Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un interpréteur abstrait de programmes prolog

Bourgeois, Alain

Award date:
1992

Awarding institution:
Universite de Namur

Link to publication

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

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

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

Download date: 20. Apr. 2024

FACULTES
UNIVERSITAIRES
N. D. DE LA PAIX

N A M U R

INSTITUT D'INFORMATIQUE

IMPLEMENTATION D'UN INTERPRETEUR ABSTRAIT DE PROGRAMMES PROLOG

par Alain Bourgeois

Promoteur :

Professeur B. Le Charlier

Mémoire présenté en vue
de l'obtention du titre de
Licencié et Maître
en Informatique

Année académique : 1991-1992

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
rue de Bruxelles 61, B-5000 NAMUR
Tél. 081-72.41.11
Télex 59222 facnam-b
Téléfax 081-23.03.91

Implémentation d'un interpréteur abstrait de programmes prolog

Alain Bourgeois

Résumé

Ces dernières années, les langages de programmation déclaratifs ont été de plus en plus reconnus et utilisés. Ceux-ci obligent les programmeurs à travailler proprement et à faire des programmes simples. Cependant, pour que le code généré soit efficace, certaines analyses sont nécessaires. Une de ces analyses est l'interprétation abstraite, dont l'application la plus importante est l'analyse de types. Elle permet de détecter les optimisations pouvant être effectuées et de déterminer certaines erreurs de programmation. Le but de ce mémoire est d'implémenter un interpréteur abstrait de programmes prolog, qui, étant donné un programme, une requête et des informations sur cette requête, essaiera de déterminer les types et les formes des variables de la substitution résultat.

Abstract

The declarative programming languages have been more and more recognized and used in recent years. They demand the programmers to work clearly and to write simple programs. However, the generated code must be efficient, and so some analyses are needed. One of these analyses is abstract interpretation, whose most important application is type analysis. It permits to detect the optimizations that can be realized and to point out some programmation errors. The aim of this dissertation is to implement an abstract interpretor of prolog programs, which, given a program, a query and some informations on the query, will try to determine the types and the forms of the variables in the resulting substitution.

Mémoire de maîtrise en Informatique
Septembre 1992
Promoteur : B. Le Charlier

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué, de près ou de loin, à la rédaction de ce mémoire. En premier lieu, mon chef de stage et ami, Mr Kaninda Musumbu, pour sa patience et ses nombreuses explications. Ensuite, Messieurs Antoine Rauzy et Achille Braquelaire, pour leurs explications et conseils sur la programmation en C. Viennent ensuite les étudiants de DEA informatique de Bordeaux de l'année académique 1991-1992, pour leurs renseignements et leur aide à manipuler les stations Sun et les utilitaires. Mr François Pellegrini me fut d'une aide précieuse concernant l'utilisation de lex et yacc, et Mr Frédéric Goudal pour ses informations sur UNIX.

Ensuite, je tiens à remercier mes parents pour avoir rendu mon séjour à Bordeaux possible et agréable : mes parents, et mon promoteur, Mr Baudouin Le Charlier, pour ses conseils et pour avoir accepté de lire les drafts. La qualité finale de ce texte lui doit énormément.

Table des matières

Introduction 1

Première partie : Présentation théorique

Chapitre I : Qu'est-ce que l'interprétation abstraite ? 4
1. Généralités 4
2. La théorie du point fixe 9
3. Applications à l'interprétation abstraite 12
Chapitre II : Interprétation abstraite de prolog 14
1. Syntaxe 14
2. Sémantique concrète : SLD-résolution 18
3. Domaine abstrait 19
4. Algorithmes 24

Deuxième partie : le système implémenté

Introduction 32
Chapitre I : Implémentation du domaine 33
1) Tableaux 33
Représentation des ps 33
Exemple 35
2) Listes chaînées 36
Représentation des types 39
Chapitre II : Normalisation et représentation interne des programmes 41
Normalisation des programmes 41
Chapitre III : Implémentation des algorithmes sur le domaine 46
Chapitre IV: Implémentation des algorithmes génériques 49
Lecture des données 53
Chapitre V: Résultats obtenus 55

Bibliographie 77

Introduction

L'interprétation abstraite est un secteur de recherche actif à l'heure actuelle. Elle est souvent utilisée dans le cas de prolog et constitue un outil utile pour l'analyse des programmes. Les résultats qu'elle fournit peuvent aider un programmeur à optimiser son code et à y déceler des erreurs.

L'interprétation abstraite est un cadre dont le but est d'aider à la création, de manière systématique, d'outils permettant de déduire des propriétés sur l'exécution des programmes. C'est une analyse statique, c'est-à-dire indépendante d'une exécution quelconque. Les calculs sont exécutés non pas sur le domaine réel des objets mais sur un domaine abstrait bien choisi.

Une première partie est destinée à expliquer la théorie générale sous-jacente au système implémenté, c'est-à-dire la théorie de l'interprétation abstraite en général d'une part, et l'adaptation à prolog d'autre part. La théorie de l'interpétation abstraite comprend la sémantique mathématique et la théorie du point fixe, ainsi que quelques applications. L'adaptation à prolog explique la syntaxe de prolog, son fonctionnement, le choix du domaine abstrait et donne quelques algorithmes de base. La deuxième partie explique le système implémenté, la représentation des données et explique la manière dont les algorithmes ont été implémentés.

La thèse de doctorat de Mr Kaninda Musumbu comprend les fondements théoriques sous-jacents à tous les algorithmes, ainsi que toutes les démonstrations. Elle contient des spécifications formelles de procédures ainsi que des algorithmes exprimés dans un langage de haut niveau totalement indépendant de tout choix d'implémentation. Un autre complément de cet ouvrage est l'article "Efficient and accurate algorithms for the abstract interpretation of prolog programs", qui contient l'algorithme principal déterminant la marche à suivre. Cet algorithme appelle des procédures spécifiées dans la thèse, et est basé sur une sémantique de point fixe, également décrite dans cet article. Ce dernier comprend quatre versions de l'algorithme, chaque version constituant une optimisation de la précédente. La première version ne fut pas implémentée pour des raisons évidentes d'inefficacité. La seconde version est équivalente à la première du point de vue des spécifications, mais est efficace et fut implémentée. Les versions suivantes partent de la seconde, mais en optimisent les imperfections. La version 3 fut également implémentée,

mais les traces ont été vérifiées complètement uniquement pour la version 2, laquelle sera explicitée dans ce mémoire.

Partie I :

Présentation théorique

Chapitre I :
Qu'est-ce que l'interprétation abstraite ?

1. Généralités[1]

L'interprétation abstraite est une méthode d'analyse mathématique permettant de déduire des propriétés de programmes via une analyse statique. L'analyse statique regroupe tous les traitements que l'on peut appliquer à un programme en dehors de son exécution. Cette analyse peut être employée, par exemple, par un compilateur afin de produire un code plus efficace; par exemple : détection de bouclages, des expressions constantes au sein d'une boucle, des traitements inutiles, etc. On peut aussi vérifier le caractère bien typé des expressions. L'interprétation abstraite est une méthode générale et mathématiquement correcte. Une des applications consiste à exécuter le programme non pas sur des valeurs qui fourniraient des résultats différents pour chaque exécution, mais bien sur des valeurs prises dans un domaine abstrait tel que chaque élément de ce domaine soit représentatif d'un ensemble de valeurs concrètes vérifiant certaines propriétés, et à déterminer la valeur de domaine représentant le mieux le résultat.

L'interprétation abstraite peut être employée sur tout type de langage et permet d'obtenir des informations substantielles. Dans les langages impératifs séquentiels ou traditionnels tels que C, PASCAL, BASIC, ..., l'application la plus importante est l'analyse de types. Cette analyse permettra par exemple de déterminer le ou les types possibles d'une expression à l'exécution, donc de déterminer des erreurs de programmation ou d'optimiser le code en conséquence. Dans les langages impératifs parallèles, l'analyse des communications entre processus permettra entre autres de détecter les cas d'interblocage. Dans les langages fonctionnels, elle permettra de décider quand l'appel par valeur pourra être utilisé plutôt que l'appel par nom afin de gagner du temps. Elle permettra aussi de faire du compile time garbage collection, c'est-à-dire de remplacer certaines allocations et libérations dynamiques de mémoire par des allocations statiques plus efficaces. En ce qui concerne les langages logiques tels que PROLOG, ils offrent une caractéristique très

[1] Basé sur [ABR87], [LEC91-3].

importante, la *multidirectionnalité* : tout paramètre d'une procédure pourra être utilisé indifféremment comme donnée ou comme résultat. Le caractère déclaratif du langage cache la démarche grâce à laquelle le programme va trouver le résultat demandé et les possibilités d'utilisation de tels langages sont quasi infinies. Cependant, cette multidirectionnalité sera très peu employée en pratique, car une procédure est en général écrite dans un but, pour fonctionner dans un seul sens particulier. Dans l'exemple d'un programme de tri, on lui donnera un ensemble de valeurs à trier, mais on n'emploiera en général pas ce programme pour obtenir toutes les permutations quelconques d'une liste triée (exécution du programme dans l'autre sens). L'interprétation abstraite permettra de détecter quelles seront les utilisations faites et ainsi d'obtenir un code plus rapide et plus compact. Elle permettra aussi l'analyse de mode et de type, la suppression de l'occur-check, la détection des procédures déterministes. Les résultats qu'elle apportera permettront de détecter les opportunités de faire du compile time garbage collection. Les langages orientés objet offrent des possibilités d'analyse semblables à celles des langages logiques, et une analyse de types permettra de spécialiser le code généré et de remplacer certains liens dynamiques par des liens statiques. L'interprétation abstraite servira aussi à détecter des incohérences au niveau de la programmation.

L'interprétation abstraite fut d'abord introduite pour les langages impératifs classiques. Ces dernières années, l'importance des langages déclaratifs a été de plus en plus reconnue : le programmeur va plus vite, et est pratiquement obligé de programmer proprement (ou presque), se doit de faire des programmes simples et faciles à maintenir. L'application de l'interprétation abstraite à ces langages a beaucoup d'intérêt, car les gens qui emploient ces langages ne sont pas toujours informaticiens. Les langages déclaratifs travaillent à un niveau d'abstraction très élevé et le code généré par le compilateur peut être optimisé. Par exemple, quelqu'un qui programme en SQL ne connaît pas nécessairement les techniques d'indexation, et une requête d'interrogation sur une base de données peut être effectuée de différentes manières avec des temps d'exécution pouvant aller du simple au triple. L'interprétation abstraite peut faire opter le compilateur pour le choix d'une méthode plutôt qu'une autre, en fonction d'une évaluation par exemple du nombre d'accès effectués. De plus, elle peut déterminer si un index est nécessaire et provoquerait un gain de temps significatif. Un autre exemple est le cas de prolog. Le seul moyen permettant de faire des boucles proprement est d'employer la récursivité. Si cette récursivité n'est pas terminale, elle s'accompagnera de la gestion d'une pile dont le programmeur peut ignorer l'existence. De plus, les variables sont universellement quantifiées en prolog. En général, les langages déclaratifs de haut niveau cachent au programmeur les difficultés d'une implémentation en langage machine, ainsi que les besoins réels en ressources du programme. De plus, le programme ne s'exécute pas

toujours de la façon que le programmeur a imaginée en écrivant son programme car l'algorithme "impératif" est généré par le compilateur. Pour toutes ces raisons, les langages déclaratifs s'adaptent bien à toute une série d'optimisations qui peuvent être effectuées grâce aux résultats fournis par une analyse telle que l'interprétation abstraite.

Exemples d'utilisation :

Le premier exemple est le suivant : supposons qu'il me faille partir en vacances. J'ai plusieurs possibilités : je peux partir à pied, en voiture ou en avion. Mon choix se fera en fonction de la distance à parcourir. La carte est donc une représentation abstraite du voyage et en mesurant la distance on peut abstraire le transport.

Le deuxième exemple est plus proche de la programmation : considérons que nous voulons connaître le signe de l'expression (-5125)*4589. Nous pouvons affirmer que le résultat sera négatif sans effectuer le calcul, sur base de la règle des signes bien connue. L'interprétation abstraite permet de déduire des informations sur un programme sans pour cela l'exécuter.

Nous pouvons obtenir des réponses précises dans ces deux exemples. Cela devient moins clair si l'on inclut l'addition :

$+ \pm 0 = 0 \pm + = +$
$- \pm 0 = 0 \pm - = -$
$+ \pm + = +$
$- \pm - = -$

Ces premières règles ne posent pas de problèmes.

$+ \pm - = ??$
$- \pm + = ??$

Ces deux dernières règles d'utilisation posent un problème : nous ne pouvons déduire de résultat précis ou ce serait une erreur. Nous pouvons seulement dire que ?? représente un ensemble de réels, et nous voyons que l'interprétation abstraite a des limites. Il y a des choses qu'elle ne pourra pas résoudre (c'est-à-dire, le résultat sera moins précis que le résultat réel qui dépendra de données bien précises).

Le principe de l'interprétation abstraite est d'exécuter un programme sur un domaine "non standard" ou abstrait, sur lequel nous pourrons raisonner mathématiquement. Pour que nos raisonnements soient corrects, la théorie exige deux hypothèses :

1. Les éléments du domaine abstrait représentent des propriétés utiles du domaine standard;
2. Les calculs sur le domaine abstrait peuvent être réalisés de manière suffisamment efficace, et convergent en un temps fini.

Ce domaine abstrait (noté A) est nécessaire car il est impossible d'exécuter le programme pour chacune des valeurs du domaine concret (noté C) qui est infini. De plus, l'interprétation abstraite est une analyse statique dont le but est de tirer des conclusions générales en dehors d'une exécution quelconque. A chaque élément du domaine concret correspond un élément du domaine abstrait. Un élément du domaine abstrait peut correspondre à plusieurs éléments du domaine concret. On exige en général que A soit un treillis complet et qu'il existe deux fonctions monotones α (fonction d'abstraction) et γ (fonction de concrétisation) telles que :

$$\alpha: C \longrightarrow A$$
$$\gamma: A \longrightarrow C$$

Ces deux fonctions doivent de plus vérifier les conditions suivantes :

1. $\forall c \in C: \gamma(\alpha(c)) \supseteq C$
2. $\forall a \in A: \alpha(\gamma(a)) = a$

Illustrons la théorie mathématique par un exemple. Soit un langage fonctionnel quelconque, et D le domaine des valeurs qu'il manipule. Tout programme calcule une fonction de D $\rightarrow$ D. Soit la procédure suivante dans un langage du style du C :

```
int f(int x)
{
    return (x > 100) ? x - 10 : f(f(x + 11));
}
```

ou, en style fonctionnel :

f(x) = **si** x>100 **alors** x-10 **sinon** f(f(x+11))

Cette procédure calcule une fonction de **Z→Z**. Il serait intéressant de pouvoir déterminer la valeur de la procédure en différents points, ce qui nous amène à travailler avec des ensembles de valeurs plutôt qu'avec des valeurs individuelles de D. Le domaine concret sera un ensemble de valeurs : la fonction donnera tel résultat pour n'importe quelle valeur prise dans un ensemble donné. Cet ensemble d'ensembles de valeurs sera noté C (domaine concret), et nous pouvons réécrire notre programme pour qu'il travaille sur C, et X représente un ensemble de valeurs.

f(X) = { x-10 : x>100 **et** x∈X} U f(f({x+11 : x≤100 et x∈X}))

Cette amélioration est bonne mais insuffisante : nous ne pouvons nous permettre en pratique de travailler sur des ensembles quelconques de valeurs. Le domaine abstrait constitue un sous-ensemble de C tel que chaque élément de C peut être approximé par un seul élément de A. Un bon choix pour A est l'ensemble des intervalles : un intervalle [i..s], où i,s ∈ **Z**U{+∞,-∞}, représente l'ensemble des entiers x tels que i≤x≤s. De plus, A constitue un treillis complet pour l'inclusion avec un élément minimal : l'intervalle vide, et une borne supérieure [-∞..+∞]. Tout ensemble de valeurs peut être approximé par le plus petit intervalle contenant tous ses éléments. Nous pouvons maintenant réécrire la fonction précitée comme suit :

f([i..s]) = [max(91,i-10)..(s-10)] *U* f(f([(i+11)..min(s+11,111)]))

En effet, tout ensemble de valeurs A peut être approximé par l'intervalle [min(A)..max(A)]. A chaque ensemble de valeurs correspond un et un seul intervalle, et à chaque intervalle correspondent plusieurs ensembles de valeurs (sauf dans le cas où l'intervalle contient moins de deux éléments). L'ensemble X de notre fonction travaillant sur C peut être approximé par l'intervalle [i..s]. L'ensemble { x-10 : x>100 **et** x∈X } peut être transformé comme suit : { x-10 : x>100 et i≤x≤s }. Cet ensemble peut être approximé par un intervalle : [i-10..s-10] mais on perd le fait que x doit être supérieur à 100, d'où l'introduction du max(91,i-10) qui garantit, le cas échéant, que l'intervalle sera réduit aux valeurs pour lesquelles x est supérieur à 100 (100+1-10=91). Le raisonnement est semblable pour la transformation de l'ensemble { x+11 : x≤100 et x∈X }.

Nous pouvons essayer d'approximer les valeurs produites par la fonction :

f([-∞..+∞]) = [91..+∞] *U* f(f([-∞..111])),

$f([-\infty..111]) = [91..101] \cup f(f([-\infty..111]))$.

Le calcul au sens habituel est impossible : il y a un bouclage car un calcul peut se générer lui-même. Le calcul sur le domaine abstrait ne peut se faire de manière similaire au calcul sur le domaine standard. Ce problème sera résolu par la théorie du point fixe explicitée ci-après.

Le but de ce mémoire est de réaliser un interpréteur abstrait de programmes prolog, c'est-à-dire de déterminer la valeur d'une fonction dans le domaine abstrait étant donné un programme, une valeur du domaine et une requête.

2. La théorie du point fixe[1]

La théorie du point fixe va essayer de résoudre le problème d'ambigüité des définitions récursives. En effet, une telle définition s'appelle elle-même et il est difficile d'en saisir le sens.

Considérons une définition de fonction récursive quelconque. Cette fonction peut être transformée en une autre fonction renvoyant la même valeur, mais effectuant moins d'appels récursifs. Pour éviter que la définition s'appelle elle-même, nous pouvons définir une transformation de fonction τ qui, à chaque appel de la fonction, renvoie une autre fonction pour calculer le résultat. Le résultat s'améliore à chaque fois, si bien que les fonctions approximent de mieux en mieux le résultat, et forment une chaîne dans le domaine. Lorsque le résultat est atteint, il ne peut plus être amélioré et le point fixe de la transformation est atteint.

Les deux représentations sont équivalentes : une définition de fonction récursive revient à définir une transformation de fonction.

Cette théorie nécessite une transformation continue qui améliore toujours le résultat et forme une chaîne dans le domaine, limitée par une borne supérieure. Cette borne constituera un point fixe de la transformation (noté $\mu(\tau)$). Cette transformation doit être monotone et continue, et travailler sur un domaine qui est un treillis complet. Il est

[1] Basé sur [LEC91-2], [LEC91-3], [LER77].

évidemment nécessaire d'avoir une **relation d'ordre**, notée $\sqsubseteq$. Cette suite doit être monotone pour converger vers un résultat, et donc former une chaîne. Une chaîne est une suite infinie d'éléments $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq ... \sqsubseteq d_i \sqsubseteq ...$ Dans les ensembles fonctionnels, chaque élément est une fonction (un programme). Dans ces ensembles, chaque chaîne admet une **borne supérieure** (ou lub, noté U), c'est-à-dire qu'à un moment donné, si la suite converge, un élément se répète à l'infini. Il est nécessaire d'adjoindre un élément **bottom** (noté $\perp$) à la chaîne afin d'y désigner tous les programmes qui bouclent. Cet élément est l'élément minimal de l'ensemble fonctionnel (il est plus petit que tous les autres). Lorsqu'un ensemble admet ces trois propriétés (relation d'ordre, borne supérieure et élement minimal), il s'agit d'un **ensemble inductif**. Ces ensembles inductifs possèdent plusieurs propriétés intéressantes, notamment le fait que le produit cartésien d'ensembles inductifs soit un ensemble inductif, ce qui nous permet de travailler sur des domaines complexes formés à partir de domaines simples. Pour que les fonctions représentent des algorithmes, la monotonie n'est pas suffisante, car une fonction peut dépendre d'une infinité de valeurs. C'est pour cela que nous avons besoin d'une propriété plus forte qui est la continuité.

Définition :

Soient E, F deux ensembles inductifs,
une fonction f: $E \rightarrow F$,

f est continue si, et seulement si,

1) f est monotone : $\forall d_1, d_2 \in E : d_1 \sqsubseteq d_2 \Rightarrow f(d_1) \sqsubseteq f(d_2)$,

2) Pour toute chaîne $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq ... \sqsubseteq d_i \sqsubseteq ...$, $f(\bigcup_{i=1}^{\infty} d_i) = \bigcup_{i=1}^{\infty} f(d_i)$.

Donc, toute fonction continue est monotone, mais l'inverse n'est pas vrai. L'ensemble des fonctions peut également être muni d'un ordre :

$f \sqsubseteq g$ si, et seulement si $\forall d \in E : f(d) \sqsubseteq g(d)$

Un point fixe d'une transformation de fonction τ est atteint lorsque $\tau(f)=f$. Ce point fixe est la borne supérieure de la chaîne et c'est lui qu'il faut essayer de déterminer car il constitue le résultat le plus précis tout en étant correct.

Il y a moyen de montrer que les domaines concrets et abstraits vérifient les propriétés énoncées précédemment, tout point fixe de τ fournit une approximation correcte des propriétés de la procédure associée; de plus, τ possède un plus petit point

fixe (le plus précis) qui est égal à la borne supérieure de la suite croissante d'approximations, quel que soit le τ choisi pour autant que la transformation soit bien choisie et donc équivalente à la définition récursive.

Le problème consiste maintenant à trouver des méthodes de calcul de point fixe. Toute définition récursive peut être calculée de deux manières : ascendante ou descendante. Prenons l'exemple de la factorielle : x!= si x=0 alors 1 sinon x*(x-1)!. Pour calculer 5! de manière ascendante, nous partons de 0, calculons 0!, stockons le résultat, puis calculons 1!, puis 2!, et ainsi de suite. L'avantage d'une telle méthode est qu'étant donné que chaque appel récursif pour une valeur X ne nécessite que les valeurs de la fonction pour les valeurs inférieures à X, en connaissant les valeurs entre 0 et X-1, nous pouvons connaître la valeur de la fonction pour X sans gérer de pile et sans faire d'appels récursifs (puisque nous avons les valeurs); et calculer la table de la fonction de cette manière. Nous ne calculons donc qu'une fois chaque valeur. L'inconvénient de la méthode est que nous calculons toutes les valeurs de la fonction entre 0 et X-1 pour obtenir la valeur pour X, alors que quelques-unes seulement sont nécessaires. La deuxième manière d'évaluer une fonction récursive : partir de la valeur que l'on veut et calculer les valeurs qui en dépendent récursivement. L'avantage est que nous ne calculons que les valeurs dont nous avons besoin, mais par contre une même valeur peut être calculée plusieurs fois et une pile doit être gérée.

L'algorithme le plus naïf est un algorithme ascendant et consiste à calculer la table complète :

$$f_0(a) = \bot,\ \forall a \in A$$
$$f_{i+1} = \tau(f_i),\ \forall i \geq 0$$
$$f = f_n,\ \text{si } f_{n+1} = f_n$$

Cette méthode exige que le domaine abstrait A soit fini et est presque toujours impossible à utiliser (de même qu'exécuter un programme avec chaque donnée d'entrée possible). De plus, le domaine doit être petit sans quoi cela devient vite infaisable en pratique.

Cette méthode calcule le point fixe de manière ascendante et donc toutes les valeurs. L'idée serait d'employer une méthode descendante pour ne calculer que ce dont on a besoin. Pour ne pas calculer deux fois la même valeur, il est possible de garder en mémoire les valeurs pour lesquelles une approximation a déjà été trouvée et renvoyer cette approximation en cas de demande de calcul. On garde aussi en mémoire les

approximations entamées et non terminées auxquelles une approximation minimale est donnée au départ. Le résultat est amélioré à chaque itération et le calcul s'arrête lorsqu'aucune amélioration du résultat n'est constatée.

En présence d'un domaine infini, la méthode explicitée ci-dessus peut boucler, dans le cas où une infinité d'appels différents sont enclenchés. Une des solutions pour éviter ce cyclage est de se limiter à des domaines abstraits finis, mais de tels domaines abstraits ne représentent pas toujours adéquatement les domaines concrets de par le manque de précision qu'ils occasionnent.

3. Applications à l'interprétation abstraite[1]

Quelques analyses importantes pour les langages déclaratifs et logiques sont citées ci-après :

Strictness analysis : permet l'optimisation de langages fonctionnels en déterminant quels paramètres peuvent être passés par valeur et ouvre des portes pour l'évaluation parallèle.

In-place Update Analysis : permet de déterminer dans un programme les points auxquels un objet peut être détruit (il n'y a plus de références vers cet objet).

Relevant Clause Analysis : dans les architectures de cinquième génération, les programmes sont autorisés à accéder à des définitions de fonctions non locales, c'est-à-dire à appeler dynamiquement des fonctions quelconques, qui peuvent ne pas encore être écrites au moment de la compilation du code, et sans nécessiter de recompilation du code. Cela permet de créer des liens dynamiques entre applications. Un exemple est l'imbrication d'un tableau dans un traitement de texte : le tableur est appelé dynamiquement à partir du traitement de texte, et le résultat du tableur (le tableau encodé) est inséré dans le texte. Pour que cela soit possible, un surplus de données doit être associé au programme pour lui permettre de communiquer avec d'autres applications. Une analyse permettra de déterminer quelles définitions et

[1] Basé sur [ABR87].

quelles parties de définitions sont susceptibles d'être utilisées par le programme et donc de réduire le surplus dans certains cas.

Mode Analysis : des améliorations de performances significatives peuvent être effectuées dans les interpréteurs PROLOG si on sait la manière dont les variables seront employées dans une relation. Ce mémoire permettra d'analyser les programmes PROLOG.

Chapitre II :
Interprétation abstraite de prolog

1. Syntaxe[1]

La syntaxe de prolog est très simple : il n'y a pas de mots réservés si ce n'est les expressions mathématiques usuelles (cos, sin, ...), et l'opérateur d'évaluation is. Un programme est une suite de clauses et de faits. Une clause est constituée d'une tête de clause, d'une flèche (notée :-) et d'une série de sous-buts séparés par des virgules. Chaque clause (fait) se termine par un point. Chaque tête de clause et chaque fait est un prédicat de la forme p(p1, p2, ..., pn). Soit l'exemple suivant :

carré(X,Y) :- X is Y*Y.

Cette clause peut se lire "carré (X,Y) est vrai si la valeur de X est la valeur de Y*Y". Chaque majuscule indique qu'il s'agit d'une variable. Un fait est une tête de clause, il représente une clause sans sous-buts (exemple : père(joseph, jésus)). Prolog travaille avec des substitutions; une substitution est un ensemble de valeurs à associer à des variables. Par exemple, si on pose la question père (X, jésus) au programme constitué du fait cité plus haut, le programme va fournir pour résultat la substitution {X/joseph}. La seule manière de faire des boucles est d'employer la récursivité. Toutes les variables sont locales à la clause à laquelle elles appartiennent, et sont universellement quantifiées; il n'est donc pas nécessaire de les déclarer. Le programme "grand-père" peut être facilement écrit à partir des prédicats père et mère :

grand-père (X,Y) :- père(X,Z), mère(Z,Y).
grand-père (X,Y) :- père(X,Z), père(Z,Y).

Deux clauses distinctes représentent un 'ou' logique : on trouve un résultat si l'exécution d'une des deux réussit. Les sous-buts au sein d'une même clause représentent

[1] Basé sur [MUS90], [STE86], [JAQ90].

une conjonction : l'exécution réussit si chaque sous-but réussit. Les boucles peuvent être exprimées récursivement, comme le montre la procédure ancêtre :

```
ancêtre (X,Y) :- parent (X,Y).
ancêtre (X,Y) :- parent (X,Z), ancêtre (Z,Y).
```

où parent peut être défini comme suit :

```
parent (X,Y) :- père (X,Y).
parent (X,Y) :- mère (X,Y).
```

Il existe un type particulier de variable en prolog : le type liste. Une liste est une suite d'éléments, et est définie comme en lisp, à savoir qu'une liste est constituée d'un élément quelconque, puis d'une liste. Chaque liste se termine par une constante prédéfinie qui est la liste vide. En prolog, '[' marque le début d'une liste; '|' est le symbole de concaténation et ']' marque la fin d'une liste. [] désigne la liste vide. Le programme suivant réalise la concaténation de deux listes pour en fournir une troisième qui est le résultat :

```
append ([], X, X).
append ([X1|X2], X3, [X1|X4]) :- append (X2,X3,X4).
```

Une caractéristique intéressante des programmes prolog est la multidirectionnalité, à savoir que le programme est son propre inverse. Par exemple, la procédure "carré" permet non seulement d'élever un nombre au carré, mais aussi de calculer la racine carrée d'un nombre. Si nous exécutons append en donnant une liste comme troisième paramètre, il va nous fournir toutes les paires de listes dont la concaténation donne la liste que nous lui avons donnée.

Prolog essaie de répondre aux requêtes qui lui sont posées en substituant les bonnes valeurs aux variables non instanciées (qui n'ont pas encore reçu de valeur). Soient Pi l'ensemble des symboles de prédicat d'arité i, Fi l'ensemble des foncteurs d'arité i. Il y a deux types de sous-buts : "t1 **op** t2" et "p(t1, t2, ..., tn)", où pn est une procédure du programme ($p \in Pn$). **op** représente un opérateur de comparaison (= pour l'unification, is pour l'évaluation arithmétique, <, <=, >, >=, <> pour les comparaisons mathématiques usuelles). Les ti sont des termes. Un terme est soit une variable, soit une construction de la forme f(t1, ..., ti) où $f \in Fi$ et t1, ..., ti sont des termes. Une substitution est un ensemble de type { X1/t1, X2/t2, ..., Xn/tn } où à chaque variable Xi on associe un terme ti, $Xi \neq Xj$ si $i \neq j$ et $Xi \neq ti$ $\forall i,j$. Soit la substitution σ, alors $t\sigma$ désigne le terme obtenu à partir de t en

remplaçant chaque occurence d'Xi par le ti associé. Les variables X1, ..., Xn constituent le **domaine** de σ (noté dom σ) et les variables présentes dans les ti constituent le codomaine de σ (noté codom σ).

L'interprétation abstraite est une théorie générale capable de s'appliquer sur tout langage; néanmoins, quelques adaptations préalables du programme sont nécessaires pour simplifier la théorie. La source prolog sera d'abord vérifiée syntaxiquement via lex et yacc, et le programme s'arrêtera en cas d'erreur. Après quoi, le programme sera normalisé, à savoir que chaque programme sera réécrit en respectant les conditions suivantes.

Soient F_i l'ensemble des foncteurs d'arité i, P_i l'ensemble des symboles de prédicat d'arité i.

- Les **variables** au sein d'une clause seront renommées X_i, $1 \leq i \leq n$ où n est le nombre de variables dans la clause normalisée.

- Un **terme** est :
  - soit une variable,
  - soit une construction de la forme $f(t_1, ..., t_i)$ où $f \in F_i$ et $t_1, ..., t_i$ sont des termes.

- Un **atome normalisé** est :
  - soit un built-in,
  - soit un appel de procédure.

- Un **built-in** est une construction de la forme "$X_i=X_j$" où $i \neq j$, ou une construction de la forme "$X_i=f(X_{j1}, ..., X_{jn})$" où $f \in F_n$ et les variables X_i, $X_{j1}, ..., X_{jn}$ sont distinctes deux à deux.

- Un **appel de procédure** est une construction de la forme "$p(X_{i1}, ..., X_{in})$" où $p \in P_n$ et les variables $X_{i1}, ..., X_{in}$ sont distinctes deux à deux.

- Une **clause normalisée** est de la forme "$p(X_1, ..., X_n) \leftarrow$ SB", où n est positif et SB est une suite d'atomes normalisés ne contenant que des variables de programme. $p(X_1, ..., X_n)$ est la tête de la clause et p_n est son foncteur, SB est le corps de la clause. Chaque atome appartenant à SB est un **sous-but** de la clause ou un **littéral**.

- Une **procédure normalisée** est une suite non vide de clauses normalisées de même foncteur.

- Un **programme normalisé** est un ensemble de procédures normalisées de foncteurs distincts tel que tout foncteur figurant dans le corps d'une clause est le foncteur d'une procédure.

Tout programme peut s'écrire sous forme normalisée : des variables peuvent être créées par la normalisation et chaque terme sera aplati. Chaque terme du genre "f(X)=f(Y)" sera remplacé par "X_m=f(X_i), X_n=f(X_j), X_m=X_n". Les arguments des foncteurs seront remplacés par des variables, lesquelles seront garnies avant l'appel de procédure ("normalisation préfixée").

