utiliserons les notations traditionnelles pour décrire ces transformations.

$$\frac{p \leftarrow S}{p \leftarrow S'}$$

La clause, ou procédure, en dessous de la barre est la transformée de la clause, ou procédure, au dessus de la barre. P,Q,C seront des littéraux positifs et les lettres grasses **S** et **T** seront des séquences de littéraux (elles peuvent être éventuellement des séquences vides). $\underline{x}$ est le vecteur des variables de la tête d'une clause. Les autres conventions serons introduites en temps utiles.

Dans la référence ((2)), les transformations sont regroupées dans deux classes : les transformations interprocédurales et intraprocédurales. En un mot, les transformations interprocédurales remplacent un littéral dans une clause par une disjonction des clauses de définition de ce littéral; elles mettent donc en jeu plusieurs procédures. Sur base de cette disjonction, sont effectuées des transformations intraprocédurales pour la réduire (suppression de littéraux identiques...) ou des transformations plus générales telles que les unifications (appelées aussi les substitutions d'égalités)... Tout cela sous des conditions tantôt simples, tantôt abominables.

Comme nous le justifierons plus tard, nous pensons que les transformations interprocédurales doivent être ramenées à leur plus simple expression. De plus, si le traitement des disjonctions est en soi intéressant, il y a plus urgent. Par exemple, l'introduction de cuts.

Si le français le permet, nous divisons la classe des transformations intraprocédurales en deux classes : les transformations interclausales et intraclausales. Non seulement c'est plus joli mais notre optimiseur sera basé sur les trois classes de transformations :

les transformations interprocédurales,

les transformations interclausales, c'est-à-dire au sein de la procédure et mettant en jeu deux ou plusieurs clauses et

les transformations intraclausales, c'est-à-dire au sein d'une clause et mettant en jeu un ou plusieurs littéraux.

La preuve de la validité d'une transformation d'optimisation repose sur la sémantique procédurale car la transformation modifie le contrôle de l'interpréteur Prolog sous-jacent. Lors de cette validation, nous devrons tenir compte de la séquence des substitutions résultantes, des effets de bord, des métaprédicats et des cuts.

Pour exécuter un goal, l'interpréteur que nous utilisons recherche de haut en bas dans le programme une clause dont la tête s'unifie avec le goal. L'interpréteur recherche le plus général commun unificateur entre le goal et la tête des clauses. Si une telle clause est trouvée, il y a exécution des goals du corps de la clause de gauche à droite. Si l'interpréteur ne parvient pas à trouver une clause unifiable à un goal, il effectue un backtracking. Il rejette la clause la plus récente incluant toute substitution provenant de son unification. Ensuite il recherche une autre clause subséquente unifiable.

Chaque transformation sera présentée avec l'en-tête suivant: "Transformation X ((y))" où ((y)) sera le numéro de la référence dans

laquelle elle figure, à cela sera parfois ajouté "implémentée".

2.1 Les transformations interclausales

Soit une procédure à deux clauses. Si dans chaque clause il y a un littéral tel que les deux forment une paire de littéraux incompatibles alors il y a une possibilité d'effectuer une transformation d'optimisation.

Transformation 1 implémentée

$$\frac{\begin{array}{l} p(x) \leftarrow T, C_1, S_1 \\ p(x) \leftarrow T, C_2, S_2 \end{array}}{\begin{array}{l} p(x) \leftarrow T, C_1, !, S_1 \\ p(x) \leftarrow T, C_2, S_2 \end{array}}$$

Conditions d'application:

C_1 et C_2 incompatibles.

T, C_1 ,C_2 n'ont pas d'effets de bord.

T déterministe.

C_1 déterministe.

Preuve:

L'insertion du cut oblige la séquence **T** et le littéral C_1 à être déterministes sinon le cut supprimerait d'autres substitutions. D'autre part, le cut ne permettant pas le backtracking à travers **T** et C_1, ces objets ne doivent pas contenir d'effets de bord. (exemple: si **T** contient le prédicat retract/1 et si dans la séquence S_1, il y a un prédicat dont la procédure de définition contient la séquence " ! , fail ". En insérant le cut après C_1, on ne réalise plus la boucle à travers le prédicat retract/1).

Cette transformation est très importante, la présence du cut évite de nombreuses recherches inutiles. Nombreux sont ses cas particuliers. Dans le cas d'une séquence **T** vide nous avons la transformation 2.

<u>Transformation 2</u> ((1)) implémentée

$$\frac{\begin{array}{l} p(\underline{x}) \leftarrow C_1, S_1 \\ p(\underline{x}) \leftarrow C_2, S_2 \end{array}}{\begin{array}{l} p(\underline{x}) \leftarrow C_1, !, S_1 \\ p(\underline{x}) \leftarrow C_2, S_2 \end{array}}$$

<u>Conditions d'application:</u>

C_1 est déterministe.

C_1 et C_2 n'ont pas d'effets de bord.

C_1 et C_2 sont incompatibles.

<u>Preuve:</u>

C'est bien sûr le même raisonnement que pour la transformation 1. C_1 étant déterministe et C_1—C_2 étant une paire de littéraux incompatibles, le cut supprime et une recherche inutile à travers C_1 et une recherche à travers C_2 lorsque C_1 réussit.

Lorsque pour la paire C_1—C_2 de littéraux incompatibles nous avons la paire C—not(C), voici les transformations possibles immédiatement déductibles de la transformation 1.

<u>Transformation 3</u> ((1)) implémentée

$$\frac{\begin{array}{l} p(\underline{x}) \leftarrow C, S_1 \\ p(\underline{x}) \leftarrow \text{not}(C), S_2 \end{array}}{\begin{array}{l} p(\underline{x}) \leftarrow C, !, S_1 \\ p(\underline{x}) \leftarrow S_2 \end{array}}$$

<u>Condition d'application:</u>

C n'a pas d'effets de bord.

<u>Preuve:</u>

Du point de vue de la correction de la procédure, C doit toujours

être "ground" avant sa sélection et doit être déterministe. D'où la seule condition par rapport à la première transformation. Voici ce que nous gagnons: pour tout goal p(t) respectant les préconditions de p, si C réussit alors S_1 est exécutée. Par backtracking, C ne peut plus réussir puisqu'il est déterministe et d'autre part not(C) échoue. Inversément, si C échoue alors not(C) réussit et S_2 est exécutée. Par backtracking not(C) ne peut plus réussir. Avec l'insertion du cut et la suppression du littéral not(C) dans la seconde clause, il n'y a plus de backtracking à travers C ou une nouvelle exécution de C pour évaluer not(C).

De la même manière:

Transformation 4 ((1)) implémentée

$$\frac{\begin{array}{l} p(\underline{x}) \leftarrow not(C), S_1 \\ p(\underline{x}) \leftarrow C, S_2 \end{array}}{\begin{array}{l} p(\underline{x}) \leftarrow not(C), !, S_1 \\ p(\underline{x}) \leftarrow S_2 \end{array}}$$

Condition d'application:
C n'a pas d'effets de bord.

Les deux transformations qui suivent tiennent compte d'une séquence **T** non vide:

Transformation 5 ((1)) implémentée

$$\frac{\begin{array}{l} p(\underline{x}) \leftarrow T, C, S_1 \\ p(\underline{x}) \leftarrow T, not(C), S_2 \end{array}}{\begin{array}{l} p(\underline{x}) \leftarrow T, C, !, S_1 \\ p(\underline{x}) \leftarrow T, S_2 \end{array}}$$

Conditions d'application:
T et C n'ont pas d'effets de bord.
T est déterministe.

Transformation 6 ((1)) implémentée

$p(x) \leftarrow T, not(C), S_1$
$p(x) \leftarrow T, C, S_2$

$p(x) \leftarrow T, not(C), !, S_1$
$p(x) \leftarrow T, S_2$

Conditions d'application:

T et C n'ont pas d'effets de bord.
T est déterministe.

De nombreux exemples seront exposés plus loin.

Lorsqu'une procédure est construite sur un paramètre d'induction, la paire de littéraux incompatibles reflète les propriétés de ce paramètre; par exemple:

concat(L_1,L_2,L_3) :- **L_1=[]** , L_2 = L_3 .
concat(L_1,L_2,L_3) :- **L_1=[H|T]** , concat(T,L_2,T_1) , L_3=[H|T] .

---

concat(L_1,L_2,L_3) :- **L_1=[]** , ! , L_2 = L_3 .
concat(L_1,L_2,L_3) :- **L_1=[H|T]** , concat(T,L_2,T_1) , L_3=[H|T] .

L'induction porte sur l'aspect de L_1. Tout naturellement découle alors la généralisation de la transformation 1 pour une procédure à plus de deux clauses:

Transformation 7 ((1))

$$
\frac{\begin{array}{l}
p(\underline{x}) \leftarrow T, C_1, S_1 \\
p(\underline{x}) \leftarrow T, C_2, S_2 \\
\quad \cdot \qquad \cdot \\
p(\underline{x}) \leftarrow T, C_{n-1}, S_{n-1} \\
p(\underline{x}) \leftarrow T, C_n, S_n
\end{array}}{\begin{array}{l}
p(\underline{x}) \leftarrow T, C_1, !, S_1 \\
p(\underline{x}) \leftarrow T, C_2, !, S_2 \\
\quad \cdot \qquad \cdot \\
p(\underline{x}) \leftarrow T, C_{n-1}, !, S_{n-1} \\
p(\underline{x}) \leftarrow T, C_n, S_n
\end{array}}
$$

Conditions d'application:

Les C_1 sont déterministes

et n'ont pas d'effets de bord.

C_i est incompatible avec C_j ($j \neq i$).

T est déterministe.

Ou sous forme particulière,

Transformation 8 ((1))

$$
\frac{\begin{array}{l}
p(\underline{x}) \leftarrow T, C_1, S_1 \\
p(\underline{x}) \leftarrow T, C_2, S_2 \\
\quad \cdot \qquad \cdot \\
p(\underline{x}) \leftarrow T, C_{n-1}, S_{n-1} \\
p(\underline{x}) \leftarrow T, not(C_1), not(C_2), \ldots, not(C_{n-1}), S_n
\end{array}}{\begin{array}{l}
p(\underline{x}) \leftarrow T, C_1, !, S_1 \\
p(\underline{x}) \leftarrow T, C_2, !, S_2 \\
\quad \cdot \qquad \cdot \\
p(\underline{x}) \leftarrow T, C_{n-1}, !, S_{n-1} \\
p(\underline{x}) \leftarrow T, S_n
\end{array}}
$$

Conditions d'application:

Les C_i n'ont pas effets de bord.

C_i est incompatible avec C_j ($j \neq i$).

T est déterministe.

Preuve:

Par la présence des not/1, nous n'avons plus à exiger le déterminisme des littéraux C_i car, par la correction de p et des C_i, il est implicite. A part cela, la preuve est similaire à celle de la transformation 1.

Il existe bien sûr la correspondance des transformations 7 et 8 lorsque la séquence déterministe **T** est vide.

L'action des transformations 1 à 8 est d'éviter des backtrackings inutiles puisque leur issue est prévue vouée à l'échec. La difficulté est de trouver les ensembles de littéraux incompatibles. **Dans les transformations 1 à 8, il y a intérêt de placer les littéraux incompatibles le plus en avant dans les corps des clauses.** L'idéal est d'avoir des séquences **T** vides car la recherche à travers la procédure sera d'autant plus accélérée.

La transformation suivante évite à l'interpréteur d'effectuer deux fois le même travail par la présence de deux sous-arbres de recherche identiques. Elle s'applique sur deux clauses contiguës d'une procédure.

Transformation 9 ((1))

$$\frac{\begin{array}{l} p(\underline{x}) \leftarrow T, S_1 \\ p(\underline{x}) \leftarrow T, S_2 \end{array}}{\begin{array}{l} p(\underline{x}) \leftarrow T, p1(\underline{y}) \\ p1(\underline{y}) \leftarrow S_1 \\ p1(\underline{y}) \leftarrow S_2 \end{array}}$$

Conditions d'application:

$\underline{y}$ est le vecteur de toutes les variables se trouvant dans S_1 et S_2.

p n'a pas d'effets de bord.

T, S_1 et S_2 ne contiennent pas de cut.

p n'est pas infini.

Preuve:

Cette transformation modifie la séquence des substitutions résultantes. Si p est infinie, toutes les substitutions ne sont pas rendues par la procédure transformée. Par exemple, si S_1 n'est pas déterministe et si S_2 est une séquence infinie (c'est-à-dire qui contient au moins un littéral infini), après un échec de S_1, le backtracking passera à travers S_2 dans la procédure transformée au lieu de traverser la séquence **T** dans la procédure initiale. La séquence des substitutions due à la réalisation de la séquence S_1 sera perdue. D'où la condition: p n'est pas infini. Il faut aussi que p n'ait pas d'effets de bord. La troisième condition assure aussi qu'il n'y a pas de modifications de la séquence résultante en supprimant certaines alternatives.

Cette transformation est généralisable dans le nombre de clauses mises en jeu:

Transformation 10

$$\frac{\begin{array}{l} p(\underline{x}) \leftarrow T, S_1, S \\ p(\underline{x}) \leftarrow T, S_2, S \\ \quad . \qquad . \\ p(\underline{x}) \leftarrow T, S_n, S \end{array}}{\begin{array}{l} p(\underline{x}) \leftarrow T, pl(\underline{y}), S \\ pl(\underline{y}) \leftarrow S_1 \\ pl(\underline{y}) \leftarrow S_2 \\ \quad . \qquad . \\ pl(\underline{y}) \leftarrow S_n \end{array}}$$

Conditions d'application:

y est le vecteur de toutes les variables se trouvant dans S_1 et S_2, ..., S_n.

p n'a pas d'effets de bord.

T, S_1, S_2, ..., S_n et S ne contiennent pas de cut.

p n'est pas infini.

Les dix transformations précédentes élaguent l'arbre de recherche, associé à la procédure, des rejetons inutiles.

Un deuxième groupe de transformations interclausales peut être envisagée: des permutations de clauses. La transformation suivante tient plus du mot d'ordre:

Transformation 11 ((1))

Une procédure peut être correcte sous des permutations différentes de ses clauses. Choisir celle qui ordonne les clauses de la plus générale à la plus spécifique. Par plus générale, nous entendons celle qui donne des substitutions plus volontier que d'autres.

Le problème est de trouver un paramètre qui permette d'ordonner ces clauses. Un paramètre immédiat peut être le nombre de variables dans la tête d'une clause. La transformation suivante est basée sur le genre des clauses:

Une clause est de **genre SL** (straight-line) ssi elle ne contient pas de littéral récursif.

Une clause est de **genre TR** (tail recursive) ssi elle contient un seul littéral récursif et celui-ci est en fin de corps.

Une clause est de **genre GR** (general recursive) ssi elle n'est ni de genre SL, ni de genre TR.

Transformation 12 implémentée

Choisir une permutation permise des clauses d'une procédure de telle manière que la procédure ait l'aspect:

groupe de clauses de genre SL
groupe de clauses de genre TR
groupe de clauses de genre GR

Dans le cas d'anthologie ou la procédure se compose de clauses de genre SL et d'une seule clause de genre TR, l'efficacité gagnée par l'application de cette transformation correspond au passage d'une procédure récursive à une procédure itérative en Pascal (comparaison tirée de ((1))). De plus si cette clause de genre TR est du type

$$p(\underline{x}) :- S , p(\underline{y})$$

où S est une séquence déterministe, aucune information n'a besoin d'être retenue par l'interpréteur Prolog. Le cas de figure général, préconisé par la transformation 12 tient de l'heuristique suivante: "plus loin seront rejetés les littéraux récursifs dans une procédure, moins fastidieuse sera la gestion des appels récursifs". Les effets de cette transformation peuvent donc être discutés.

2.2 Les transformations interprocédurales

L'épine dorsale de l'optimiseur proposé par les auteurs de la référence ((2)) est l'expansion des littéraux d'une procédure.

Exemple:

```
reverse(X,Y) :- r(X,Y,[]) , ! .
r([],Z,Z).
r([H|T],W,Z) :- r(T,W,[H|Z]).
```

```
reverse(X,Y) :- r(X,Y,[]) , ! .
r([],Z,Z).
r([H|T],W,Z) :- r(T,W,[H|Z]) = r([],Z1,Z1) ;
                r(T,W,[H|Z]) = r([H1|T1],W1,Z1) , r(T1,W1,[H1|Z1]) .
```

Le littéral est remplacé par une disjonction comprenant autant de parties que de clauses composant la procédure de définition de ce littéral. Chaque partie comprend un littéral =/2 remplaçant l'unification du littéral avec la tête de clause candidate et d'une conjonction correspondant au corps de cette clause. L'efficacité exprimée en temps pour chaque littéral ainsi transformé correspond à la soustraction suivante:

temps de recherche d'une clause unifiable au littéral

—

temps de recherche dans la disjonction d'une unification réussie (succès de =/2)

Cette transformation d'expansion dépend d'une part du genre de la procédure à étendre (SL, TR, GR), pour chaque littéral à étendre, du genre de la procédure de définition de ce littéral et d'autre part, de la présence ou non de cuts dans cette procédure de définition (ceci n'est pas une liste exhaustive des conditions d'application !). Les auteurs de la référence ((2)) soulignent que leurs algorithmes d'application de cette transformation ne sont pas encore au point et font l'objet d'une recherche intense.

Remarquons dès maintenant que ce développement des littéraux n'est pas complet, ainsi le montre le dernier littéral de la dernière clause de la procédure r/3 de l'exemple.

Je suis convaincu que l'expansion des littéraux est hors de propos dans une optimisation au niveau source d'un programme Prolog. Le Prolog est un langage de haut niveau et le problème d'expansion doit

être plus un problème de compilation ou de design d'interpréteur qu'un problème posé au niveau source. Le prolog est basé entre autres sur la recherche de clauses unifiables; tel est le profil de Prolog. Ce n'est pas par les outils que propose Prolog que ce caractère de Prolog sera corrigé (même si c'est de manière partielle d'après la remarque ici plus haut). De plus, la transformation d'expansion coûte cher en validation des conditions d'application. Les programmes ainsi étendus deviendraient vite kilométriques. Ce qui paie réellement, c'est la recherche de littéraux incompatibles afin d'éviter des backtrackings inutiles par l'insertion de cuts comme expliqué aussi plus haut. Ceci est très important, nous assistons en direct à un changement de paradigme. Au lieu de retranscrire ici toutes les transformations que les auteurs de ((2)) présentent, nous ne prendrons que les cas plus simples, implémentés en vue de tests.

<u>Transformation 13</u> ((2)) implémentée

$p(\underline{x})$ — le littéral à étendre

$p(\underline{x}_1) \leftarrow S_1$

$\vdots \quad \vdots$

$p(\underline{x}_n) \leftarrow S_n$

la procédure de définition du littéral

$p(\underline{x}) = p(\underline{x}_1), S_1 ; ....; p(\underline{x}) = p(\underline{x}_n), S_n$

$p(\underline{x}_1) \leftarrow S_1$

$\vdots \quad \vdots$

$p(\underline{x}_n) \leftarrow S_n$

<u>Conditions d'application:</u>

1. Les S_i ne contiennent pas de cuts.
2. Il n'y a pas de variables communes entre les $p(\underline{x}_i)$, S_i, $p(\underline{x})$ et les autres parties de la clause contenant $p(\underline{x})$.
3. La procédure de définition du littéral à étendre est de genre SL.
4. Dans le terme disjonctif, lequel est généré en étendant chaque littéral, aucun nom de prédicat apparaît lequel a déjà été étendu, incluant le nom de prédicat p.
5. Pour le terme disjonctif résultant, répéter les opérations.

Preuve:

Les conditions 1 et 2 sont évidentes. D'une part, on ne perd pas de substitutions résultantes et d'autre part, aucune instanciation ne sera faite involontairement. La condition 3. assure que l'expansion d'une procédure de genre SL restera de genre SL. La condition 4. est une condition de terminaison pour l'expansion de la procédure, elle interdit une expansion circulaire.

Exemple:

Soit le programme

$$p :- p_1 , q , p_2 .$$
$$q :- q_1 , r , q_2 .$$
$$r :- r_1 , p , r_2 .$$

Chaque clause est de genre SL. Le littéral q dans la première clause est étendu par la deuxième clause donnant

$$p :- p_1 , q{=}q , q_1 , r , q_2 , p_2 .$$

Aucune autre expansion ne peut s'appliquer sur cette clause. La deuxième est étendue par la troisième, donnant

$$q :- q_1 , r{=}r , r_1 , p , r_2 , q_2 .$$

En étendant p, on obtient

$$q :- q_1 , r{=}r , r_1 , p{=}p , p_1 , q{=}q , q_1 , r , q_2 , p_2 , q_2 .$$

La dernière restera inchangée.

