



## THESIS / THÈSE

### MASTER EN SCIENCES INFORMATIQUES

#### Interprétation Abstraite : Analyse de Mode au moyen de Formules Logiques

Nelissen, Jean-Pierre

*Award date:*  
1991

*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.

# **Interprétation Abstraite :** Analyse de Mode au moyen de Formules Logiques

Promoteur

**Professeur Le Charlier**

Mémoire présenté par

**Jean-Pierre Nelissen**

en vue de l'obtention du grade de  
Licencié et Maître en Informatique

---

Année académique 1990-1991

# Résumé

---

Il va s'agir de réaliser un *interpréteur abstrait* capable d'*analyser statiquement des programmes logiques* : plus particulièrement, on s'intéresse à l'analyse des modes des variables dans les programmes logiques.

L'interpréteur abstrait est basé sur un ensemble d'algorithmes génériques présentés et prouvés corrects dans [Le Charlier 2]. L'utilisation de cet interpréteur à l'inférence des modes nécessite en outre la définition d'un domaine abstrait et d'opérations primitives sur ce domaine abstrait. Celui-ci est inspiré de [Marriot] et utilise des formules logiques normalisées disjonctives pour représenter les ensembles de substitutions abstraites.

L'implémentation de l'algorithme et des opérations primitives correspondantes est décrite. Le programme, réalisé en ANSI C, est testé sur quelques exemples de programmes Prolog et les résultats obtenus sont commentés.

## Abstract

The purpose of this work is to provide a description of the design and implementation of an abstract interpreter for logic programs. The interpretation is oriented toward the analysis of mode for the program variables.

It is based on a generic algorithm presented and proved correct in [Le Charlier 2]. An appropriate abstract domain has been chosen in conjunction with the corresponding abstract operations on that domain. The domain itself has been inspired by [Marriot] and uses normalised disjunctive formulas to represent sets of abstract substitutions.

The implementation of the algorithm and of its primitives has been done in ANSI C and has been tested on a few examples of Prolog programs. The results are discussed in this paper.

# Résumé

---

Abstract

<b>I. Introduction</b>	<b>1</b>
A. De l'interprétation abstraite...	1
B. De l'utilité d'un exemple sur la compréhension...	1
C. De son application à un programme logique...	2
D. Plan	3
<b>II. Présentation théorique</b>	<b>5</b>
A. Une sémantique...	5
B. Sémantique concrète	5
1. Domaine Concret	5
a. Programme logique	5
b. Substitutions normalisées	9
2. Opérations concrètes	10
a. Opérations de Restriction	10
b. Opérations d'Extension	10
c. Opération d'Union	11
d. Opérations d'Unification	11
e. Exemple	12
3. Ensembles de tuples concrets	12
4. Sémantique de point fixe	13
a. Interprétation de Herbrand	13
b. Sémantique de point fixe	13
C. Sémantique abstraite	14
1. Domaine Abstrait	14
a. Substitutions abstraites	15
2. Relation d'ordre	16
3. Opérations abstraites	16
a. Opérations de Restriction	16
b. Opérations d'Extension	17
c. Opération d'Union	17
d. Opérations d'Unification	17
e. Exemple	18
4. Ensembles de tuples abstraits	18
5. Fonction de concrétisation	19
6. Sémantique abstraite	20
D. Algorithme Générique	20
1. Présentation intuitive	20
2. Présentation formelle	23
<b>III. Implémentation</b>	<b>25</b>
A. Structures de données	25
1. Cjct	26
a. Structure	26
b. Invariant	26
c. Constructeur	26
d. Modificateur	26
e. Observateur	27
2. Formule	29
a. Structure	29
b. Invariant	29

c. Constructeur	29
d. Modificateur	30
e. Observateur	31
3. Tuple2	33
a. Structure	33
b. Invariant	33
c. Constructeur	33
d. Observateur	33
4. Tuple3	34
a. Structure	34
b. Invariant	34
c. Constructeur	34
d. Observateur	34
5. ListeTuple2	35
a. Structure	35
b. Constructeur	35
c. Modificateur	35
d. Observateur	35
6. ListeTuple3	36
a. Structure	36
b. Observateur	36
B. Opérations Primitives	37
1. Opérations Principales	37
a. SOLVE	37
b. SOLVEGOAL	37
c. SOLVEPROCEDURE	37
d. SOLVEALLCLAUSES	37
e. SOLVECLAUSE	37
2. Opérations Supplémentaires	37
a. EXTEND	37
b. ADJUST	38
c. UNION	38
d. ETA	38
3. Opérations sur le domaine	38
a. EXTC	38
b. RESTRC	38
c. EXTB	38
d. RESTRB	38
e. AIFUNC	39
f. AIVAR	39
g. LEXT	39
h. CHGVAR	39
C. Parsing	39
<b>IV. Exemples d'exécution</b>	<b>41</b>
A. Mode d'emploi	41
B. append	42
C. quicksort	45
D. queens	46
<b>V. Conclusion</b>	<b>48</b>
<b>VI. Bibliographie</b>	<b>50</b>
<b>VII. Annexes</b>	<b>1</b>
A. Trace de Append	1

1. x3 est ground (étendue)	1
2. x1 ground	6
3. x2 est ground	7
4. x1 et x2 sont ground	8
5. x1,x2 et x3 sont ground	9
6. rien n'est ground	10
B. Trace de Quicksort	11
C. Trace de Queens	17
D. Code	31
1. Programme principal	31
a. fichier iab.h	31
b. fichier iab.c	33
2. Algorithmes d'interprétation abstraite	36
a. fichier primitives.h	36
b. fichier primitives.c	37
3. Opérations sur les Tuples	45
a. fichier tuplisation.h	45
b. fichier tuplisation.c	46
4. Opérations sur les Betas	49
a. fichier formulation.h	49
b. fichier formulation.c	50
5. Opérations de parsing	57
a. fichier parser.h	57
b. fichier parser.c	57
6. Listes linéaires	59
a. fichier liste linéaire.h	59
b. fichier liste linéaire.c	60

# I. Introduction

---

## A. De l'interprétation abstraite...

L'*interprétation abstraite* permet d'étudier systématiquement certaines propriétés de l'exécution d'un programme : cette étude est *statique* en cela qu'on n'exécute pas le programme lui-même, mais qu'on fait subir à son code (le texte Prolog ici) un traitement ayant pour but non pas de produire un programme exécutable sur une machine déterminée, mais bien de fournir *avant exécution* certains renseignements sur le comportement du programme si il était effectivement exécuté.

La signification d'un programme est représentée par sa *sémantique* et c'est à partir de cette *sémantique concrète* que peuvent être établies les propriétés étudiées. Cependant, l'interprétation abstraite ne va pas travailler directement sur cette *sémantique concrète*, mais bien sur une *sémantique abstraite* appropriée.

L'interprétation se fera par *approximations* des propriétés des objets faisant partie du domaine de la *sémantique abstraite*; les propriétés retenues seront vérifiées pour la *sémantique concrète* et donc le programme lui-même. L'algorithme générique d'interprétation abstraite sur lequel est basé ce travail opère donc sur un domaine abstrait approximant la *sémantique concrète* du programme étudié.

L'interprétation abstraite<sup>1</sup> est l'association d'un domaine abstrait à un domaine concret.

Dans le cas particulier de la programmation logique, il va s'agir d'établir une *sémantique concrète* d'un programme logique normalisé, c'est-à-dire un programme logique dont on a simplifié la syntaxe. Ensuite, il faudra *abstraire* cette *sémantique* pour obtenir une *sémantique abstraite* appropriée.

## B. De l'utilité d'un exemple sur la compréhension...

Ainsi, prenons comme domaine concret l'ensemble infini des nombres entiers<sup>2</sup>. On peut imaginer de lui substituer un ensemble de trois éléments : les entiers négatifs, l'entier nul et les entiers positifs. Ce sera un domaine abstrait possible pour le domaine concret choisi.

domaine concret :  $\{-\infty\dots, -2, -1, 0, 1, 2, \dots\infty\} \Rightarrow \{\text{Neg}, \text{Nul}, \text{Pos}\}$  : domaine abstrait

---

<sup>1</sup> [Musumbu : I.23]

<sup>2</sup> [Le Charlier 3 : 1..2]

Il est à présent possible d'associer aux opérateurs existant sur le domaine concret une contrepartie portant sur le domaine abstrait, qui soit une bonne approximation des premiers. Si l'on prend par exemple la multiplication entre entier, on peut lui associer un opérateur  $\circ$  portant sur le domaine abstrait { Neg, Nul, Pos }.

Cette opération abstraite serait caractérisée par les relations suivantes :

Pos $\circ$ Pos = Pos	Neg $\circ$ Neg = Pos
Pos $\circ$ Neg = Neg	Neg $\circ$ Pos = Neg
Pos $\circ$ Nul = Nul	Neg $\circ$ Nul = Nul
Nul $\circ$ Pos = Nul	Nul $\circ$ Neg = Nul
Nul $\circ$ Nul = Nul	

Cette version abstraite de la multiplication n'est bien sûr qu'une simplification, qu'une approximation de l'opération de départ. Elle peut ne posséder en effet qu'une partie des propriétés de la multiplication : ceci a l'avantage de cerner un nombre restreint de caractéristiques de l'opérateur, tout en ignorant celles qui ne sont pas pertinentes dans une problématique donnée.

Ainsi notre sémantique abstraite ne permet-elle pas d'exprimer que l'entier 1 est neutre pour la multiplication ( $i \times 1 = 1 \times i = i$ ).

Le passage entre domaine concret et domaine abstrait est assuré par une *fonction de concrétisation*. Celle-ci est caractérisée comme suit :

$$\begin{aligned} Cc(\text{Pos}) &= \{ X \mid X \text{ est un entier et } X > 0 \} \\ Cc(\text{Neg}) &= \{ X \mid X \text{ est un entier et } X < 0 \} \\ Cc(\text{Nul}) &= \{ X \mid X = 0 \} \end{aligned}$$

Si l'on s'intéresse au signe du résultat de la multiplication de deux entiers, le résultat de l'opérateur  $\circ$  appliqué aux éléments associés aux opérands de la multiplication dans le domaine abstrait, suffira à déterminer le signe de cette multiplication.

Ainsi, si l'on recherche le signe de Res pour  $\text{Res} = -1 \times 2$ , on voit que

- $-1 \in Cc(\text{Neg})$  et  $2 \in Cc(\text{Pos})$  cf la fonction de concrétisation Cc,
- $\text{Neg} \circ \text{Pos} = \text{Neg}$  cf la table ci-dessus

On peut tirer de cela que Res respectera toutes les propriétés de  $Cc(\text{Neg})$  et donc que Res sera négatif.

## C. De son application à un programme logique...

La programmation logique et Prolog en particulier sont actuellement<sup>1</sup> l'un des domaines les plus étudiés . Cependant, Prolog est un langage de très haut niveau

<sup>1</sup> Il suffit de se rappeler à ce propos l'orientation des gros programmes de recherche en Intelligence Artificielle en Europe et au Japon...



et les compilateurs actuels ne permettent souvent que des exécutions entachées de redondance et d'inefficacité.

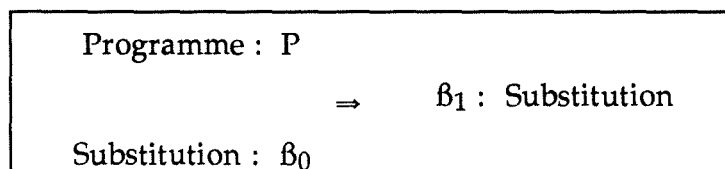
Ainsi, à des fins d'optimisation est-il important de déterminer *avant* compilation si une variable apparaissant en entête d'un programme Prolog sera *ground*<sup>1</sup> après exécution de ce programme. C'est dans cette étude du mode des variables que l'interprétation abstraite vient à notre secours.

La sémantique concrète d'un programme Prolog est habituellement décrite sous la forme de "SLD-résolution"<sup>2</sup>. Il s'agit d'un procédé par réfutation adopté par le système; il va "calculer" une série de "réponses" si on lui soumet une requête et un programme Prolog. A cette sémantique, on peut faire correspondre une sémantique de point fixe équivalente. Elle servira de sémantique *concrète*.

On peut considérer que l'exécution d'un programme Prolog peut être ramenée à une suite d'étapes successives, chacune d'elles étant caractérisée par une substitution donnée des variables qu'il renferme. Cette nouvelle sémantique est basée dans un domaine concret : les substitutions et les programmes logiques normalisés.

Après abstraction de cette sémantique, on peut disposer d'une sémantique *abstraite* adaptée spécifiquement à l'étude des modes des variables. Ses substitutions abstraites sont représentées par des formules propositionnelles. Par exemple, l'expression " $X_1 \wedge X_2 \Leftrightarrow X_3$ " va signifier que la variable  $X_1$  est *ground* et que la variable  $X_2$  sera *ground* si et seulement si la variable  $X_3$  l'est aussi.

Directement inspiré de cette sémantique abstraite, il est possible d'établir un algorithme d'interprétation abstraite. Pour une substitution abstraite de départ et pour un programme Prolog, l'algorithme va fournir une substitution devant approcher l'état des variables après exécution du programme. En d'autres termes, quel sera le mode des variables après exécution ?



L'objet de ce travail est de mettre en oeuvre la réalisation effective de l'algorithme d'interprétation abstraite décrit dans [Le Charlier 2], de le munir d'un domaine abstrait —inspiré de [Marriot]— adapté à l'étude du mode des variables d'un programme Prolog, et, enfin, de tester l'interpréteur ainsi obtenu sur quelques exemples simples.

## D. Plan

La première partie de ce travail va présenter théoriquement l'interprétation abstraite. Après un bref rappel sur le concept de sémantique, on va définir une sémantique concrète pour un programme logique normalisé. Cette sémantique va

<sup>1</sup> "lié à un terme de base", il n'y a pas de traduction élégante de "ground" aussi sera-t-il maintenu dans le texte français.

<sup>2</sup> cf infra

ensuite être abstraite pour fournir une sémantique abstraite appropriée à un domaine d'inférence déterminé : l'inférence des modes construite sur une représentation des tuples abstraits par des formules de la logique propositionnelle. Puis il sera possible de présenter l'algorithme générique d'interprétation abstraite.

Dans une deuxième partie, il conviendra de décrire la structuration des données représentant les divers éléments du domaine et les étapes de programmation implémentant indépendamment l'algorithme générique et les opérations primitives propres au domaine choisi. L'architecture en niveaux sera mise en évidence.

Finalement, on pourra décrire l'utilisation du programme réalisé sur quelques exemples de programmes Prolog.

## II. Présentation théorique

---

### A. Une sémantique...

La sémantique<sup>1</sup> d'un langage va donner une *signification* précise à tout programme écrit dans ce langage. Ainsi, dans le cas de la programmation logique, il est possible de distinguer trois types de sémantiques<sup>2</sup> : une sémantique opérationnelle, une sémantique déclarative et une sémantique dénotationnelle.

La sémantique *opérationnelle* fonde la signification d'un programme sur l'ensemble des réponses *calculées* que l'on peut obtenir de l'exécution de ce programme par un interpréteur pour une requête donnée : il s'agit donc d'une description *procédurale* dans le sens où c'est un algorithme qui détermine la sémantique.

La sémantique *déclarative* se base, elle, sur l'interprétation de Herbrand en logique du premier ordre : il s'agit de donner une description déclarative d'un programme pour une requête grâce à l'ensemble des réponses *correctes* vis-à-vis du modèle minimal de Herbrand pour ce programme.

La sémantique *dénotationnelle* est une sémantique de point fixe<sup>3</sup>. Elle associe à un programme une fonction sur le domaine calculé par ce programme. Le point fixe de cette fonction donne la signification du programme.

C'est cette dernière qui sera utilisée dans ce travail pour établir la sémantique concrète, mais avant ça, il faut décrire les objets dont on va donner la signification : les programmes et les substitutions logiques normalisés.

### B. Sémantique concrète

#### 1. Domaine Concret

##### a. Programme logique

La programmation logique est basée sur la théorie de la logique des prédicats du premier ordre. Le langage le plus courant à mettre en pratique cette théorie est Prolog. La sémantique de Prolog est plus restrictive que celle de la programmation logique en général. De plus, leur syntaxe n'est pas tout à fait identique<sup>4</sup>.

---

1 [Lloyd : 10]

2 [Sterling : 80]

3 [Sterling : 82], [Le Charlier 2 : 9]

4 [Lloyd : 94]

Les résultats qui vont suivre, bien que valables dans le cas le plus général, ont pour finalité l'interprétation de programmes Prolog. Par la suite, il sera fait référence indifféremment à l'un ou à l'autre concept : il faut cependant toujours avoir à l'esprit que la programmation logique ne se réduit pas à Prolog.

Quant à un programme normalisé, il s'agit d'une restriction de la syntaxe<sup>1</sup> habituelle des programmes logiques. Il est toujours possible de passer d'un programme logique à son équivalent normalisé. Dans le cadre de ce travail, les programmes retenus devront toujours être normalisés, ce qui facilite grandement leur traitement.

### Description

- Soit  $F_i$  est l'ensemble des foncteurs d'arité  $i$ . ( $i \geq 0$ )  
Soit  $P_i$  est l'ensemble des symboles prédicatifs d'arité  $i$ . ( $i \geq 0$ )  
Soit  $VP$  est l'ensemble infini des variables de programme avec  $x_1, \dots, x_n \in VP$ .  
Soit  $VR$  est l'ensemble infini des variables de renommage avec  $y, z, \dots \in VR$ .
- Un *programme logique normalisé* est un ensemble fini de procédures normalisées.
- Une *procédure normalisée* est une suite non vide de clauses normalisées de même foncteur.
- Une *clause normalisée* a la forme :  
$$p(x_1, \dots, x_n) \leftarrow \text{SAN}$$
où  $n \geq 0$ ,  $p \in P_n$  et SAN est une Suite d'Atomes Normalisés.  
Si il existe  $m$  variables distinctes dans la clause normalisée, alors ces variables doivent être des variables de programme ( $x_1, \dots, x_m \in VP$ ).
- Un *atome normalisé* est soit un built-in, soit un appel de procédure.
- Un *built-in normalisé* a la forme :  
$$x_i = x_j \quad (i \neq j)$$
ou  
$$x_i = f(x_{j1}, \dots, x_{jn})$$
où  $x_i, x_{j1}, \dots, x_{jn}$  sont des variables distinctes et  $f \in F_n$ .
- Un *appel de procédure normalisé* a la forme :  
$$p(x_{i1}, \dots, x_{in})$$
où  $x_{i1}, \dots, x_{in}$  sont des variables distinctes et  $p \in P_n$ .
- Un *terme normalisé* est soit une variable, soit une construction de la forme :  
$$f(t_1, \dots, t_n)$$

<sup>1</sup>[Musumbu : I.3], [Le Charlier 2 : 2]

où  $f \in F_n$  et  $t_1, \dots, t_n$  sont des termes normalisés.

- Un *foncteur* est caractérisé par son nom et son arité, c'est-à-dire le nombre de ses arguments.
- Une *constante* est un foncteur d'arité nulle.

### Extension

Il peut être utile d'étendre la syntaxe précédente grâce aux conventions de représentation et de manipulation de listes, telles qu'elles apparaissent dans le langage Prolog<sup>1</sup>. Le foncteur de construction de liste "." est le seul foncteur possible dans cette version du langage.

- Une *liste* est
  - soit la *liste vide*, dénotée []
  - soit une structure à deux composants : sa tête et sa queue

On indique la fin d'une liste en lui donnant pour queue la liste vide. Une liste non vide est représentée par le foncteur "." dont

- le premier argument est la tête de la liste,
- le second argument en est la queue.

Ainsi, ".(T,[])" correspond à la liste du seul élément T; elle peut être alternativement représentée par "[T]".

De même, "[X<sub>1</sub>,X<sub>2</sub>,X<sub>3</sub>]" correspond à ".(X<sub>1</sub>.(X<sub>2</sub>.(X<sub>3</sub>,[])))", c.-à-d. une liste de trois éléments.

- On introduit par ailleurs la notation "[T|Q]" qui permet de considérer d'un côté la tête de la liste —soit T— et de l'autre la queue restante —soit Q—. Ainsi, le but "[1,2,3,4]=[T|Q]" amène l'unification<sup>2</sup> de la variable T avec la tête de la liste —soit [1]—, et de la variable Q avec la queue de la liste —soit [2,3,4]—.

### Éléments de syntaxe

Il s'agit de préciser<sup>3</sup> quelques-unes des définitions précédentes :

Zéro = "0".  
ChiffreNonNul = "1" | ... | "9".  
Chiffre = Zéro | ChiffreNonNul.  
Entier = Zéro | (ChiffreNonNul { Chiffre } ).  
LettreMin = "a" | ... | "z".  
AlphaNum = LettreMin | Chiffre.  
SymbVariable = "X" { Entier }.  
ListeVide = "[" " ]".  
ListeUnitaire = "[" SymbVariable " ]".  
ListeMultiple = "[" SymbVariable "|" SymbVariable { "," SymbVariable } " ]".  
Liste = ListeVide | ListeUnitaire | ListeMultiple.

<sup>1</sup> [Clocksin : 81]

<sup>2</sup> Cf Infra

<sup>3</sup> On utilise ici la syntaxe grammaticale EBNF. Pour rappel,

[X] : 0 ou 1 instance de X

{X} : 0 ou n instances de X

(X|Y) : un groupement : X ou Y

Constante = Entier | ListeVide.  
 SymbPred = LettreMin { AlphaNum | "\_" }.  
 ListeVar = "(" SymbVariable { "," SymbVariable }").  
 UnifVar = SymbVariable "=" SymbVariable.  
 UnifFonct = SymbVariable "=" Liste.  
 BuiltIn = UnifVar | UnifFonct.  
 AppelProc = SymbPred ListeVar.  
 AtomeNorm = BuiltIn | AppelProc.  
 ClauseNorm = AppelProc ("←" | ":-" ) "(" { AtomeNorm } ")" ".".  
 ProcNorm = ClauseNorm { ClauseNorm }.  
 PgmNorm = ProcNorm { ProcNorm }.

### Exemple de programme normalisé

Voici un programme normalisé composé d'une procédure normalisée, elle-même divisée en deux clauses normalisées :

append(x<sub>1</sub>,x<sub>2</sub>,x<sub>3</sub>) ← x<sub>1</sub>=[], x<sub>2</sub>=x<sub>3</sub>.

et

append(x<sub>1</sub>,x<sub>2</sub>,x<sub>3</sub>) ← x<sub>3</sub>=[x<sub>4</sub> | x<sub>6</sub>], append(x<sub>5</sub>,x<sub>2</sub>,x<sub>6</sub>), x<sub>1</sub>=[x<sub>4</sub> | x<sub>5</sub>].

F<sub>3</sub>={[]}

P<sub>3</sub>={"append"}

VP={x<sub>1</sub>,x<sub>2</sub>,x<sub>3</sub>,x<sub>4</sub>,x<sub>5</sub>,x<sub>6</sub>}

VR={}

La première clause normalisée est formée de

- une entête : append(x<sub>1</sub>,x<sub>2</sub>,x<sub>3</sub>) où "append" est un symbole prédicatif et x<sub>1</sub>,x<sub>2</sub>,x<sub>3</sub> sont des variables de programme.
- un corps : formé lui-même de deux atomes normalisés
  - x<sub>1</sub>=[] : est un built-in normalisé où x<sub>1</sub> est une variable de programme et [] représente une constante c.-à-d. un foncteur 0-aire.
  - x<sub>2</sub>=x<sub>3</sub> : est un built-in normalisé où x<sub>2</sub> et x<sub>3</sub> sont des variables de programme.

La deuxième clause est formée de

- une entête : ... la même que dans la première clause
- un corps : formé lui-même de trois atomes normalisés
  - x<sub>3</sub>=[x<sub>4</sub> | x<sub>6</sub>] : est un built-in normalisé où [x<sub>4</sub> | x<sub>6</sub>] est équivalent à ". (x<sub>4</sub>,x<sub>6</sub>)" avec "." un foncteur et {x<sub>3</sub>,x<sub>4</sub>,x<sub>6</sub>} des variables de programme.
  - append(x<sub>5</sub>,x<sub>2</sub>,x<sub>6</sub>) : est un appel de procédure normalisé où "append" est un symbole prédicatif, {x<sub>2</sub>,x<sub>5</sub>,x<sub>6</sub>} sont des variables de programme.
  - x<sub>1</sub>=[x<sub>4</sub> | x<sub>5</sub>] : est un built-in normalisé où [x<sub>4</sub> | x<sub>5</sub>] est équivalent à ". (x<sub>4</sub>,x<sub>5</sub>)" avec "." un foncteur et {x<sub>1</sub>,x<sub>4</sub>,x<sub>5</sub>} des variables de programme.

## b. Substitutions normalisées

### Description

- Une *liaison normalisée* a la forme :

$$v \leftarrow t$$

où  $v$  est une variable et  $t$  est un terme normalisé.

- Une *substitution normalisée* est un ensemble de liaisons normalisées de la forme :

$$\theta = \{v_1 \leftarrow t_1, \dots, v_n \leftarrow t_n\}$$

telle que  $v_1, \dots, v_n$  sont des variables distinctes et  $t_1, \dots, t_n$  des termes normalisés avec  $v_i \neq t_j \forall i$ .

- Le *domaine* d'une substitution  $\theta$  — $\text{dom}(\theta)$ — correspond à l'ensemble  $\{v_1, \dots, v_n\}$ .

Son *codomaine* — $\text{codom}(\theta)$ — correspond à l'ensemble des variables qui apparaissent dans  $\{t_1, \dots, t_n\}$ .

- Une substitution  $\theta$  est dite "de variable de programme" si  $\text{dom}(\theta) \subseteq VP$  et  $\text{codom}(\theta) \subseteq VR$ .

Une substitution  $\theta$  est dite "de variable de renommage" si  $\text{dom}(\theta) \subseteq VR$  et  $\text{codom}(\theta) \subseteq VR$ .

- Soit CS une Construction Syntaxique quelconque et  $\theta$  une substitution,  $\text{var}(CS)$  représente l'ensemble des variables qui figurent dans CS. On a donc  $\text{var}(\theta) = \text{dom}(\theta) \cup \text{codom}(\theta)$ .

- *Instancier* CS par  $\theta$  consiste à remplacer dans CS toute occurrence d'une variable de  $\text{dom}(\theta)$  par le terme qui lui est lié dans  $\theta$ .

- Soit  $\text{dom}(\theta) \supseteq D$ , on appelle *restriction*  $\theta|_D$  de  $\theta$  à  $D$  la substitution telle que  $\text{dom}(\theta|_D) = D$  et  $x\theta = x\theta|_D \forall x \in D$ .

### Exemple de substitution

Reprenons la première clause du programme normalisé décrit précédemment :

$$\text{append}(x_1, x_2, x_3) \leftarrow x_1 = [], x_2 = x_3.$$

Envisageons la substitution  $\theta = \{x_1 \leftarrow y, x_2 \leftarrow [1, 2, 3]\}$ .

Il s'agit bien d'une substitution normalisée formée de deux liaisons normalisées :

- $x_1 \leftarrow y$  : où le terme  $y$  correspond à une variable de renommage n'appartenant pas à  $\{x_1, x_2, x_3\}$
- $x_2 \leftarrow [1, 2, 3]$  : où le terme  $[1, 2, 3]$  correspond à une liste d'entiers, c.-à-d. une constante

$\theta$  est une substitution de variable de programme puisque  $\text{dom}(\theta) = \{x_1, x_2\} \subseteq VP$  et  $\text{codom}(\theta) = \{y\} \subseteq VR$ ; de plus,  $\text{var}(\theta) = \{x_1, x_2, y\}$ .

L'instanciation du but "append( $x_1, x_2, x_3$ )" par la substitution  $\theta$  donne "append( $y, [1, 2, 3], x_3$ )".

Si l'on considère un sous-ensemble  $D = \{x_1\}$  pour lequel on vérifie que  $D \subseteq \text{dom}(\theta)$ , la restriction de  $\theta$  à  $D$ , correspond à la nouvelle substitution  $\theta|_D = \{x_1 \leftarrow y\}$ .

## 2. Opérations concrètes

Afin de définir une sémantique de point fixe pour les programmes logiques, on a besoin d'un jeu d'opérations sur des ensembles de substitutions. On peut trouver la spécification rigoureuse de ces opérations dans [Le Charlier 2 : 6...8] ou [Musumbu : I/15...I/17]. La description qui suit ne reprendra que l'essentiel.

Soit  $c$  une clause normalisée, soit  $g$  un but normalisé, soit  $f$  un foncteur, soit  $\Theta, \Theta'$  des ensembles de substitutions.

### a. Opérations de Restriction

Il s'agit de restreindre un ensemble de substitutions par rapport à un sous-ensemble des variables figurant dans l'ensemble initial des substitutions.

#### RESTRC( $c, \Theta$ )

Cette opération va restreindre aux seules variables de la tête de la clause  $c$  un ensemble de substitutions portant sur les variables de l'ensemble de la clause.

Soit  $x_1, \dots, x_m$  les variables représentées dans  $\Theta$ , qui sont les variables de la clause  $c$  (tête+corps), soit  $x_1, \dots, x_n$  ( $m \geq n$ ) les variables de la tête de la clause  $c$ , le résultat de cette opération est obtenu en projetant  $\Theta$  sur les variables  $x_1, \dots, x_n$ .

#### RESTRG( $g, \Theta$ )

Cette opération va restreindre aux seules variables du but  $g$  un ensemble de substitutions  $\Theta$ . Les variables seront renommées à partir de  $X_1$  et une trace<sup>1</sup> de ce renommage devra être gardée afin d'en assurer la réversibilité.

Soit  $x_1, \dots, x_m$  les variables représentées dans  $\Theta$ , soit le but  $g$  de la forme

- $p(x_{i1}, \dots, x_{in})$  ou
- $x_{i1} = x_{i2}$  ou
- $x_{i1} = f(x_{i2}, \dots, x_{in})$ .

Le résultat de cette opération est obtenu en deux étapes : il faut

- 1° projeter  $\Theta$  sur les variables  $\{x_{i1}, \dots, x_{in}\}$ , ce qui donne  $\Theta'$
- 2° renommer  $\Theta'$  en terme de  $\{x_1, \dots, x_n\}$  c'est-à-dire  $x_{ik}$  devient  $x_k$ .

### b. Opérations d'Extension

Il s'agit d'étendre un ensemble de substitutions à de nouvelles variables.

#### EXTC( $c, \Theta$ )

Cette opération va étendre à l'ensemble des variables de la clause  $c$  une substitution limitée aux seules variables de la tête de la clause.

---

<sup>1</sup> On peut penser à une fonction de concordance  $\gamma$  : il s'agit d'un "mapping" d'un ensemble de variables sur un autre.



Soit  $x_1, \dots, x_m$  les variables représentées dans  $\Theta$  et  $x_1, \dots, x_n$  ( $m \leq n$ ) les variables de la clause  $c$  (tête+corps), le résultat de cette opération est obtenu en étendant  $\Theta$  pour tenir compte des variables libres du corps de  $c$ , soit  $x_{m+1}, \dots, x_n$ .

#### EXTG(g, $\Theta$ , $\Theta'$ )

Cette opération va étendre aux variables de  $\Theta$  un ensemble de substitutions  $\Theta'$  limité aux seules variables du but  $g$ . Lors de l'extension, on renverse un renommage précédent dont la trace demeure au travers de  $\gamma^1$ .

Soit  $x_1, \dots, x_m$  les variables représentées dans  $\Theta$ , soit le but  $g$  de la forme

- $p(x_{i1}, \dots, x_{in})$  ou
- $x_{i1} = x_{i2}$  ou
- $x_{i1} = f(x_{i2}, \dots, x_{in})$ .

Le résultat de cette opération est produit en étendant  $\Theta$  de manière à ce qu'il tienne compte de  $\Theta'$ , lui-même obtenu pour le but  $g$  et précédemment restreint aux seules variables de  $g$  —cf RESTRG(g,  $\Theta$ )—.

#### *c. Opération d'Union*

Il s'agit d'obtenir l'union d'ensembles de substitutions; si chacune des clauses d'une procédure donne pour résultat un ensemble de substitutions, leur union participe au résultat de la procédure. L'opération correspondante est UNION( $\Theta, \Theta'$ ).

#### *d. Opérations d'Unification*

Les opérations d'unification sont au coeur du processus d'interprétation. Le programme à interpréter est composé au moins en partie par des built-in normalisés : c'est l'exécution de ces derniers qui va éventuellement modifier les substitutions sur les variables du programme.

#### AIVAR( $\Theta$ )

Au départ du built-in normalisé de la forme " $x_1 = x_2$ ", cette opération produit un résultat obtenu par l'unification des deux variables du built-in sur l'ensemble des variables des substitutions de  $\Theta$ , qui est lui défini sur ces deux variables.

Si par exemple  $\Theta$  exprime qu'une certaine propriété est vérifiée pour l'une des deux variables, on peut s'attendre à ce que, après unification, elle soit vérifiée pour les deux. Mais on anticipe déjà ici sur le sens que l'on va pouvoir donner aux substitutions dans le domaine abstrait.