Exemple : considérons le programme *append* cité précédemment. Il sera normalisé comme suit :

```
append(X1, X2, X3) :- X1=[], X2=X3.
append(X1, X2, X3) :- X1=[X4|X5], X3=[X4|X6], append(X5, X2, X6).
```

Un autre exemple (sans signification utile) sera :

```
p(X,Y) :- X=f(g(h(Z)),Y), p(h(g(X)),Y)
```

et qui sera normalisé de la manière suivante :

```
p(X1,X2) :=  X4=h(X3), X5=g(X4), X1=f(X5,X2), X6=g(X1), X7=h(X6),
             p(X7,X2).
```

Nous pouvons remarquer que les variables remplaçant les arguments des foncteurs sont garnies avant l'exécution du sous-but, excepté pour la tête de clause (cas de append). Il est important de normaliser les programmes prolog car cela simplifie la théorie, les notations, et la gestion des données (on ne prévoit que trois types de sous-buts différents, ...). Lorsque tout est aplati, il ne reste que des atomes normalisés : soit "X_i=X_j", soit "X_i=f(X_{j1}, ..., X_{jn})", soit "p(X_{i1}, ..., X_{in})". En fait, en pratique, il y aura quatre types de littéraux différents. Un troisième type de built-in sera ajouté : "X_i<X_j" où i est différent de j, permet d"indiquer une comparaison.

2. Sémantique concrète : SLD-résolution[1]

Cette partie est destinée à expliquer la sémantique dénotationnelle de prolog, c'est-à-dire la manière dont il s'y prend pour répondre aux requêtes. Pour répondre à une question, le moteur d'inférence de prolog (aussi appelé horloge prolog) travaille comme suit : étant donné une question, soit il possède une substitution de départ sur une des variables (dans le cas d'un programme de tri, la liste non triée), soit il va trouver comme résultat des variables qui risquent de ne pas être instanciées. Il recherche la première clause dont la tête matche la requête (mêmes foncteurs et arguments) et l'exécute. S'il y a plusieurs clauses dont la tête matche la requête, il mémorise cet endroit comme point de choix et la substitution courante. Chaque sous-but est soit une unification (=), soit un test (<, >, <=, >=, <>), soit un appel de procédure. Dans le cas d'un test, s'il réussit, il continue. Si le test échoue, il revient au point de choix précédent et effectue le choix suivant. S'il n'y a plus de point de choix, il renvoie un échec. S'il s'agit d'une unification, il effectue un test d'unification. Si ce test réussit, il continue; sinon il fait la même chose que lorsqu'un test de comparaison échoue. Dans le cas d'un appel de procédure, il stocke la substitution courante ainsi que le point de choix, effectue un changement de variable et exécute la première clause dont la tête matche l'appel. Lorsqu'une clause se termine, la substitution obtenue est restreinte aux variables de la tête de clause, et un changement de variables inverse à celui de effectué lors du début d'exécution de la clause se produit. Toutes les variables instanciées dans la substitution résultat sont remplacées par leur valeur dans la substitution stockée lors de l'entrée de la clause, et l'exécution continue. Lorsque la première clause est terminée, il affiche le résultat trouvé et l'utilisateur peut arrêter ou continuer. Si le résultat contient une variable non instanciée, il va donner toutes les valeurs possibles pour cette variable. Sinon, si l'utilisateur décide de continuer, l'ordinateur remonte au dernier point de choix, change de choix et repart de l'endroit auquel il est arrivé. Si l'ordinateur ne trouve aucun résultat, il affiche un échec.

Le test d'unification est décrit par la procédure suivante :

répéter tant que possible

- cas 1 : l'équation est de type $f(t_1, ..., ...t_m)=f(u_1, ..., u_m)$ avec $m \geq 0$,
  action : remplacer l'équation par $t_1=u_1, ..., t_m=u_m$ et effectuer les unifications.

- cas 2 : l'équation est de type $f(t_1, ..., ...t_m)=g(u_1, ..., u_n)$ avec $f \neq g$ ou $m \neq n$,

[1] Basé sur [JAQ90], [STE86].

action : arrêter avec échec.

- cas 3 (cas dégénéré) : l'équation est de type X=X,
  action : éliminer l'équation.

- cas 4 : l'équation est de type t=X et t n'est pas une variable,
  action : remplacer l'équation par X=t.

- cas 5 (cas X=t).
  action : Si X apparaît dans t (occur-check)
  alors arrêter avec erreur (bouclage infini, exemple : X=f(X))
  sinon remplacer toute occurence de X dans le système par t.

3. Domaine abstrait[1]

Le domaine abstrait sera constitué de substitutions abstraites. Pour trouver un domaine abstrait adéquat, nous devons nous intéresser à ce qui caractérise les substitutions concrètes. A chaque substitution concrète correspondra une substitution abstraite, et à chaque substitution abstraite correspondront plusieurs substitutions concrètes. Une fonction de concrétisation (notée cc) associe à chaque substitution abstraite l'ensemble des substitutions concrètes qui y correspondent.

Elles sont caractérisées par un ensemble de variables dont certaines peuvent être identiques, et par des termes. Chaque terme possède un type et une forme. Nous supposerons que le type appartient à l'ensemble des types qui sera fixé au préalable. Il comprendra au minimum l'élément minimal $\perp$ et la borne supérieure ANY, qui n'apporte aucune information car il indique que le type est quelconque. Ce domaine des types doit constituer un treillis complet. Les termes d'une substitution seront indicés de 1 à p; j'appellerai forme tout terme s'écrivant $f(i_1,...,i_q)$ où $f \in F_q$, chaque $i_1,...,i_q$ étant l'indice d'un terme de la substitution abstraite. Chaque terme sera caractérisé par un triplet (tp,frm,ps). La composante tp contient le type du terme (type pris dans un domaine), frm contient sa forme et ps (possible sharing) contient les partages possibles avec d'autres termes. La composante ps est un ensemble de couples de termes dont la forme est indéfinie. Lorsqu'il y a une forme, le partage est implicite. Pour obtenir l'ensemble des partages "réels", nous

[1] Basé sur [MUS90].

avons besoin de la **fermeture** des ps, notée ps^*, qui étend permet de déterminer les répercutions que peut avoir la modification d'un terme sur les autres. Ps^* est aussi une liste de couples de termes mais porte aussi sur les termes possédant déjà une forme, et peut être calculée comme suit :

soit p le nombre de termes de la substitution;

$\forall$ i,j,k : $1 \leq i,j,k \leq p$:

$(ps(i,j) \Rightarrow ps^*(i,j))$ et $((frm(k)=f(...,j,...)$ et $ps^*(i,j)) \Rightarrow ps^*(k,i))$.

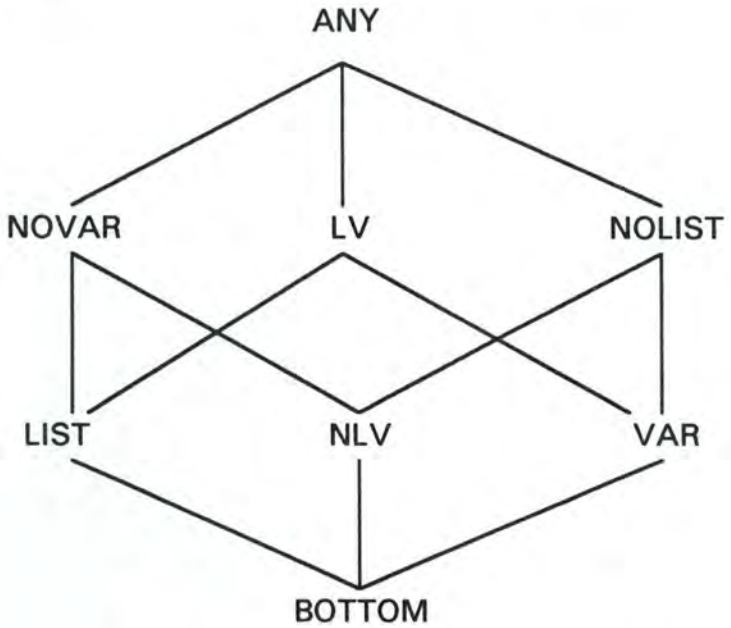
Ainsi, ps^* inclut ps et des partages avec des termes ayant une forme. Il existe des ps forcés qui ne sont pas nécessairement décelés par la méthode ci-dessus, c'est-à-dire qu'un même terme appartient à deux formes différentes, auquel cas le sharing est même obligatoire :

$\forall$ i,j,k : $1 \leq i,j,k \leq p$:

$(frm(i)=f(...,k,...)$ et $frm(j)=g(...,k,...)) \Rightarrow ps^*(i,j))$.

Une substitution abstraite sera donc caractérisée par un quadruplet (sv, tp, frm, ps). La composante sv (same value) exprime des contraintes d'égalité de valeurs de variables : deux variables identiques auront la même valeur de sv. Sv est une fonction surjective de $D \rightarrow [1..m]$ où m est un entier, cette fonction partage l'ensemble des variables en classes de variables identiques. Tp est une fonction de $[1..p] \rightarrow Ty-\{\perp\}$, où p est un entier supérieur ou égal à m correspondant aux nombre de termes de la substitution, et Ty l'ensemble des types. Tp permet d'assigner un type défini ($\neq\perp$) à chaque classe de termes. La composante frm est une fonction de $[1..p] \rightarrow Frm$, qui permet d'assigner une forme à chaque classe de termes. Une forme peut être indéfinie. Ps (possible sharing) est un ensemble de couples de classes de termes. Si un couple (i,j) se trouve dans l'ensemble Ps, cela signifie que les termes des classes i et j pourraient avoir une variable en commun (ce n'est pas obligatoire); i et j désignent des classes de termes dont les formes sont indéfinies, ce sont des entiers inférieurs ou égaux à p. Pour simplifier nos notations, nous défissons le prédicat **ps(i,j)** comme étant vrai si $(i,j) \in Ps$, et faux sinon. Les sharing réels peuvent être obtenus via ps^*. Le quadruplet <sv, tp, frm, ps> constitue le type β et représente une substitution abstraite.

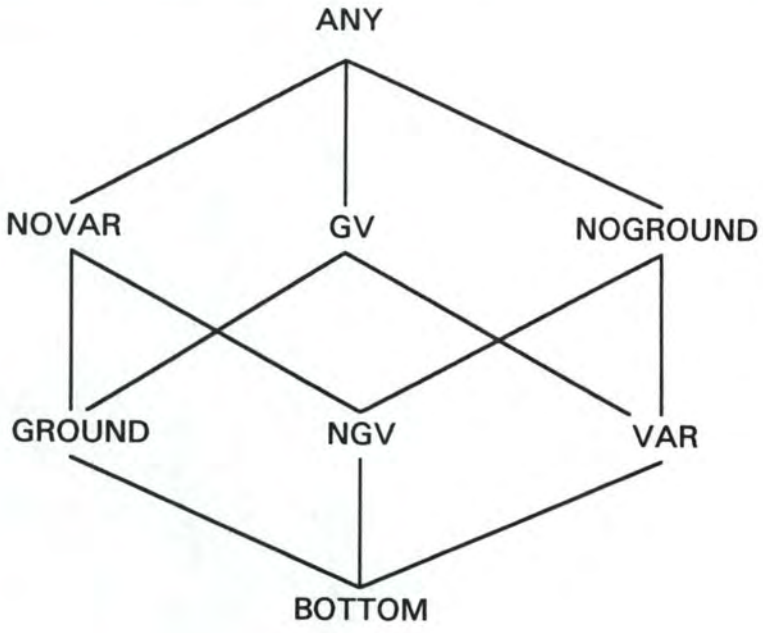
Le domaine des types est fixé au préalable, et permet d'associer un type abstrait à chaque terme. L'algorithme implémenté travaille sur le domaine des listes en prolog, représenté par le schéma suivant :



Ce domaine forme bien sûr un treillis complet : il y a un élément minimal bottom (aussi noté ⊥), une borne supérieure ANY, et chaque élément peut s'exprimer comme une combinaison d'éléments de base (LIST, NLV, VAR). Le système des types est paramétrable : beaucoup d'opérations restent les mêmes quel que soit le système employé. Le mémoire se concentre surtout sur la détermination du domaine d'arrivée du programme. Ce domaine est général, c'est-à-dire que s'il répond LV pour un paramètre d'entrée, cela veut dire que ce paramètre pourra être instancié par une liste ou ne pas être instancié (variable), mais certainement pas par un foncteur.

Un foncteur est prédéfini : cons, le constructeur de liste. Une liste est définie comme en lisp, à savoir qu'elle est soit vide, soit composée d'un élément quelconque puis d'une liste. La constante nil est prédéfinie et représente la liste vide. Pour exemple, la liste [1|2] peut être décrite avec la forme suivante : cons(1(),cons(2(),nil())).

Un autre domaine qui aurait pu être implémenté est le domaine des modes schématisé ci-après, et permet de faire de l'analyse de modes. Quelques modifications théoriques doivent cependant être répercutées aux algorithmes mais les deux domaines sont très semblables; le domaine des listes fut préféré par mon maître de stage, Mr Kaninda Musumbu, qui le trouvait intéressant et ne l'avait pas encore implémenté par opposition au domaine des modes.



Illustrons maintenant la théorie par un exemple, soit la substitution abstraite suivante :

sv : <X1, 1>
<X2, 2>
<X3, 1>
<X4, 3>

tp : <1, VAR>
<2, LIST>
<3, ANY>
<4, ANY>
<5, LIST>
<6, NLV>
<7, LIST>

frm : <2, cons(4,5)>
<3, f(2)>
<5, cons(6,7)>
<7, nil()>

Ps : { (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (5,6), (2,4), (2,5), (2,3) }

Cela signifie que :

- X1 et X3 ont la même valeur, et sont de type VAR; de plus, ils n'ont pas de forme,
- X2 a la forme suivante : cons(ANY,cons(NLV,nil())) où nil est la liste vide, X2 est de type LIST de même que le deuxième argument du premier cons,
- X4 est de la forme f(2) où 2 représente la forme citée précédemment cons(ANY,cons(NLV,nil())),
- Toute classe de terme a un partage possible avec elle-même sauf dans le cas d'une constante (nil()),
- Les classes 5 et 6 ont un partage de variable possible, de même que les classes 2 et 4, 2 et 5, 2 et 3.

Cette substitution sera référencée par la suite sous le symbole ***S1*** et permet de représenter une multitude de substitutions concrètes (en effet, chaque type décomposable peut être instancié par des types plus élémentaires). Voici un exemple de substitution concrète correspondante :

{ X1/T,X2/[[g(A)|[h(A)|[]]],X3/T,X4/f(X2) }

Il est de plus possible d'avoir une relation d'ordre sur les substitutions abstraites portant sur un même ensemble de variables. Soient m le nombre de composantes sv, p le nombre de composantes tp et frm, α le triplet (tp, frm, Ps), β et β' deux substitutions abstraites portant sur un ensemble de variables D. Une substitution abstraite A est plus grande qu'une autre B si elle représente plus de substitutions que B, et si chaque substitution représentée par B peut aussi être représentée par A ($cc(B) \subseteq cc(A)$). La relation d'ordre peut être considérée comme une relation d'inclusion ensembliste. Chaque composante de A doit être plus générale que chaque composante de B. Pour être comparables, deux substitutions abstraites doivent porter sur des mêmes domaines de variables; à savoir que chaque variable présente dans une des substitutions abstraites doit se retrouver aussi dans l'autre substitution abstraite. La plus grande substitution doit comporter au moins autant de variables distinctes que la plus petite, il doit pour cela exister une fonction qui permet d'associer une valeur de sv de B à chaque valeur de sv de A. Le type de chaque terme du plus grand doit être supérieur ou égal au type du terme correspondant dans le plus petit. L'ensemble des types constituant un treillis complet, un

type est plus grand qu'un autre s'il est un ancêtre de cet autre type dans l'arbre des types. Si un type est plus grand qu'un autre, il l'inclut et peut donc représenter plus de choses (exemple : dans le domaine des listes, LV est plus grand que VAR car il permet aussi de représenter les listes). Si une forme est présente dans A, elle doit aussi être présente dans B, mais l'inverse n'est pas vrai. En effet, si A possède une forme indéfinie, alors il peut représenter plus de substitutions puisque sa forme peut prendre n'importe quelle valeur. Néanmoins, si une forme de A a une valeur précise, alors la forme du terme de B correspondant doit être identique, sans quoi A ne saura représenter toutes les substitutions représentables par B. Chaque sharing possible "réel" (obtenu par ps^*) de B doit aussi se retrouver dans A.

4. Algorithmes[1]

Ce sous-chapitre est destiné à expliquer l'algorithme de base dans les détails. En premier lieu, j'explique les opérations de base permettant de manipuler le domaine des types, puis l'algorithme général qui est indépendant du système de types choisi.

Les informations les plus utiles concernent les types des variables. Le système de type doit pouvoir représenter précisément les substitutions concrètes. Pour déterminer le type d'une forme sur base des types de ses arguments, nous avons besoin d'un opérateur de construction : cons. Soit nil la constante représentant la liste vide et cons l'opérateur de construction de liste; cons reçoit un foncteur f, un nombre d'arguments n et n types Ti, et renvoie un type T'. Le type de la forme est $\perp$ si un des arguments vaut $\perp$; list si la forme est la liste vide ou s'il s'agit du constructeur de liste cons avec deux arguments dont le deuxième est une liste. Dans tous les autres cas, le type d'une forme est NLV.

L'opérateur inverse est celui d'extraction : étant donné un foncteur, un nombre d'arguments n (non nul) et le type de la forme, il détermine quels sont les types des arguments. Si le type de la forme est VAR, alors le résultat est $\perp$ (pour chaque argument). Si le type de la forme est LIST et que le foncteur n'est pas cons, alors le résultat vaut aussi $\perp$. Si le type de la forme est LIST et que le foncteur est cons, alors le premier argument est de type quelconque (ANY) et le deuxième est de type LIST. Dans tous les autres cas, chaque argument est de type quelconque (ANY).

[1] Basé sur [MUS90], [LEC90].

Le troisième opérateur est le lub, il permet de trouver la borne supérieure de deux types, à savoir le plus petit ancêtre commun des deux types. A l'opposé, nous avons l'opérateur glb (greatest lower bound) qui trouve le plus grand descendant commun de deux types. Ces plus grands ancêtres ou descendants communs existent toujours grâce à l'existance de bornes supérieure (any) et inférieure ($\perp$).

L'opérateur suivant, MaT (matching abstrait), permet de déterminer le nouveau type d'un terme possédant une forme et dont le type des arguments est modifié, en fonction de son ancien type et du nouveau type de ses arguments. Il s'agit de la borne inférieure de l'instanciation de son type et du type fourni par la construction du foncteur et des nouveaux types.

L'opérateur UaT, lui, permettra de déterminer le type fourni par l'unification abstraite de deux termes. Si un des deux termes vaut $\perp$, alors le résultat est $\perp$. Si un des deux termes est une variable, alors le résultat est le type de l'autre terme. Si un des deux termes est une liste, alors le résultat est une liste. Si les deux types sont nlv, alors le résultat est nlv ou list (donc novar). Si aucun de ces cas ne se présente, alors il faut décomposer un des types non élémentaires en deux autres types qui sont ses descendants directs (c'est faisable car chaque type n'est ni $\perp$, ni VAR, ni LIST, et au moins un des deux n'est pas NLV), et prendre la borne supérieure des unifications abstraites de chaque type obtenu avec le type non décomposé.

L'opérateur suivant, IaT, permet d'effectuer l'instanciation abstraite d'un type, c'est-à-dire de déterminer si le type d'un terme quelconque (sans forme définie) change lorsqu'une substitution est modifiée. Si le type du terme est $\perp$, list ou novar, alors il ne change pas. S'il s'agit de nlv, alors le type devient novar (le terme peut devenir une liste). Dans les autres cas, le type redevient quelconque. Une version spécialisée a été écrite et reçoit comme premier paramètre le type du terme, et comme deuxième paramètre le type du terme spécialisé (auquel on ajoute une forme), calculé indépendamment de son ancien type. Si un des deux types vaut $\perp$, alors le résultat est $\perp$. Si le premier terme est une liste, alors le résultat est list. Si le premier terme est VAR, alors le résultat est la borne supérieure entre VAR et le deuxième type. Si le premier terme est NLV et le deuxième NLV ou VAR, alors le résultat est VAR. Si le premier terme est NLV et le deuxième LIST, alors le résultat est NOVAR. Si aucun de ces cas ne se présente, alors il faut décomposer un des types décomposables et prendre la borne supérieure des instanciations abstraites spécialisées de chaque type trouvé avec le type non décomposé.

Une première version du programme est la version 2 de l'article "Efficient and accurate algorithms for the abstract interpretation of prolog programs", exposée par les procédures pseudo-pascal de l'annexe 1.

La méthode employée travaille de manière descendante et stocke les résultats déjà obtenus en cours de route dans un ensemble *sat* (*s*et of *a*bstract *t*uples). Chacun de ses éléments aura trois composantes : βin, p, βout. p est l'identifiant d'une procédure (composé du symbole de prédicat de la tête et du nombre d'arguments), et βin est une substitution abstraite d'entrée de la clause, c'est-à-dire portant sur les variables présentes dans la tête de la clause. βout représente le résultat obtenu lors de l'exécution de cette clause, il s'agit encore une fois d'une substitution abstraite portant sur les variables de la tête de clause. Cet ensemble sat permettra d'associer à chaque procédure et substitution d'entrée, une substitution donnant un résultat provisoire, résultat qui vaudra ⊥ au départ. Cette substitution résultat sera améliorée à chaque simulation d'exécution de la procédure, jusqu'à ce qu'un point fixe soit obtenu.

La démarche est la suivate : le programme principal initialise les ensembles (qui sont vides au départ), et appelle solve_goal dont l'effet est de résoudre une requête. Chaque résultat temporaire trouvé est stocké dans le sat. Un ensemble suspended contient les demandes de calcul qui sont commencées et pas encore terminées. Si le calcul demandé à solve_goal est déjà commencé, cette procédure s'arrête. Sinon, deux cas se présentent : soit un résultat provisoire a déjà été trouvé pour cette procédure et cette substitution, soit c'est la première fois que ce calcul est lancé, auquel cas il faut ajouter cette substitution d'entréé et cette procédure dans le sat avec ⊥ comme résultat provisoire correspondant. Ensuite il faut résoudre la question, cela se fait par la procédure solve_procedure. La procédure solve_procedure ajoute la procédure et la substitution correspondante à suspended, exécute solve_all_clauses qui va calculer la réponse la plus générale de toutes les clauses pouvant résoudre la procédure. Une des différence entre l'exécution abstraite et pratique est que lors de l'exécution pratique, lorsqu'un résultat est trouvé, le programme peut en général s'arrêter. Lors d'une exécution abstraite, il est indispensable d'exécuter toutes les clauses et de prendre le résultat le plus général possible, pour éviter de tomber sur un résultat qui n'est vrai que dans un cas particulier. La procédure solve_procédure recommence cela jusqu'à ce qu'elle ait itéré une fois sans que le sat n'ait été modifié et que le résultat dans le sat n'ait pas été amélioré. Un booléen same_sat retient si le sat a été modifié lors de la dernière exécution de procédure et est mis à jour en conséquence (si le sat a été modifié, alors il faut relancer le calcul car des éléments nouveaux peuvent influencer le résultat). La procédure solve_all_clauses renvoie un booléen qui indique si le sat a été modifié. Elle appelle la procédure solve_clause pour chaque clause de la

procédure et calcule le lub des substitutions résultat obtenues. Si ce lub n'est pas plus petit que le résultat provisoire du sat, le sat est mis à jour en conséquence. La procédure solve_clause résout la clause en en exécutant chaque sous-but. Suivant le cas du type de sous-but, il exécute AI_VAR, AI_FUNC ou lance solve_goal récursivement pour obtenir une substitution résultat. Lorsque tous les sous-buts sont exécutés, la substitution est restreinte aux variables de la tête de clause et est renvoyée à solve_all_clauses.

A l'entrée de solve_clause, le βin est étendu à toute la clause (chaque variable n'appartenant pas à la tête de clause reçoit une forme indéfinie) via la procédure EXTC, puis, pour chaque sous-but, on restreint la substitution aux variables du sous-but et on renomme les variables de X1 à Xn (procédure RESTRB). Ainsi, un sous-but peut avoir quatre formes :

- "X1=X2" simule l'unification des deux termes. En cas d'échec, l'exécution de la clause entière s'arrête et le résultat renvoyé est ⊥. En cas de succès, X1 et X2 ont la même composante sv, leur type est le *lub* des types des variables respectives.

- "X1=f(X2, ..., Xn)" : si X1 a une forme et qu'elle est différente de celle du deuxième opérande, alors un échec se produit. Si X1 n'a pas de forme, on lui donne la même forme que le deuxième opérande (f(X2, ..., Xn)). Ensuite, on essaie d'unifier chaque argument de la forme de X1 avec l'argument correspondant de la forme de X2. Si un échec se produit, l'exécution de la clause s'arrête en renvoyant comme résultat ⊥. (Dans ces deux premiers cas, on lance l'algorithme de test d'unification).

- "X1 is X2", "X1 is f(X2, ..., Xn)", "X1<X2" : impose que le type des deux opérandes soit NLV (dans le cas du domaine des listes) après exécution de la clause, de même que pour les autres comparaisons (≤, ≥, >).

- "p(X1, ..., Xn)" : dans le cas d'un appel de procédure, il exécute la procédure, et recherche le point fixe de cette procédure, c'est-à-dire qu'il trouve un résultat et l'améliore jusqu'à ce qu'il boucle sur un même résultat. En cas d'échec, l'exécution de la clause entière est arrêté.

Après chaque exécution de sous-but, le programme effectue le changement de variables inverse à celui exécuté avant l'exécution du sous-but, puis exécute une opération d'extension (EXTB), qui a pour but d'effectuer le changement de variable inverse de celui

effectué avant de simuler l'exécution du sous-but, et de propager la substitution résultat obtenue à l'entièreté de la clause, car le résultat obtenu peut avoir des répercutions sur les autres variables présentes dans la clause. Par exemple, si le sous-but permet de déterminer qu'une variable X est une liste et qu'il existe une forme cons(Y,X), le type de ce cons(Y,X) deviendra également LIST, alors qu'il était peut-être NLV auparavant.. Pour cela, il ajoute à la substitution globale les termes de la substitution résultat du sous-but, et unifie chaque valeur de variable (sv) de la substitution résultat avec la valeur de la variable correspondante dans la substitution globale. Après avoir exécuté tous les sous-buts d'une clause, la substitution résultat obtenue est restreinte aux variables présentes dans la tête de clause (RESTRC).

La procédure EXTEND ajoute dans le sat un élément constitué du βin, du prédicat et lui affecte le résultat provisoire ⊥. dom(sat) désigne le sat restreint aux βin et au prédicat. ADJUST améliore un résultat présent dans le sat, en calculant le lub de la substitution présente et de la nouvelle substitution résultat pour chaque βin inférieur ou égal à une substitution donnée. AI_VAR et AI_FUNC exécutent respectivement les routines d'unification abi1 et abi2 décrites dans [MUS90].

La procédure AI_VAR a pour effet de lancer la procédure ualct sur les variables X1 et X2 dans le béta passé à la procédure (il n'y a que ces variables à cause de la restriction). Le nom Ualct signifie "Unification abstraite d'une liste de couples de termes", la procédure reçoit en paramètres une liste d'indices de termes et une substitution. Après unification, les variables X1 et X2 ont la même composante sv en cas de succès.

La procédure AI_FUNC reçoit en paramètres le nombre de variables et un foncteur f, et le but est de simuler l'exécution de X1=f(X2,...,Xn). Pour cela, elle crée un terme dont la forme est f(i2,...,in) où chaque ik=sv(Xk). Le type de ce terme est calculé par la procédure de construction cons à laquelle on passe f ainsi que le type de chaque variable du deuxième opérande. La procédure lance alors Ualct en demandant d'unifier le terme correspondant à X1 avec le terme créé.

La procédure UNION calcule la plus petite substitution β' en généralisant deux autres substitutions β_1, β_2 portant sur un même domaine. Si une des deux substitutions vaut ⊥, alors le réultat de cette opération est l'autre substitution. Pour cela, elle établit une correspondance entre les termes des deux substitutions. Chaque correspondance donnera lieu à un terme de la substitution résultat. Pour cela, elle crée une liste de couples de termes. En premier lieu, cette liste est composée des couples de composantes sv d'une même variable, pour chaque variable. Ensuite on descend dans les formes et agrandit la

liste de couples lorsque les formes des termes sont identiques. Chaque couple de la liste correspondra à un terme dans la substitution résultat. Le type de ce terme sera le lub des types des termes auxquels il correspond. Seules les formes possédant le même foncteur et le même nombre d'arguments seront conservées, et les partages entre les formes indéfinies de la substitution seront créés si un partage réel est possible entre les termes des couples correspondants.

La procédure Ualct exécute l'unification abstraite d'une liste de couples de termes. Elle reçoit en paramètres une liste et une substitution abstraite, et renvoie une substitution abstraite. Pour cela, elle unifie chaque couple de termes l'un après l'autre. Pour unifier deux termes, elle procède comme suit : si les deux termes ont une forme et qu'ele est différente, alors il faut renvoyer un échec. Si les deux termes sont identiques, alors il n'y a rien à faire. Si les deux termes ont une forme indéfinie, l'unification se fait par un appel à la procédure Uact1, après quoi ces termes doivent être fusionnés en un seul. Si un des deux termes a une forme et l'autre n'en a pas, alors il faut spécialiser le terme sans forme, c'est-à-dire lui donner la même forme que l'autre et se ramener au cas où les deux formes sont identiques. Si les deux formes sont identiques, alors il faut unifier les arguments deux à deux par la même méthode. Si aucun échec ne se produit; alors il faut fusionner les deux termes en un seul.

La procédure uact1 unifie un couple de termes dont les formes sont indéfinies; elle reçoit une substitution α et deux termes à unifier, et renvoie une substitution α'. La composante sv n'est pas modifiée. En ce qui concerne les termes, chaque type doit être recalculé. Pour un terme donné, s'il n'y a pas de sharing réel entre ce terme et un des termes unifiés, alors le type ne change pas. Les types des termes unifiés est déterminé par l'unification abstraite des deux types. Si un terme possède une forme, son type peut être recalculé par l'opérateur de matching abstrait MaT en fonction de son type actuel, de son foncteur et des nouveaux types des arguments. Si le terme n'a pas de forme, il est recalculé par l'opérateur d'instanciation abstraite IaT. Les formes restent identiques. Dans les ps, il faut garder seulement ceux pour lesquels les termes correspondent à des types différents de $\perp$. Si le nouveau type est différent de $\perp$, alors on ajoute les ps pouvant être déduits par le fait que les deux termes sont identiques.

La procédure spécat permet de spécialiser un terme étant donné un premier terme i dont la forme est indéfinie, un second terme j dont la forme est définie et une substitution abstraite. Cette procédure renvoie une substitution abstraite où, en cas de succès, le terme à la forme indéfinie du départ possède la même forme que l'autre terme. Elle renverra $\perp$ au cas où une incohérence est constatée entre le foncteur et les types. Elle crée un terme pour

chaque argument du nouveau foncteur dont le type est déterminé par l'opérateur extr. Si un des types vaut $\perp$ et que le type de i n'inclut pas VAR, alors il faut renvoyer $\perp$. Si i est accessible à partir de j (il appartient à sa forme ou peut être atteint à partir de la forme d'un des arguments de j), il faut aussi renvoyer $\perp$. Il faut recalculer le type de chaque terme. Pour chaque terme, s'il n'y a pas de partage réel entre lui et i ou si ce terme est accessible à partir de j, alors le type ne change pas. Soit T le type calculé par l'opérateur de matching abstrait auquel on donne le type de i, le foncteur et les types des termes créés pour les arguments. Si le type de i n'inclut pas VAR, son type devient T; sinon il devient le lub de T et du type fourni par l'opérateur de construction cons auquel on donne le foncteur et on spécifie une variable pour chaque argument. Pour les autres termes, leur type est déterminé par l'opérateur de matching MaT s'ils ont une forme. Si leur forme est indéfinie, le type n'est pas modifié s'il n'inclut pas var; sinon le noveau type est déterminé par la version spécialisée de l'opérateur d'instanciation abstraite IaT2 auquel on donne le type du terme avant modification et le type généré par l'opérateur de construction cons avec le foncteur de j et VAR pour le type de chaque argument. En ce qui concerne les termes créés correspondant aux argument de i, si le type de i inclut VAR, ils sont aussi modifiés pour inclure VAR. Une forme est ajoutée pour i. En ce qui concerne les partages, il faut enlever tous les anciens partages de i puisque i a reçu une forme. Chaque terme qui avait un partage avec i et dont le nouveau type est différent de $\perp$ recevra un partage avec chaque argument de i dont le type est différent de $\perp$. Si le type de i inclut VAR, chacun de ses arguments recevra un sharing avec lui-même. Si le type calculé initialement pour les arguments est différent de $\perp$, alors tous les arguments dont les types sont différents de $\perp$ recevront des sharings entre eux.