Les conditions d'application de la transformation 13 sont les plus restrictives de ((2)). Si la procédure à étendre est de genre TR ou GR, les auteurs de ((2)) proposent d'étendre une seule fois les littéraux récursifs (tel est le cas dans le premier exemple qui ne pourra donc pas être obtenu par la transformation 13). Ils proposent en outre d'accompagner cette transformation d'une décomposition de clause:

$$\frac{p :- S , (S_1 ; ... ; S_n) , T .}{\begin{array}{c} p :- S, S_1 , T . \\ \vdots \\ p :- S , S_n , T . \end{array}}$$

cela parce qu'ainsi décomposée, d'autres transformations sont applicables. Bien qu'implémentée, la décomposition va manifestement à l'encontre des efforts fournis pour réaliser l'expansion et aussi à

l'encontre des transformations 9 et 10. Elle ne reçoit donc pas le badge de "transformation".

<u>Exemple:</u>

Reprenons la procédure transformée du premier exemple et décomposons-la:

```
reverse(X,Y) :- r(X,Y,[]) , ! .
r([],Z,Z).
r([H],[H|Z],Z) .
r([H|[H1|T1]],W1,Z) :- r(T1,W1,[H1|[H|Z]]) .
```

Ceci en utilisant des transformations intraclausales. Les éléments de la liste sont traités deux par deux par r/3. Ce n'est pas mal mais le jeu en vaut-il la chandelle ? Si le littéral à étendre a comme procédure de définition une centaine de faits, que se passera-t-il après décomposition ? Une catastrophe ("le mont Fuji est proche du Capitole"), la décomposition va à l'encontre de la transformation 10. D'où le besoin que ce processus d'optimisation soit interactif ou que l'optimiseur devine la situation (il n'est pas bon que l'utilisateur ait le choix quant à l'utilisation de telle ou telle transformation, comme expliqué dans le chapitre suivant). La transformation suivante est un compromis:

<u>Transformation 14</u> ((1)) implémentée

$$
\begin{array}{l}
p(\underline{x}) \leftarrow S_1 \\
\quad\vdots \\
p(\underline{x}) \leftarrow S_{i1} \, , \, q(\underline{t}) \, , \, S_{i2} \\
\quad\vdots \\
p(\underline{x}) \leftarrow S_n \\
q(\underline{y}) \leftarrow T \\
\hline
p(\underline{x}) \leftarrow S_1 \\
\quad\vdots \\
p(\underline{x}) \leftarrow S_{i1} \, , \, \underline{y} = \underline{t} \, , \, T \, , \, S_{i2} \\
\quad\vdots \\
p(\underline{x}) \leftarrow S_n \\
q(\underline{y}) \leftarrow T
\end{array}
$$

<u>Conditions d'application:</u>

T ne contient pas de cuts ni de littéraux de foncteur p.

Il n'y a pas de variables communes entre $\underline{y}$, **T** et $\underline{x}$, $\underline{t}$, S_{i1}, S_{i2}.

2.3. Les transformations intraclausales

Transformation 15 ((1))

Une clause peut être correcte sous diverses permutations de ses littéraux. D'où l'idée de choisir celle qui ordonne les littéraux par ordre croissant de clauses unifiables.

Transformation 16 ((1))

Dans le choix des permutations permises, choisir celle qui amène les littéraux de type ground en tête de corps de clause.

Transformation 17 ((1))

Ou amener les littéraux déterministes en tête de corps de clauses.

Ces trois transformations peuvent sembler contradictoires, elles le seront certainement à l'application. Elles ont cependant le même but: tenter d'accélérer au mieux la résolution d'une séquence de littéraux, il s'agit de placer des littéraux en tête de corps de telle façon que le maximum de renseignements sur la résolution de la séquence soit obtenu. Ces trois transformations sont difficiles à appliquer. Le caractère déterministe d'un littéral est très intéressant.

Transformation 18 ((1)) implémentée

$p(\underline{x}) \leftarrow S_1$

$\vdots \qquad \vdots$

$p(\underline{x}) \leftarrow S_{n1}, S_{n2}$

$p(\underline{x}) \leftarrow S_1$

$\vdots \qquad \vdots$

$p(\underline{x}) \leftarrow S_{n1}, !, S_{n2}$

Conditions d'application:

S_{n1} n'a pas d'effets de bord.

S_{n1} est déterministe.

Preuve:

Puisque la séquence S_{n1} n'a pas d'effets de bord et est déterministe, le cut évite le backtracking inutile après la seule

substitution résultante. Cette séquence S_{n1} doit être la plus grande possible pour un maximum d'efficacité.

Transformation 19 ((1)) implémentée

$$\frac{\begin{array}{l} p(\underline{x}) \leftarrow S_1 \\ \vdots \\ p(\underline{x}) \leftarrow S_{i1}, !, S_{i2}, S_{i3} \\ \vdots \\ p(\underline{x}) \leftarrow S_n \end{array}}{\begin{array}{l} p(\underline{x}) \leftarrow S_1 \\ \vdots \\ p(\underline{x}) \leftarrow S_{i1}, !, S_{i2}, !, S_{i3} \\ \vdots \\ p(\underline{x}) \leftarrow S_n \end{array}}$$

Conditions d'application:

S_{i2} est déterministe.

S_{i2} n'a pas d'effets de bord.

Preuve:

Les transformations 18 et 19 sont en fait les mêmes où la qualité de dernière clause d'une procédure équivaut à un cut en tête de corps. De nouveau, inutile d'insérer un cut après chaque littéral déterministe, pour un maximum d'efficacité, il suffit de trouver la séquence S_{i2} la plus grande.

Transformation 20 ((1)) implémentée

$$\frac{p(\underline{x}) \leftarrow S_1, !, q(\underline{t}), S_2}{p(\underline{x}) \leftarrow S_1, q(\underline{t}), !, S_2}$$

Conditions d'application:

q n'a pas d'effets de bord.

$q(\underline{t})$ est entièrement déterministe.

Preuve:

q(t) est entièrement déterministe,la transformation évite donc le backtracking inutile à travers q si celui-ci n'a pas d'effets de bord. Un autre avantage est la permutation produite qui sera utile dans les substitutions d'égalités.

Pour terminer la série des transformations sur les cuts, voici la

Transformation 21 implémentée

$$\frac{p(\underline{x}) \leftarrow S_1 , ! , ! , S_2}{p(\underline{x}) \leftarrow S_1 , ! , S_2}$$

Son intérêt est bien sûr mineur mais rentre dans un cadre de réécriture ou de simplification dans l'optimiseur. Il y en aura d'autres.

Dans les permutations des littéraux d'une clause, outre les transformations 15, 16 et 17, il en existe une d'importance:

Transformation 22 ((1)) implémentée

$$\frac{p(\underline{x}) \leftarrow S_1}{p(\underline{x}) \leftarrow S_2}$$

avec S_2 ayant tous les littéraux récursifs à la fin.

Condition d'application:

S_2 est le résultat d'une permutation permise des littéraux de S_1.

Cette transformation doit être menée de front avec la transformation 12 afin d'obtenir, si possible, des procédures de genre TR.

Les quatre transformations suivantes traitent les égalités: ce sont les transformations par substitution d'égalités.

Transformation 23 ((1)) implémentée

$$\frac{p(\underline{x}) \leftarrow S_1 \,,\, Y{=}T \,,\, S_2}{p(\underline{x}) \leftarrow S_1 \,,\, Y{=}T \,,\, S_2\{t/Y\}}$$

Preuve:

Si Y=T réussit, Y est lié à T. Toute occurence de Y dans S_1 peut donc être remplacée par T.

Transformation 24 ((1)) implémentée

$$\frac{p(\underline{x}) \leftarrow Y{=}T \,,\, S}{p(\underline{x}\{t/Y\}) \leftarrow Y{=}T \,,\, S\{t/Y\}}$$

Pourquoi ne pas généraliser la transformation 23, substituer l'effet de l'égalité dans la séquence S_1 et dans la tête de clause ?

L'effet serait catastrophique, comme le montre l'exemple suivant. Soit la procédure

p(a,0) :- ! .
p(X,1) .

Suivant les notations bien pratiques de ((1)), cette procédure a les directivités

in(ground, var) : out(ground, ground) <1-1>
in(ground, ground) : out(ground, ground) <0-1>

Dans le premier cas, la procédure est entièrement déterministe, dans l'autre, elle est déterministe. Soit :- **p(a,V), V=1,** le goal à exécuter. La forme de p(a,V) suggère le comportement entièrement déterministe. p(a,V) réussit avec pour V la valeur 0. Par après, le goal échoue car V=1 n'est pas vérifié. Substituons l'égalité pour V dans p(a,V), cela donne le goal :- **p(a,1)** qui lui réussit ! La procédure a été employée sous des directivités différentes. Les goals :- **p(a,V),V=1** et :- **p(a,1)** sont donc différents ce qui justifie la transformation 23 qui pouvait paraître incomplète. La transformation suivante est évidente:

Transformation 25 ((1)) implémentée

$$\frac{p(\underline{x}) \leftarrow S_1 , Y{=}T , S_2}{p(\underline{x}) \leftarrow S_1 , S_2}$$

Condition d'application:

Y ne figure pas dans $\underline{x}$, S_1, S_2 et T.

Cette transformation réalise l' "occur check", celui-ci a été implémenté.

La transformation 26 transforme l'égalité entre deux termes de formes quelconques en une série d'égalités si elle existe sinon en résultat elle donne un littéral "fail".

Transformation 26 ((2)) implémentée

$$\frac{f(A_1, _, A_n) = f(B_1, _, B_n)}{A_1 = B_1, _ A_n = B_n} \quad \text{ou} \quad \frac{f(_) = f(_)}{\text{fail}}$$ (c-à-d ces deux termes ne sont pas unifiables)

Conditions d'application:

Les X_i sont les variables se trouvant dans l'égalité initiale et les T_i sont des termes.

Elle est particulièrement utile pour les transformations d'expansion parce qu'elle peut constituer un test d'unification et donc le cas échéant, en résultat de l'expansion d'un littéral, lui substituer le littéral "fail". La combinaison de ces quatre transformations permet de diminuer le nombre d'unifications dans une clause. Elles sont capables en fait de prendre en charge les types d'égalités suivants:

objets identiques	type 1
variable = variable	type 2
variable = terme	type 3
terme = variable	type 4
terme = terme	type 5

Des exemples illustrerons l'efficacité des substitutions d'égalités.

Les variables anonymes sont prises en charge de manière

différente par l'interpréteur Prolog. Etant donné qu'elles n'apparaissent qu'une seule fois dans une clause, leurs valeurs n'ont pas d'importance. Le processus d'unification est donc plus simple.

Transformation 27 ((1)) implémentée

$$\frac{p(\underline{x}) \leftarrow S}{p(\underline{x}\{_/Y\}) \leftarrow S\{_/Z\}}$$

Conditions d'application:

Y et Z n'apparaissent qu'une seule fois dans la clause (deux variables anonymes sont deux variables distinctes).

Cette transformation provoque une légère amélioration de l'efficacité de la procédure.

Les deux transformations suivantes sont des transformations du même type que celle de numéro 21.

Transformation 28 ((2)) implémentée

$$\frac{p(\underline{x}) \leftarrow S_1 , \text{true} , S_2}{p(\underline{x}) \leftarrow S_1 , S_2}$$

Transformation 29 ((2)) implémentée

$$\frac{p(\underline{x}) \leftarrow S_1 , \text{fail} , S_2}{p(\underline{x}) \leftarrow S_1 , \text{fail}}$$

Celles qui suivent peuvent être d'application lors de l'expansion des littéraux et sont données sans preuve.

Transformation 30 ((2))

$$\frac{p(\underline{x}) \leftarrow S_1 \; ; \; \text{fail} \; ; S_2}{p(\underline{x}) \leftarrow S_1 \; ; S_2}$$

La transformation suivante effectue une mise en évidence de littéraux.

Transformation 31 ((2))

$$\frac{p(\underline{x}) \leftarrow S \, , S_1 \, , T \, ; S \, , S_2 \, , T}{p(\underline{x}) \leftarrow S \, , (S_1 \, ; S_2) \, , T}$$

Conditions d'application:

S et **T** sont sans effets de bord.

Cette transformation ressemble beaucoup à la transformation 10.

La dernière transformation propose de regrouper les égalités qui subsistent malgré nos efforts pour les éliminer, les annihiler. Elle s'appelle, suivant ((2)): "intégration de littéraux".

Transformation 32 ((2))

$$\frac{X_1 = T_1, \ldots X_n = T_n}{f(X_1, \ldots, X_n) = f(T_1, \ldots, T_n)}$$

Conditions d'application:

f est un symbole de fonction approprié.

3
Les heuristiques de transformation

Une transformation est **applicable** ssi le programme Prolog à transformer remplit les conditions d'application de cette transformation.

Une transformation T_1 **inhibe** une transformation T_2 ssi l'application de T_1 rend T_2 inapplicable.

La forme souhaitée pour les procédures à optimiser est la présence de variables dans la tête des clauses, et uniquement des variables (pas de termes et de constantes). C'est une précondition essentielle pour le programme à optimiser sinon, comme nous allons le voir, beaucoup de transformations ne seront pas applicables.

Soit la procédure p à transformer

```
p :- a , b , c , d .
p :- a , b , not(c) , e .
```

avec a, b, c déterministes et sans effets de bord, p non infini.

Deux transformations sont applicables: la 5 et la 9.

Par 5, p est transformé en

```
p :- a , b , c , ! , d .        (1)
p :- a , b , e .
```

Par la présence du cut, la 9 n'est plus applicable. Par 9,

```
p :- a , b , pl .
pl :- c , d .
pl :- not(c) , e .
```

et, de nouveau par 3 sur pl

```
p :- a , b , pl .
pl :- c , ! , d .               (2)
pl :- e .
```

D'une manière générale, il semble que la transformée (2) soit plus intéressante que (1). Mais si la procédure initiale est correcte pour la

permutation [3, 1, 2, 4] pour la première clause (une permutation de littéraux est décrite par une liste d'entiers où chaque entier identifie un littéral et a pour valeur la position de ce littéral dans le corps de la clause initiale. La clause initiale est donc le résultat de la permutation identique [1, 2, 3, 4]) et [3, 1, 2, 4] pour la seconde, on obtient le schéma de transformation suivant:

```
p :- c , a , b , d .
p :- not(c) , a , b , e .
________________________          (3)
p :- c , ! , a , b , d .
p :- a , b , e .
```

par 5. (3) est donc plus intéressant que (2).

L'objet de ce chapitre est de trouver un algorithme d'application des transformations qui assure un agencement harmonieux entre les transformations qui tienne compte des inhibitions et de leurs opportunités d'application. Cet algorithme est aussi appelé **heuristique de transformation.**

3.1. Les heuristiques de transformation au sein des trois groupes de transformations

3.1.1. Le groupe des transformations intraclausales

Au sein du groupe des transformations intraclausales, nous pensons qu'il n'existe aucune inhibition. Bien mieux, ces transformations sont complémentaires (exemple: les transformations 23, 24, 25, 26 et 20). L'heuristique de transformation dans ce groupe se résume au processus itératif de détection-application de ces transformations jusqu'à ce qu'il n'y ait plus de transformation applicable. Exprimons ce fait à l'aide de la fonction **appliquer**.

appliquer(L,P) est une fonction qui exécute toutes les transformations applicables parmi celles contenues dans la liste L sur le programme P. Le résultat de cette fonction est la transformée de P. Lorsqu'aucune transformation n'est applicable, le programme P n'ayant pas été transformé, le résultat de la fonction est simplement P. La liste L ne contient que des transformations qui ne s'inhibent pas. Il s'agit d'un processus itératif car une transformation peut lever l'inhibition d'une autre transformation (exemple: la 20 peut lever l'inhibition de la 24). Puisque nous allons donner un code en pseudo-langage (hybride provenant du C, du Pascal et de bien d'autres) de cette fonction, L sera une liste d'entiers où chaque entier sera le numéro de la transformation à appliquer (la transformation 5 aura l'entier 5 dans L).

```
appliquer([T1, ..., Tn],P0) : P1
{  P1 = P0;
   L = L1 = [T1, ..., Tn];
   while (L1 ≠ ∅)
   {  T = premier(L1);
      if cond(T,P0,C) then {
                              P1 = T(T,P1,C);
                              L1 = L;
                           }
                      else {
                              L1 = L1 \ T;
                           }
   }
}
```

cond(T,P,C) est une fonction booléenne qui prend la valeur "true" si les conditions d'application de la transformation T dans le programme P sont remplies. **C** est la liste, en résultat, des conditions d'application de T exprimées dans ce cas présent

T(T,P,C) est une fonction qui applique la transformation T sur le programme P ayant les conditions d'application C.

Cette fonction **appliquer** est, je trouve, un moyen pratique pour exprimer le fait que l'on essaie d'appliquer un groupe de transformations non inhibitrices entre elles, cela jusqu'à ce qu'il n'y ait plus moyen de les appliquer. Elle évitera de nombreuses redites, sera commode pour expliquer les heuristiques et sera un outil précieux pour les spécifications des procédures de l'optimiseur.

Voici l'heuristique de transformation au sein du groupe des transformations intraclausales.

Heuristique H_1

précondition: P_1 = clause à transformer

P_2=appliquer([18,19,20,21,23,24,25,26,28,29],P_1)

P_3=appliquer([34],P_2)

P_4=appliquer([27],P_3)

postcondition: P_4 est la transformée de P_1 par les transformations intraclausales applicables.

Justification:

Les transformations 34 et 27, par leur nature, ne doivent être appliquées que une seule fois. La transformation 34 est appliquée nécessairement en fin d'heuristique. Celle de numéro 27 est aussi à la fin puisqu'il suffit alors de l'effectuer seulement une fois. Les autres transformations sont appliquées sans préséance aucune. Les transformations traitant les disjonctions ne sont pas prises en compte.

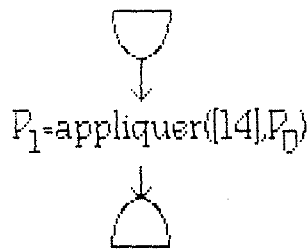
H_1 assure donc un maximum d'efficacité dans l'optimisation d'une clause.

3.1.2. Le groupe des transformations interprocédurales

D'après les raisons évoquées dans la partie 2.2., les transformations interprocédurales se ramènent à la transformation 14.

<u>Heuristique H_2</u>

précondition: P_0 = procédure à étendre



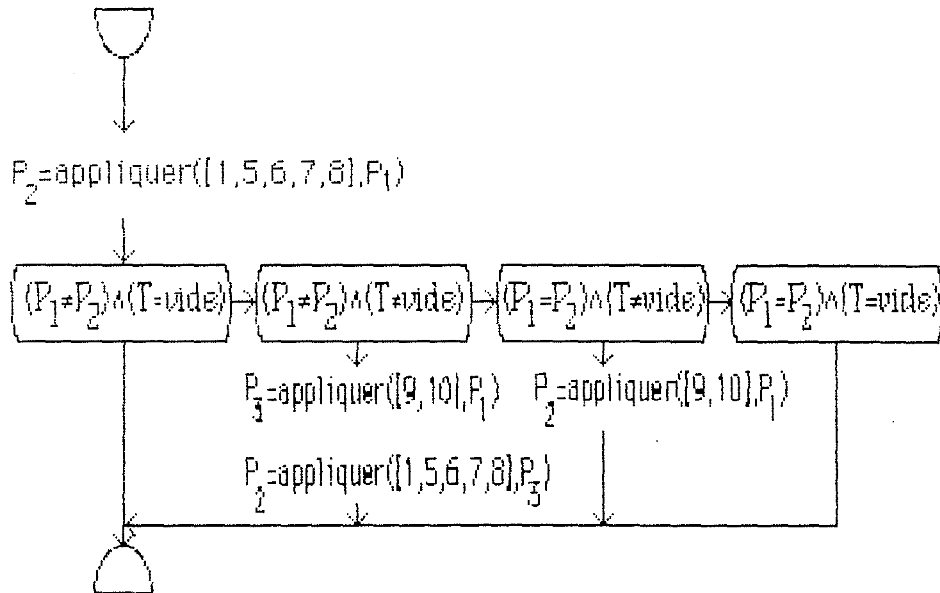
postcondition: P_1 est la transformée de P_0 par l'expansion de ses littéraux.