#### AIFUNC( $\Theta, f$ )

Au départ du built-in normalisé de la forme " $x_1 = f(x_2, \dots, x_n)$ ", cette opération produit un résultat obtenu par l'unification de  $x_1$  et de  $f(x_2, \dots, x_n)$  sur l'ensemble des variables des substitutions de  $\Theta$ .  $\Theta$  est défini sur  $\{x_1, \dots, x_n\}$  et  $f$  est un foncteur d'arité  $n-1$ .

---

<sup>1</sup> cf supra

e. Exemple

c:  $\text{append}(x_1, x_2, x_3) \leftarrow x_3 = [x_4 | x_6], \text{append}(x_5, x_2, x_6), x_1 = [x_4 | x_5].$

g:  $\text{append}(x_5, x_2, x_6).$

$\Theta_c$  est une substitution définie sur les variables de c, soit :

$$\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, x_3 \leftarrow t_3, x_4 \leftarrow t_4, x_5 \leftarrow t_5, x_6 \leftarrow t_6\}$$

$\Theta_{tc}$  est la substitution  $\Theta_c$  limitée aux variables de la tête de c, soit :

$$\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, x_3 \leftarrow t_3\}$$

$\Theta_g$  est la substitution  $\Theta_c$  limitée aux variables de g, soit :

$$\{x_5 \leftarrow t_5, x_2 \leftarrow t_2, x_6 \leftarrow t_6\}$$

$\Theta_{ec}$  est la substitution résultant de  $\text{EXTC}(c, \Theta_{tc})$ , soit :

$$\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, x_3 \leftarrow t_3, x_4 \leftarrow t'_4, x_5 \leftarrow t'_5, x_6 \leftarrow t'_6\}$$

où  $t'_4, t'_5, t'_6$  ne représentent aucune "information" susceptible de contraindre  $x_4, x_5, x_6$  par la suite.  $x_4, x_5, x_6$  sont des variables libres et on<sup>1</sup> peut décider de les représenter en les liant à des variables de renommage n'apparaissant nulle part ailleurs.

$\Theta_{rc}$  est la substitution résultant de  $\text{RESTRC}(c, \Theta_{ec})$ , soit :

$$\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, x_3 \leftarrow t_3\}$$

$\Theta_{rg}$  est la substitution résultant de  $\text{RESTRG}(g, \Theta_c)$ , soit :

$$\{x_1 \leftarrow t''_1, x_2 \leftarrow t''_2, x_3 \leftarrow t''_3\}$$

où  $t''_1, t''_2, t''_3$  sont issus du renommage de  $t_5, t_2, t_6$ ; cette restriction génère de plus une table du type suivant :

$\Theta_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$
$\Theta_1$	$\emptyset$	$x_2$	$\emptyset$	$\emptyset$	$x_1$	$x_3$

$\Theta_{eg}$  est la substitution résultant de  $\text{EXTG}(g, \Theta_c, \Theta_{rg})$ , soit :

$$\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, x_3 \leftarrow t_3, x_4 \leftarrow t_4, x_5 \leftarrow t_5, x_6 \leftarrow t_6\}$$

cette extension fait appel à la table précédente.

### 3. Ensembles de tuples concrets

La sémantique concrète d'un programme doit établir quelle sera l'ensemble de substitutions qui caractérise son état *après* exécution, en fonction d'un ensemble de substitutions *avant* exécution.

Si l'on considère une procédure p et deux ensembles de substitutions  $\Theta$  et  $\Theta'$ , on définit le triplet<sup>2</sup>  $(\Theta, p, \Theta')$  où  $\Theta$  est l'ensemble de substitutions avant exécution de la procédure et  $\Theta'$  l'ensemble de substitutions après exécution. Il s'agit là d'un *tuple concret*.

<sup>1</sup> [Le Charlier 2:4]

<sup>2</sup> [Le Charlier 2 : 8]

#### 4. Sémantique de point fixe

Tout est à présent réuni pour définir la sémantique d'un programme logique. Mais il est auparavant nécessaire de revenir sur les notions d'interprétation, de transformation et de point fixe.

##### a. Interprétation de Herbrand

###### Quelques définitions...

- L'univers de Herbrand d'un programme logique<sup>1</sup> est l'ensemble des termes sans variable construits grâce aux constantes et aux foncteurs présents dans le programme.
- La base de Herbrand est l'ensemble des atomes sans variable construits grâce aux prédicats du programme et aux termes de l'univers de Herbrand.
- Une interprétation de Herbrand est un sous-ensemble de sa base de Herbrand. Un but de la base de Herbrand est *vrai* selon une interprétation s'il appartient à celle-ci, il est *faux* sinon.

###### Exemple d'interprétation

Reprenons une fois encore le programme P présenté précédemment :

$\text{append}(x_1, x_2, x_3) \leftarrow x_1 = [], x_2 = x_3.$

et

$\text{append}(x_1, x_2, x_3) \leftarrow x_3 = [x_4 | x_6], \text{append}(x_5, x_2, x_6), x_1 = [x_4 | x_5].$

- L'univers de Herbrand —UH— pour P se réduit à un ensemble de termes construits sur base de la constante []; il n'y a en effet pas d'autre foncteur dans P. C'est donc l'ensemble des listes renfermant uniquement [] :

$UH = \{[], [ [] ], [ [ [] ] ], \dots\}$

- Le programme P renferme le seul prédicat "append". La base de Herbrand —BH— pour P est l'ensemble de tous les buts combinant ce prédicat avec les listes de UH :

$BH = \{\text{append}([], [], []), \text{append}([], [ [] ], [ [] ]), \dots\}$

###### Point fixe

On peut envisager une transformation T d'une interprétation I de P en une autre. Il est possible de voir que cette transformation est à la fois *monotone* et *continue*. Dans ce cas, elle possède au moins un plus petit point fixe, soit

$$T(I) = I$$

Ce plus petit point fixe donne sa signification au programme P dans le cadre de la sémantique dénotationnelle. Cette sémantique est à la base de la construction de l'algorithme d'interprétation abstraite.

##### b. Sémantique de point fixe

Ainsi, va-t-on définir une sémantique de point fixe basée sur une transformation et trois fonctions; celles-ci sont définies au moyen de l'ensemble des opérations concrètes présentées ci-dessus.

---

<sup>1</sup> [Sterling : 80]

Les trois fonctions portent sur des ensembles de tuples à trois niveaux :

- celui du programme P :  $T_p$ ,
- celui d'une clause dont le prédicat est tiré de P :  $T_c$ ,
- celui d'une suite d'atomes :  $T_b$ .

On peut présenter succinctement celles-ci. Pour une description plus détaillée, on se reportera à [Le Charlier 2].

- soit etc un ensemble de tuples concrets,
- soit  $\Theta, \Theta'$  des ensembles de substitutions,
- soit D l'ensemble des paires  $(\Theta, p)$  formées des deux premières composantes —le domaine— des tuples  $(\Theta, p, \Theta')$  où les variables de  $\Theta$  sont celles de p.
- soit  $b_1.b_0$  la "composition" de deux buts, c'est-à-dire l'équivalent fonctionnel de la séquence d'exécution de ceux-ci.

$$TETC(etc) = \{ (\Theta, p, \Theta') : (\Theta, p) \in D \text{ et } \Theta' = T_p(\Theta, p, etc) \}$$

$$T_p(\Theta, p, etc) = \text{UNION}(\Theta_1, \dots, \Theta_n)$$

$$\text{où } \Theta_i = T_c(\Theta, c_i, etc) \text{ et } c_1, \dots, c_n \text{ sont les clauses de p.}$$

$$T_c(\Theta, c, etc) = \text{RESTRC}(c, \Theta')$$

$$\text{où } \Theta' = T_b(\text{EXTC}(c, \Theta), b, etc) \text{ et } b \text{ est le corps de la clause } c.$$

$$T_b(\Theta, \langle \rangle, etc) = \Theta$$

$$T_b(\Theta, b_1 b_0, etc) = T_b(\Theta_3, b_0, etc)$$

$$\text{où } \Theta_3 = \text{EXTG}(g, \Theta, \Theta_2),$$

$$\Theta_2 = \text{etc}(\Theta_1, p) \text{ si } g \text{ est un appel de procédure ou}$$

$$\Theta_2 = \text{AIVAR}(\Theta_1) \text{ si } g \text{ est l'unification de deux variables ou}$$

$$\Theta_2 = \text{AIFUNC}(\Theta_1, f) \text{ si } g \text{ est de la forme "x}_i = f(\dots)",$$

$$\Theta_1 = \text{RESTRG}(g, \Theta)$$

On<sup>1</sup> peut prouver que cette sémantique est cohérente par rapport à la sémantique opérationnelle habituellement proposée.

## C. Sémantique abstraite

### 1. Domaine Abstrait

Le domaine abstrait a pour finalité de permettre l'étude du mode des variables d'un programme logique normalisé. La sémantique<sup>2</sup> a permis de déterminer les tuples  $\{\Theta_{in}, p, \Theta_{out}\}$  où, pour un programme P donné et pour un ensemble de substitutions d'entrée  $\Theta_{in}$ , on obtient pour un appel de procédure p un ensemble de substitutions  $\Theta_{out}$  après exécution de p avec  $\Theta_{in}$ .

<sup>1</sup> [Le Charlier 2 : 10]

<sup>2</sup> [Le Charlier 3 : 5]

Il reste à déterminer la forme d'une substitution abstraite capable de représenter au mieux le mode des variables, mais aussi les relations que peuvent avoir ces relations entre elles et l'impact que cela peut avoir sur le calcul des modes.

*a. Substitutions abstraites*

Une substitution abstraite va être représentée par une formule logique disjonctive. Ainsi, si l'on veut représenter le fait qu'un terme est ground, il apparaîtra positivement dans la formule logique. Une formule logique sera formée de conjonctions d'éléments associés chacun à une et une seule variable : la substitution correspondra à la disjonction de ces conjonctions.

La formulation normalisée disjonctive a été choisie pour les facilités qu'elle offre dans la simplification des expressions. Une formule logique ne possède en effet qu'une seule représentation, ce qui facilite grandement le test d'équivalence entre deux formules.

Par exemple ce qui est habituellement exprimé sous forme de la substitution suivante va donner la formule logique formée de la conjonction correspondante :

$$\{X_1 \leftarrow \text{noground}, X_2 \leftarrow \text{noground}, X_3 \leftarrow \text{ground}\} \equiv \{\neg X_1 \wedge \neg X_2 \wedge X_3\}$$

Si le mode d'une variable est indéterminé —any—, on va recourir à la disjonction de deux conjonctions exprimant les deux alternatives pour la variable indéterminée :

$$\{X_1 \leftarrow \text{noground}, X_2 \leftarrow \text{any}, X_3 \leftarrow \text{ground}\} \equiv \{\neg X_1 \wedge X_2 \wedge X_3\} \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3\}$$

Par ailleurs, il est possible d'exprimer par une équivalence le fait que deux termes sont *liés*, c'est-à-dire que si l'un devient ground, l'autre le deviendra aussi —c'est le résultat d'une unification entre variables par exemple—. En outre, l'équivalence logique est transformable en une disjonction de deux conjonctions:

$$X_1 \Leftrightarrow X_2 \equiv \{X_1 \wedge X_2\} \vee \{\neg X_1 \wedge \neg X_2\}$$

De même, un terme peut être lié à un ensemble d'autres —c'est le résultat d'une unification entre une variable et un foncteur par exemple—. On obtient cette fois :

$$X_1 \Leftrightarrow \{X_2 \wedge X_3\} \equiv$$

$$\{X_1 \wedge X_2 \wedge X_3\} \vee \{\neg X_1 \wedge \neg X_2 \wedge \neg X_3\} \vee \{\neg X_1 \wedge X_2 \wedge \neg X_3\} \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3\}$$

Il est à noter qu'on assiste très vite à une croissance non linéaire du nombre de disjonctions et donc de la taille de la formule lorsque le nombre de variables croît un petit peu. On peut donc en conséquence anticiper des problèmes d'emplacement mémoire lors du stockage de ces formules avec un nombre non trivial de variables.

On peut présenter la forme normale disjonctive d'une manière un peu plus formelle<sup>1</sup> :

- Var = ["-"] Symbvariable.
- Conjonction = Var | Conjonction "^" Conjonction.
- Disjonction = Conjonction | Disjonction "v" Disjonction.
- Formule = Disjonction.

<sup>1</sup> On poursuit la présentation sous forme EBNF

## 2. Relation d'ordre

On peut se demander s'il est possible d'établir une relation d'ordre partiel entre substitutions abstraites. Soit  $\beta$  et  $\beta'$  deux substitutions, on peut dire que

- $\beta = \beta'$  si les deux formules possèdent les mêmes conjonctions.
- $\beta$  et  $\beta'$  sont *compatibles* si, considérant les deux formules comme des ensembles de conjonctions,  $\beta \supseteq \beta'$  ou  $\beta' \supseteq \beta$ .
- $\beta < \beta'$  si  $\beta$  est moins général que  $\beta'$ ; en d'autres termes, que  $\beta \Rightarrow \beta'$ , c'est-à-dire que  $\beta' \supseteq \beta$ . Par exemple,

$$\{X_1 \wedge X_2 \wedge X_3\} \vee \{\neg X_1 \wedge \neg X_2 \wedge \neg X_3\} > \{X_1 \wedge X_2 \wedge X_3\}$$

## 3. Opérations abstraites

Soit  $c$  une clause normalisée, soit  $g$  un but normalisé, soit  $f$  un foncteur, soit  $\beta, \beta'$  des substitutions abstraites, c'est-à-dire des formules logiques disjonctives.

### a. Opérations de Restriction

#### RESTRC(c, $\beta$ )

On rappelle qu'il s'agit de restreindre aux seules variables de la tête de la clause  $c$  un ensemble de substitutions portant sur les variables de l'ensemble de la clause.

Si la clause  $a$  dans sa tête  $n$  variables et dans son corps  $m$  variables distinctes des  $n$  premières, le fait que le programme soit normalisé nous assure que les variables sont numérotées de 1 à  $m+n$ .

La spécification de la version concrète de RESTRC nous garantit aussi que les variables qui apparaissent dans  $\beta$  correspondent à un sous-ensemble des  $m+n$  variables de la clause  $c$  complète. De plus, celles qui sont indicées de 1 à  $m$  sont celles de la tête. Aussi, la restriction de  $\beta$  à ces  $m$  variables peut être effectuée en *éliminant* simplement les autres variables.

#### RESTRG(g, $\beta$ )

On rappelle qu'il s'agit de restreindre aux seules variables du but  $g$  un ensemble de substitutions  $\beta$ .

On va commencer par restreindre chacune des conjonctions de  $\beta$  aux variables de  $g$ , c'est-à-dire faire disparaître des formules les variables qui n'apparaissent pas dans  $g$ . Il faut ensuite renommer les variables dans ces conjonctions "allégées".

Trois cas sont à envisager, selon le type du but  $g$  auquel on a affaire : dans les trois cas, le renommage des variables va donner une suite de variables numérotées à partir de 1.

Cette restriction doit être réversible et il faut donc créer une table  $\gamma$  de concordance entre la numérotation des variables dans la substitution  $\beta$  initiale et celle dans la substitution après restriction. Cette table peut avoir la forme d'une table ordonnée de variables; si la variable  $X_i$  de la substitution initiale est remplacée par la variable  $X_j$ , l'élément  $i$  de  $\gamma$  vaut  $j$ ; si la variable  $X_i$  de la substitution initiale n'apparaît plus dans la nouvelle substitution, l'élément  $i$  de  $\gamma$  est vide.

## b. Opérations d'Extension

### EXTC(c, β)

On rappelle qu'il s'agit d'étendre à l'ensemble des variables de la clause  $c$  une substitution limitée aux seules variables de la tête de la clause.

Comme ci-dessus, puisque la clause  $a$  dans sa tête  $n$  variables et dans son corps  $m$  variables distinctes des  $n$  premières, le fait que le programme soit normalisé nous assure que les variables sont numérotées de 1 à  $m+n$ .

La spécification de la version concrète de EXTC nous garantit que les variables qui apparaissent dans les substitutions de  $\beta$  correspondent à un sous-ensemble des  $m+n$  variables de la clause  $c$  complète : celles qui sont indicées de 1 à  $m$ , c'est-à-dire celles de la tête. Aussi, l'extension de  $\beta$  aux  $m+n$  variables peut être effectuée en *ajoutant*  $n$  variables libres, numérotées de  $m+1$  à  $m+n$ .

On sait que le mode de ces variables est *var*, mais c'est inexprimable dans notre domaine abstrait, aussi devra-t-on se contenter de les faire figurer avec *any* comme mode : chaque variable devra apparaître tantôt positivement, tantôt négativement. Ceci amène une croissance non linéaire du nombre de conjonctions dans le nouvel ensemble de substitutions.

### EXTG(g, β, β')

On rappelle qu'il s'agit d'étendre aux variables de  $\beta$  un ensemble de substitutions  $\beta'$  limité aux seules variables du but  $g$ .

Ainsi, on supprime de  $\beta$  les conjonctions dont les variables ne correspondent pas, du moins quand elles sont présentes dans  $\beta'$ , à celles d'au moins l'une des conjonctions de  $\beta'$ .

Cette interprétation de l'extension permet de répercuter sur la nouvelle substitution les contraintes sur les variables absentes de  $\beta'$  préexistant à l'exécution de  $g$ .

## c. Opération d'Union

On rappelle qu'il s'agit d'obtenir l'union de substitutions. On a affaire à l'union ensembliste habituelle  $\beta \cup \beta'$  appliquée aux formules. Il faut bien veiller à ne pas laisser apparaître de doublons de conjonctions dans la nouvelle substitution.

## d. Opérations d'Unification

Distinguons deux cas :

### AIVAR(β)

On rappelle qu'au départ du built-in normalisé de la forme " $x_1 = x_2$ ", cette opération permet l'unification des deux variables sur  $\beta$ .

Cette opération peut être réalisée simplement en générant la formule :

$$\{X_1 \wedge X_2\} \vee \{\neg X_1 \wedge \neg X_2\}$$

qui, il faut s'en souvenir, correspond à  $X_1 \Leftrightarrow X_2$ .

### AIFUNC(β, f)

On rappelle qu'au départ du built-in normalisé de la forme " $x_1 = f(x_2, \dots, x_n)$ ", cette opération permet l'unification des  $n$  variables sur  $\beta$ .

Cette opération peut être réalisée simplement en générant la formule :

$$X_1 \Leftrightarrow \{X_2 \wedge \dots \wedge X_n\}$$

Comme on l'a vu ci-dessus, cette formule peut aussi être exprimée sous la forme d'une disjonction de conjonctions. Il faudra pour y arriver générer toutes les combinaisons possibles des variables du foncteur, c'est-à-dire toutes les manières de combiner les variables  $X_2$  à  $X_n$  en les prenant tantôt comme Vraies, tantôt comme Fausses.

e. Exemple

c:  $\text{append}(x_1, x_2, x_3) \leftarrow x_3 = [x_4 | x_6], \text{append}(x_5, x_2, x_6), x_1 = [x_4 | x_5].$

g:  $\text{append}(x_5, x_2, x_6).$

$\beta_{in} : \{\neg X_1 \wedge \neg X_2 \wedge X_3\}$

$\beta_{ec}$  est la substitution résultant de  $\text{EXTC}(c, \beta_{in})$  :

$$\begin{aligned} & \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4 \wedge X_5 \wedge X_6\} \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge \neg X_4 \wedge X_5 \wedge X_6\} \\ & \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4 \wedge \neg X_5 \wedge X_6\} \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge \neg X_4 \wedge \neg X_5 \wedge X_6\} \\ & \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4 \wedge X_5 \wedge \neg X_6\} \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge \neg X_4 \wedge X_5 \wedge \neg X_6\} \\ & \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4 \wedge \neg X_5 \wedge \neg X_6\} \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge \neg X_4 \wedge \neg X_5 \wedge \neg X_6\} \end{aligned}$$

$\beta_{rc}$  est la substitution résultant de  $\text{RESTRC}(c, \beta_{ec})$  avec  $\beta_{rc} = \beta_{in}$

$\beta_{rg}$  est la substitution résultant de  $\text{RESTRG}(g, \beta_{ec})$  :

$$\{X_1 \wedge \neg X_2 \wedge X_3\} \vee \{\neg X_1 \wedge \neg X_2 \wedge X_3\} \vee \{X_1 \wedge \neg X_2 \wedge \neg X_3\} \vee \{\neg X_1 \wedge \neg X_2 \wedge \neg X_3\}$$

cette restriction génère de plus une table du type suivant :

i	1	2	3	4	5	6
$\gamma_i$	$\emptyset$	2	$\emptyset$	$\emptyset$	1	3

$\beta_{eg}$  est la substitution résultant de  $\text{EXTG}(g, \beta_{ec}, \beta_{rg})$  avec  $\beta_{eg} = \beta_{ec}$

cette extension fait appel à la table précédente.

$\beta_u$  est la substitution résultant de  $\text{UNION}(\beta_{rg}, \beta_{in})$  et  $\beta_{rg} = \beta_u$

#### 4. Ensembles de tuples abstraits

Pour obtenir un tuple abstrait, on abstrait un tuple concret en utilisant des substitutions abstraites telles qu'elles ont été présentées ci-dessus : on a le triplet  $(\beta_{in}, p, \beta_{out})$ .

On peut considérer des ensembles de tels tuples : ce seront les ensembles de tuples abstraits ou *eta*. Au sein d'un *eta*, on peut se demander si il est possible de trier les tuples, existe-t-il une relation d'ordre parmi ceux-ci ?

Soit  $t$  et  $t'$  deux tuples d'un *eta*, faute de l'énoncé rigoureux de la relation qui peut les lier, on peut admettre que :

- $t$  et  $t'$  sont *comparables* si ils ont même  $p$  et si leur  $\beta_{in}$  sont comparables



- $t < t'$  si  $p_t = p_{t'}$  et  $\beta_{int} < \beta_{int'}$

Il s'agit d'un ordre partiel parmi les tuples.

L'algorithme qui va être exposé par la suite fait appel à des ensembles de tuples; il faut donc définir les opérations qui permettent de mettre ces  $\eta$  à jour : deux opérations sont envisagées<sup>1</sup>, EXTEND et ADJUST.

L'opération EXTEND( $\beta, p, \eta$ ) étend  $\eta$  en  $y$  définissant un tuple dont les deux premiers éléments correspondent à la paire ( $\beta, p$ ).

L'opération ADJUST( $\beta, p, \beta', \eta$ ) étend  $\eta$  en  $y$  incorporant l'information contenue dans  $\beta'$ . Tous les tuples de  $\eta$  dont le premier élément est  $\beta$  et dont le second est  $p$  sont "ajustés" de telle sorte qu'ils comprennent au moins les substitutions de  $\beta'$  dans leur troisième élément.

### 5. Fonction de concrétisation

Cette fonction doit nous permettre de passer du domaine abstrait qui vient d'être décrit au domaine concret initial.

Soit  $\theta$  une substitution concrète, on peut considérer l'ensemble qui renferme les substitutions comparables à  $\theta$  mais moins instanciées que  $\theta$  :

$$\Delta_\theta = \{\theta' \mid \theta' \text{ est moins instanciée que } \theta\}$$

Soit aussi la fonction assignation qui prend en argument l'instanciation d'une variable par une substitution concrète et a pour résultat 1 ou 0 —Vrai ou Faux— selon que la substitution lie cette variable à un terme de base:

$$\text{assignation} : \theta \rightarrow \{x_1, \dots, x_n\} \rightarrow \{0, 1\} \text{ où } x_i \in \text{Dom}(\theta)$$

$$\text{assign}_\theta : \{x_1, \dots, x_n\} \rightarrow \{0, 1\} \text{ où } x_i \in \text{Dom}(\theta)$$

$\text{assign}_\theta$  est une fonction qui pour une variable  $va$  donner la valeur de vérité correspondant au terme avec lequel la variable est liée dans  $\theta$ :

$$\text{assign}_\theta(x_i) = 1 \text{ si } x_i \text{ est ground après instanciation par } \theta, 0 \text{ sinon}$$

On peut considérer une substitution abstraite, c'est-à-dire une formule, comme une fonction Booléenne dont les arguments sont les valeurs de vérité produites par les variables de la formule pour une table de vérité donnée.

$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

Une table de vérité  $r$  satisfait  $f$  si  $f(r) = 1$ .

Pour qu'une substitution abstraite représente valablement une substitution concrète vis-à-vis de la propriété de "groundness", il faut que l'application de la fonction assignation à  $\theta$  et aux substitutions de  $\Delta_\theta$  satisfasse la substitution abstraite  $f$ .

Soit une substitution abstraite  $\beta$ , on peut établir<sup>2</sup> la fonction de concrétisation suivante :

$$Cc(\beta) = \{\theta \in \Theta : \forall \theta' \in \Delta_\theta : \beta(\text{assignation}(\theta')) = 1\}$$

<sup>1</sup> [Le Charlier 1: 8], [Le Charlier 2:17], [Musumbu: 1.59] pour une spécification plus complète

<sup>2</sup> [Musumbu : 1.14], [Le Charlier 2 : 11], [Cortesi : 2...3]

Prenons un exemple. Soit la substitution abstraite suivante :

$$\beta = X_1 \wedge (X_2 \Leftrightarrow X_3)$$

et une substitution concrète dont on peut espérer que  $\beta$  soit une abstraction :

$$\theta = \{X_1 \leftarrow [], X_2 \leftarrow f(X_4), X_3 \leftarrow f(X_4)\}$$

$$\text{et assignation } (\theta) \{X_1, X_2, X_3\} = (1, 0, 0)$$

On peut voir que  $\Delta_\theta = \{ \{X_1 \leftarrow [], X_2 \leftarrow f(t), X_3 \leftarrow f(t)\} \}$ , où  $t$  est un terme quelconque.

Finalement,  $\forall \theta' \in \Delta_\theta : (\text{assignation } (\theta')) = \{ (1, 0, 0), (1, 1, 1) \}$  et donc  $\theta \in \text{Cc}(\beta)$  car  $\beta(1, 0, 0) = 1$  et  $\beta(1, 1, 1) = 1$ .

## 6. Sémantique abstraite

La sémantique abstraite correspond à la sémantique concrète où le domaine et les opérations ont été remplacés par le domaine abstrait et les opérations abstraites décrites ci-dessus.

## D. Algorithme Générique

La description de l'algorithme est celle présentée dans [Le Charlier 2]. On trouve dans ce papier quatre versions d'un algorithme générique de base.

La première, de par sa simplicité, est davantage destinée à faciliter la compréhension de l'algorithme générique, qu'à servir de support à une implémentation effective.

La deuxième représente un niveau de raffinement supplémentaire par rapport à la première version. On optimise le test de terminaison de l'algorithme au moyen d'un Booléen signalant l'intégrité de l'ensemble des tuples calculés. C'est celle qui a été choisie dans [Musumbu].

La quatrième version représente le degré final de raffinement de l'algorithme et a été choisie par Pascal Van Hentenryck pour être implémentée en Pascal. Les améliorations portent cette fois sur la reconnaissance des tuples définitifs, c'est-à-dire dont la valeur ne doit plus être améliorée. On peut trouver l'évaluation de cette implémentation dans [Le Charlier 3].

C'est la troisième version de l'algorithme qui a été choisie dans ce travail et qui va maintenant être exposée.

### 1. Présentation intuitive

D'une manière générale, l'algorithme reprend la structuration inspirée de la sémantique. On peut rappeler<sup>1</sup> que le but de l'interpréteur est de trouver une substitution de sortie  $\beta_{\text{out}}$  lorsqu'on lui procure une substitution d'entrée  $\beta_{\text{in}}$  et un appel de procédure  $p$  contenu dans un programme  $P$  à sa disposition. Il s'agit de trouver le tuple  $(\beta_{\text{in}}, p, \beta_{\text{out}})$  tel que  $\beta_{\text{out}}$  est la meilleure approximation possible de la substitution après exécution de  $p$ , c'est-à-dire un point fixe.

Le coeur de la sémantique était la transformation TETC, qui appelait en cascade les trois fonctions, jusqu'à obtention du point fixe; de même, on peut envisager une

---

<sup>1</sup> cf [Musumbu : I.78], [Le Charlier 2 : 15]

transformation abstraite, TETA, et les trois fonctions abstraites correspondantes. Comme précédemment, la sémantique est définie par le point fixe de TETA:

- $\eta_0 = \perp$
- $\eta_{i+1} = \text{TETA}(\eta_i)$
- $\eta_n = \text{TETA}(\eta_n)$

Cette approche va avoir le défaut de générer un grand nombre de tuples superflus, aussi va-t-on chercher à ne calculer que les tuples vraiment nécessaires ou, plus vraisemblablement, à limiter le nombre de calcul de ceux qui s'avéreront inutiles.

L'approche adoptée dans l'algorithme utilisé consiste à calculer par raffinements successifs une suite d'approximations  $\eta_0 < \eta_1 < \dots < \eta_n$ , où  $(\beta_{in}, p, \beta_{out}) \in \eta_n$ , tout en ne calculant qu'un minimum d'éléments superflus. Cette suite est caractérisée comme suit :

- $\eta_0 = \{(\beta_{in}, p, \perp)\}$
- $\eta_{i+1} = \eta_i \cup (\alpha_{in}, q, \alpha_{out})$  avec
  - $(\alpha_{in}, q)$  n'est pas basé dans  $\eta_i$  ou
  - $T_p(\alpha_{in}, q, \eta_i) > \eta_i(\alpha_{in}, q)$
- $\eta_n \mid \forall (\alpha, q) \in \eta_n : T_p(\alpha, q, \eta_n) \leq \eta_n(\alpha, q)$  et  $(\alpha, q)$  est basé dans  $\eta_n$

On voit que l'algorithme progresse à  $\eta_{i+1}$  en sélectionnant un nouveau tuple dont les deux premiers éléments — $\beta_{in}$  et  $p$ — n'apparaissent dans aucun tuple de  $\eta_i$ , ou parmi ceux de  $\eta_i$  un dont le troisième élément peut être amélioré.

Soit  $\eta$  et  $\eta'$  des ensembles de tuples abstraits, une paire  $(\beta, p)$  est *basée* dans un  $\eta$  lorsque :  $\forall \eta' \mid \eta \subseteq \eta' : \text{le calcul de } T_p(\beta, p, \eta') \text{ ne nécessite pas des valeurs de } \eta' \text{ qui n'appartiennent pas à } \eta$ .

$\eta(\beta, p)$  est la fonction d'extraction d'un tuple à partir d'un  $\eta$  : pour la paire  $(\beta, p)$ , on recherche le tuple dont les deux premiers éléments correspondent à cette paire et on renvoie le troisième élément.

Dans ces conditions,  $T_p(\beta, p, \eta)$  peut être étendu à de tels  $\eta$ .

L'algorithme va donc partir du couple de départ  $(\beta_{in}, p)$  et lui appliquer  $T_p(\beta, p, \eta)$  avec un  $\eta$  vide. Celle-ci va appeler à son tour les deux autres fonctions et ainsi de suite puisque l'algorithme est récursif. En effet, dans le cas où le but en entrée de la fonction  $T_b$  est un appel de procédure, on va aller voir dans  $\eta$  pour y trouver la substitution de sortie qui correspond à cet appel de procédure. Celle-ci risque de n'être qu'une approximation non définitive du tuple final pour cette procédure, auquel cas on devra de nouveau faire appel à  $T_p$ . Le calcul de la clause sera poursuivi après que ce nouvel appel à  $T_p$  ait fourni un résultat définitif.

L'algorithme procède donc par enrichissement progressif de  $\eta$ . Il faut donc définir deux nouvelles opérations devant permettre cet enrichissement : ce sera les deux opérations introduites ci-dessus, ADJUST et EXTEND.

L'algorithme va être réparti sur cinq procédures :

Soit EPA désigne un ensemble de paires  $(\beta, p)$  et ETA un ensemble de tuples.

- SOLVE(Données  $\beta_{in}$ : Substitution,  $p$ : Procédure; Résultats  $\beta_{out}$ : Substitution)

Cette procédure sera appelée une fois avec la paire  $(\beta_{in}, p)$  et fournira la substitution résultant de l'interprétation abstraite. Elle va initialiser les ensembles de tuples utilisés par l'algorithme.

- SOLVEGOAL(**Données**  $\beta_{in}$ : Substitution,  $p$ : Procédure, **suspended**: EPA; **Données modifiées** use:EPA, sameSat: Booléen, eta: ETA, def: EPA)

Cette procédure reçoit en entrée la paire  $(\beta, p)$  et les ensembles de Tuples utilisés par l'algorithme. Il faut noter que la procédure ne sera pas nécessairement celle d'entrée dans SOLVE, ce peut être une autre procédure auquel il est fait appel dans la procédure de départ, et qui est définie dans le programme. SOLVEGOAL va renvoyer une approximation peut-être meilleure de eta, l'ensemble des tuples qui représente l'information récoltée jusqu'à présent par l'interprétation.