La seconde partie va détailler sur l'implémentation du système, des algorithmes, des données et donner quelques exemples où ça marche ainsi que des exemples où le résultat est correct mais pas assez précis et indiquer pourquoi.

Partie II :

Le système implémenté

Introduction

L'implémentation d'un tel système n'est pas chose aisée; néanmoins, elle a été faite dans un but de simplicité de maintenance du programme. Des méthodes efficaces mais compliquées n'ont pas été implémentées pour cette raison.

En premier lieu, je vais expliquer la manière dont les substitutions abstraites ont été représentées en mémoire. Plusieurs méthodes sont possibles, et j'en citerai deux. Ensuite, je m'attarderai sur la normalisation des programmes prolog et sur la manière dont les programmes sont représentés en mémoire. Après quoi j'expliquerai l'implémentation de diverses procédures sur le domaine et la manière dont les algorithmes génériques ont été implémentés. Je cite aussi quelques améliorations possibles même si elles n'ont pas encore été implémentées. Après quoi, je citerai quelques exemples d'exécution du programme.

Afin de faciliter la maintenance de ce programme, toutes les routines ont été regroupées dans des modules suivant un critère de cohésion. Les petites fonctions d'unification, de restriction et d'extension ont été regroupées dans un même module. Les fonctions de manipulation du système de types ont aussi été regroupées dans un même module, de même pour les fonctions manipulant les ps, les sv, les ensembles en général, les comparaisons de substitutions (β).

Chapitre I :
Implémentation du domaine

Ce chapitre va discuter les choix d'implémentation choisis en ce qui concerne les domaines, c'est-à-dire la représentation des substitutions abstraites et du système de types. Il présente les avantages et les inconvénients de chaque méthode. Une première implémentation est basée sur des tables, une deuxième sur des listes chaînées et des pointeurs. La substitution abstraite ***S1*** citée précédemment sera représentée en exemple par chaque méthode.

1) Tableaux

Une substitution abstraite est une structure de type *beta* qui possède quatre entrées : <sv,tp,frm,ps>. La composante *sv* sera implémentée comme un tableau de couples <x_i, sv_i>, et une variable *indice* contiendra le nombre d'entrées de ce tableau. Le principe est le même pour le triplet <tp,frm,ps> : un tableau de structures *t* contiendra les données, et une variable p contiendra le nombre d'entrées de ce tableau. sv est un tableau de structures de type *tsv*, type qui comprend deux éléments : le string *xi* contenant x_i, et un entier *sv* comprenant la valeur sv correspondante. Cette valeur est l'indice du triplet associé dans t. t est un tableau de structures de type *triple*, où chaque élément contient un entier *tp* qui est le type correspondant dans le système de type, un string *functor* qui contient le foncteur, un entier *nbeltsfrm* qui contient le nombre d'éléments dans la forme (arité du foncteur), un tableau d'entiers *tabfrm* qui contient les indices des arguments du foncteur (il s'agit des indices des triplets correspondants aux arguments dans t), un entier *nbeltsps* et un tableau d'entiers *tabps*. Cette représentation a semblé compliquée pour l'équipe de Bordeaux, qui trouva qu'une implémentation via pointeurs et listes chaînées était beaucoup plus simple.

Représentation des ps

La composante Ps(i,j) est représentée si et seulement si $i \leq j$. Si ce n'est pas le cas, Ps(j,i) est représentée. On ne perd aucune information puisque Ps(i,j) est

équivalent à Ps(j,i). La composante nbeltsps du i-ème triple de *t* contient le nombre de j tels que i ≤ j et Ps(i,j). La composante *tabps* est un tableau d'entiers trié en ordre strictement croissant contenant tous ces j. Ainsi, pour tester si Ps(i,j) est vrai dans un β, il suffit de swapper les variables au cas où i > j, et de regarder si j appartient au tableau *tabps* de la i-ème composante de t. Puisque ce tableau est trié, il est possible de faire une recherche dichotomique. Cela donne un accès direct aux Ps, qui sera perdu avec l'implémentation via pointeurs et listes chaînées.

Voici les définitions C des structures de données correspondantes :

```
#define TOPTREE        8
                       /* ANY + 1 */
#define ANY            7
#define NOVAR          6
#define LV             5
#define NOLIST         3
#define LIST           4
#define NLV            2
#define VAR            1
#define BOTTOM         0

typedef int type_tp

typedef struct
        {      char *xi;
               int sv;          }      tsv;

typedef struct
        {      type_tp tp;
               char *functor;
               int nbeltsfrm;
               int *tabfrm;
               int nbeltsps;
               int *tabps;      }      tclasse;

char *typestring[TOPTREE];
```

```
typestring[0]=strdup("bottom");
typestring[1]=strdup("var");
typestring[2]=strdup("nlv");
typestring[3]=strdup("nolist");
typestring[4]=strdup("list");
typestring[5]=strdup("lv");
typestring[6]=strdup("novar");
typestring[7]=strdup("any");
```

Exemple

Le schéma ci-après représente la substitution abstraite ***S1*** citée précédemment.

Attention : les indices commencent à zéro. Toutes les références dans les formes (tabfrm), les ps (tabsv) ou les sv (sv) sont des indices du tableau t, valant de 0 à p-1. De même, la valeur de tp est un indice d'un tableau de types, allant de 0 à la valeur prédéfinie TOPTREE qui vaut le nombre de types du système de types choisi (soit #Ty).

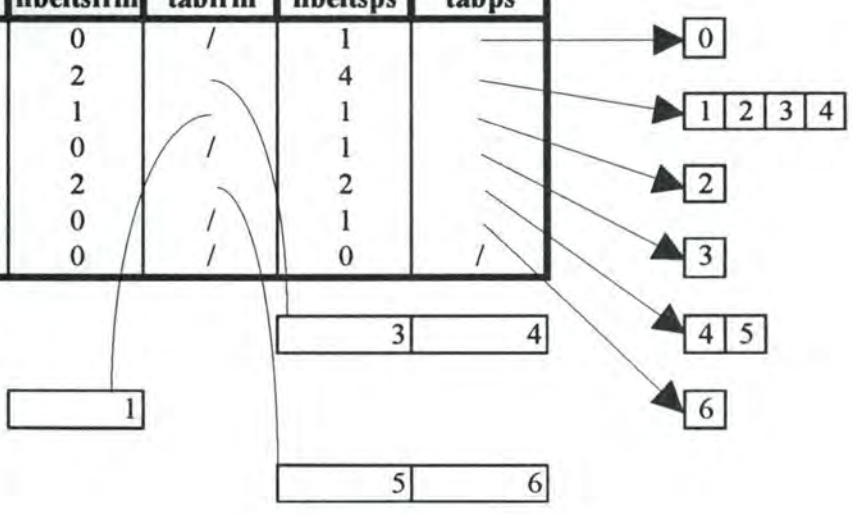
sv :

Xi	SV
X1	0
X2	1
X3	0
X4	2

indice : 4

t :

tp	functor	nbeltsfrm	tabfrm	nbeltsps	tabps
VAR	/	0	/	1	
LIST	cons	2		4	
ANY	f	1		1	
ANY	/	0	/	1	
LIST	cons	2		2	
NLV	/	0	/	1	
LIST	nil	0	/	0	/



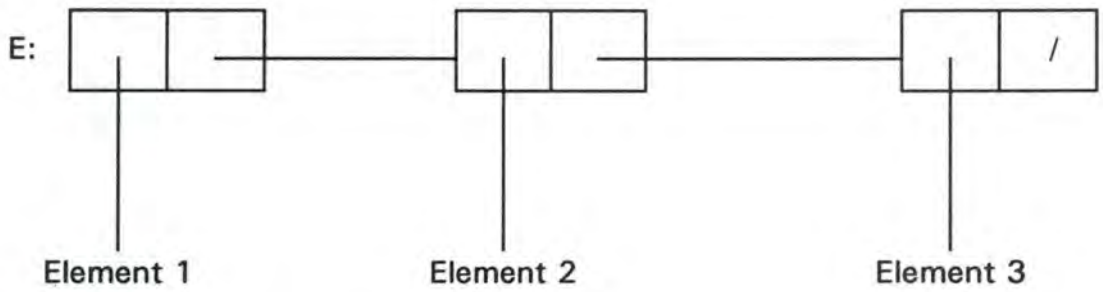
p : 7

Le gros inconvénient de cette méthode est que l'ajout d'une entrée à une table est effectué par un "realloc", qui en général recopie la table entièrement dans une nouvelle zone allouée dynamiquement. Enlever un élément se fait en décalant les éléments qui le suivent vers le bas. Cela rend le programme inefficace. Pour remédier à cela, il faudrait une implémentation plus efficace des primitives d'allocation de mémoire. De plus, on ne peut insérer un terme n'importe où sans changer la quasi totalité des références aux termes.

2) Listes chaînées

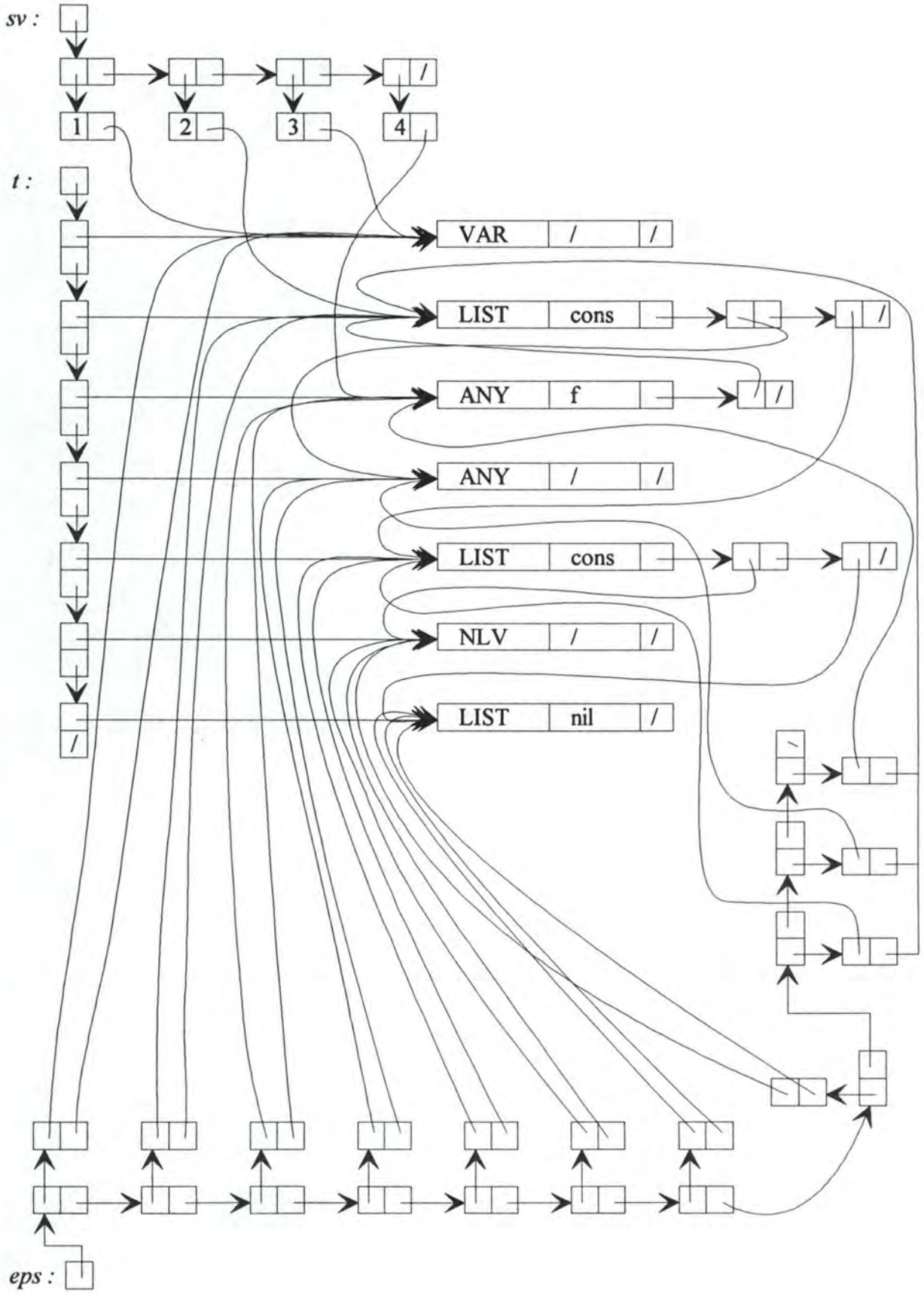
La représentation suivante est basée sur la manipulation d'ensembles. Un ensemble sera représenté par deux pointeurs : un pointeur vers une valeur d'élément et un pointeur vers l'élément suivant de l'ensemble.

Représentation de l'ensemble E :



Cette représentation a des avantages : chaque élément est indépendant de l'ensemble auquel il appartient, et une suppression ou un ajout est peu coûteux. L'inconvénient est la perte de l'accès direct. La composante sv de l'ensemble sv sera représentée par un pointeur vers une classe. Un β sera représenté par trois ensembles : sv, t, eps où t est une liste chaînée de classes et eps un ensemble de paires de pointeurs de classes. Chaque classe est caractérisée par un type, et une forme. La forme est composée d'une chaîne de caractères pour le foncteur, et d'un ensemble de classes pour ses arguments, classes appartenant aussi à t. Les types ont été représentés d'une manière plus astucieuse sur base de l'assertion que le domaine forme un treillis complet. La substitution $S1$ peut être représentée de cette manière comme le montre le schéma ci-après.

Le gros inconvénient est la perte de l'accès direct aux ps, mais chaque élément devient indépendant de sa position dans l'ensemble.



Il existe toutefois une troisième manière de représenter les Ps qui permet dans certains cas de gagner de la place mais qui est plus complexe à manipuler en ce qui concerne, par exemple, l'ajout et la suppression de classes. Les ps peuvent être représentés par une fonction booléenne à deux arguments, et il est très possible de numéroter les classes de 0 à p-1 où p est le nombre de classes. Les ps peuvent être représentés par un champ de p^2 bits : pour stocker ps(i,j), il suffit de mettre le (i*p+j)-ème bit à 1 (comme s'il s'agissait d'un tableau à deux dimensions). Le champ est bien entendu initialisé à 0. Les problèmes se posent lors d'ajouts ou de suppressions de classes. Cette méthode a ses avantages et ses inconvénients et aurait pu être implémentée. Elle offre l'avantage de l'accès direct aux Ps, et en général elle occupe moins de place mémoire; par contre, l'ajout et la suppression d'un élément dans l'ensemble n'est pas aisée. Les listes chaînées nous permettent de rendre les éléments indépendants de leur position dans l'ensemble, et cette représentation rend à nouveau l'élément dépendant vis-à-vis de sa position. Plus efficace du point de vue du temps calcul mais pas du point de vue de l'occupation en mémoire, nous pouvons remplacer le champ de bits par un champ de bytes, selon des tests de performances réalisés dans un mémoire antérieur. L'ensemble des ps de ***S1*** peut être représenté par le tableau suivant, et il est intéressant de remarquer que seule la partie inférieure doit être mise à jour du fait que la fonction est commutative (on peut ne représenter que la moitié du tableau car Ps(i,j) $\Leftrightarrow$ Ps(j,i)) :

1	0	0	0	0	0	0
0	1	1	1	1	0	0
0	1	1	0	0	0	0
0	1	0	1	0	0	0
0	1	0	0	1	1	0
0	0	0	0	1	1	0
0	0	0	0	0	0	0

Cette méthode, quoique beaucoup plus efficace, n'a pas encore été implémentée. Il faut faire attention à ne pas perdre tous les gains de temps potentiels à faire correspondre les positions dans l'ensemble et les valeurs de pointeurs (ps[3][2]=1 veut dire qu'il y a ps entre le troisième élément de l'ensemble et le deuxième), car les ps nous donnent des informations sur les éléments en fonction de leur position, mais nous n'avons pas d'accès direct au i-ième élément de l'ensemble.

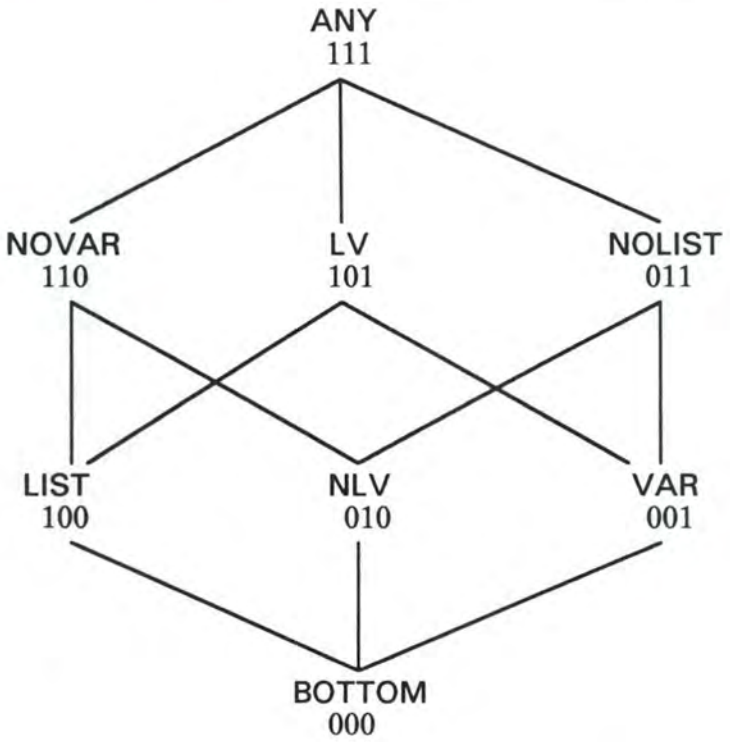
Les structures en C permettant de manipuler ces données :

```
typedef char type_tp;

typedef struct ens
    {   void *element;
        struct ens *next;   }   type_element;

typedef type_element* type_ensemble;

typedef struct
    {   char *functor;
        type_ensemble arguments;   }   type_forme;

typedef struct
    {   type_tp tp;
        type_forme forme;   }   classe;

typedef struct
    {   int xi;
        classe *sv;   }   type_sv;

typedef struct
    {   void *e1,*e2; }   paire;

typedef struct
    {   type_ensemble sv;
        type_ensemble t;
        type_ensemble eps;   }   beta;
```

Représentation des types

Etant donné que les types à représenter constituent un treillis complet, ils furent implémentés comme des entiers (ou plutôt comme suites de bits). Le type Bottom vaut 0, et chaque type de base constitue une puissance de 2. Pour obtenir le lub de deux types, il suffit par exemple de faire un *or* au niveau du bit sur les deux

types. (Par symétrie, un *and* pour obtenir le glb). Un noeud a est un ancêtre d'un autre b si et seulement si a *and* b = b (comparaison au niveau du bit). (cfr schéma).



Il est aussi facile de connaître les descendants directs d'un noeud (mettre à 0 un des bits à 1) ou les parents d'un noeud (mettre à 1 un des bits à 0). Ainsi, chaque type peut être représenté par un byte.

Chapitre II :

Normalisation et représentation interne des programmes

Ce chapitre est consacré à expliquer la manière dont la normalisation est implémentée, ainsi que la représentation des programmes en mémoire.

Normalisation des programmes

Tout d'abord, la syntaxe du programme va être vérifiée par lex et yacc, qui vont le réécrire de telle manière qu'il n'y aura plus de blancs superflus, de ':-', que chaque clause sera représentée sur une ligne et que les sous-buts seront séparés par un blanc. A chaque constante seront ajoutées deux parenthèses, les '[]' seront remplacés par 'nil()', le '[' sera remplacé par 'cons(', le ']' par ')', et le | par une virgule. Les "cut" ('!') seront enlevés du fichier car l'algorithme ne les traite pas.

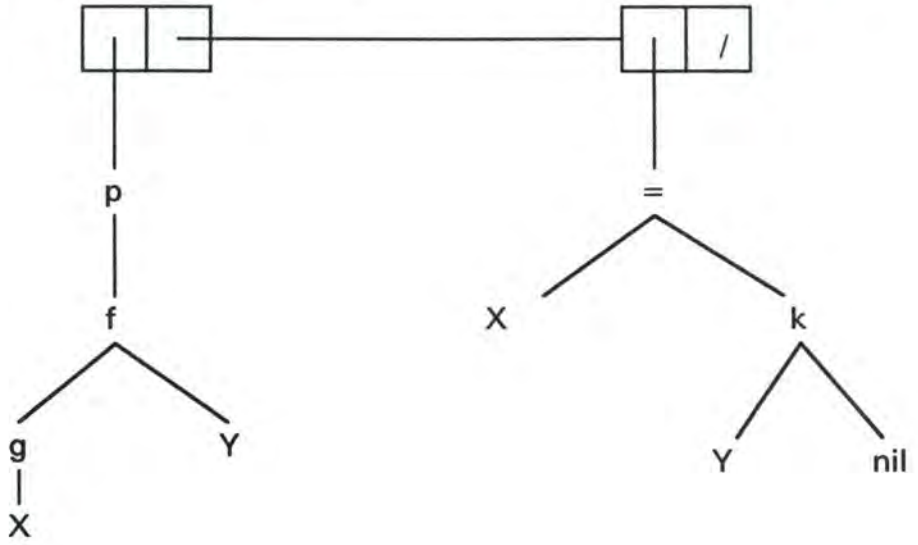
Ensuite le programme réécrit va être lu et chaque clause va être stockée sous forme d'arbres binaires. Exemple : soit à normaliser la clause suivante :

p(f(g(X),Y)) :- X=k(Y,[]).

Cette clause va d'abord être vérifiée puis réécrite de manière à éliminer les particularités ([]) et les signes superflus (,), ce qui va donner :

p(f(g(X),Y)) X=k(Y,nil())

La clause va d'abord être découpée en un ensemble de sous-clauses (deux dans l'exemple), ce qui va donner ceci :



La normalisation va se faire clause par clause, et en trois passes pour chaque clause : une première étape va détecter les opérations contenant des opérateurs infixés (+, -, *, mod, exp, ^) ou préfixés sans parenthèses (moins unaire), les stocker sous forme d'arbre, évaluer les sous-expressions constantes et réécrire cet arbre sous forme préfixée, et dans lequel les opérations constantes seront remplacées par des constantes (Exemple : "6()+3()" sera remplacé par "9()"). La deuxième passe va détecter les variables et les numéroter (afin de les remplacer par des entiers); elle va ensuite aplatir les expressions en rajoutant des variables au besoin. Dans la mesure du possible, la normalisation sera préfixée, dans le sens où X=g(h(Y,Z)) sera normalisé en X3=h(X1,X2), X4=g(X3) et pas l'inverse. La troisième passe va scanner la tête et chaque sous-but de la clause de telle manière qu'une même variable ne soit pas deux fois dans la même clause (l'appel p(X1,X1,X2,X3) pourrait poser des problèmes de changement de variable), et ensuite numéroter les variables de la tête de clause de telle sorte qu'elle soit de la forme p(X1,X2,...,Xn) via échange de numéros de variables respectif à travers la clause. Après une telle normalisation, cinq types d'atomes sont possibles :

- ABI : soit deux variables, et un symbole de comparaison (=,<,is,...).
  Exemple : X=Y.

- ABI2 : soit une variable, une chaîne de caractères pour le foncteur, et un ensemble (liste chaînée) pour les arguments et un symbole de comparaison.
  Exemples : X=f(Y,g(Z)), X=[], X=[Y|Z].

- ABI3 : (soit une clause de la forme foncteur = foncteur). Elle est caractérisée par un symbole de comparaison, deux chaînes de caractères, et deux

ensembles d'arguments. Ce type de clause sera éliminé par la normalisation et remplacé par une clause de type ABI et deux clauses de type ABI2.

Exemples : f(X)=f(g(Z)), ln(X) is 3.

- PROC : (appel de procédure). Un appel est caractérisé par une chaîne de caractères, et un ensemble d'arguments.

Exemple : append(X5,X2,X6).

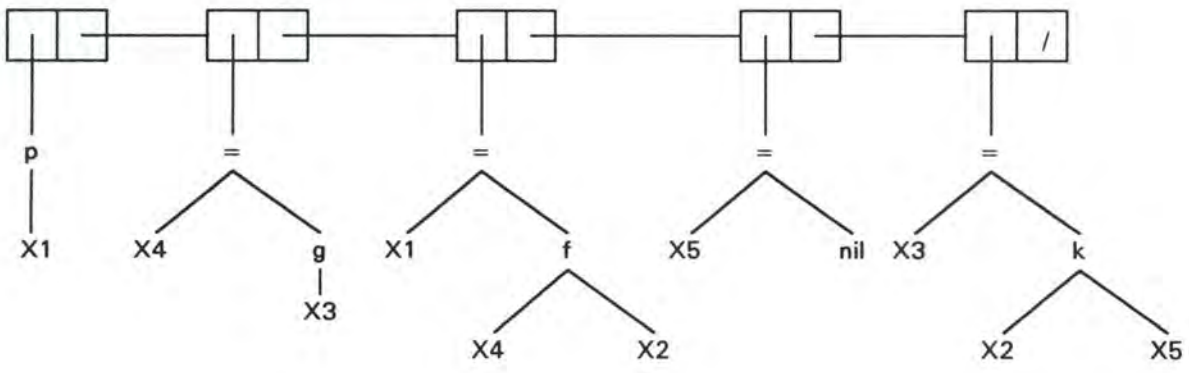
- VARIABLE.

Exemples : X,Y.

Après la normalisation, le type ABI3 est éliminé; de plus, les paramètres de PROC et d'ABI2 sont uniquement des variables, représentées en mémoire par des entiers. Chaque atome ne contient que des variables différentes. De plus, à chaque appel de procédure, on adjoindra un ensemble contenant les clauses dont la tête matche l'appel. Le fichier normalisé sera sauvé sous le nom "normed.txt".

Il est aussi possible de représenter les clauses du programme en employant les ensembles : un programme est un ensemble de clauses. Une clause est un ensemble de sous-buts, dont le premier est la tête de clause. Chaque sous-but normalisé est représenté comme suit : s'il s'agit d'un atome de type ABI, le sous-but est composée de deux variables. Si son type est ABI2, ce sera une variable, un foncteur, et un ensemble de variables pour les paramètres du foncteur. S'il s'agit d'un appel de procédure, il y aura un string et un ensemble de variables pour les paramètres.

La normalisation de la clause ci-dessus va donner un ensemble d'atomes normalisés schématisé ci-dessous :



Les arbres constituant les arguments des foncteurs (X1, X3, X4 et X2, X2 et X5) sont aussi des listes chaînées.

Ensuite, pour chaque appel de procédure (et pour l'appel de procédure initial), on lui adjoint l'ensemble des clauses matchant cet appel. Cela permettra d'accéder directement aux clauses nécessaires lors de l'exécution abstraite d'un appel de procédure prolog. Donc, chaque procédure se verra associer une liste chaînée des clauses qui la composent. Les routines concernant cette normalisation furent regroupées dans le module "norm.c". Les structures de données utilisées sont exposées ci-dessous :

```
typedef struct
    {   char *v;
        type_ensemble e;
        type_ensemble matching;  }   TPROC;

typedef struct
    {   int v;  }   TNVAR;

typedef struct
    {   char typ_egal;
        int v1,v2;   }   TNABI;

typedef struct
    {   char typ_egal;
        int v1;
        char *v2;
        type_ensemble e;   }   TNABI2;

typedef struct
    {   char typ_egal;
        char *v1;
        type_ensemble e1;
        char *v2;
        type_ensemble e2;   }   TABI3;
```

```
typedef struct
    {       char type;
            union
                    {       TNABI  abi1;
                            TNABI2 abi2;
                            TPROC app;
                            TABI3 fun;
                            TNVAR v;    }       ne;
                                                    } Nnoeud_ex;
```

Chapitre III :
Implémentation des algorithmes sur le domaine

Ce chapitre, comme son nom l'indique, est destiné à expliquer comment les algorithmes de base ont été implémentés, ce qui y a été ajouté, modifié, et pourquoi.

Les fonctions de manipulation du domaine de base (MaT, UaT, ...) ne posent pas de problèmes : ce sont des petites fonctions C traduites des définitions. Les opérateurs lub et glb ont été implémentés simplement par un 'or' ou un 'and' au niveau du bit, cela grâce à la représentation des types de base comme des puissances de 2 et grâce au fait que le domaine des listes constitue un treillis complet.

L'implémentation des algorithmes dépend en grande partie de la représentation en mémoire du domaine utilisé, car celle-ci dernier peut faciliter grandement l'implémentation des algorithmes. L'implémentation du domaine fut présentée au chapitre précédent.

Lors des fonctions uact1 et spécat, il est nécessaire de connaître l'ancien type des termes pour en calculer le nouveau. C'est ainsi que les nouveaux types sont stockés dans une table à part tant qu'ils ne sont pas tous calculés. Il y a néanmoins un avantage : la table est initialisée avec des valeurs de type valant 0xFF, étant considéré comme "type pas encore calculé", ce qui a pour avantage qu'un type ne sera calculé qu'une et une seule fois, même si le terme appartient à plusieurs formes. Une manière d'éviter la récursivité pour le calcul des types aurait été la suivante : calculer tous les types calculables sans récursivité, puis calculer tous les types déductibles à partir des types déjà calculés, et ainsi de suite. L'avantage est qu'il n'y a pas de pile, mais il faut à chaque itération vérifier si un type non calculé est calculable, ce qui fait que le gain est très minime. La récursivité fut souvent éliminée et traitée "manuellement", c'est-à-dire gérée avec des piles et des "gotos" car il était parfois intéressant d'accéder à la pile. Les "gotos" ont été remplacés par des automates d'états finis car ils n'étaient pas très appréciés à Bordeaux, du fait qu'ils rendaient la maintenance compliquée. Pour exemple, je vais expliquer l'opération Ualct, qui unifie une liste de couples de termes.

Sa définition théorique est la suivante :

Soit l la liste, δ la substitution;
si l est vide, $\delta'=\delta$;
sinon $l=(i,j).l_0$, on pose $\delta'=Ualct(l_1,\delta_1)$ où l_1,δ_1 sont définis comme indiqué ci-dessous :

- **i=j (fi)**
  $l_1=l_0$, $\delta_1=\delta$.

- **(i≠j et frm(i)=ind=frm(j) (fi))**
  $l_1=l_0$, $\delta_1=Uact1(i,j,\delta)$.
- **(i≠j et (frm(i)≠ind ou frm(j)≠ind) (fi))**

  $\delta'=\perp$ dans les cas suivants :
  - frm(i)=ind (fi) et Spécat(i,j,δ)=$\perp$,
  - frm(j)=ind (fi) et Spécat(j,i,δ)=$\perp$,
  - frm(i)=f(i1,...,in) et frm(j)=g(i1,...,im) (fi) et (f≠g ou m≠n).

  En-dehors de ces cas, on pose
  $l_1=l_0$, $\delta_1=Fcta(i,j,\delta)$ si (frm(i)=frm(j) (fi)),
  sinon,
  $\delta_1=Spécat(i,j,\delta)$ si (frm(i)=ind (fi)),
  $\delta_1=Spécat(j,i,\delta)$ si (frm(j)=ind (fi)),
  $\delta_1=\delta$.

  Dans tous les cas de figure, il existe un foncteur f tel que :
  $frm_1(i)=f(i1,...,in)$ et $frm_1(j)=f(j1,...,jn)$ (fi_1),
  On pose :
  $l_1=(i1,j1).(i2,j2). \ldots .(in,jn).(i,j).l_0$.

La liste a été implémentée comme une liste chaînée. L'algorithme implémenté travaille comme suit : tant que la liste n'est pas vide, alors il unifie les deux premiers termes. Cette unification peut rajouter des couples de termes en début de liste. Lorsque l'on essaie d'unifier deux foncteurs, il faut d'abord unifier ses arguments deux à deux, on ajoute donc la liste des couples d'arguments à unifier en tête de liste, après quoi on insère un élément vide dans la liste. De cette manière, lorsque la procédure ualct rencontrera un terme vide, elle saura qu'il faut unifier les deux termes suivants et les enlever de la liste (il

s'agira des deux foncteurs). En procédant ainsi, la liste sert aussi de pile. Une deuxième chose à laquelle on doit faire attention est la fusion : lorsque l'on fusionne deux termes, on libère l'espace occupé par un des deux et on remplace toutes les références au terme libéré par des références au terme restant; que ces références soient dans la substitution abstraite ou dans la liste. Les pointeurs facilitent grandement la tâche. Le listing se trouve en annexe, module o_operat.c, fonctions ualct et uactgen.