3.1.3. Le groupe des transformations interclausales

Dans ce groupe, les transformations 1, 2, 3, 4, 5, 6, 7 et 8 inhibent les transformations 9 et 10 par l'insertion d'un cut. Il faut donc trouver un agencement entre ces deux classes de transformations. Les transformées (1), (2) et (3) de la procédure initiale p suggère l'heuristique suivante: (La condition (T = vide) veut dire que dans la procédure analysée, il n'y a aucun préfixe commun autre que celui égal au vide entre les clauses (en rapport avec la notation adoptée au chapitre 2))

Heuristique H_3

précondition: P_1 = procédure à transformer



postcondition: P_2 est la transformée de P_1 par les transformations interclausales applicables.

Justification:

Comme nous l'avons dit dans le chapitre 2, il est préférable d'avoir une séquence **T** vide. Les transformations 1, 5, 6, 7 et 8 sont appliquées ensembles, elles sont les généralisations de 2, 3 et 4. Si l'une de ces transformations est appliquée ($P_2 \neq P_1$) et si **T** n'est pas vide, les transformations 9 ou 10 sont appliquées pour supprimer cette séquence **T** puis de nouveau celles de numéro 1, 5, 6, 7 et 8.

H_3 assure donc un maximum d'efficacité dans l'optimisation d'une procédure.

Remarques générales sur les heuristiques H_1, H_2 et H_3:

Elles ne sont pas des routes proposées à un utilisateur pour simplifier ses procédures mais un mécanisme qui, à l'aide de certaines informations, effectue sans interruption le processus d'optimisation. De ce point de vue, le sénario est complètement différent de celui proposé dans ((2)) où le choix est donné à l'utilisateur quant à l'application de telle ou telle transformation. En fait, le problème sous-jacent est la collecte des informations sur chaque procédure, sur chaque clause et sur chaque littéral composant le programme Prolog à analyser. Ce besoin d'informations a son incidence sur l'heuristique générale qui va être exposée.

3.2. La collecte des informations

Une information capitale sur un littéral est son degré de déterminisme. Est-il déterministe, entièrement déterministe ou non déterministe ?

Il est évident que le Prolog est un langage apte à exprimer tout algorithme (on peut reproduire en Prolog les instructions traditionnelles du type "while", ":=",...). D'après la théorie de la calculabilité, il ne peut pas exister d'algorithme capable de déterminer, pour tout programme P et toute donnée X si l'exécution de ce programme avec cette donnée produira ou ne produira pas de résultat (théorie de la calculabilité, H. Leroy, cours fndp). Nous voyons poindre avec regret la thèse suivante:

Il n'existe pas d'algorithme pour décider si, pour tout littéral, il est déterministe ou non.

Preuve: (tirée de ((2)))

Supposons qu'il existe un algorithme det(p) tel que
det(p) réussit si p, un littéral, est déterministe.
det(p) échoue si p n'est pas déterministe.

Soit maintenant la procédure q

```
q :- det(q) .
q.
```

Supposons que le goal :- **det(q)** réussisse. Alors q n'est pas déterministe puisque le goal :- **q** réussit par unification et résolution avec la première clause et réussit de nouveau, par backtracking, par unification et résolution avec la seconde clause.

Supposons que le goal :- **det(q)** échoue. Alors q est déterministe puisque le goal :- **q** réussit par unification et résolution avec la seconde clause.

Nous obtenons deux contradictions (la seconde n'étant pas nécessaire).

c.q.f.d.

Dans le sillage de ce théorème, traine, tels des marsouins, une nichée de corollaires; en particulier pour le caractère infini d'un littéral.

Donc, il faut se tourner vers un algorithme forcément incomplet. C. S. Mellish, dans son article "Some global optimizations for a Prolog compiler", en propose un basé sur la définition suivante:

Un prédicat est déterministe si
chaque clause, à part la dernière, contient un cut comme conséquent
chaque littéral, se situant avant un cut dans une clause est lui-même déterministe.

Cette définition est la plus faible qu'on puisse trouver. Sans cut, pas moyen de savoir si un prédicat est déterministe. Mellish ajoute qu'avec les déclarations de modes, il y a moyen d'étoffer cette définition. J'ai regardé les articles en rapport et décidai que l'information concernant le déterminisme des prédicats est donnée par l'utilisateur.

Une autre information de première importance concerne les permutations correctes de littéraux dans une clause ou de clauses dans une procédure. Là, je n'ai même pas cherché d'articles en rapport. Cette information a une incidence sur l'heuristique globale de transformation. En effet, supposons donnée une liste des permutations permises des littéraux d'une clause. Appliquons sur cette clause une ou plusieurs transformations intraclausales. L'information est perdue et dès lors, de nouveau demandée. Ce n'est pas nécessaire d'effectuer les transformations intraclausales en premier. Ce qu'il y a à faire, c'est d'abord appliquer les transformations interclausales. Elles auront à leur disposition cette liste de permutations.

Savoir si un littéral a des effets de bord est aussi de premier chef. L'algorithme permettant de répondre à cette question est facilement réalisable mais coûte les yeux de la tête en comparaisons de toutes sortes. Cette information sera considérée comme donnée.

Que conclure de tout ceci?

Il est paradoxal de devoir confier à des logiciels le soin de deviner les caractéristiques d'un programme. Une spécification complète (comme le modèle proposé dans ((1))) contient la plupart de ces renseignements. Ceci bien sûr dans le cadre de l'optimiseur. Je suis partisan d'une sorte de déclaration qui reprendrait ces informations.

En ce qui concerne l'heuristique de transformation, les transformations interclausales doivent être appliquées avant les transformations intraclausales sinon il y a gaspillage d'informations. Mais ce n'est pas la seule raison. Il ne faut pas oublier que la première information reste le programme à optimiser. Sa forme peut mieux se prêter au processus d'optimisation.

Supposons que l'on ait la procédure (tirée de ((1)))

```
efface(X,L,Leff) :- L = [ H | T ] , H = X , Leff = T .                     (1)
efface(X,L,Leff) :- L = [ H | T ] , not(H = X) , Leff = [ H | Teff ] , efface(X,T,Teff) .
```

transformée par 5 en la procédure

```
efface(X,L,Leff) :- L = [ H | T ] , H = X , ! , Leff = T .
efface(X,L,Leff) :- L = [ H | T ] , Leff = [ H | Teff ] , efface(X,T,Teff) .
```

puis, par les transformations intraclausales, on obtient

```
efface(H,[ H | T ],T) :- ! .                                               (2)
efface(X,[ H | T ],[ H | Teff ]) :- efface(X,T,Teff)                        .
```

ceci pour une directivité
in(ground, ground, ground) : out(ground, ground, ground).

Supposons qu'à la place de (1), on ait la procédure équivalente

```
efface(H,[ H | T ],T) .                                                    (3)
efface(X,[ H | T ],Leff) :- not(H = X) , Leff = [ H | Teff ] , efface(X,T,Teff) .
```

alors il est impossible d'insérer un cut et de réduire efface/3 pour obtenir un résultat analogue à (2). La forme de (3) interdit d'office l'application de la plupart des transformations interclausales.

La forme souhaitée pour les procédures à optimiser est la présence de variables dans la tête des clauses, et uniquement des variables (pas de termes et de constantes),

quitte à avoir une série d'unifications en corps de clause comme c'est le cas pour (1). De plus, cette façon de faire rentre parfaitement dans la méthodologie de programmation proposée par ((1)). De ces consignes ressort le fait qu'il ne faut pas appliquer le groupe des transformations intraclausales avant celui des interclausales car (3) est simplement la transformée de (1) par les transformations intraclausales.

3.3. L'heuristique générale de transformation

Voici l'heuristique générale de transformation que je propose:

Heuristique H_4

précondition: P_1 = procédure à transformer

P_2=goalperm(P_1)

P_3=H_3(P_2)

P_4=H_2(P_3)

P_5=H_1(P_4)

postcondition: P_5 est la transformée de P_1 par les transformations d'optimisation applicables.

Justification:

goalperm(P) est une fonction qui trouve des permutations de littéraux de telle manière qu'il n'y ait qu'une séquence **T** minime avant les paires de littéraux incompatibles. En second lieu, elle tente de trouver des permutations de littéraux et de clauses qui sont les effets des transformations 12 et 22. De cette manière, nous privilégions la recherche des paires de littéraux incompatibles au détriment des transformations 12 et 22.

Cet algorithme est, comme H_1 ,H_2 et H_3, parfaitement implémentable, mais il n'est absolument pas optimisé. D'abord quelques remarques sur sa construction. Dans la précondition, il est spécifié que H_4 est à appliquer sur une procédure, ce qui assure la validité de l'application de H_3 car il y a un test sur la séquence **T**. Pour H_2 et H_1, en précondition, P_1 peut être un programme puisque la fonction **appliquer** cherche par l'intermédiaire de la fonction **cond** l'endroit où appliquer telle ou telle transformation. Cela évite des boucles

sur les procédures (pour H_2) et sur les clauses (pour H_1) dans l'heuristique H_4.

H_4 est donc un algorithme lourd mais son intérêt réside dans le fait qu'il exprime très bien la manière dont chaque transformation s'applique et la préséance à adopter pour optimiser au mieux un programme. Là est le but de l'heuristique H_4 et nulle part ailleurs. Lorsqu'un problème n'est pas trivial, il y a souvent moyen de trouver un algorithme plus optimisé qu'un précédent optimisé etc. Il s'agit alors de montrer l'équivalence entre ces différents algorithmes pour s'assurer qu'ils font réellement la même chose que H_4.

L'optimiseur que nous proposons est équivalent à H_4 compte tenu des transformations qui n'ont pas été implémentées.

Le chapitre suivant présente l'optimiseur implémenté.

4
L'optimiseur

Dans ce chapitre, l'implémentation d'un optimiseur est présentée. Celui-ci est programmé en **C-Prolog version 1.5** sur **Vax-750**. Son code est reproduit en annexe. Il y a d'abord le problème de

4.1 La représentation des informations

Dans le chapitre trois, nous avons mentionné le problème crucial de la collecte des informations. Le fichier où se trouve le programme à optimiser sera appelé **fichier source**. Dans celui-ci, nous exigeons la présence de faits predicat/8 analogues à celui qui suit pour chaque procédure du programme:

**predicat('nom','arite','deter','effet_de_bord','infini','genre',
'nbre_de_clauses','liste_c_clauses')**

où 'nom' = nom de la procédure (ou du prédicat)
'arite' = arité du prédicat
'deter' = 0 = non déterministe
1 = déterministe
2 = entièrement déterministe
'effet_de_bord' = 0 = il n'y en a pas dans cette procédure
1 = il y en a
'infini' = 0 si le prédicat n'est pas infini
1 sinon
'genre' = 0 si la procédure est de genre SL
1 de genre TR
2 de genre GR
'nbre_de_clauses' = entier égal au nombre de clauses dans la procédure
'liste_c_clauses' = liste de couples (Cl, N) où Cl est une clauses de la procédure de nom 'nom' et N est un entier égal au nombre de variables distinctes dans Cl (cf 5.5).

Les entités suivantes **doivent être instanciées**: 'nom', 'arite', 'deter', 'effet_de_bord', 'infini'.

'genre' est trouvé par l'optimiseur ainsi que 'nbre_de_clauses' instancié au départ à 0. 'liste_c_clauses' est construite par l'optimiseur et instanciée au départ à la liste vide.

Pour le programme lui-même, voici la condition déjà citée assurant une optimisation maximum:

la forme souhaitée pour les procédures à optimiser est la présence de variables dans la tête des clauses, et uniquement des variables (pas de termes et de constantes).

Voici un fichier source conforme aux préconditions du processus d'optimisation:

```
****************************************************************
predicat(efface,3,1,0,0,6,0,[]).

efface(H,L,Leff):-
        L=[H|T],H=H,Leff=T.
efface(H,L,Leff):-
        L=[H|T],not(H=H),Leff=[H|Teff],efface(H,T,Teff).
****************************************************************
```

(Les autres informations nécessaires seront demandées en temps voulu par l'optimiseur: 'H=H' est-il déterministe ? ...)

4.2 La structure générale de l'optimiseur

L'optimiseur est un ensemble de sept fichiers **C-Prolog**. Ces fichiers, ou modules, correspondent à une étape précise dans le processus d'optimisation.

Structure de l'optimiseur

Module	Actions (en fonction du temps)
optimiseur	chargement du fichier source **C-Prolog** contenant le code du programme à optimiser
genre	permutation des littéraux récursifs, permutation des clauses récursives, détermination du genre de la procédure
notetincomp	insertion de cuts par les transformations applicables sur des paires de littéraux incompatibles
expansion	expansion des littéraux
transclause	transformations intraclausales applicables
geldegel	détection des variables anonymes
optimiseur	remplissage du fichier de sortie comprenant le code du programme optimisé

Les entrées et les sorties sont disséminées dans le programme et aisément repérables car le nom des procédures en rapport commence toujours par le mot **oracle**. Pour faciliter la lecture du programme, les procédures sont rangées par ordre d'apparition au sein d'un module. Dans ce chapitre, chaque module est présenté de manière chronologique. Il s'agit donc de la présentation d'un sénario mais nous pensons que les modules sont relativement indépendants et du mode de représentation (predicat/8) et de la séquence des événements.

Chaque module est construit suivant une approche "top-down" du problème. Ainsi, chaque procédure aura à sa suite ses procédures composantes.

4.3. Le fichier (ou module) init

Sa seule tâche est de permettre le chargement des six fichiers mentionnés dans le tableau précédent. D'un point de vue technique, lorsque l'interpréteur **C-Prolog** nous invite à introduire une commande, il suffit de consulter le fichier **init** pour que tout l'optimiseur soit chargé.

4.4. Le module optimiseur

Ce module est composé de trois parties: le coordinateur de tâches (la procédure optimiseur/2), le chargement du fichier source à analyser (la procédure chargement/1) et le remplissage du fichier source qui contiendra le programme optimisé (la procédure remplissage/1).

Procédure optimiseur(In,Out)

In est un nom de fichier source existant conforme aux préconditions citées dans 4.1.

Out est un nom de fichier source non existant.

Cette procédure construit le fichier source de nom Out qui contiendra la transformée de celui contenu dans In suivant l'heuristique générale de transformation.

Conditions d'application:

in(*ground*, *ground*) : *out*(*ground*, *ground*) <1-1>

Les préconditions concernant les noms de fichier peuvent être affaiblies puisque les procédures suivantes sont implémentées:

Procédure verifin(In1,In2)

In1 est un nom de fichier

In2 est un nom de fichier existant

Cette procédure détermine si In1 est un nom de fichier existant. Si oui, In2 = In1 sinon In2 est un nom de fichier existant donné par l'utilisateur.

Conditions d'application:

in(*ground*, *var*) : out(*ground*, *ground*) <1-1>

Procédure verifout(Out1,Out2)

Out1 est un nom de fichier.

Out2 est un nom de fichier non existant.

Cette procédure détermine si Out1 est un nom de fichier non existant. Si oui, Out2 = Out1 sinon Out2 est un nom de fichier non existant donné par l'utilisateur.

Conditions d'application:

in(*ground*, *var*) : out(*ground*, *ground*) <1-1>

Les petites procédures composant ces deux procédures sont trop simples pour mériter une spécification.

La procédure oracle_oui_non/2 est un moyen pratique pour obtenir une décision de l'utilisateur:

Procédure oracle_oui_non(Message,Rep)

Message est un terme.

Rep appartient à { 0, 1}.

Cette procédure, après avoir affiché 'Message' demande à l'utilisateur 'oui ou non ? '. Si l'utilisateur répond en introduisant un terme commençant par o ou y alors Rep = 1 sinon Rep = 0.

Conditions d'application:

in(ground, any) : out(ground, ground) <0-1>

Procédure chargement(In)

In est le nom d'un fichier source existant qui contient un programme conforme aux préconditions établies dans 5.1.

Cette procédure lit les termes contenus dans le fichier source de nom In de telle manière que tout fait predicat/8 dans In se rapportant à une procédure de nom 'nom' de In, soit inclus dans la base de donnée de référence de l'interpréteur **C-Prolog**. Cette procédure détermine le nombre de clauses 'nbre_de_clauses' et construit 'liste_c_clauses' de la façon suivante:

$$[(Cl_n, _), ..., (Cl_2, _), (Cl_1, _)]$$

où Cl_1, Cl_2, ..., Cl_n sont les clauses lues dans In de la procédure de nom 'nom' prises dans l'ordre lexicographique et les _ , des variables anonymes.

Conditions d'application:

in(ground) : out(ground) <1-1>

Les préconditions concernant le fichier In peuvent être affaiblies car, en cas d'absence de fait predicat/8 pour une procédure, oracle5/2 et consorts se chargent de demander à l'utilisateur les informations manquantes. Il existe dès lors deux courants d'entrée: le fichier In et la console du terminal. Cela justifie la présence des procédures incrcompt/0 et itread/1. La première incrémente un compteur à chaque lecture d'un terme dans le fichier In. La seconde effectue une itération comprenant la lecture d'un terme dans le fichier In et la décrémentation du compteur. Elle s'arrête quand le compteur est nul.

Ces deux procédures servent à retenir le bon terme à lire dans le fichier In car à chaque fermeture du fichier In pour donner la main à la console du terminal puis réouverture du fichier In, red/1 relit le premier terme dans In.

La procédure remplissage/1 s'occupe de remplir un fichier source de nom donné du programme optimisé.

Procédure remplissage(Out)

Out est un nom de fichier non existant.

Cette procédure enlève tout fait predicat/8 de la base de donnée de référence de l'interpréteur **C-Prolog**. Soit

predicat(_,_,_,_,_,_,_, [(Cl_1, _), (Cl_2, _), ..., (Cl_n, _)])

où Cl_1, Cl_2, ..., Cl_n sont les clauses formant la procédure optimisée. remplissage/1 crée le fichier de nom Out et écrit dedans ces clauses dans l'ordre d'apparition dans la liste après les avoir dégelées (définition dans 5.5.) et leur avoir appliqué la transformation 2.7.

Conditions d'application:

in(ground) : out(ground) <1-1>

4.5. Le module geldegel

Dans ce module se trouvent d'abord les procédures bien connues concatenate/3 et member/2.

Procédure constant(X)

X est un terme.

Cette procédure détermine si X est un atome ou un nombre entier.

Conditions d'application:

in(ground) : out(ground) <0-1>

Procédure compound(X)

X est un terme.

Cette procédure détermine si X est un terme *composé*, c'est-à-dire ni une variable ni un atome ou un nombre entier.

Conditions d'application:

in(ground) : out(ground) <0-1>

Comme le problème de l'optimisation est un problème de traitement de clauses, il faut trouver un moyen simple pour les manipuler sans effectuer d'instanciations involontaires. Nous allons utiliser un vieux truc de programmeur (expression consacrée).

Un terme a ses variables **gelées** ssi elles sont instanciées à des constantes de format déterminé.

Un terme a ses variables **dégelées** ssi elles n'ont plus de liaisons avec leur instances.

Le truc est de geler les variables des clauses puis les dégeler en temps voulu.

Procédure gel(A,B,N)

A et B sont des termes.

N est un naturel.

Cette procédure détermine si B, une copie de A sans liaison entre eux, a ses variables gelées selon la convention suivante:

toute variable de B est instanciée à une constante du type 'Var'(i) où i est un entier appartenant à [0 , N[et

pour tout X, Y variables appartenant à B, X = 'Var'(j), Y = 'Var'(k) j et k appartenant à $\mathbb{N}$, $j < k$ implique que X apparaît avant Y dans B, N étant le nombre de variables dans B.

Conditions d'application:

in(ground, var, var) : out(ground, ground, ground) <1-1>

N sera retenu pour chaque clause C dans la 'liste_c_clauses'

apparaissant dans chaque fait predicat/8. Il sert à trouver instantanément une numérotation telle que, si dans la clause C nous voulons rajouter des littéraux, elle permette à ces littéraux de ne pas avoir de variables communes. Nous avons donc besoin de la procédure suivante:

Procédure nomtranslat(X,Y,N)

X et Y sont des termes.

N est un entier.

Cette procédure crée, à partir de X, une copie Y n'ayant pas de liaisons avec X telle que toutes les variables de X instanciées à des constantes 'Var'(i) (i appartenant à |N) soient instanciées en Y à 'Var'(i+N) (i appartenant à |N).

Conditions d'application:

in(ground, var, ground) : out(ground, ground, ground) <1-1>

Le dégel des variables arrive en fin de processus d'optimisation. Il est donc naturel, pendant ce dégel, d'appliquer la transformation 27 qui s'occupe de générer des variables anonymes.

Procédure degel(X,Y)

X et Y sont des termes.

Cette procédure crée, à partir de X, une copie Y, ayant pas de liaisons avec X telle que toutes les variables de X instanciées à 'Var'(i) (i appartenant à |N) soient écrites sous la forme

_ si la variable n'apparaît qu'une seule fois dans X.