Afin d'accélérer la reconnaissance d'un cas déjà présent dans eta, on utilise un ensemble de paires, **suspended**. SOLVEGOAL va tester l'appartenance de  $(\beta_{in}, p)$  à **suspended** et, si la paire est présente, renvoyer directement l'approximation contenue dans eta. Sinon, on va étendre eta avec cette paire et faire appel à la procédure SOLVEPROCEDURE, si du moins la paire  $(\beta_{in}, p)$  ne correspond pas à un tuple définitif.

- SOLVEPROCEDURE(**Données**  $\beta_{in}$ : Substitution,  $p$ : Procédure, **suspended**: EPA; **Résultats** use: EPA; **Données modifiées** sameSat: Booléen, eta: ETA, def: EPA)

Cette procédure renferme la boucle principale de l'algorithme. Il y aura itération jusqu'à obtention d'un résultat définitif pour la procédure  $p$  relativement à la substitution  $\beta_{in}$ . L'itération se fait par appel à la procédure SOLVEALLCLAUSES.

La terminaison est assurée par l'utilisation d'un signal —sameSat— qui indiquera quand la dernière itération n'apporte plus aucune information et donc plus de modifications à l'ensemble eta.

- SOLVEALLCLAUSES(**Données**  $\beta_{in}$ : Substitution,  $p$ : Procédure, **suspended**: EPA; **Résultats** use: EPA, sameSat: Booléen; **Données modifiées** eta: ETA, def: EPA)

La procédure SOLVEALLCLAUSES s'occupe de l'itération pour l'ensemble des clauses d'une procédure. On retrouve ici la fonction  $T_p$  de la sémantique. Lorsque l'ensemble des clauses d'une procédure ont été prises en compte, il faut ajuster l'ensemble eta si le résultat pour la procédure peut lui apporter quelque information nouvelle.

C'est dans cette procédure qu'on trouve l'initialisation du signal sameSat à Vrai. Il sera abaissé à Faux dans deux cas : le recours à EXTEND dans SOLVEGOAL et celui à ADJUST ici dans SOLVEALLCLAUSES, c'est-à-dire qu'un eta est enrichi et que l'ensemble des résultats partiels sont à reconsidérer.

C'est dans SOLVEALLCLAUSES aussi qu'on va trouver l'initialisation de l'ensemble use de paire  $(\beta, p)$ . Il va contenir les deux premiers éléments des tuples de eta en utilisation depuis le dernier appel à SOLVEPROCEDURE. Conjointement, on utilise un ensemble similaire def qui répertorie les tuples de eta dont la valeur est définitive et auquel on pourra faire appel sans avoir besoin de vérifier si une approximation supplémentaire n'est pas possible. Ces deux ensembles servent en quelque sorte d'index pour l'ensemble eta et accélèrent les recherches de tuples dans cet ensemble.

- SOLVECLAUSE(Données  $\beta_{in}$ : Substitution, c: Clause, suspended: EPA; Résultats  $\beta_{out}$ : Substitution; Données modifiées use: EPA, sameSat: Booléen, eta: ETA, def: EPA)

Cette procédure implémente le contenu sémantique des fonctions  $T_c$  et  $T_b$ . Il s'agit de calculer la substitution *après* exécution de la clause c. On itère ici sur chacun des buts de la clause. Si un but est un appel de procédure, on fait appel à SOLVEGOAL, ce qui remplace le recours direct à eta, tel qu'il est spécifié dans la sémantique.

## 2. Présentation formelle

L'algorithme est présenté<sup>1</sup> sous la forme de quatre procédures. On utilisera un pseudo-langage "à la Pascal".

```

procedure SOLVE(Données  $\beta_{in}$ : Substitution, p: Procédure;
|           Résultats  $\beta_{out}$ : Substitution)
| variables sameSat: Booléen, use, def:EPA, eta:ETA;
| sameSat  $\leftarrow$  Vrai;
| use  $\leftarrow$  VIDE; def  $\leftarrow$  VIDE; eta  $\leftarrow$  VIDE;
| SOLVEGOAL( $\beta_{in}$ , p, VIDE, use, sameSat, eta, def);
|  $\beta_{out}$   $\leftarrow$  eta( $\beta_{in}$ , p);

```

```

procedure SOLVEGOAL(Données  $\beta_{in}$ : Substitution, p:Procédure,
|           suspended: EPA;
|           Données modifiées use:EPA, sameSat: Booléen,
|           eta: ETA, def: EPA)
| variables useaux: EPA;
| si ( $\beta_{in}$ , p)  $\in$  suspended alors
|   | use  $\leftarrow$  use  $\cup$  {( $\beta_{in}$ , p)}
| sinon
|   | si ( $\beta_{in}$ , p)  $\notin$  def alors
|     |   | si ( $\beta_{in}$ , p)  $\notin$  dom(eta) alors
|       |   |   | sameSat  $\leftarrow$  Faux;
|       |   |   | eta  $\leftarrow$  EXTEND( $\beta_{in}$ , p, eta);
|       |   |   | SOLVEPROCEDURE( $\beta_{in}$ , p, suspended, useaux, sameSat, eta, def);
|       |   |   | si useaux = VIDE alors
|       |   |     | def  $\leftarrow$  def  $\cup$  {( $\beta_{in}$ , p)}
|       |   |     | sinon
|       |   |       | use  $\leftarrow$  use  $\cup$  useaux

```

<sup>1</sup> [Le Charlier 2: 64...66]

```

procedure SOLVEPROCEDURE(Données  $\beta_{in}$ : Substitution,
p:Procedure,
|
|           suspended : EPA;
|           Résultats use:EPA;
|           Données modifiées sameSat: Booléen, eta: ETA,
|           def: EPA)
| variables sameSataux: Booléen;
| répéter
|   | SOLVEALLCLAUSES( $\beta_{in}$ ,p,suspended U  $\{(\beta_{in},p)\}$ ,use,sameSataux,
|   |   eta,def);
|   | sameSat  $\leftarrow$  sameSat  $\wedge$  sameSataux
| jusqu'à sameSataux
| use  $\leftarrow$  use \  $\{(\beta_{in},p)\}$ 

```

```

procedure SOLVEALLCLAUSES(Données  $\beta_{in}$ : Substitution,
p:Procedure,
|
|           suspended : EPA;
|           Résultats use:EPA,sameSat: Booléen;
|           Données modifiées eta: ETA,def: EPA)
| variables  $\beta_{out}$ , $\beta_{aux}$ : Substitution;
|  $\beta_{out} \leftarrow \perp$ 
| use  $\leftarrow$  VIDE
| sameSat  $\leftarrow$  Vrai
| pour i variant de 1 à m où  $c_1, \dots, c_m$  clauses de p
|   | SOLVECLAUSE( $\beta_{in}$ , $c_i$ ,suspended, $\beta_{aux}$ ,use,sameSat,eta,def);
|   |  $\beta_{out} \leftarrow$  UNION( $\beta_{out}$ , $\beta_{aux}$ )
| si  $\neg(\beta_{out} \leq eta(\beta_{in},p))$  alors
|   | eta  $\leftarrow$  ADJUST( $\beta_{in},p,\beta_{out},eta$ );
|   | sameSat  $\leftarrow$  Faux

```

```

procedure SOLVECLAUSE(Données  $\beta_{in}$ : Substitution, c:Clause,
|
|           suspended : EPA;
|           Résultats  $\beta_{out}$ : Substitution;
|           Données modifiées use: EPA,sameSat: Booléen,
|           eta: ETA,def: EPA)
| variables  $\beta_{ext}$ , $\beta_{aux}$ : Substitution;
|  $\beta_{ext} \leftarrow$  EXTC(c, $\beta_{in}$ )
| pour i variant de 1 à m où  $b_1, \dots, b_m$  buts du corps de c
|   |  $\beta_{aux} \leftarrow$  RESTRG( $b_i,\beta_{ext}$ )
|   | décider pour  $b_i$  entre
|   |   |  $x_j = x_k$  :
|   |   |   |  $\beta_{aux} \leftarrow$  AIVAR( $\beta_{aux}$ )
|   |   |  $x_j = f(\dots)$  :
|   |   |   |  $\beta_{aux} \leftarrow$  AIFUNC( $\beta_{aux},f$ )
|   |   |  $x_j = p(\dots)$  :
|   |   |   | SOLVEGOAL( $\beta_{aux},p$ ,suspended,use,sameSat,eta,def);
|   |   |   |  $\beta_{aux} \leftarrow eta(\beta_{aux},p)$ 
|   |  $\beta_{ext} \leftarrow$  EXTG( $b_i,\beta_{ext},\beta_{aux}$ )
|  $\beta_{out} \leftarrow$  RESTRC(c, $\beta_{ext}$ )

```

## III. Implémentation

---

Le programme qui va être décrit ci-dessous tente de mettre en oeuvre l'algorithme qui vient d'être exposé.

La programmation a été réalisée en ANSI C, en raison de la grande flexibilité du langage, notamment en ce qui concerne la déclaration de types et la gestion dynamique de la pile mémoire. De plus, le programme est portable sur toute machine disposant d'un compilateur ANSI C, car il ne fait appel à aucune caractéristique propre à une machine et étrangère au standard ANSI.

On peut commencer par décrire les structures de données représentant le domaine abstrait.

### A. Structures de données

Les éléments entrant dans la composition du domaine abstrait vont être décrits en faisant ressortir la structuration en objets et le regroupement des fonctions qui s'y rapportent en méthodes de ces objets. La description elle-même mélangera des éléments informels du langage naturel et des éléments plus formels inspirés de la logique des prédicats<sup>1</sup>.

On va faire référence à la constante MAXVARIABLES qui désigne le nombre de variables distinctes représentables lors de l'interprétation. Sa valeur va dépendre des ressources disponibles dans le système. Dans cette implémentation, la valeur 16 a été choisie car elle correspond à la taille en bits d'un entier court, tout en permettant l'étude de la plupart des programmes simples. On peut faire remarquer que le standard ANSI C permet d'utiliser des entiers représentés sur 64 bits.

Les fonctions d'affichage ne seront pas décrites.

La description des structures de données va suivre une approche bottom-up, aussi peut-il être utile de donner un bref aperçu de l'approche inverse : un ensemble de tuples abstraits est une liste linéaire de tuples, un tuple est une structure composée d'un appel de procédure et de deux formules, une formule contient une liste linéaire de conjonctions, une conjonction est un entier représentant une table de bits.

L'objet liste linéaire est développé dans un module séparé et ne sera pas décrit : il s'agit d'une implémentation avec doubles sentinelles et index.[cf Sedgewick]

---

<sup>1</sup> PC désigne un produit cartésien, INT une valeur entière et SEQ une séquence d'éléments.

## 1. *Cjct*

Pour des raisons de facilité et d'efficacité, une conjonction est représentée par une suite de binaires : il y a un binaire par variable et celui-ci est à 1 pour signifier que cette variable est ground, il est à 0 sinon. La variable  $X_i$  apparaît donc positivement dans la conjonction si le  $i^{\text{ème}}$  bit est à 1.

Cette représentation permettra d'utiliser simplement un entier pour la réaliser.

Il faut remarquer qu'au niveau de la conjonction, le nombre de variables à représenter est inconnus; seule existe la borne supérieure MAXVARIABLES.

### a. *Structure*

$Cjct$ : SEQ[{0,1}]

$c$  :  $Cjct$

### b. *Invariant*

Length( $c$ ) <= MAXVARIABLES

- \*  $c_i = 1$  si  $X_i$  est Vrai,  $c_i = 0$  si  $X_i$  est Faux
- \* MAXVARIABLES désigne la taille *utile* d'un entier, pas nécessairement sa taille réelle.

### c. *Constructeur*

nouvelleCjct() :  $Cjct$

- \* cette fonction alloue la mémoire nécessaire à une conjonction et l'initialise à 0

Postcondition :

$\exists$  nouvelleCjct

$\wedge \forall i : 1 \leq i \leq \text{MAXVARIABLES} : \text{nouvelleCjct}_i = 0$

copieCjct(c :  $Cjct$ ) :  $Cjct$

- \* cette fonction alloue la mémoire nécessaire à une nouvelle conjonction et y copie le contenu de la conjonction passée en argument.

Précondition :

$\exists c$

Postcondition :

$\exists$  copieCjct

$\wedge \text{Length}(c) = \text{Length}(\text{copieCjct})$

$\wedge \forall i : 1 \leq i \leq \text{Length}(c) : c_i = \text{copieCjct}_i$

$\wedge c$  est inchangé

### d. *Modificateur*

affirmeXiCjct(c :  $Cjct$ ,  $i$  : INT)

- \* cette fonction va mettre le bit  $i$  de la conjonction  $c$  à 1, ce qui revient à s'assurer que la variable  $X_i$  y apparaît positivement



Précondition :

$\exists c \wedge (1 \leq i \leq \text{Length}(c))$

Postcondition :

$\forall j : 1 \leq j \leq \text{Length}(c) : c_j = 1 \text{ si } i=j, c_j \text{ sinon}$

nieXiCjct(c : Cjct, i : INT)

- \* cette fonction va mettre le bit i de la conjonction c à 0, ce qui revient à s'assurer que la variable  $X_i$  n'y apparait pas positivement

Précondition :

$1 \leq i \leq \text{Length}(c)$

Postcondition :

$\forall j : 1 \leq j \leq \text{Length}(c) : c_j = 0 \text{ si } i=j, c_j \text{ sinon}$

effaceCjct(c : Cjct)

- \* cette fonction va mettre tous les bits de la conjonction c à 0, ce qui revient à s'assurer qu'aucune variable n'y apparait positivement

Précondition :

$\exists c$

Postcondition :

$\forall j : 1 \leq j \leq \text{Length}(c) : c_j = 0$

e. *Observateur*

vraiXiCjct(c : Cjct, i : INT) : Booléen

- \* cette fonction renvoie la valeur Vrai si le bit i de la conjonction c est à 1, ce qui revient à tester si la variable  $X_i$  y apparait positivement

Précondition :

$\exists c \wedge 1 \leq i \leq \text{Length}(c)$

Postcondition :

vraiXiCjct = Vrai si  $c_i=1$ , Faux sinon

premierXiCjct(c : Cjct, nbX : INT) : INT

- \* cette fonction renvoie l'indice du premier bit à 1 de la conjonction c parmi les nbX premiers, ce qui revient à tester qu'elle est la première variable  $X_i$  qui y apparait positivement

Précondition :

$\exists c \wedge 1 \leq \text{nbX} \leq \text{Length}(c)$

Postcondition :

$0 \leq \text{premierXiCjct} \leq \text{nbX}$

$\wedge \forall i : 1 \leq i \leq \text{nbX} \wedge c_i = 0 : \text{premierXiCjct} = 0$

$\wedge \text{premierXiCjct} \neq 0 \Rightarrow \forall j : 1 \leq j < \text{premierXiCjct} : c_j = 0$

$\wedge c_{\text{premierXiCjct}} = 1$

$\wedge \forall i : \text{premierXiCjct} \leq i \leq \text{nbX} \wedge c_i = 1 : i \geq \text{premierXiCjct}$

compareCjct(c1, c2 : Cjct) : Booléen

- \* cette fonction renvoie la valeur Vrai si les deux conjonctions sont identiques : l'identité s'entend bit à bit jusqu'au bit d'indice MAXVARIABLES

Précondition :

$\exists c1 \wedge \exists c2 \wedge \text{Length}(c1) = \text{Length}(c2)$

Postcondition :

compareCjct = Vrai si  $\forall i : 1 \leq i \leq \text{Length}(c1) : c1_i = c2_i$ , Faux  
sinon

## 2. Formule

Il s'agit de la structure de base devant représenter les substitutions abstraites. Cette structure est composée de trois éléments : une liste linéaire de conjonctions, la taille de cette liste et la taille d'une conjonction de la liste. La liste de conjonctions ne contient pas de doublons.

### a. Structure

Formule : PC[nbXCjct: INT, nbCjct: INT, djct: SEQ[Cjct]]

f : Formule

### b. Invariant

(0 < nbXCjct(f) <= MAXVARIABLES)

^  $\forall c \in \text{djct}(f) : \text{Length}(c) = \text{nbXCjct}(f)$

^  $\forall c_i, c_j \in \text{djct}(f) : c_i \neq c_j \text{ si } i \neq j$

^ (nbCjct(f) >= 0)

^ (nbCjct(f) = Length(djct(f)))

### c. Constructeur

NouvelleFormule(tailleCjct: INT) : Formule

- \* cette fonction alloue la mémoire nécessaire à une formule et initialise la taille des conjonctions à l'entier passé en argument; la liste de conjonctions est initialisée à 0 éléments

Précondition :

tailleCjct <= NBVARIABLES

Postcondition :

$\exists$  NouvelleFormule

^ nbXCjct(NouvelleFormule) = tailleCjct

^ nbCjct(NouvelleFormule) = 0

copieFormule(f : Formule) : Formule

- \* cette fonction alloue la mémoire nécessaire à une formule et y copie les éléments de la formule passée en argument

Précondition :

$\exists$  f

Postcondition :

$\exists$  copieFormule

^ nbXCjct(f) = nbXCjct(copieFormule)

^ nbCjct(f) = nbCjct(copieFormule)

^ djct(f) = djct(copieFormule)

^ f est inchangé

generettCFormule(nbVar: INT) : Formule

- \* cette fonction alloue la mémoire nécessaire à une formule et initialise la taille des conjonctions à l'entier passé en argument; la liste de conjonctions est initialisée et on la remplit avec les conjonctions représentant toutes les combinaisons distinctes des nbVar bits pris tantôt à 1, tantôt à 0. En fait, il suffit de générer tous les entiers de 0 à  $2^{\text{nbVar}} - 1$

Précondition :  
tailleCjct <= NBVARIABLES

Postcondition :  
 $\exists$  generettCFormule  
 $\wedge$  nbXCjct(generettCFormule) = nbVar  
 $\wedge \forall i : 1 \leq i \leq \text{nbVar} : \exists c \in \text{djct}(\text{generettCFormule})$   
 $\wedge c$  est une combinaison distincte possible des nbVar  $c_j$

*d. Modificateur*

insereCFormule(f: Formule, c: Cjct)

- \* cette fonction ajoute la conjonction c à la liste de la formule f si elle n'y est pas déjà présente; l'endroit de l'insertion est indéterminé

Précondition :  
 $\exists f \wedge \exists c \wedge \text{nbXCjct}(f) = \text{Length}(c)$   
 $\wedge$  il n'y a pas de doublon dans djct(f)

Postcondition :  
 $f = f_0 \cup \{c\}$

ajouteCFormule(f: Formule, c: Cjct)

- \* cette fonction ajoute la conjonction c à la liste de la formule f; l'endroit de l'insertion est déterminé par la position de l'index de la liste

Précondition :  
 $\exists f \wedge \exists c \wedge \text{nbXCjct}(f) = \text{Length}(c)$

Postcondition :  
 $f = f_0 \cup \{c\}$   
 $\wedge c$  est ajouté à l'endroit pointé par l'index de djct(f)

ajouteApresCFormule(f: Formule, c: Cjct, pt : Pointeur)

- \* cette fonction ajoute la conjonction c à la liste de la formule f; l'endroit de l'insertion est déterminé par la position de l'élément suivant celui pointé par l'index de la liste

Précondition :  
 $\exists f \wedge \exists c \wedge \text{nbXCjct}(f) = \text{Length}(c)$   
 $\wedge$  pt pointe sur un élément de djct(f)

Postcondition :  
 $f = f_0 \cup \{c\} \wedge \text{pts} = \text{pt} \rightarrow \text{suisant} \wedge c$  est inséré en pts

supprimeCFormule(f: Formule, c: Cjct)

- \* cette fonction supprime la conjonction c de la liste de la formule f si elle y est présente

Précondition :

$\exists f \wedge \exists c \wedge \text{nbXCjct}(f) = \text{Length}(c)$

Postcondition :

$f = f_0 \setminus \{c\}$

supprimeApresCFormule(f: Formule, pt : Pointeur)

- \* cette fonction supprime de la liste de la formule f la conjonction pointée par pt

Précondition :

$\exists f \wedge \text{pt}$  pointe sur un élément de  $\text{djct}(f)$

Postcondition :

$c = \text{pt} \rightarrow \text{suisant} \wedge f = f_0 \setminus \{c\}$

ajouteXiFormule(f: Formule, i : INT)

- \* cette fonction modifie les conjonctions de la liste de la formule f : toute conjonction voit son bit i mis à 1 si  $i \leq \text{nbXCjct}(f)$ ; si  $i = \text{nbXCjct}(f) + 1$ , on ajoute un bit supplémentaire à chacune des conjonctions de f, ce bit pouvant être à 0 ou à 1

Précondition :

$\exists f \wedge 1 \leq i \leq \text{nbXCjct}(f) + 1$

Postcondition :

si  $i \leq \text{nbXCjct}(f)$  alors  $\forall c \in \text{djct}(f) : c_i = 1$

si  $i = \text{nbXCjct}(f) + 1$  alors  $\text{nbXCjct}(f) = \text{nbXCjct}(f_0) + 1$  et on insère dans f les nouvelles conjonctions  $\text{djct}(f_0) \parallel "0"$  et  $\text{djct}(f_0) \parallel "1"$

retireXiFormule(f: Formule, i : INT)

- \* cette fonction modifie les conjonctions de la liste de la formule f : toute conjonction voit son bit i mis à 0

Précondition :

$\exists f \wedge 1 \leq i \leq \text{nbXCjct}(f)$

Postcondition :

$\forall c \in \text{djct}(f) : c_i = 0$

e. *Observateur*

compareFormule(f1, f2: Formule): Booléen

- \* cette fonction renvoie la valeur Vrai si les deux formules sont identiques : l'identité s'entend pour les trois éléments et donc pour la liste de conjonctions qui doit être triée

Précondition :

$\exists f1 \wedge djct(f1) \text{ est triée } \wedge$   
 $\exists f2 \wedge djct(f2) \text{ est triée } \wedge nbXCjct(f1) = nbXCjct(f2)$

Postcondition :

compareFormule=  
Vrai si  $\forall i: c1_i \in djct(f1) \text{ ou } c2_i \in$   
 $djct(f2) : compareCjct(c1_i, c2_i)$   
Faux sinon

compareBeta(beta1, beta2: Formule): INT

\* Sachant que beta1 et beta2 sont déclarés, comparables et triés, sachant qu'un beta vide est plus petit que tout, cette fonction renvoie un entier

>0 si beta1 > beta2 ou beta1 <> vide et beta2 = vide

<0 si beta1 < beta2 ou beta1 = vide et beta2 <> vide

=0 si beta1 = beta2

l'identité entre formule s'entend pour les trois éléments et donc pour la liste de conjonctions qui doit être triée

Précondition :

$\exists beta1 \wedge djct(beta1) \text{ est triée } \wedge$   
 $\exists beta2 \wedge djct(beta2) \text{ est triée } \wedge nbXCjct(beta1) = nbXCjct(beta2)$

Postcondition :

compareFormule=  
Vrai si  $\forall i: c1_i \in djct(beta1) \text{ ou } c2_i \in$   
 $djct(beta2) : compareCjct(c1_i, c2_i)$   
Faux sinon

rechercheCFormule(f: Formule, c: Cjct): Booléen

\* cette fonction renvoie Vrai si la conjonction c est présente dans la liste de la formule f

Précondition :

$\exists f \wedge \exists c \wedge nbXCjct(f) = Length(c)$

Postcondition :

rechercheCFormule=  
Vrai si  $\exists c' \in djct(f) : compareCjct(c, c')$   
Faux sinon

### 3. Tuple2

Cette structure de donnée est formée de deux éléments : une formule et un appel de procédure. Il ne s'agit pas à proprement parler d'un tuple, mais plutôt des deux premiers éléments d'un tuple. Tuple2 peut servir à identifier un tuple dans une liste de tuples uniques quant à leur deux premiers éléments.

L'appel de procédure est un entier qui sert d'indice dans une table qui reprend tous les prédicats : cette table a été générée lors du parsing.

#### a. Structure

Tuple2: PC[beta : Formule, p : INT]

t : Tuple2

#### b. Invariant

\* p(t) est un indice vers la table de prédicats (issues du parsing du programme)

#### c. Constructeur

nouveauTuple2(beta : Formule, p : INT) : Tuple2

\* cette fonction alloue la mémoire nécessaire à un Tuple2 et l'initialise avec les deux arguments qui lui sont passés

Précondition :

$\exists \text{ beta} \wedge \exists \text{ p}$

Postcondition :

$\exists \text{ nouveauTuple2} \wedge \text{beta}(\text{nouveauTuple2}) = \text{beta} \wedge \text{p}(\text{nouveauTuple2}) = \text{p}$

#### d. Observateur

compareTuple2(t1, t2 : Tuple2) : Booléen

\* cette fonction renvoie la valeur Vrai si les deux Tuple2 sont identiques : la comparaison des formules correspond à la fonction compareBeta

Précondition :

$\exists \text{ t1} \wedge \exists \text{ t2}$

Postcondition :

compareTuple2=

Vrai si  $\text{p}(\text{t1}) = \text{p}(\text{t2}) \wedge \text{compareBeta}(\text{beta}(\text{t1}), \text{beta}(\text{t2}))$

Faux sinon

#### 4. Tuple3

Cette structure correspond au tuple proprement dit, avec ses deux formules et son appel de procédure.

##### a. Structure

Tuple3: PC[beta1: Formule, p: INT, beta2: Formule]

t: Tuple3

##### b. Invariant

p(t) est un indice vers la table de prédicats (issues du parsing du programme)

##### c. Constructeur

nouveauTuple2(beta1 : Formule, p : INT, beta2 : Formule) : Tuple3

\* cette fonction alloue la mémoire nécessaire à un Tuple3 et l'initialise avec les trois arguments qui lui sont passés

Précondition :

$\exists \text{ beta1} \wedge \exists \text{ p} \wedge \exists \text{ beta2}$

Postcondition :

$\exists \text{ nouveauTuple3} \wedge \text{beta1}(\text{nouveauTuple3}) = \text{beta1}$   
 $\wedge \text{p}(\text{nouveauTuple3}) = \text{p} \wedge \text{beta2}(\text{nouveauTuple3}) = \text{beta2}$

##### d. Observateur

compareDomTuple3(t1,t2: Tuple3) : Booléen

\* cette fonction renvoie la valeur Vrai si les deux premiers éléments des deux Tuple3 sont identiques : la comparaison des formules correspond à la fonction compareBeta

Précondition :

$\exists \text{ t1} \wedge \exists \text{ t2}$

Postcondition :

compareDomTuple3 =  
Vrai si  $\text{p}(\text{t1}) = \text{p}(\text{t2}) \wedge \text{compareBeta}(\text{beta1}(\text{t1}), \text{beta1}(\text{t2}))$   
Faux sinon



## 5. ListeTuple2

La liste de Tuple2 est une liste linéaire : elle va servir à désigner un sous-ensemble des éléments contenus dans une liste de Tuple3, les éléments dont le domaine correspond aux éléments de Liste Tuple2.

C'est de ce type que seront les ensembles use, suspended et def.

### a. Structure

ListeTuple2: SEQ[Tuple2]

lt : ListeTuple2

### b. Constructeur

uniListeTuple2(lt1, lt2: ListeTuple2) : ListeTuple2

- \* cette fonction alloue la mémoire nécessaire à une liste de Tuple2 et la remplit avec les éléments des deux listes qui lui sont passées en argument : les éléments de l'intersection des deux listes n'apparaissent qu'une seule fois dans la nouvelle liste

Précondition :

$\exists \text{ lt1} \wedge \exists \text{ lt2}$

Postcondition :

$\exists \text{ uniListeTuple2} \wedge \text{ uniListeTuple2} = \text{lt1} \cup \text{lt2}$

### c. Modificateur

enleveTuple2(lt: ListeTuple2, t: Tuple2)

- \* cette fonction supprime le tuple t de la liste linéaire lt si il s'y trouve

Précondition :

$\exists \text{ lt} \wedge \exists \text{ t}$

Postcondition :

$\text{enleveTuple2} = \text{enleveTuple2}_0 \setminus \{t\}$

### d. Observateur

appartientTuple2(lt: ListeTuple2, t: Tuple2): Booléen

- \* cette fonction renvoie la valeur Vrai si t est un élément de la liste lt

Précondition :

$\exists \text{ lt} \wedge \exists \text{ t}$

Postcondition :

$\text{appartientTuple2} =$   
VRAI si  $t \in \text{lt}$   
FAUX sinon

## 6. ListeTuple3

Cette structure correspond à une liste linéaire de Tuple3. Les éléments de cette liste ne sont pas triés, mais sont uniques dans la liste.

C'est de ce type que sera l'ensemble eta.

### a. Structure

ListeTuple3: SEQ [Tuple3]

lt : ListeTuple3

### b. Observateur

\* cette fonction renvoie Vrai il existe un tuple dans lt dont le domaine correspond à la paire t

appartientDomTuple3(lt: ListeTuple2, t: Tuple2)

Précondition :

$\exists lt \wedge \exists t$

Postcondition :

appartientDomTuple3 =

VRAI si  $t \in lt$

FAUX sinon

## B. Opérations Primitives

Les opérations primitives sont spécifiées d'une manière détaillée dans [Le Charlier 2]. On peut cependant les passer en revue pour indiquer la manière dont elles ont été réalisées pour le domaine abstrait choisi.

### 1. Opérations Principales

Ces opérations suivent de très près l'algorithme donné dans [Le Charlier 2: 64...66]. On va se borner ici à donner les quelques aménagements apportés dans l'implémentation réelle.

#### a. SOLVE

Cette fonction initialise les ensembles de paires —use def, suspended— et l'ensemble des tuples abstraits —eta—. Le signal sameSat est mis à Vrai, ce qui donne 1 en C.

L'appel à SOLVEGOAL ne demande pas de commentaires. Cette fonction renvoie au programme principal la substitution de sortie —une formule—.

#### b. SOLVEGOAL

Dans cette fonction, les tests d'appartenance aux ensembles suspended, def et eta sont assurés par les fonctions précédemment décrites. De même pour la mise à jour de ces ensembles.

On utilise un compteur local pour comptabiliser le nombre d'appels à cette procédure.

#### c. SOLVEPROCEDURE

Dans cette fonction, il faut ajouter puis retirer l'élément  $(\beta_{in}, p)$  de l'ensemble suspended car, en C, on ne passe bien sûr jamais que le pointeur vers une structure.

#### d. SOLVEALLCLAUSES

Dans cette fonction, donner la valeur  $\perp$  à  $\beta_{out}$  revient simplement à initialiser cette formule et à la laisser vide.

L'itération sur les clauses de p se fait par le parcours de la liste linéaire regroupant les clauses d'une procédure dans la représentation interne d'un programme.

#### e. SOLVECLAUSE

L'itération sur les buts de p se fait par le parcours à partir du deuxième élément de la liste linéaire regroupant les clauses d'une procédure dans la représentation interne d'un programme.

### 2. Opérations Supplémentaires

Ces opérations apparaissent dans les précédentes et agissent sur eta.

#### a. EXTEND

L'extension de eta consiste à renvoyer l'union du eta initial et de  $(\beta, p, \beta')$  où  $\beta'$  est le plus grand des  $\beta_2$  —troisième élément du tuple— des tuples trouvés lorsqu'on recherche dans eta les tuples de domaine  $(\beta_{in}, p)$  et de  $\beta_{in}$  inférieur ou égal à  $\beta$  ( $\beta_{in}$

est l'un des arguments de EXTEND). Si  $\eta$  ne contient pas de tel tuple, on utilise une formule de 0 conjonctions pour représenter  $\perp$ .

*b. ADJUST*

Cette fonction reçoit en argument la paire  $(\beta, p)$  : elle va ajuster dans  $\eta$  les tuples de domaine égal à cette paire en remplaçant leur troisième élément par le  $\beta'$  fourni en argument si il est plus grand.

*c. UNION*

Cette fonction se limite à un appel à la fonction uniFormule présentée plus haut.

*d. ETA*

Cette fonction renvoie le troisième élément du tuple de domaine égal aux deux premiers arguments. On utilise ici la méthode rechercheElement de l'objet ListeLinéaire.

### *3. Opérations sur le domaine*

Il s'agit des opérations primitives propres au domaine abstrait et utilisées par l'algorithme générique.

*a. EXTC*

Cette fonction fait appel à la fonction LEXT avec en argument une copie de la formule qu'elle reçoit elle-même en argument.

*b. RESTRC*

On crée ici une nouvelle formule reprenant les conjonctions de celle qui est passée en argument : les conjonctions de la nouvelle formule n'ont des bits significatifs qu'à concurrence du nombre de variables dans l'entête de la clause passée en argument.

*c. EXTB*

Après un appel à CHGVAR qui ajoute à  $\beta_{aux}$  et y renomme l'ensemble des variables représentées dans la table de concordance  $\gamma$ , cette fonction commence par construire un extracteur, une espèce de filtre permettant de ne garder d'une formule que les bits à considérer, c'est-à-dire ceux qui selon la table  $\gamma$ , on fait l'objet d'un renommage. Le filtre est en fait un masque logique qu'on utilisera avec un ET logique sur une conjonction de même taille.