Le deuxième exemple sur lequel je vais m'étendre est la procédure chargée de tester si une substitution abstraite est plus petite qu'une autre selon la relation d'ordre choisie. Les substitutions abstraites doivent tout d'abord porter sur un même domaine. Il faut en premier lieu créer une fonction qui, à chaque terme correspondant à une variable de la plus grande substitution, fait correspondre le terme associé à la même variable dans la plus petite substitution. Cela se fait en créant la table de la fonction sous forme d'une liste chaînée car nous allons y ajouter des termes. Il faut vérifier que c'est bien une fonction lorsque l'on crée la correspondance. Chaque terme associé à une variable de la plus petite substitution abstraite doit se trouver dans l'ensemble d'arrivée de la fonction. Ensuite, il faut vérifier que chaque forme de la plus grande substitution se trouve dans la plus petite, ajouter les arguments deux à deux à la table en vérifiant qu'il s'agit toujours d'une fonction. Après quoi il faut déterminer les partages réels du plus petit et vérifier que chaque ps réel de la plus petite substitution entre des éléments de l'ensemble d'arrivée de la fonction dont les éléments correspondent à des formes indéfinies de la plus grande substitution, est un ps de la plus grande substitution. Il faut ensuite vérifier que le type de chaque terme de la plus petite substitution est inférieur ou égal au type du terme correspondant dans la plus grande. Le listing se trouve en annexe, module betaordr.c, fonction betainf.

La fonction fcta de la thèse n'a pas été implémentée. Une autre fonction, remover, est lancée à la fin de chaque unification d'ualct (par opposition à la thèse), et enlève un des termes unifiés du β courant.

Comme exemple de traitement manuel de la récursivité, l'organigramme de spécat se trouve en annexe. La récursivité a été tranformée avec des "gotos" et les gotos ont été remplacés par des automates d'états finis.

Chapitre IV:
Implémentation des algorithmes génériques

Cette partie est destinée à expliquer quels ont été les changements effectués sur l'algorithme de base (Annexe 1). De plus, il expose quels seraient des améliorations possibles sans pour autant que ces optimisations aient déjà été implémentées.

Chaque ensemble sera représenté par une liste chaînée comme pour l'implémentation du domaine. La structure suivante est ajoutée pour désigner un élément du sat :

```
typedef struct
    {       beta bin;
            char *p;
            int arite;
            beta bout;      }       tsat;
```

Une première "optimisation" pouvant être réalisée est la suivante : le βin étant passé par valeur, mais n'étant modifié par aucune procédure, il peut être passé par adresse. L'ensemble sat étant passé en "inout" à chaque procédure, il sera également passé par adresse. En ce qui concerne l'ensemble suspended, ses éléments ne sont jamais modifiés, mais solve_procedure le modifie avant d'appeler solve_all_clauses. Plutôt que le passer par valeur et le recopier lors de chaque appel à solve_all_clauses, il est plus intéressant de le passer par adresse, mais il faut pouvoir revenir à la valeur de l'ensemble avant l'appel. Les listes chaînées vont simplifier la tâche : la seule chose que l'on change à suspended, c'est lui ajouter un élément lors de l'appel à solve_all_clauses. Il est aisé d'ajouter l'élément en tête de liste (avant le repeat), et de l'enlever après le until. Une deuxième amélioration est de considérer que suspended est un sous-ensemble du sat étant donné que chaque ajout dans suspended correspond à un élément déjà basé dans sat (c'est-à-dire dans dom(sat)). L'ensemble suspended sera donc représenté par une liste chaînée de certains éléments du sat.

La procédure solve_clause peut également être optimisée : si un des paramètres de RESTRB, EXTB, RESTRC est $\perp$, alors le résultat vaut $\perp$. Donc, dès qu'un $\perp$ est détecté, nous pouvons directement sortir de la boucle for pour renvoyer $\perp$, sans continuer de simuler l'exécution de chaque sous-clause.

Des améliorations supplémentaires peuvent être apportées à cet algorithme quoiqu'elles n'aient pas encore été implémentées. Les exécutions d'une clause avec une substitution d'entrée identique ne commenceront à différencier que lors du premier appel de procédure, c'est-à-dire que l'exécution des premiers built-in sera identique. Il serait intéressant de stocker le résultat après l'exécution de ces premiers built-in pour ne pas les recalculer. Il y a encore moyen de faire mieux : si la première procédure a atteint un point fixe lors de la première exécution de la clause, il est possible que ce point fixe soit correct et non améliorable, c'est-à-dire que la réexécution de cette procédure n'apprendrait rien de nouveau. Pour le savoir, il faut représenter les dépendances sur un graphe et il est alors possible de déterminer si le point fixe d'une procédure peut être amélioré. En général, s'il n'y a pas de récursivité indirecte, la procédure peut être exécutée indépendemment du calcul en cours et le résultat obtenu lui est propre, et ne saurait donc être amélioré. Donc, certains appels de procédure ne devraient plus être exécutés, et même certaines clauses entières. Les exécutions des clauses d'une même procédure pourraient être faites en parallèle afin de gagner du temps. La version du programme présentée plus haut réexécute chaque clause, il serait intéressant de savoir quand le résultat obtenu ne peut plus être amélioré, et donc quand il suffit de prendre le résultat obtenu plutôt que de relancer le calcul (détection des éléments définitifs du sat).

Il a néanmoins fallu tordre légèrement la théorie afin de faire fonctionner le programme dans tous les cas, car ces algorithmes ont des limites inhérentes au domaine et il serait très coûteux de les éliminer. Néanmoins, pour le comprendre, une bonne compréhension du comportement général du programme est nécessaire. C'est pourquoi je m'attarderai à expliquer son comportement via un exemple sur la procédure append permettant d'effectuer la concaténation de listes, qui, une fois normalisée, s'écrit comme suit :

```
append(X1, X2, X3) :-   X1=nil(), X2=X3.
append(X1, X2, X3) :-   X1=cons(X4, X5), X3=cons(X4, X6),
                        append(X5, X2, X6).
```

Supposons que l'on exécute cette procédure avec le βin suivant :

```
x1 1
x2 2
x3 3
1 var
2 var
3 list
```

Le programme s'exécute alors comme suit : l'exécution de la première clause est simulée avec le βin comme substitution d'entrée, ce qui donne comme résultat LIST pour X1, X2 et X3. La deuxième clause est exécutée, aussi avec le βin comme substitution d'entrée. Les deux premiers sous-buts permettent de déterminer que X1 est NLV, X2 est VAR, X3 est LIST, X4 est ANY, X5 est VAR et X6 est LIST (cons (ANY,LIST)=LIST). Lorsqu'on lance append(X5,X2,X6), on essaie donc de réexécuter append avec VAR, VAR, LIST dans trois classes différentes comme substitution d'entrée. La simulation d'exécution de la clause s'arrête car ce calcul a déjà été commencé et n'est pas encore terminé, elle renvoie donc bottom (résultat provisoire du sat). Le résultat global est donc celui de la première clause. On réitère ensuite ce procédé jusqu'à ce qu'on trouve un point fixe. L'exécution de la première clause est identique à l'exécution précédente; quant à la deuxième, lors de l'exécution de l'appel récursif, le calcul s'arrête à nouveau car il est commencé sans être terminé et renvoyer le résultat provisoire du sat, qui est le résultat trouvé lors de la première exécution (résultat de la première clause). On calcule ensuite le lub des trois sous-buts, ce qui donne une fois restreint à X1, X2 et X3 une liste pour les trois variables. Le lub des deux clauses donne trois listes sans formes avec des partages possibles entre tous les termes. Une troisième itération se produit et fournit un résultat identique à la seconde; le point fixe est atteint et l'algorithme s'arrête.

Des cyclages peuvent se produire : les ensembles sont infinis. Une partie de ces cyclages est évitée grâce au lub qui élimine des formes. Ce n'est pas suffisant. Les formes peuvent croître indéfiniment avant même de passer par le lub. Pour exemple, lançons le programme sur append avec trois variables dans la même classe, à la première itération et lors de l'exécution de la deuxième clause, il va lancer l'exécution de append(X5,cons(X4,X5),X5). A la deuxième itération, il essaie d'exécuter append(X5', cons(cons(X4,X5)),X5'), et ainsi de suite. La version implémentée du programme est capable de détecter les cyclages *directs* (f(f(....))), mais pas les cyclages *indirects* (f(g(h(f(g(h(f(....)))))))) car essayer de les détecter pourrait revenir à détecter les cycles dans un graphe (problème de complexité qui pourrait ralentir considérablement le programme), et le fait de la présence d'une telle forme ne provoque pas nécessairement un bouclage. Le programme "arrête les frais" dès que dans une forme, un des élements a la

même forme que l'élément de départ. Cela se fait lorsque l'on vérifie si une demande de calcul appartient à suspended. Le résultat peut être imprécis au cas où on arrête l'exécution d'une clause en croyant qu'elle boucle alors qu'en fait elle ne boucle pas. Le programme boucle aussi au cas où un cyclage indirect se produit.

Une deuxième manière d'empêcher le cyclage serait de forcer chaque sous-ensemble de l'ensemble suspended pour une procédure donnée à être ordonné strictement sur les substitutions d'entrées. Cela est rendu facile grâce au lub qui permet d'empêcher une croissance des formes, tout en restant correct puisqu'il généralise le résultat. On pourrait croire que cela provoquerait une grande perte de précision, mais une implémentation antérieure sur le domaine des modes montre que ce n'est pas le cas [VEN91]; les résultats étant même parfois plus précis.

Un autre exemple est celui du quicksort. Considérons sa version normalisée :

```
append(x1,x2,x3) :- x1=nil(), x2=x3.
append(x1,x2,x3) :- x1=cons(x5,x6), x3=cons(x5,x4), append(x6,x2,x4).
coupe(x1,x2,x3,x4) :- x4=nil(), x3=nil(), x2=nil().
coupe(x1,x2,x3,x4) :- x2=cons(x6,x7), x3=cons(x6,x5), x6<x1,
                      coupe(x1,x7,x5,x4).
coupe(x1,x2,x3,x4) :- x2=cons(x6,x7), x4=cons(x6,x5), x6<x1,
                      coupe(x1,x7,x3,x5).
tri(x1,x2) :- x2=nil(), x1=nil().
tri(x1,x2) :- x1=cons(x8,x3), coupe(x8,x3,x4,x5), tri(x4,x6), tri(x5,x7),
              x9=cons(x8,x7), append(x6,x9,x2).
```

Exécutons ce programme avec (list, var) comme substitution d'entrée de la procédure *tri*, et avec un sharing possible entre les deux variables. La première clause renverra "list" pour les deux variables. Exécutons la première clause : x8 et x3 sont mis à ANY et LIST respectivement. La procédure coupe est exécutée et se termine avec (ANY, LIST, LIST, LIST) comme résultat. Ensuite tri est appelé avec (LIST, VAR) et la procédure est réexécutée car il n'y a pas de sharing possible entre x4 et x6. L'exécution recommence de la même manière jusqu'au premier appel récursif de tri de la deuxième clause. A ce moment, l'algorithme prend le résultat provisoire du sat qui vaut BOTTOM. Le sat est mis à jour et reçoit le résultat de la première clause. L'exécution recommence une deuxième fois et lors de l'appel tri(x4,x6), prend le résultat provisoire. x4 et x6 deviennent donc des termes de type LIST, de même pour X5 et x7. Ensuite x9 devient LIST et x8 devient ANY. Ensuite, on lance append sur (LIST, LIST, VAR) et X2 devient

LIST. Le calcul recommence et fournit le même résultat : le point fixe est atteint. Le problème est qu'à chaque exécution de tri, le point fixe de la procédure coupe est calculé alors qu'il suffit de prendre le résultat dans le sat après l'avoir trouvé lors de sa première exécution.

Il peut néanmoins arriver que le résultat soit imprécis, mais sans être faux : il est trop général. Un moyen de le rendre plus précis serait de modifier quelques opérations telles que l'extension par exemple, mais ce serait alors risquer d'obtenir un résultat faux, à moins de démontrer que les modifications effectuées restent correctes.

Lecture des données.

Le programme doit aussi lire les données concernant la substitution d'entrée, β_{in}, qui doit permettre de donner un type et une forme à chaque paramètre de la requête. Le foncteur de la requête et son arité seront lus au clavier. La substitution d'entrée β_{in} sera lue dans un fichier ascii fourni par l'utilisateur, qui devra posséder le format suivant :

$|| x_i \quad sv_i || \ldots$
$|| sv_j \quad tp_j || \ldots$
$[frm || sv_k \quad foncteur_k \quad [|| sv_{k1} || \ldots] || \ldots]$
$[|| (sv_m,sv_n) || \ldots]$

Attention : chaque x_i, sv_j, sv_k se trouve sur une ligne distincte. L'ordre est important (d'abord les sv, puis les types, ...). L'utilisateur encode les références comme des indices numérotés entre 1 et n. Il suffit de considérer que l'indice de classe est la position dans l'ensemble où la classe sera stockée, et évidemment de vérifier qu'il n'y a ni contradiction, ni classe sans type. Chaque sv est un entier compris entre 1 et p, chaque x_i est un x minuscule immédiatement suivi (sans blanc) d'un chiffre entre 1 et n, où n est le nombre de paramètres de la requête. "x_i" référencera le i-ème paramètre de la requête. Chaque type est un string correspondant à un des types définis dans le système de types. Chaque foncteur est une chaîne de caractères sans blancs, ne commençant ni par un chiffre ni par un underscore. Chaque (sv_m,sv_n) représente $Ps(sv_m,sv_n)$. Le fichier comporte des blancs et ne peut pas comporter de tabulations. Les sv doivent être présentés au

programme dans l'ordre croissant. Toute erreur dans ce fichier ou dans la source prolog produira un effet indéfini.

Grâce à la représentation en listes chaînées, il est aisé de construire le β et de le compléter au fur et à mesure que le fichier est lu. Tout doit partir de l'ensemble t qui contient les types et les formes. Les couples Ps(i,i) seront ajoutés automatiquement par le programme pour autant que l'indice i ne soit pas la classe d'une constante. De plus, si on essaie de mettre un Ps à une constante, celui-ci sera purement et simplement ignoré.

Chaque ligne constitue une information pour le β, que ce soit une composante sv, un type de classe, une forme ou un Ps. Les types sont encodés par leur "nom de code" : var, list, nlv, novar, nolist, lv, any, bottom; et en minuscules.

Exemple : pour représenter le β référencé précédemment par ***S1***, il faudrait taper :

```
x1  1
x2  2
x3  1
x4  3
1  var
2  list
3  any
4  any
5  list
6  nlv
7  list
frm  2 cons 4 5
3  f  2
5  cons  6 7
7  nil
(5,6) (2,4) (2,5) (2,3)
```

La substitution sera ensuite représentée en mémoire de la manière vue précédemment.

Chapitre 5 :
Résultats obtenus

Ce chapitre est destiné à commenter les résultats obtenus par l'exécution du programme sur diverses procédures. Je vais d'abord détailler des exécutions d'append avec différentes substitutions abstraites d'entrée, après quoi j'expliquerai l'exécution de l'interpréteur sur un programme plus consistent tel que le quick sort.

Soit la procédure append normalisée comme suit :

append(X1,X2,X3) := X2=X3, X1=nil().
append(X1,X2,X3) := X1=cons(X4,X5), X3=cons(X4,X6), append(X5,X2,X6).

Mon premier exemple consistera à lancer append le βin suivant en entrée :

```
x1 1
x2 2
x3 3
1 list
2 list
3 var
(1,1) (2,2) (3,3)
```

Les deux premières variables sont des listes et la troisième est une variable. Il stocke cette substitution d'entrée dans le sat et lui affecte bottom comme résultat provisoire. Ensuite il exécute la première clause, et trouve le résultat suivant :

```
sv : x3, 2
sv : x2, 2
sv : x1, 1
1 list
nil
2 list
(2,2)
```

Cela veut dire que X2 et X3 sont des termes identiques de type LIST dont la forme est indéfinie et partageant avec eux-mêmes. De plus, X1 est un terme dont la valeur est la liste vide et n'acceptant pas de sharing puisqu'il a une forme. Le programme exécute

ensuite la deuxième clause. Après l'exécution des deux premiers built-in, le résultat est le suivant :

```
sv : x3, 3
sv : x2, 2
sv : x1, 1
sv : x4, 4
sv : x5, 5
sv : x6, 6
1 list
cons(4,5)
2 list
3 nlv
cons(4,6)
4 any
5 list
6 var
(6,6) (5,5) (4,4) (2,2) (4,5)
```

Il restreint cette substitution à (X5,X2,X6), effectue un changement de variables, et retrouve exactement la substitution d'entrée. Il prend donc le résultat provisoire du sat (bottom) et le renvoie, l'exécution de la clause donne bottom. Ensuite le sat est mis à jour :

```
bin[0] :
sv : x3, 3
sv : x2, 2
sv : x1, 1
1 list
2 list
3 var
(3,3)   (2,2)   (1,1)

bout[0] :
sv : x1, 1
sv : x2, 2
sv : x3, 2
1 list
nil
2 list
(2,2)
```

L'exécution de la première clause recommence de manière identique à la première fois pour donner le même résultat. L'exécution des deux built-in recommence, elle aussi, de la même manière que la première exécution. Le résultat provisoire de append ne vaut plus bottom, et le résultat de la clause vaut :

```
sv : x3, 3
sv : x2, 2
sv : x1, 1
```

```
sv : x4, 4
sv : x5, 5
sv : x6, 2
1 list
cons(4,5)
2 list
3 list
cons(4,2)
4 any
5 list
nil
(4,4) (2,2)
```

Le sat est ensuite mis à jour et le βout est remplacé par le lub des deux clauses :

```
bin[0] :
sv : x3, 3
sv : x2, 2
sv : x1, 1
1 list
2 list
3 var
(3,3)   (2,2)   (1,1)

bout[0] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
1 list
2 list
3 list
(3,3)   (2,3)   (2,2)   (1,3)   (1,1)
```

La troisième exécution va rajouter un ps entre X1 et X2 et βout[0] devient :

```
sv : x1, 1
sv : x2, 2
sv : x3, 3
1 list
2 list
3 list
(3,3)   (3,2)   (2,2)   (1,3)   (1,2)   (1,1)
```

La quatrième itération fournit le même résultat, nous sommes donc arrivé au point fixe et le programme se termine.

Le deuxième exemple emploie toujours la procédure append mais avec une substitution d'entrée différente :

```
x1 1
x2 2
x3 3
1  var
2  var
3  list
(1,1) (2,2) (3,3)
```

La première exécution de la procédure se passe comme dans l'exemple précédent, excepté que le résultat avant l'appel récursif est différent :

```
sv : x3, 3
sv : x2, 2
sv : x1, 1
sv : x4, 4
sv : x5, 5
sv : x6, 6
1 nlv
cons(4,5)
2 var
3 list
cons(4,6)
4 any
5 var
6 list
(6,6)   (5,5)   (4,4)   (2,2)   (4,6)
```

La restriction à (X5,X2,X6) et le changement de variable fournissent la substitution d'entrée, la claues renvoie bottom et le sat est mis à jour :

```
bin[0] :
sv : x3, 3
sv : x2, 2
sv : x1, 1
1 var
2 var
3 list
(3,3)   (2,2)   (1,1)

bout[0] :
sv : x1, 1
sv : x2, 2
sv : x3, 2
1 list
nil
2 list
(2,2)
```

L'exécution recommence, et à la deuxième itération la deuxième clause fournit un résultat :

```
sv : x3, 3
sv : x2, 2
sv : x1, 1
sv : x4, 4
sv : x5, 5
sv : x6, 2
1 list
cons(4,5)
2 list
3 list
cons(4,2)
4 any
5 list
nil
(4,4)   (2,2)   (2,4)
```

Le sat est mis à jour en conséquence :

```
bin[0] :
sv : x3, 3
sv : x2, 2
sv : x1, 1
1 var
2 var
3 list
(3,3)   (2,2)   (1,1)

bout[0] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
1 list
2 list
3 list
(3,3)   (3,2)   (2,2)   (3,1)   (2,1)   (1,1)
```

Une troisième itération fournira le même résultat et le programme se termine.

Le troisième et dernier exemple d'exécution du programme sur append concerne les cas de cyclages. Si append est lancé avec trois variables dans la même classe, les formes des termes peuvent croître indéfiniment. Voyons comment ce cyclage a pu être évité, soit le βin :

```
x1 1
x2 1
x3 1
1 var
(1,1)
```

L'exécution de la première clause fournira le résultat suivant :

```
sv : x3, 1
sv : x2, 1
sv : x1, 1
1 list
nil
```

L'exécution de la deuxième clause fournira le résultat suivant avant l'appel récursif :

```
sv : x3, 1
sv : x2, 1
sv : x1, 1
sv : x4, 2
sv : x5, 3
sv : x6, 3
1 nlv
cons(2,3)
2 var
3 var
(3,3)   (2,2)
```

Ce résultat restreint à (X5,X2,X6) ne se trouve pas dans le suspended. On lance donc le calcul avec la substitution suivante :

```
sv : x1 1
sv : x2 2
sv : x3 1
1 var
2 nlv
cons(3,1)
3 var
(1,1) (3,3)
```

La première clause fournit un échec : spécat ne peut donner au premier terme la forme du deuxième (occur-check). La deuxième clause est exécutée, et avant l'appel récursif, le résultat est le suivant :

```
sv : x1, 5
sv : x2, 1
sv : x3, 5
sv : x4, 2
sv : x5, 3
sv : x6, 3
1 nlv
cons(4,5)
2 var
3 var
4 var
5 nlv
cons(2,3)
(3,3)   (2,2)   (4,4)
```

A ce moment, le programme détecte que la forme de X2 a commencé à croître et arrête les frais. Il prend le résultat provisoire de cette substitution dans le sat et renvoie bottom. Les deux clauses renvoyant bottom, le résultat du calcul est renvoyé à la première exécution de la deuxième clause qui renvoie donc également append. Le sat est ensuite mis à jour :

```
bin[0] :
sv : x3, 1
sv : x2, 1
sv : x1, 1
1 var
(1,1)

bout[0] :
sv : x1, 1
sv : x2, 1
sv : x3, 1
1 list
nil

bin[1] :
sv : x1, 3
sv : x2, 1
sv : x3, 3
1 nlv
cons(2,3)
2 var
3 var
(3,3)   (2,2)

bout[1] :
bottom
```

Une seconde itération refait exactement la même chose et fournit le même résultat. Le programme se termine en faisant deux itérations pour le βin donné et deux itérations pour la substitution générée lors de l'appel récursif.

Le dernier exemple est celui du quicksort dont voici la version normalisée :

```
append(x1,x2,x3) :- x2=x3, x1=nil().
append(x1,x2,x3) :- x1=cons(x5,x6), x3=cons(x5,x4), append(x6,x2,x4).
coupe(x1,x2,x3,x4) :- x2=nil(), x3=nil(), x4=nil().
coupe(x1,x2,x3,x4) :- x2=cons(x6,x7), x3=cons(x6,x5), x6<x1, coupe(x1,x7,x5,x4).
coupe(x1,x2,x3,x4) :- x2=cons(x6,x7), x4=cons(x6,x5), x6<x1, coupe(x1,x7,x3,x5).
tri(x1,x2) :-    x1=nil(), x2=nil().
tri(x1,x2) :-    x1=cons(x8,x3), coupe(x8,x3,x4,x5), tri(x4,x6), tri(x5,x7),
                 x9=cons(x8,x7), append(x6,x9,x2).
```

Nous l'exécuterons avec la substitution d'entrée suivante :

```
x1 1
x2 2
1 list
2 var
(1,1) (1,2) (2,2)
```

La première clause de tri s'exécute et se termine avec le résultat suivant :

```
sv : x2, 2
sv : x1, 1
1 list
nil
2 list
nil
```

Ensuite la deuxième clause s'exécute, et avant l'appel à coupe, fournit la substitution abstraite suivante :

```
sv : x2, 2
sv : x1, 1
sv : x3, 3
sv : x4, 4
sv : x5, 5
sv : x6, 6
sv : x7, 7
sv : x8, 8
sv : x9, 9
1 list
cons(8,3)
2 any
3 list
4 var
5 var
6 var
7 var
8 any
```

```
9 var
(9,9)   (8,8)   (7,7)   (6,6)   (5,5)   (4,4)
(3,3)   (2,2)   (2,8)   (3,8)   (2,3)
```

La procédure coupe est appelée avec les variables X8,X3,X4,X5 et le sat est étendu. La première clause ne pose pas de problèmes et fournit le résultat suivant :

```
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
1 any
2 list
nil
3 list
nil
4 list
nil
(1,1)
```

La deuxième clause est exécutée et avant l'appel récursif, elle fournit le résultat suivant :

```
sv : x1, 4
sv : x2, 1
sv : x3, 2
sv : x4, 3
sv : x5, 5
sv : x6, 6
sv : x7, 7
1 list
cons(6,7)
2 nlv
cons(6,5)
3 var
4 novar
5 var
6 novar
7 list
(7,7) (6,6) (5,5) (4,4) (3,3) (4,6) (6,7) (4,7)
```

L'appel est lancé récursivement une fois que la substitution a été restreinte aux variables (X1,X7,X5,X4). La première clause de coupe s'exécute de la même façon que précédemment, mais cette fois, avant l'appel récursif, le résultat est le suivant :

```
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
sv : x5, 5
sv : x6, 6
sv : x7, 7
1 var
```

```
2 novar
3 nlv
cons(6,5)
4 list
cons(6,7)
5 var
6 novar
7 list
(7,7) (6,6) (5,5) (2,2) (1,1) (2,6) (6,7) (2,7)
```

Réexécuter coupe avec ce résultat restreint à (X1,X7,X5,X4) revient à relancer un calcul déjà commencé. Le résultat provisoire associé, bottom, est renvoyé. La troisième clause de coupe s'exécute de manière similaire à la deuxième pour fournir le même résultat. Le sat est mis à jour comme suit :

```
bin[0] :
sv : x2, 2
sv : x1, 1
1 list
2 var
(2,2)   (1,2)   (1,1)

p=tri,arite=2

bout[0] :
bottom

bin[1] :
sv : x1, 4
sv : x2, 1
sv : x3, 2
sv : x4, 3
1 list
2 var
3 var
4 any
(4,4)   (3,3)   (2,2)   (1,1)   (1,4)

p=coupe,arite=4

bout[1] :
sv : x1, 1
sv : x2, 3
sv : x3, 2
sv : x4, 4
1 novar
2 list
nil
3 list
nil
4 list
nil
(1,1)
```

```
bin[2] :
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
1 var
2 novar
3 var
4 list
(4,4)   (3,3)   (2,2)   (1,1)   (2,4)

p=coupe,arite=4

bout[2] :
sv : x1, 1
sv : x2, 3
sv : x3, 2
sv : x4, 4
1 novar
2 list
nil
3 list
nil
4 list
nil
(1,1)
```

La procédure coupe est ensuite relancée. La première clause s'exécute de la même façon. Avant l'appel récursif, la deuxième clause renvoie :

```
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
sv : x5, 5
sv : x6, 6
sv : x7, 7
1 var
2 novar
3 nlv
cons(6,5)
4 list
cons(6,7)
5 var
6 novar
7 list
(7,7) (6,6) (5,5) (2,2) (1,1) (2,6) (6,7) (2,7)
```

Réexécuter coupe reviendrait à lancer un calcul déjà commencé, le programme prend donc le résultat provisoire et étend l'étend à la clause entière. La clause renvoie finalement :

```
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
sv : x5, 5
sv : x6, 6
sv : x7, 7
1 list
nil
2 novar
3 list
cons(6,5)
4 list
cons(6,7)
5 list
nil
6 novar
7 list
nil
(6,6)   (2,2)   (2,6)
```

Le calcul de la troisième clause est similaire à celui de la deuxième; son résultat est :

```
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
sv : x5, 7
sv : x6, 6
sv : x7, 5
1 list
cons(6,7)
2 novar
3 list
nil
4 list
cons(6,5)
5 list
nil
6 novar
7 list
nil
(2,2)   (6,6)   (2,6)
```

Le lub de ces trois clauses vaut permet d'ajuster le sat :

```
bin[1] :
sv : x1, 4
sv : x2, 1
sv : x3, 2
sv : x4, 3
1 list
2 var
3 var
4 any
```

```
(4,4)   (3,3)   (2,2)   (1,1)   (1,4)

p=coupe,arite=4

bout[1] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
1 novar
2 list
3 list
4 list
(4,4)(3,3)(4,2)(3,2)(2,2)(4,1)(3,1)(2,1)(1,1)

bin[2] :
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
1 var
2 novar
3 var
4 list
(4,4)   (3,3)   (2,2)   (1,1)   (2,4)

p=coupe,arite=4

bout[2] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
1 novar
2 list
3 list
4 list
(4,4)(3,3)(4,2)(3,2)(2,2)(4,1)(3,1)(2,1)(1,1)
```

Une troisième exécution de la procédure coupe va donner le même résultat excepté qu'il y aura des sharings en plus :

```
bin[1] :
sv : x1, 4
sv : x2, 1
sv : x3, 2
sv : x4, 3
1 list
2 var
3 var
4 any
(4,4)   (3,3)   (2,2)   (1,1)   (1,4)

p=coupe,arite=4
```

```
bout[1] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
1 novar
2 list
3 list
4 list
(4,4)(3,4)(3,3)(4,2)(3,2)(2,2)(4,1)(3,1)(1,2)(1,1)

bin[2] :
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
1 var
2 novar
3 var
4 list
(4,4)   (3,3)   (2,2)   (1,1)   (2,4)

p=coupe,arite=4

bout[2] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
1 novar
2 list
3 list
4 list
(4,4)(4,3)(3,3)(2,4)(2,3)(2,2)(4,1)(3,1)(2,1)(1,1)
```

Une exécution supplémentaire de coupe donnera le même résultat, le point fixe est atteint. On revient donc un appel récursif en arrière et nous sommes à l'exécution de la deuxième clause de coupe qui renvoie :

```
sv : x1, 4
sv : x2, 1
sv : x3, 2
sv : x4, 3
1 list
cons(6,7)
2 list
cons(6,5)
3 list
4 novar
5 list
6 novar
7 list
(7,7)(6,6)(5,5)(4,4)(3,3)(4,6)(6,7)(4,7)(4,5)(5,6)
(5,7)(3,5)(3,4)(3,6)(3,7)
```

La troisième clause de coupe s'exécute de la même manière, en effectuant un appel récursif. Le sat est encore mis à jour :

```
bin[1] :
sv : x1, 4
sv : x2, 1
sv : x3, 2
sv : x4, 3
1 list
2 var
3 var
4 any
(4,4)   (3,3)   (2,2)   (1,1)   (1,4)

p=coupe,arite=4

bout[1] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
1 any
2 list
3 list
4 list
(4,4)(4,3)(3,3)(4,2)(2,3)(2,2)(1,4)(1,3)(1,2)(1,1)

bin[2] :
sv : x1, 2
sv : x2, 4
sv : x3, 3
sv : x4, 1
1 var
2 novar
3 var
4 list
(4,4)   (3,3)   (2,2)   (1,1)   (2,4)

p=coupe,arite=4

bout[2] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
1 novar
2 list
3 list
4 list
(4,4)(4,3)(3,3)(2,4)(2,3)(2,2)(4,1)(3,1)(2,1)(1,1)
```

La procédure coupe est ensuite réexécutée pour donner le même résultat. Le contrôle revient à l'exécution de la procédure tri. Avant l'exécution du sous-but tri(x4,x6), le résultat est le suivant :

```
sv : x2, 2
sv : x1, 1
sv : x3, 3
sv : x4, 4
sv : x5, 5
sv : x6, 6
sv : x7, 7
sv : x8, 8
sv : x9, 9
1 list
cons(8,3)
2 any
3 list
4 list
5 list
6 var
7 var
8 any
9 var
(9,9)(8,8)(7,7)(6,6)(5,5)(4,4)(3,3)(2,2)(2,8)(3,8)
(2,3)(4,8)(2,4)(3,4)(4,5)(5,8)(2,5)(3,5)
```

Etant donné qu'il n'y a pas de sharing entre X4 et X6, la procédure tri est exécutée récursivement. La première clause renvoie :

```
sv : x1, 1
sv : x2, 2
1 list
nil
2 list
nil
```

La deuxième clause s'exécute de manière identique pour ce qui est des deux premiers sous-buts puisqu'X2 n'y est pas concerné. Avant l'exécution du sous-but tri(X4,X6), le résultat est donc :

```
sv : x1, 1
sv : x2, 2
sv : x3, 3
sv : x4, 4
sv : x5, 5
sv : x6, 6
sv : x7, 7
sv : x8, 9
sv : x9, 8
1 list
cons(9,3)
2 var
3 list
4 list
5 list
6 var
7 var
```

```
8 var
9 any
(9,9)(8,8)(7,7)(6,6)(5,5)(4,4)(3,3)(2,2)(3,9)(4,9)
(3,4)(4,5)(5,9)(3,5)
```

Si tri est restreint à X4 et X6, il correspond à un appel déjà commencé. On lui donne donc le résultat provisoire bottom et tri est réexécuté. Le sat est mis à jour :

```
bin[0] :
sv : x2, 2
sv : x1, 1
1 list
2 var
(2,2)   (1,2)   (1,1)

p=tri,arite=2

bout[0] :
sv : x1, 1
sv : x2, 2
1 list
nil
2 list
nil

bin[3] :
sv : x1, 1
sv : x2, 2
1 list
2 var
(2,2)   (1,1)

p=tri,arite=2

bout[3] :
sv : x1, 1
sv : x2, 2
1 list
nil
2 list
nil
```

Le calcul de tri est recommencé, mais au lieu d'affecter bottom aux appels récursifs de tri, on leur affecte le résultat de la première clause. Après l'exécution des deux sous-buts tri(X4,X6) et tri(X5,X7), le résultat est le suivant :

```
sv : x1, 7
sv : x2, 1
sv : x3, 9
sv : x4, 2
sv : x5, 3
sv : x6, 4
sv : x7, 5
```

```
sv : x8, 8
sv : x9, 6
1 var
2 list
nil
3 list
nil
4 list
nil
5 list
nil
6 var
7 list
cons(8,9)
8 any
9 list
(9,8)   (9,9)   (8,8)   (6,6)   (1,1)
```

Le built-in X9=cons(X8,X7) est exécuté. Le résultat est alors le suivant :

```
sv : x1, 6
sv : x2, 1
sv : x3, 8
sv : x4, 2
sv : x5, 3
sv : x6, 4
sv : x7, 5
sv : x8, 7
sv : x9, 9
1 var
2 list
nil
3 list
nil
4 list
nil
5 list
nil
6 list
cons(7,8)
7 any
8 list
9 list
cons(7,5)
(8,7)   (8,8)   (7,7)   (1,1)
```

La procédure append est alors lancée sur (X6,X9,X2). Comme X6 est nil(), seule la première clause réussit et le résultat est obtenu en deux itérations. Le sat est mis à jour en ajoutant :

```
bin[4] :
sv : x1, 2
sv : x2, 5
sv : x3, 1
```

```
1 var
2 list
nil
3 list
nil
4 any
5 list
cons(4,3)
(4,4)   (1,1)

p=append,arite=3

bout[4] :
sv : x1, 2
sv : x2, 1
sv : x3, 1
1 list
cons(4,3)
2 list
nil
3 list
nil
4 any
(4,4)
```

Le résultat de la deuxième clause de tri est :

```
sv : x1, 5
sv : x2, 8
sv : x3, 7
sv : x4, 1
sv : x5, 2
sv : x6, 3
sv : x7, 4
sv : x8, 6
sv : x9, 8
1 list
nil
2 list
nil
3 list
nil
4 list
nil
5 list
cons(6,7)
6 any
7 list
8 list
cons(6,4)
(7,6)   (7,7)   (6,6)
```

Le sat est mis à jour :

```
bin[0] :
sv : x2, 2
sv : x1, 1
1 list
2 var
(2,2)   (1,2)   (1,1)

p=tri,arite=2

bout[0] :
sv : x1, 1
sv : x2, 2
1 list
2 list
(2,2)   (1,2)   (1,1)

bin[3] :
sv : x1, 1
sv : x2, 2
1 list
2 var
(2,2)   (1,1)

p=tri,arite=2

bout[3] :
sv : x1, 1
sv : x2, 2
1 list
2 list
(2,2)   (1,2)   (1,1)
```

Le calcul est relancé et tout se passe de la même façon excepté que X6 et X7 ne sont plus nil après l'exécution des sous-buts tri(X4,X6) et tri(X5,X7). La procédure append est donc lancée avec le βin suivant :

```
x1, 1
x2, 2
x3, 3
1   list
2   list
cons(4,5)
3   var
4   any
5   list
(6,6) (6,4) (6,5) (3,3) (4,5) (4,4) (5,5)
```

Le résultat est trouvé en quatre itérations et le sat est mis à jour :

```
bin[5] :
sv : x1, 2
sv : x2, 4
sv : x3, 1
```

```
1 var
2 list
3 list
4 list
cons(5,3)
5 any
(5,5)(3,3)(2,2)(1,1)(5,2)(5,3)(2,3)

p=append,arite=3

bout[5] :
sv : x1, 1
sv : x2, 2
sv : x3, 3
1 list
2 list
cons(4,5)
3 list
cons(6,7)
4 any
5 list
6 any
7 list
(7,7)(7,6)(6,6)(7,5)(6,5)(5,5)(4,7)(4,6)(4,5)(4,4)
(1,7)(1,6)(1,5)(1,4)(1,1)
```

Le calcul est relancé, les points fixes sont complètement recalculés et le résultat n'est plus amélioré, le sat n'est donc plus mis à jour.