_i sinon.

Conditions d'application:

in(ground, var) : out(ground, ground) <1-1>

Examinons les grandes lignes de l'algorithme qui réalise la transformation 27. Une première lecture du terme X permet de trier dans une liste L les variables dans deux classes: celles qui n'apparaissent qu'une seule fois dans X et les autres (procédure degelnet1/3 et listerep/4). La liste L est épurée de toutes les variables apparaissant plusieurs fois dans X (procédure listesansrep/3). Une seconde lecture du terme X permet de réécrire les variables sous la forme _ ou _i suivant qu'elles appartiennent ou non à L (procédure degelnet2/3).

4.6. Le module genre

A ce stade, la base de donnée de référence de l'interpréteur **C-Prolog** contient les faits predicat/8 associés à chaque procédure à optimiser: seuls le genre de la procédure et la liste_c_clauses restent à déterminer.

Le module genre propose des permutations de clauses et de littéraux conformément aux transformations 12 et 22. En ce sens, il diffère de l'heuristique générale de transformation car le fait de trouver des paires de littéraux incompatibles et des séquences **T** (cf notation chapitre 2) est ici indépendant de la proposition de permutations; les transformations interclausales autres que la transformation 12 pourront peut-être changer les effets générés par le module genre (il s'agit de la seule différence avec l'heuristique générale de transformation).

Procédure genre
Les faits predicat/8, s'ils existent dans la base de donnée de référence doivent être de la forme
predicat(nom,arité,deter,effet_de_bord,infini,G,nbre_de_clauses,
[(Cl_n,_), ...,(Cl_2,_),(Cl_1,_)])
où Cl_1, Cl_2, ...,Cl_n sont des clauses de la procédure de nom 'nom' prises dans l'ordre lexicographique. Cette procédure modifie chaque fait predicat/8 présent dans cette base de donnée de la manière suivante:
la 'liste_c_clauses' contient des couples formés d'une clause dont les variables sont gelées et d'un nombre entier égal au nombre de variables dans la clause, et rangés selon l'ordre donné par les transformations 12 et 22 ou lexicographique suivant l'utilisateur. G, un des arguments de predicat/8 est instancié au genre de la procédure de nom 'nom' et d'arité 'arité'.

Procédure genrepr(F,A,L_1,[],L_2,[[],[],[]],L_3,G_1,G,0,I)

F est un atome
A est un naturel
F et A identifient une procédure
L_1 est une liste du type 'liste_c_clauses'
L_2 est une liste du type 'liste_c_clauses'
L_3 est une liste de trois listes du type 'liste_c_clauses'
où la première ne contient que des clauses de type SL, la deuxième, de type TR et la troisième, de type GR
G_1 et G appartiennent à [0, 1, 2]
I appartient à [0, 1]

L_2 est la liste inverse de L_1 où, pour chaque couple (C, N), chaque clause C a ses variables gelées et chaque entier N est instancié au nombre de variables de C.
Si I = 0, alors L_3 est **semblable** à L2 (égalité de chaque couple (C, N) pris deux à deux dans l'ordre d'apparition) et L_2 a ses couples (C, N) ordonnés suivant les transformations 12 et 22, G étant le genre de la procédure sous-jacente (si on écrit les clauses contenues dans L_2 l'une après l'autre).
Si I = 1, alors L_3 n'est pas semblable à L_2 et a ses couples (C, N) ordonnés comme plus haut.
Conditions d'application:
in(g, g, g, g, v, g, v, v, v, g, v) : out(g, g, g, g, g, g, g, g, g, g, g) <1-1>

Dans le cas où ce I est égal à 1, on soumet la permutation trouvée à l'utilisateur:

Procédure choixliste(L_1,L_2,I,L)
L_1 et L sont des listes de type 'liste_c_clauses'
L_2 est une liste de trois listes du type 'liste_c_clauses'
où la première ne contient que des clauses de type SL, la deuxième, de type TR et la troisième, de type GR
I appartient à [0, 1]
Si I = 0 alors L est égal à L_1.
Sinon L est semblable à L_2 (définition de semblable dans la spécification précédente) ou égale à L_1 suivant l'utilisateur.
Conditions d'application:
in(g, g, g, v,) : out(g, g, g, g) <1-1>

Procédure mepl(L_1,L_2)
L_1 est une liste de 3 listes
L_2 est une liste
Cette procédure détermine si la liste L_2 est la concaténation des listes de L_1
Conditions d'application:
in(ground, var) : out(ground, ground) <1-1>

Pour trouver une permutation des littéraux d'une clause conforme à la transformation 22 et déterminer le genre de cette clause, nous avons la

Procédure genrecl(Cl,F,A,C,G)

Cl et C sont des clauses
F est un atome
A est un naturel
F et A sont les foncteur et arité de la tête de clause
G est le genre de la clause C

S'il n'y a pas de littéraux dans le corps de Cl ayant pour foncteur et arité F/A alors C = Cl et G = 0. Sinon, s'il y en a plusieurs alors C est la clause Cl où ces littéraux se retrouvent à la fin du corps dans leur ordre d'apparition et G = 2.

Conditions d'application:

in(g, g, g, v, v) : out(g, g, g, g, g) <1-1>

Procédure genrecl(C,F,A,[],L,0,Gp,Gv)

C est le corps d'une clause de nom et d'arité F/A
L est une liste
Gp et Gv appartiennent à [0, 1, 2]

Si C ne contient pas de littéraux avec un foncteur et une arité F/A alors
Gp = Gv = 0 et L est la liste formée des littéraux de C pris en ordre inverse.

Si C ne contient qu'un littéral avec un foncteur et une arité F/A alors
si ce littéral est le dernier de C alors Gp = Gv = 1 et L est la liste formée des littéraux de C pris en ordre inverse.

Si ce littéral n'est pas le dernier de C alors Gp = 1 et le genre Gv = 2 et L est la liste formée des littéraux de C pris en ordre inverse mais où le littéral est en tête de liste.

Si C en contient plusieurs alors Gp = Gv = 2 et L est la liste formée des littéraux de C pris en ordre inverse mais où tous les littéraux F/A forment une conjonction placée en tête de liste. Cette conjonction a ses termes rangés en ordre inverse d'apparition de ces termes dans C.

Conditions d'application:

in(g, g, g, g, v, g, v, v) : out(g, g, g, g, g, g, g, g) <1-1>

Procédure choixcl(X,Y,Gp,Gv,C,G)

X, Y et C sont des clauses
Gp, Gv et G appartiennent à [0, 1, 2]

Gp est le genre de Y, Gv celui de X.

Si Gv = 2 alors
si l'utilisateur accepte la permutation des littéraux de X en Y alors C = Y et G = Gp sinon C = X et G = Gv

sinon C = X et G = Gv.

Conditions d'application:

in(g, g, g, g, v, v) : out(g, g, g, g, g, g,) <1-1>

Procédure linear(L,T_1,T)

L est une liste

T_1 et T sont des conjonctions de termes

T est la conjonction des éléments de L pris en ordre inverse avec le terme T_1. Si un élément de L est une conjonction de termes alors ils figurent en conjonction dans T mais pris en ordre inverse.

Conditions d'application:

in(g, g, v) : out(g, g, g) <1-1>

Procédure lpp(C,N,Gc,G_1,G,C_1,I,V_1,V)

C est une clause

G_1, Gc et G appartiennent à [0, 1, 2]

N est un naturel

I_1 et I appartiennent à [0, 1]

V_1 et V sont des listes de trois listes du type 'liste_c_clauses'

où la première ne contient que des clauses de type SL, la deuxième, de type TR et la troisième, de type GR

Gc est le genre de la clause C et N le nombre de ses variables. G_1 est le genre de la procédure dont les clauses figurent dans l'ordre dans la liste V_1.

Si la clause C à rajouter dans la liste V_1 (sous le couple (C, N)) pour donner V (en fonction de Gc) a un genre Gc =< G1 alors G = G_1 et I = I_1 sinon G = Gc et I = 1.

Conditions d'application:

in(g, g, g, v, g, v, g, v) : out(g, g, g, g, g, g, g, g) <1-1>

4.7. Le module notetincomp

Ce module prend en charge les transformations 1, 2, 3, 4, 5 et 6 du chapitre 2. Il exécute plus exactement la transformation 1 avec des paires de littéraux incompatibles indiquées par l'utilisateur et la paire C-not(C) qu'il est capable de détecter, il traite donc uniquement les procédures à deux clauses. Nous sommes donc confronté avec des permutations de littéraux. Comme il a été dit au début du troisième chapitre, nous codons les permutations de littéraux sous la forme d'une liste d'entiers: soit la clause

$$H :- S_1 , S_2 , S_3 , S_4 \qquad (1)$$

l'arrangement des littéraux S_i est codé sous la forme [1, 2, 3, 4]. Si la clause suivante est équivalente à (1)

$$H :- S_3 , S_1 , S_2 , S_4$$

la permutation des S_i est codée [3, 1, 2, 4]. A chaque littéral du corps de la clause correspond donc un entier qui est la position de ce littéral dans la clause primitive (1), celle qui n'a pas encore subi de permutations.

Soit la procédure

$$H :- S_1 , S_2 , S_3 , S_4 \qquad (2)$$
$$H :- T_1 , T_2 , T_3 , T_4$$

Supposons que S_1-T_3 et S_2-T_1 soient des paires de littéraux incompatibles. Nous coderons ces renseignements sous forme d'une liste de couples de naturels

$$[(1, 3), (2, 1)] \qquad (3)$$

où chaque entier représente la position de chacun des littéraux dans les clauses primitives correspondantes. Une permutation est donc ici synonyme de liste de naturels et la liste des paires de littéraux incompatibles, de liste de couples de naturels. Nous les utiliserons indifféremment dans les spécifications sans en rappeler leur sens.

Un autre problème est le fait d'accorder une place privilégiée aux paires de littéraux incompatibles qui se situent le plus en avant dans un corps de clause (cf chapitre 2 et 3 pour les transformations interclausales) pour un ensemble donné de permutations !

Chaque clause de la procédure (2) peut s'écrire sous forme de liste

$$L_{11} = [1, 2, 3, 4]$$
$$L_{21} = [1, 2, 3, 4]$$

Supposons que les permutations de littéraux suivantes soient permises: pour la première clause:

$$L_{12} = [2, 3, 1, 4]$$
$$L_{13} = [1, 4, 3, 2]$$

et pour la seconde

$$L_{22} = [2, 4, 1, 3]$$
$$L_{23} = [3, 4, 2, 1]$$

Nous inventons une structure de donnée qui va singulièrement nous simplifier la tâche. L'idée est d'associer chaque liste de permutation L_{ij} à un nombre naturel N qui correspond à l'apparition en Nième place d'un littéral, qui fait partie d'une paire de littéraux incompatibles, dans cette liste L_{ij}. Dans cette association, il y aura aussi la paire de littéraux incompatibles en question, pour la première clause de (2), nous obtenons:

$$\begin{aligned}
[\,&(\,[\,(\,[\,L_{11}, L_{13}\,]\,,(\,1, 3\,)\,)\,,(\,[\,L_{12}\,]\,,(\,2, 1\,)\,)\,]\,, 1\,)\,,\\
&(\,[\,(\,[\,L_{11}\,]\,,(\,2, 1\,)\,)\,]\,, 2\,)\,,\\
&(\,[\,(\,[\,L_{12}\,]\,,(\,1, 3\,)\,)\,]\,, 3\,)\,,\\
&(\,[\,(\,[\,L_{13}\,]\,,(\,2, 1\,)\,)\,]\,, 4\,)\,]
\end{aligned} \qquad (4)$$

Une liste sera dite de **type** Lt ssi elle a la *même* structure et sémantique que (4). Voici cette sémantique.

Supposons que l'on ait une paire de littéraux incompatibles L_1-L_2 où L_1 apparaît à la Nième position dans la permutation permise L_{1i} de la première clause. La liste (4) est composée de termes (quatre dans notre cas). Chacun de ces termes est un couple dont le second élément est N. Le premier élément de ce couple est une liste de couples. Le second élément de ces couples est une paire de littéraux L_1-L_2 et le premier élément de ces couples est une liste de permutations permises où le littéral L_1, associé à la paire L_1-L_2 se trouve en Nième position. Les termes de cette liste (4) sont triés suivant les valeurs croissantes de N. La liste qui est le premier élément de chaque couple de la liste (4) a ses éléments triés suivant les valeurs croissantes du naturel indiquant la position du littéral L_1 dans le couple de naturels associé à la paire de littéraux incompatibles. Une même liste que (4) sera construite pour la seconde clause de la procédure (2) où N sera la position du littéral L_2 dans les permutations permises. Elle sera triée cependant de la même façon.

Les mauvaises langues dirons que " plus lourde que la liste (4), tu meurs !", cependant du point de vue dynamique elle est très pratique. En effet, après avoir défini chaque liste de type Lt pour chacune des

clauses d'une procédure à deux clauses, il suffit de les parcourir en parallèle pour trouver d'abord des naturels "N" identiques, après de parcourir les sous-listes pour trouver les mêmes paires de littéraux incompatibles; ceci pour trouver le plus rapidement possible (tri sur ces paires) une paire de permutations permises qui ait les paires de littéraux incompatibles le plus en avant dans les corps de clauses correspondants (tri sur N).

Les spécifications de la procédure tripermgoal/5 (et consorts: triprnetgl/6, triprnetg2/6, trin/7 et trig/4) peuvent donc se résumer à ceci: à partir d'une liste de type Lt, d'une permutation permise L_{ij} et de la liste des paires de littéraux incompatibles, la procédure tripermgoal/5 complète la liste de type Lt en lui ajoutant, là ou il faut, la permutation permise L_{ij}. Quant à la procédure selectprn/8, à partir de deux listes de couples, elle sélectionne les couples ayant les mêmes seconds éléments et se présentant le plus près des têtes de ces listes.

Procédure notetincomp

Cette procédure modifie tous les faits predicat/8 de la base de donnée de référence de l'interpréteur **C-Prolog**, associés à une procédure P_1 à deux clauses, s'ils existent de telle manière que la 'liste_c_clauses' contienne dans l'ordre lexicographique les clauses d'une procédure P_2, résultat de

P_2 = appliquer([1, 2, 3, 4, 5, 6], P_1)

Procédure constrlcg(Lcg,Lt_1,Lt_2)

Lcg est une liste de couples de naturels

Lt_1 et Lt_2 sont de type Lt

Dans la base de référence de l'interpréteur **C-Prolog** il doit y avoir le fait cl(C_1,C_2) où

C_1 et C_2 sont deux clauses

Cette procédure détermine toutes les paires de littéraux incompatibles C-not(C) et demande les autres à l'utilisateur (oracle9/2), qu'elle met dans la liste Lcg. Lt_1 correspond à la liste de type Lt de toutes les permutations permises de littéraux données par l'utilisateur (oracle10/5) pour la clause C_1; Lt_2 correspondant à la clause C_2. Réussit si Lcg ≠ [].

Conditions d'application:

in(v, v, v,) : out(g, g, g) <0-1>

Procédure makelist1(B_1,LB,L,B_2,Lcg,1)

B_1 est un corps de clause C_1

LB est une liste de termes

L est une liste de naturels

B_2 est un corps de clause C_2

Lcg est une liste de couples de naturels

Cette procédure détermine Lcg qui contient toutes les paires de littéraux incompatibles not(C)-C où not(C) appartient à B_1 et C à B_2. LB est une liste qui contient dans l'ordre tous les littéraux de B_1 et L est une liste [1,2,...,N] ou N est le nombre de littéraux de B_1.

Conditions d'application:

in(*g*, *v*, *v*, *g*, *v*, *g*) : out(*g*, *g*, *g*, *g*, *g*, *g*) <1-1>

Procédure makelist2(B_1,LB,L,B_2,Lcg,1)

B_1 est un corps de clause C_2

LB est une liste de termes

L est une liste de naturels

B_2 est une liste dont les éléments sont, dans l'ordre, les littéraux d'un corps de clause C_1

Lcg est une liste de couples de naturels

Cette procédure détermine Lcg qui contient toutes les paires de littéraux incompatibles not(C)-C où not(C) appartient à B_1 et C à B_2. LB est une liste qui contient dans l'ordre tous les littéraux de B_1 et L est une liste [1,2,...,N] ou N est le nombre de littéraux de B_1.

Conditions d'application:

in(*g*, *v*, *v*, *g*, *v*, *g*) : out(*g*, *g*, *g*, *g*, *g*, *g*) <1-1>

Procédure rempl11(C,I,B,L,L_1)

C est un littéral

I est un naturel

B est un corps de clause

L et L_1 sont des listes de couples de naturels

Cette procédure détermine si C se trouve dans B et en quelle position *j*. S'il s'y trouve alors L = [(I, *j*) | L_1] sinon L = L_1.

Conditions d'application:

in(*g*, *g*, *g*, *v*, *g*) : out(*g*, *g*, *g*, *g*, *g*) <1-1>

Procédure rempl12(C,I,B,L,L_1)

C est un terme
I est un naturel
B est une liste de termes
L et L_1 sont des listes de couples de naturels

Cette procédure détermine si C se trouve dans B et en quelle position j. S'il s'y trouve alors L = [(I, j) | L_1] sinon L = L_1.

Conditions d'application:

in(g, g, g, v, g) : out(g, g, g, g, g) ‹1-1›

Procédure mofconj(C,B,I,J)

C est un littéral
B est un corps de clause
J est un naturel

Réussit si C se trouve dans B à la Jième position.

Conditions d'application:

in(g, g, g, v) : out(g, g, g, g) ‹0-1›

Procédure membernbr(C,B,I,J)

C est un littéral
B est une liste de termes
J est un naturel

Réussit si C se trouve dans B à la Jième position.

Conditions d'application:

in(g, g, g, v) : out(g, g, g, g) ‹0-1›

A ce stade des opérations, les listes de genre Lt sont formées pour chaque clause de la procédure à transformer. Il faut maintenant trouver des permutations permises des deux clauses telles qu'elles puissent subir l'une des transformations 1, 2, 3, 4, 5 et 6.

Les procédures trouvertransf/3, itersurn/6 et itersurg/7 forment une série d'itérations. Elles sont exactement les mêmes dans leur conception. Nous ne présenterons que la

Procédure trouvertransf(L_1,L_2,L)

L_1 et L_2 sont de type Lt
L est une liste de deux clauses
Dans la base de donnée de référence de l'interpréteur **C-Prolog** il doit y avoir le fait cl(C_1,C_2) où C_1 et C_2 sont deux clauses

Si cette procédure, à partir de L_1 et L_2, trouve une paire de clauses telles qu'elles répondent aux conditions des transformations 1,2,3,4,5 et 6, elle construit L = (Ct_1, Ct_2) où Ct_1 et Ct_2 sont les clauses transformées sinon

$L = (C_1,C_2)$.

Conditions d'application:

in(g, g, v) : out(g, g, g) <1-1>

Procédure prodcltrans(L_1,L_2,N,G,L)

L_1 et L_2 sont des listes de listes de naturels

N est un naturel

G est un couple (G_1,G_2) où

G_1 et G_2 sont des naturels

L est une liste de deux clauses

N est la position des littéraux de numéro G_1 et G_2 respectivement dans les listes de L_1 et de L_2. S'il existe une paire de listes dans L_1 et L_2 telles que leur préfixes jusqu'à G_1 et G_2 non compris correspondent à des séquences de littéraux déterministes identiques et sans effet de bord alors $L = (C_1,C_2)$ où C_1 et C_2 sont les clauses transformées par les transformations 1, 2, 3, 4, 5 et 6 à partir de celles codées sous la forme L_1 et L_2; sinon elle échoue.

Conditions d'application:

in(g, g, g, g, v) : out(g, g, g, g, g) <0-1>

Procédure verifsedet(S)

S est une séquence de littéraux

Elle réussit si tous les littéraux de S sont déterministes ou entièrement déterministes et sans effet de bord.

Conditions d'application:

in(g) : out(g) <0-1>

Procédure makeprefixe(I,X,1,N,[],L,S)

X et S sont des listes de naturels

I est le numéro de la clause dans la procédure ayant pour permutation permise X

N est un naturel

L est une liste de littéraux

L est la liste formée des littéraux correspondant aux naturels inclus dans X jusqu'au naturel de position N non compris, pris en ordre inverse. S est le suffixe de X à partir de l'élément de position N non compris.