Ensuite, pour toutes les conjonctions de  $\beta_{ext}$ , qui sont celles qui apparaissent dans le but traité juste avant l'appel à EXTB, on peut extraire les bits significatifs. La conjonction filtrée, extrait, aura ses bits à 0 sauf si le bit correspond à un bit à un du masque.

Pour toute conjonction de la formule  $\beta_{aux}$  obtenue après traitement du but précédent, on va effectuer une comparaison avec l'extrait : si la conjonction filtrée apparaît dans  $\beta_{aux}$ , on la maintient dans  $\beta_{ext}$  sinon on la supprime.

*d. RESTRB*

Cette formule commence par créer une table de concordance  $\gamma$  qui tient trace des indices précédents des variables retenues dans le but passé en argument. La formule à restreindre est recopiée et modifiée avant d'être renvoyée.

La table  $\gamma$  est déclarée globalement pour le module "primitive.c" afin de ne pas modifier les entêtes des procédures telles qu'elles apparaissent dans [Le Charlier 2];  $\gamma$  est de type formule.

Pour chacun des bits de chacune des conjonctions de la formule, on va vérifier si il existe un bit de même indice et mis à 1 dans l'une des conjonctions représentant la liste des variables du but. Si c'est le cas, la variable est présente dans le but et il faut la renommer. Le renommage s'effectue de manière incrémentale en partant de 1: la  $i^{\text{ème}}$  variable à renommer correspondra au  $i^{\text{ème}}$  bit de la nouvelle conjonction. La table de concordance va mettre en rapport ce nouvel indice  $i$  et l'indice précédent du bit dans la conjonction de la formule avant renommage.

On a donc une triple itération : pour les conjonctions de la formule, pour les bits de la conjonction, pour chaque variable du but.

*e. AIFUNC*

Cette fonction se limite à insérer la éléments de la représentation binaire de l'unification de  $x_1$  et de  $x_2...x_n$  : on remarque que la conjonction où toutes les variables apparaissent positivement —11...111—, est toujours présente, ce qui est cohérent avec le résultat de [Cortesi: 4].

*f. AIVAR*

Cette fonction se limite à insérer 11 en lieu et place d'éventuelles conjonctions 01 et 10 dans la formule passée en argument.

*g. LEXT*

Cette fonction effectue une itération sur le nombre de variables à ajouter : l'itération consiste à appeler la fonction ajouteXiFormule en spécifiant que la variable à ajouter a un indice de 1 supérieur à la taille de la formule auquel il faudra l'ajouter.

*h. CHGVAR*

Cette fonction renvoie une copie de la formule passée en argument : cette copie est étendue à l'ensemble des variables représentées dans la table de concordance  $\gamma$ . Les variables sont renommées, ce qui renverse un précédent renommage. Ceci correspond au niveau de la conjonction à une permutation des bits.

## C. Parsing

L'acquisition du texte du programme logique à interpréter se fait par l'intermédiaire d'un tampon en mémoire centrale dans lequel l'entièreté du texte sera tout d'abord lue.

La conversion en représentation interne de ce texte est assurée principalement par deux fonction : LireFichier et ParseBut.

Les chaînes de caractères identifiant les appels de procédures sont stockés dans une table —tablePred—, qui lie par ailleurs ces appel de procédure à leur représentation interne. Cette table est déclarée globalement et est initialisée avec l'opérateur de liste —[]— en premier élément.

Les variables sont représentées par des listes de binaires qui sont tous à 0 sauf celui dont le rang correspond à l'indice de la variable. La taille de la liste est fixée par le nombre de variables dans la clause à laquelle appartient la variable. Afin de

pouvoir représenter une variable dès la première lecture du texte, on requiert de chaque clause d'être précédée du nombre des variables qui y apparaissent.

La représentation interne d'un programme se fait de nouveau par une hiérarchie de liste linéaire de structures : un programme est une liste de procédures, une procédure contient l'appel de procédure qui l'identifie —un entier indice de la table des prédicats— et une liste de clauses, une clause contient une liste de buts ainsi que le nombre de variables dans son entête et dans son corps, un but est une structure qui contient :

- un type : une énumération {UNIVAR, UNIFONCT, PREDICAT}
- un symbole : NULL pour UNIVAR, l'entier un pour UNIFONCT puisque le seul foncteur admis est le constructeur de liste, et un indice de la table des prédicats sinon
- une liste de variables : la structure vient d'en être décrite; pour une unification du type  $X_1=...$ ,  $X_i$  est la première variable.

## IV. Exemples d'exécution

---

Afin d'illustrer la réalisation de l'interpréteur qui vient d'être décrite, il s'agit de tester le programme obtenu sur quelques exemples de programmes logiques normalisés. Mais d'abord, voici un mode d'emploi sommaire.

### A. Mode d'emploi

Le programme logique à interpréter doit se trouver dans un fichier de type texte, appelé "programme.iab" et situé dans le même répertoire que la version exécutable du programme d'interprétation. Cette interprétation portera sur la première procédure du programme logique.

La substitution de départ se trouve dans le fichier de type texte "betaIn.iab". Il est composé d'une suite d'entiers séparés par des blancs : le premier désigne le nombre de variables de la première procédure du programme, les éventuels suivants désignent les indices des variables dont on sait qu'elles sont ground avant exécution.

Ainsi, par exemple,

betaIn : "3 1 2 <EOF>"

représente une substitution portant sur trois variables où la première et la deuxième sont ground, ce qui va donner après transformations :

$\{X_1 \leftarrow \text{ground}, X_2 \leftarrow \text{ground}, X_3 \leftarrow \text{any}\}$

↓

$X_1 \wedge X_2 \wedge X_3 \vee X_1 \wedge X_2 \wedge \neg X_3$

↓

$\{\{1,1,1\}, \{1,1,0\}\}$  ou  $\{7,3\}$

De même,

betaIn : "2 <EOF>"

représente une substitution portant sur deux variables dont on ne sait rien, ce qui va donner après transformations :

$$\begin{aligned}
& \{X_1 \leftarrow \text{any}, X_2 \leftarrow \text{any}\} \\
& \Downarrow \\
& X_1 \wedge X_2 \quad \vee \quad \neg X_1 \wedge X_2 \quad \vee \quad X_1 \wedge \neg X_2 \quad \vee \quad \neg X_1 \wedge \neg X_2 \\
& \Downarrow \\
& \{\{1,1\}, \{1,0\}, \{0,1\}, \{0,0\}\} \text{ ou } \{3,2,1,0\}
\end{aligned}$$

Pour revenir au programme lui-même, il suivra la syntaxe décrite précédemment pour les programmes normalisés, à ceci près : chaque clause devra être précédée d'un préfixe de la forme "{m,n}" où m sera le nombre de variables dans l'entête de la clause suivante; n sera le nombre de variables présentes dans le corps de cette clause et qui ne sont pas reprises dans l'entête.

De plus, on suppose une clause vide en fin de programme et donc le préfixe "{0,0}" qui signale la fin du programme. Ces ajouts syntaxiques permettent de faciliter une lecture du texte et une allocation dynamique de la mémoire en un seul passage.

On peut modifier la syntaxe déjà introduite :

Préfixe = "{" Entier "," Entier "}".

ClauseNorm = Préfixe AppelProc ("←" | ":-" ) "(" { AtomeNorm } ")" ".".

PgmNorm = ProcNorm { ProcNorm } "{0,0}".

Ainsi, voici le texte de l'exemple déjà présenté tel qu'il devra apparaître:

```

{3,0}
append (X1, X2, X3) :- X1=[], X2=X3.
{3,3}
append (X1, X2, X3) :- X1=[X4 | X5], X3=[X4 | X6],
    append (X5, X2, X6).
{0,0}

```

La substitution résultant de l'interprétation sera affichée à l'écran et sera contenue dans le fichier "résultat.iab" avec une trace sommaire de l'exécution. Une trace plus complète reprenant la représentation des états intermédiaires des structures de données utilisées se trouve dans le fichier "debug.iab".

## B. append

Append est l'exemple classique de programme de concaténation de listes : il est composé de plusieurs clauses et présente les trois cas possibles d'atomes normalisés. Le programme a déjà été présenté à plusieurs reprises.

On peut commencer avec la substitution d'entrée telle que la troisième variable sera ground, soit

$$\begin{aligned}
& \{X_1 \leftarrow \text{any}, X_2 \leftarrow \text{any}, X_3 \leftarrow \text{ground}\} \\
& \Downarrow \\
& X_1 \wedge X_2 \wedge X_3 \quad \vee \quad \neg X_1 \wedge X_2 \wedge X_3 \quad \vee \quad X_1 \wedge \neg X_2 \wedge X_3 \quad \vee \quad \neg X_1 \wedge \neg X_2 \wedge X_3
\end{aligned}$$

Voici la trace sommaire telle qu'elle apparaît dans le fichier "résultat.iab":



```

lecture de betaIn
  ( $\neg X1 \wedge \neg X2 \wedge X3$ ) | ( $X1 \wedge \neg X2 \wedge X3$ ) | ( $\neg X1 \wedge X2 \wedge X3$ ) | ( $X1 \wedge X2 \wedge X3$ )
SOLVEGOAL in #1 pour append
  EXTEND #1 pour append
    ( $\neg X1 \wedge \neg X2 \wedge X3$ ) | ( $X1 \wedge \neg X2 \wedge X3$ ) | ( $\neg X1 \wedge X2 \wedge X3$ ) | ( $X1 \wedge X2 \wedge X3$ )
  SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #1
      SOLVECLAUSE 1 in #1
        AIFUNC dans clause 1
        AIVAR dans clause 1
      SOLVECLAUSE 1 out #1
        ( $X1 \wedge X2 \wedge X3$ )
      SOLVECLAUSE 2 in #2
        AIFUNC dans clause 2
        AIFUNC dans clause 2
SOLVEGOAL in #2 pour append
SOLVEGOAL out #2 pour append
  formule vide de nbX= 3
  SOLVECLAUSE 2 out #2
  formule vide de nbX= 3
  ADJUST #1
  SOLVEALLCLAUSES out #1
    ( $X1 \wedge X2 \wedge X3$ )
  SOLVEALLCLAUSExS in #2
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
    SOLVECLAUSE 1 out #1
      ( $X1 \wedge X2 \wedge X3$ )
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
SOLVEGOAL in #3 pour append
SOLVEGOAL out #3 pour append
  ( $X1 \wedge X2 \wedge X3$ )
  SOLVECLAUSE 2 out #2
  ( $X1 \wedge X2 \wedge X3$ )
  SOLVEALLCLAUSES out #2
  ( $X1 \wedge X2 \wedge X3$ )
  SOLVEPROCEDURE out
SOLVEGOAL out #1 pour append
eta en sortie
  Tuple3 de symbole #2 : append
    beta1
      4 001
      5 101
      6 011
      7 111
    beta2
      7 111
  ... 1 tuples dans la liste

... and the winner is ...
  ( $X1 \wedge X2 \wedge X3$ )

```

En regard de cette trace et de sa version plus étendue<sup>1</sup> que l'on trouvera en annexe, on va pouvoir commenter les étapes successives de l'exécution de l'interpréteur.

Lors du premier appel à SOLVEGOAL, eta est vide et doit être "étendu" pour contenir le tuple ( $\beta_{in}$ , "append",  $\perp$ ) [A1:55]. eta est modifié et sameSat doit être mis à Faux. La procédure append n'a pas encore été traitée et doit être mise dans l'ensemble des tuples en suspens (suspended) [A1:65]. Ça a lieu dans SOLVEPROCEDURE.

Ensuite, SOLVECLAUSE va traiter la première clause, qui est formée de deux built-in: une unification de foncteur qui est prise en charge par AIFUNC [A1:94] et une unification de variables, prise en charge par AIVAR [A1:112]. Avant tout cela, le  $\beta_{in}$  aurait dû être étendu aux variables supplémentaires du corps de la clause, mais il n'y en a pas et EXTC n'a pas d'effet [A1:78].

Avant l'appel à la procédure d'unification correspondante, la substitution devra être filtrée pour ne garder que les variables du but: une première fois, on garde  $X_1$  qui peut être Vrai ou Faux [A1:87]. Dans le sens inverse, on va étendre le résultat de l'unification aux variables du but par EXTB [A1:96]. Une seconde fois, on garde  $X_2$  et  $X_3$ , ce qui donne après renommage de la nouvelle substitution :  $\neg X_1 \wedge X_2 \vee X_1 \wedge X_2$  [A1:104]. Le renommage devra pouvoir être effectué dans l'autre sens, aussi doit-on créer une table de concordance entre les anciens indices et les nouveaux [A1:107]. La substitution est passée à AIVAR puis à EXTB : le résultat vaut  $X_1 \wedge X_2 \wedge X_3$  [A1:114].

De même que EXTC n'avait pas eu d'effet, RESTRC ne modifie rien à la substitution issue du deuxième appel à EXTB.

SOLVEALLCLAUSES, qui assure l'itération pour toutes les clauses, appelle SOLVECLAUSE une deuxième fois pour la deuxième clause.

Cette fois EXTC va modifier la substitution d'entrée : de trois variables, on passe à six et on obtient une disjonction de 32 conjonctions après extension aux trois variables libres [A1:141]. Les deux premiers buts de cette clause, qui sont des built-in, sont traités d'une manière similaire à ce qui vient d'être décrit ([A1:196] puis [A1:236]). Le troisième est un appel de procédure récursif à append avec une substitution d'entrée identique au  $\beta_{in}$  de départ. SOLVEGOAL est appelé [A1:261] et cette fois l'ensemble suspended contient la paire formée de append et du  $\beta_{in}$ .

append qui était en suspens est à présent utilisé par un autre prédicat —lui même—. On le range donc dans l'ensemble de paires use [A1:290] et on renvoie à la procédure SOLVEGOAL appelante l'approximation contenue dans eta. La substitution est donc  $\perp$  [A1:297] et EXTB n'a aucun effet, pas plus que RESTRC [A1:305].

L'union des substitutions issues de l'interprétation des deux clauses du programme donne  $X_1 \wedge X_2 \wedge X_3$  [A1:311] et ce résultat est évidemment plus précis que celui contenu dans eta, aussi faut-il ajuster celui-ci [A1:324]. Cette nouvelle modification de eta met le Booléen sameSat à Faux et l'itération dans SOLVEPROCEDURE génère un nouvel appel à SOLVEALLCLAUSES et ses deux appels à SOLVECLAUSE ([A1:342] puis [A1:404]). Tout se déroule comme précédemment jusqu'à l'appel récursif à SOLVEGOAL. Cette fois, la valeur

---

<sup>1</sup> les références à cette trace étendue auront la forme suivante "[A1:n], où n sera le numéro de la ligne à laquelle il faut se référer.

trouvée dans  $\beta_{in}$  est  $X_1 \wedge X_2 \wedge X_3$  [A1:566] et c'est cette substitution aussi qui résultera de l'union pour les deux clauses [A1:580].

Cette fois,  $\beta_{in}$  n'est pas ajusté car la substitution n'apporte rien par rapport au tuple déjà présent. `sameSat` avait été mis à Vrai en début de `SOLVEALLCLAUSES` et puisqu'on ne touche pas à  $\beta_{in}$ , il le reste. L'itération de `SOLVEPROCEDURE` se termine et on peut retirer le tuple de ceux qui sont en utilisation [A1:599] et l'approximation actuelle est fournie comme réponse [A1:608].

On peut maintenant présenter un tableau récapitulatif des résultats<sup>1</sup> obtenus pour diverses utilisations de `append` :

$X_i$ ground dans $\beta_{in}$	$\beta_{out}$	itérations
$X_1$	$X_1 \wedge (X_2 \Leftrightarrow X_3)$	3
$X_2$	$X_2 \wedge (X_1 \Leftrightarrow X_3)$	4
$X_3$	$X_1 \wedge X_2 \wedge X_3$	3
$X_1$ et $X_2$	$X_1 \wedge X_2 \wedge X_3$	3
$X_1, X_2$ et $X_3$	$X_1 \wedge X_2 \wedge X_3$	3
aucun	$(X_1 \wedge (X_2 \Leftrightarrow X_3)) \vee (X_2 \wedge (X_1 \Leftrightarrow X_3)) \vee (X_1 \Leftrightarrow X_2 \Leftrightarrow X_3)$	4

La trace de ces différents cas de  $\beta_{in}$  est présentée en Annexe.

Pour ce qui est du temps d'exécution, il est perceptiblement nul sur un Macintosh SE/30.

## C. quicksort

Il est intéressant de tester à présent un programme moins trivial : le tri rapide — Quick Sort—.

Vu la simplicité du domaine couvert par cet interpréteur, le programme a dû être adapté tout en maintenant une parfaite équivalence pour ce qui concerne l'interprétation abstraite. Le domaine ne permet pas l'utilisation d'opérateur arithmétique : on a remplacé celui-ci par une nouvelle procédure, `inegal`, dont le seul effet est d'instancier les opérandes

Voici le programme,

```
{2,1}
sort (X1, X2) :-
  X3=[], quicksort (X1, X2, X3).
{3,0}
quicksort (X1, X2, X3) :-
  X1=[], X3=X2.
{3,6}
quicksort (X1, X2, X3) :-
  X1=[X4|X5],
  partition (X5, X4, X6, X7),
  quicksort (X7, X8, X3),
  X9=[X4|X8],
  quicksort (X6, X2, X9).
```

<sup>1</sup> par nombre d'itérations on entend le nombre d'appels à `SOLVEGOAL`

```

quicksort (X6, X2, X9).

{4,0}
partition (X1, X2, X3, X4) :-
  X1=[], X3=[], X4=[].
{4,3}
partition (X1, X2, X3, X4) :-
  X1=[X5|X6], X3=[X5|X7], inegal(X5, X2),
  partition (X6, X2, X7, X4).
{4,3}
partition (X1, X2, X3, X4) :-
  X1=[X5|X6], X4=[X5|X7], inegal(X5, X2),
  partition (X6, X2, X3, X7).

{2,0}
inegal (X1, X2) :-
  X1=[], X2=[].
{0,0}

```

On peut interpréter ce programme pour les deux possibilités suivantes de  $\beta_{in}$  et obtenir les résultats ci-dessous :

$X_i$ ground dans $\beta_{in}$	$\beta_{out}$	itérations
$X_1$	$X_1 \wedge X_2$	32
$X_1, X_2$	$X_1 \wedge X_2$	38

La trace pour le premier cas est reproduite en annexe.

On remarque que près de la moitié de l'exécution sert à obtenir des tuples inutiles dans lesquels  $\beta_1$  est vide. Ce phénomène est dû à la succession des deux appels de procédure à append : après le premier appel récursif à SOLVEGOAL, le programme renvoie la liste vide et c'est de cette liste que doit partir le deuxième appel récursif, ce qui demande un recalcul de toutes les procédures appelées avec la liste vide en  $\beta_{in}$ .

## D. queens

On peut, pour terminer, envisager un programme d'une taille un peu plus conséquente. Il s'agit du problème classique dit des "huit reines". Pour rappel, l'objectif de ce programme est de déterminer le positionnement de huit reines sur un échiquier d'une manière telle qu'elles ne se menacent pas l'une l'autre.

Ce programme est composé de sept procédures regroupant un total de onze clauses, pour un maximum de dix variables par clause. On trouve ci-dessous le texte du programme.

```

{2,0}
queens (X1, X2) :-
  perm (X1, X2), safe (X2).

{2,0}
perm (X1, X2) :-
  X1=[], X2=[].
{2,6}
perm (X1, X2) :-
  X1=[X3|X4], X2=[X5|X6], X7=[X3|X4],
  delete (X5, X7, X8),
  perm (X8, X6).

```

```

{3,0}
delete (X1, X2, X3) :-
    X2=[X1|X3].
{3,3}
delete (X1, X2, X3) :-
    X2=[X4|X5], X3=[X4|X6],
    delete (X1, X5, X6).

{1,0}
safe (X1) :-
    X1=[].
{1,3}
safe (X1) :-
    X1=[X2|X3], X4= [],
    noattack (X2, X3, X4),
    safe(X3).

{3,0}
noattack (X1, X2, X3) :-
    X2=[].
{3,7}
noattack (X1, X2, X3) :-
    X2=[X4|X5],
    inegal(X1,X4), plus(X6, X4, X3),
    inegal(X1,X6), plus(X7, X1, X3),
    inegal(X4,X7), X10=[], plus(X9, X3, X10),
    inegal(X8,X9),
    noattack(X1, X5, X8).

{2,0}
inegal (X1, X2) :-
    X1=[], X2=[].
{3,0}
plus (X1, X2, X3) :-
    X1=[], X2=[], X3=[].
{0,0}

```

Ce programme va être testé comme précédemment :

$X_i$ ground dans $\beta_{in}$	$\beta_{out}$	itérations
$X_1$	$(X_1 \wedge \neg X_2) \vee (X_1 \wedge X_2)$	90

La trace pour le premier cas est reproduite en annexe.

Pour ce qui est du temps d'exécution, il est de quelques secondes, la grande partie de ce temps étant consacrée à créer un fichier de trace étendue —debug.iab— long de 411Ko.

## V. Conclusion

---

Ce travail a tenté de décrire la réalisation d'une application spécifique de l'interprétation abstraite. Que faut-il en retenir ?

Tout d'abord, que le sujet est complexe. On manie plusieurs niveaux d'abstractions, on analyse une syntaxe précise, on se fonde sur des sémantiques, on crée de nouveaux domaines... Ce qui est bien complexe, face à la simplicité apparente de l'énoncé "une variable après exécution du programme est-elle ground ?".

Si la question semble triviale, l'univers dans lequel s'exprime les éléments de réponse que l'on peut lui apporter l'est bien moins. Aussi, préalablement à toute ébauche de réalisation, s'agissait-il de "pénétrer" dans cet univers. Cette remarque est bien sûr évidente : "comprendre un problème, c'est déjà y répondre...". Il fallait cependant noter que cette fois, comprendre le problème n'était pas chose triviale. Mais revenons au travail lui-même.

L'originalité de ce travail n'est pas dans la simple implémentation des algorithmes décrits dans [Le Charlier 2], celle-ci était réalisée<sup>1</sup> en parallèle à ce mémoire par P. Van Hentenryck et est décrite dans [Le Charlier 3]. Il s'agit davantage de tester les avantages et défauts de l'utilisation effective d'un domaine abstrait original inspiré de [Cortesi] et de [Marriot].

D'une manière générale, l'implémentation du domaine abstrait telle qu'elle vient d'être décrite pose un problème : le nombre des variables qui peuvent apparaître dans une clause est limité par la taille des entiers utilisés pour représenter une conjonction dans la formulation disjonctive des substitutions abstraites.

Ce problème a une contrepartie évidente : la simplicité<sup>2</sup> relative de représentation des substitutions abstraites et la possibilité d'utiliser l'arsenal d'opérateurs logiques sur les entiers.

La puissance expressive du domaine abstrait n'est pas vraiment comparable à celle du domaine choisi dans [Musumbu]. Certaines propriétés sont exprimables dans un domaine sans l'être dans l'autre, et vice-versa.

Si l'on tente de comparer les résultats obtenus par le programme décrit dans ce travail et celui décrit dans [Le Charlier 3], on voit que ce dernier effectue un

---

<sup>1</sup> 7000 lignes de Pascal

<sup>2</sup> le programme, dont le code est reproduit en annexe, dépasse les 2000 lignes, mais il est futile de vouloir comparer deux programmes sur base de leur taille respective, surtout quand le langage choisi n'est pas le même...

nombre d'itérations plusieurs fois inférieur au nôtre dans l'exemple du programme logique QUEENS. Avant d'attribuer le bénéfice de cette efficacité aux seuls raffinements de la version 4 de l'algorithme générique de [Le Charlier 2], il conviendrait de vérifier que la notion de nombre d'itérations est bien la même dans les deux cas. De même, il faudrait éclaircir la réelle nécessité de recalculer une série de tuple avec  $\perp$  en entrée lorsque deux appels récursifs se font suite.

Quoiqu'il en soit, il serait intéressant d'évaluer le comportement de cet interpréteur sur la diversité des programmes et selon les multiples critères présentés dans [Le Charlier 3] et de vérifier si les prévisions de complexité théorique calculées dans [Le Charlier 2] sont vérifiées.

De plus, il est vraisemblable que, étant donné la simplicité du domaine abstrait choisi ici, une itération correspond à un temps CPU bien inférieur à ce qu'il peut être pour l'autre implémentation.

Ce travail, on le voit, n'est qu'une étape, ni la première, ni la dernière, une participation à un ensemble de recherches effectuées aux FUNDP et visant à perfectionner l'interprétation abstraite des programmes Prolog. La voie esquissée par ce travail devrait pouvoir être prolongée et permettre ainsi de révéler davantage les potentialités offertes tant par l'algorithme abstrait générique que par l'utilisation de formes logiques disjonctives comme domaine abstrait.

## VI. Bibliographie

---

- [Clocksin] : W.F. Clocksin et C.S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.
- [Cortesi] : A. Cortesi, G. Filé et W. Winsborough, *Prop revisited : Propositional Formula as Abstract Domain for Groundness Analysis*, Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, 1991.
- [Le Charlier 1] : B. Le Charlier, K. Musumbu et P. Van Hentenryck, *Efficient and Accurate algorithms for the Abstract Interpretation of Prolog programs*, Research paper RP-90/9, FUNDP, Namur, Août 1990.
- [Le Charlier 2] : B. Le Charlier, K. Musumbu et P. Van Hentenryck, *A Generic Abstract Interpretation Algorithm and its complexity analysis*, Proceedings of the eighth International Conference on Logic Programming (ICLP 91), Paris, Juin 1991.
- [Le Charlier 3] : B. Le Charlier et P. Van Hentenryck, *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, unpublished paper, FUNDP Namur, Août 1991.
- [Lloyd] : J. W. Lloyd, *Foundations of Logic Programming*, 2° edition, Springer-Verlag, Berlin, 1987.
- [Marriott] : K. Marriott et H. Søndergaard, *Notes for a tutorial on abstract interpretation of logic programs*, NACLPL, Cleveland, 1989.
- [Musumbu] : K. Musumbu, *Interprétation Abstraite de programme Prolog*, thèse de doctorat, FUNDP, Namur, Septembre 1990.
- [Shapiro] : L. Sterling et E. Shapiro, *The Art of Prolog : Advanced Programming Techniques*, MIT Press, Cambridge, Ma, 1986.



## VII. Annexes

---

### A. Trace de Append

#### 1. *x3 est ground (étendue)*

```
1.  symbole 2 :
2.  {      append}
3.  Procédure append :
4.  Clause de 3 var en tête et 0 var
dans le corps
5.  but type PREDICAT, append
6.  Nombre de variables dans le
but : 3
7.  100
8.  010
9.  001
10. but type UNIFONCT []
11. Nombre de variables dans le
but : 1
12. 100
13. but type UNIVAR
14. Nombre de variables dans le
but : 2
15. 010
16. 001
17. Clause de 3 var en tête et 3 var
dans le corps
18. but type PREDICAT, append
19. Nombre de variables dans le
but : 3
20. 100000
21. 010000
22. 001000
23. but type UNIFONCT []
24. Nombre de variables dans le
but : 3
25. 100000
26. 000100
27. 000010
28. but type UNIFONCT []
29. Nombre de variables dans le
but : 3
30. 001000
31. 000100
32. 000001
33. but type PREDICAT, append
34. Nombre de variables dans le
but : 3
35. 000010
36. 010000
37. 000001
38. betaIn en entrée de SOLVEGOAL #1
pour append
39. 001
40. 101
41. 011
42. 111
43. suspended en entrée de SOLVEGOAL
44. ... 0 Tuple2 dans la liste
45. use en entrée de SOLVEGOAL
46. ... 0 Tuple2 dans la liste
47. sameSat en entrée de SOLVEGOAL :
1
48. eta en entrée de SOLVEGOAL
49. ... 0 tuples dans la liste
50. def en entrée de SOLVEGOAL
51. ... 0 Tuple2 dans la liste
52. tuple n'appartient pas à
suspended
53. tuple n'appartient pas à def
54. tuple n'appartient pas au dom de
eta
55. eta après EXTEND
56. Tuple3 de symbole #2 : append
57. beta1
58. 001
59. 101
60. 011
61. 111
62. beta2
63. formule vide de nbX=3
64. ... 1 tuples dans la liste
65. suspended dans SOLVEPROCEDURE
66. Tuple2 de symbole #2 : append
67. 001
68. 101
69. 011
70. 111
71. ... 1 Tuple2 dans la liste
72. SOLVECLAUSE 1 in #1
```

73. betaIn à l'entrée de SOLVECLAUSE	134. ... 1 tuples dans la liste
74. 001	135. SOLVECLAUSE 2 in #2
75. 101	136. betaIn à l'entrée de
76. 011	SOLVECLAUSE
77. 111	137. 001
78. betaExt après EXTC	138. 101
79. 001	139. 011
80. 101	140. 111
81. 011	141. betaExt après EXTC
82. 111	142. 001111
83. but à traiter dans solveclause	143. 001110
84. but type UNIFONCT []	144. 001101
85. Nombre de variables dans le	145. 001100
but : 1	146. 001011
86. 100	147. 001010
87. betaAux après RESTRB	148. 001001
88. 0	149. 001000
89. 1	150. 101111
90. gamma	151. 101110
91. 100	152. 101101
92. 000	153. 101100
93. 000	154. 101011
94. betaAux après AIFUNC dans clause	155. 101010
1	156. 101001
95. 1	157. 101000
96. betaExt après EXTB	158. 011111
97. 101	159. 011110
98. 111	160. 011101
99. but à traiter dans solveclause	161. 011100
100. but type UNIVAR	162. 011011
101. Nombre de variables dans le	163. 011010
but : 2	164. 011001
102. 010	165. 011000
103. 001	166. 111111
104. betaAux après RESTRB	167. 111110
105. 01	168. 111101
106. 11	169. 111100
107. gamma	170. 111011
108. 000	171. 111010
109. 100	172. 111001
110. 010	173. 111000
111. AIVAR in	174. but à traiter dans
112. betaAux après AIVAR	solveclause
113. 11	175. but type UNIFONCT []
114. betaExt après EXTB	176. Nombre de variables dans le
115. 111	but : 3
116. RESTRC in	177. 100000
117. betaOut après RESTRC	178. 000100
118. 111	179. 000010
119. betaAux après SOLVECLAUSE	180. betaAux après RESTRB
120. 111	181. 011
121. betaOut avant UNION	182. 010
122. formule vide de nbX=3	183. 001
123. betaOut après UNION dans	184. 000
SOLVEALLCLAUSES	185. 111
124. 111	186. 110
125. eta après UNION dans	187. 101
SOLVEALLCLAUSES	188. 100
126. Tuple3 de symbole #2 : append	189. gamma
127. betal	190. 100000
128. 001	191. 000000
129. 101	192. 000000
130. 011	193. 010000
131. 111	194. 001000
132. beta2	195. 000000
133. formule vide de nbX=3	

196.	betaAux après AIFUNC dans clause 2	256.	010000		
	197.	111	257.	000000	
	198.	010	258.	000000	
	199.	001	259.	100000	
	200.	000	260.	001000	
201.	betaExt après EXTB	261.	appel récursif à SOLVEGOAL		
	202.	001101	262.	betaIn en entrée de SOLVEGOAL #2 pour append	
	203.	001100		263.	001
	204.	001011		264.	101
	205.	001010		265.	011
	206.	001001		266.	111
	207.	001000	267.	suspended en entrée de SOLVEGOAL	
	208.	101111	268.	Tuple2 de symbole #2 : append	
	209.	101110		269.	001
	210.	011101		270.	101
	211.	011100		271.	011
	212.	011011		272.	111
	213.	011010	273.	... 1 Tuple2 dans la liste	
	214.	011001	274.	use en entrée de SOLVEGOAL	
	215.	011000	275.	... 0 Tuple2 dans la liste	
	216.	111111	276.	sameSat en entrée de SOLVEGOAL : 1	
	217.	111110	277.	eta en entrée de SOLVEGOAL	
218.	but à traiter dans solveclause	278.	Tuple3 de symbole #2 : append		
219.	but type UNIFONCT []	279.	beta1		
220.	Nombre de variables dans le but : 3		280.	001	
	221.	001000		281.	101
	222.	000100		282.	011
	223.	000001		283.	111
224.	betaAux après RESTRB	284.	beta2		
	225.	111	285.	formule vide de nbX=3	
	226.	110	286.	... 1 tuples dans la liste	
	227.	101	287.	def en entrée de SOLVEGOAL	
	228.	100	288.	... 0 Tuple2 dans la liste	
229.	gamma	289.	tuple appartient à suspended		
	230.	000000	290.	use après ajout dans SOLVEGOAL	
	231.	000000	291.	Tuple2 de symbole #2 : append	
	232.	100000		292.	001
	233.	010000		293.	101
	234.	000000		294.	011
	235.	001000		295.	111
236.	betaAux après AIFUNC dans clause 2	296.	... 1 Tuple2 dans la liste		
	237.	111	297.	betaAux après appel récursif à SOLVEGOAL	
238.	betaExt après EXTB		298.	001	
	239.	001101		299.	101
	240.	101111		300.	011
	241.	011101		301.	111
	242.	111111	302.	betaExt après EXTB	
243.	but à traiter dans solveclause	303.	formule vide de nbX=6		
244.	but type PREDICAT, append	304.	RESTRC in		
245.	Nombre de variables dans le but : 3	305.	betaOut après RESTRC		
	246.	000010	306.	formule vide de nbX=3	
	247.	010000	307.	betaAux après SOLVECLAUSE	
	248.	000001	308.	formule vide de nbX=3	
249.	betaAux après RESTRB	309.	betaOut avant UNION		
	250.	001		310.	111
	251.	101	311.	betaOut après UNION dans SOLVEALLCLAUSES	
	252.	011		312.	111
	253.	111	313.	eta après UNION dans SOLVEALLCLAUSES	
254.	gamma	314.	Tuple3 de symbole #2 : append		
	255.	000000			