Le résultat final est donc :

```
sv : x1, 1
sv : x2, 2
1 list
2 list
(2,2)   (1,2)   (1,1)
```

Nombre d'itérations : 53 (toutes procédures confondues).

Temps d'exécution : 25 secondes.

Bibliographie

[ABR87] Samson Abramsky, Chris Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987, pp 9-31.

[JAQ90] J.-M. Jacquet, *Intelligence Artificielle*, cours, Institut d'Informatique, F.U.N.D.P., Namur.

[LEC90] Baudouin Le Charlier, Kaninda Musumbu, P. Van Hentenryck, Efficient and Accurate Algorithms for the Abstract Interpretation of Logic Programs, Technical report 37/90, Institute of Computer Science, University of Namur, Belgium, 1990.

[LEC91-1] Baudouin Le Charlier, Kaninda Musumbu, Une sémantique opérationnelle instrumentale pour prolog et son application à la preuve de consistance d'un modèle d'interprétation abstraite, in J.-P. Delahaye, editor, *Actes des Journées Francophones de Programmation Logique (JFPL'92)*, Lille, May 1992.

[LEC91-2] Baudouin Le Charlier, *Théorie des Programmes*, cours, Institut d'Informatique, F.U.N.D.P., Namur.

[LEC91-3] Baudouin Le Charlier, "L'analyse statique des programmes par interprétation abstraite", Institut d'Informatique, F.U.N.D.P., Namur, 1991.

[LEC91-4] Baudouin Le Charlier, P. Van Hentenryck, Experimental evaluation of a generic abstract interpretation algorithm for prolog, Technical report 91-42, Institute of Computer Science, University of Namur, Belgium, (also Brown University), 1991.

[LER77] H. Leroy, *La Fiabilité des Programmes*, Ecole d'été de l'A.F.C.E.T., 1977, pp. 5.1-5.15.

[LLO87] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, second edition, 1987.

[MUS90] Kaninda Musumbu, *Interprétation abstraite des programmes prolog*, Thèse de doctorat, Institut d'Informatique, F.U.N.D.P., Namur, septembre 1990.

[STE86] Leon Sterling, Ehud Shapiro, *The Art of Prolog : Advanced Programming Techniques*, MIT Press, 1986.

[VEN91] V. Englebert, B. Le Charlier, D. Roland, P. Van Hentenyck, Generic abstract interpretation algorithms for prolog : Two optimization techniques and their experimental evaluation, Technical Report, Institute of Computer Science, University of Namur, Belgium, (also Brown University : Technical Report No. Cs-91-67), December 1991.

FACULTES
UNIVERSITAIRES
N. D. DE LA PAIX

N A M U R

INSTITUT D'INFORMATIQUE

IMPLEMENTATION D'UN INTERPRETEUR ABSTRAIT DE PROGRAMMES PROLOG

(annexes)

par Alain Bourgeois

Promoteur :

Professeur B. Le Charlier

Mémoire présenté en vue
de l'obtention du titre de
Licencié et Maître
en Informatique

Année académique : 1991-1992

Table des matières

1. Algorithmes génériques 1

2. Organigramme de specat 3

3. Listing 12

abi.c 12
abi.h 14
betaordr.c 15
betaordr.h 20
defines.h 21
ensembles.c 22
ensembles.h 27
essais.c 28
essais.h 33
ext2.c 34
ext2.h 35
ia.c 36
litfich.c 38
litfich.h 42
litfichm.c 43
litfichm.h 45
majlub.c 46
majlub.h 49
managest.c 50
managest.h 52
messtruc.c 53
messtruc.h 54
norm.c 57
norm.h 74

o_access.c 75
o_access.h 76
o_couple.c 77
o_couple.h 79
o_operat.c 80
o_operat.h 89
o_ps.c 90
o_ps.h 97
o_sv.c 98
o_sv.h 101
o_types.c 102
o_types.h 105
solve.c 106
solve.h 113
sortie.c 114
sortie.h 119
stdfunct.c 120
stdfunct.h 123

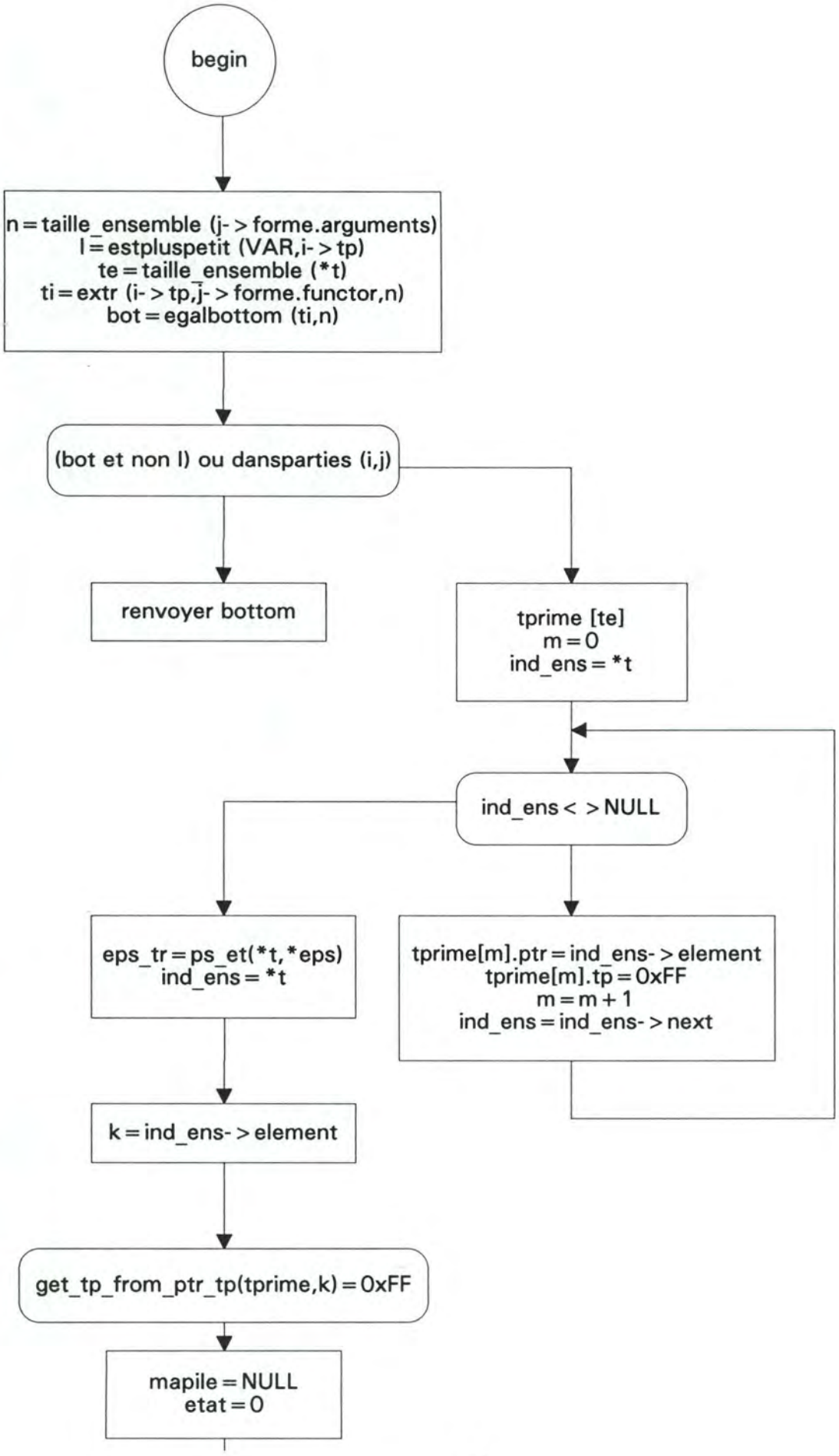
Annexe 1 :
Algorithmes génériques

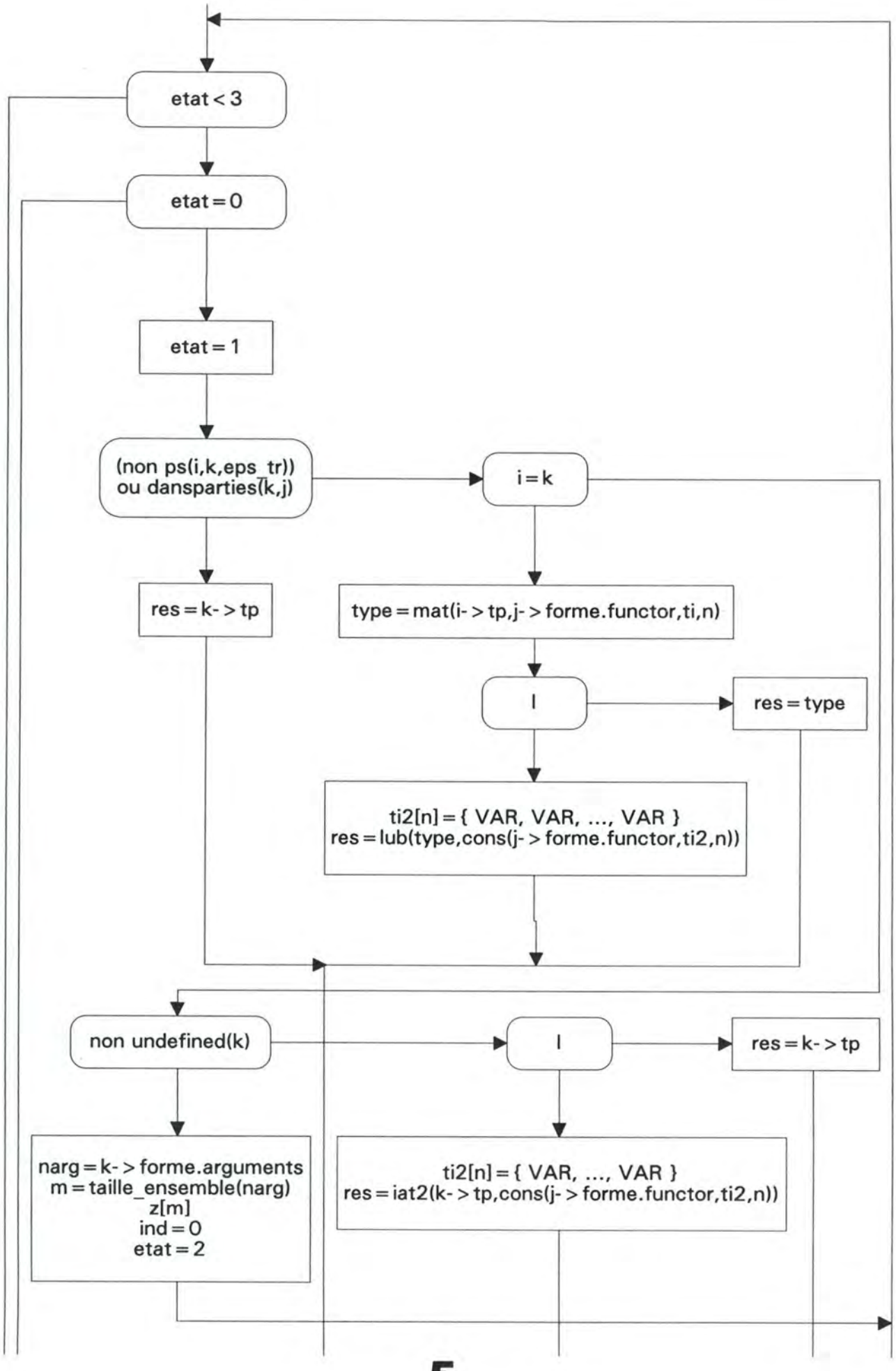
```
procedure solve (in β_in,p; out β_out)
begin
        sat := ∅;
        same_sat := true;
        solve_goal(β_in,p,∅,same_sat,sat);
        β_out := sat(β_in,p)
end


procedure solve_goal(in β_in,p,suspended; inout same_sat,sat)
begin
        if (β_in,p)∉suspended then
        begin
                if (β_in,p)∉dom(sat) then
                begin
                        same_sat := false;
                        sat := EXTEND(β_in,p,sat)
                end;
                solve_procedure(β_in,p,suspended,same_sat,sat)
        end
end


procedure solve_procedure(in β_in,p,suspended; inout same_sat,sat)
begin
        repeat
                same_sat_aux := solve_all_clauses(β_in,p,suspended∪{(β_in,p)}, sat);
                same_sat := same_sat et same_sat_aux
        until same_sat_aux
end
```

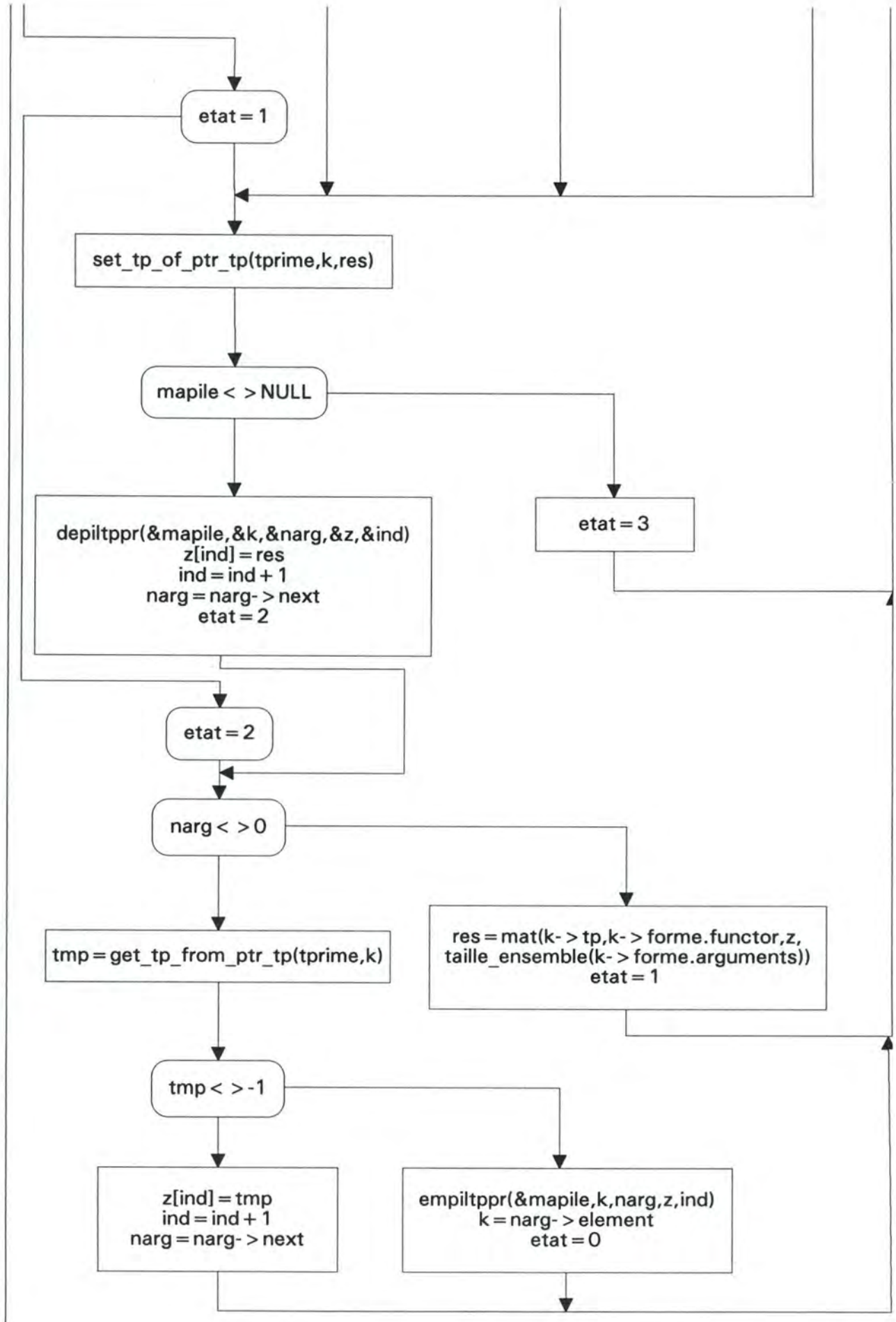
```
function solve_all_clauses(in β_in,p,suspended; inout sat) : boolean
begin
    β_out := ⊥;
    solve_all_clauses := true;
    for i := 1 to m with c_1, ..., c_m clauses-of p do
    begin
        β_aux := solve_clause(β_in,c_i,suspended,solve_all_clauses,sat);
        β_out := UNION(β_out,β_aux)
    end;
    if non(β_out≤sat(β_in,p)) then
    begin
        sat := ADJUST(β_in,p,β_out,sat);
        solve_all_clauses := faux
    end
end

function solve_clause(in β_in,c,suspended, inout same_sat, sat) : beta
begin
    β_ext := EXTC(c,β_in);
    for i := 1 to m with b_1,...,b_m body-of c do
    begin
        β_aux := RESTRB(b_i,β_ext);
        switch (b_i) of
            case x_j = x_k :
                β_aux := AI_VAR(β_aux)
            case x_j = f(...) :
                β_aux := AI_FUNC(β_aux,f)
            case p(...) :
                solve_goal(β_aux,p,suspended,same_sat,sat);
                β_aux := sat(β_aux,p)
        end;
        β_ext := EXTB(b_i,β_ext,β_aux)
    end;
    solve_clause := RESTRC(c,β_ext)
end
```

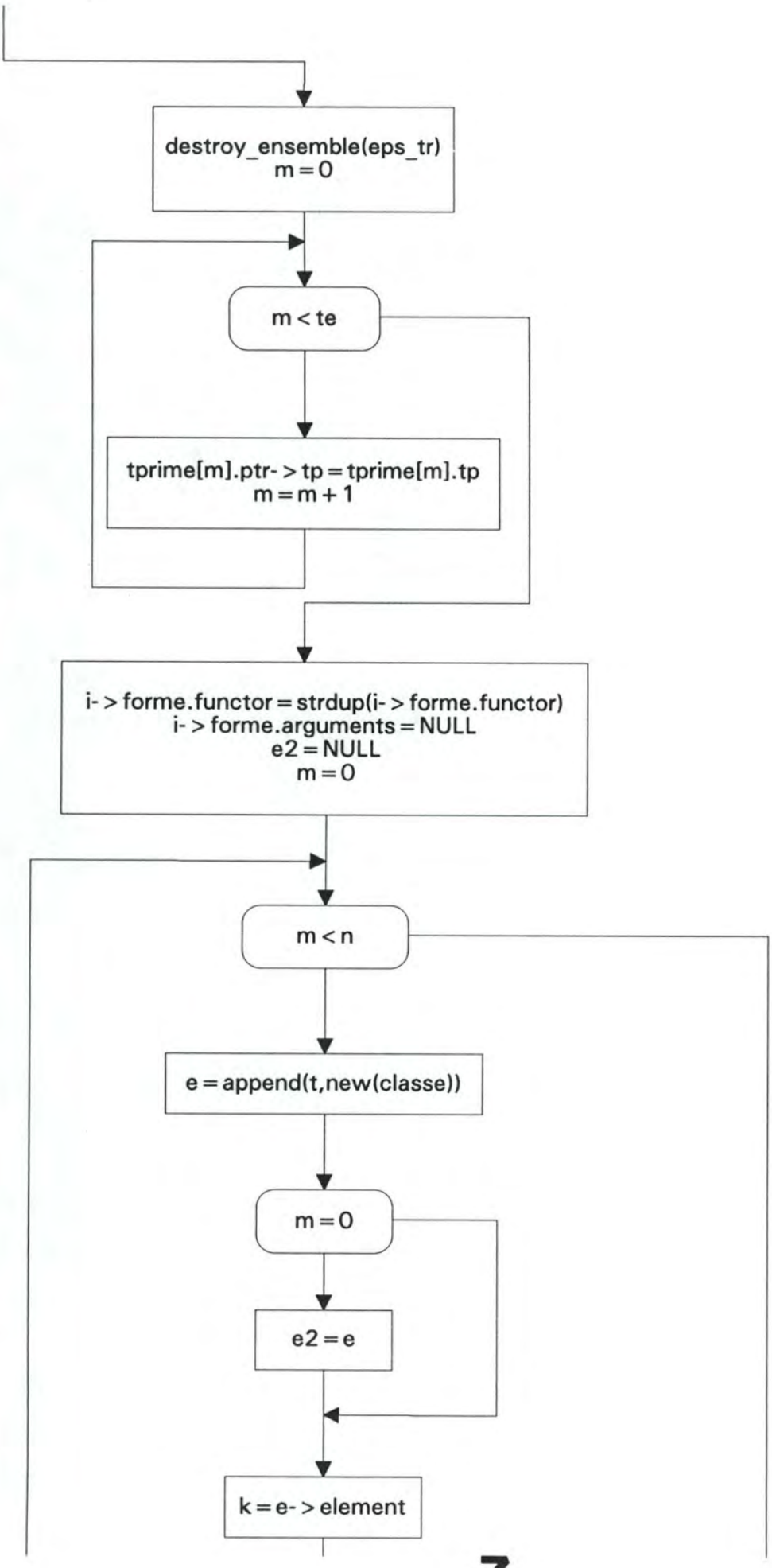
Annexe 2 :
Organigramme de specat

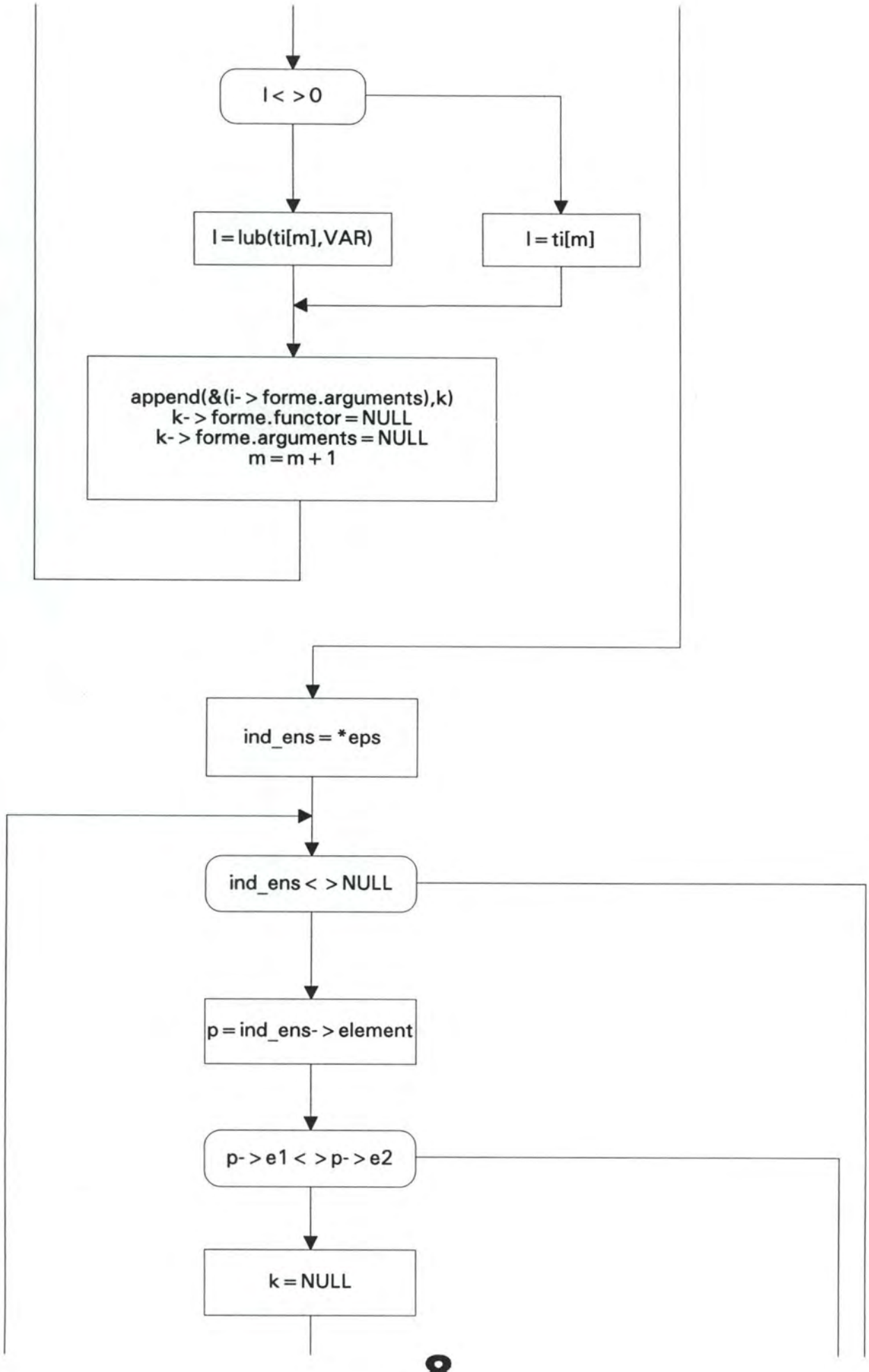
La procédure spécat reçoit un paramètre i correspondant à un terme dont la forme est indéfinie et un autre paramètre j correspondant à un terme possédant une forme. Elle reçoit aussi la substitution abstraite contenant ces termes. Son but est de donner au terme i la même forme que le terme j en vue d'une unification, elle doit recalculer tous les types qui sont susceptibles d'être modifiés.