Conditions d'application:

in(g, g, g, g, g, v, v) : out(g, g, g, g, g, g, g) <1-1>

Procédure makesuffixe(I,S,T)

I est un naturel

S est une liste de naturels
T est une conjonction de littéraux
T est le suffixe du corps d'une clause qui correspond à une permutation permise de la clause de numéro I dans la procédure. Le code de T est S.
Conditions d'application:
in(g, g, v) : out(g, g, g) <1-1>

Procédure xversg(I,X,G)
I et X sont des naturels
G est un littéral
Dans la base de donnée de référence de l'interpréteur **C-Prolog**, il doit y avoir des faits corps(L_1) et corps(L_2) où
L_1 et L_2 sont des listes de littéraux qui, dans l'ordre, forment les corps des clauses primitives de la procédure à transformer
Si I est le numéro de la clause primitive alors G est le littéral de Xième position dans L_I.
Conditions d'application:
in(g, g, v) : out(g, g, g) <1-1>

Procédure makegt(G_1,G_2,Gt_1,Gt_2)
G_1 et G_2 sont des naturels
Gt_1 et Gt_2 sont des littéraux
G_1 et G_2 sont les positions des littéraux L_1 et L_2, formant une paire de littéraux incompatibles, dans respectivement la première clause et la seconde clause de la procédure à transformer. Gt_1 et Gt_2 forment la paire de littéraux incompatibles telle que
si L_1 = not(C) et L_2 = C alors Gt_1 = not(C) et Gt_2 = true
sinon si L_1 = C et L_2 = not(C) alors Gt_1 = C et Gt_2 = true
sinon Gt_1 = L_1 et Gt_2 = L_2.
Conditions d'application:
in(g, g, v, v) : out(g, g, g, g) <1-1>

4.8. Le module expansion

Ce module réalise la transformation 13, la décomposition d'une clause et la transformation 14. Pour chaque clause de chaque procédure, nous construisons d'abord la disjonction de littéraux, c'est-à-dire l'expansion de la clause, puis, nous la décomposons.

Procédure expansion
Cette procédure modifie chaque fait predicat/8 présent dans la base de donnée de référence de l'interpréteur **C-Prolog** de telle manière que, chaque procédure P contenue sous la forme de 'liste_c_clauses' soit modifiée en P_2 selon les étapes suivantes:

$$P_1 = \text{appliquer}([14],P)$$

et P_2 est le résultat de la décomposition de chaque clause contenant des disjonctions et de l'application sur ces clauses de l'heuristique H_1.

Procédure exl(N,A,L,L_1)

N est un atome
A est un naturel
L est de type 'liste_c_clauses'
L_1 est une liste de couples (Lcl,Ns) tels que Lcl est une liste de clauses et Ns est le nombre de variables contenues dans la clause qui en contient le plus.

Lcl est le résultat de l'expansion, décomposition et application de l'heuristique H_1 de la clause correspondante dans L.

Conditions d'application:
in(g, g, g, v) : out(g, g, g, g) <1-1>

Procédure flatten(L_1,L_2,0,Nc_1)

L_1 est une liste de couples (Lcl,Ns) tels que Lcl est une liste de clauses et Ns est le nombre de variables contenues dans la clause qui en contient le plus.
L_2 est une liste de type 'liste_c_clauses'
Nc_1 est un naturel

L2 est formée des couples (Cli,Ncj) où toute les clauses Cli sont les clauses, appartenant aux listes Lcl associées à un entier Ncj, où l'heuristique H1 leur a été appliquées. Elles figurent dans L2 dans le même ordre d'apparition que dans L1. Nc1 est le nombre de clauses.

Conditions d'application:
in(g, v, g, v) : out(g, g, g, g) <1-1>
La transformation 13 doit être réalisée sous certaines conditions.

Soit un littéral N/A à étendre. Sa procédure de définition ne doit pas contenir de cuts, doit être de genre SL et ne doit pas contenir un littéral de même foncteur et arité que la clause dans laquelle est inclus le littéral N/A. De plus, elle ne doit pas contenir des littéraux de même foncteur et arité que ceux déjà étendus dans la même clause. Nous avons la

Procédure excl(B,B_1,[(N,A)],_,Ns,Ns_1)

B est une conjonction de littéraux
B1 est une conjonction et/ou disjonction de littéraux
N et A sont les foncteur et arité du littéral à étendre
Ns et Ns_1 sont des naturels.

Cette procédure effectue l'expansion des littéraux de B et donne en résultat B_1. Ns_1 est le nombre de variables incluses dans B_1 et Ns est le nombre de variables incluses dans B.

Conditions d'application:

in(g, v, g, v, g, v) : out(g, g, g, g, g, g) <1-1>

Procédure tcl(S,S_1,Lg,Lg_1,Ns,Ns_1)

S est un littéral
S_1 est une conjonction ou une disjonction de littéraux
Lg et Lg_1 sont des listes de couples (foncteur, arité)
Ns et Ns_1 sont des naturels

Si S est un littéral n'ayant pas de procédure de définition adéquate, alors, S_1 = S, Lg_1 = Lg et Ns_1 = Ns.

Sinon, la procédure de définition existe et vérifie les conditions d'application de la transformation 13. Lg_1 contient tous les foncteurs et arités des littéraux étendus provenant de l'expansion de S. Ns est le nombre de variables contenues dans la clause de S et Ns_1 est le nombre de variables contenues dans la clause de S après expansion de S et de tous les littéraux possibles issus de cette expansion.

Conditions d'application:

in(g, v, g, v, g, v) : out(g, g, g, g, g, g) <1-1>

La procédure suivante effectue la vérification des conditions d'application 1 et 4 de la transformation 13.

Procédure validexp(Lg,L)

L est une liste de type 'liste_c_clauses'
Lg est une liste de couples (foncteur, arité)

Réussit si L ne contient pas de clauses ayant des cuts ou des littéraux de foncteur et arité appartenant à Lg.

Conditions d'application:

in(g, g) : out(g, g) <0-1>

Procédure constrdis(L,S,Ns,D,Ns1)

L est une liste de type 'liste_c_clauses'
S est un littéral
D est une disjonction de littéraux
Ns et Ns_1 sont des naturels

L est la liste des clauses de la procédure de définition du littéral S. D est le résultat de l'expansion de S suivant la procédure de définition contenue dans L. Seuls les termes disjonctifs différents de 'fail' sont pris en compte et s'il n'existe aucun autre terme disjonctif différent de 'fail', alors, D = 'fail'. Ns est le nombre de variables de la clause contenant S et Ns_1 est le nombre de variables de la clause contenant l'expansion de S.

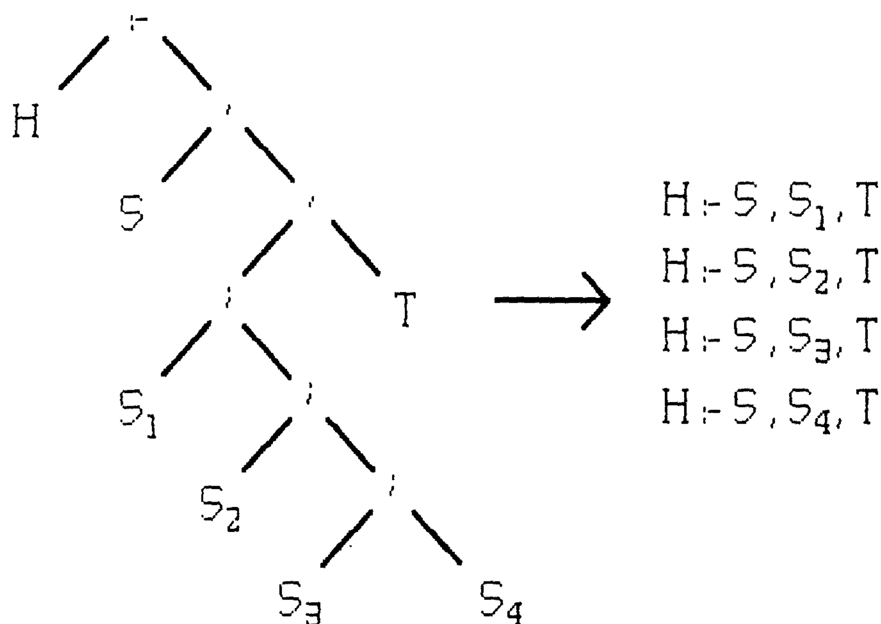
Conditions d'application:

in(g, g, g, v, v) : out(g, g, g, g, g) <1-1>

transclause/8 a été employée pour vérifier s'il y a unification entre S et la tête de chacune des clauses contenue dans L.

Les naturels Ns servent à trouver instantanément des noms de variables qui n'ont jamais été employées dans la clause qui est en train de s'étendre. Cette opération s'effectue grâce à la procédure nomtranslat/3.

Nous avons maintenant une clause avec des disjonctions (clause disjonctive). Comme l'optimiseur ne prend pas en compte le traitement des disjonctions, il faut ramener la clause disjonctive à un système de clauses ordonné, c'est-à-dire, dans un cas simple:



C'est l'opération de décomposition.

Procédure decomposition(C,Lc)

C est une clause disjonctive

Lc est une liste de clauses.

Cette procédure décompose la clause disjonctive C en un système de clauses ordonné Lc.

Conditions d'application:

in(g, v) : out(g, g) <1-1>

Procédure dec(B,[[]],L)

B est une conjonction ou disjonction de littéraux

L est une liste de listes.

Chaque liste de L contient les littéraux formant le corps d'une clause pris en ordre inverse. Cette clause sera l'une des clauses d'un système de clauses ordonné issu de la clause disjonctive de corps B.

Conditions d'application:

in(g, g, v) : out(g, g, g) <1-1>

Procédure empilgoal(S,L,L_1)

S est un littéral

L et L1 sont des listes de listes.

L1 est formée à partir de L où est ajouté en tête de chacune des listes de L, S.

Conditions d'application:

in(g, g, v) : out(g, g, g) <1-1>

Procédure clcl(L,Lc,H)

L est une liste de listes comprenant les littéraux d'un corps de clause pris en ordre inverse.

Lc est une liste de clauses

H est une tête de clause.

La ième clause de Lc est fabriquée à partir de la ième liste de L en formant le corps de clause et en ajoutant "H :-".

Conditions d'application:

in(*g*, *v*, *g*) : out(*g*, *g*, *g*) <1-1>

Procédure incorporer_l_t(L,T,Tf)

L est une liste de termes

T et Tf sont des termes.

Tf est la conjonction de tous les éléments de L pris en ordre inverse et de T.

Conditions d'application:

in(*g*, *g*, *v*) : out(*g*, *g*, *g*) <1-1>

4.9. Le module transclause

Ce module réalise l'heuristique H_1. Il applique sur une clause les transformations 18, 19, 20, 21, 23, 24, 25, 26, 28 et 29.

Une clause a la forme

H :- S , T

où H est la tête de clause, S est la séquence de littéraux déjà analysés par le module et T est la séquence de littéraux à analyser. Pour faciliter l'implémentation, la séquence S sera sous forme de liste **Ls** dont chaque élément sera un littéral de S et dont l'ordre sera inversé. Ainsi, si S est la séquence P_1,P_2,P_3, Ls aura la forme $[P_3,P_2,P_1]$. Ls est dit **équivalent** à S.

Pour réaliser les transformations 18 et 19, le module doit savoir si la clause qu'on lui soumet est la dernière ou non d'une procédure. De plus, il doit savoir à un moment donné si ces transformations sont applicables. Un moyen simple de coder ces informations est de définir un entier S tel que

S = 2 = Le littéral précédemment analysé est un cut ou, la clause est la dernière d'une procédure et, il n'y a pas encore de littéral analysé.

S = 1 = Les littéraux analysés précédemment forment une séquence déterministe après, soit un cut, soit le début de la dernière clause.

S = 0 = Aucun des cas précédents n'a été rencontré.

Lorsque, dans les spécifications, nous dirons que S est de type cascut, cela sous-entendra la définition précédente.

Procédure transclause(C_1,C_2,S)

C_1 et C_2 sont des clauses

S est de type cascut

C_2 = appliquer([18,19,20,21,23,24,25,26,28,29],C_1).

Conditions d'application:

in(g, v, g) : out(g, g, g) <1-1>

Procédure transclause(H,Hm,[],Ls,T,_,S,_)

H et Hm sont des têtes de clauses

Ls est de type Ls

T est une séquence de littéraux

S est de type cascut.

H :- T forme la clause C1 et Ls est équivalent à S, alors cette procédure effectue

Hm :- S = appliquer([18,19,20,21,23,24,25,26,28,29],H :- T).

Conditions d'application:

in(g, v, g, v, g, v, g, v) : out(g, g, g, g, g, g, g, g) <1-1>

Analysons d'abord l'implémentation des procédures auxiliaires qui permettent la réalisation des transformations 23, 24, 25 et 26. Dans 23., nous avons défini les différents types d'égalités:

objets identiques	**type 1**
variable = variable	**type 2**
variable = terme	**type 3**
terme = variable	**type 4**
terme = terme	**type 5**

La procédure suivante permet de trouver le type d'une égalité:

Procédure nature(X,Y,T,[],L)

X et Y sont des termes

T appartient à {1, 2, 3, 4, 5, 6}

L est une liste

Cette procédure détermine le type T de l'égalité X = Y et L a pour valeur

[X = Y] pour T = 1, 2, 3 ou 4; ou

$[X_n = Y_n, ..., X_2 = Y_2, X_1 = Y_1]$ pour T = 5 et pour X et Y ayant respectivement comme valeur $f(X_1, X_2, ..., X_n)$, $f(Y_1, Y_2, ..., Y_n)$; ou

[fail] pour T = 6 lorsque X et Y ne sont pas unifiables (occur check compris).

Conditions d'application:

in(g, g, v, g, v) : out(g, g, g, g, g) <1-1>

La procédure suivante réalise l'occur check:

Procédure not_occur_in(X,Y)

X et Y sont des termes.

Cette procédure détermine si X est égal ou est compris dans le terme Y.

Conditions d'application:

in(g, g) : out(g, g) <0-1>

Procédure subst(X,Y,T_1,T_2)

X, Y, T_1 et T_2 sont des termes.

Cette procédure substitue X par Y dans T_1 pour donner T_2.

Conditions d'application:

in(g, g, g, v) : out(g, g, g, g) <1-1>

Cette procédure est dangereuse lorsque les variables ne sont plus gelées. Elle effectue alors des instantiations involontaires.

Les traitements pour ces différents types d'égalités dépendent aussi de la forme de la clause; à

H :- X = Y	**est associé**	**caseg1/7,**
H :- X = Y , T		**caseg2/8,**
H :- S , X = Y , T		**caseg3/11,**
H :- S , X = Y		**caseg4/10.**

Procédure caseg1(Ty,H,Hm,Lsm,X,Y,L)

Ty est le type de l'égalité X = Y où
X et Y sont des termes
H et Hm sont des têtes de clause
Lsm est de type Ls
L est une liste issue de nature/5.

Sm est équivalent à Lsm. Hm :- Sm est une clause ainsi que H :- X = Y.
Soit Ty = 1 alors Hm = H et Lsm = [true],
soit Ty = 2 ou 3 alors Hm = H où X est substitué par Y et Lsm = [true],
soit Ty = 4 alors Hm = H où Y est substitué par X et Lsm = [true],
soit Ty = 5 alors Hm et Lsm sont issus de transclause/8,
soit Ty = 6 alors Hm = H et Lsm = [fail].

Conditions d'application:

in(*g*, *g*, *v*, *v*, *g*, *g*, *g*) : out(*g*, *g*, *g*, *g*, *g*, *g*, *g*) <1-1>

Procédure caseg2(Ty,H,Hm,T,Tm,X,Y,L)

Ty est le type de l'égalité X = Y où
X et Y sont des termes
H et Hm sont des têtes de clause
T et Tm sont des séquences de littéraux
L est une liste issue de nature/5.

Hm :- Tm est une clause ainsi que H :- X = Y , T.
Soit Ty = 1 alors Hm = H et Tm = T,
soit Ty = 2 ou 3 alors Hm = H où X est substitué par Y et Tm = T où X est substitué par Y,
soit Ty = 4 alors Hm = H où Y est substitué par X et Tm = T où Y est substitué par X,
soit Ty = 5 alors Hm = H et Tm est égal à la conjonction des éléments de L pris en ordre inverse et de T,
soit Ty = 6 alors Hm = H et Tm = fail , T.

Conditions d'application:

in(*g*, *g*, *v*, *g*, *v*, *g*, *g*, *g*) : out(*g*, *g*, *g*, *g*, *g*, *g*, *g*, *g*) <1-1>

Procédure caseg3(Ty,H,Ls,T,Lsm,Tm,X,V,L,S_1,S_2)

Ty est le type de l'égalité X = V où
X et V sont des termes
H est une tête de clause
Lsm et Ls sont de type Ls
T et Tm sont des séquences de littéraux
L est une liste issue de nature/5
S_1 et S_2 sont de type cascut.

S et Sm sont équivalents respectivement à Ls et Lsm, H :- Sm , Tm est une clause ainsi que H :- S , X = V , T.
Soit Ty = 1 alors Lsm = Ls, S_2 = S_1 et Tm = T,
soit Ty = 2 alors
si X ne se trouve pas dans Ls et H, alors Tm = T où X est substitué par V, S_2 = S_1 et Lsm = Ls,
sinon si V ne se trouve pas dans Ls et H, alors Tm = T où V est substitué par X, S_2 = S_1 et Lsm = Ls,
sinon Tm = T où X est substitué par V et Lsm et S_2 sont issus de cascut/5 pour Ls, S_1 et X = V,
soit Ty = 3 alors
si X ne se trouve pas dans Ls et H, alors Tm = T où X est substitué par V, S_2 = S_1 et Lsm = Ls,
sinon Tm = T où X est substitué par V et Lsm et S_2 sont issus de cascut/5 pour Ls, S_1 et X = V,
soit Ty = 4 alors
si V ne se trouve pas dans Ls et H, alors Tm = T où V est substitué par X, S_2 = S_1 et Lsm = Ls,
sinon Tm = T où V est substitué par X et Lsm et S_2 sont issus de cascut/5 pour Ls, S_1 et X = V,
soit Ty = 5 alors Lsm = Ls, S_2 = S_1 et Tm est égal à la conjonction des éléments de L pris en ordre inverse et de T,
soit Ty = 6 alors Lsm = Ls et Tm = fail , T.

Conditions d'application:

in(g, g, g, g, v, v, g, g, g, g, v) : out(g, g, g, g, g, g, g, g, g, g, g) <1-1>

Procédure caseg4(Ty,H,Ls,Lsm,Tm,X,Y,L,S_1,S_2)

Ty est le type de l'égalité X = Y où
X et Y sont des termes
H est une tête de clause
Lsm et Ls sont de type Ls
Tm est une séquence de littéraux
L est une liste issue de nature/5
S_1 et S_2 sont de type cascut.

S et Sm sont équivalents respectivement à Ls et Lsm, H :- Sm , Tm est une clause ainsi que H :- S , X = Y.
Soit Ty = 1 alors Lsm = Ls et $S_2 = S_1$,
soit Ty = 2 alors
si X ne se trouve pas dans Ls et H, alors $S_2 = S_1$ et Lsm = Ls,
sinon si Y ne se trouve pas dans Ls et H, alors $S_2 = S_1$ et Lsm = Ls,
sinon Lsm et S_2 sont issus de cascut/5 pour Ls, S_1 et X = Y,
soit Ty = 3 alors
si X ne se trouve pas dans Ls et H, alors $S_2 = S_1$ et Lsm = Ls,
sinon Lsm et S_2 sont issus de cascut/5 pour Ls, S_1 et X = Y,
soit Ty = 4 alors
si Y ne se trouve pas dans Ls et H, alors $S_2 = S_1$ et Lsm = Ls,
sinon Lsm et S_2 sont issus de cascut/5 pour Ls, S_1 et X = Y,
soit Ty = 5 alors Lsm et Tm sont issus de transclause/8.
soit Ty = 6 alors Lsm = [fail | Ls] et $S_2 = S_1$.

Conditions d'application:

in(g, g, g, *v*, *v*, g, g, g, g, *v*) : out(g, g, g, g, g, g, g, g, g, g) <1-1>

Analysons ensuite l'implémentation des procédures auxiliaires qui permettent la réalisation des transformations 18, 19, 20 et 21.

Procédure cascutpm(H,Hm,Ls,Lm,(!,T),Tm,S_1,S_2)

H et Hm sont des têtes de clauses
Ls et Lsm sont de type Ls
T et Tm sont des séquences de littéraux
S_1 et S_2 sont de type cascut.

S et Sm sont équivalentes respectivement à Ls et Lsm. H :- S , ! , T et Hm :- Sm , Tm sont des clauses.
Hm :- Sm , Tm = appliquer([18,19,20,21,23,24,25,26,28,29],H :- S , ! , T).