315.       beta1  
      316. 001  
      317. 101  
      318. 011  
      319. 111  
320.       beta2  
321.       formule vide de nbX=3  
322.       ... 1 tuples dans la liste  
323.       ADJUST in  
324.       eta après ADJUST dans  
SOLVEALLCLAUSES  
325.       Tuple3 de symbole #2 : append  
326.       beta1  
      327. 001  
      328. 101  
      329. 011  
      330. 111  
331.       beta2  
      332. 111  
333.       ... 1 tuples dans la liste  
334.       use après SOLVEALLCLAUSES  
dans SOLVEPROCEDURE  
335.       Tuple2 de symbole #2 : append  
      336. 001  
      337. 101  
      338. 011  
      339. 111  
340.       ... 1 Tuple2 dans la liste  
341.       SOLVECLAUSE 1 in #1  
342.       betaIn à l'entrée de  
SOLVECLAUSE  
      343. 001  
      344. 101  
      345. 011  
      346. 111  
347.       betaExt après EXTC  
      348. 001  
      349. 101  
      350. 011  
      351. 111  
352.       but à traiter dans  
solveclause  
353.       but type UNIFONCT []  
354.       Nombre de variables dans le  
but : 1  
      355. 100  
356.       betaAux après RESTRB  
      357. 0  
      358. 1  
359.       gamma  
      360. 100  
      361. 000  
      362. 000  
363.       betaAux après AIFUNC dans  
clause 1  
      364. 1  
365.       betaExt après EXTB  
      366. 101  
      367. 111  
368.       but à traiter dans  
solveclause  
369.       but type UNIVAR  
370.       Nombre de variables dans le  
but : 2  
      371. 010  
      372. 001  
373.       betaAux après RESTRB  
      374. 01  
      375. 11  
376.       gamma  
      377. 000  
      378. 100  
      379. 010  
380.       AIVAR in  
381.       betaAux après AIVAR  
      382. 11  
383.       betaExt après EXTB  
      384. 111  
385.       RESTRC in  
386.       betaOut après RESTRC  
      387. 111  
388.       betaAux après SOLVECLAUSE  
      389. 111  
390.       betaOut avant UNION  
391.       formule vide de nbX=3  
392.       betaOut après UNION dans  
SOLVEALLCLAUSES  
      393. 111  
394.       eta après UNION dans  
SOLVEALLCLAUSES  
395.       Tuple3 de symbole #2 : append  
396.       beta1  
      397. 001  
      398. 101  
      399. 011  
      400. 111  
401.       beta2  
      402. 111  
403.       ... 1 tuples dans la liste  
404.       SOLVECLAUSE 2 in #2  
405.       betaIn à l'entrée de  
SOLVECLAUSE  
      406. 001  
      407. 101  
      408. 011  
      409. 111  
410.       betaExt après EXTC  
      411. 001111  
      412. 001110  
      413. 001101  
      414. 001100  
      415. 001011  
      416. 001010  
      417. 001001  
      418. 001000  
      419. 101111  
      420. 101110  
      421. 101101  
      422. 101100  
      423. 101011  
      424. 101010  
      425. 101001  
      426. 101000  
      427. 011111  
      428. 011110  
      429. 011101  
      430. 011100  
      431. 011011  
      432. 011010  
      433. 011001  
      434. 011000  
      435. 111111

436. 111110  
437. 111101  
438. 111100  
439. 111011  
440. 111010  
441. 111001  
442. 111000  
443. but à traiter dans  
solveclause  
444. but type UNIFONCT []  
445. Nombre de variables dans le  
but : 3  
446. 100000  
447. 000100  
448. 000010  
449. betaAux après RESTRB  
450. 011  
451. 010  
452. 001  
453. 000  
454. 111  
455. 110  
456. 101  
457. 100  
458. gamma  
459. 100000  
460. 000000  
461. 000000  
462. 010000  
463. 001000  
464. 000000  
465. betaAux après AIFUNC dans  
clause 2  
466. 111  
467. 010  
468. 001  
469. 000  
470. betaExt après EXTB  
471. 001101  
472. 001100  
473. 001011  
474. 001010  
475. 001001  
476. 001000  
477. 101111  
478. 101110  
479. 011101  
480. 011100  
481. 011011  
482. 011010  
483. 011001  
484. 011000  
485. 111111  
486. 111110  
487. but à traiter dans  
solveclause  
488. but type UNIFONCT []  
489. Nombre de variables dans le  
but : 3  
490. 001000  
491. 000100  
492. 000001  
493. betaAux après RESTRB  
494. 111  
495. 110  
496. 101  
497. 100  
498. gamma  
499. 000000  
500. 000000  
501. 100000  
502. 010000  
503. 000000  
504. 001000  
505. betaAux après AIFUNC dans  
clause 2  
506. 111  
507. betaExt après EXTB  
508. 001101  
509. 101111  
510. 011101  
511. 111111  
512. but à traiter dans  
solveclause  
513. but type PREDICAT, append  
514. Nombre de variables dans le  
but : 3  
515. 000010  
516. 010000  
517. 000001  
518. betaAux après RESTRB  
519. 001  
520. 101  
521. 011  
522. 111  
523. gamma  
524. 000000  
525. 010000  
526. 000000  
527. 000000  
528. 100000  
529. 001000  
530. appel récursif à SOLVEGOAL  
531. betaIn en entrée de SOLVEGOAL  
#3 pour append  
532. 001  
533. 101  
534. 011  
535. 111  
536. suspended en entrée de  
SOLVEGOAL  
537. Tuple2 de symbole #2 : append  
538. 001  
539. 101  
540. 011  
541. 111  
542. ... 1 Tuple2 dans la liste  
543. use en entrée de SOLVEGOAL  
544. ... 0 Tuple2 dans la liste  
545. sameSat en entrée de  
SOLVEGOAL : 1  
546. eta en entrée de SOLVEGOAL  
547. Tuple3 de symbole #2 : append  
548. beta1  
549. 001  
550. 101  
551. 011  
552. 111  
553. beta2  
554. 111  
555. ... 1 tuples dans la liste  
556. def en entrée de SOLVEGOAL

```

557. ... 0 Tuple2 dans la liste
558. tuple appartient à suspended
559. use après ajout dans
SOLVEGOAL
560. Tuple2 de symbole #2 : append
    561. 001
    562. 101
    563. 011
    564. 111
565. ... 1 Tuple2 dans la liste
566. betaAux après appel récursif
à SOLVEGOAL
    567. 001
    568. 101
    569. 011
    570. 111
571. betaExt après EXTB
    572. 111111
573. RESTRC in
574. betaOut après RESTRC
    575. 111
576. betaAux après SOLVECLAUSE
    577. 111
578. betaOut avant UNION
    579. 111
580. betaOut après UNION dans
SOLVEALLCLAUSES
    581. 111
582. eta après UNION dans
SOLVEALLCLAUSES
583. Tuple3 de symbole #2 : append
584. beta1
    585. 001
    586. 101
    587. 011
    588. 111
589. beta2
    590. 111
591. ... 1 tuples dans la liste
592. use après SOLVEALLCLAUSES
dans SOLVEPROCEDURE
593. Tuple2 de symbole #2 : append
    594. 001
    595. 101
    596. 011
    597. 111
598. ... 1 Tuple2 dans la liste
599. use après avoir enlevé le
tuple dans SOLVEPROCEDURE
600. ... 0 Tuple2 dans la liste
601. def après ajout dans
SOLVEGOAL
602. Tuple2 de symbole #2 : append
    603. 001
    604. 101
    605. 011
    606. 111
607. ... 1 Tuple2 dans la liste
608. ... and the winner is ...
    609. 111

```

## 2. x1 ground

```

lecture de betaIn
  (X1^¬X2^¬X3) | (X1^X2^¬X3) | (X1^¬X2^X3) | (X1^X2^X3)
SOLVEGOAL in #1 pour append
  (X1^¬X2^¬X3) | (X1^X2^¬X3) | (X1^¬X2^X3) | (X1^X2^X3)
EXTEND #1 pour append
  (X1^¬X2^¬X3) | (X1^X2^¬X3) | (X1^¬X2^X3) | (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #1
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^¬X2^¬X3) | (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
  SOLVEGOAL in #2 pour append
    (X1^¬X2^¬X3) | (X1^X2^¬X3) | (X1^¬X2^X3) | (X1^X2^X3)
  SOLVEGOAL out #2 pour append
    formule vide de nbX= 3
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
    ADJUST #1
  SOLVEALLCLAUSES out #1
    (X1^¬X2^¬X3) | (X1^X2^X3)
  SOLVEALLCLAUSES in #2
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^¬X2^¬X3) | (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2

```

```

AIFUNC dans clause 2
SOLVEGOAL in #3 pour append
  (X1^¬X2^¬X3) | (X1^X2^¬X3) | (X1^¬X2^X3) | (X1^X2^X3)
SOLVEGOAL out #3 pour append
  (X1^¬X2^¬X3) | (X1^X2^X3)
SOLVECLAUSE 2 out #2
  (X1^¬X2^¬X3) | (X1^X2^X3)
SOLVEALLCLAUSES out #2
  (X1^¬X2^¬X3) | (X1^X2^X3)
SOLVEPROCEDURE out
SOLVEGOAL out #1 pour append
eta en sortie
  Tuple3 de symbole #2 : append
    beta1
    1 100
    3 110
    5 101
    7 111
    beta2
    1 100
    7 111
  ... 1 tuples dans la liste

```

```

... and the winner is ...
  (X1^¬X2^¬X3) | (X1^X2^X3)

```

### 3. *x2 est ground*

```

lecture de betaIn
  (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL in #1 pour append
  (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
EXTEND #1 pour append
  (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #1
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
      SOLVECLAUSE 1 out #1
      (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
  SOLVEGOAL in #2 pour append
    (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
  SOLVEGOAL out #2 pour append
    formule vide de nbX= 3
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
    ADJUST #1
  SOLVEALLCLAUSES out #1
    (X1^X2^X3)
  SOLVEALLCLAUSES in #2
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
      SOLVECLAUSE 1 out #1
      (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
  SOLVEGOAL in #3 pour append
    (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
  SOLVEGOAL out #3 pour append
    (X1^X2^X3)
  SOLVECLAUSE 2 out #2

```

```

      (¬X1^X2^¬X3) | (X1^X2^X3)
ADJUST #2
SOLVEALLCLAUSES out #2
      (¬X1^X2^¬X3) | (X1^X2^X3)
SOLVEALLCLAUSES in #3
  SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    AIVAR dans clause 1
  SOLVECLAUSE 1 out #1
    (X1^X2^X3)
  SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #4 pour append
      (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #4 pour append
      (¬X1^X2^¬X3) | (X1^X2^X3)
  SOLVECLAUSE 2 out #2
    (¬X1^X2^¬X3) | (X1^X2^X3)
  SOLVEALLCLAUSES out #3
    (¬X1^X2^¬X3) | (X1^X2^X3)
  SOLVEPROCEDURE out
SOLVEGOAL out #1 pour append
eta en sortie
  Tuple3 de symbole #2 : append
    beta1
    2 010
    3 110
    6 011
    7 111
    beta2
    2 010
    7 111
  ... 1 tuples dans la liste

... and the winner is ...
      (¬X1^X2^¬X3) | (X1^X2^X3)

```

#### 4. *x1 et x2 sont ground*

```

lecture de betaIn
      (X1^X2^¬X3) | (X1^X2^X3)
SOLVEGOAL in #1 pour append
      (X1^X2^¬X3) | (X1^X2^X3)
EXTEND #1 pour append
      (X1^X2^¬X3) | (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #1
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
  SOLVEGOAL in #2 pour append
    (X1^X2^¬X3) | (X1^X2^X3)
  SOLVEGOAL out #2 pour append
    formule vide de nbX= 3
  SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
  ADJUST #1
  SOLVEALLCLAUSES out #1
    (X1^X2^X3)
  SOLVEALLCLAUSES in #2
    SOLVECLAUSE 1 in #1

```



```

    AIFUNC dans clause 1
    AIVAR dans clause 1
    SOLVECLAUSE 1 out #1
    (X1^X2^X3)
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
    SOLVEGOAL in #3 pour append
    (X1^X2^¬X3) | (X1^X2^X3)
    SOLVEGOAL out #3 pour append
    (X1^X2^X3)
    SOLVECLAUSE 2 out #2
    (X1^X2^X3)
    SOLVEALLCLAUSES out #2
    (X1^X2^X3)
    SOLVEPROCEDURE out
    SOLVEGOAL out #1 pour append
    eta en sortie
    Tuple3 de symbole #2 : append
    beta1
    3 110
    7 111
    beta2
    7 111
    ... 1 tuples dans la liste

```

... and the winner is ...  
 (X1^X2^X3)

### 5. $x_1, x_2$ et $x_3$ sont ground

```

lecture de betaIn
  (X1^X2^X3)
SOLVEGOAL in #1 pour append
  (X1^X2^X3)
EXTEND #1 pour append
  (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #1
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
      SOLVECLAUSE 1 out #1
      (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
  SOLVEGOAL in #2 pour append
    (X1^X2^X3)
  SOLVEGOAL out #2 pour append
    formule vide de nbX= 3
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
    ADJUST #1
  SOLVEALLCLAUSES out #1
    (X1^X2^X3)
  SOLVEALLCLAUSES in #2
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
      SOLVECLAUSE 1 out #1
      (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
  SOLVEGOAL in #3 pour append
    (X1^X2^X3)

```

```

SOLVEGOAL out #3 pour append
  (X1^X2^X3)
SOLVECLAUSE 2 out #2
  (X1^X2^X3)
SOLVEALLCLAUSES out #2
  (X1^X2^X3)
SOLVEPROCEDURE out
SOLVEGOAL out #1 pour append
eta en sortie
  Tuple3 de symbole #2 : append
    beta1
    7 111
    beta2
    7 111
  ... 1 tuples dans la liste

```

```

... and the winner is ...
  (X1^X2^X3)

```

## 6. rien n'est ground

```

lecture de betaIn
  (¬X1^¬X2^¬X3) | (X1^¬X2^¬X3) | (¬X1^X2^¬X3) | (X1^X2^¬X3) |
(¬X1^¬X2^X3) | (X1^¬X2^X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL in #1 pour append
  (¬X1^¬X2^¬X3) | (X1^¬X2^¬X3) | (¬X1^X2^¬X3) | (X1^X2^¬X3) |
(¬X1^¬X2^X3) | (X1^¬X2^X3) | (¬X1^X2^X3) | (X1^X2^X3)
EXTEND #1 pour append
  (¬X1^¬X2^¬X3) | (X1^¬X2^¬X3) | (¬X1^X2^¬X3) | (X1^X2^¬X3) |
(¬X1^¬X2^X3) | (X1^¬X2^X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #1
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIVAR dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^¬X2^¬X3) | (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
SOLVEGOAL in #2 pour append
  (¬X1^¬X2^¬X3) | (X1^¬X2^¬X3) | (¬X1^X2^¬X3) | (X1^X2^¬X3) |
(¬X1^¬X2^X3) | (X1^¬X2^X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #2 pour append
  formule vide de nbX= 3
  SOLVECLAUSE 2 out #2
  formule vide de nbX= 3
  ADJUST #1
SOLVEALLCLAUSES out #1
  (X1^¬X2^¬X3) | (X1^X2^X3)
SOLVEALLCLAUSES in #2
  SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    AIVAR dans clause 1
  SOLVECLAUSE 1 out #1
    (X1^¬X2^¬X3) | (X1^X2^X3)
  SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #3 pour append
  (¬X1^¬X2^¬X3) | (X1^¬X2^¬X3) | (¬X1^X2^¬X3) | (X1^X2^¬X3) |
(¬X1^¬X2^X3) | (X1^¬X2^X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #3 pour append
  (X1^¬X2^¬X3) | (X1^X2^X3)
  SOLVECLAUSE 2 out #2
  (¬X1^¬X2^¬X3) | (X1^¬X2^¬X3) | (¬X1^X2^¬X3) | (X1^X2^X3)
  ADJUST #2

```

```

SOLVEALLCLAUSES out #2
( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) | ( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
SOLVEALLCLAUSES in #3
SOLVECLAUSE 1 in #1
AIFUNC dans clause 1
AIVAR dans clause 1
SOLVECLAUSE 1 out #1
( $X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
SOLVECLAUSE 2 in #2
AIFUNC dans clause 2
AIFUNC dans clause 2
SOLVEGOAL in #4 pour append
( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) | ( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) |
( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) | ( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
SOLVEGOAL out #4 pour append
( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) | ( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
SOLVECLAUSE 2 out #2
( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) | ( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
SOLVEALLCLAUSES out #3
( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) | ( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
SOLVEPROCEDURE out
SOLVEGOAL out #1 pour append
eta en sortie
Tuple3 de symbole #2 : append
beta1
0 000
1 100
2 010
3 110
4 001
5 101
6 011
7 111
beta2
0 000
1 100
2 010
7 111
... 1 tuples dans la liste

... and the winner is ...
( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ ) | ( $\neg X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )

```

## B. Trace de Quicksort

```

lecture de betaIn
( $X1^{\neg} X2$ ) | ( $X1^{\neg} X2$ )
SOLVEGOAL in #1 pour sort
( $X1^{\neg} X2$ ) | ( $X1^{\neg} X2$ )
EXTEND #1 pour sort
( $X1^{\neg} X2$ ) | ( $X1^{\neg} X2$ )
SOLVEPROCEDURE in
SOLVEALLCLAUSES in #1
SOLVECLAUSE 1 in #1
AIFUNC dans clause 1
SOLVEGOAL in #2 pour quicksort
( $X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
EXTEND #2 pour quicksort
( $X1^{\neg} X2^{\neg} X3$ ) | ( $X1^{\neg} X2^{\neg} X3$ )
SOLVEPROCEDURE in
SOLVEALLCLAUSES in #2
SOLVECLAUSE 1 in #1
AIFUNC dans clause 1
AIVAR dans clause 1
SOLVECLAUSE 1 out #1
( $X1^{\neg} X2^{\neg} X3$ )

```

```

SOLVECLAUSE 2 in #2
  AIFUNC dans clause 2
SOLVEGOAL in #3 pour partition
  (X1^X2^¬X3^¬X4) | (X1^X2^X3^¬X4) | (X1^X2^¬X3^X4) | (X1^X2^X3^X4)
EXTEND #3 pour partition
  (X1^X2^¬X3^¬X4) | (X1^X2^X3^¬X4) | (X1^X2^¬X3^X4) | (X1^X2^X3^X4)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #3
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^X2^X3^X4)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
SOLVEGOAL in #4 pour inegal
  (X1^X2)
EXTEND #4 pour inegal
  (X1^X2)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #4
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^X2)
    ADJUST #4
  SOLVEALLCLAUSES out #4
    (X1^X2)
  SOLVEALLCLAUSES in #5
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^X2)
    SOLVEALLCLAUSES out #5
      (X1^X2)
  SOLVEPROCEDURE out
SOLVEGOAL out #4 pour inegal
  (X1^X2)
SOLVEGOAL in #5 pour partition
  (X1^X2^¬X3^¬X4) | (X1^X2^X3^¬X4) | (X1^X2^¬X3^X4) | (X1^X2^X3^X4)
SOLVEGOAL out #5 pour partition
  formule vide de nbX= 4
  SOLVECLAUSE 2 out #2
  formule vide de nbX= 4
  SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #6 pour inegal
  (X1^X2)
SOLVEGOAL out #6 pour inegal
  (X1^X2)
SOLVEGOAL in #7 pour partition
  (X1^X2^¬X3^¬X4) | (X1^X2^X3^¬X4) | (X1^X2^¬X3^X4) | (X1^X2^X3^X4)
SOLVEGOAL out #7 pour partition
  formule vide de nbX= 4
  SOLVECLAUSE 2 out #2
  formule vide de nbX= 4
  ADJUST #3
  SOLVEALLCLAUSES out #3
    (X1^X2^X3^X4)
  SOLVEALLCLAUSES in #6
    SOLVECLAUSE 1 in #1

```

```

    AIFUNC dans clause 1
    AIFUNC dans clause 1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    (X1^X2^X3^X4)
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
    SOLVEGOAL in #8 pour inegal
    (X1^X2)
    SOLVEGOAL out #8 pour inegal
    (X1^X2)
    SOLVEGOAL in #9 pour partition
    (X1^X2^¬X3^¬X4) | (X1^X2^X3^¬X4) | (X1^X2^¬X3^X4) | (X1^X2^X3^X4)
    SOLVEGOAL out #9 pour partition
    (X1^X2^X3^X4)
    SOLVECLAUSE 2 out #2
    (X1^X2^X3^X4)
    SOLVECLAUSE 3 in #3
    AIFUNC dans clause 3
    AIFUNC dans clause 3
    SOLVEGOAL in #10 pour inegal
    (X1^X2)
    SOLVEGOAL out #10 pour inegal
    (X1^X2)
    SOLVEGOAL in #11 pour partition
    (X1^X2^¬X3^¬X4) | (X1^X2^X3^¬X4) | (X1^X2^¬X3^X4) | (X1^X2^X3^X4)
    SOLVEGOAL out #11 pour partition
    (X1^X2^X3^X4)
    SOLVECLAUSE 3 out #3
    (X1^X2^X3^X4)
    SOLVEALLCLAUSES out #6
    (X1^X2^X3^X4)
    SOLVEPROCEDURE out
    SOLVEGOAL out #3 pour partition
    (X1^X2^X3^X4)
    SOLVEGOAL in #12 pour quicksort
    (X1^¬X2^X3) | (X1^X2^X3)
    SOLVEGOAL out #12 pour quicksort
    formule vide de nbX= 3
    AIFUNC dans clause 2
    SOLVEGOAL in #13 pour quicksort
    formule vide de nbX= 3
    EXTEND #13 pour quicksort
    formule vide de nbX= 3
    SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #7
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    AIVAR dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 3
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    SOLVEGOAL in #14 pour partition
    formule vide de nbX= 4
    EXTEND #14 pour partition
    formule vide de nbX= 4
    SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #8
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    AIFUNC dans clause 1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 4

```

```

SOLVECLAUSE 2 in #2
  AIFUNC dans clause 2
  AIFUNC dans clause 2
SOLVEGOAL in #15 pour inegal
  formule vide de nbX= 2
EXTEND #15 pour inegal
  formule vide de nbX= 2
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #9
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      formule vide de nbX= 2
    SOLVEALLCLAUSES out #9
      formule vide de nbX= 2
  SOLVEPROCEDURE out
SOLVEGOAL out #15 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #16 pour partition
  formule vide de nbX= 4
SOLVEGOAL out #16 pour partition
  formule vide de nbX= 4
  SOLVECLAUSE 2 out #2
    formule vide de nbX= 4
  SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #17 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #17 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #18 pour partition
  formule vide de nbX= 4
SOLVEGOAL out #18 pour partition
  formule vide de nbX= 4
  SOLVECLAUSE 2 out #2
    formule vide de nbX= 4
  SOLVEALLCLAUSES out #8
    formule vide de nbX= 4
  SOLVEALLCLAUSES in #10
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      formule vide de nbX= 4
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
SOLVEGOAL in #19 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #19 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #20 pour partition
  formule vide de nbX= 4
SOLVEGOAL out #20 pour partition
  formule vide de nbX= 4
  SOLVECLAUSE 2 out #2
    formule vide de nbX= 4
  SOLVECLAUSE 3 in #3
    AIFUNC dans clause 3
    AIFUNC dans clause 3
SOLVEGOAL in #21 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #21 pour inegal

```

formule vide de nbX= 2  
 SOLVEGOAL in #22 pour partition  
   formule vide de nbX= 4  
 SOLVEGOAL out #22 pour partition  
   formule vide de nbX= 4  
   SOLVECLAUSE 3 out #3  
   formule vide de nbX= 4  
   SOLVEALLCLAUSES out #10  
   formule vide de nbX= 4  
   SOLVEPROCEDURE out  
 SOLVEGOAL out #14 pour partition  
   formule vide de nbX= 4  
 SOLVEGOAL in #23 pour quicksort  
   formule vide de nbX= 3  
 SOLVEGOAL out #23 pour quicksort  
   formule vide de nbX= 3  
   AIFUNC dans clause 2  
 SOLVEGOAL in #24 pour quicksort  
   formule vide de nbX= 3  
 SOLVEGOAL out #24 pour quicksort  
   formule vide de nbX= 3  
   SOLVECLAUSE 2 out #2  
   formule vide de nbX= 3  
   SOLVEALLCLAUSES out #7  
   formule vide de nbX= 3  
   SOLVEALLCLAUSES in #11  
   SOLVECLAUSE 1 in #1  
     AIFUNC dans clause 1  
     AIVAR dans clause 1  
   SOLVECLAUSE 1 out #1  
   formule vide de nbX= 3  
   SOLVECLAUSE 2 in #2  
     AIFUNC dans clause 2  
 SOLVEGOAL in #25 pour partition  
   formule vide de nbX= 4  
 SOLVEGOAL out #25 pour partition  
   formule vide de nbX= 4  
 SOLVEGOAL in #26 pour quicksort  
   formule vide de nbX= 3  
 SOLVEGOAL out #26 pour quicksort  
   formule vide de nbX= 3  
   AIFUNC dans clause 2  
 SOLVEGOAL in #27 pour quicksort  
   formule vide de nbX= 3  
 SOLVEGOAL out #27 pour quicksort  
   formule vide de nbX= 3  
   SOLVECLAUSE 2 out #2  
   formule vide de nbX= 3  
   SOLVEALLCLAUSES out #11  
   formule vide de nbX= 3  
   SOLVEPROCEDURE out  
 SOLVEGOAL out #13 pour quicksort  
   formule vide de nbX= 3  
   SOLVECLAUSE 2 out #2  
   formule vide de nbX= 3  
   ADJUST #2  
   SOLVEALLCLAUSES out #2  
      $(X1^X2^X3)$   
   SOLVEALLCLAUSES in #12  
   SOLVECLAUSE 1 in #1  
     AIFUNC dans clause 1  
     AIVAR dans clause 1  
   SOLVECLAUSE 1 out #1  
      $(X1^X2^X3)$   
   SOLVECLAUSE 2 in #2  
     AIFUNC dans clause 2

```

SOLVEGOAL in #28 pour partition
  (X1^X2^~X3^~X4) | (X1^X2^X3^~X4) | (X1^X2^~X3^X4) | (X1^X2^X3^X4)
SOLVEGOAL out #28 pour partition
  (X1^X2^X3^X4)
SOLVEGOAL in #29 pour quicksort
  (X1^~X2^X3) | (X1^X2^X3)
SOLVEGOAL out #29 pour quicksort
  (X1^X2^X3)
  AIFUNC dans clause 2
SOLVEGOAL in #30 pour quicksort
  (X1^~X2^X3) | (X1^X2^X3)
SOLVEGOAL out #30 pour quicksort
  (X1^X2^X3)
  SOLVECLAUSE 2 out #2
  (X1^X2^X3)
  SOLVEALLCLAUSES out #12
  (X1^X2^X3)
  SOLVEPROCEDURE out
SOLVEGOAL out #2 pour quicksort
  (X1^X2^X3)
  SOLVECLAUSE 1 out #1
  formule vide de nbX= 2
  SOLVEALLCLAUSES out #1
  formule vide de nbX= 2
  SOLVEALLCLAUSES in #13
  SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
SOLVEGOAL in #31 pour quicksort
  (X1^~X2^X3) | (X1^X2^X3)
SOLVEGOAL out #31 pour quicksort
  (X1^X2^X3)
  SOLVECLAUSE 1 out #1
  (X1^X2)
  ADJUST #13
  SOLVEALLCLAUSES out #13
  (X1^X2)
  SOLVEALLCLAUSES in #14
  SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
SOLVEGOAL in #32 pour quicksort
  (X1^~X2^X3) | (X1^X2^X3)
SOLVEGOAL out #32 pour quicksort
  (X1^X2^X3)
  SOLVECLAUSE 1 out #1
  (X1^X2)
  SOLVEALLCLAUSES out #14
  (X1^X2)
  SOLVEPROCEDURE out
SOLVEGOAL out #1 pour sort
eta en sortie
  Tuple3 de symbole #2 : sort
  beta1
  1 10
  3 11
  beta2
  3 11
  Tuple3 de symbole #3 : quicksort
  beta1
  5 101
  7 111
  beta2
  7 111
  Tuple3 de symbole #4 : partition
  beta1
  3 1100
  7 1110

```



```

11 1101
15 1111
  beta2
15 1111
Tuple3 de symbole #5 : inegal
  beta1
  3 11
  beta2
  3 11
Tuple3 de symbole #3 : quicksort
  beta1
formule vide de nbX=3
  beta2
formule vide de nbX=3
Tuple3 de symbole #4 : partition
  beta1
formule vide de nbX=4
  beta2
formule vide de nbX=4
Tuple3 de symbole #5 : inegal
  beta1
formule vide de nbX=2
  beta2
formule vide de nbX=2
... 7 tuples dans la liste

... and the winner is ...
  (X1^X2)

```

## C. Trace de Queens

```

lecture de betaIn
  (X1^¬X2) | (X1^X2)
SOLVEGOAL in #1 pour queens
  (X1^¬X2) | (X1^X2)
EXTEND #1 pour queens
  (X1^¬X2) | (X1^X2)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #1
  SOLVECLAUSE 1 in #1
SOLVEGOAL in #2 pour perm
  (X1^¬X2) | (X1^X2)
EXTEND #2 pour perm
  (X1^¬X2) | (X1^X2)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #2
  SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
  AIFUNC dans clause 1
  SOLVECLAUSE 1 out #1
  (X1^X2)
  SOLVECLAUSE 2 in #2
  AIFUNC dans clause 2
  AIFUNC dans clause 2
  AIFUNC dans clause 2
SOLVEGOAL in #3 pour delete
  (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
EXTEND #3 pour delete
  (¬X1^X2^¬X3) | (X1^X2^¬X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #3
  SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
  SOLVECLAUSE 1 out #1
  (¬X1^X2^X3)
  SOLVECLAUSE 2 in #2

```

```

    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #4 pour delete
    ( $\neg X1^X2^{\neg X3}$ ) | ( $X1^X2^{\neg X3}$ ) | ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
SOLVEGOAL out #4 pour delete
    formule vide de nbX= 3
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
    ADJUST #3
    SOLVEALLCLAUSES out #3
    ( $\neg X1^X2^X3$ )
    SOLVEALLCLAUSES in #4
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    ( $\neg X1^X2^X3$ )
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #5 pour delete
    ( $\neg X1^X2^{\neg X3}$ ) | ( $X1^X2^{\neg X3}$ ) | ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
SOLVEGOAL out #5 pour delete
    ( $\neg X1^X2^X3$ )
    SOLVECLAUSE 2 out #2
    ( $\neg X1^X2^X3$ )
    SOLVEALLCLAUSES out #4
    ( $\neg X1^X2^X3$ )
    SOLVEPROCEDURE out
SOLVEGOAL out #3 pour delete
    ( $\neg X1^X2^X3$ )
SOLVEGOAL in #6 pour perm
    formule vide de nbX= 2
EXTEND #6 pour perm
    formule vide de nbX= 2
SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #5
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 2
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #7 pour delete
    formule vide de nbX= 3
EXTEND #7 pour delete
    formule vide de nbX= 3
SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #6
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 3
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #8 pour delete
    formule vide de nbX= 3
SOLVEGOAL out #8 pour delete
    formule vide de nbX= 3
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
    SOLVEALLCLAUSES out #6
    formule vide de nbX= 3