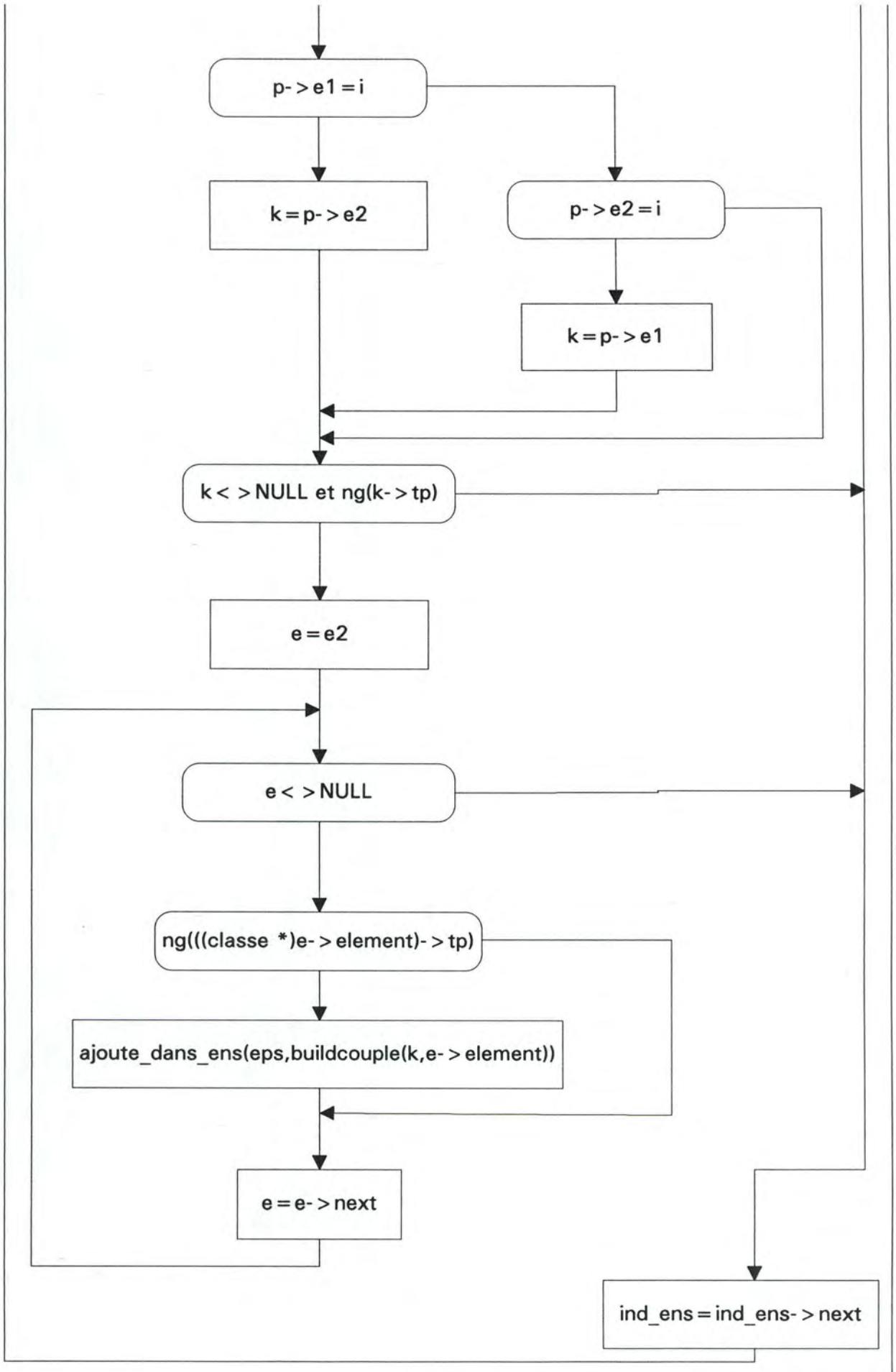

begin
n = taille_ensemble (j- > forme.arguments)
l = estpluspetit (VAR,i- > tp)
te = taille_ensemble (*t)
ti = extr (i- > tp,j- > forme.functor,n)
bot = egalbottom (ti,n)
(bot et non l) ou dansparties (i,j)
renvoyer bottom
tprime [te]
m = 0
ind_ens = *t
ind_ens < > NULL
eps_tr = ps_et(*t,*eps)
ind_ens = *t
tprime[m].ptr = ind_ens- > element
tprime[m].tp = 0xFF
m = m + 1
ind_ens = ind_ens- > next
k = ind_ens- > element
get_tp_from_ptr_tp(tprime,k) = 0xFF
mapile = NULL
etat = 0

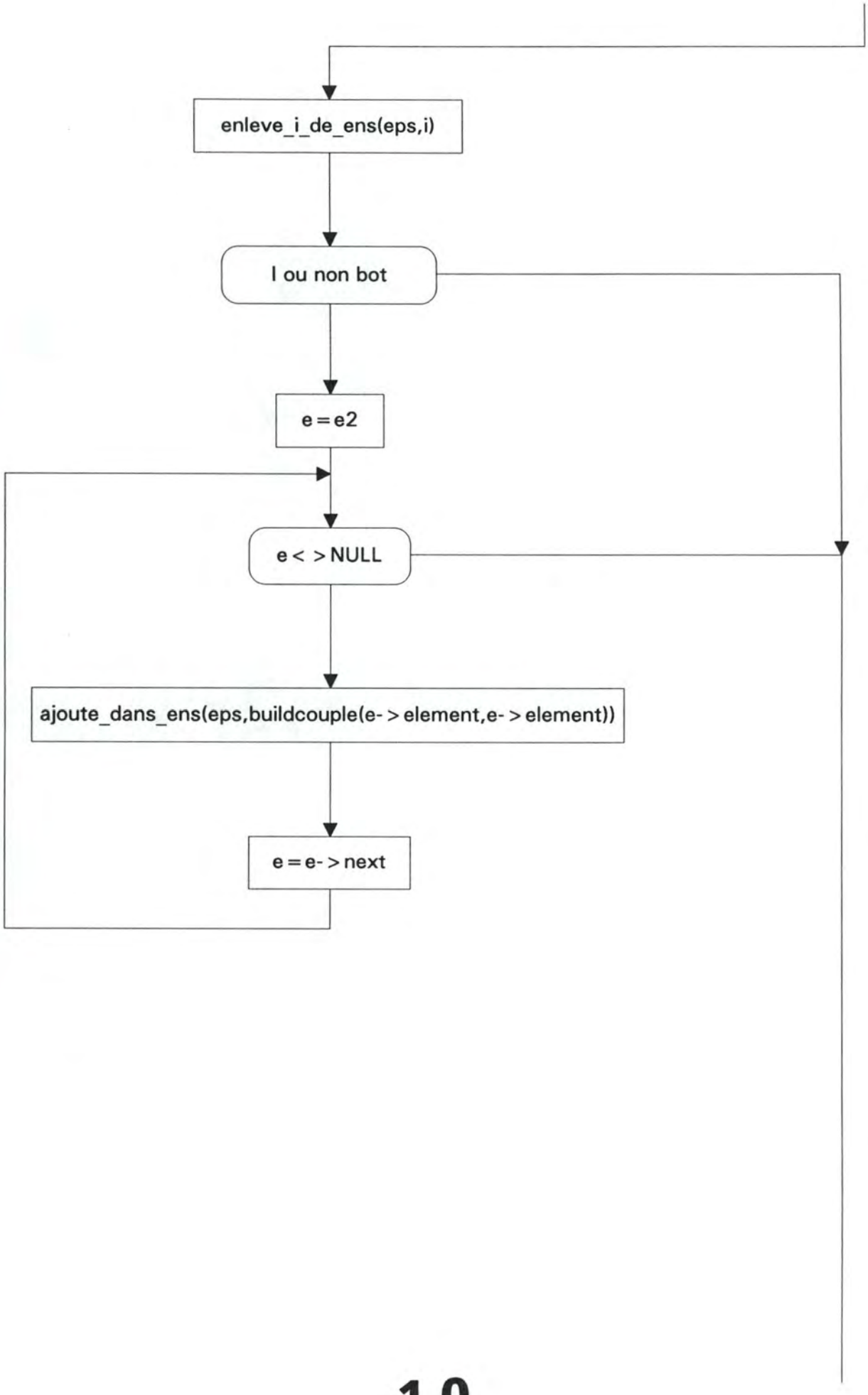

etat < 3
etat = 0
etat = 1
(non ps(i,k,eps_tr))
ou dansparties(k,j)
i = k
res = k- > tp
type = mat(i- > tp,j- > forme.functor,ti,n)
I
res = type
ti2[n] = { VAR, VAR, ..., VAR }
res = lub(type,cons(j- > forme.functor,ti2,n))
non undefined(k)
I
res = k- > tp
narg = k- > forme.arguments
m = taille_ensemble(narg)
z[m]
ind = 0
etat = 2
ti2[n] = { VAR, ..., VAR }
res = iat2(k- > tp,cons(j- > forme.functor,ti2,n))

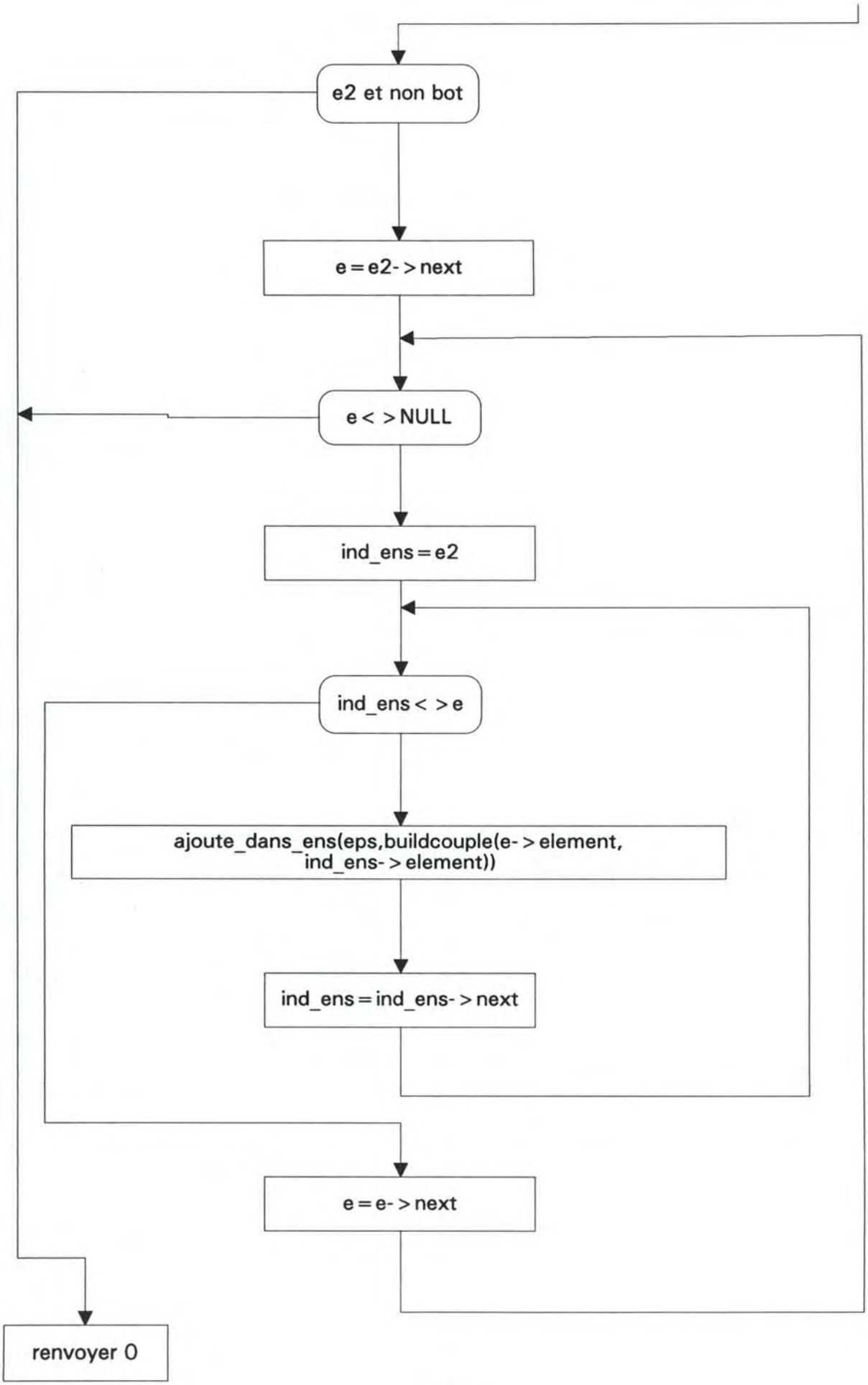

etat = 1
set_tp_of_ptr_tp(tprime,k,res)
mapile < > NULL
depiltppr(&mapile,&k,&narg,&z,&ind)
z[ind] = res
ind = ind + 1
narg = narg- > next
etat = 2
etat = 3
etat = 2
narg < > 0
tmp = get_tp_from_ptr_tp(tprime,k)
res = mat(k- > tp,k- > forme.functor,z,
taille_ensemble(k- > forme.arguments))
etat = 1
tmp < > -1
z[ind] = tmp
ind = ind + 1
narg = narg- > next
empiltppr(&mapile,k,narg,z,ind)
k = narg- > element
etat = 0


destroy_ensemble(eps_tr)
m = 0
m < te
tprime[m].ptr- > tp = tprime[m].tp
m = m + 1
i- > forme.functor = strdup(i- > forme.functor)
i- > forme.arguments = NULL
e2 = NULL
m = 0
m < n
e = append(t,new(classe))
m = 0
e2 = e
k = e- > element


l< >0
l = lub(ti[m],VAR)
l = ti[m]
append(&(i- > forme.arguments),k)
k- > forme.functor = NULL
k- > forme.arguments = NULL
m = m + 1
ind_ens = *eps
ind_ens < > NULL
p = ind_ens- > element
p- > e1 < > p- > e2
k = NULL


p->e1=i
k=p->e2
p->e2=i
k=p->e1
k<>NULL et ng(k->tp)
e=e2
e<>NULL
ng(((classe *)e->element)->tp)
ajoute_dans_ens(eps,buildcouple(k,e->element))
e=e->next
ind_ens=ind_ens->next


enleve_i_de_ens(eps,i)
l ou non bot
e = e2
e < > NULL
ajoute_dans_ens(eps,buildcouple(e- > element,e- > element))
e = e- > next


e2 et non bot
e = e2- > next
e < > NULL
ind_ens = e2
ind_ens < > e
ajoute_dans_ens(eps,buildcouple(e- > element,
ind_ens- > element))
ind_ens = ind_ens- > next
e = e- > next
renvoyer 0

Annexe 3 :

Listing

```
************************************************************************************** abi.c ***************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "messtruc.h"
#include "ensemble.h"

#include "managest.h"
#include "o_operat.h"
#include "o_types.h"
#include "o_access.h"
#include "o_couple.h"
#include "o_ps.h"
#include "sortie.h"
#include "o_sv.h"
#include "abi.h"

void betabi(beta *b)
{
if (abi1((*b).sv,&((*b).t),&((*b).eps))==-1)
          { liberebeta(*b);
            (*b).t=NULL;
            (*b).sv=NULL;
            (*b).eps=NULL; }
}

void betabi2(beta *b,char *f)
{
if (abi2((*b).sv,&((*b).t),&((*b).eps),f)==-1)
          { liberebeta(*b);
            (*b).t=NULL;
            (*b).sv=NULL;
            (*b).eps=NULL; }
}

int abi1(type_ensemble sv,type_ensemble *t,type_ensemble *eps)
{
   struct ptrcouple *a=(struct ptrcouple *) safe_malloc(sizeof(struct ptrcouple));
   type_ensemble liste=NULL; ajoute_dans_ens(&liste,a);
   a->i=((type_sv *)sv->element)->sv;a->j=((type_sv *) sv->next->element)->sv;
   return ualct(&liste,sv,t,eps);
}
```

```
int abi2(type_ensemble sv,type_ensemble *t,type_ensemble *eps,char *functor)
{
int j,tail_sv=taille_ensemble(sv);
type_ensemble ind;
classe *new=(classe *) safe_malloc (sizeof(classe)),**tab;
type_tp *ti=(type_tp *) safe_malloc((tail_sv-1)*sizeof(type_tp));
tab=(classe **) safe_malloc (tail_sv*sizeof(classe *));
for (ind=sv;ind;ind=ind->next)
   {
    if ((j=((type_sv *)ind->element)->xi)!=1)
        ti[j-2]=((classe *) ((type_sv *) ind->element)->sv)->tp;

    tab[j-1]=((type_sv *)ind->element)->sv;
   }

new->tp=cons(functor,ti,tail_sv-1);
new->forme.functor=functor ? strdup(functor) : NULL;
new->forme.arguments=NULL;

for (j=tail_sv-1;j>0;j--) ajoute_dans_ens(&(new->forme.arguments),tab[j]);

ajoute_dans_ens(t,new);

afftabs2(0,2,*t,sv,*eps);

ind=NULL;

append(&ind,buildcouple(*tab,new));

free(tab);
free(ti);

return ualct(&ind,sv,t,eps);
}
```

```
********************************************************************************** abi.h ***************
void betabi(beta *b);
void betabi2(beta *b,char *f);
int abi1(type_ensemble sv,type_ensemble *t,type_ensemble *eps);
int abi2(type_ensemble sv,type_ensemble *t,type_ensemble *eps,char *functor);
```

```
********************************************************************************** betaordr.c **********
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "messtruc.h"
#include "ensemble.h"

#include "o_operat.h"
#include "o_types.h"
#include "stdfunct.h"
#include "o_sv.h"
#include "o_couple.h"
#include "o_ps.h"

#include "betaordr.h"

int cmpbbeta(beta *b1,beta *b2)
{
return cmpbeta((*b1).sv,(*b1).t,(*b1).eps,(*b2).sv,(*b2).t,(*b2).eps);
}

int cmpbbetatrafique(beta *b1,beta *b2)
{
return cmpbetatrafique((*b1).sv,(*b1).t,(*b1).eps,(*b2).sv,(*b2).t,(*b2).eps);
}

int betaorder(beta b1,beta b2)
{
if (b1.sv==NULL && b1.t==NULL && b1.eps==NULL) return 1;
return betainf(b1.sv,b1.t,b1.eps,b2.sv,b2.t,b2.eps);
}

int betainf(type_ensemble sv1,type_ensemble t1,type_ensemble eps1,type_ensemble sv2,type_ensemble t2,type_ensemble eps2)
{
if (taille_ensemble(sv1)!=taille_ensemble(sv2)) return 0;
else
{
struct ptrcouple *c,*c2;
classe *cl,*cl2;
type_ensemble ie,je,ke,ti=NULL;
for (ie=sv2;ie;ie=ie->next)
   if (!est_dans_ens_couples_ptrs(ti,((type_sv *)ie->element)->sv,cl=trouvsv(sv1,((type_sv *) ie->element)->xi)))
      if (!est_dans_liste_couple_1(ti,((type_sv *)ie->element)->sv))
        append(&ti,buildcouple(((type_sv *)ie->element)->sv,cl));
      else goto non;

for (ie=sv1;ie;ie=ie->next)
   if (!est_dans_liste_couple_2(ti,((type_sv *)ie->element)->sv))
      goto non;

for (ie=ti;ie;ie=ie->next)
  { cl=  ((struct ptrcouple *) ie->element)->i;
    cl2= ((struct ptrcouple *) ie->element)->j;
    if (!undefined(cl))
      { if (!memeforme(cl,cl2))
          goto non;
        else for (je=cl->forme.arguments,ke=cl2->forme.arguments;je;je=je->next,ke=ke->next)
```

```
                { c=est_dans_liste_couple_1(ti,je->element);
                  if (!c) append(&ti,buildcouple(je->element,ke->element));
                  else if (c->j!= ke->element) goto non; }
        }
    }

ke=ps_et(t1,eps1);

for (ie=ti;ie;ie=ie->next)
   for (je=ie;je;je=je->next)
     { c=ie->element; c2=je->element;
       if (ps(c->j,c2->j,ke) && undefined(c->i) && undefined(c2->i) && !ps(c->i,c2->i,eps2)) { destroy_ensemble(ke); goto non;
}
     }

destroy_ensemble(ke);

for (ie=ti;ie;ie=ie->next)
  {
   c=ie->element;
   c1=c->i; /* biggest */
   c12=c->j;
   if (!estpluspetit(c12->tp,c1->tp)) goto non;
  }
destroy_ensemble(ti);
return 1;

non:
destroy_ensemble(ti);
return 0;
}

)


int cmpbetatrafique(type_ensemble sv1,type_ensemble t1,type_ensemble eps1,type_ensemble sv2,type_ensemble t2,type_ensemble
eps2)
{
if (taille_ensemble(sv1)!=taille_ensemble(sv2)) return 0;
else
if (sv1==NULL && t1==NULL && eps1==NULL) return 1;
else
{
struct ptrcouple *c;
classe *c1,*c12;
type_ensemble ie,je,ke,ti=NULL;
/*
puts("1r sv");
afftabs2(1,11,t1,sv1,eps1); */
/*for (ie=sv1;ie;ie=ie->next)
   printf("x%d, %d\n",((type_sv *) ie->element)->xi, getpos(t1,((type_sv *) ie->element)->sv));*/
/* puts("2e sv");
afftabs2(2,11,t2,sv2,eps2); */
/*for (ie=sv2;ie;ie=ie->next)
   printf("x%d, %d\n",((type_sv *) ie->element)->xi, getpos(t2,((type_sv *) ie->element)->sv));*/

for (ie=sv2;ie;ie=ie->next)
   if (!est_dans_ens_couples_ptrs(ti,((type_sv *)ie->element)->sv,c1=trouvsv(sv1,((type_sv *) ie->element)->xi)))
```

```
      if (!est_dans_liste_couple_1(ti,((type_sv *)ie->element)->sv) && !est_dans_liste_couple_2(ti,cl))
         append(&ti,buildcouple(((type_sv *)ie->element)->sv,cl));
      else goto non;

for (ie=sv1;ie;ie=ie->next)
   if (!est_dans_liste_couple_2(ti,((type_sv *)ie->element)->sv))
      goto non;

for (ie=ti;ie;ie=ie->next)
  { cl=  ((struct ptrcouple *) ie->element)->i;
    cl2= ((struct ptrcouple *) ie->element)->j;
    if (!undefined(cl))
        if (!nemeforme(cl,cl2))
            goto non;
        else { for (je=cl2->forme.arguments;je;je=je->next)
                   if (nemeforme(cl,je->element)) { puts("bouclage detecte"); goto oui;   /* cons(cons(...)) */ }
               for (je=cl->forme.arguments,ke=cl2->forme.arguments;je;je=je->next,ke=ke->next)
                { c=est_dans_liste_couple_1(ti,je->element);
                  if (!c)
                     if (!est_dans_liste_couple_2(ti,ke->element)) append(&ti,buildcouple(je->element,ke->element));
                     else goto non;
                  else if (c->j != ke->element) goto non; }
             }
    else if (!undefined(cl2)) goto non;
  }

if (taille_ensemble(eps1)!=taille_ensemble(eps2)) goto non;

for (ie=eps1;ie;ie=ie->next)
     { cl= est_dans_liste_couple_2(ti,((struct ptrcouple *) ie->element)->i) ->i;
       cl2=est_dans_liste_couple_2(ti,((struct ptrcouple *) ie->element)->j) ->i;
       if (!ps(cl,cl2,eps2)) goto non;
     }

for (ie=ti;ie;ie=ie->next)
  {
   c=ie->element;
   cl=c->i;
   cl2=c->j;
   if (cl2->tp!=cl->tp) goto non;
  }

oui:
puts("DANS SUSPENDED SELON COMPARAISON TRAFIQUEE ********");
destroy_ensemble(ti);
return 1;

non:
puts("PAS DANS SUSPENDED SELON COMPARAISON TRAFIQUEE ****");
destroy_ensemble(ti);
return 0;
}

}

int cmpbeta(type_ensemble sv1,type_ensemble t1,type_ensemble eps1,type_ensemble sv2,type_ensemble t2,type_ensemble eps2)
{
if (taille_ensemble(sv1)!=taille_ensemble(sv2) || taille_ensemble(t1)!=taille_ensemble(t2)) return 0;
```

```
else
if (sv1==NULL && t1==NULL) return 1;
{
int k=0;
type_ensemble i;
classe *c1,*c2;
struct ptrcouple *p=(struct ptrcouple *)safe_malloc((taille_ensemble(t1)+taille_ensemble(sv1))*sizeof(struct ptrcouple));
for (i=sv1;i;i=i->next)
   if ((c1=trouvsv(sv2,((type_sv *) i->element)->xi))==NULL) goto cloture;
   else if (cmpterms(p,&k,((type_sv *) i->element)->sv,c1)==0) goto cloture;

if (k!=taille_ensemble(t1)) goto cloture;

if (taille_ensemble(eps1)!=taille_ensemble(eps2)) goto cloture;

for (i=eps1;i;i=i->next)

     if ((c1=(classe *) trvcorj(((paire *) i->element)->e1,p,k))==NULL ||
         (c2=(classe *) trvcorj(((paire *) i->element)->e2,p,k))==NULL ||
         !ps(c1,c2,eps2))

             goto cloture;


printf("cmpbeta\n");
return 1;

cloture:
free(p);
return 0;
}

}

int cmpterms(struct ptrcouple *p,int *k,classe *i,classe *j)
{
int l,z;
uactpile *pile=NULL;

debut:
  z=corresp(i,j,p,*k);
  if (z==1) goto zero;
  if (z==2) goto un;
  if ((i->tp!=j->tp) || (taille_ensemble(i->forme.arguments)!=taille_ensemble(j->forme.arguments))) goto zero;
  if ((i->forme.functor && !j->forme.functor) || (!i->forme.functor && j->forme.functor)) goto zero;
  if (i->forme.functor && j->forme.functor)
     if (strcmp(i->forme.functor,j->forme.functor)) goto zero;
  p[*k].i=i;p[(*k)++].j=j;

  l=0;

boucle:
  if (l<taille_ensemble(i->forme.arguments))
    {
     empiluact(&pile,i,j,l);
     i=(classe *) get_nieme(i->forme.arguments,l+1);j=(classe *) get_nieme(j->forme.arguments,l+1);
     goto debut;
    }
```

```
un:
   if (pile)
     { depiluact(&pile,&i,&j,&l);
       l++;
       goto boucle; }
   else
       return 1;

zero:
   libere(pile);
   return 0;
}
```

```
********************************************************************************** betaordr.h **********
int cmpbbeta(beta *b1,beta *b2);
int cmpbbetatrafique(beta *b1,beta *b2);
int cmpbetatrafique(type_ensemble sv1,type_ensemble t1,type_ensemble eps1,type_ensemble sv2,type_ensemble t2,type_ensemble
eps2);
int betaorder(beta b1,beta b2);
int betainf(type_ensemble sv1,type_ensemble t1,type_ensemble eps1,type_ensemble sv2,type_ensemble t2,type_ensemble eps2);
int cmpbeta(type_ensemble sv1,type_ensemble t1,type_ensemble eps1,type_ensemble sv2,type_ensemble t2,type_ensemble eps2);
int cmpterms(struct ptrcouple *p,int *k,classe *i,classe *j);
int cmptermstraf(struct ptrcouple *p,int *k,classe *i,classe *j,type_ensemble eps1,type_ensemble eps2);
```

```
********************************************************************************** defines.h ***********
#define TOPTREE 8
                /* ANY + 1 */
#define ANY     7
#define NOVAR   6
#define LV      5
#define NOLIST  3
#define LIST    4
#define NLV     2
#define VAR     1
#define BOTTOM  0
```

```
********************************************************************************** ensemble.c **********
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "messtruc.h"
#include "ensemble.h"

void set_tp_of_ptr_tp(paire_ptr_tp *tab,classe *ptr,type_tp tp)
{
int i=0;
while (tab[i].ptr!=ptr) i++;
tab[i].tp=tp;
}

type_tp get_tp_from_ptr_tp(paire_ptr_tp *tab,classe *ptr)
{
int i;
for (i=0;tab[i].ptr!=ptr;i++) ;
return tab[i].tp;
}

classe *get_ptr_from_ptr_tp(paire_ptr_tp *tab,type_tp tp)
{
int i=0;
while (tab[i].tp!=tp) i++;
return tab[i].ptr;
}

void kill_ensemble(type_ensemble e)
{
type_ensemble p;
while (p=e)
  { e=e->next; free(p); }
}

type_element *est_dans_ens(type_ensemble e,void *element)
{
while (e)
  { if (e->element==element) return e;
    e = e -> next;
  }
return NULL;
}

void enleve_de_ens(type_ensemble *ensemble,void *element)
{ type_ensemble *ptr=ensemble, ptr2, ens=*ensemble;
  while (ens)
        if (ens->element==element)
          { ptr2=ens;
            *ptr=ens->next;
            free(ptr2);
            return; }
        else
          { ptr=&(ens->next);
            ens=ens->next; }
}
```

```
void enleve_elt_de_ens(type_ensemble *ensemble,type_element *element)
{ type_ensemble ptr, ens=*ensemble;
  if (ens==NULL || element==NULL) return;
  if (ens==element)
     { *ensemble=ens->next;free(ens->element); free(ens); }
  else
     {
       while (ens && ens->next != element) ens=ens->next;
       if (ens && ens->next==element)
          {
           ptr=ens->next;
           ens->next=ens->next->next;
           free(ptr->element);
           free(ptr);
          }
     }
}

int taille_ensemble(type_ensemble ensemble)
{
 int i=0;
 for (i=0;ensemble;i++) ensemble=ensemble->next;
 return i;
}

void ajoute_dans_ens(type_ensemble *ensemble,void *element)
{
type_element *w=(type_element *)safe_malloc(sizeof(type_element));
w->element=element;
w->next=*ensemble;
*ensemble=w;
}

int est_dans_ens_paires(type_ensemble eps,classe *e1,classe *e2)
{
paire *p;
while(eps)
  { p = (paire *) eps->element;
    if ((p->e1==e1 && p->e2==e2) || (p->e1==e2 && p->e2==e1))
       return 1;
    eps=eps->next;
  }
return 0;
}

void enleve_paire_de_ens(type_ensemble *ensemble,void *z1,void *z2)
{ type_ensemble *ptr=ensemble,ptr2,ens=*ensemble;
  paire *zut;
  while (ens)
    {
       zut=ens->element;
       if ((zut->e1==z1 && zut->e2==z2) || (zut->e2==z1 && zut->e1==z2))
         { ptr2=ens;
           *ptr=ens->next;
           free(ens->element);
           free(ptr2);
           return; }
       else
```

```
        { ptr=&(ens->next);
          ens=ens->next; }
    }
}

void get_selection1(type_ensemble ens,type_ensemble *ens_sel,int (*f)())
{
while (ens)
    { if (f(ens->element))
        ajoute_dans_ens(ens_sel,ens->element);
      ens=ens->next; }
}

paire *buildpaire(void *i,void *j)
{
paire *pa=(paire *) safe_malloc (sizeof(paire));
if (i>j)
  {
   pa->e1 = j;
   pa->e2 = i;
  }
else
  {
   pa->e1 = i;
   pa->e2 = j;
  }
return pa;
}


void destroy_non_sous_ens(type_ensemble eps,type_ensemble p1)
{
while (eps)
  { if (!est_dans_ens(p1,eps->element)) free(eps->element);
    eps=eps->next;
  }
}

type_ensemble union_disj(type_ensemble ps1,type_ensemble ps2)
{
type_ensemble i=ps1;
if (!ps1) return ps2;
if (!ps2) return ps1;
while (i->next) i=i->next;
/* i->next == NULL */
i->next=ps2;
return ps1;
}

void *get_nieme(type_ensemble e,int i)
{
 while (e && --i) e=e->next;

 return e ? e->element : NULL;
}

type_element *get_nieme_element(type_ensemble e,int i)
{
```

```
 while (e && --i) e=e->next;

 return e;
}

type_ensemble append(type_ensemble *e, void *elt)
{
type_ensemble ptr=(type_element *) safe_malloc(sizeof(type_element)),z=*e;
ptr->next=NULL;
ptr->element=elt;
if (!*e) *e=ptr;
else
   { while (z->next) z = z->next;
     z->next=ptr; }
return ptr;
}

void destroy_ensemble(type_ensemble e)
{
type_ensemble p;
while (p=e)
  { e=e->next; free(p->element);free(p); }
}

type_ensemble *adrlast(type_ensemble *i)
{
if (*i==NULL) return i;
else
  { type_ensemble j=*i;
    while (j->next) j=j->next;
    return &(j->next);
  }
}

int getpos(type_ensemble e,void *elt)
{
int i;
for (i=1;e;e=e->next,i++)
    if (e->element==elt) return i;
return -1;
}

int get_pos_adr(type_ensemble i, type_ensemble j)
{
int cpt;
for (cpt=0;i;i=i->next,cpt++)
         if (i==j) return cpt;
return -1;
}

type_ensemble build_liste(int j)
{
type_ensemble e=NULL,f;
while (j--)
   { f=(type_element *) safe_malloc(sizeof(type_element));
     f->next=e;
     f->element=NULL;
     e=f; }
```

```
return e;
}
```

```
********************************************************************************************** ensemble.h **********
void enleve_elt_de_ens(type_ensemble *e,type_element *p);
void set_tp_of_ptr_tp(paire_ptr_tp *tab,classe *ptr,type_tp tp);
type_tp get_tp_from_ptr_tp(paire_ptr_tp *tab,classe *ptr);
classe *get_ptr_from_ptr_tp(paire_ptr_tp *tab,type_tp tp);
void kill_ensemble(type_ensemble e);
type_element *est_dans_ens(type_ensemble e,void *element);
void enleve_de_ens(type_ensemble *ensemble,void *element);
int taille_ensemble(type_ensemble ensemble);
void ajoute_dans_ens(type_ensemble *ensemble,void *element);
int est_dans_ens_paires(type_ensemble eps,classe *e1,classe *e2);
void enleve_paire_de_ens(type_ensemble *ensemble,void *z1,void *z2);
void get_selection1(type_ensemble ens,type_ensemble *ens_sel,int (*f)());
paire *buildpaire(void *i1,void *i2);
void destroy_non_sous_ens(type_ensemble eps,type_ensemble p1);
type_ensemble union_disj(type_ensemble ps1,type_ensemble ps2);
void *get_nieme(type_ensemble e,int i);
type_element *get_nieme_element(type_ensemble e,int i);
type_ensemble append(type_ensemble *e, void *elt);
void destroy_ensemble(type_ensemble e);
type_ensemble *adrlast(type_ensemble *i);
int getpos(type_ensemble e,void *elt);
int get_pos_adr(type_ensemble i, type_ensemble j);
type_ensemble build_liste(int j);
```

```
********************************************************************************** essais.c ************
#include <string.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "defines.h"

#include "messtruc.h"

#include "o_operat.h"
#include "abi.h"
#include "ext2.h"
#include "o_types.h"
#include "majlub.h"
#include "solve.h"
#include "sortie.h"
#include "o_sv.h"
#include "essais.h"
#include "litfich.h"
#include "betaordr.h"
#include "o_ps.h"
#include "managest.h"

#define NBESSAIS 10


void essais(char *fn)
{
int rep,p[10],i,j=i=0,indice;

char buf[30];
beta b1,b2,bpr;

system("clear");

printf ("Quelle est la fonction a tester :\n\n");
printf ("   1. majlub\n");
printf ("   2. extension 2\n");
printf ("   3. abi\n");
printf ("   4. abi2\n");
printf ("   5. uact\n");
printf ("   6. restriction\n");
printf ("   7. extension\n");
printf ("   8. Comparaison de betas\n");
printf ("   9. Psrecover\n");
printf ("  10. UaT\n");

do
  {
   printf("\nVotre choix :");
   scanf("%d",&rep);
   if (rep<=0 || rep > NBESSAIS) printf("Votre choix doit etre compris entre 1 et %d\n\n",NBESSAIS);
 } while (rep <=0 || rep > NBESSAIS);

system("clear");

switch(rep)
```

```
{
 case 1 : /* test majlub */
  {
   litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));
   printf("Nom du deuxieme fichier : ");
   scanf ("%29s",buf);
   litfich(buf,&(b2.sv),&(b2.t),&(b2.eps));

   afftabs2(1,55,b1.t,b1.sv,b1.eps);
   afftabs2(1,55,b2.t,b2.sv,b2.eps);

   bpr=betalub(b1,b2);

   afftabs2(2,55,bpr.t,bpr.sv,bpr.eps);

   liberebeta(b1);
   liberebeta(b2);
   liberebeta(bpr);

   break;
  }

 case 2 : /* test extension 2 */
  {
   litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));

   printf("Nom du deuxieme fichier : ");
   scanf ("%29s",buf);
   litfich(buf,&(b2.sv),&(b2.t),&(b2.eps));

   afftabs2(0,55,b1.t,b1.sv,b1.eps);
   afftabs2(1,55,b2.t,b2.sv,b2.eps);

   bpr=EXTB(b1,b2);

   afftabs2(2,55,bpr.t,bpr.sv,bpr.eps);

   liberebeta(bpr);

   break;
  }

 case 4 : /* test abi 2 */
  {
   litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));

   printf("Entrez le foncteur : ");
   scanf("%s",buf);

   afftabs2(0,55,b1.t,b1.sv,b1.eps);

   betabi2(&b1,buf);

   afftabs2(1,55,b1.t,b1.sv,b1.eps);
   liberebeta(b1);
   break;
  }
```

```
case 3 : /* test abi */
 {
  litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));

  afftabs2(0,55,b1.t,b1.sv,b1.eps);

  betabi(&b1);

  afftabs2(1,55,b1.t,b1.sv,b1.eps);

  liberebeta(b1);

  break;
}

case 5 : /* test uact */
 {
  classe *c1,*c2;
  litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));

  afftabs2(0,55,b1.t,b1.sv,b1.eps);

  printf("Variables pour l'unification ?\n");
  do
    {
     scanf("%d",&j);
     if (j>0) p[i++]=j;
    }
  while (i<2);

  c1=trouvsv(b1.sv,p[0]);c2=trouvsv(b1.sv,p[1]);
  if (c1 && c2)
     uact(c1,c2,b1.sv,&(b1.t),&(b1.eps));
  else
     printf("Essais - test Uact - Variables non trouvées\n");

  afftabs2(1,55,b1.t,b1.sv,b1.eps);

  liberebeta(b1);
  break;
}

case 6 : /* test restriction */
 {
  litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));
  afftabs2(0,55,b1.t,b1.sv,b1.eps);
  printf("Variables pour la restriction ?\n");
  do
     scanf("%d",p+i);
  while (p[i++]>0);
  i--;

  printf("i=%d\n",i);
  restr1(p,i,&(b1.sv),&(b1.t),&(b1.eps),0);

  afftabs2(1,55,b1.t,b1.sv,b1.eps);
  liberebeta(b1);
  break;
```

```
  }

  case 7 : /* test extension */
   {
    litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));
    afftabs2(0,55,b1.t,b1.sv,b1.eps);

    printf("Nombre de variables pour l\'extension :\n");
    scanf ("%d",&indice);

    extension(indice,&(b1.sv),&(b1.t),&(b1.eps));

    afftabs2(1,55,b1.t,b1.sv,b1.eps);
    liberebeta(b1);
    break;
   }
  case 8 : /* test comparaison de betas */
   {
    litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));
    printf("Nom du deuxieme fichier : ");
    scanf ("%29s",buf);
    litfich(buf,&(b2.sv),&(b2.t),&(b2.eps));

    afftabs2(0,55,b1.t,b1.sv,b1.eps);
    afftabs2(1,55,b2.t,b2.sv,b2.eps);

    printf("Resultat : %d\n",betaorder (b1,b2));

    liberebeta(b1);
    liberebeta(b2);
    break;
   }
  case 9 : /* test psrecover */
   {
    litfich(fn,&(b1.sv),&(b1.t),&(b1.eps));

    afftabs2(0,55,b1.t,b1.sv,b1.eps);

    psrecover(b1.t,&(b1.eps));

    afftabs2(1,55,b1.t,b1.sv,b1.eps);

    liberebeta(b1);
    break;
   }
  case 10 : /* test UaT */
   {
    type_tp t1,t2;
    int i1,i2;
    printf("Entrez le premier type (sous forme d'entier) :");
    scanf ("%d",&i1);
    printf("Entrez le second type (sous forme d'entier) : ");
    scanf ("%d",&i2);
    t1=i1;
    t2=i2;
    printf("UaT(%x,%x)=%x\n",i1,i2,(int)uat(t1,t2));
   }
 }
```