Conditions d'application:

in(g, *v*, g, *v*, g, *v*, g, *v*) : out(g, g, g, g, g, g, g, g) <1-1>

Quelle est donc la différence entre cascutpm/8 et transclause/8 ? Le simple fait que la séquence des littéraux à analyser commence par un cut. Ce dédoublement permet de ne pas avoir une procédure transclause/8 trop importante, elle augmenterait alors le temps d'exécution.

Les transformations 18 et 19 sont exécutées par la

Procédure cascut(G,S_1,S_2,Ls,Lsm)

G est un littéral

S_1 et S_2 sont de type cascut

Ls et Lsm sont de type Ls

Si S_1 = 0, alors S_2 = 0 et Lsm = [G | Ls], sinon

si S_1 = 1, alors

si G est sans effet de bord et déterministe ou entièrement déterministe, alors S_2 = 1 et Lsm = [G | Ls]

sinon S_2 = 0 et Lsm = [G , ! | Ls], sinon

si S_1 = 2, alors

si G est sans effet de bord et déterministe ou entièrement déterministe, alors S_2 = 1 et Lsm = [G | Ls]

sinon S_2 = 0 et Lsm = [G | Ls], sinon

Conditions d'application:

in(g, g, v, g, v) : out(g, g, g, g, g) <1-1>

4.10. Conclusions sur l'implémentation

Malgré les particularités du module **genre**, nous pensons que notre optimiseur est équivalent à l'heuristique générale de transformation H_4 (cfr 3.3).

Le module **notetincomp** ne prend en compte que les procédures à deux clauses. Pour qu'il soit capable de traiter les cas généraux, il faut trouver une méthode qui trie d'abord les clauses d'une procédure de telle manière que la forme de la procédure soit un cas de figure transformable. Alors le module serait capables, sous de faibles modifications, d'appliquer toutes les transformations qui n'ont pas été implémentées (cf chapitre 2).

Nous n'avons pas réellement cherché à optimiser l'optimiseur, mais bien à ce que l'optimiseur optimise.

Il a aussi été construit avec un profil qui permet l'insertion aisée d'autres transformations d'optimisation.

Le chapitre suivant donne des exemples chiffrés en temps d'exécution et un exemple d'exécution qui montre comment utiliser l'optimiseur.

5
Analyse de l'heuristique et de l'implémentation

Maintenant que nous possédons un optimiseur, il serait tentant de donner un coefficient d'efficacité à chaque transformation d'optimisation du chapitre deux. Ce coefficient serait par exemple le rapport entre les temps d'exécution du programme non optimisé et sa transformée. Un tel coefficient serait, par la nature même des programmes, fort imprécis. Dans certains cas les transformations d'optimisation peuvent être diablement efficaces, dans d'autres, apportent un léger plus. Nous allons plutôt appliquer l'optimiseur sur des procédures usuelles. Afin de se rendre compte des effets de l'optimisation, les temps d'exécution des procédures non transformées et optimisées seront déterminés.

5.1. Détermination du temps d'exécution d'une procédure pour un goal donné

Le prédicat cputime/0 du **C-Prolog** donne le temps cpu exprimé en secondes. Il existe un programme Prolog qui détermine de manière précise le temps d'exécution d'un goal.

Si le temps d'exécution d'un goal est rapide, il sera imprécis, ceci à cause de la précision des temps fournis par le système d'exploitation. Le programme qui détermine le temps d'exécution d'un goal, exécute une séquence de littéraux (gestion d'un compteur, ...) dans laquelle se trouve le goal, cela un certain nombre de fois x. Il détermine le temps d'exécution de l'itération où est compris le temps d'exécution du goal un certain nombre x de fois. Ensuite, il détermine celui de l'itération sans le goal. La soustraction entre ces deux temps donne le temps de l'exécution du goal x fois. Dans ce temps n'est pas compris celui de l'affichage des résultats. Dans les exemples qui suivent, x prendra arbitrairement la valeur 100.

5.2. Les influences de la directionalité des procédures

Effectuons une première série de mesures sur la procédure efface/3. Les spécifications de efface/3 sont données dans ((1)).

Procédure efface(X,L,Leff)

X est un terme.

L et Leff sont des listes.

Cette procédure détermine si X est un élément de L et Leff est la liste L sans la première occurence de X dans L.

Conditions d'application

in(any, ground, any) : out(ground, ground, ground) <0-n>
in(ground, any, ground) : out(ground, ground, ground) <0-n>

Le but des manipulations suivantes est de varier les conditions dans lesquelles l'optimisation a lieu. Soit la procédure efface/3 de directionalité suivante:

in(ground, ground, any) : out(ground, ground, ground)
in(any, ground, any) : out(ground, ground, ground)

```
efface(X,L,Leff) :- L = [ H | T ] , X = H , Leff = T .                          (1)
efface(X,L,Leff) :- L = [ H | T ] , efface(X,T,Teff) , not X = H , Leff = [ H | Teff ] .
```

Par l'optimiseur, nous obtenons la procédure

in(ground, ground, any) : out(ground, ground, ground)

```
efface(H,[ H | T ],Leff) :- ! , Leff = T .                                       (2)
efface(X,[ H | T ],[ H | Teff ]) :- efface(X,T,Teff) .
```

Si nous voulons une directionalité

in(ground, ground, var) : out(ground, ground, ground)

Leff = T devient entièrement déterministe. L'optimiseur produit alors, à partir de (1), la procédure

```
efface(H,[ H | T ],T) :- ! .                                                     (3)
efface(X,[ H | T ],[ H | Teff ]) :- efface(X,T,Teff) .
```

grâce à la transformation 20.

Pour la directivité

in(any, ground, any) : out(ground, ground, ground)

l'optimiseur, à partir de (1), génère

```
efface(H,[ H | T ],T) .                                                   (4)
efface(X,[ H | T ],Leff) :- efface(X,T,Teff) , not X = H , Leff = [ H | Teff ] .
```

car il n'existe pas de permutation adéquate sinon X = H n'est pas déterministe et la procédure ne serait pas correcte (cf la transformation 5).

Construisons, en outre, une procédure efface/3 qui ait la directivité

in(ground, any, ground) : out(ground, ground, ground)

```
efface(X,L,Leff) :- L = [ H | T ] , X = H , Leff = T .                     (5)
efface(X,L,Leff) :- L = [ H | T ] , Leff = [ H | Teff ] , efface(X,T,Teff) , not X = H
```

La procédure est correcte car X = H est déterministe puisque H et X sont ground avant le littéral not X = H. Pour cette directionalité, l'optimiseur nous donne

```
efface(H,[ H | T ],T) .                                                   (6)
efface(X,[ H | T ],[ H | Teff ]) :- not X = H , efface(X,T,Teff) .
```

L'optimiseur produit, à la place de (6), la procédure suivante si nous refusons la permutation des littéraux proposée par la transformation 22 (cf le module genre).

```
efface(H,[ H | T ],T) .                                                   (7)
efface(X,[ H | T ],[ H | Teff ]) :- efface(X,T,Teff) , not X = H .
```

Pour ces sept procédures, le temps d'exécution du goal :- **efface(X,L,Leff)** sera déterminé pour des X, L et Leff couvrant toutes les directivités. Nous obtenons le tableau des résultats suivant:

Données de test			Temps en secondes pour 100 exécutions des procédures de numéro						
X L	Leff	resultats	1	2	3	4	5	6	7
3 [1,2,3,4,5]	[1,2,4,5]	yes	3,0	0,5	-	1,5	2,0	1,0	1,0
3 [1,2,3,4,5]	[2,3,4,5]	no	2,9	0,1	-	1,5	0,4	0,1	0,1
3 [1,2,3,4,5]	Leff	Leff = [1,2,4,5]	3,0	0,5	0,4	1,5	-	-	-
6 [1,2,3,4,5]	Leff	no	2,1	0,7	0,7	0,7	-	-	-
3 L	[1,2,4,5]	L = [3,1,2,4,5] [1,3,2,4,5] [1,2,3,4,5] [1,2,4,3,5] [1,2,4,5,3]	-	-	-	-	5,6	2,1	3,8
X [1,2,3,4,5]	[1,2,4,5]	X = 3	6,5	-	-	4,9	-	-	-
X [1,2,3,4,5]	[1,2,5]	no	6,4	-	-	4,8	-	-	-
X [1,2,3,4,5]	Leff	X = 1 Leff = [2,3,4,5] X = 2 Leff = [1,3,4,5] X = 3 Leff = [1,2,4,5] X = 4 Leff = [1,2,3,5] X = 5 Leff = [1,2,3,4]	6,6	-	-	4,8	-	-	-

Seuls les deux premiers chiffres significatifs sont donnés, les autres dépendant trop du moment où est réalisé le test.

Ce tableau est particulièrement révélateur: on y trouve des rapports de temps allant jusqu'au vingt-neuvième !

Il montre d'abord que la recherche des paires de littéraux incompatibles est très fructueuse (comparaison des temps d'exécution de la procédure (1) et (2)). Les temps des procédures (6) et (7) montrent que les transformations 12 et 22 qui essaient d'amener une procédure de genre quelconque à un genre TR peut être intéressante.

Ces résultats suggèrent enfin aux programmeurs de construire des procédures avec des directivités spécifiques au problème à résoudre (comparaison des temps d'exécution des procédures (2), (3) et (4); si

nous essayons d'exécuter la procédure (3) avec les goals qui ne lui conviennent pas, nous obtenons des séquences des substitutions résultantes incorrectes ou incomplètes).

5.3. L'expansion des littéraux d'une procédure

Cette expansion a été critiquée dans le chapitre deux. Néanmoins elle donne de bons résultats en temps. Puisqu'elle est implémentée, elle aussi, profitons-en pour étendre les littéraux d'une procédure empruntée à Leon Sterling et Ehud Shapiro dans **"The art of Prolog"**, MIT Press 1986.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y) , ancestor(Y,Z) .
parent(terach,abraham) .                                        (8)
parent(abraham,isaac) .
parent(isaac,jacob) .
parent(jacob,benjamin) .
```

L'optimiseur génère, à partir de ce programme, un autre fichier où se trouve

```
ancestor(terach,abraham) .
ancestor(abraham,isaac) .
ancestor(isaac,jacob) .
ancestor(jacob,benjamin) .
ancestor(terach, ⊥ ) :-  ancestor(abraham, ⊥ ) .
ancestor(abraham, ⊥ ) :- ancestor(isaac, ⊥ ) .
ancestor(isaac, ⊥ ) :- ancestor(jacob, ⊥ ) .                   (9)
ancestor(jacob, ⊥ ) :- ancestor(benjamin, ⊥ ) .
parent(terach,abraham) .
parent(abraham,isaac) .
parent(isaac,jacob) .
parent(jacob,benjamin) .
```

Exécutons maintenant le goal :- **ancestor(terach,benjamin)**. Par (8), le temps pour 100 exécutions est de 1,7 secondes et par (9), de 0,9 secondes. Le rapport est donc à peu près de deux. C'est un bon score mais la longueur du programme optimisé est bien deux fois plus grande. Toutes nos critiques sont aussi fondées (cf chapitres deux et trois).

5.4. Conclusions sur l'heuristique générale de transformation

Au vu du tableau des résultats, notre heuristique respecte bien la préséance que doit avoir la recherche des paires de littéraux incompatibles sur la tentative d'amener une procédure vers le genre TR. Les transformations intraclausales servent à peaufiner: à instancier au maximum la tête d'une clause, à insérer des cuts supplémentaires et à éviter des unifications superflues. Elles doivent donc être appliquées en fin de processus d'optimisation.

Quant à l'expansion des littéraux, malgré de bons résultats, elle ne doit certainement pas être menée à outrance. La procédure (8) est relativement simple et n'est donc pas un exemple typique. De plus, au vu de la référence (2), les algorithmes d'expansion peuvent devenir très compliqués pour un rapport en temps modeste (de 2) par rapport à la recherche de paires incompatibles (3 pour le plus mauvais, 29 pour le meilleur).

5.5. Un exemple de l'exécution du processus d'optimisation par l'optimiseur

Il s'agit ici de montrer le besoin en informations du processus d'optimisation: nous reprenons l'exécution qui a conduit, à partir de la procédure (1), à la procédure optimisée (3) dans le cas où le fichier source contenant la procédure (1) n'a pas le fait predicat/8 de définition.

```
Aug 25 17:25 1988  sorties Page 1


C-Prolog version 1.5
| ?- [init]
optimiseur consulted 5788 bytes 2.63333 sec.
genre consulted 5780 bytes 2.23334 sec.
geldegel consulted 3888 bytes 1.6 sec.
expansion consulted 5296 bytes 2.21667 sec.
transclause consulted 10576 bytes 4.95 sec.
notetincomp consulted 13036 bytes 5.46667 sec.
init consulted 46068 bytes 19.7333 sec.

yes

| ?- optimiseur(efface1,resultat).
Il n'existe aucune information concernant la procedure
efface/3
Cette procedure est-elle
    non-deterministe           (0)
    deterministe               (1)
    entierement deterministe   (2)
? 1.

A-t-elle des effets de bord,
oui ou non ? n.

C'est une procedure deterministe sans effet de bord
Est-ce exact
oui ou non ? o.

Peut-on transformer la clause
efface(_36,_37,_38):-_37=[_39|_40],efface(_36,_40,_41),not
_36=_39,_38=[_39|_41]
en la clause
```

```
efface(_36,_37,_38):-_37=[_39|_40],not
_36=_39,_38=[_39|_41],efface(_36,_40,_41),true
repondre par
oui ou non ? o.

Le but Var(0)=Var(3)
dans la procedure
efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],Var(0)=Var(3),Var(2)=Var(4
)
efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],not
Var(0)=Var(3),Var(2)=[Var(3)|Var(5)],efface(Var(0),Var(4),Var(5)),true
est-il sans effet de bord ?
oui ou non ? o.

Y a-t-il dans la procedure

efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],Var(0)=Var(3),Var(2)=Var(4
)
                              efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],not
Var(0)=Var(3),Var(2)=[Var(3)|Var(5)],efface(Var(0),Var(4),Var(5)),true
des paires de goals incompatibles
n'ayant aucun effet de bord et dont celui de la 1ere
clause est deterministe ou entierement deterministe ?
Entrer ces paires dans une liste de couples d'entiers
du type [(1,4),(3,5),..], ou 1 et 4 correspondent a la
position de ces goals dans la 1ere et 2d clause
respectivement. [] s'il n'y a pas de couples.
|: [].


Aug 25 17:25 1988  sorties Page 2


Pour la clause
efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],Var(0)=Var(3),Var(2)=Var(4
)
entrer sous forme de liste les permutations permises
des buts du corps de la clause.
Ces listes seront du type [2,3,5,1,4,6] ou 2 est le
2e but du corps de la clause ayant permute a la 1e place
[] pour terminer
|: [].
Ok
```

```
Pour la clause
efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],not
Var(0)=Var(3),Var(2)=[Var(3)|Var(5)],efface(Var(0),Var(4),Var(5)),true
memes operations
|: [].
Ok

Le but Var(1)=[Var(3)|Var(4)]
dans la procedure
efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],Var(0)=Var(3),Var(2)=Var(4
)
efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],not
Var(0)=Var(3),Var(2)=[Var(3)|Var(5)],efface(Var(0),Var(4),Var(5)),true
est-il sans effet de bord et det. ou ent. det.
oui ou non ? o.

Le but Var(2)=Var(4)
dans la clause
efface(Var(0),Var(1),Var(2)):-Var(1)=[Var(3)|Var(4)],Var(0)=Var(3),!,Var(2)=Var(
4),true,true
est-il sans effet de bord et ent deterministe
oui ou non ? o.

Le fichier resultat est cree
Fin de l'Optimisation_
yes
| ?- halt.

[ Prolog execution halted ]

*** fin de l'execution ***
```

6
Conclusion

L'optimisation des programmes au niveau source est donc une étape nécessaire dans l'évolution d'un programme. Celle présentée dans ce mémoire est aussi une étape naturelle dans la méthodologie de programmation présentée dans la référence ((1)).

Les principaux axes de recherche dans ce domaine sont d'abord de trouver de nouvelles transformations, de trouver de nouvelles paires de littéraux incompatibles et, en conséquence, de modifier l'heuristique de transformation. L'heuristique de transformation du chapitre 3 exploite au maximum les transformations d'optimisation du chapitre 2 mais n'est donc que temporaire, je le crains. Tout dépend de l'idée de base. Celle qui sert de charpente à l'heuristique de la référence ((2)) a été critiquée dans le chapitre 3. La nôtre accorde un billet de faveur en première loge aux paires de littéraux incompatibles.

D'autres axes de recherche concernent des problèmes plus fondamentaux tels les algorithmes suffisants de détection du déterminisme des procédures, les modes de déclaration… L'optimiseur implémenté serait d'autant plus pratique !

En conclusion, le sujet est loin d'être clos et mon mémoire n'est qu'une pincée de feuilles dans l'automne des cogitations intenses.

Références

((1)) V. Deville

"A Methodology for Logic Program Construction"

Thèse de doctorat, université de Namur, 1987

((2)) H. Sawamura, T. Takeshima, A. Kato

"Source-level Optimization Techniques for Prolog"

Numazu, Shizuoka, 1985

Annexe: Le code de l'optimiseur

Dans les pages qui suivent se trouve le code de l'optimiseur. Celui-ci a été écrit en **C-Prolog version 1.5** sur **Vax-750**.

```
r rrr     m m mm
rr   r    mm m  m
r         m  m  m
r         m  m  m
r         m  m  m
r         m  m  m
```

```
                    t          i                 i
                    t
 oooo     p ppp   ttttt        ii     m m mm     ii     zzzzzz     eeee    r rrr
o    o    pp   p    t           i     mm m  m     i          z    e    e  rr   r
o    o    p    p    t           i     m  m  m     i         z     eeeeee  r
o    o    p    p    t           i     m  m  m     i        z      e       r
o    o    pp   p    t  t        i     m  m  m     i       z       e    e  r
 oooo     p ppp      tt        iii    m  m  m    iii     zzzzzz    eeee   r
          p
          p
          p
```

```
  ff
 f  f
 f
 f       u    u   n nnn
fffff    u    u   nn   n
 f       u    u   n    n
 f       u    u   n    n
 f       u   uu   n    n
 f        uuu u   n    n
```

```
/*****************************************************************************/
/**                                 -init-                                  **/
/*****************************************************************************/


:- op(1150,fx,[\]).

\ o :-
        system("emacs optimiseur"),[-optimiseur].
\ g :-
        system("emacs genre"),[-genre].
\ d :-
        system("emacs geldegel"),[-geldegel].
\ t :-
        system("emacs transclause"),[-transclause].
\ e :-
        system("emacs expansion"),[-expansion].
\ n :-
        system("emacs notetincomp"),[-notetincomp].
\ i :-
        system("emacs init"),[-init].

:- [optimiseur,genre,geldegel,expansion,transclause,notetincomp].