```

```

SOLVEPROCEDURE out
SOLVEGOAL out #7 pour delete
    formule vide de nbX= 3
SOLVEGOAL in #9 pour perm
    formule vide de nbX= 2
SOLVEGOAL out #9 pour perm
    formule vide de nbX= 2
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 2
    SOLVEALLCLAUSES out #5
    formule vide de nbX= 2
    SOLVEALLCLAUSES in #7
    SOLVECLAUSE 1 in #1
        AIFUNC dans clause 1
        AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 2
    SOLVECLAUSE 2 in #2
        AIFUNC dans clause 2
        AIFUNC dans clause 2
        AIFUNC dans clause 2
SOLVEGOAL in #10 pour delete
    formule vide de nbX= 3
SOLVEGOAL out #10 pour delete
    formule vide de nbX= 3
SOLVEGOAL in #11 pour perm
    formule vide de nbX= 2
SOLVEGOAL out #11 pour perm
    formule vide de nbX= 2
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 2
    SOLVEALLCLAUSES out #7
    formule vide de nbX= 2
    SOLVEPROCEDURE out
SOLVEGOAL out #6 pour perm
    formule vide de nbX= 2
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 2
    ADJUST #2
    SOLVEALLCLAUSES out #2
        (X1^X2)
    SOLVEALLCLAUSES in #8
    SOLVECLAUSE 1 in #1
        AIFUNC dans clause 1
        AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
        (X1^X2)
    SOLVECLAUSE 2 in #2
        AIFUNC dans clause 2
        AIFUNC dans clause 2
        AIFUNC dans clause 2
SOLVEGOAL in #12 pour delete
    ( $\neg X1^X2^{\neg X3}$ ) | ( $X1^X2^{\neg X3}$ ) | ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
SOLVEGOAL out #12 pour delete
    ( $\neg X1^X2^X3$ )
SOLVEGOAL in #13 pour perm
    ( $X1^{\neg X2}$ ) | ( $X1^X2$ )
SOLVEGOAL out #13 pour perm
    ( $X1^X2$ )
    SOLVECLAUSE 2 out #2
    ( $X1^{\neg X2}$ )
    ADJUST #8
    SOLVEALLCLAUSES out #8
    ( $X1^{\neg X2}$ ) | ( $X1^X2$ )
    SOLVEALLCLAUSES in #9
    SOLVECLAUSE 1 in #1

```

```

    AIFUNC dans clause 1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    (X1^X2)
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
    SOLVEGOAL in #14 pour delete
    ( $\neg X1^X2^{\neg X3}$ ) | ( $X1^X2^{\neg X3}$ ) | ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
    SOLVEGOAL out #14 pour delete
    ( $\neg X1^X2^X3$ )
    SOLVEGOAL in #15 pour perm
    ( $X1^{\neg X2}$ ) | ( $X1^X2$ )
    SOLVEGOAL out #15 pour perm
    ( $X1^{\neg X2}$ ) | ( $X1^X2$ )
    SOLVECLAUSE 2 out #2
    ( $X1^{\neg X2}$ )
    SOLVEALLCLAUSES out #9
    ( $X1^{\neg X2}$ ) | ( $X1^X2$ )
    SOLVEPROCEDURE out
    SOLVEGOAL out #2 pour perm
    ( $X1^{\neg X2}$ ) | ( $X1^X2$ )
    SOLVEGOAL in #16 pour safe
    formule vide de nbX= 1
    EXTEND #16 pour safe
    formule vide de nbX= 1
    SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #10
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 1
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
    SOLVEGOAL in #17 pour noattack
    formule vide de nbX= 3
    EXTEND #17 pour noattack
    formule vide de nbX= 3
    SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #11
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 3
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    SOLVEGOAL in #18 pour inegal
    formule vide de nbX= 2
    EXTEND #18 pour inegal
    formule vide de nbX= 2
    SOLVEPROCEDURE in
    SOLVEALLCLAUSES in #12
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 2
    SOLVEALLCLAUSES out #12
    formule vide de nbX= 2
    SOLVEPROCEDURE out
    SOLVEGOAL out #18 pour inegal
    formule vide de nbX= 2
    SOLVEGOAL in #19 pour plus
    formule vide de nbX= 3

```

```

EXTEND #19 pour plus
  formule vide de nbX= 3
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #13
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      formule vide de nbX= 3
  SOLVEALLCLAUSES out #13
    formule vide de nbX= 3
  SOLVEPROCEDURE out
SOLVEGOAL out #19 pour plus
  formule vide de nbX= 3
SOLVEGOAL in #20 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #20 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #21 pour plus
  formule vide de nbX= 3
SOLVEGOAL out #21 pour plus
  formule vide de nbX= 3
SOLVEGOAL in #22 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #22 pour inegal
  formule vide de nbX= 2
  AIFUNC dans clause 2
SOLVEGOAL in #23 pour plus
  formule vide de nbX= 3
SOLVEGOAL out #23 pour plus
  formule vide de nbX= 3
SOLVEGOAL in #24 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #24 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #25 pour noattack
  formule vide de nbX= 3
SOLVEGOAL out #25 pour noattack
  formule vide de nbX= 3
  SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
  SOLVEALLCLAUSES out #11
    formule vide de nbX= 3
  SOLVEALLCLAUSES in #14
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      formule vide de nbX= 3
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
SOLVEGOAL in #26 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #26 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #27 pour plus
  formule vide de nbX= 3
SOLVEGOAL out #27 pour plus
  formule vide de nbX= 3
SOLVEGOAL in #28 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #28 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #29 pour plus
  formule vide de nbX= 3
SOLVEGOAL out #29 pour plus

```

```

    formule vide de nbX= 3
SOLVEGOAL in #30 pour inegal
    formule vide de nbX= 2
SOLVEGOAL out #30 pour inegal
    formule vide de nbX= 2
    AIFUNC dans clause 2
SOLVEGOAL in #31 pour plus
    formule vide de nbX= 3
SOLVEGOAL out #31 pour plus
    formule vide de nbX= 3
SOLVEGOAL in #32 pour inegal
    formule vide de nbX= 2
SOLVEGOAL out #32 pour inegal
    formule vide de nbX= 2
SOLVEGOAL in #33 pour noattack
    formule vide de nbX= 3
SOLVEGOAL out #33 pour noattack
    formule vide de nbX= 3
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
    SOLVEALLCLAUSES out #14
    formule vide de nbX= 3
    SOLVEPROCEDURE out
SOLVEGOAL out #17 pour noattack
    formule vide de nbX= 3
SOLVEGOAL in #34 pour safe
    formule vide de nbX= 1
SOLVEGOAL out #34 pour safe
    formule vide de nbX= 1
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 1
    SOLVEALLCLAUSES out #10
    formule vide de nbX= 1
    SOLVEALLCLAUSES in #15
    SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 1
    SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
    AIFUNC dans clause 2
SOLVEGOAL in #35 pour noattack
    formule vide de nbX= 3
SOLVEGOAL out #35 pour noattack
    formule vide de nbX= 3
SOLVEGOAL in #36 pour safe
    formule vide de nbX= 1
SOLVEGOAL out #36 pour safe
    formule vide de nbX= 1
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 1
    SOLVEALLCLAUSES out #15
    formule vide de nbX= 1
    SOLVEPROCEDURE out
SOLVEGOAL out #16 pour safe
    formule vide de nbX= 1
    SOLVECLAUSE 1 out #1
    formule vide de nbX= 2
    SOLVEALLCLAUSES out #1
    formule vide de nbX= 2
    SOLVEALLCLAUSES in #16
    SOLVECLAUSE 1 in #1
SOLVEGOAL in #37 pour perm
    (X1^¬X2) | (X1^X2)
SOLVEGOAL out #37 pour perm
    (X1^¬X2) | (X1^X2)

```

```

SOLVEGOAL in #38 pour safe
  (¬X1) | (X1)
EXTEND #38 pour safe
  (¬X1) | (X1)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #17
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (X1)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
      AIFUNC dans clause 2
SOLVEGOAL in #39 pour noattack
  (¬X1^¬X2^X3) | (X1^¬X2^X3) | (¬X1^X2^X3) | (X1^X2^X3)
EXTEND #39 pour noattack
  (¬X1^¬X2^X3) | (X1^¬X2^X3) | (¬X1^X2^X3) | (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #18
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (¬X1^X2^X3) | (X1^X2^X3)
    SOLVECLAUSE 2 in #2
      AIFUNC dans clause 2
SOLVEGOAL in #40 pour inegal
  (¬X1^¬X2) | (X1^¬X2) | (¬X1^X2) | (X1^X2)
EXTEND #40 pour inegal
  (¬X1^¬X2) | (X1^¬X2) | (¬X1^X2) | (X1^X2)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #19
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^X2)
    ADJUST #19
  SOLVEALLCLAUSES out #19
    (X1^X2)
  SOLVEALLCLAUSES in #20
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^X2)
  SOLVEALLCLAUSES out #20
    (X1^X2)
  SOLVEPROCEDURE out
SOLVEGOAL out #40 pour inegal
  (X1^X2)
SOLVEGOAL in #41 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
EXTEND #41 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
SOLVEPROCEDURE in
  SOLVEALLCLAUSES in #21
    SOLVECLAUSE 1 in #1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
      AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
      (X1^X2^X3)
    ADJUST #21
  SOLVEALLCLAUSES out #21
    (X1^X2^X3)
  SOLVEALLCLAUSES in #22

```

```

SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
  AIFUNC dans clause 1
  AIFUNC dans clause 1
SOLVECLAUSE 1 out #1
  (X1^X2^X3)
SOLVEALLCLAUSES out #22
  (X1^X2^X3)
SOLVEPROCEDURE out
SOLVEGOAL out #41 pour plus
  (X1^X2^X3)
SOLVEGOAL in #42 pour inegal
  (¬X1^¬X2) | (X1^¬X2) | (¬X1^X2) | (X1^X2)
SOLVEGOAL out #42 pour inegal
  (X1^X2)
SOLVEGOAL in #43 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #43 pour plus
  (X1^X2^X3)
SOLVEGOAL in #44 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #44 pour inegal
  formule vide de nbX= 2
  AIFUNC dans clause 2
SOLVEGOAL in #45 pour plus
  formule vide de nbX= 3
SOLVEGOAL out #45 pour plus
  formule vide de nbX= 3
SOLVEGOAL in #46 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #46 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #47 pour noattack
  formule vide de nbX= 3
SOLVEGOAL out #47 pour noattack
  formule vide de nbX= 3
SOLVECLAUSE 2 out #2
  formule vide de nbX= 3
ADJUST #18
SOLVEALLCLAUSES out #18
  (¬X1^X2^X3) | (X1^X2^X3)
SOLVEALLCLAUSES in #23
SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
SOLVECLAUSE 1 out #1
  (¬X1^X2^X3) | (X1^X2^X3)
SOLVECLAUSE 2 in #2
  AIFUNC dans clause 2
SOLVEGOAL in #48 pour inegal
  (¬X1^¬X2) | (X1^¬X2) | (¬X1^X2) | (X1^X2)
SOLVEGOAL out #48 pour inegal
  (X1^X2)
SOLVEGOAL in #49 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #49 pour plus
  (X1^X2^X3)
SOLVEGOAL in #50 pour inegal
  (X1^X2)
EXTEND #50 pour inegal
  (X1^X2)
SOLVEPROCEDURE in
SOLVEALLCLAUSES in #24
SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
  AIFUNC dans clause 1
SOLVECLAUSE 1 out #1

```



```

(X1^X2)
ADJUST #24
SOLVEALLCLAUSES out #24
(X1^X2)
SOLVEALLCLAUSES in #25
SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
  AIFUNC dans clause 1
SOLVECLAUSE 1 out #1
(X1^X2)
SOLVEALLCLAUSES out #25
(X1^X2)
SOLVEPROCEDURE out
SOLVEGOAL out #50 pour inegal
(X1^X2)
SOLVEGOAL in #51 pour plus
(-X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #51 pour plus
(X1^X2^X3)
SOLVEGOAL in #52 pour inegal
formule vide de nbX= 2
SOLVEGOAL out #52 pour inegal
formule vide de nbX= 2
  AIFUNC dans clause 2
SOLVEGOAL in #53 pour plus
formule vide de nbX= 3
SOLVEGOAL out #53 pour plus
formule vide de nbX= 3
SOLVEGOAL in #54 pour inegal
formule vide de nbX= 2
SOLVEGOAL out #54 pour inegal
formule vide de nbX= 2
SOLVEGOAL in #55 pour noattack
formule vide de nbX= 3
SOLVEGOAL out #55 pour noattack
formule vide de nbX= 3
  SOLVECLAUSE 2 out #2
  formule vide de nbX= 3
SOLVEALLCLAUSES out #23
(-X1^X2^X3) | (X1^X2^X3)
SOLVEALLCLAUSES in #26
SOLVECLAUSE 1 in #1
  AIFUNC dans clause 1
  SOLVECLAUSE 1 out #1
  (-X1^X2^X3) | (X1^X2^X3)
  SOLVECLAUSE 2 in #2
  AIFUNC dans clause 2
SOLVEGOAL in #56 pour inegal
(-X1^-X2) | (X1^-X2) | (-X1^X2) | (X1^X2)
SOLVEGOAL out #56 pour inegal
(X1^X2)
SOLVEGOAL in #57 pour plus
(-X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #57 pour plus
(X1^X2^X3)
SOLVEGOAL in #58 pour inegal
(X1^X2)
SOLVEGOAL out #58 pour inegal
(X1^X2)
SOLVEGOAL in #59 pour plus
(-X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #59 pour plus
(X1^X2^X3)
SOLVEGOAL in #60 pour inegal
formule vide de nbX= 2
SOLVEGOAL out #60 pour inegal

```

```

    formule vide de nbX= 2
      AIFUNC dans clause 2
SOLVEGOAL in #61 pour plus
    formule vide de nbX= 3
SOLVEGOAL out #61 pour plus
    formule vide de nbX= 3
SOLVEGOAL in #62 pour inegal
    formule vide de nbX= 2
SOLVEGOAL out #62 pour inegal
    formule vide de nbX= 2
SOLVEGOAL in #63 pour noattack
    formule vide de nbX= 3
SOLVEGOAL out #63 pour noattack
    formule vide de nbX= 3
      SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
      SOLVEALLCLAUSES out #26
        ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
      SOLVEPROCEDURE out
SOLVEGOAL out #39 pour noattack
    ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
SOLVEGOAL in #64 pour safe
    ( $\neg X1$ ) | ( $X1$ )
SOLVEGOAL out #64 pour safe
    formule vide de nbX= 1
      SOLVECLAUSE 2 out #2
    formule vide de nbX= 1
      ADJUST #17
      SOLVEALLCLAUSES out #17
        ( $X1$ )
      SOLVEALLCLAUSES in #27
        SOLVECLAUSE 1 in #1
          AIFUNC dans clause 1
        SOLVECLAUSE 1 out #1
          ( $X1$ )
        SOLVECLAUSE 2 in #2
          AIFUNC dans clause 2
          AIFUNC dans clause 2
SOLVEGOAL in #65 pour noattack
    ( $\neg X1^{\neg X2^X3}$ ) | ( $X1^{\neg X2^X3}$ ) | ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
SOLVEGOAL out #65 pour noattack
    ( $\neg X1^X2^X3$ ) | ( $X1^X2^X3$ )
SOLVEGOAL in #66 pour safe
    ( $X1$ )
      EXTEND #66 pour safe
        ( $X1$ )
      SOLVEPROCEDURE in
        SOLVEALLCLAUSES in #28
          SOLVECLAUSE 1 in #1
            AIFUNC dans clause 1
          SOLVECLAUSE 1 out #1
            ( $X1$ )
          SOLVECLAUSE 2 in #2
            AIFUNC dans clause 2
            AIFUNC dans clause 2
SOLVEGOAL in #67 pour noattack
    ( $X1^X2^X3$ )
      EXTEND #67 pour noattack
        ( $X1^X2^X3$ )
      SOLVEPROCEDURE in
        SOLVEALLCLAUSES in #29
          SOLVECLAUSE 1 in #1
            AIFUNC dans clause 1
          SOLVECLAUSE 1 out #1
            ( $X1^X2^X3$ )
          SOLVECLAUSE 2 in #2

```

```

      AIFUNC dans clause 2
SOLVEGOAL in #68 pour inegal
  (X1^X2)
SOLVEGOAL out #68 pour inegal
  (X1^X2)
SOLVEGOAL in #69 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #69 pour plus
  (X1^X2^X3)
SOLVEGOAL in #70 pour inegal
  (X1^X2)
SOLVEGOAL out #70 pour inegal
  (X1^X2)
SOLVEGOAL in #71 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
SOLVEGOAL out #71 pour plus
  (X1^X2^X3)
SOLVEGOAL in #72 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #72 pour inegal
  formule vide de nbX= 2
      AIFUNC dans clause 2
SOLVEGOAL in #73 pour plus
  formule vide de nbX= 3
SOLVEGOAL out #73 pour plus
  formule vide de nbX= 3
SOLVEGOAL in #74 pour inegal
  formule vide de nbX= 2
SOLVEGOAL out #74 pour inegal
  formule vide de nbX= 2
SOLVEGOAL in #75 pour noattack
  formule vide de nbX= 3
SOLVEGOAL out #75 pour noattack
  formule vide de nbX= 3
  SOLVECLAUSE 2 out #2
  formule vide de nbX= 3
  ADJUST #29
  SOLVEALLCLAUSES out #29
  (X1^X2^X3)
  SOLVEALLCLAUSES in #30
  SOLVECLAUSE 1 in #1
    AIFUNC dans clause 1
  SOLVECLAUSE 1 out #1
  (X1^X2^X3)
  SOLVECLAUSE 2 in #2
    AIFUNC dans clause 2
  SOLVEGOAL in #76 pour inegal
  (X1^X2)
  SOLVEGOAL out #76 pour inegal
  (X1^X2)
  SOLVEGOAL in #77 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
  SOLVEGOAL out #77 pour plus
  (X1^X2^X3)
  SOLVEGOAL in #78 pour inegal
  (X1^X2)
  SOLVEGOAL out #78 pour inegal
  (X1^X2)
  SOLVEGOAL in #79 pour plus
  (¬X1^X2^X3) | (X1^X2^X3)
  SOLVEGOAL out #79 pour plus
  (X1^X2^X3)
  SOLVEGOAL in #80 pour inegal
  formule vide de nbX= 2
  SOLVEGOAL out #80 pour inegal
  formule vide de nbX= 2

```

```

    AIFUNC dans clause 2
SOLVEGOAL in #81 pour plus
    formule vide de nbX= 3
SOLVEGOAL out #81 pour plus
    formule vide de nbX= 3
SOLVEGOAL in #82 pour inegal
    formule vide de nbX= 2
SOLVEGOAL out #82 pour inegal
    formule vide de nbX= 2
SOLVEGOAL in #83 pour noattack
    formule vide de nbX= 3
SOLVEGOAL out #83 pour noattack
    formule vide de nbX= 3
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 3
    SOLVEALLCLAUSES out #30
    (X1^X2^X3)
    SOLVEPROCEDURE out
SOLVEGOAL out #67 pour noattack
    (X1^X2^X3)
SOLVEGOAL in #84 pour safe
    (X1)
SOLVEGOAL out #84 pour safe
    formule vide de nbX= 1
    SOLVECLAUSE 2 out #2
    formule vide de nbX= 1
    ADJUST #28
    SOLVEALLCLAUSES out #28
    (X1)
    SOLVEALLCLAUSES in #31
    SOLVECLAUSE 1 in #1
        AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    (X1)
    SOLVECLAUSE 2 in #2
        AIFUNC dans clause 2
        AIFUNC dans clause 2
SOLVEGOAL in #85 pour noattack
    (X1^X2^X3)
SOLVEGOAL out #85 pour noattack
    (X1^X2^X3)
SOLVEGOAL in #86 pour safe
    (X1)
SOLVEGOAL out #86 pour safe
    (X1)
    SOLVECLAUSE 2 out #2
    (X1)
    SOLVEALLCLAUSES out #31
    (X1)
    SOLVEPROCEDURE out
SOLVEGOAL out #66 pour safe
    (X1)
    SOLVECLAUSE 2 out #2
    ( $\neg$ X1) | (X1)
    ADJUST #27
    SOLVEALLCLAUSES out #27
    ( $\neg$ X1) | (X1)
    SOLVEALLCLAUSES in #32
    SOLVECLAUSE 1 in #1
        AIFUNC dans clause 1
    SOLVECLAUSE 1 out #1
    (X1)
    SOLVECLAUSE 2 in #2
        AIFUNC dans clause 2
        AIFUNC dans clause 2
SOLVEGOAL in #87 pour noattack

```

```

      ( $\neg X1 \wedge \neg X2 \wedge X3$ ) | ( $X1 \wedge \neg X2 \wedge X3$ ) | ( $\neg X1 \wedge X2 \wedge X3$ ) | ( $X1 \wedge X2 \wedge X3$ )
SOLVEGOAL out #87 pour noattack
      ( $\neg X1 \wedge X2 \wedge X3$ ) | ( $X1 \wedge X2 \wedge X3$ )
SOLVEGOAL in #88 pour safe
      (X1)
SOLVEGOAL out #88 pour safe
      (X1)
      SOLVECLAUSE 2 out #2
      ( $\neg X1$ ) | (X1)
      SOLVEALLCLAUSES out #32
      ( $\neg X1$ ) | (X1)
      SOLVEPROCEDURE out
SOLVEGOAL out #38 pour safe
      ( $\neg X1$ ) | (X1)
      SOLVECLAUSE 1 out #1
      ( $X1 \wedge \neg X2$ ) | ( $X1 \wedge X2$ )
      ADJUST #16
      SOLVEALLCLAUSES out #16
      ( $X1 \wedge \neg X2$ ) | ( $X1 \wedge X2$ )
      SOLVEALLCLAUSES in #33
      SOLVECLAUSE 1 in #1
SOLVEGOAL in #89 pour perm
      ( $X1 \wedge \neg X2$ ) | ( $X1 \wedge X2$ )
SOLVEGOAL out #89 pour perm
      ( $X1 \wedge \neg X2$ ) | ( $X1 \wedge X2$ )
SOLVEGOAL in #90 pour safe
      ( $\neg X1$ ) | (X1)
SOLVEGOAL out #90 pour safe
      ( $\neg X1$ ) | (X1)
      SOLVECLAUSE 1 out #1
      ( $X1 \wedge \neg X2$ ) | ( $X1 \wedge X2$ )
      SOLVEALLCLAUSES out #33
      ( $X1 \wedge \neg X2$ ) | ( $X1 \wedge X2$ )
      SOLVEPROCEDURE out
SOLVEGOAL out #1 pour queens
eta en sortie
  Tuple3 de symbole #2 : queens
    beta1
    1 10
    3 11
    beta2
    1 10
    3 11
  Tuple3 de symbole #3 : perm
    beta1
    1 10
    3 11
    beta2
    1 10
    3 11
  Tuple3 de symbole #5 : delete
    beta1
    2 010
    3 110
    6 011
    7 111
    beta2
    6 011
  Tuple3 de symbole #3 : perm
    beta1
  formule vide de nbX=2
    beta2
  formule vide de nbX=2
  Tuple3 de symbole #5 : delete
    beta1
  formule vide de nbX=3

```

```

    beta2
formule vide de nbX=3
Tuple3 de symbole #4 : safe
    beta1
formule vide de nbX=1
    beta2
formule vide de nbX=1
Tuple3 de symbole #6 : noattack
    beta1
formule vide de nbX=3
    beta2
formule vide de nbX=3
Tuple3 de symbole #7 : inegal
    beta1
formule vide de nbX=2
    beta2
formule vide de nbX=2
Tuple3 de symbole #8 : plus
    beta1
formule vide de nbX=3
    beta2
formule vide de nbX=3
Tuple3 de symbole #4 : safe
    beta1
    0 0
    1 1
    beta2
    0 0
    1 1
Tuple3 de symbole #6 : noattack
    beta1
    4 001
    5 101
    6 011
    7 111
    beta2
    6 011
    7 111
Tuple3 de symbole #7 : inegal
    beta1
    0 00
    1 10
    2 01
    3 11
    beta2
    3 11
Tuple3 de symbole #8 : plus
    beta1
    6 011
    7 111
    beta2
    7 111
Tuple3 de symbole #7 : inegal
    beta1
    3 11
    beta2
    3 11
Tuple3 de symbole #4 : safe
    beta1
    1 1
    beta2
    1 1
Tuple3 de symbole #6 : noattack
    beta1
    7 111
    beta2

```

```

    7 111
    ... 16 tuples dans la liste

... and the winner is ...
    (X1^¬X2) | (X1^X2)

```

## D. Code

Voici pour terminer ce mémoire le programme lui-même écrit en ANSI C. Les déclarations et définitions des structures de données et des fonctions sont groupées en modules.

On remarquera que le code est *encombré* des fonctions devant permettre le stockage des données intermédiaires et de la trace dans les deux fichiers déjà mentionnés.

On remarquera aussi l'utilisation un peu partout de la fonction `assert` qui a permis de déceler un grand nombre d'erreurs et d'incohérences lors de la mise au point du programme : les tests qu'elle effectue au moment de l'exécution peuvent être évités par une option lors de la compilation, ce qui allège le programme objet et accélère l'exécution.

### 1. Programme principal

#### a. fichier `iab.h`

```

#if !defined(IABCHARGE)
#define IABCHARGE

/* INCLUDES */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "listeLineaire.h"

/* CONSTANTES */
#define PUNCH
/* getchar(); */
#define OUT fichierOut
/* fichierOut ou stdout */
#define DEBUG fichierDebug
/* fichierDebug ou stdout */

#define MAXVARIABLES 16 /* nombre maximal de variables par formules */
#define MAXSYMBLES 16 /* taille maximale de la table des symboles */
#define LONGMAXSYMBLE 16 /* taille maximale d'un symbole */

#define TAILLE_BUFFER 1024 /* taille maximale d'un programme */
#define NOM_FICHER_BETAIn "betaIn.iab"
#define NOM_FICHER_ENTREE "programme.iab"
#define NOM_FICHER_SORTIE "résultat.iab"
#define NOM_FICHER_DEBUG "debug.iab"

/* TYPES */
typedef int Cjct;
typedef Cjct *ptrCjct;

typedef Liste Djct;
typedef ptrListe ptrDjct;

typedef struct tFormule{
    int nbXCjct; /* nombre de bits dans les Cjct */
    int nbCjct; /* nombre de Cjct */
    ptrDjct pDjct;

```

```

} Formule;
typedef Formule *ptrFormule;
typedef ptrFormule ptrBeta;

/* K&R : 146, M&B : 270, S : 2 */
typedef enum {UNIVAR, UNIFONCT, PREDICAT} EnumType;

typedef ptrFormule ptrListeVars; /* liste des variables */

typedef struct tBut { /* but */
    EnumType type;
    int iSymbole; /* indice de la table des symboles */
    int nbVarBut; /* nombre de variables dans le but */
    ptrListeVars plVar;
} But;
typedef But *ptrBut;
typedef Liste ListeButs;
typedef ptrListe ptrListeButs;

typedef struct tclause {
    int nbVarTete;
    int nbVarCorps; /* variables supplémentaires */
    ptrListeButs plBut;
} Clause;
typedef Clause *ptrClause;
typedef Liste ListeClauses;
typedef ptrListe ptrListeClauses;

typedef struct tProc{
    ptrListeClauses plClause;
    int iSymbole;
} Proc;
typedef Proc *ptrProc;
typedef ptrListe ptrListeProc;
typedef Liste ListeProc;

typedef ListeProc Prgm;
typedef ptrListeProc ptrPrgm;

typedef struct {
    ptrFormule beta;
    ptrProc p;
} Tuple2;
typedef Tuple2 *ptrTuple2;
typedef Liste ListeTuple2;
typedef ptrListe ptrListeTuple2;

typedef struct {
    ptrFormule beta1;
    ptrProc p;
    ptrFormule beta2;
} Tuple3;
typedef Tuple3 *ptrTuple3;
typedef Liste ListeTuple3;
typedef ptrListe ptrListeTuple3;

typedef struct {
    char symbole[LONGMAXSYMBOLE]; /* symbole */
    ptrProc procedure;
} ElementSymbolique;
typedef ElementSymbolique TableSymboles[MAXSYMBOLE]; /* tableau de
symboles */

/* PROTOTYPES */

ptrProc lireProgramme(ptrPrgm);

```



```

ptrBeta lireBetaIn(void);

void montreBut(ptrBut pb);
void montreClause(ptrClause pc);
void montreProcEDURE(ptrProc p);

#endif

b. fichier iab.c
#include "iab.h"
#include "parser.h"
#include "formulation.h"
#include "primitives.h"

/* VARIABLES GLOBALES */
FILE *fichierIn, *fichierOut, *fichierDebug;
char tampon[TAILLE_BUFFER];
int dernierSymbole= 0;
TableSymboles tablePred;

int main(int argc, char *argv[])
{ /* Interpretation Abstraite, version 1
  Jean-Pierre Nelissen 1991
  */
  ptrBeta betaIn, betaOut;
  ptrProc p;
  ptrPrgm prg;

  /* initialisation */
  fichierIn= fopen(NOM_FICHIER_ENTREE, "r");
  if (OUT!=stdout)
    fichierOut=fopen(NOM_FICHIER_SORTIE, "w");
  if (DEBUG!=stdout)
    fichierDebug=fopen(NOM_FICHIER_DEBUG, "w");
  strcpy(tablePred[++dernierSymbole].symbole, "[]");

  prg= nouvelleListe();
  p= lireProgramme(prg);
  betaIn= lireBetaIn();
  betaOut= SOLVE(betaIn, p);
  fputs("\n... and the winner is ...\n", stdout);
  montreFormuleBin(stdout, betaOut);
  fputs("\n... soit ...\n", stdout);
  montreFormuleLog(stdout, betaOut);
  fputs("\n... and the winner is ...\n", OUT);
  fputs("\n... and the winner is ...\n", DEBUG);
  montreFormuleBin(DEBUG, betaOut);
  montreFormuleLog(OUT, betaOut);
  fclose(fichierIn);
  fclose(fichierOut);
  return 0;
}

ptrProc lireProgramme(ptrPrgm pgr)
{ /* pre:
  post: lit le programme à partir d'un fichier et renvoie
  la 1° procédure qui sera celle à interpréter
  */
  ptrBut pb;
  ptrClause pc;
  ptrProc pp;
  int debut= 1;
  char *suite;
  int j, nbVarTete, nbVarCorps, symboleProc;

  /* lecture du fichier */

```

```

assert( fread(tampon, sizeof(char), TAILLE_BUFFER, fichierIn) != 0);
suite= tampon;

while(1) {
/* lecture du nombre de variables */
assert( (suite= strpbrk(suite,"(") != NULL);
assert( sscanf(suite,"%d,%d",&nbVarTete,&nbVarCorps) ==2);
assert( (suite= strpbrk(suite,")") != NULL);
if ((nbVarTete == 0) && (nbVarCorps == 0))
    break;
suite++;
pc= nouvelleClause(nbVarTete, nbVarCorps);
suite= parseBut(suite, pc);
symboleProc= ((ptrBut)pc->plBut->tete->suisant->cle)->iSymbole;
if( debut || (pp->iSymbole != symboleProc) ) {
    pp= nouvelleProc(symboleProc);
    ajouteApresElement(pgr, pp);
    pgr->index= pgr->index->suisant;
    debut= 0;
}
ajouteApresElement(pp->plClause, pc);
pp->plClause->index= pp->plClause->index->suisant;
assert((suite= strpbrk(suite, "-") != NULL);
while( *(suite= parseBut(suite, pc) ) != '.');
suite++; /* passe le point */
}
assert(dernierSymbole<10);
for(j=2; j <= dernierSymbole;j++) {
    fprintf(DEBUG,"symbole %d : \n {%16.10s}\n",j,tablePred[j].symbole);
    montreProcedure(tablePred[j].procedure);
}
return pgr->tete->suisant->cle;
}

ptrBeta lireBetaIn(void)
{
/* pre : le premier entier du fichier est le nombre des variables
post : renvoie une formule représentant le betaIn
*/
ptrBeta betaIn;
FILE *fichier;
int i, nbX;

fputs("lecture de betaIn\n",OUT);
fichier= fopen(NOM_FICHIER_BETA_IN,"r");
assert( fscanf(fichier,"%d",&nbX) == 1 );
assert( nbX <= MAXVARIABLES);
betaIn= generettCFormule(nbX);
while(fscanf(fichier,"%d",&i)==1) {
    assert( i<=nbX);
    ajouteXiFormule(betaIn, i);
}
fclose(fichier);
montreFormuleLog(OUT,betaIn);
return betaIn;
}

void montreProcedure(ptrProc pp)
{
/* pre : p existe
post : fait apparaître dans DEBUG la procédure p
*/
ptrElement pe;
ptrClause pc;

fprintf(DEBUG,"Procédure %s : \n",tablePred[pp->iSymbole].symbole);
if (listeVide(pp->plClause))
    fprintf(DEBUG,"Procédure vide de clause\n");
}