}

```
********************************************************************************** essais.h ************
void essais(char *fn);
```

```
********************************************************************************** ext2.c **************
#include <stdio.h>
#include <stdlib.h>

#include "messtruc.h"
#include "ensemble.h"

#include "o_sv.h"
#include "o_operat.h"
#include "sortie.h"
#include "o_couple.h"

#include "ext2.h"

int extension2(type_ensemble sv1,type_ensemble sv2,type_ensemble t1,type_ensemble t2,type_ensemble eps1,type_ensemble
eps2,type_ensemble *svpr, type_ensemble *tpr,type_ensemble *eps)
{
type_ensemble l,liste=NULL;
classe *p;

for (*svpr=sv1,l=sv2;l;l=l->next)
   if ((p=trouvsv(sv1,((type_sv *)(l->element))->xi))!=NULL)
      append(&liste,buildcouple(p,((type_sv *) (l->element))->sv));
   else { puts("ERREUR EXTENSION2  - ext2.c"); exit(1); }

destroy_ensemble(sv2);

*tpr=union_disj(t1,t2);
*eps=union_disj(eps1,eps2);

return ualct(&liste,*svpr,tpr,eps);
}
```

```
********************************************************************************** ext2.h **************
int extension2(type_ensemble sv1,type_ensemble sv2,type_ensemble t1,type_ensemble t2,type_ensemble eps1,type_ensemble
eps2,type_ensemble *svpr, type_ensemble *tpr,type_ensemble *eps);
```

```
********************************************************************************** ia.c ***************
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

#include "messtruc.h"

#include "ensemble.h"
#include "o_types.h"
#include "solve.h"
#include "o_ps.h"
#include "litfich.h"
#include "stdfunct.h"
#include "sortie.h"
#include "essais.h"
#include "norm.h"

type_ensemble E_clauses=NULL;

int line_number,Glevel,iter_count,Glevelcroc;

extern FILE *yyin,*yyout;
extern int yyparse(void);

void main(int argc,char *argv[])
{
  type_ensemble sv, t, eps, i,cl=NULL;
  int sw,nquest;
  beta z;
  char *buf=(char *)safe_malloc(1000),*prologfile,*betafile,*quest;
  time_t hdeb,hfin;

  if (argc=1)
    { printf("Nom du programme prolog : ");scanf("%999s",buf);prologfile=strdup(buf);
      printf("Nom du fichier beta : ");scanf("%999s",buf);betafile=strdup(buf);
      printf("Foncteur : ");scanf("%999s",buf);quest=strdup(buf);
      printf("Arite : ");scanf("%d",&nquest); }
  else if (argc=5)
        { prologfile=argv[1];betafile=argv[2];quest=argv[3];nquest=atoi(argv[4]); }
       else
        { printf("Usage : ia\nou      ia prolog beta foncteur arite\n"); exit(1); }

  if (empty(quest)) { printf("Requete vide\n"); exit(1); }

  printf("Use essais(0/1) : ");fflush(stdout);
  scanf ("%d",&sw);

  if (!sw)
     {
      yyout=fopen("outpars.tmp","wt");

      Glevel=Glevelcroc=0;
      line_number=1;

      puts(prologfile);

      if (yyin = fopen(prologfile,"rt"))
```

```
        yyparse();
      else {
           printf("Prolog file %s not found\n",prologfile);
           exit(1);
          }
      fclose (yyin);
      fclose (yyout);

      free(prologfile);

      printf("Parsed Ok\n");

      free(buf);

      litfich (betafile,&sv,&t,&eps);

      printf("litfich\n");

      z.sv=sv;z.t=t;z.eps=eps;

      for (i=t;i;i=i->next) if (!ps(i->element,i->element,z.eps)) addps(&(z.eps),i->element,i->element);
      removpsfaux(&(z.eps));

      printf("Normalisation\n");

      cl=normalisation(&E_clauses,quest,nquest,"outpars.tmp");

      hdeb=time(NULL);

      z=solve(z,cl);

      hfin=time(NULL);

      afftabs(z.sv,z.t,z.eps);

      printf("Nombre d'iterations : %d\nTemps d'execution : %ld seconde(s)\n",iter_count,hfin-hdeb);
     }
   else
     {
      free(buf);
      essais(betafile);
     }
}
```

```
********************************************************************************** litfich.c ***********
#define NEWLINE 10

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#include "defines.h"

#include "messtruc.h"

#include "litfichm.h"
#include "ensemble.h"
#include "o_ps.h"
#include "o_sv.h"
#include "stdfunct.h"
#include "o_types.h"
#include "litfich.h"

FILE *fich;

void skipline(int *c,int *nl)
{
  if (*c!=NEWLINE) while ((*c=getc(fich)!=NEWLINE) && !feof(fich)) ;
  (*nl)++;
}

void erreurlig(int *nl)
{
  printf("Erreur a la ligne : %d\n",*nl);
  exit(1);
}

int getpo(int *c,int *nl)
{
if (*c!=40)
   return 0;
do *c=getc(fich); while ((*c<=0x20) && !feof(fich) && *c!=NEWLINE);
if (feof(fich)) { printf("Unexpected feof at line %d\n",*nl); exit(1); }
else if (*c==NEWLINE) { printf("Unexpected eol at line %d\n",(*nl)++); exit(1); }
    else if (!isdigit(*c)) { printf("Expected integer at line %d\n",*nl); exit(1); }
return 1;
}

int getpf(int *c,int *nl)
{
if (*c!=41)
   return 0;
while ((*c=getc(fich))<=0x20 && !feof(fich)) if (*c==NEWLINE) (*nl)++;
if (!feof(fich)) if (*c!=40) return 0;
return 1;
}

int getint(int *res,int *c)
{
int i=0;
char tab[10];
```

```
if (!isdigit(*c)) return -1;
tab[0]=*c;
while (isdigit(*c=getc(fich)) && !feof(fich)) tab[++i]=*c;
tab[++i]=NULL;
*res=atoi(tab);
while (*c==0x20 && !feof(fich)) *c=getc(fich);
if (feof(fich)) { printf("Unexpected eof\n"); exit(1); }
return 0;
}

void getvirg(int *c,int *nl)
{
while (*c==0x20 && !feof(fich)) *c=getc(fich);
if (*c!=44) { printf("Expected , at line %d\n",*nl); exit(1); }
do *c=getc(fich); while (*c==0x20 && !feof(fich));
if (feof(fich)) { printf("Unexpected feof at line %d\n",*nl); exit(1); }
if (!isdigit(*c)) { printf("Expected integer at line %d\n",*nl); exit(1); }
}

int isalpdig(int i)
{ return (isalpha(i) || isdigit(i)); }

int getstring(char *str,int *c)
{
int i=0;
if (!isalpha(*c)) return -1;
str[i++]=*c;
while(isalpdig(*c=getc(fich)) && i<20 && !feof(fich)) str[i++]=*c;
if (i=20) printf("Max 19 lettres pour un id., pris les 19 premieres\n");
else str[i]=NULL;
while (*c==0x20 && !feof(fich)) *c=getc(fich);
return 0;
}

void litfich(char *nomfich,type_ensemble *sv,type_ensemble *t,type_ensemble *eps)
{
if (fich=fopen(nomfich,"rt"))
  { int etape=0, c, nl=1,i,j;
    type_ensemble ie,je;
    type_sv *psv;
    classe *cl,*cl2;
    *sv=*t=*eps=NULL;
    while ((c=getc(fich)) && !feof(fich))
       if (c=NEWLINE) nl++;
       else
       if (isalpha(c) || isdigit(c) || c=40)
         switch(etape) {
           case 0: if (isdigit(c)) etape = 1;
                   else
                      {
                       if (c!=120) erreurlig(&nl); /* 'x' */
                       else
                       { int i;
                         c=getc(fich);
                         if (getint(&i,&c)==-1) erreurlig(&nl);
                         else
                            { while (c==0x20) c=getc(fich);
                              if (getint(&j,&c)!=-1)
```

```
                    { if (est_dans_sv_gauche(*sv,i)) { printf ("Variable presente deux fois dans le beta in :
x%d\n",i); exit(1); }
                      setsv (sv,i,t,j);
                      skipline(&c,&nl); }
                    else erreurlig(&nl);
                  }
                }
                break;
              }
        case 1:if (isalpha(c)) etape=2;
               else if (c==40) etape=3;
               else
               {
                int i;char f[20];
                if (getint(&i,&c)!=-1)
                  if (getstring(f,&c)!=-1)
                    { if (set_type_classe(t,i,get_type(f)) == -1) erreurlig(&nl);
                      skipline(&c,&nl); }
                  else erreurlig(&nl);
                else erreurlig(&nl);
                break;
               }
        case 2:
       trtfrm: if (c==40) etape=3;
               else
                {
                 char f[20];
                 if (c='f')
                   {
                    getstring(f,&c);
                    if (strcmp(f,"frm")) erreurlig(&nl);
                    else goto trtfrm;
                   }
                 else
                   {
                    int i,j=0,z[1000];
                    if (getint(&i,&c)!=-1)
                      {
                       f[0]=NULL;
                       if (isalpha(c)) getstring(f,&c);
                       while (isdigit(c)) getint(&z[j++],&c);
                       if (!j && !f[0]) erreurlig(&nl);
                       else
                         if (cl=(classe *) get_nieme(*t,i))
                           if (cl->forme.functor || cl->forme.arguments)
                               { printf ("Classe %d doublement definie\n",i); erreurlig(&nl); }
                           else
                               {
                                cl->forme.functor=strdup(f);
                                for (i=0,cl->forme.arguments=ie=build_liste(j);ie;ie=ie->next,i++)
                                   ie->element=get_nieme(*t,z[i]);
                                skipline(&c,&nl);
                               }
                         else
                           { printf ("Classe %d indefinie\n",i); erreurlig(&nl); }
                      }
                    else erreurlig(&nl);
                    break;
```

```
                    }
                  }
        case 3:{
                int j,k;
                while (getpo(&c,&nl))
                  { if (getint(&j,&c)==-1) { puts("Expected integer\n"); erreurlig(&nl); }
                    getvirg(&c,&nl);
                    if (getint(&k,&c)==-1) { puts("Expected integer\n"); erreurlig(&nl); }
                    if (getpf(&c,&nl)==0)  { puts("Expected \')\'\n");   erreurlig(&nl); }
                    else
                      {
                       cl =(classe *) get_nieme(*t,j);
                       cl2=(classe *) get_nieme(*t,k);
                       if (cl && cl2)
                          if (!ps(cl,cl2,*eps)) addps(eps,cl,cl2);
                          else ;
                       else { if (!cl)
                                 printf("Classe %d indefinie\n",j);
                              if (!cl2)
                                 printf("Classe %d indefinie\n",k);
                              erreurlig(&nl);
                            }
                      }
                  }
              }
        }
   fclose(fich);
   { int error = 0;
   for (ie=*sv,i=0;ie;ie=ie->next,i++)
      if ((psv=ie->element)==NULL)
        { printf("sv %d indefini\n",i); error=1; }
      else if (((type_sv *) psv)->sv==NULL)
             { printf("classe du sv %d indefinie\n",i); error=1; }
   for (ie=*t,i=1;ie;ie=ie->next,i++)
      if ((cl=ie->element) == NULL) { printf("Classe %d indefinie\n",i); error=1; }
      else
        { if (cl->tp==-1) { printf("Type %d indefini\n",i); error=1; }
          if (cl->forme.functor==NULL) printf("Foncteur %d indefini\n",i);
          for (je=cl->forme.arguments,j=1;je;je=je->next,j++) if (je->element==NULL) { printf("%deme element du foncteur %d
indefini\n",j,i);error=1; } }
   for (ie=*sv,i=0;ie;ie=ie->next,i++)
             printf("sv[%d]=%x , xi=%d, sv=%x\n",i,ie->element,((type_sv *) ie->element)->xi,((type_sv *) ie->element)-
>sv);
   for (ie=*t,i=0;ie;ie=ie->next,i++)
             printf("t[%d]=%x , tp=%d, functor=%s, args=%x\n",i,ie->element,(int)((classe *) ie->element)->tp,((classe *)
ie->element)->forme.functor,((classe *) ie->element)->forme.arguments);
   if (error) exit(1);
   }
  }
else
  { printf("Fichier non trouve\n");exit(1); }
}
```

```
********************************************************************************** litfich.h ***********
void skipline(int *c,int *nl);
void erreurlig(int *nl);
int getpo(int *c,int *nl);
int getpf(int *c,int *nl);
int getint(int *res,int *c);
void getvirg(int *c,int *nl);
int isalpdig(int i);
int getstring(char *str,int *c);
void litfich(char *nomfich,type_ensemble *sv,type_ensemble *t,type_ensemble *eps);
```

```
********************************************************************************** litfichm.c **********
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "defines.h"

#include "messtruc.h"

#include "ensemble.h"
#include "o_sv.h"
#include "litfichm.h"

type_tp get_type(char *f)
{
 if (strcmp(f,"any")==0)    return ANY;
 if (strcmp(f,"bottom")==0) return BOTTOM;
 if (strcmp(f,"list")==0)   return LIST;
 if (strcmp(f,"nolist")==0) return NOLIST;
 if (strcmp(f,"var")==0)    return VAR;
 if (strcmp(f,"novar")==0)  return NOVAR;
 if (strcmp(f,"lv")==0)     return LV;
 if (strcmp(f,"nlv")==0)    return NLV;
 return NULL;
}

void setsv(type_ensemble *sv, int indsv, type_ensemble *t, int indclasse)
{
 classe *ptrcl;
 type_sv *varsv;
 if ((ptrcl=trouvsv(*sv,indsv))==NULL)
    { varsv=(type_sv *) safe_malloc(sizeof(type_sv));
      varsv->xi=indsv;
      varsv->sv=prolonj_classe(t,indclasse);
      ajoute_dans_ens(sv,varsv);
    }
 else
    if (ptrcl!=get_nieme(*t,indclasse))
     { printf("Erreur : deux classes differentes pour x%d\n",indsv); exit(1); }
}

classe *prolonj_classe(type_ensemble *t,int j)
{
 int r=0;
 type_ensemble i,*k=t,current;
 classe *cl;
 for (r=0,i=*t;r<j;r++,i=i->next)
    { if (i==NULL)
        { *k=i = (type_element *) safe_malloc(sizeof(type_element));
          (*k)->element=(*k)->next=NULL;
          k=&((*k)->next); }
      else
         k=&(i->next);
      current=i; }

 if (current->element == NULL)
   {
    current->element=cl = (classe *) safe_malloc(sizeof(classe));
```

```
    cl->tp=-1;
    cl->forme.functor=NULL;
    cl->forme.arguments=NULL;
   }
 return current->element;
}

int set_type_classe(type_ensemble *t,int i,type_tp l)
{
 classe *cl;
 type_ensemble k=*t;
 if (taille_ensemble(*t)<i)
   {
    int j=1;
    type_ensemble *z=t;
    while (i != j && k) { z=&(k->next);k=k->next; j++; }
    if (i==j)
        if (!k) { k=*z = (type_element *) safe_malloc (sizeof(type_element));
                  (*z)->element=(*z)->next=NULL; }
        else ;
    else
        while (j<=i) { k=*z = (type_element *) safe_malloc (sizeof(type_element));
                       (*z)->element=(*z)->next=NULL;j++; z=&((*z)->next); }
 }
 else
   k=get_nieme_element(*t,i);
 cl=k->element;
 if (cl==NULL)
    {
     cl = (classe *) safe_malloc(sizeof(classe));
     cl->tp=l;
     cl->forme.functor=NULL;
     cl->forme.arguments=NULL;
     k->element=cl;
    }
 else if (cl->tp==-1) cl->tp=l;
 return cl->tp==l ? 0 : -1;
}
```

```
********************************************************************************************** litfichm.h **********
type_tp get_type(char *f);
void setsv(type_ensemble *sv, int indsv, type_ensemble *t, int indclasse);
classe *prolonj_classe(type_ensemble *t,int j);
int set_type_classe(type_ensemble *t,int i,type_tp l);
```

```
********************************************************************************** majlub.c ************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "defines.h"

#include "messtruc.h"

#include "ensemble.h"
#include "o_operat.h"
#include "o_types.h"
#include "stdfunct.h"
#include "managest.h"
#include "o_ps.h"
#include "o_sv.h"
#include "sortie.h"
#include "majlub.h"

int addf(type_ensemble *f,classe *i,classe *j,classe **cl)
{
/* renvoie vrai si un ajout reel a ete fait */
type_ensemble ind,*p=f;
triplet_ptrs *elt;

for (ind=*f;ind;p=&(ind->next),ind=*p)
  {
    elt=ind->element;
    if (elt->e1=i && elt->e2=j) { *cl=elt->e3; return 0; }
  }

elt=(triplet_ptrs *) safe_malloc (sizeof(triplet_ptrs));
elt->e1=i;elt->e2=j;elt->e3=*cl=(classe *) safe_malloc(sizeof(classe));
append(p,elt);
return 1;
}

void *in_f(type_ensemble f,classe *i,classe *j)
{
type_ensemble ind;
triplet_ptrs *elt;

for (ind=f;ind;ind=ind->next)
  {
    elt=ind->element;
    if (elt->e1=i && elt->e2=j) return elt->e3;
  }

return NULL;
}

int infrm(classe *i,classe *j)
{
if (est_dans_ens(i->forme.arguments,j)) return 1;
if (est_dans_ens(j->forme.arguments,i)) return 1;
return 0;
}
```

```
void maj(type_ensemble sv1,type_ensemble sv2,type_ensemble t1,type_ensemble t2,type_ensemble eps1,type_ensemble
eps2,type_ensemble *svprim,type_ensemble *tprim,type_ensemble *eps)
{
classe *ci,*cj,*cl,*cl2;
type_ensemble i,j,f=NULL,l,m,*ens,ps1,ps2;
uactpile *pile=NULL;
type_sv *ptrsv;
int k,etat;
*svprim=NULL;
for (i=sv1;i;i=i->next)
   if ((ptrsv=est_dans_sv_gauche(sv2,((type_sv *) i->element)->xi))!=NULL)
     { addf(&f,((type_sv *) i->element)->sv,ptrsv->sv,&cl);
       addsv(svprim,((type_sv *)(i->element))->xi,cl); }

for (l=f;l;l=l->next)
  { ci=((triplet_ptrs *)(l->element))->e1; cj=((triplet_ptrs *)(l->element))->e2;
    etat=0;
    while (etat < 3)
      switch(etat) {

      case 0: if (ci->forme.arguments && memeforme(ci,cj))
                  { k=0; etat=1; }
              else { etat=2; break; }

      case 1: if (k<taille_ensemble(ci->forme.arguments))
                  { if (addf(&f,get_nieme(ci->forme.arguments,k+1),get_nieme(cj->forme.arguments,k+1),&cl))
                      { empiluact(&pile,ci,cj,k);
                        ci=(classe *) get_nieme(ci->forme.arguments,k+1);
                        cj=(classe *) get_nieme(cj->forme.arguments,k+1);
                        etat=0; }
                    else k++;
                    break; }
              else etat=2;

      case 2: if (pile) { depiluact(&pile,&ci,&cj,&k);k++;etat=1; }
              else etat=3;
      }
  }

*tprim=build_liste(taille_ensemble(f));

for (i=f,l=*tprim;l;l=l->next,i=i->next)
        {
          cl=l->element=((triplet_ptrs *)(i->element))->e3;
          ci=((triplet_ptrs *)(i->element))->e1;
          cj=((triplet_ptrs *)(i->element))->e2;
          cl->tp=lub(ci->tp,cj->tp);
          if (memeforme(ci,cj))
              {
               cl->forme.functor=ci->forme.functor ? strdup(ci->forme.functor):NULL;

               if (ci->forme.arguments)

                   {
                    cl->forme.arguments=NULL;ens=&(cl->forme.arguments);
                    for (m=ci->forme.arguments,j=cj->forme.arguments;m;m=m->next,j=j->next)
                    if ((cl2 = (classe *) in_f(f,m->element,j->element))!=NULL)
                       { append(ens,cl2); ens=&((*ens)->next); }
```

```
                    else { printf("Erreur majlub\n"); exit(1); }
                   }

                else cl->forme.arguments=NULL;
               }

            else { cl->forme.arguments=NULL; cl->forme.functor=NULL; }
          }

*eps=NULL;

ps1=ps_et(t1,eps1);
ps2=ps_et(t2,eps2);

for (i=f;i;i=i->next)
   for (j=f;j;j=j->next)
      if (undefined(((triplet_ptrs *)(i->element))->e3) && undefined(((triplet_ptrs *)(j->element))->e3))
        { ci=((triplet_ptrs *)(i->element))->e1;cj =((triplet_ptrs *)(i->element))->e2;
          cl=((triplet_ptrs *)(j->element))->e1;cl2=((triplet_ptrs *)(j->element))->e2;
          if (ps(ci,cl,ps1) || ps(cj,cl2,ps2)) addpsver(eps,((triplet_ptrs *) i->element)->e3,((triplet_ptrs *) j->element)
->e3); }

destroy_ensemble(f);
destroy_ensemble(ps1);
destroy_ensemble(ps2);
}

beta betalub(beta b1,beta b2)
{
if (b1.sv==NULL && b1.t==NULL && b1.eps==NULL)
   return copybeta(b2);
if (b2.sv==NULL && b2.t==NULL && b2.eps==NULL)
   return copybeta(b1);
 {
  beta b;
  maj(b1.sv,b2.sv,b1.t,b2.t,b1.eps,b2.eps,&(b.sv),&(b.t),&(b.eps));
  return b;
 }
}
```

```
********************************************************************************************** majlub.h ************
int addf(type_ensemble *f,classe *i,classe *j,classe **cl);
void *in_f(type_ensemble f,classe *i,classe *j);
int infrm(classe *i,classe *j);
void maj(type_ensemble sv1,type_ensemble sv2,type_ensemble t1,type_ensemble t2,type_ensemble eps1,type_ensemble
eps2,type_ensemble *svprim,type_ensemble *tprim,type_ensemble *eps);
beta betalub(beta b1,beta b2);
```

```
********************************************************************************** managest.c **********
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "messtruc.h"
#include "ensemble.h"
#include "managest.h"

void liberebeta(beta b)
{
type_ensemble i,j;
for (i=b.t;i;i=j)
   { j=i->next;
     kill_classe(i->element);
     free(i); }
destroy_ensemble(b.sv);
destroy_ensemble(b.eps);
}

beta copybeta(beta b)
{
type_ensemble i,j,k,l;
type_sv *nsv1, *nsv2;
classe *cl1, *cl2;
paire *p1, *p2;
beta bout;
bout.sv =build_liste(taille_ensemble(b.sv));
bout.t  =build_liste(taille_ensemble(b.t));
bout.eps=build_liste(taille_ensemble(b.eps));

for (i=bout.sv ;i;i=i->next) i->element=(type_sv *) safe_malloc (sizeof(type_sv));
for (i=bout.t  ;i;i=i->next) i->element=(classe *)  safe_malloc (sizeof(classe));
for (i=bout.eps;i;i=i->next) i->element=(paire *)   safe_malloc (sizeof(paire));

for (i=b.sv,j=bout.sv;i;i=i->next,j=j->next)
  {
    nsv1=(type_sv *) i->element;
    nsv2=(type_sv *) j->element;
    j->element=nsv2;
    nsv2->xi=nsv1->xi;
    nsv2->sv=(classe *) get_nieme(bout.t,getpos(b.t,nsv1->sv));
  }

for (i=b.t,j=bout.t;i;i=i->next,j=j->next)
  {
    cl1 = i->element;
    cl2 = j->element;
    cl2->tp = cl1->tp;
    cl2->forme.functor   = cl1->forme.functor ? strdup(cl1->forme.functor) : NULL;
    for (k = cl2->forme.arguments = build_liste(taille_ensemble(cl1->forme.arguments)), l = cl1->forme.arguments; k; k = k-
>next, l=l->next)
       k->element = get_nieme(bout.t,getpos(b.t,l->element));
  }

for (i=b.eps,j=bout.eps;i;i=i->next,j=j->next)
   {
    p1=(paire *) i->element;
```

```
      p2=(paire *) j->element;
      p2->e1=get_nieme(bout.t,getpos(b.t,p1->e1));
      p2->e2=get_nieme(bout.t,getpos(b.t,p1->e2));
    }

return bout;
}

void kill_classe(classe *z)
{ if (z->forme.functor) free (z->forme.functor);
  kill_ensemble(z->forme.arguments);
  free(z);
}

void kill_class_element(type_ensemble *t,classe *p)
{
kill_classe(p);
enleve_de_ens(t,p);
}
```

```
********************************************************************************** managest.h **********
void liberebeta(beta b);
beta copybeta(beta b);
void kill_classe(classe *z);
void kill_class_element(type_ensemble *t,classe *p);
```

```
********************************************************************************** messtruc.c **********
#include <stdio.h>
#include <stdlib.h>

char *safe_malloc(int i)
{
char *p;
if (i<=0) return NULL;
p=malloc(i);
if (p) return p;
puts("Erreur d'allocation de mémoire");
exit(1);
}
```

```
********************************************************************************** messtruc.h **********
typedef char type_tp;
typedef unsigned char utype_tp;

#define egale_sv(a,b) ((a).xi=(b).xi && (a).sv=(b).sv)

#define PROC 0x03
#define ABI3 0x04
#define NABI  0x06
#define NABI2 0x07
#define NVAR  0x08

typedef struct ens { void *element; struct ens *next; } type_element;

typedef type_element* type_ensemble;

typedef struct
{ char *functor;
  type_ensemble arguments; } type_forme;

typedef struct
{ type_tp tp;
  type_forme forme; } classe;

typedef struct
{ classe *ptr;
  type_tp tp; } paire_ptr_tp;

typedef struct
{ int xi;
  classe *sv; } type_sv;

typedef struct
{ void *e1,*e2; } paire;

typedef struct chvstack
{ struct chvstack *prev;
  int *valeur; } pilechvar;

typedef struct stacktpprim
{ struct stacktpprim *next;
  classe *k;
  type_ensemble l;
  type_tp *z;
  int indice; } piletpprim;

typedef struct stackuact
{ struct stackuact *next;
  classe *i,*j;
  int l; } uactpile;

typedef struct {
      char *valeur;
      int nbdesc, nbanc, level, *tab; } noeud;
                  /* ancetres ... descendants ...*/

typedef struct { char boolean;
                 char *string; } substitution;
```

```
struct couple { int i,j; } ;

struct ptrcouple { void *i,*j; } ;

typedef struct { type_ensemble sv;
                 type_ensemble t;
                 type_ensemble eps; } beta;

typedef struct { void *e1, *e2, *e3; } triplet_ptrs;

typedef struct { char *head;
                 int m;
                 char **body; } predicat;

typedef struct { int nbsuspended;
                 int *tab; } setsuspended;

typedef struct { beta bin;
                 char *p;
                 int arite;
                 beta bout; } tsat;

typedef struct
{ char *v;
  type_ensemble e;
  type_ensemble matching; } TPROC;

typedef struct
{ int v; } TNVAR;

typedef struct
{ char typ_egal;
  int v1,v2; } TNABI;

typedef struct
{ char typ_egal;
  int v1;
  char *v2;
  type_ensemble e; } TNABI2;

typedef struct
{ char typ_egal;
  char *v1;
  type_ensemble e1;
  char *v2;
  type_ensemble e2; } TABI3;

typedef struct
{ char type;
  union { TNABI  abi1;
          TNABI2 abi2;
          TPROC app;
          TABI3 fun;
          TNVAR v; } ne; } Nnoeud_ex;

typedef struct
   { char type;
```

```
    char *string;
    int entier;
    double reel;
    type_ensemble args; } noeud_expr;

typedef struct
{ char typ_egal;
  char *v1,*v2; } TABI;

typedef struct
{ char typ_egal;
  char *v1,*v2;
  type_ensemble e; } TABI2;

typedef struct
{ char *v; } TVAR;

typedef struct
{ char type;
  union { TABI  abi;
          TABI2 abi2;
          TPROC app;
          TABI3 fun;
          TVAR  var; } ne; } noeud_ex;

char *safe_malloc(int i);
```

```
********************************************************************************** norm.c **************
#include <math.h>
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "messtruc.h"

#include "ensemble.h"
#include "stdfunct.h"

#define ABI  0x01
#define ABI2 0x02
#define VAR  0x05

#define SIN     -1
#define COS     -2
#define TAN     -3
#define LN      -4
#define EXP     -5
#define MOINS_U -6
#define PLUS     1
#define MOINS    2
#define FOIS     3
#define DIVISE   4
#define PUIS     5
#define RESTE    6

#define INTNUMBER   0x00
#define FLOATNUMBER 0x01
#define PROC        0x03
#define VARIABLE    0x02

#include "norm.h"

type_ensemble normalisation(type_ensemble *clauses,char *quest,int arite,char *nfich)
{
char buf[1000];
type_ensemble t,E_clauses=NULL,ie,je,ke;
Nnoeud_ex *p,*p2;
FILE *fich=fopen(nfich,"rt");
while (readln(fich,buf)!=-1)
    { arithmnormalize(buf);
      t=arbore(buf);
      t=normalise(t);
      append(&E_clauses,t); }
for (t=E_clauses;t;t=t->next)
    for (ie=t->element;ie;ie=ie->next)
       { p=ie->element;
         if (p->type==PROC) p->ne.app.matching=NULL; }
for (t=E_clauses;t;t=t->next)
    for (ie=((type_ensemble) t->element)->next;ie;ie=ie->next)
       { p=ie->element;
         if (p->type==PROC && p->ne.app.matching==NULL)
            { for (je=E_clauses;je;je=je->next)
                 { p2=((type_ensemble)je->element)->element;
```

```
                if (p2->type==PROC && strcmp(p2->ne.app.v,p->ne.app.v)==0
                   && taille_ensemble(p2->ne.app.e)==taille_ensemble(p->ne.app.e))
                   append(&(p->ne.app.matching),je->element); }
             for (je=E_clauses;je;je=je->next)
                 for (ke=((type_ensemble)je->element)->next; ke; ke=ke->next)
                   { p2=ke->element;
                     if (p2->type==PROC && p2->ne.app.matching==NULL &&
                         strcmp(p2->ne.app.v,p->ne.app.v)==0 &&
                         taille_ensemble(p2->ne.app.e)==taille_ensemble(p->ne.app.e))
                          p2->ne.app.matching=p->ne.app.matching;
                     }
          }
       }
/*
for (t=E_clauses;t;t=t->next)
   for (ie=((type_ensemble)t->element)->next;ie;ie=ie->next)
      { p=ie->element;
        if (p->type==PROC)
           for (ke=p->ne.app.matching;ke;ke=ke->next)
             printf("%s\n",((Nnoeud_ex *)((type_ensemble)ke->element)->element)->ne.app.v);
      }
*/
*clauses=E_clauses;
E_clauses=NULL;
for (t=*clauses;t;t=t->next)
   {
    renumerote(t->element);
    p=((type_ensemble)t->element)->element;
    if (strcmp(p->ne.app.v,quest)==0 && taille_ensemble(p->ne.app.e)==arite)
        append(&(E_clauses),t->element);
   }
fclose(fich);
if (!E_clauses)
  { printf("Foncteur %s d'arite %d non trouve dans la source", quest, arite); exit(1); }
fich=fopen("normed.txt","wt");