/*****************************************************************************/
/**                             -optimiseur-                                **/
/*****************************************************************************/


oracle_oui_non(X,Rep):-
        nl,write(X),nl,write('oui ou non ? '),
        read(X1),name(X1,[Y|_]),oracle_rep(Y,Rep),!.
oracle_rep(111,1).
oracle_rep(121,1).
oracle_rep(_,0).


optimiseur(Iin,Oout):-
        verifin(Iin,In),verifout(Oout,Out),
        chargement(In),
        genre,
        notetincomp,
        expansion,
        remplissage(Out),
        oracle4(Out).
```

```
verifin(In,In):-
        exists(In),!.
verifin(IIn,In):-
        oracle1(Nom),verifin(Nom,In).

oracle1(Nom):-
        nl,write('Le fichier d''entree n''existe pas'),
        nl,write('Entrer un autre nom: '),read(Nom).

verifout(Out,Out):-
        not(exists(Out)),!.
verifout(Oout,Out):-
        oracle2(Oout,Rep),verifout(Rep,Oout,Out).
verifout(1,Out,Out).
verifout(0,Oout,Out):-
        oracle3(Nom),verifout(Nom,Out).

oracle2(Out,Rep):-
        nl,write('Le fichier de sortie existe deja'),
        oracle_oui_non('Gardez-vous le meme nom',Rep).

oracle3(Nom):-
        nl,write('Quel est ce nom ? '),read(Nom).

oracle4(Nom):-
        nl,write('Le fichier '),write(Nom),write(' est cree'),
        nl,write('Fin de l''Optimisation_').

chargement(In):-
        see(In),assert(compteur(0)),!,
        repeat,incrcompt,read(C),
        charge_clause(C),retract(compteur(_)),!,seen,!.
charge_clause('end_of_file'):-
        !.
charge_clause(predicat(X,Y,Z,T,U,V,W,S)):-
        \+ predicat(X,Y,_,_,_,_,_,_),
        assert(predicat(X,Y,Z,T,U,V,W,S)),!,fail.
charge_clause(predicat(_,_,_,_,_,_,_,_)):-
        !,fail.
charge_clause((H:-B)):-
        charge((H:-B)),!,fail.
charge_clause(C):-
        charge((C:-true)),!,fail.
charge((H:-B)):-
        functor(H,Nom,Arity),
        retract(predicat(Nom,Arity,D,S,I,G,N,L)),!,N1 is N+1,
        assert(predicat(Nom,Arity,D,S,I,G,N1,[((H:-B),_)|L])).
charge((H:-B)):-
        oracle5((H:-B),P),assert(P),charge((H:-B)).



incrcompt:-
        retract(compteur(I)),!,I1 is I+1,assert(compteur(I1)),!.
```

```
itread(0):-
        !.
itread(I):-
        read(_),I1 is I-1,itread(I1).


oracle5((H:-B),predicat(Nom,Arity,R1,R2,R3,G,0,[])):-
        seeing(F),seen,functor(H,Nom,Arity),
        nl,write('Il n''existe aucune information concernant la procedure '),
        nl,write(Nom),write('/'),write(Arity),oracle5_det(Rrr1),
        oracle5_ani(Rrr1,Rr1,Rr3),oracle5_eff(Rr2),
        oracle5_exact(Rr1,Rr2,Rr3,R1,R2,R3),see(F),
        compteur(Cpt),itread(Cpt).
oracle5_det(R):-
        nl,write('Cette procedure est-elle'),
        nl,tab(5),write('non-deterministe          (0)'),
        nl,tab(5),write('deterministe              (1)'),
        nl,tab(5),write('entierement deterministe  (2)'),
        nl,write('? '),read(R).
oracle5_ani(2,2,0):-!.
oracle5_ani(1,1,0):-!.
oracle5_ani(0,0,Rep):-
        oracle_oui_non('Cette procedure est-elle infinie',Rep),!.
oracle5_ani(_,R1,Rep):-
        nl,write('reponse incorrecte'),
        oracle5_det(R),oracle5_ani(R,R1,Rep).
oracle5_eff(Rep):-
        oracle_oui_non('A-t-elle des effets de bord,',Rep).
oracle5_exact(Xx,Yy,Zz,X,Y,Z):-
        oracle5_verif(Xx,Yy,Zz,M),nl,write(M),
        oracle_oui_non('Est-ce exact',Rep),oracle5_route(Rep,Xx,Yy,Zz,X,Y,Z).
oracle5_route(1,X,Y,Z,X,Y,Z).
oracle5_route(0,_,_,_,R1,R2,R3):-
        oracle5_det(Rrr1),oracle5_ani(Rrr1,Rr1,Rr3),oracle5_eff(Rr2),
        oracle5_exact(Rr1,Rr2,Rr3,R1,R2,R3).
oracle5_verif(0,0,0,'C''est une procedure non-deterministe,sans effet de bord
et non infinie').
oracle5_verif(0,1,0,'C''est une procedure non-deterministe,avec effet de bord
et non infinie').
oracle5_verif(0,1,1,'C''est une procedure non-deterministe,avec effet de bord
et infinie').
oracle5_verif(0,0,1,'C''est une procedure non-deterministe,sans effet de bord
et infinie').
oracle5_verif(1,0,0,'C''est une procedure deterministe sans effet de bord').
oracle5_verif(1,1,0,'C''est une procedure deterministe avec effet de bord').
oracle5_verif(2,0,0,'C''est une procedure entierement deterministe sans
effet de bord').
oracle5_verif(2,1,0,'C''est une procedure entierement deterministe avec
effet de bord').


remplissage(Out):-
        tell(Out),retract(predicat(_,_,_,_,_,_,_,L)),sortie(L).
remplissage(Out):-
        told.
sortie([]):-
```

```
        !,fail.
sortie([((X:-true),_)|T]):-
        nl,degel(X,C),write(C),write('.'),!,sortie(T).
sortie([(X,_)|T]):-
        nl,degel(X,C),write(C),write('.'),!,sortie(T).




/*******************************************************************************/
/**                                  -genre-                                  **/
/*******************************************************************************/



genre:-
        predicat(F,A,D,E,I,G,Nc,L),gp(F,A,D,E,I,G,Nc,L).
genre.


gp(F,A,D,E,If,G,Nc,L1):-
        genrepr(F,A,L1,[],L2,[[],[],[]],L3,G1,G,0,I),
        choixliste(L2,L3,I,L),!,
        retract(predicat(F,A,D,E,If,G,Nc,L1)),!,
        asserta(predicat(F,A,D,E,If,G,Nc,L)),!,fail.

genrepr(F,A,[],X,X,Y,Y,G,G,I,I).
genrepr(F,A,[(X,_)|L],X1,X2,Y1,Y2,G1,G2,I1,I2):-
        genrecl(X,F,A,Hh,Gc),gel(Hh,H,Ns1),
        lpp((H,Ns1),Gc,G1,G,I1,I,Y1,Y),!,
        genrepr(F,A,L,[(H,Ns1)|X1],X2,Y,Y2,G,G2,I,I2).

choixliste(L,_,0,L).
choixliste(L2,L3,1,L):-
        mepl(L3,L4),oracle6(L2,L4,L).


oracle6(L1,L2,L):-
        nl,write('Pour la procedure'),oracle6affl(L1),
        nl,write('peut-on envisager la permutation des clauses suivantes ?'),
        oracle6affl(L2),oracle_oui_non('repondre par',Rep),
        oracle6concl(Rep,L1,L2,L).
oracle6concl(0,L,_,L).
oracle6concl(1,_,L,L).
oracle6affl([]).
oracle6affl([(X,_)|L]):-
        nl,tab(2),write(X),write('.'),oracle6affl(L).


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

```
mepl([[],[],[]],[]).


lpp(C,0,0,0,I,I,[Nr1,Tr,Gr],[[C|Nr1],Tr,Gr]).
lpp(C,0,X,X,I,I,[Nr1,Tr,Gr],[[C|Nr1],Tr,Gr]).
lpp(C,1,2,2,I,I,[Nr,Tr1,Gr],[Nr,[C|Tr1],Gr]).
lpp(C,1,1,1,I,I,[Nr,Tr1,Gr],[Nr,[C|Tr1],Gr]).
lpp(C,1,0,1,I,1,[Nr,Tr1,Gr],[Nr,[C|Tr1],Gr]).
lpp(C,2,2,2,I,I,[Nr,Tr,Gr1],[Nr,Tr,[C|Gr1]]).
lpp(C,2,1,2,I,1,[Nr,Tr,Gr1],[Nr,Tr,[C|Gr1]]).
lpp(C,2,0,2,I,1,[Nr,Tr,Gr1],[Nr,Tr,[C|Gr1]]).

genrecl((H:-B),F,A,C,G):-
        genrecl(B,F,A,[],L1,0,Gp,Gv),
        linear(L1,true,Bb),choixcl((H:-B),(H:-Bb),Gp,Gv,C,G).

genrecl(T,F,A,L,[T|L],0,1,1):-
        functor(T,F,A),!.
genrecl(T,F,A,[X|L],[(T,X)|L],_,2,2):-
        functor(T,F,A),!.
genrecl((T,B),F,A,L1,L,0,G,P):-
        functor(T,F,A),!,genrecl(B,F,A,[T|L1],L,1,G,P).
genrecl((T,B),F,A,[X|L1],L,_,2,P):-
        functor(T,F,A),!,genrecl(B,F,A,[(T,X)|L1],L,2,2,P).
genrecl((R,B),F,A,L1,L,0,G,P):-
        genrecl(B,F,A,[R|L1],L,0,G,P),!.
genrecl((R,B),F,A,[X|L1],L,Gg,G,P):-
        genrecl(B,F,A,[X,R|L1],L,Gg,G,P).
genrecl(R,F,A,L,[R|L],0,0,0):-
        !.
genrecl(R,F,A,[X|L],[X,R|L],G,G,2).


choixcl(X,Y,Gp,2,C,G):-
        !,oracle7(X,Y,Gp,C,G).
choixcl(X,_,_,Y,X,Y).


oracle7(X,Y,Gp,C,G):-
        nl,write('Peut-on transformer la clause'),nl,write(X),
        nl,write('en la clause'),nl,write(Y),
        oracle_oui_non('repondre par',Rep),oracle7concl(X,Y,Gp,C,G,Rep).
oracle7concl(_,C,G,C,G,1).
oracle7concl(X,_,_,X,2,0).


linear([(X,Y)|T],B1,B):-
        !,linear([Y|T],(X,B1),B).
linear([X|T],B1,B):-
        !,linear(T,(X,B1),B).
linear([],B,B).



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

```
/**                              -geldegel-                                **/
/****************************************************************************/


constant(X):-
        atomic(X).
compound(X):-
        nonvar(X),\+ atomic(X).

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


member(X,[X|_]).
member(X,[_|L]):-
        member(X,L).



gel(A,B,Nsup):-
        copy(A,B),numbervars(B,0,Nsup).
copy(A,B):-
        assert(acc(A)),retract(acc(B)).
numbervars('Var'(N),N,N1):-
        N1 is N+1.
numbervars(Term,N1,N2):-
        nonvar(Term),functor(Term,Name,N),numbervars(0,N,Term,N1,N2).
numbervars(N,N,Term,N1,N1).
numbervars(I,N,Term,N1,N3):-
        N > I,I1 is I+1,arg(I1,Term,Arg),numbervars(Arg,N1,N2),
        numbervars(I1,N,Term,N2,N3).



nomtranslat('Var'(N),'Var'(N2),N1):-
        N2 is N+N1.
nomtranslat(X,X,_):-
        constant(X),!.
nomtranslat(X,Y,I):-
        compound(X),functor(X,F,N),functor(Y,F,N),nomtranslat(N,X,Y,I).
nomtranslat(N,X,Y,I):-
        N > 0,arg(N,X,Argx),nomtranslat(Argx,Argy,I),arg(N,Y,Argy),
        N1 is N-1,nomtranslat(N1,X,Y,I).
nomtranslat(0,X,Y,_).


degel(X,Y):-
        degelnet1(X,[],L1),listesansrep(L1,[],L),degelnet2(X,Y,L).
degelnet1('Var'(N),L1,L):-
        listerep(N,L1,[],L),!.
degelnet1(X,L1,L):-
```

```
        compound(X),functor(X,F,N),degelnet1(N,X,L1,L),!.
degelnet1(X,L,L).
degelnet1(N,X,L1,L):-
        N > 0,arg(N,X,Argx),degelnet1(Argx,L1,L2),
        N1 is N-1,degelnet1(N1,X,L2,L).
degelnet1(0,X,L,L).


listerep(X,[(X,_)|T],L1,[(X,1)|C]):-
        concatenate(T,L1,C),!.
listerep(X,[Y|T],L1,L):-
        listerep(X,T,[Y|L1],L),!.
listerep(X,[],L,[(X,0)|L]).


listesansrep([(X,1)|T],L1,L):-
        listesansrep(T,L1,L).
listesansrep([(X,0)|T],L1,L):-
        listesansrep(T,[X|L1],L).
listesansrep([],L,L).


degelnet2('Var'(N),X,L):-
        member(N,L),!,name(X,[95]).
degelnet2('Var'(N),X,L):-
        name(N,L1),name(X,[95|L1]).
degelnet2(X,X,_):-
        constant(X).
degelnet2(X,Y,L):-
        compound(X),functor(X,F,N),functor(Y,F,N),degelnet2(N,X,Y,L).
degelnet2(N,X,Y,L):-
        N > 0,arg(N,X,Argx),degelnet2(Argx,Argy,L),
        arg(N,Y,Argy),N1 is N-1,degelnet2(N1,X,Y,L).
degelnet2(0,X,Y,L).



/**********************************************************************************
/**                              -notetincomp-                                   **
/**********************************************************************************



notetincomp:-
        predicat(Nm,A,D,E,If,Gr,2,L),
        neti(Nm,A,D,E,If,Gr,2,L).
notetincomp.


neti(Nm,A,D,E,If,Gr,2,[(Cl1,Ns1),(Cl2,Ns2)]):-
        assert(cl(Cl1,Cl2)),constrlcg(Lcg,Lt1i,Lt2i),
        oracle10(Lcg,Lt1i,Lt2i,Lt1,Lt2),trouvertransf(Lt1,Lt2,[Cl1t,Cl2t]),
        retract(predicat(Nm,A,D,E,If,Gr,2,[(Cl1,Ns1),(Cl2,Ns2)])),
        asserta(predicat(Nm,A,D,E,If,Gr,2,[(Cl1t,Ns1),(Cl2t,Ns2)])),
        retract(cl(_,_)),!,retract(corps1(_)),!,retract(corps2(_)),!,fail.
neti(_,_,_,_,_,_,_,_):-
```

```
        abolish(cl,2),!,retract(corps1(_)),!,retract(corps2(_)),!,fail.


constrlcg(Lcg,Lt1,Lt2):-
        cl((H1:-B1),(H2:-B2)),
        makelist1(B1,Lb1,Lapl1,B2,Lcg1,1),makelist2(B2,Lb2,Lapl2,Lb1,Lcg2,1),
        assert(corps1(Lb1)),assert(corps2(Lb2)),
        concatenate(Lcg1,Lcg2,Lcg3),oracle9(Lcg3,Lcg),
        tripermgoal(1,Lapl1,[],Lt1,Lcg),tripermgoal(2,Lapl2,[],Lt2,Lcg),!,
        lcgvide(Lcg).

lcgvide([]):-
        !,fail.
lcgvide(_).


makelist1((not(C),Y),[not(C)|T],[I|Tt],B,L,I):-
        I1 is I+1,!,rempll1(C,I,B,L,L1),makelist1(Y,T,Tt,B,L1,I1).
makelist1((X,Y),[X|T],[I|Tt],B,L,I):-
        I1 is I+1,!,makelist1(Y,T,Tt,B,L,I1).
makelist1(not(C),[not(C)],[I],B,L,I):-
        rempll1(C,I,B,L,L1),!,L1=[].
makelist1(X,[X],[I],B,[],I).


rempll1(C,I,B,[(I,J)|L],L):-
        mofconj(C,B,1,J),functor(C,N,A),oracle9se(C,N,A),!.
rempll1(C,I,B,L,L).



mofconj(X,(X,T),J,J):-
        !.
mofconj(X,(_,T),J,J1):-
        J2 is J+1,!,mofconj(X,T,J2,J1).
mofconj(X,X,J,J).



makelist2((not(C),Y),[not(C)|T],[I|Tt],B,L,I):-
        I1 is I+1,!,rempll2(C,I,B,L,L1),makelist2(Y,T,Tt,B,L1,I1).
makelist2((X,Y),[X|T],[I|Tt],B,L,I):-
        I1 is I+1,!,makelist2(Y,T,Tt,B,L,I1).
makelist2(not(C),[not(C)],[I],B,L,I):-
        rempll2(C,I,B,L,L1),!,L1=[].
makelist2(X,[X],[I],B,[],I).


rempll2(C,I,B,[(J,I)|L],L):-
        membernbr(C,B,1,J),functor(C,N,A),oracle9se(C,N,A),!.
rempll2(C,I,B,L,L).


membernbr(X,[X|_],I,I):-
        !.
membernbr(X,[_|L],I,I1):-
```

```
        I2 is I+1,membernbr(X,L,I2,I1).


oracle9se(X,N,A):-
        predicat(N,A,_,E,_,_,_,_),!,E=0.
oracle9se(X,N,A):-
        nl,write('Le but '),write(X),cl(Cl1,Cl2),
        nl,write('dans la procedure'),nl,write(Cl1),nl,write(Cl2),
        oracle_oui_non('est-il sans effet de bord ?',1).


oracle9(L,L1):-
        nl,write('Y a-t-il dans la procedure'),
        cl(Cl1,Cl2),nl,tab(2),write(Cl1),nl,tab(2),write(Cl2),
        nl,write('des paires de goals incompatibles'),
        nl,write('n''ayant aucun effet de bord et dont celui de la 1ere'),
        nl,write('clause est deterministe ou entierement deterministe ?'),
        nl,write('Entrer ces paires dans une liste de couples d''entiers'),
        nl,write('du type [(1,4),(3,5),..], ou 1 et 4 correspondent a la '),
        nl,write('position de ces goals dans la 1ere et 2d clause '),
        nl,write('respectivement. [] s''il n''y a pas de couples.'),
        nl,read(Lr),concatenate(L,Lr,L1).



oracle10(Lcg,Lt1i,Lt2i,Lt1,Lt2):-
        oracle10cl1(Lcg,Lt1i,Lt1),oracle10cl2(Lcg,Lt2i,Lt2).

oracle10cl1(Lcg,Lt1,Lt2):-
        nl,write('Pour la clause'),cl(Cl,_),
        nl,write(Cl),
        nl,write('entrer sous forme de liste les permutations permises'),
        nl,write('des buts du corps de la clause.'),
        nl,write('Ces listes seront du type [2,3,5,1,4,6] ou 2 est le '),
        nl,write('2e but du corps de la clause ayant permute a la 1e place'),
        nl,write('[] pour terminer'),
        nl,read(X),oracle10verif1(X,Lcg,Lt1,Lt2),nl,write('Ok').


oracle10cl2(Lcg,Lt1,Lt2):-
        nl,write('Pour la clause'),cl(_,Cl),
        nl,write(Cl),nl,write('memes operations'),
        nl,read(X),oracle10verif2(X,Lcg,Lt1,Lt2),nl,write('Ok').


oracle10verif1([],_,L,L):-
        !.
oracle10verif1(X,Lcg,Lt1,Lt2):-
        tripermgoal(1,X,Lt1,Lt3,Lcg),nl,write('autre permutation'),
        nl,read(Y),oracle10verif1(Y,Lcg,Lt3,Lt2).

oracle10verif2([],_,L,L):-
        !.
oracle10verif2(X,Lcg,Lt1,Lt2):-
        tripermgoal(2,X,Lt1,Lt3,Lcg),nl,write('autre permutation'),
        nl,read(Y),oracle10verif2(Y,Lcg,Lt3,Lt2).
```

```
trouvertransf(L1,L2,L):-
        selectprn(L1,L2,L2,Lt1,Lt2,L1n,L2n,N),
        itersurn(L1n,L2n,Lt1,Lt2,N,L),!.
trouvertransf(_,_,[Cl1,Cl2]):-
        cl(Cl1,Cl2).

itersurn(L1,L2,Lr1,Lr2,N,L):-
        selectprn(L1,L2,L2,Lt1,Lt2,L1g,L2g,G),
        itersurg(L1g,L2g,Lt1,Lt2,N,G,L),!.
itersurn(_,_,Lr1,Lr2,_,L):-
        trouvertransf(Lr1,Lr2,L),!.

itersurg(L1,L2,Lr1,Lr2,N,G,L):-
        prodcltrans(L1,L2,N,G,L),!.
itersurg(_,_,Lr1,Lr2,_,_,L):-
        itersurn(Lr1,Lr2,_,_,_,L),!.


tripermgoal(1,Lapl,Lt1,Lt2,Lcg):-
        triprnetg1(Lapl,Lapl,Lcg,1,Lt1,Lt2).
tripermgoal(2,Lapl,Lt1,Lt2,Lcg):-
        triprnetg2(Lapl,Lapl,Lcg,1,Lt1,Lt2).

triprnetg1([],_,_,_,Lt,Lt):-
        !.
triprnetg1([X|L],Lapl,Lcg,I,Lt1,Lt2):-
        memberoflcg((X,Y),Lcg),!,trin(Lapl,(X,Y),I,Lt1,Lt3),I1 is I+1,
        triprnetg1(L,Lapl,Lcg,I1,Lt3,Lt2).
triprnetg1([_|L],Lapl,Lcg,I,Lt1,Lt2):-
        I1 is I+1,triprnetg1(L,Lapl,Lcg,I1,Lt1,Lt2).



triprnetg2([],_,_,_,Lt,Lt):-
        !.
triprnetg2([Y|L],Lapl,Lcg,I,Lt1,Lt2):-
        memberoflcg((X,Y),Lcg),!,trin(Lapl,(X,Y),I,Lt1,Lt3),I1 is I+1,
        triprnetg2(L,Lapl,Lcg,I1,Lt3,Lt2).
triprnetg2([_|L],Lapl,Lcg,I,Lt1,Lt2):-
        I1 is I+1,triprnetg2(L,Lapl,Lcg,I1,Lt1,Lt2).