```

```

else
    for(pe = pp->plClause->tete->suisvant;
        pe!= pp->plClause->fin;
        pe = pe->suisvant)
        montreClause(pe->cle);
}

void montreClause(ptrClause pc)
{
    /* pre : c existe
    post : fait apparaître dans DEBUG la clause c
    */
    ptrElement pe;
    ptrBut pb;

    fprintf(DEBUG,"Clause de %d var en tête et %d var dans le corps\n",
        pc->nbVarTete, pc->nbVarCorps);
    assert(pc->nbVarTete > 0);
    if (listeVide(pc->plBut))
        fprintf(DEBUG,"Clause vide de but\n");
    else
        for(pe = pc->plBut->tete->suisvant;
            pe!= pc->plBut->fin;
            pe = pe->suisvant)
            montreBut(pe->cle);
}

void montreBut(ptrBut pb)
{
    /* pre : b existe
    post : fait apparaître dans DEBUG le but b
    */
    switch(pb->type) {
        case(UNIVAR) :
            fputs("\tbut type UNIVAR\n", DEBUG);
            break;
        case(UNIFONCT) :
            if (pb->iSymbole==1) {
                fputs("\tbut type UNIFONCT []\n",DEBUG);
            } else fputs("\tunifonct inconnu\n", DEBUG);
            break;
        case(PREDICAT) :
            fprintf(DEBUG, "\tbut type PREDICAT, %s\n",tablePred[pb->iSymbole].symbole);
            break;
        default: assert("But de type inconnu"==NULL);
    }
    fprintf(DEBUG,"\t\tNombre de variables dans le but : %d\n", pb->nbVarBut);
    if (pb->nbVarBut>0)
        montreFormuleBin(DEBUG, pb->plVar);
}

ptrBut nouveauBut(int nbVar)
{
    /* pre
    post : declaration d'un but de 0 variables
    */
    ptrBut pb;

    /* allocation mémoire */
    assert((pb= (ptrBut) malloc(sizeof(But)))!=NULL);
    pb->nbVarBut= 0;
    pb->plVar= nouvelleFormule(nbVar);
    return pb;
}

ptrClause nouvelleClause(int nbVarTete, int nbVarCorps)

```

```

/* pre
 post : declaration d'une clause de 0 variables
*/
 ptrClause pc;

/* allocation mémoire */
 assert((pc= (ptrClause) malloc(sizeof(Clause)))!=NULL);
 pc->nbVarTete= nbVarTete;
 pc->nbVarCorps= nbVarCorps;
 pc->plBut= nouvelleListe();
 return pc;
}

ptrProc nouvelleProc(int symbole)
/* pre
 post : declaration d'une procédure de 0 clauses
*/
 ptrProc pp;

/* allocation mémoire */
 assert((pp= (ptrProc) malloc(sizeof(Proc))) != NULL);
 pp->iSymbole= symbole;
 pp->plClause= nouvelleListe();
 tablePred[pp->iSymbole].procedure= pp;
 return pp;
}

```

## 2. Algorithmes d'interprétation abstraite

### a. fichier primitives.h

```

/* CE FICHER CONTIENT LA DECLARATION DES OPERATIONS PRIMITIVES
 D'INTERPRETATION */
#if !defined(PRIMCHARGE)
#define PRIMCHARGE

/* INCLUDES */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include "iab.h"

/* PROTOTYPES */

/* Opérations principales */
ptrBeta SOLVE(ptrBeta betaIn, ptrProc p);
void SOLVEGOAL(ptrBeta betaIn, ptrProc p, ptrListeTuple2 suspended,
 ptrListeTuple2 *pUse, int *pSameSat, ptrListeTuple3 *pEta, ptrListeTuple2
 *pDef);
void SOLVEPROCEDURE(ptrBeta betaIn, ptrProc p, ptrListeTuple2 suspended,
 ptrListeTuple2 *pUse, int *pSameSat, ptrListeTuple3 *pEta, ptrListeTuple2
 *pDef);
void SOLVEALLCLAUSES(ptrBeta betaIn, ptrProc p, ptrListeTuple2 suspended,
 ptrListeTuple2 *pUse, int *pSameSat, ptrListeTuple3 *pEta, ptrListeTuple2
 *pDef);
void SOLVECLAUSE(ptrBeta betaIn, ptrClause pc, ptrListeTuple2 suspended,
 ptrBeta *pBetaOut,
 ptrListeTuple2 *pUse, int *pSameSat, ptrListeTuple3 *pEta, ptrListeTuple2
 *pDef);

/* Opérations supplémentaires */
ptrListeTuple3 EXTEND(ptrBeta, ptrProc p, ptrListeTuple3 eta);
ptrListeTuple3 ADJUST(ptrBeta, ptrProc p, ptrBeta, ptrListeTuple3 eta);
ptrBeta UNION(ptrBeta, ptrBeta);
ptrBeta ETA(ptrBeta beta, ptrProc p, ptrListeTuple3 eta);

```

```

/* Opérations sur le domaine */
ptrFormule EXTC(ptrClause, ptrFormule);
ptrFormule RESTRC(ptrClause pc, ptrFormule);
ptrFormule EXTB(ptrFormule betaExt, ptrFormule betaAux);
ptrFormule RESTRB(ptrBut, ptrFormule beta0);
ptrFormule AIFUNC(ptrFormule, ptrBut);
ptrFormule AIVAR(ptrFormule);
ptrFormule LEXT(ptrFormule, int nbVar);
ptrFormule LRESTR(ptrFormule f, int nbVar);
ptrFormule CHGVAR(ptrFormule beta0);

#endif

    b. fichier primitives.c
#include "primitives.h"
#include "iab.h"
#include "formulation.h"
#include "tuplisation.h"

ptrFormule pGamma;
extern TableSymboles tablePred;
extern FILE *fichierOut, *fichierDebug;
int cptSAC=1, cptC=1, cptSG=1;

ptrBeta SOLVE(ptrBeta betaIn, ptrProc p)
{
/* pre :
post :
*/
int sameSat= 1;      /* vrai */
ptrListeTuplet2 use, def, suspended;
ptrListeTuplet3 eta;
ptrBeta betaOut;

use= nouvelleListe();
eta= nouvelleListe();
def= nouvelleListe();
suspended= nouvelleListe();
SOLVEGOAL(betaIn, p, suspended, &use, &sameSat, &eta, &def);
betaOut= ETA(betaIn, p, eta);
fprintf(OUT, "eta en sortie\n");
montreListeTuplet3(OUT, eta);
return betaOut;
}

void SOLVEGOAL(ptrBeta betaIn, ptrProc p, ptrListeTuplet2 suspended,
ptrListeTuplet2 *pUse, int *pSameSat, ptrListeTuplet3 *pEta,
ptrListeTuplet2 *pDef)
{
/* pre :
post : Le Charlier 64
*/
ptrTuplet2 pt;
ptrListeTuplet2 useAux;
int compteurLocal;

compteurLocal= cptSG++;
fprintf(OUT, "SOLVEGOAL in #%d pour %s\n", compteurLocal, tablePred[p->iSymbole].symbole);
fprintf(DEBUG, "betaIn en entrée de SOLVEGOAL #%d pour %s\n",
compteurLocal, tablePred[p->iSymbole].symbole);
montreFormuleBin(DEBUG, betaIn);
montreFormuleLog(OUT, betaIn);
fputs("suspended en entrée de SOLVEGOAL\n", DEBUG);
montreListeTuplet2(suspended);
fputs("use en entrée de SOLVEGOAL\n", DEBUG);
montreListeTuplet2(*pUse);

```

```

fprintf(DEBUG, "sameSat en entrée de SOLVEGOAL : %d\n", *pSameSat);
fputs("eta en entrée de SOLVEGOAL\n", DEBUG);
montreListeTuple3(DEBUG, *pEta);
fputs("def en entrée de SOLVEGOAL\n", DEBUG);
montreListeTuple2(*pDef);
pt= nouveauTuple2(betaIn, p);
if (appartientTuple2(pt, suspended)) {
    fputs("tuple appartient à suspended\n", DEBUG);
    insereElement(*pUse, pt, compareTuple2);
    fputs("use après ajout dans SOLVEGOAL\n", DEBUG);
    montreListeTuple2(*pUse);
} else {
    fputs("tuple n'appartient pas à suspended\n", DEBUG);
    if (!appartientTuple2(pt, *pDef)) {
        fputs("tuple n'appartient pas à def\n", DEBUG);
        if (!appartientDomTuple3(pt, *pEta)) {
            fputs("tuple n'appartient pas au dom de sat\n", DEBUG);
            *pSameSat= 0;
            fprintf(OUT, "\tEXTEND #d pour %s\n", compteurLocal, tablePred[p-
>iSymbole].symbole);
            montreFormuleLog(OUT, betaIn);
            *pEta= EXTEND(betaIn, p, *pEta);
            fputs("eta après EXTEND\n", DEBUG);
            montreListeTuple3(DEBUG, *pEta);
        }
        SOLVEPROCEDURE(betaIn, p, suspended, &useAux, pSameSat, pEta, pDef);
        if (listeVide(useAux)) {
            ajouteApresElement(*pDef, pt);
            fputs("def après ajout dans SOLVEGOAL\n", DEBUG);
            montreListeTuple2(*pDef);
        } else {
            *pUse= uniListeTuple2(*pUse, useAux);
            fputs("use après union dans SOLVEGOAL\n", DEBUG);
            montreListeTuple2(*pUse);
        }
    }
}
fprintf(OUT, "SOLVEGOAL out #d pour %s\n", compteurLocal, tablePred[p-
>iSymbole].symbole);
}

void SOLVEPROCEDURE(ptrBeta betaIn, ptrProc p, ptrListeTuple2 suspended,
    ptrListeTuple2 *pUse, int *pSameSat, ptrListeTuple3 *pEta, ptrListeTuple2
*pDef)
{
    int sameSatAux;
    ptrTuple2 pt, pt2;
    ptrListeTuple2 suspendedAux;

    fputs("\tSOLVEPROCEDURE in\n", OUT);
    /* mise à jour de suspendedAux : attention copie nécessaire */
    pt= nouveauTuple2(betaIn, p);
    ajouteApresElement(suspended, pt);
    fputs("suspended dans SOLVEPROCEDURE\n", DEBUG);
    montreListeTuple2(suspended);
    do {
        SOLVEALLCLAUSES(betaIn, p, suspended, pUse, &sameSatAux, pEta, pDef);
        fputs("use après SOLVEALLCLAUSES dans SOLVEPROCEDURE\n", DEBUG);
        montreListeTuple2(*pUse);
        *pSameSat= *pSameSat && sameSatAux;
    } while (!sameSatAux);
    pt2= nouveauTuple2(betaIn, p);
    enleveTuple2(*pUse, pt2);
    enleveTuple2(suspended, pt2);
    fputs("use après avoir enlevé le tuple dans SOLVEPROCEDURE\n", DEBUG);
    montreListeTuple2(*pUse);
}

```

```

    fputs("suspended en sortie de SOLVEPROCEDURE\n", DEBUG);
    montreListeTuple2(suspended);
    fputs("\tSOLVEPROCEDURE out\n",OUT);
}

void SOLVEALLCLAUSES(ptrBeta betaIn, ptrProc p, ptrListeTuple2 suspended,
    ptrListeTuple2 *pUse, int *pSameSat, ptrListeTuple3 *pEta, ptrListeTuple2
    *pDef)
{
    ptrBeta betaOut, betaAux;
    ptrElement pe;
    ptrClause pc;
    int compteurLocal=cptSAC++;

    fprintf(OUT, "\t\tSOLVEALLCLAUSES in #%d\n", compteurLocal);
    betaOut= nouvelleFormule(betaIn->nbXCjct);
    *pUse= nouvelleListe();
    *pSameSat= 1; /* True */
    /* pour toute clause */
    for(pe= p->plClause->tete->suisant, cptC=1;
        pe!=p->plClause->fin ;
        pe= pe->suisant) {
        pc= (ptrClause) pe->cle;
        SOLVECLAUSE(betaIn, pc, suspended, &betaAux, pUse, pSameSat, pEta,
pDef);
        fputs("betaAux après SOLVECLAUSE\n",DEBUG);
        montreFormuleBin(DEBUG,betaAux);
        fputs("betaOut avant UNION\n",DEBUG);
        montreFormuleBin(DEBUG,betaOut);
        betaOut= UNION(betaOut, betaAux);
        fputs("betaOut après UNION dans SOLVEALLCLAUSES\n",DEBUG);
        montreFormuleBin(DEBUG, betaOut);
        fputs("eta après UNION dans SOLVEALLCLAUSES\n",DEBUG);
        montreListeTuple3(DEBUG,*pEta);
    }
    if (compareBeta(betaOut, ETA(betaIn, p, *pEta))>0) {
        trieListe(betaOut->pDjct, compareCjct);
        *pEta= ADJUST(betaIn, p, betaOut, *pEta);
        fprintf(OUT, "\t\t\tADJUST #%d\n", compteurLocal);
        fprintf(DEBUG,"eta après ADJUST dans SOLVEALLCLAUSES\n");
        montreListeTuple3(DEBUG,*pEta);
        *pSameSat= 0;
    }
    fprintf(OUT, "\t\tSOLVEALLCLAUSES out #%d\n", compteurLocal);
    montreFormuleLog(OUT,betaOut);
}

void SOLVECLAUSE(ptrBeta betaIn, ptrClause pc, ptrListeTuple2 suspended,
    ptrBeta *pBetaOut,
    ptrListeTuple2 *pUse, int *pSameSat, ptrListeTuple3 *pEta, ptrListeTuple2
    *pDef)
{
    /* pre :
    post :
    */
    ptrProc p;
    ptrBut pb;
    ptrElement pe; /* contient un but dans sa clé */
    int iGamma=1;
    ptrFormule betaExt, betaAux;
    int compteurLocal;

    compteurLocal= cptC++;
    fprintf(OUT, "\t\t\tSOLVECLAUSE %d in #%d\n", compteurLocal, compteurLocal);
    fprintf(DEBUG, "\t\t\tSOLVECLAUSE %d in
    #%d\n", compteurLocal, compteurLocal);
    fputs("betaIn à l'entrée de SOLVECLAUSE\n",DEBUG);

```

```

    montreFormuleBin(DEBUG,betaIn);
    betaExt= EXTC(pc, betaIn);
    fputs("betaExt après EXTC\n",DEBUG);
    montreFormuleBin(DEBUG,betaExt);
/* pour tout but du corps */
    for(pe= pc->plBut->tete->suivant->suivant;
        pe != pc->plBut->fin;
        pe= pe->suivant) {
        fputs("but à traiter dans solveclause\n",DEBUG);
        montreBut((ptrBut) pe->cle);
        betaAux= RESTRB(pe->cle, betaExt);
        fputs("betaAux après RESTRB\n",DEBUG);
        montreFormuleBin(DEBUG,betaAux);
        fputs("gamma\n",DEBUG);
        montreFormuleBin(DEBUG, pGamma);
        pb= ((ptrBut)pe->cle);
        switch(pb->type) {
            case UNIVAR :
                betaAux= AIVAR(betaAux);
                fprintf(OUT,"\t\t\t\tAIVAR dans clause %d\n",compteurLocal);
                fputs("betaAux après AIVAR\n",DEBUG);
                montreFormuleBin(DEBUG,betaAux);
                break;
            case UNIFONCT :
                betaAux= AIFUNC(betaAux, pb);
                fprintf(OUT,"\t\t\t\tAIFUNC dans clause %d\n",compteurLocal);
                fprintf(DEBUG,"betaAux après AIFUNC dans clause
%d\n",compteurLocal);
                montreFormuleBin(DEBUG,betaAux);
                break;
            case PREDICAT :
                fputs("appel récursif à SOLVEGOAL\n",DEBUG);
                fprintf(stderr,"appel récursif à SOLVEGOAL %d\n", cptSG); PUNCH
                p= tablePred[pb->iSymbole].procedure;
                trieListe(betaAux->pDjct, compareCjct);
                SOLVEGOAL(betaAux, p,suspended, pUse, pSameSat, pEta, pDef);
                fputs("betaAux après appel récursif à SOLVEGOAL\n",DEBUG);
                montreFormuleBin(DEBUG, betaAux);
                betaAux=ETA(betaAux, p, *pEta);
                montreFormuleLog(OUT,betaAux);
                break;
        }
        betaExt= EXTB(betaExt, betaAux);
        fputs("betaExt après EXTB\n",DEBUG);
        montreFormuleBin(DEBUG,betaExt);
    }
    *pBetaOut= RESTRC(pc,betaExt);
    fputs("betaOut après RESTRC\n",DEBUG);
    montreFormuleBin(DEBUG,*pBetaOut);
    fprintf(OUT,"\t\t\t\tSOLVECLAUSE %d out #%d\n",compteurLocal,
compteurLocal);
    montreFormuleLog(OUT,*pBetaOut);
}

ptrListeTuple3 EXTEND(ptrBeta beta, ptrProc p, ptrListeTuple3 eta)
{
/* pre :
    post : modificateur de eta
    renvoie l'union de eta et de (beta,p,beta2) où beta2
    est le plus grand des betaOut des tuples trouvés quand on recherche
    dans eta si il existe des tuples3 de même p et de betaIn <= à beta
    sinon c'est bottom, une beta vide
*/
    ptrElement pe; /* contient un tuple3 dans sa clé */
    ptrTuple3 pt, ptMax;
    ptrBeta pbMax;

```



```

/* pbMax contient bottom, soit la liste vide */
pbMax= nouvelleFormule(beta->nbXCjct);
/* pour tout tuple de eta */
for(pe= eta->tete->suivant; pe != eta->fin ;pe= pe->suivant) {
    pt= (ptrTuple3) pe->c1e;
/* les tuples ont même pointeur de procédure */
    if ( (p == pt->p) &&
/* les betaIn des tuples sont <= à beta */
        (compareBeta(beta, pt->beta1) >=0)) {
/* si le betaOut est > au maximum actuel */
        if (compareBeta(pbMax, pt->beta2)>0)
            pbMax= pt->beta2;
        }
    }
    ptMax= nouveauTuple3(beta, p, copieFormule(pbMax));
    insereElement(eta, ptMax, compareDomTuple3);
    return eta;
}

ptrListeTuple3 ADJUST(ptrBeta beta, ptrProc p, ptrBeta betaPrime,
ptrListeTuple3 eta)
{
/* pre:
    post: modificateur de eta
        on remplace le beta2 des tuples de même domaine
        par le max de betaPrime et leur valeur existante
*/
    ptrTuple3 pt;
    ptrElement pe;
    int i;

fputs("ADJUST in\n", DEBUG);
/* pour tout tuple de eta */
for(pe= eta->tete->suivant; pe!=eta->fin; pe=pe->suivant) {
    pt= (ptrTuple3) pe->c1e;
    if ( (p==pt->p) && ((i=compareBeta(beta, pt->beta1))<=0) )
        if (compareBeta(betaPrime, pt->beta2)>0)
            pt->beta2= copieFormule(betaPrime);
    }
    return eta;
}

ptrFormule RESTRB(ptrBut but, ptrFormule beta0)
{
/* pre : beta0 et les var de b ont même nombre de variables
    post : on renvoie une copie de beta0 restreint aux variables
        de b et ont met à jour gamma; beta0 est intacte
*/
    int iX, j;
    ptrElement pcBut, pcBeta0, pcGamma;
    ptrDjct plcBut= but->p1Var->pDjct;
    ptrDjct plcBeta0= beta0->pDjct;
    ptrDjct plcGamma;
    ptrCjct pc, pci;
    ptrFormule beta1;

/* Définition de gamma */
    pGamma= nouvelleFormule(but->p1Var->nbXCjct);
    for(j=1; j <= but->p1Var->nbXCjct; j++)
        ajouteCFormule(pGamma, nouvelleCjct() );
    plcGamma= pGamma->pDjct;
    beta1= nouvelleFormule(but->nbVarBut);
    pc= nouvelleCjct();
/* pour toute Cjct de beta0 */
    for(pcBeta0= plcBeta0->tete->suivant;
        pcBeta0!= plcBeta0->fin ;
        pcBeta0= pcBeta0->suivant) {
        effaceCjct(pc);
    }
}

```

```

/* pour tout bit de la Cjct*/
for(iX= 1,pcGamma= plcGamma->tete->suisvant;
    iX<= beta0->nbXCjct ;
    iX++, pcGamma= pcGamma->suisvant) {
    if (vraiXiCjct(pcBeta0->cle, iX)) {
/* pour tout bit de la liste de var du but */
        for(pcBut= plcBut->tete->suisvant, j=1;
            pcBut!= plcBut->fin ;
            pcBut= pcBut->suisvant, j++) {
/* si les variables qui apparaissent dans beta0, soit iX, sont présentes
dans le but b */
                if (vraiXiCjct(pcBut->cle, iX)) {
                    affirmeXiCjct(pc, j);
                    affirmeXiCjct(pcGamma->cle, j);
                }
            }
        }
    }
    insereCFormule(beta1,pc);
}
return(beta1);
}

ptrFormule EXTB(ptrFormule betaExt, ptrFormule betaAux)
{
/* pre :
post : on renvoie OR des disjonctions des 2 betas
*/
    int iX;
    Cjct d;
    ptrElement pe;
    ptrDjct plcBetaExt= betaExt->pDjct;
    ptrDjct plcGamma= pGamma->pDjct;
    ptrDjct plcBetaAux;
    ptrElement pcGamma, pcBetaExt, pcBetaAux;
    ptrCjct extracteur, extrait, pcExt, pcAux;
    int existant;

    betaAux= CHGVAR(betaAux);
/* construction de l'extracteur */
    extrait= nouvelleCjct();
    extracteur= nouvelleCjct();
/* pour toute Cjct de gamma */
    for(iX= 1,pcGamma= plcGamma->tete->suisvant;
        pcGamma!= plcGamma->fin ;
        iX++, pcGamma= pcGamma->suisvant) {
        if (premierXiCjct(pcGamma->cle,pGamma->nbXCjct) != 0)
            affirmeXiCjct(extracteur, iX);
    }
    plcBetaAux= betaAux->pDjct;
/* pour toute Cjct de betaExt */
    pcBetaExt= plcBetaExt->tete;
    while(pcBetaExt->suisvant != plcBetaExt->fin){
        pcExt= (ptrCjct) pcBetaExt->suisvant->cle;
        existant= 0;
/* on extrait les bits significatifs des cjcts de betaExt */
        *extrait= *extracteur & *pcExt;
/* pour toute Cjct de betaAux */
        for(pcBetaAux= plcBetaAux->tete->suisvant;
            pcBetaAux!= plcBetaAux->fin ;
            pcBetaAux= pcBetaAux->suisvant) {
            pcAux= (ptrCjct) pcBetaAux->cle;
            if (existant= !compareCjct(extrait, pcAux))
                break;
        }
        if (!existant)
            supprimeApresCFormule(betaExt, pcBetaExt);
    }
}

```

```

        else
            pcBetaExt= pcBetaExt->suisvant;
    }
    return betaExt;
}

ptrFormule CHGVAR(ptrFormule beta0)
{ /* pre :
    post : on étend beta0 aux variables de gamma qu'on supprime
           on renvoie betal
*/
    int iX= 0, j;
    ptrElement pcBeta0, pcGamma;
    ptrDjct plcBeta0= beta0->pDjct;
    ptrDjct plcGamma;
    ptrCjct pc;
    ptrFormule betal;

    plcGamma= pGamma->pDjct;
    betal= nouvelleFormule(pGamma->nbXCjct);
    pc= nouvelleCjct(); /* à libérer */
    /* pour toute Cjct de beta0 */
    for(pcBeta0= plcBeta0->tete->suisvant;
        pcBeta0!= plcBeta0->fin ;
        pcBeta0= pcBeta0->suisvant) {
        effaceCjct(pc
    );
    /* pour toute Cjct de gamma */
    for(iX= 1,pcGamma= plcGamma->tete->suisvant;
        pcGamma!= plcGamma->fin ;
        iX++, pcGamma= pcGamma->suisvant) {
        if (j= premierXiCjct(pcGamma->cle, iX)) {
            if (vraiXiCjct(pcBeta0->cle, j)) {
                affirmeXiCjct(pc, iX);
            }
        }
        insereCFormule(betal,pc);
    }
    return(betal);
}

ptrFormule AIVAR(ptrFormule beta0)
{ /* pre :
    post : constructeur de formule
           implémente uni_var, si 01 ou 10, on insère 11
           M II/31
*/
    ptrElement pd;
    ptrCjct pc, pc0;
    ptrBeta pb;

    fputs("AIVAR in\n",DEBUG);
    pb= nouvelleFormule(beta0->nbXCjct);
    /* pour toute Cjct de beta0 */
    for(pd= beta0->pDjct->tete;pd->suisvant != beta0->pDjct->fin ;pd= pd-
>suisvant) {
        pc0= (ptrCjct) pd->suisvant->cle;
    /* si x1 et pas x2 ou x2 et pas x1 */
        if ((vraiXiCjct(pc0,1) && !vraiXiCjct(pc0,2))
            || (vraiXiCjct(pc0,2) && !vraiXiCjct(pc0,1))) {
            pc= nouvelleCjct();
            affirmeXiCjct(pc, 1);
            affirmeXiCjct(pc, 2);
            insereCFormule(pb, pc);
        } else

```

```

        insereCFormule(pb, pc0);
    }
    return(pb);
}

ptrFormule AIFUNC(ptrFormule beta0, ptrBut pb)
{
    /* pre :
    post : implémente uni_fonct, si x1=constante, x1
    */
    ptrDjct plc = beta0->pDjct;
    ptrCjct pc, pc2;
    int i, trouve;

    /* x1 = constante */
    if (pb->nbVarBut == 1) {
        ajouteXiFormule(beta0, 1);
        return(beta0);
    }
    else {
        /* rechercher 111...111 */
        pc = nouvelleCjct();
        for(i=1; i<=beta0->nbXCjct ; i++)
            affirmeXiCjct(pc, i);
        trouve = rechercheCFormule(beta0, pc);
        /* retire tout ce qui commence par 0 */
        retireXiFormule(beta0, 1);
        /* retire un éventuel 011...111 */
        pc2 = nouvelleCjct();
        for(i=2; i<=beta0->nbXCjct ; i++)
            affirmeXiCjct(pc2, i);
        supprimeCFormule(beta0, pc2);
        /* rajoute 111...111 */
        ajouteCFormule(beta0, pc);
        return(beta0);
    }
}

ptrFormule LEXT(ptrFormule pf, int nbVar)
{
    /* pre : nbVar > 0
    post : on ajoute nbVar nouvelles variables libres
    */
    int i;

    for(i=1 ; i <= nbVar ; i++) {
        ajouteXiFormule(pf, (pf->nbXCjct)+1) ;
    }
    return pf;
}

ptrFormule EXTC(ptrClause pc, ptrFormule beta0)
{
    /* pre :
    post : on ajoute les variables libres du corps à celles de la tête
           on crée une copie de beta0
    */
    int n=0;
    ptrBeta betal;

    betal = copieFormule(beta0);
    n = pc->nbVarCorps;
    return (n>0) ? LEXT(betal, n) : betal;
}

ptrFormule RESTRC(ptrClause pcl, ptrFormule beta0)
{
    /* pre :
    post : créateur de beta
    */

```

```

int i;
ptrCjct pc0, pc;
ptrElement pe;
ptrBeta pb;

fputs("RESTRC in\n",DEBUG);
pb= nouvelleFormule(pc1->nbVarTete);
for(pe=beta0->pDjct->tete->suivant;pe!=beta0->pDjct->fin; pe= pe->suivant)
{
    pc0= (ptrCjct)pe->cle;
    pc= nouvelleCjct();
    for(i=1;i<=pc1->nbVarTete;i++)
        if (vraiXiCjct(pc0,i))
            affirmeXiCjct(pc,i);
    insereCFormule(pb, pc);
}
return pb;
}

ptrBeta UNION(ptrBeta beta1, ptrBeta beta2)
{
    return uniFormule(beta1, beta2);
}

ptrBeta ETA(ptrBeta beta, ptrProc p, ptrListeTuple3 eta)
{
    /* pre
    post : renvoie le beta2 du tuple3 dont beta1=beta et p=p
    NULL sinon
    */
    ptrTuple2 pt;
    ptrElement pe;
    ptrBeta pb;
    ptrTuple3 pt3;

    pt= nouveauTuple2(beta, p);
    pe= rechercheElement(eta, pt, compareTuple32);
    if (pe==eta->fin) {
        pb= NULL;
        assert(pb!=NULL);
    } else {
        pt3= (ptrTuple3) pe->cle;
        pb= pt3->beta2;
    }
    return pb;
}

```

### 3. Opérations sur les Tuples

#### a. fichier *tuplisation.h*

```

/* CE FICHIER CONTIENT LA DECLARATION DES METHODES DES OBJETS Tuple2 ET
Tuple3 */
/* INCLUDES */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include "iab.h"

/* PROTOTYPES */

/* Méthodes de Tuple2 */
/* Constructeur */
ptrTuple2 nouveauTuple2(ptrBeta beta, ptrProc p);
/* Observateur */
int compareTuple2(ptrTuple2 pTuple1, ptrTuple2 pTuple2);

```

```

/* Méthodes de ListeTuple2 */
/* Modificateur */
ptrListeTuple2 uniListeTuple2(ptrListeTuple2 plt1, ptrListeTuple2 plt2);
void enleveTuple2(ptrListeTuple2 use, ptrTuple2 tuple);
/* Observateur */
int appartientTuple2(ptrTuple2, ptrListeTuple2 ensemble);
void montreListeTuple2(ptrListeTuple2 plt);

/* Méthodes de Tuple3 */
/* Constructeur */
ptrTuple3 nouveauTuple3(ptrBeta beta1, ptrProc p, ptrBeta beta2);

/* Méthodes de ListeTuple3 */
/* Observateur */
int compareDomTuple3(ptrTuple3 pTuple1, ptrTuple3 pTuple2);
int appartientDomTuple3(ptrTuple2, ptrListeTuple3);
void montreListeTuple3(FILE *, ptrListeTuple3 plt);
ptrFormule extraitBeta2Tuple3(ptrBeta, ptrProc, ptrListeTuple3);

/* Mixte */
int compareTuple32(ptrTuple3 pTuple1, ptrTuple2 pTuple2);

```

### *b. fichier tuplisation.c*

```

/* CE FICHIER CONTIENT LA DEFINITION DES METHODES DES OBJETS Tuple2 ET
Tuple3 */
#include "tuplisation.h"
#include "formulation.h"
extern FILE *fichierOut, *fichierDebug;
extern TableSymboles tablePred;

/*****
/* OBJET: Tuple2 */
/* type Cf iab.h */

/* CONSTRUCTEUR : Tuple2 */

ptrTuple2 nouveauTuple2(ptrBeta beta, ptrProc p)
{ /* pre: les 2 arguments sont définis
  post : le tuple est défini et rempli avec p et beta
  */
  ptrTuple2 pt;

  pt= (ptrTuple2) malloc(sizeof(Tuple2));
  assert(pt != NULL );
  pt->beta= beta;
  pt->p= p;
  return pt;
}

/* OBSERVATEUR de Tuple2 */

int compareTuple2(ptrTuple2 pTuple1, ptrTuple2 pTuple2)
{ /* pre : les deux arguments sont définis
  post : 0 si comparable, 1 sinon
  */
  int memeP= 0, memeBeta= 0;

  memeP= (pTuple1->p == pTuple2->p);
  if (memeP)
    memeBeta= !compareBeta(pTuple1->beta, pTuple2->beta);
  return !(memeP && memeBeta);
}

```

```

/*****
/* OBJET: ListeTuple2 */
/* type Cf iab.h */

/* MODIFICATEUR : ListeTuple2 */

ptrListeTuple2 uniListeTuple2(ptrListeTuple2 plt1, ptrListeTuple2 plt2)
{
  /* pre : plt2 ne sera jamais libéré de la mémoire
  post : la première liste devient la liste qui reprend
  l'union des tuples des deux listes (sans doublons)
  */
  ptrElement pe;

  for(pe= plt2->tete->suisant;pe != plt2->fin;pe= pe->suisant){
    insereElement(plt1, (ptrTuple2)pe->cle, compareTuple2);
  }
  return plt1;
}

void enleveTuple2(ptrListeTuple2 plt, ptrTuple2 pt)
{
  /* pre : plt et pt sont définis
  post : enleve un tuple de la liste si il existe
  */
  ptrElement trouve;

  if ((trouve= rechercheAvantElement(plt, pt, compareTuple2))!= plt->fin) {
    plt->index= trouve;
    supprimeApresElement(plt);
  }
}

/* OBSERVATEUR : ListeTuple2 */

int appartientTuple2(ptrTuple2 pt, ptrListeTuple2 ensemble)
{
  /* pre : pt et ensemble sont définis
  post : renvoie Vrai (1) si pt est un élément de la liste, 0 sinon
  */
  ptrElement pe;

  if ((pe= rechercheElement(ensemble, pt, compareTuple2))
    == ensemble->fin)
    return 0;
  else return 1;
}

void montreListeTuple2(ptrListeTuple2 plt)
{
  /* pre : plt est défini
  post : affiche la liste de tuples dans DEBUG
  */
  ptrElement pe;
  ptrTuple2 pt;
  int i=0;

  for(pe= plt->tete->suisant ;pe != plt->fin ; pe= pe->suisant, i++) {
    pt= (ptrTuple2) pe->cle;
    fprintf(DEBUG, "\tTuple2 de symbole #d : %s\n", pt->p->iSymbole,
tablePred[pt->p->iSymbole].symbole);
    montreFormuleBin(DEBUG,pt->beta);
  }
  fprintf(DEBUG, "\t... %d Tuple2 dans la liste\n", i);
}

/*****
/* OBJET: Tuple3 */