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


trin(Lapl,G,N,[],[([([Lapl],G)],N)]).
trin(Lapl,G,N,[(X,N1)|T],[([([Lapl],G)],N),(X,N1)|T]):-
        N < N1.
trin(Lapl,G,N,[(L,N)|T],[(L1,N)|T]):-
```

```
        trig(Lapl,G,L,L1).
trin(Lapl,G,N,[(X,N1)|T],[(X,N1)|T1]):-
        trin(Lapl,G,N,T,T1).


trig(Lapl,G,[],[([Lapl],G)]).
trig(Lapl,(G1,G2),[(X,(Gg1,Gg2))|T],[([Lapl],(G1,G2)),(X,(Gg1,Gg2))|T]):-
        G1 < Gg1.
trig(Lapl,(G1,G2),[([X|T],(G1,G2))|T],[([Lapl,X|T],(G1,G2))|T]).
trig(Lapl,G,[(X,G1)|T],[(X,G1)|T1]):-
        trig(Lapl,G,T,T1).


selectprn([],_,_,_,_,_,_,_):-
        !,fail.
selectprn(_,[],_,_,_,_,_,_):-
        !,fail.
selectprn([(L1,N)|T1],[(L2,N)|T2],_,T1,T2,L1,L2,N):-
        !.
selectprn([X|T1],[_|T2],Lr,Lt1,Lt2,L1,L2,Nf):-
        selectprn([X|T1],T2,Lr,Lt1,Lt2,L1,L2,Nf),!.
selectprn([_|T1],_,Lr,Lt1,Lt2,L1,L2,Nf):-
        !,selectprn(T1,Lr,Lr,Lt1,Lt2,L1,L2,Nf),!.


prodcltrans([],_,_,_,_):-
        !,fail.
prodcltrans(_,[],_,_,_):-
        !,fail.
prodcltrans([X|_],[Y|_],N,(G1,G2),[(H:-B1),(H:-B2)]):-
        makeprefixe(1,X,1,N,[],L1,Ss1),makeprefixe(2,Y,1,N,[],L1,Ss2),
        verifsedet(L1),!,makegt(G1,G2,Gt1,Gt2),
        makesuffixe(1,Ss1,S1),makesuffixe(2,Ss2,S2),cl((H:-_),_),
        linear(L1,(Gt1,!,S1),B1),linear(L1,(Gt2,S2),B2),!.
prodcltrans([_|T1],T2,N,G,L):-
        prodcltrans(T1,T2,N,G,L),!.


verifsedet([]):-
        !.
verifsedet([X|T]):-
        functor(X,N,A),oraclell(X,N,A),verifsedet(T).

oraclell(X,N,A):-
        predicat(N,A,D,E,_,_,_,_),!,oraclellpconcl(D,E).
oraclell(X,N,A):-
        nl,write('Le but '),write(X),cl(Cl1,Cl2),
        nl,write('dans la procedure'),nl,write(Cl1),nl,write(Cl2),
        oracle_oui_non('est-il sans effet de bord et det. ou ent. det.',1)

oraclellpconcl(1,0):-
        !.
oraclellpconcl(2,0).
```

```
makeprefixe(_,[X|T],N,N,L,L,T):-
        !.
makeprefixe(1,[X|T],I,N,L,L1,S):-
        xversg(1,X,G),I1 is I+1,makeprefixe(1,T,I1,N,[G|L],L1,S),!.
makeprefixe(2,[X|T],I,N,L,L1,S):-
        xversg(2,X,G),I1 is I+1,makeprefixe(2,T,I1,N,[G|L],L1,S).


makegt(G1,G2,Gt1,Gt2):-
        xversg(1,G1,X),xversg(2,G2,Y),mgtconcl(X,Y,Gt1,Gt2).

mgtconcl(not(C),C,not(C),true):-!.
mgtconcl(C,not(C),C,true):-!.
mgtconcl(X,Y,X,Y).


makesuffixe(_,[],true):-
        !.
makesuffixe(1,[X|T],(G,B)):-
        xversg(1,X,G),makesuffixe(1,T,B),!.
makesuffixe(2,[X|T],(G,B)):-
        xversg(2,X,G),makesuffixe(2,T,B),!.


xversg(1,X,G):-corps1(L),selectprpos(1,X,L,G).
xversg(2,X,G):-corps2(L),selectprpos(1,X,L,G).


selectprpos(X,X,[G|_],G):-
        !.
selectprpos(I,X,[_|T],G):-
        I1 is I+1,selectprpos(I1,X,T,G).




/*************************************************************************************/
/**                              -expansion-                                       **/
/*************************************************************************************/


expansion:-
        expansionsl,expansiontr,expansiongr,!.


expansionsl:-
        predicat(N,A,D,E,I,O,Nc,L),exsl(N,A,D,E,I,O,Nc,L).
expansionsl.
```

```
expansiontr:-
        predicat(N,A,D,E,I,1,Nc,L),exsl(N,A,D,E,I,1,Nc,L).
expansiontr.


expansiongr:-
        predicat(N,A,D,E,I,2,Nc,L),exsl(N,A,D,E,I,2,Nc,L).
expansiongr.



exsl(N,A,D,E,I,G,Nc,L):-
        exl(N,A,L,L1),flatten(L1,L2,0,Nc1),!,
        retract(predicat(N,A,D,E,I,G,Nc,L)),!,
        asserta(predicat(N,A,D,E,I,G,Nc1,L2)),!,fail.



exl(N,A,[((H:-B),Ns)|T],[(Lcl,Ns1)|T1]):-
        excl(B,B1,[(N,A)],_,Ns,Ns1),decomposition((H:-B1),Lcl),
        exl(N,A,T,T1).
exl(N,A,[],[]).


excl((S,T),(S1,T1),L,L1,N,N1):-
        !,excl(S,S1,L,L2,N,N2),excl(T,T1,L2,L1,N2,N1).
excl((S;T),(S1;T1),L,L1,N,N1):-
        !,excl(S,S1,L,L2,N,N2),excl(T,T1,L2,L1,N2,N1).
excl(S,S1,L,L1,N,N1):-
        tcl(S,S1,L,L1,N,N1).


tcl(S,S1,Lg,Lg1,Ns,Ns1):-
        functor(S,N,A),predicat(N,A,D,E,I,0,Nc,L),validexp(Lg,L),!,
        constrdis(L,S,Ns,Ds,Ns2),excl(Ds,S1,[(N,A)|Lg],Lg1,Ns2,Ns1).
tcl(S,S,Lg,Lg,Ns,Ns).


validexp(_,[]).
validexp(L,[((H:-B),_)|T]):-
        validexpcl(L,B),validexp(L,T).

validexpcl(L,(!,T)):-
        !,fail.
validexpcl(L,(X,T)):-
        functor(X,F,A),\+ member((F,A),L),validexpcl(L,T).
validexpcl(L,X):-
        functor(X,F,A),\+ member((F,A),L).


constrdis(L,S,Ns,D,Ns1):-
        cdis(L,S,Ns,D1,Ns1),failure(D1,D).

cdis([((H1:-B1),Ns1)],S,Ns,(S=H2,B2),Ns2):-
        nomtranslat((H1:-B1),(H2:-B2),Ns),Ns2 is Ns+Ns1,
        transclause(test,_,[],Ls,(S=H),_,0,_),Ls=[true],!.
```

```
cdis([((H:-B),_)],_,Ns,true,Ns):-
        !.
cdis([((H1:-B1),Ns1)|T],S,Ns,(S=H,B:D),Ns2):-
        nomtranslat((H1:-B1),(H:-B),Ns),Ns3 is Ns+Ns1,
        transclause(test,_,[],Ls,(S=H),_,0,_),Ls=[true],!,
        cdis(T,S,Ns3,D,Ns2).
cdis([((H1:-B1),_)|T],S,Ns,D,Ns1):-
        cdis(T,S,Ns,D,Ns1).


failure(true,fail):-
        !.
failure(D,D).


decomposition((H:-B),Lcl):-
        dec(B,[[]],L),clcl(L,Lcl,H).

dec((S,T),L,L1):-
        !,dec(S,L,L2),dec(T,L2,L1).
dec((S;T),L,L1):-
        !,dec(S,L,L2),dec(T,L,L3),concatenate(L2,L3,L1).
dec(S,L,L1):-
        empilgoal(S,L,L1).

empilgoal(S,[[]],[[S]]).
empilgoal(S,[],[]).
empilgoal(S,[B|L],[[S|B]|L1]):-
        empilgoal(S,L,L1).

clcl([],[],_).
clcl([X|T],[(H:-X1)|T1],H):-
        incorporer_1_t(X,true,X1),clcl(T,T1,H).

incorporer_1_t([],T,T).
incorporer_1_t([X|T],Terme,Termef):-
        incorporer_1_t(T,(X,Terme),Termef).


flatten([([X|[]],Ns)],[(Y,Ns)],N,N1):-
        N1 is N+1,transclause(X,Y,2),!.
flatten([([],_)|T],L,X,Y):-
        flatten(T,L,X,Y).
flatten([([X|T],Ns)|Tt],[(Y,Ns)|T1],N,N1):-
        N2 is N+1,transclause(X,Y,0),
        flatten([(T,Ns)|Tt],T1,N2,N1).




/***************************************************************************
/**                            -transclause-                              **
/***************************************************************************
```

```
transclause((H:-B),(Hm:-Bm),S):-
        assert(cltrans((H:-B))),
        transclause(H,Hm,[],[X|T],B,_,S,_),incorporer_l_t(T,X,Bm),
        retract(cltrans(_)),!.


transclause(H,Hm,[],Lsm,X=Y,_,S,S):-
        nature(X,Y,Ty,[],L),caseg1(Ty,H,Hm,Lsm,X,Y,L),!.
transclause(H,Hm,[],Ls,((X=Y),T),Tm,S1,S2):-
        nature(X,Y,Ty,[],L),caseg2(Ty,H,Htm,T,Ttm,X,Y,L),!,
        transclause(Htm,Hm,[],Ls,Ttm,Tm,S1,S2).
transclause(H,H,Ls,Lsm,X=Y,Tm,S1,S2):-
        nature(X,Y,Ty,[],L),caseg4(Ty,H,Ls,Lsm,Tm,X,Y,L,S1,S2),!.
transclause(H,H,Ls,Lsm,((X=Y),T),Tm,S1,S2):-
        nature(X,Y,Ty,[],L),caseg3(Ty,H,Ls,T,Lstm,Ttm,X,Y,L,S1,S3),!,
        transclause(H,H,Lstm,Lsm,Ttm,Tm,S3,S2).
transclause(H,Hm,Ls,Lsm,(true,T),Tm,S1,S2):-
        !,transclause(H,Hm,Ls,Lsm,T,Tm,S1,S2).
transclause(H,H,[],[true],true,_,S,S):-
        !.
transclause(H,H,Ls,[!|Ls],true,_,1,2):-
        !.
transclause(H,H,Ls,Ls,true,_,S,S):-
        !.
transclause(H,H,Ls,[fail,!|Ls],(fail,T),_,1,1):-
        !.
transclause(H,H,Ls,[fail|Ls],(fail,T),_,S,S):-
        !.
transclause(H,H,Ls,Ls,!,_,2,_):-
        !.
transclause(H,H,Ls,[!|Ls],!,_,S,S):-
        !.
transclause(H,Hm,Ls,Lsm,(!,T),Tm,S1,S2):-
        !,cascutpm(H,Hm,Ls,Lsm,(!,T),Tm,S1,S2).
transclause(H,H,Ls,Lsm,(A,T),Tm,S1,S2):-
        !,cascut(A,S1,S3,Ls,Lstm),transclause(H,H,Lstm,Lsm,T,Tm,S3,S2).
transclause(H,H,Ls,Lsm,T,_,S1,S2):-
        cascut(T,S1,S2,Ls,Lsm).


nature(X,Y,1,L,L):-
        X==Y,!.
nature('Var'(I),'Var'(J),2,L,[('Var'(I)='Var'(J))|L]):-
        !.
nature('Var'(I),Y,3,L,[('Var'(I)=Y)|L]):-
        not_occur_in('Var'(I),Y),!.
nature(X,'Var'(I),4,L,[('Var'(I)=X)|L]):-
        not_occur_in('Var'(I),X),!.
nature(X,Y,5,L,Lm):-
        compound(X),compound(Y),functor(X,F,N),functor(Y,F,N),!,
        nat_des_args(N,X,Y,L,Lm),!.
nature(X,Y,6,L,[fail]).

nat_des_args(N,X,Y,L,Lm):-
```

```
        N>0,nat_de_arg(N,X,Y,L,Ltm),N1 is N-1,nat_des_args(N1,X,Y,Ltm,Lm).
nat_des_args(0,X,Y,L,L).

nat_de_arg(N,X,Y,L,Lm):-
        arg(N,X,Argx),arg(N,Y,Argy),nature(Argx,Argy,T,L,Lm).


not_occur_in(X,Y):-
        X == Y,!,fail.
not_occur_in(X,Y):-
        compound(Y),!,functor(Y,F,N),not_occur_in(N,X,Y).
not_occur_in(X,Y):-
        X \== Y.
not_occur_in(N,X,Y):-
        N>0,arg(N,Y,Arg),not_occur_in(X,Arg),N1 is N-1,not_occur_in(N1,X,Y).
not_occur_in(0,X,Y).


caseg1(1,H,H,[true],_,_,_).
caseg1(2,H,Hm,[true],X,Y,_):-
        subst(X,Y,H,Hm).
caseg1(3,H,Hm,[true],X,Y,_):-
        subst(X,Y,H,Hm).
caseg1(4,H,Hm,[true],X,Y,_):-
        subst(Y,X,H,Hm).
caseg1(5,H,Hm,Ls,X,Y,L):-
        incorporer_l_t(L,true,Lt),transclause(H,Hm,[],Ls,Lt,Tm,0,_).
caseg1(6,H,H,[fail],_,_,_).



caseg2(1,H,H,T,T,_,_,_).
caseg2(2,H,Hm,T,Tm,X,Y,_):-
        subst(X,Y,H,Hm),subst(X,Y,T,Tm).
caseg2(3,H,Hm,T,Tm,X,Y,_):-
        subst(X,Y,H,Hm),subst(X,Y,T,Tm).
caseg2(4,H,Hm,T,Tm,X,Y,_):-
        subst(Y,X,H,Hm),subst(Y,X,T,Tm).
caseg2(5,H,H ,T,Tm,X,Y,L):-
        incorporer_l_t(L, T  ,Tm).

caseg2(6,H,H,T,(fail,T),_,_,_).




caseg3(1,_,L,T,L,T,_,_,_,S,S).
caseg3(2,H,L,T,L,Tm,X,Y,_,S,S):-
        incorporer_l_t(L,H,Lt),not_occur_in(X,Lt),!,subst(X,Y,T,Tm).
caseg3(2,H,L,T,L,Tm,X,Y,_,S,S):-
        incorporer_l_t(L,H,Lt),not_occur_in(Y,Lt),!,subst(Y,X,T,Tm).
caseg3(2,H,L,T,Lm,Tm,X,Y,_,S1,S2):-
        cascut((X=Y),S1,S2,L,Lm),subst(X,Y,T,Tm).
caseg3(3,H,L,T,L,Tm,X,Y,_,S,S):-
```

```
        incorporer_l_t(L,H,Lt),not_occur_in(X,Lt),!,subst(X,Y,T,Tm).
caseg3(3,H,L,T,Lm,Tm,X,Y,_,S1,S2):-
        cascut((X=Y),S1,S2,L,Lm),subst(X,Y,T,Tm).
caseg3(4,H,L,T,L,Tm,X,Y,_,S,S):-
        incorporer_l_t(L,H,Lt),not_occur_in(Y,Lt),!,subst(Y,X,T,Tm).
caseg3(4,H,L,T,Lm,Tm,X,Y,_,S1,S2):-
        cascut((Y=X),S1,S2,L,Lm),subst(Y,X,T,Tm).
caseg3(5,H,L,T,L,Tm,X,Y,L1,S,S):-
        incorporer_l_t(L1, T   ,Tm).
caseg3(6,H,L,T,L,(fail,T),_,_,_,_,_).



caseg4(1,_,L,L,_,_,_,_,S,S).
caseg4(2,H,L,L,_,X,_,_,S,S):-
        incorporer_l_t(L,H,Lt),not_occur_in(X,Lt),!.
caseg4(2,H,L,L,_,_,Y,_,S,S):-
        incorporer_l_t(L,H,Lt),not_occur_in(Y,Lt),!.
caseg4(2,_,L,Lm,_,X,Y,_,S1,S2):-
        cascut((X=Y),S1,S2,L,Lm).
caseg4(3,H,L,L,_,X,_,_,S,S):-
        incorporer_l_t(L,H,Lt),not_occur_in(X,Lt),!.
caseg4(3,_,L,Lm,_,X,Y,_,S1,S2):-
        cascut((X=Y),S1,S2,L,Lm).
caseg4(4,H,L,L,_,_,Y,_,S,S):-
        incorporer_l_t(L,H,Lt),not_occur_in(Y,Lt),!.
caseg4(4,_,L,Lm,_,X,Y,_,S1,S2):-
        cascut((Y=X),S1,S2,L,Lm).
caseg4(5,H,L,L,Tm,X,Y,L1,S1,S2):-
        incorporer_l_t(L1,true,Lt),transclause(H,H,L,L,Lt,tm,S1,S2).
caseg4(6,_,L,[fail|L],_,_,_,_,S,S).


subst(Old,New,Old,New).
subst(Old,New,Term,Term):-
        constant(Term),Term \== Old.
subst(Old,New,Term,Term1):-
        compound(Term),functor(Term,F,N),functor(Term1,F,N),
        subst(N,Old,New,Term,Term1).
subst(N,Old,New,Term,Term1):-
        N>0,arg(N,Term,Arg),subst(Old,New,Arg,Arg1),arg(N,Term1,Arg1),
        N1 is N-1,subst(N1,Old,New,Term,Term1).
subst(0,Old,New,Term,Term1).


cascut(G,0,0,Ls,[G|Ls]):-
        !.
cascut(G,1,1,Ls,[G|Ls]):-
        oracle8douedse(G),!.
cascut(G,1,0,Ls,[G,!|Ls]):-
        !.
cascut(G,2,1,Ls,[G|Ls]):-
        oracle8douedse(G),!.
```

```
cascut(G,2,0,Ls,[G|Ls]).


oracle8douedse(G):-
        functor(G,N,A),predicat(N,A,D,E,_,_,_,_),!,oracle8dedsecl(D,E).
oracle8douedse(G):-
        nl,write('Le but '),write(G),cltrans(Cl),nl,write('dans la clause'),
        nl,write(Cl),
        oracle_oui_non('est-il sans effet de bord et det ou ent det',1).
oracle8dedsecl(1,0).
oracle8dedsecl(2,0).


cascutpm(H,H,Ls,Ls,(!,!),_,2,2):-
        !.
cascutpm(H,H,Ls,[!|Ls],(!,!),_,S,S):-
        !.
cascutpm(H,Hm,Ls,Lsm,(!,(!,T)),Tm,S1,S2):-
        !,cascutpm(H,Hm,Ls,Lsm,(!,T),Tm,S1,S2).
cascutpm(H,Hm,Ls,Lsm,(!,(A,T)),Tm,2,S):-
        !,transclause(H,Hm,Ls,Lsm,(A,T),Tm,2,S).
cascutpm(H,Hm,Ls,Lsm,(!,(A,T)),Tm,S1,S):-
        oracle8edse(A),!,transclause(H,Hm,Ls,Lsm,(A,(!,T)),Tm,S1,S).
cascutpm(H,Hm,Ls,Lsm,(!,(A,T)),Tm,S1,S):-
        !,transclause(H,Hm,[!|Ls],Lsm,(A,T),Tm,2,S).
cascutpm(H,Hm,Ls,Lsm,(!,A),Tm,2,S):-
        !,transclause(H,Hm,Ls,Lsm,A,Tm,2,S).
cascutpm(H,Hm,Ls,Lsm,(!,A),Tm,S1,S):-
        oracle8edse(A),!,transclause(H,Hm,Ls,Lsm,(A,!),Tm,S1,S).
cascutpm(H,Hm,Ls,Lsm,(!,A),Tm,S1,S):-
        !,transclause(H,Hm,[!|Ls],Lsm,A,Tm,2,S).


oracle8edse(G):-
        functor(G,N,A),predicat(N,A,D,E,_,_,_,_),!,oracle8edsecl(D,E).
oracle8edse(G):-
        nl,write('Le but '),write(G),cltrans(Cl),nl,write('dans la clause'),
        nl,write(Cl),
        oracle_oui_non('est-il sans effet de bord et ent deterministe',1).

oracle8edsecl(2,0).
```

****

L'optimiseur au niveau source Prolog