```

```

/* type Cf iab.h */

/* CONSTRUCTEUR de Tuple3 */

ptrTuple3 nouveauTuple3(ptrBeta betal, ptrProc p, ptrBeta beta2)
{
  /* pre: les trois arguments sont déclarés
   post : le tuple est défini et rempli des trois éléments
  */
  ptrTuple3 pt;

  pt= (ptrTuple3) malloc(sizeof(Tuple3));
  assert(pt != NULL );
  pt->betal= betal;
  pt->p= p;
  pt->beta2= beta2;
  return pt;
}

/* OBSERVATEUR de Tuple3 */

int compareDomTuple3(ptrTuple3 pTuple1, ptrTuple3 pTuple2)
{
  /* pre : les deux arguments sont définis
   post : renvoie 0 si identiques
  */
  int memeP= 0, memeBeta= 0;

  memeP= (pTuple1->p == pTuple2->p);
  if (memeP)
    memeBeta= !compareBeta(pTuple1->betal, pTuple2->betal);
  return !(memeP && memeBeta);
}

/*****
/* OBJET: ListeTuple3 */
/* type Cf iab.h */

/* OBSERVATEUR : ListeTuple3 */

void montreListeTuple3(FILE *fichier, ptrListeTuple3 plt)
{
  /* pre : plt est défini
   post : montre plt dans fichier
  */
  ptrElement pe;
  ptrTuple3 pt;
  int i=0;

  for(pe= plt->tete->suivant ; pe != plt->fin ; pe= pe->suivant, i++) {
    pt= (ptrTuple3) pe->cle;
    fprintf(fichier, "\tTuple3 de symbole #%d : %s\n", pt->p->iSymbole,
tablePred[pt->p->iSymbole].symbole);
    fprintf(fichier, "\t\tbetal\n");
    montreFormuleBin(fichier, pt->betal);
    fprintf(fichier, "\t\tbeta2\n");
    montreFormuleBin(fichier, pt->beta2);
  }
  fprintf(fichier, "\t... %d tuples dans la liste\n", i);
}

int appartientDomTuple3(ptrTuple2 pt, ptrListeTuple3 ensemble)
{
  /* pre : les deux arguments sont définis
   post : retourne Vrai (1) si pt apparait dans les deux premiers éléments
   des tuples de l'ensemble, 0 sinon
  */
  return !(rechercheElement(ensemble, pt, compareTuple32)
== ensemble->fin);
}

```



```

}

/* Méthode mixte */

int compareTuple32(ptrTuple3 pTuple1, ptrTuple2 pTuple2)
{
  /* pre : les deux arguments sont définis
  post : renvoie 0 si identiques, 1 sinon
  */
  int memeP= 0, memeBeta= 0;

  memeP= (pTuple1->p == pTuple2->p);
  if (memeP)
    memeBeta=!compareBeta(pTuple1->beta1, pTuple2->beta);
  return !(memeP && memeBeta);
}

```

## 4. Opérations sur les Betas

### a. fichier formulation.h

```

/* CE FICHIER CONTIENT LA DECLARATION DES METHODES DES OBJETS Cjct ET
Formule */
#ifdef FORMCHARGE
#define FORMCHARGE

/* INCLUDES */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include <assert.h>
#include "listeLineaire.h"
#include "iab.h"

/* PROTOTYPES */

/* Méthodes de Cjct */
/* Constructeur */
ptrCjct nouvelleCjct(void);
ptrCjct copieCjct(ptrCjct);
/* Modificateur */
void effaceCjct(ptrCjct);
void affirmeXiCjct(ptrCjct, int i);
void nieXiCjct(ptrCjct, int i);
/* Observateur de Cjct */
int compareCjct(ptrCjct pdonnee1, ptrCjct pdonnee2);
int vraiXiCjct(ptrCjct, int i);
int premierXiCjct(ptrCjct, int nbX);
void montreCjctBin(FILE *, ptrCjct, int nbX);
void montreCjctLog(FILE *, ptrCjct, int nbX);

/* Méthodes de Formule */
/* Constructeur de Formule */
ptrFormule nouvelleFormule(int tailleCjct);
ptrFormule copieFormule(ptrFormule);
ptrFormule generettCFormule(int nbVar);
ptrFormule uniFormule(ptrFormule pf1, ptrFormule pf2);
/* Modificateur de Formule */
void insereCFormule(ptrFormule, ptrCjct);
void ajouteCFormule(ptrFormule, ptrCjct);
void ajouteApresCFormule(ptrFormule, ptrCjct, ptrElement);
void supprimeCFormule(ptrFormule, ptrCjct);
void supprimeApresCFormule(ptrFormule, ptrElement);
void ajouteXiFormule(ptrFormule, int i);
void retireXiFormule(ptrFormule, int i);

```

```

/* Observateur de Formule */
int compareFormule(ptrFormule, ptrFormule);
int compareBeta(ptrBeta beta1, ptrBeta beta2);
int rechercheCFormule(ptrFormule, ptrCjct);
void montreFormuleBin(FILE *, ptrFormule);
void montreFormuleLog(FILE *, ptrFormule);

#endif

b. fichier formulation.c
/* CE FICHER CONTIENT LES METHODES DES OBJETS Cjct ET Formule */

#include "formulation.h"
extern FILE *fichierOut, *fichierDebug;

/*****
/* OBJET: Conjonction */
/* type Cf iab.h */

/* CONSTRUCTEUR : Conjonction */

ptrCjct nouvelleCjct(void)
{
/* Pre :
Post : définition d'une conjonction vide, soit 0
*/
ptrCjct pc;

pc = (ptrCjct) malloc(sizeof(Cjct));
assert(pc != NULL);
*pc = 0;
return pc;
}

ptrCjct copieCjct(ptrCjct pc0)
{
/* Pre : pc0 est défini
Post : declaration d'une conjonction
dont le contenu est celui de c0 qui reste intacte
*/
ptrCjct pcl;

pcl = nouvelleCjct();
*pcl = *pc0;
return pcl;
}

/* MODIFICATEUR : Conjonction */

void effaceCjct(ptrCjct pc)
{
/* pre : pc est défini
post : efface c, soit = 0
*/
*pc = 0;
}

void affirmeXiCjct(ptrCjct pc, int i)
{
/* pre : i <= le nombre de bits significatifs de pc et
pc est défini
post : si le bit représentant xi n'est pas 1, il le devient
on effectue un OU logique entre c0 et 000100
*/
assert(i <= MAXVARIABLES);
*pc = (i > 0) ? (*pc | (1 << (i-1))) : *pc;
}

void nieXiCjct(ptrCjct pc, int i)

```

```

/* pre : i <= le nombre de bits significatifs de pc et pc est défini
   post : si le bit representant xi n'est pas 0, il le devient

   on effectue un ET logique entre c0 et 111011
*/
assert( i <= MAXVARIABLES );
*pc= ( (i>0) ? ( *pc & ~(1<<(i-1))) ) : *pc );
}

/* OBSERVATEUR : Conjonction */

int compareCjct(ptrCjct pc1, ptrCjct pc2)
/* pre : pc1 et pc2 sont définis
   post : >0 si pc1>pc2, <0 si pc1<pc2, =0 sinon
*/
return (*pc1)-(*pc2);
}

int vraiXiCjct(ptrCjct pc, int i)
/* pre : i <= le nombre de bits significatifs de c
   post : vrai si le bit representant xi est a 1, faux sinon

   on effectue un ET logique entre c et 000100
*/
assert( i <= MAXVARIABLES );
return( (*pc) & (1<<(i-1)) );
}

int premierXiCjct(ptrCjct pc, int nbX)
/* pre : nbX est le nombre de bits significatifs dans la conjonction
   post : retourne le plus petit indice d'un Xi présent
         dans les nbX premiers bits de pc, renvoie 0 sinon
*/
Cjct c;
int i= 1;

assert( nbX <= MAXVARIABLES );
c= *pc;
if (c != 0) {
    while ( ((c & 1) == 0) && (i <= nbX) ) {
        i++;
        c>>=1;
    }
    return((i > nbX) ? 0 : i);
}
else return 0;
}

void montreCjctBin(FILE *fichier, ptrCjct pc0, int nbX)
/* pre : nbX est le nombre de bits significatifs dans la conjonction
   pc0 est défini, fichier est défini
   post : affiche la conjonction pc0 et renvoie 110010 dans fichier
         pour tous les bits
*/
int i,c;

assert( nbX <= MAXVARIABLES );
c= *pc0;
fprintf(fichier, "\t%3d\t", c);
for(i = 1; i <= nbX; i++, c>>=1)
    fprintf(fichier, "%d", c & 1);      /* 0 ou 1 */
fprintf(fichier, "\n");
}

void montreCjctLog(FILE *fichier, ptrCjct pc0, int nbX)
/* pre : nbX est le nombre de bits significatifs dans la conjonction

```

```

    pc0 est défini, fichier est défini
    post : affiche la conjonction (Xi ^ ... ^ Xj) dans fichier
           pour les bits à 1
*/
int i,c;
enum { PRIME,PASPRIME } x=PRIME;

assert( nbX <= MAXVARIABLES );
c=*pc0;
fprintf(fichier,"");
for(i= 1;i <= nbX;i++,c)>= 1)
    if (c & 1) {
        fprintf(fichier,"%sX%d", (x==PRIME) ? "" : "^",i);
        if (x == PRIME)
            x = PASPRIME;
    } else {
        fprintf(fichier,"%s~X%d", (x==PRIME) ? "" : "^",i);
        if (x == PRIME)
            x = PASPRIME;
    }
    fprintf(fichier,"");
}

/*****
/* OBJET: Formule */
/* type Cf iab.h */

/* CONSTRUCTEUR : Formule */

ptrFormule nouvelleFormule(int tailleCjct)
{
/* Pre : tailleCjct <= NBVARIABLES
   Post : définition d'une formule vide,l'index pointe sur la première
*/
    ptrFormule pf;
    ptrCjct pc;

    assert( tailleCjct <= MAXVARIABLES );
    pf= (ptrFormule) malloc(sizeof(Formule));
    assert(pf != NULL);
    pf->nbXCjct= tailleCjct;
    pf->nbCjct= 0;
    pf->pDjct= nouvelleListe();
    return pf;
}

ptrFormule generettCFormule(int nbVar)
{
/* pre : f est défini et est vide
   post : génère toutes les combinaisons de nbVar variables
           soit les entiers de 1...(2^nbVar)-1
*/
    Cjct c;
    ptrFormule pf;
    double i;

    assert((int)(i= ldexp(1.0,nbVar)) <= INT_MAX);
    pf= nouvelleFormule(nbVar);
    for(c=0; c<i; c++)
        ajouteCFormule(pf,&c);
    pf->nbCjct= ((int) i);
    return pf;
}

ptrFormule copieFormule(ptrFormule pf)
{
/* pre : pf est défini
   post : crée une copie de pf

```

```

*/
ptrFormule pf2;
ptrElement pe;
ptrCjct pc2;

pf2= nouvelleFormule(pf->nbXCjct);
for(pe= pf->pDjct->tete->suivant;
    pe != pf->pDjct->fin ;pe= pe->suivant) {
    pc2= copieCjct(pe->cle);
    ajouteCFormule(pf2, pc2);
}
return pf2;
}

ptrFormule uniFormule(ptrFormule pf1, ptrFormule pf2)
{/* pre : pf1 et pf2 sont définis et ont des éléments de même taille
   post : crée une nouvelle formule qui reprend
         l'union des conjonctions des deux formules
*/
ptrFormule pf3= NULL;
ptrElement pe;

assert(pf1->nbXCjct == pf2->nbXCjct);
pf3= copieFormule(pf2);
for(pe= pf1->pDjct->tete->suivant;pe!=pf1->pDjct->fin;pe= pe->suivant){
    insereCFormule(pf3, (ptrCjct)pe->cle);
}
return pf3;
}

/* MODIFICATEUR : Formule */

void insereCFormule(ptrFormule pf, ptrCjct pc)
{/* pre : pf est défini et pc est défini
   post : ajoute une copie de la conjonction c à f
         si elle n'y est pas déjà présente, l'index pointe dessus
*/
ptrCjct pcl;

pcl= copieCjct(pc);
if (insereElement(pf->pDjct, pcl, compareCjct))
    pf->nbCjct++;
}

void ajouteCFormule(ptrFormule pf, ptrCjct pc)
{/* pre : pf est défini et pc est défini
   post : ajoute une copie de la conjonction pc
         à la suite de l'index de pf, qui est avancé
*/
ptrCjct pcl;

pcl= copieCjct(pc);
ajouteApresElement(pf->pDjct, pcl);
pf->nbCjct++;
pf->pDjct->index= pf->pDjct->index->suivant;
}

void ajouteApresCFormule(ptrFormule pf, ptrCjct pc, ptrElement pe)
{/* pre : c est une conjonction définie de f qui est défini
   post : ajoute une copie de la conjonction c à f à la suite de c
*/
ptrCjct pcl;

pf->pDjct->index= pe;
pcl= copieCjct(pc);

```

```

    ajouteApresElement(pf->pDjct, pcl);
    pf->nbCjct++;
}

void supprimeCFormule(ptrFormule pf, ptrCjct pc)
{
    /* pre: pf et pc sont définis
    post: supprime de pf une éventuelle copie de pc
    */
    ptrElement pe;

    pe = pf->pDjct->tete;
    pf->pDjct->fin->cle = pc;
    while(compareCjct(pe->suivant->cle, pc))
        pe = pe->suivant;
    if (pe->suivant != pf->pDjct->fin)
        supprimeApresCFormule(pf, pe);
}

void supprimeApresCFormule(ptrFormule pf, ptrElement pe)
{
    /* pre : pe est un élément de f qui est défini
    post : on supprime la conjonction suivant pe dans f
    */
    pf->pDjct->index = pe;
    supprimeApresElement(pf->pDjct);
    pf->nbCjct--;
}

void ajouteXiFormule(ptrFormule pf, int i)
{
    /* pre : i <= nbX+1, pf est défini
    post : on elimine de f les conjonction où  $\neg X_i$ 
    */
    ptrElement pe, pel;
    ptrDjct pd;
    ptrCjct pc;

    pd = pf->pDjct;
    if (i <= pf->nbXCjct) {
        for(pe = pd->tete ; (pe->suivant) != (pd->fin) ; pe = pel) {
            if (vraiXiCjct(pe->suivant->cle, i))
                pel = pe->suivant; /* K&R : 167 */
            else {
                pel = pe;
                supprimeApresCFormule(pf, pe); /* on supprime le suivant de c */
            }
        }
    } else {
        assert(i == (pf->nbXCjct+1));
        pf->nbXCjct++;
        for(pe = pd->tete ; (pe->suivant) != (pd->fin) ; pe = pe->suivant->suivant)
        {
            pc = copieCjct(pe->suivant->cle);
            nieXiCjct(pc, i);
            ajouteApresCFormule(pf, pc, pe);
            affirmeXiCjct(pe->suivant->cle, i);
        }
    }
}

void retireXiFormule(ptrFormule pf, int i)
{
    /* pre : i <= nbX, pf est défini
    post : on elimine de f les conjonction où  $X_i$ 
    */
    ptrElement pe, pel;
    ptrDjct pd;
    ptrCjct pc;

```

```

pd= pf->pDjct;
assert(i<= pf->nbXCjct);
for(pe= pd->tete ; (pe->suivant) != (pd->fin) ; pe= pel) {
  if (!vraiXiCjct(pe->suivant->cle, i))
    pel= pe->suivant; /* K&R : 167 */
  else {
    pel= pe;
    supprimeApresCFormule(pf, pe); /* on supprime le suivant de c */
  }
}
}

/* OBSERVATEUR : Formule */

int compareFormule(ptrFormule pf1, ptrFormule pf2)
{ /* pre : pf1 et pf2 sont triés et définis
  post : 0 si comparable
*/
  int comparable= 1;
  ptrElement pel,pe2;
  ptrDjct pd1,pd2;

  if ((pf1->nbXCjct == pf2->nbXCjct) &&
      (pf1->nbCjct == pf2->nbCjct)) {
    pd1= pf1->pDjct;
    pd2= pf2->pDjct;
    for(pel= pd1->tete->suivant, pe2= pd2->tete->suivant;
        pel != pd1->fin, pe2 != pd2->fin,
        comparable= !compareCjct(pel->cle,pe2->cle);
        pel= pel->suivant, pe2= pe2->suivant);
  }
  return comparable;
}

int compareBeta(ptrBeta beta1, ptrBeta beta2)
{ /* pre : beta1 et beta2 sont déclarés et triés
  ou pointent vers NULL
  post : un beta vide est plus petit que tout
        >0 si beta1>beta2 ou beta1<>vide et beta2= vide
        <0 si beta1<beta2 ou beta1= vide et beta2<>vide
        =0 si beta1=beta2
*/
  ptrElement pel,pe2;
  ptrDjct pd1,pd2;
  enum {EGALE,PREMIERE,SECONDE} betaLong;

  assert(beta1 != NULL);
  assert(beta2 != NULL);
  assert(beta1->nbXCjct == beta2->nbXCjct);
  if (beta1->nbCjct > beta2->nbCjct)
    betaLong= PREMIERE;
  else if (beta1->nbCjct < beta2->nbCjct)
    betaLong= SECONDE;
  else
    betaLong= EGALE;
  pd1= beta1->pDjct;
  pd2= beta2->pDjct;
  pel= pd1->tete->suivant;
  pe2= pd2->tete->suivant;
  while( (pel != pd1->fin) && (pe2 != pd2->fin) ) {
    if ( !compareCjct(pel->cle, pe2->cle) ) {
      pel= pel->suivant; pe2= pe2->suivant;
    } else
      switch(betaLong) {
        case(EGALE): assert("betas incomparable"==0); /* incomparable */
        case(PREMIERE): pel= pel->suivant; break;

```

```

        case(SECONDE):      pe2= pe2->suitant; break;
    }
}
switch(betaLong) {
case(EGALE):
    assert( (pe1==pd1->fin)&&(pe2==pd2->fin));    /* égalité */
    return 0;
case(PREMIERE):
    if (pe2==pd2->fin)
        return 1;
    else
        assert("betas incomparable1"==0);    /* incompatible */
case(SECONDE):
    if (pe1==pd1->fin)
        return -1;
    else
        assert("betas incomparable2"==0);    /* incompatible */
}
}

int rechercheCFormule(ptrFormule pf, ptrCjct pc)
/* pre : pf et pc sont définis
post : 0 si absent, 1 si trouvé
*/
return (rechercheElement(pf->pDjct, pc, compareCjct) != pf->pDjct->fin);
}

void montreFormuleBin(FILE *fichier,ptrFormule pf)
/* pre : pf est défini
post : affiche la formule f
*/
ptrElement pe;
ptrDjct pd;
int i;

pd= pf->pDjct;
if (pf==NULL)
    fputs("\tformule inexistante\n",fichier);
else if (pd->tete->suitant == pd->fin)
    fprintf(fichier,"\tformule vide de nbX=%d\n",pf->nbXCjct);
else {
    for(pe= pd->tete->suitant, i=0 ;
        pe != pd->fin ;
        pe= pe->suitant, i++)
        montreCjctBin(fichier,pe->cle, pf->nbXCjct);
    assert(i==pf->nbCjct);
}
}

void montreFormuleLog(FILE *fichier,ptrFormule pf)
/* pre : pf est défini
post : affiche la formule f dans OUT après 3 \t
*/
ptrElement pe;
ptrDjct pd;
int i;

pd= pf->pDjct;
fputs("\t\t\t",fichier);
if (pf==NULL)
    fputs("formule inexistante\n",fichier);
else if (pd->tete->suitant == pd->fin)
    fprintf(OUT,"formule vide de nbX= %d\n",pf->nbXCjct);
else {
    for(pe= pd->tete->suitant,i=0;pe != pd->fin;pe= pe->suitant, i++) {
        if (i>0)

```



```

        fprintf(fichier," | ");
        montreCjctLog(fichier, pe->cle, pf->nbXCjct);
    }
    assert(i==pf->nbCjct);
    fprintf(fichier,"\n");
}
}

```

## 5. Opérations de parsing

### a. fichier parser.h

```

/* CE FICHIER CONTIENT LA DECLARATION DES FONCTIONS DE PARSING */
/* INCLUDES */
#include "iab.h"

/* PROTOTYPES */

char *parseListeVariable(char *texte, ptrBut);
char *parseSymbole(char *texte, ptrBut);
char *parseUnification(char *texte, ptrBut pb);
char *parseListe(char *texte, ptrBut pb);
char *parseBut(char *texte, ptrClause pc);

ptrBut nouveauBut(int nbVar);
void ajouteVariableBut(int iVariable, ptrBut pb);
ptrClause nouvelleClause(int nbVarTete, int nbVarCorps);
ptrProc nouvelleProc(int symbole);

```

### b. fichier parser.c

```

/* CE FICHIER CONTIENT LA DEFINITION DES FONCTIONS DE PARSING */
#include "iab.h"
#include "parser.h"
#include "formulation.h"

/* VARIABLES GLOBALES */
extern int dernierSymbole;
extern TableSymboles tablePred;

char *parseSymbole(char *texte, ptrBut pb)
{
    /* pre : texte est le tampon-programme, pb est défini
       texte pointe avant ou sur une minuscule
       post : un symbole est lu et rangée dans pb et dans
       tablePred au besoin; on renvoie un pointeur
       vers le caractère suivant le dernier du symbole
    */
    int taille, i;
    char *symbole;
    char caractOk[] = "abcdefghijklmnopqrstuvwxyz_";

    assert((symbole= strpbrk(texte, caractOk))!=NULL);
    /* symbole pointe sur le premier caractère d'un symbole */
    taille= strspn(symbole, caractOk);
    /* taille est la taille du symbole */
    assert(taille <= LONGMAXSYMBOLE);
    for(i=1; (i<=dernierSymbole) &&
        (strncmp(symbole, tablePred[i].symbole, taille)) ;i++);
    if (i>dernierSymbole) {
        strncpy(tablePred[++dernierSymbole].symbole, symbole, taille);
    }
    /* le symbole est présent dans la table en i */
    pb->iSymbole= i;
    return (symbole + taille);
}

char *parseListeVariable(char *texte, ptrBut pb)
{
    /* pre: texte est le tampon-programme, pb est défini

```

```

    texte pointe avant ou sur une (
post : une liste de variables est lue et rangée dans pb
on renvoie un pointeur vers )
*/
char *ouvreParenthese, *fermeParenthese, *x;
int bienLu, d;
ptrCjct pd;

assert((ouvreParenthese= strpbrk(texte,"(") != NULL);
assert((fermeParenthese= strpbrk(ouvreParenthese,")") != NULL);
x= ouvreParenthese;
while (x<fermeParenthese) {
/* avance devant X */
    x++;
    if ((x= strpbrk(x,"X") == NULL) || (x>=fermeParenthese)) {
        break;
    }
}
/* lit Xi */
assert(((bienLu= sscanf(x,"X%d",&d))==1) && (x<fermeParenthese));
ajouteVariableBut(d, pb);
}
return(fermeParenthese);
}

char *parseUnification(char *texte, ptrBut pb)
{
/* pre: texte est le tampon-programme, pb est défini
   texte pointe sur X
   post : un builtIn est lu et rangé dans pb
   on renvoie un pointeur vers ) ou ] ou un chiffre
*/
int cl,cj;
ptrCjct pc;
char *debut, *egal;
char caractOk[] = "abcdefghijklmnopqrstuvwxyz_[X";

assert( sscanf(texte,"X%d",&cl) == 1 );
/* rangement de la première variable de l'unification */
ajouteVariableBut(cl, pb);
assert((egal= strpbrk(texte,"=") != NULL);
assert((debut= strpbrk(egal,caractOk) != NULL);
if (*debut == 'X') {
    pb->type= UNIVAR;
    assert( ( sscanf(debut,"X%d",&cj)) == 1 );
}
/* rangement de la deuxième variable de l'unification */
ajouteVariableBut(cj, pb);
return ++debut;
} else if (*debut == '[') {
    pb->type= UNIFONCT;
    return parseListe(debut,pb);
} else {
    pb->type= UNIFONCT;
    texte= parseSymbole(debut, pb);
    return parseListeVariable(texte, pb);
}
}

char *parseListe(char *texte, ptrBut pb)
{
/* pre: texte est le tampon-programme, pb est défini
   texte pointe sur [
   post : une liste est lue et rangée dans pb
   on renvoie un pointeur vers ) ou ] ou un chiffre
*/
char *fermeCrochet, *x;
char caractOk[] = "abcdefghijklmnopqrstuvwxyz_[X";

pb->iSymbole= 1;

```

```

    *texte= '(';
    assert((fermeCrochet= strpbrk(texte,")") != NULL);
    *fermeCrochet= ')';
    if ((x= strpbrk(texte,"X")) != NULL) && (x<fermeCrochet)) {
        return parseListeVariable(texte,pb);
    } else
        return fermeCrochet;
}

```

```

char *parseBut(char *texte, ptrClause pc)
/* pre: texte est le tampon-programme, pc est défini
   texte pointe avant ou sur le premier car d'un symbole
   post : un but est lu, défini et rangé dans pc
   on renvoie un pointeur vers .
*/

```

```

char *debut, *suite;
ptrBut pb;
char caractOk[] = "abcdefghijklmnopqrstuvwxyz_X.";

assert((debut= strpbrk(texte,caractOk)) != NULL);
if (*debut == '.')
    return debut;
else {
    pb= nouveauBut(pc->nbVarTete + pc->nbVarCorps);
    ajouteApresElement(pc->plBut, pb);
    pc->plBut->index= pc->plBut->index->suivant;
    if (*debut == 'X') {
        suite= parseUnification(debut,pb);
    } else {
        suite= parseSymbole(debut, pb);
        pb->type= PREDICAT;
        suite= parseListeVariable(suite, pb);
    }
    return suite;
}
}

```

```

void ajouteVariableBut(int iVariable, ptrBut pb)
/* pre : la liste de variables est définie dans pb
   iVariable <= MAXVARIABLES
   post : ajoute la variable iVariable à la liste
   de variables du but pb
*/

```

```

ptrCjct pc;

assert(iVariable <= MAXVARIABLES);
assert(pb->nbVarBut++ < MAXVARIABLES);
pc= nouvelleCjct();
affirmeXiCjct(pc, iVariable);
ajouteCFormule(pb->plVar,pc);
}

```

## 6. Listes linéaires

### a. fichier liste linéaire.h

```

/* CE FICHER CONTIENT LA DECLARATION DE L'OBJET LISTE ET DE SES METHODES
*/
#if !defined(LLCHARGE)
#define LLCHARGE

/* INCLUDES */
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>

```

```

/* K&R : 146, M&B : 270, S : 2 */
/* liste lineaire a un champ */

typedef struct tElement *ptrElement; /* pointeur vers element de liste
*/
typedef struct tElement { /* element de liste */
    void *cle;
    ptrElement suivant;
} Element;

typedef struct tListe *ptrListe; /* pointeur vers la liste */
typedef struct tListe { /* liste */
    ptrElement tete;
    ptrElement index; /* indique l'element courant de la liste */
    ptrElement fin;
} Liste;

/* PROTOTYPES */

/* Méthodes de Liste */
/* Constructeur */
ptrListe nouvelleListe(void);
/* Modificateur */
int insereElement(ptrListe liste, void *cle,
    int (*cmp)( void *valeurCle, void *donnee) );
void ajouteApresElement(ptrListe liste, void *cle);
void supprimeApresElement(ptrListe liste);
void trieListe(ptrListe liste, int (*cmp)( void *valeurListe, void
*valeurDonnee) );
/* Observateur */
int listeVide(ptrListe);
ptrElement rechercheElement(ptrListe, void *cle,
    int (*cmp)( void *valeurCle, void *donnee) );
ptrElement rechercheAvantElement(ptrListe liste, void *valeur,
    int (*cmp)( void *valeurListe, void *valeurDonnee) );

#endif

    b. fichier liste linéaire.c
/* CE FICHER CONTIENT LES METHODES DE L'OBJET Liste */

#include "listeLineaire.h"
extern FILE *fichierOut, *fichierDebug;

/*****
/* OBJET: Liste */
/* type Cf listeLineaire.h */

/* CONSTRUCTEUR : Liste */

ptrListe nouvelleListe() {
/* Pre :
    Post : cette fonction définit une liste vide, 2 elements sentinelles
           et un indice, le tout formant la liste chainee vide
           : une tete et une fin; l'indice pointe sur le premier element
S : 196
*/
    ptrListe liste;

/* allocation d'espace pour la liste */
    assert((liste=(Liste *) malloc(sizeof(Liste)))!=NULL);
/* allocation d'espace pour les elements sentinelles de la liste */
    assert(((liste->tete=(Element *) malloc(sizeof(Element)))!=NULL) &&
        ((liste->fin=(Element *) malloc(sizeof(Element)))!=NULL));
    liste->fin->suivant=liste->fin;
    liste->tete->suivant=liste->fin;

```

```

    liste->index=liste->tete;
    return(liste);
}

/* MODIFICATEUR : Liste */

void ajouteApresElement(ptrListe liste, void *valeur)
{
    /* Pre : liste est défini et valeur est une copie de la clé
       index pointe vers un des elements de la liste
       Post : insere un nouvel element qui pointera sur la copie de la clé
       apres celui pointé par l'index; l'index ne bouge pas
       S : 20
    */
    ptrElement element;

    /* allocation de memoire au nouvel element */
    assert((element=(Element *) malloc(sizeof(Element)))!=NULL);
    element->cle= valeur;
    element->suisant= liste->index->suisant;
    liste->index->suisant=element;
}

int insereElement(ptrListe liste, void *valeur,
    int (*cmp)( void *valeurCle, void *donnee) )
{
    /* Pre : la liste est formée d'éléments uniques et
       la valeur est une copie de la clé
       Post : insere un nouvel element en fin de liste s'il n'est
       pas déjà présent auquel cas on renvoie vrai; faux (0) sinon
       l'index pointe vers l'élément
       S : 197
    */
    ptrElement element;
    int insertion;

    /* allocation de memoire au nouvel element */
    if (insertion= ((liste->index= rechercheElement(liste, valeur, cmp))
        == liste->fin)) {
    /* creation d'un nouvel element de fin, l'ancien contient la valeur apres la
    recherche */
        assert((element=(Element *) malloc(sizeof(Element)))!= NULL);
        element->suisant= element;
        liste->fin->suisant= element;
        liste->fin= element;
    }
    return(insertion);
}

void supprimeApresElement(ptrListe liste)
{
    /* Pre : l'index pointe vers un des elements de la liste
       Post : supprime l'element suivant celui designe par l'index
       S : 20
    */
    liste->index->suisant=liste->index->suisant->suisant;
}

void trieListe(ptrListe liste,int (*cmp)( void *valeurListe, void
*valeurDonnee) )
{
    /* Pre :
       Post : trie de manière croissante liste
       S : 98
    */
    ptrElement pei,pej, min;

    for(pei=liste->tete->suisant;pei->suisant!=liste->fin;pei= pei->suisant) {
        min= pei;
        for(pej=pei->suisant;pej!=liste->fin;pej= pej->suisant) {

```

```

        if (cmp(pej->cle,min->cle)<0) {
            min=pej;
        }
    }
    liste->tete->cle= min->cle;
    min->cle= pei->cle;
    pei->cle= liste->tete->cle;
}
}

/* OBSERVATEUR : Liste */

int listeVide(ptrListe pl)
{
    /* pre: pl est défini
    post : 0 si liste non vide, 1 sinon
    */
    return (pl->tete->suivant == pl->fin);
}

ptrElement rechercheElement(ptrListe liste, void *valeur,
    int (*cmp)( void *valeurListe, void *valeurDonnee) )
{
    /* Pre : cmp renvoie 0 si les valeurs sont comparables
    Post : on renvoie l'adresse du 1° élément qui correspond
    à la valeur et recherché à partir du début;
    en cas d'absence, on renvoie l'adresse de la fin qui contiendra la
    valeur
    S : 195
    */
    ptrElement t;

    liste->fin->cle= valeur;
    if (listeVide(liste))
        return liste->fin;
    for(t= liste->tete->suivant;cmp(t->cle, valeur);t= t->suivant);
    return t;
}

ptrElement rechercheAvantElement(ptrListe liste, void *valeur,
    int (*cmp)( void *valeurListe, void *valeurDonnee) )
{
    /* Pre : cmp renvoie 0 si les valeurs sont comparables
    Post : on renvoie l'adresse du l'élément précédant le 1°
    à partir de la tête qui corresponde à la valeur
    à partir du début; en cas d'absence, on renvoie
    l'adresse qui précède la fin qui contiendra la valeur
    S : 195
    */
    ptrElement t;
    int i=0;

    liste->fin->cle= valeur;
    for(t= liste->tete;cmp(t->suivant->cle, valeur);t= t->suivant);
    return t;
}
}

```