for (t=*clauses;t;t=t->next,fputc('\n',fich))
   for (ie=t->element;ie;ie=ie->next)
      { p=ie->element;
        switch(p->type)
          { case NABI : { fprintf(fich,"x%d%cx%d ",p->ne.abi1.v1,p->ne.abi1.typ_egal,p->ne.abi1.v2);
                          break; }
            case NABI2: { fprintf(fich,"x%d%c%s(",p->ne.abi2.v1,p->ne.abi2.typ_egal,p->ne.abi2.v2);
                          for (je=p->ne.abi2.e;je;je=je->next)
                               { fprintf(fich,"x%d",((Nnoeud_ex *)je->element)->ne.v.v);
                                 if (je->next) fputc(',',fich); }
                          fputs(") ",fich);
                          break; }
            case PROC:  { fprintf(fich,"%s(",p->ne.app.v);
                          for (je=p->ne.app.e;je;je=je->next)
                               { fprintf(fich,"x%d",((Nnoeud_ex *)je->element)->ne.v.v);
                                 if (je->next) fputc(',',fich); }
                          fputs(") ",fich);
                          break; }
            default : { printf("type=%c\n",p->type);
                        puts("Erreur normalisation : norm.c fonction normalisation"); }
          }
      }
fclose(fich);
```

```
return E_clauses;
}

void renumerote(type_ensemble t)
{
int i;
Nnoeud_ex *p=t->element;
type_ensemble ie=p->ne.app.e;
for (i=1;ie;ie=ie->next,i++) if (((Nnoeud_ex *)ie->element)->ne.v.v!=i) swappe_vars_clause(((Nnoeud_ex *)ie->element)-
>ne.v.v,i,t);
}

void swappe_vars_clause(int i, int j,type_ensemble t)
{
Nnoeud_ex *p,*p2;
type_ensemble ie;
for (;t;t=t->next)
   { p=t->element;
     switch(p->type)
      { case NABI:  { swapints(i,j,&(p->ne.abi1.v1));
                      swapints(i,j,&(p->ne.abi1.v2));
                      break; }
        case NABI2: { swapints(i,j,&(p->ne.abi2.v1));
                      for (ie=p->ne.abi2.e;ie;ie=ie->next)
                         { p2=ie->element;
                           swapints(i,j,&(p2->ne.v.v)); }
                      break; }
        case PROC:  { for (ie=p->ne.app.e;ie;ie=ie->next)
                         { p2=ie->element;
                           swapints(i,j,&(p2->ne.v.v)); }
                      break; }
     }
   }
}

void swapints(int i,int j, int *p)
{ if (*p==i) *p=j;
  else if (*p==j) *p=i;
}

void affclause(noeud_ex *p)
{
  type_ensemble t;
  switch(p->type)
   { case ABI: { printf("%s%c%s",p->ne.abi.v1,p->ne.abi.typ_egal,p->ne.abi.v2);
                 break; }
     case ABI2:{ printf("%s%c%s(",p->ne.abi2.v1,p->ne.abi2.typ_egal,p->ne.abi2.v2);
                 for (t=p->ne.abi2.e;t;t=t->next)
                     { affclause(t->element);
                       if (t->next) putchar(','); }
                 putchar(')');
                 break; }
     case PROC:{  printf("%s(",p->ne.app.v);
                  for (t=p->ne.app.e;t;t=t->next)
                      { affclause(t->element);
                        if (t->next) putchar(','); }
                  putchar(')');
                  break; }
```

```
    case VAR:  { printf("%s",p->ne.var.v);
                 break; }
    case ABI3:{  printf("%s(",p->ne.fun.v1);
                 for (t=p->ne.fun.e1;t;t=t->next)
                     { affclause(t->element);
                       if (t->next) putchar(','); }
                 printf(")%c%s(",p->ne.fun.typ_egal,p->ne.fun.v2);
                 for (t=p->ne.fun.e2;t;t=t->next)
                     { affclause(t->element);
                       if (t->next) putchar(','); }
                 putchar(',');
                 break; }
    default:{ printf("type  : %d\n",(int)p->type); }
   }
}

type_ensemble arbore(char *buf)
{
type_ensemble te=NULL;
int z;
char *p=buf;
while (*p && *p==' ') p++;
if (*p)
  while (z=strcspn(p," "))
    {
     append(&te,express(p,z));
     p+=z;
     while (*p==' ') p++;
    }
return te;
}

char *present(char *c,char *pattern,int length,int offset)
{
int d;
for (d=offset;d<length;d++)
   if (strchr(pattern,(int) c[d])) return c+d;
return NULL;
}

noeud_ex *express(char *p,int length)
{
noeud_ex *res=(noeud_ex *) safe_malloc (sizeof(noeud_ex));
int c;
for (c=0;c<length;c++)
  switch(p[c])
  {
    case '(':
       {
         char *str=present(p,"=~<",length,c),*p2;
         int l;
         if (str)
           {
            int l;
            res->type=ABI3;
            res->ne.fun.typ_egal=*str;
            res->ne.fun.v1=monstcpy(p+c-1,p);
            res->ne.fun.e1=NULL;
```

```
            for (p2=p+c+1;l=getnextarg(p2);)
               { append(&(res->ne.fun.e1),express(p2,l));
                 p2+=l;
                 if (!*p2 || *p2==')') p2=NULL;
                 else p2++; }
            str=strchr(p=str+1,'(');
            res->ne.fun.v2=monstcpy(str-1,p);
            res->ne.fun.e2=NULL;
            for (p2=str+1;l=getnextarg(p2);)
               { append(&(res->ne.fun.e2),express(p2,l));
                 p2+=l;
                 if (!*p2 || *p2==')') p2=NULL;
                 else p2++; }
          }
        else
          {
            res->type=PROC;
            res->ne.app.v=monstcpy(p+c-1,p);
            res->ne.app.e=NULL;
            for (p2=p+c+1;l=getnextarg(p2);)
               { append(&(res->ne.app.e),express(p2,l));
                 p2+=l;
                 if (!*p2 || *p2==')') p2=NULL;
                 else p2++; }
          }
        return res;
      }
    case '=':
    case '~':
    case '<':
       {
        char *str=present(p,"(",length,c),*p2;
        int l;
        if (str)
          {
            res->type=ABI2;
            res->ne.abi2.typ_egal=p[c];
            res->ne.abi2.v1=monstcpy(p+c-1,p);
            res->ne.abi2.v2=monstcpy(str-1,p+c+1);
            res->ne.abi2.e=NULL;
            for (p2=str+1;l=getnextarg(p2);)
               { append(&(res->ne.abi2.e),express(p2,l));
                 p2+=l;
                 if (!*p2 || *p2==')') p2=NULL;
                 else p2++; }
          }
        else
          {
            res->type=ABI;
            res->ne.abi.typ_egal=p[c];
            res->ne.abi.v1=monstcpy(p+c-1,p);
            res->ne.abi.v2=monstcpy(p+length-1,p+c+1);
          }
        return res;
       }
  }
res->type=VAR;
res->ne.var.v=monstcpy(p+length-1,p);
```

```
return res;
}

int getnextarg(char *p)
{
int n,level;
if (p==NULL) return 0;
for (n=level=0;level!=0 || (p[n]!=')' && p[n]!=',' && p[n]);n++)
      if (p[n]=='(') level++;
      else if (p[n]==')') level--;
return n;
}

type_ensemble normalise(type_ensemble t)
{
type_ensemble vars=NULL,ie,nouv=NULL;
int z;
for (ie=t;ie;ie=ie->next)
   append(&nouv,ncopie(ie->element,&vars));
libere_noeuds_ex(t);
z=taille_ensemble(vars);
destroy_ensemble(vars);
norma(&nouv,z);
return nouv;
}

void libere_noeuds_ex(type_ensemble t)
{
type_ensemble ie;
for (ie=t;ie;ie=ie->next)
    libere_noeud_ex(ie->element);
kill_ensemble(t);
}

void libere_noeud_ex(noeud_ex *n)
{
switch(n->type)
   {
    case ABI:
        { free(n->ne.abi.v1);
          free(n->ne.abi.v2);
          break; }
    case ABI2:
        { type_ensemble je;
          free(n->ne.abi2.v1);
          free(n->ne.abi2.v2);
          for (je=n->ne.abi2.e;je;je=je->next)
             libere_noeud_ex(je->element);
          kill_ensemble(n->ne.abi2.e);
          break; }
    case ABI3:
        { type_ensemble je;
          free(n->ne.fun.v1);
          free(n->ne.fun.v2);
          for (je=n->ne.fun.e1;je;je=je->next)
              libere_noeud_ex(je->element);
          kill_ensemble(n->ne.fun.e1);
          for (je=n->ne.fun.e2;je;je=je->next)
```

```
            libere_noeud_ex(je->element);
          kill_ensemble(n->ne.fun.e2);
          break; }
    case PROC:
        { type_ensemble je;
          free(n->ne.app.v);
          for (je=n->ne.app.e;je;je=je->next)
              libere_noeud_ex(je->element);
          kill_ensemble(n->ne.app.e);
          break; }
    case VAR:
        { free(n->ne.var.v);
          break; }
    default: { printf("cacaprout %d\n",n->type);
               exit(1); }
   }
if (n) free(n);
}

Nnoeud_ex *ncopie(noeud_ex *z, type_ensemble *vars)
{
Nnoeud_ex *new=(Nnoeud_ex *) safe_malloc(sizeof(Nnoeud_ex));
switch(z->type)
   {
    case ABI:
         {
          new->type=NABI;
          new->ne.abi1.typ_egal=z->ne.abi.typ_egal;
          new->ne.abi1.v1=trvstring(z->ne.abi.v1,vars);
          new->ne.abi1.v2=trvstring(z->ne.abi.v2,vars);
          break;
         }
    case ABI2:
         {
          type_ensemble p;
          new->type=NABI2;
          new->ne.abi2.typ_egal=z->ne.abi2.typ_egal;
          new->ne.abi2.v1=trvstring(z->ne.abi2.v1,vars);
          new->ne.abi2.v2=strdup(z->ne.abi2.v2);
          new->ne.abi2.e=NULL;
          for (p=z->ne.abi2.e;p;p=p->next) append(&(new->ne.abi2.e),ncopie(p->element,vars));
          break;
         }
    case PROC:
         {
          type_ensemble p;
          new->type=PROC;
          new->ne.app.v=strdup(z->ne.app.v);
          new->ne.app.e=NULL;
          for (p=z->ne.app.e;p;p=p->next) append(&(new->ne.app.e),ncopie(p->element,vars));
          break;
         }
    case ABI3:
         {
          type_ensemble p;
          new->type=ABI3;
          new->ne.fun.typ_egal=z->ne.fun.typ_egal;
          new->ne.fun.v1=strdup(z->ne.fun.v1);
```

```
        new->ne.fun.v2=strdup(z->ne.fun.v2);
        new->ne.fun.e1=new->ne.fun.e2=NULL;
        for (p=z->ne.fun.e1;p;p=p->next) append(&(new->ne.fun.e1),ncopie(p->element,vars));
        for (p=z->ne.fun.e2;p;p=p->next) append(&(new->ne.fun.e2),ncopie(p->element,vars));
        break;
       }
   case VAR:
       {
        new->type=NVAR;
        new->ne.v.v=trvstring(z->ne.var.v,vars);
        break;
       }
   default: printf("type :%d\n",(int)z->type);
  }
return new;
}

int trvstring(char *p,type_ensemble *m)
{
int i=1;
type_ensemble e=*m,*pe=m;
while (e && strcmp(e->element,p)!=0)
      { i++;
        pe=&(e->next);
        e=e->next; }
if (e==NULL)
  {
   *pe=(type_element *)safe_malloc(sizeof(type_element));
   (*pe)->element=strdup(p);
   (*pe)->next=NULL;
  }
return i;
}

void compte_var(Nnoeud_ex *t,type_ensemble *e)
{
switch(t->type)
  {
   case NABI:
       {
        compte(t->ne.abi1.v1,e);
        compte(t->ne.abi1.v2,e);
        break;
       }
   case NABI2:
       {
        type_ensemble p;
        compte(t->ne.abi2.v1,e);
        for (p=t->ne.abi2.e;p;p=p->next) compte_var(p->element,e);
        break;
       }
   case PROC:
       {
        type_ensemble p;
        for (p=t->ne.app.e;p;p=p->next) compte_var(p->element,e);
        break;
       }
   case ABI3:
```

```
        {
         type_ensemble p;
         for (p=t->ne.fun.e1;p;p=p->next) compte_var(p->element,e);
         for (p=t->ne.fun.e2;p;p=p->next) compte_var(p->element,e);
         break;
        }
    case NVAR:
        {
         compte(t->ne.v.v,e);
         break;
        }
    default: puts("cacaprout norm.c");
   }
}

void compte(int i, type_ensemble *n)
{
type_ensemble e=*n,*pe=n;
while (e && ((struct couple *) e->element)->i!=i)
       ( pe=&(e->next);
         e=e->next; )
if (e==NULL)
   {
    *pe=(type_element *)safe_malloc(sizeof(type_element));
    (*pe)->element=(struct couple *) safe_malloc(sizeof(struct couple));
    ((struct couple *) (*pe)->element)->i=i;
    ((struct couple *) (*pe)->element)->j=1;
    (*pe)->next=NULL;
   }
else
   (((struct couple *) e->element)->j)++;
}

void norma(type_ensemble *nouv,int n)
{
type_ensemble substs,t,s,*u,x;
struct couple *c;
int i;
for (t=*nouv;t;t=t->next)
    { substs=NULL;
      normal(&substs,t->element,&n);
      if (substs)
          if (t==*nouv) /* HEAD==>arguments APRES */
            { s=t->next;
              t->next=substs;
              while(t->next) t=t->next;
              t->next=s; }
          else
            { s=t;
              t=*u=substs;
              while (t->next) t=t->next;
              t->next=s;
              t=t->next; }
      u=&(t->next);
    }
for (t=*nouv;t;t=t->next)
   { substs=NULL;
     compte_var(t->element,&substs);
```

```
      x=t;
      for (s=substs;s;s=s->next)
          { c=s->element;
            for (i=1;i<c->j;i++)
               { Nnoeud_ex *p=(Nnoeud_ex *) safe_malloc(sizeof(Nnoeud_ex));
                 type_element *l=(type_element *) safe_malloc(sizeof(type_element));
                 type_ensemble y;
                 p->type=NABI;
                 p->ne.abi1.typ_egal='=';
                 p->ne.abi1.v1=++n;
                 p->ne.abi1.v2=c->i;
                 l->element=p;
                 if (x==*nouv)
                     { y=t->next;
                       t->next=l;
                       l->next=y;
                       t=l;
                       change_n(x->element,c->i,n); }
                 else
                     { l->next=t;
                       *u=l;
                       u=&(l->next);
                       change_n(t->element,c->i,n); }
               }
          }
       u=&(t->next);
    }
}

int change_n(Nnoeud_ex *n,int i, int j)
{
switch(n->type)
   { case NABI2:
          { type_ensemble p;
            if (n->ne.abi2.v1==i) { n->ne.abi2.v1=j; return 1; }
            for (p=n->ne.abi2.e;p;p=p->next)
               if (change_n(p->element,i,j)==1) return 1;
            return 0;
          }
     case PROC:
          { type_ensemble p;
            for (p=n->ne.app.e;p;p=p->next)
               if (change_n(p->element,i,j)==1) return 1;
            return 0;
          }
     case NABI:
          { if (n->ne.abi1.v1==i) { n->ne.abi1.v1=j;return 1; }
            if (n->ne.abi1.v2==i) { n->ne.abi1.v2=j;return 1; }
            return 0;
          }
     case NVAR:
          { if (n->ne.v.v==i) { n->ne.v.v=j; return 1; }
            else return 0;
          }
  }
assert(0);
}
```

```
void normal(type_ensemble *substs,Nnoeud_ex *n,int *p)
{
  switch (n->type)
    {
     case NABI2:
          { type_ensemble ie;
            for (ie=n->ne.abi2.e;ie;ie=ie->next)
               { Nnoeud_ex *node=ie->element;
                 if (node->type==PROC)
                    { Nnoeud_ex *node2=(Nnoeud_ex *) safe_malloc(sizeof(Nnoeud_ex)),interm;
                      node2->type=NVAR;
                      interm.ne.abi2.v1=node2->ne.v.v=++(*p);
                      ie->element=node2;
                      interm.type=NABI2;
                      interm.ne.abi2.typ_egal=(n->ne.abi2.typ_egal=='=' ? '=':'~');
                      interm.ne.abi2.v2=node->ne.app.v;
                      interm.ne.abi2.e=node->ne.app.e;
                      (*node)=interm;
                      ajoute_dans_ens(substs,node);
                      normal(substs,node,p); }
                 }
             return;
          }
     case PROC:
          { type_ensemble ie;
            for (ie=n->ne.app.e;ie;ie=ie->next)
               { Nnoeud_ex *node=ie->element;
                 if (node->type==PROC)
                    { Nnoeud_ex *node2=(Nnoeud_ex *) safe_malloc(sizeof(Nnoeud_ex)),interm;
                      node2->type=NVAR;
                      interm.ne.abi2.v1=node2->ne.v.v=++(*p);
                      ie->element=node2;
                      interm.type=NABI2;
                      interm.ne.abi2.typ_egal='=';
                      interm.ne.abi2.v2=node->ne.app.v;
                      interm.ne.abi2.e=node->ne.app.e;
                      (*node)=interm;
                      append(substs,node);
                      normal(substs,node,p); }
                 }
             return;
          }
     case ABI3:
          { Nnoeud_ex *node =(Nnoeud_ex *) safe_malloc(sizeof(Nnoeud_ex)),
                      *node2=(Nnoeud_ex *) safe_malloc(sizeof(Nnoeud_ex)),
                      interm;
            int i,j;
            node->type=NABI2;
            node->ne.abi2.typ_egal=(n->ne.fun.typ_egal=='=' ? '=':'~');
            i=node->ne.abi2.v1=++(*p);
            node->ne.abi2.v2=n->ne.fun.v1;
            node->ne.abi2.e=n->ne.fun.e1;
            node2->type=NABI2;
            node2->ne.abi2.typ_egal=(n->ne.fun.typ_egal=='=' ? '=':'~');
            j=node2->ne.abi2.v1=++(*p);
            node2->ne.abi2.v2=n->ne.fun.v2;
            node2->ne.abi2.e=n->ne.fun.e2;
            ajoute_dans_ens(substs,node);
```

```
            ajoute_dans_ens(substs,node2);
            interm.type=NABI;
            interm.ne.abi1.typ_egal=n->ne.fun.typ_egal;
            interm.ne.abi1.v1=i;
            interm.ne.abi1.v2=j;
            *n=interm;
            normal(substs,node,p);
            normal(substs,node2,p);
            return; }
    }
}

double getreste(double a,double b)
{
double s=a/b;
return floor(s)==s ? 0 : a-floor(s)*b;
}

int evaluate(noeud_expr *node)
{
type_ensemble i;
if (node->type==PROC)
  {
   int sw,operateur=sw=0;
   if (strcmp(node->string,"tan")==0) operateur = TAN;
   if (strcmp(node->string,"sin")==0) operateur = SIN;
   if (strcmp(node->string,"cos")==0) operateur = COS;
   if (strcmp(node->string,"ln" )==0) operateur = LN;
   if (strcmp(node->string,"exp")==0) operateur = EXP;
   if (operateur==0)
   switch(*(node->string))
     { case '+': { operateur = PLUS;   break; }
       case '-': { operateur = taille_ensemble(node->args)==1 ? MOINS_U : MOINS;  break; }
       case '*': { operateur = FOIS;   break; }
       case '/': { operateur = DIVISE; break; }
       case '^': { operateur = PUIS;   break; }
       case '%': { operateur = RESTE;  break; }
     }
   if (operateur<0 && taille_ensemble(node->args)!=1) operateur=0;
   if (operateur>0 && taille_ensemble(node->args)!=2) operateur=0;
   for (i=node->args;i;i=i->next)
         if (evaluate(i->element)==-1) sw=-1;
   if (sw==0 && operateur!=0)
     {
      free(node->string);
      i=node->args;
      if (operateur > 0)
        {
         noeud_expr *arg1,*arg2;
         arg1=node->args->element;
         arg2=node->args->next->element;
         if (arg1->type != arg2->type)
           if (arg1->type==INTNUMBER)
              { arg1->type = FLOATNUMBER; arg1->reel=(double)arg1->entier; }
           else
              { arg2->type = FLOATNUMBER; arg2->reel=(double)arg2->entier; }
         node->type=arg1->type;
         switch(operateur)
```

```
        {
         case PLUS:  { if (arg1->type==INTNUMBER) node->entier= arg1->entier + arg2->entier;
                       else node->reel=arg1->reel + arg2->reel; break; }
         case MOINS: { if (arg1->type==INTNUMBER) node->entier= arg1->entier - arg2->entier;
                       else node->reel=arg1->reel - arg2->reel; break; }
         case FOIS:  { if (arg1->type==INTNUMBER) node->entier= arg1->entier * arg2->entier;
                       else node->reel=arg1->reel * arg2->reel; break; }
         case DIVISE:{ if (arg1->type==INTNUMBER) node->reel= (double)arg1->entier / arg2->entier;
                       else node->reel=arg1->reel / arg2->reel;
                       node->type=FLOATNUMBER; break; }
         case PUIS:  { if (arg1->type==INTNUMBER) (node->reel= pow((double)arg1->entier,(double)arg2->entier);
                                                  node->type=FLOATNUMBER; }
                       else node->reel=pow(arg1->reel,arg2->reel); break; }
         case RESTE: { if (arg1->type==INTNUMBER) node->entier= arg1->entier % arg2->entier;
                       else node->reel=getreste(arg1->reel,arg2->reel);
                       break; }
        }
       if (node->type==FLOATNUMBER && fabs(node->reel-(double)(floor(node->reel)))<0.000001)
            { node->type=INTNUMBER;
              node->entier=floor(node->reel+0.5); }
      }
     else
       {
        noeud_expr *arg1=node->args->element;
        if (arg1->type==INTNUMBER && operateur==MOINS_U) arg1->entier=-arg1->entier;
        else
          {
           float a1= arg1->type==INTNUMBER ? (double) arg1->entier : arg1->reel;
           switch(operateur)
             {
              case SIN: { a1=sin(a1); break; }
              case COS: { a1=cos(a1); break; }
              case TAN: { a1=tan(a1); break; }
              case LN:  { a1=log(a1); break; }
              case EXP: { a1=exp(a1); break; }
              case MOINS_U: { a1=-a1; }
             }
           node->type=FLOATNUMBER;
           node->reel=a1;
          }
       }
     destroy_ensemble(i);
     return 0;
    }
   else return -1;
  }
else
   return node->type==VARIABLE ? -1 : 0;
}

int writetree(noeud_expr *node,char *buf)
{
int j;
if (node->type==PROC)
  { type_ensemble i;
    sprintf(buf,"%s(",node->string);
    j=strlen(buf);
    free(node->string);
```

```
    for (i=node->args;i;i=i->next)
      { j+=writetree (i->element,buf+j);
        if (i->next) buf[j++]=',' ; }
    buf[j++]=')';
    destroy_ensemble(node->args); }
else
  if (node->type==FLOATNUMBER) { sprintf(buf,"%f()",node->reel); j=strlen(buf); }
  else
    if (node->type==INTNUMBER) { sprintf(buf,"%d()",node->entier); j=strlen(buf); }
    else { strcpy(buf,node->string); j=strlen(node->string); free(node->string); }
free(node);
return j;
}

void afftree(noeud_expr *z)
{
type_ensemble i;
if (z->type==PROC) { printf("%s(",z->string);
                     for (i=z->args;i;i=i->next)
                       { afftree(i->element);if (i->next) putchar(','); }
                     putchar(')'); }
else if (z->type==INTNUMBER)
       printf("%d",z->entier);
     else
       if (z->type==FLOATNUMBER)
         printf("%f",z->reel);
       else
         if (z->type==VARIABLE)
           printf("%s",z->string);
}

char *evaluation_pref(noeud_expr *z)
{
char *buf=(char *)safe_malloc(1000);
evaluate(z);
writetree(z,buf);
return buf ? realloc(buf,strlen(buf)+1) : NULL;
}

char *lisfunctor(char *p)
{ return monstcpy(strchr(p,'(')-1,p); }

char *getinop(char *p,int length)
{
char *r=p,*q=NULL;
int l=0;
for (;length>0;p++,length--)
  switch(*p)
   {
   case '(': { l=1;p++;length--;
               while(l!=0)
                 { length--;
                   if (*p=='(') l++;
                   if (*p==')') l--;
                   p++; }
               p--;length++;
               break; }
   case '+': return p;
```

```
   case '-': if (p && p>r && *(p-1)!='/' && *(p-1)!='-' && *(p-1)!='*' && *(p-1)!='^' && *(p-1)!='%') return p;
             else break;
   case '*': { if (!q || *q=='/' || *q!='%' || *q=='^') q=p; break; }
   case '/': { if (!q || *q!='%' || *q=='^') q=p; break; }
   case '%': { if (!q || *q=='^') q=p; break; }
   case '^': { if (!q) q=p;break; }
   }
return q;
}

char *strnchr(char *p, int l,char c)
{
int i;
for (i=0;i<l;i++) if (p[i]==c) return p+i;
return NULL;
}

int getarg(char *p)
{
 int j,l=j=0;
 while((*p!=',' && *p!=')') || l != 0)
    { j++;
      if (*p=='(') l++;
      if (*p==')') l--;
      p++; }
 return j;
}

noeud_expr *nrm(char *p,int length)
{
noeud_expr *res=(noeud_expr *)safe_malloc(sizeof(noeud_expr));
char *op;
p=rem_par_inut(p,&length);
if (op=getinop(p,length))
  {
    res->type=PROC;
    res->string=monstcpy(op,op);
    res->args=NULL;
    append(&(res->args),nrm(p,op-p));
    append(&(res->args),nrm(op+1,p+length-op-1));
  }
else
   if (islower(*p))
     { /* functor */
      int l=0;
      res->string=lisfunctor(p);
      res->type=PROC;
      res->args=NULL;
      p=strchr(p,'('+1);
      while (*p!=')')
         {
          l=getarg(p);
          append(&(res->args),nrm(p,l));
          p+=l;
          if (*p==',') p++;
         }
     }
   else
```

```
    /* chiffre ou variable */
    if (*p=='-')
       {
        res->type=PROC;
        res->string=strdup("-");
        res->args=NULL;
        append(&(res->args),nrm(p+1,length-1));
       }
    else
       if (isupper(*p))
          { res->type=VARIABLE;
            res->string=monstcpy(p+length-1,p); }
       else
          if (strnchr(p,length,'.'))
             { res->type=FLOATNUMBER;
               res->reel=atof(p); }
          else
             { res->type=INTNUMBER;
               res->entier=atoi(p); }
return res;
}


noeud_expr *normexpr(char *expr)
{
 return nrm(expr,strlen(expr));
}

char *rem_par_inut(char *expr,int *length)
{
int i=0,minlevel,level,count,n=level=count=0;
char *p;
while (i<=*length && expr[i]=='(') i++;
if (i==0 || expr[*length-1]!=')') return expr;
minlevel=i;
i=*length-1;
while (expr[i--]==')') count++;
minlevel=min(minlevel,count);
level=0;
for (p=expr+minlevel;p<expr+*length-minlevel;p++)
   if (*p=='(') level++;
   else if (*p==')' && --level<n) n=level;
minlevel+=n;
if (minlevel)
  {
  *length=*length-2*minlevel;
  return expr+minlevel;
  }
else return expr;
}

char *normalise_expression(char *c)
{
noeud_expr *p=normexpr(c);
return evaluation_pref(p);
}

void arithmnormalize(char *c)
```

```
{
int i,j;
char *z,*z2,p[2000],*ptr=c,*ptrdest=p;

do
  {
   while (ptr[i=strcspn(ptr,"+-*/% ")]=' ' || ptr[i]=NULL)
            { if (i) { strncpy(ptrdest,ptr,i); ptrdest+=i; ptr+=i; }
              *ptrdest++=*ptr;
              if (*ptr++==NULL) { strcpy(c,p); return; } }

   j=strcspn(ptr,"=~<> ");

   if (ptr[j]==NULL || ptr[j]=' ')
      { z=normalise_expression(z2=monstcpy(ptr+j-1,ptr));
        free(z2);
        strcpy(ptrdest,z); ptrdest+=strlen(z);*ptrdest++=ptr[j]; free(z);
        if (ptr[j]) ptr+=j+1;
        else { strcpy(c,p); return; } }
   else
      if (i>j) /* f() = ... + ... */
         { strncpy(ptrdest,ptr,j+1);
           ptrdest+=j+1;ptr+=j+1; }
      else
         { z=normalise_expression(z2=monstcpy(ptr+j-1,ptr));
           free(z2);
           strcpy(ptrdest,z); ptrdest+=strlen(z);*ptrdest++=ptr[j]; free(z);
           ptr+=j+1; }
 } while(1);
}
```

```
********************************************************************************************** norm.h **************
type_ensemble normalisation(type_ensemble *clauses,char *quest,int arite,char *n);
void renumerote(type_ensemble t);
void swappe_vars_clause(int i, int j,type_ensemble t);
void swapints(int i,int j, int *p);
void affclause(noeud_ex *p);
type_ensemble arbore(char *buf);
char *present(char *c,char *pattern,int length,int offset);
noeud_ex *express(char *p,int length);
int getnextarg(char *p);
type_ensemble normalise(type_ensemble t);
void libere_noeuds_ex(type_ensemble t);
void libere_noeud_ex(noeud_ex *n);
Nnoeud_ex *ncopie(noeud_ex *z, type_ensemble *vars);
int trvstring(char *p,type_ensemble *m);
void compte_var(Nnoeud_ex *t,type_ensemble *e);
void compte(int i, type_ensemble *m);
void norma(type_ensemble *nouv,int n);
int change_n(Nnoeud_ex *n,int i, int j);
void normal(type_ensemble *substs,Nnoeud_ex *n,int *p);
int trvnbargs(char *z);
void skiparg(char **p);noeud_expr *buildtree(char *expr);
double getreste(double a,double b);
int evaluate(noeud_expr *node);
void afftree(noeud_expr *z);
int writetree(noeud_expr *node,char *buf);
char *evaluation_pref(noeud_expr *expr);
char *strnchr(char *p, int l, char c);
char *lisfunctor(char *p);
char *getinop(char *p,int length);
noeud_expr *nrm(char *p,int length);
noeud_expr *normexpr(char *expr);
int getarg(char *p);
char *rem_par_inut(char *expr,int *length);
char *normalise_expression(char *c);
void arithmnormalize(char *c);
```

```
********************************************************************************** o_access.c **********
#include <stdio.h>
#include <stdlib.h>

#include "messtruc.h"
#include "stdfunct.h"
#include "ensemble.h"

#include "o_access.h"

int dansparties(classe *i,classe *k)
{
/* i accede a k ? */
/* attention : bouclage possible : i->j->i */
if (i==k) return 1;
else
  { type_ensemble e;
    for (e=k->forme.arguments;e;e=e->next)
       if (dansparties(i,e->element)) return 1;
  }
return 0;
}

int accesvar(classe *t)
{
type_ensemble e=t->forme.arguments;

if (t->forme.arguments==NULL && t->forme.functor==NULL) return 1;

while (e)
  { if (accesvar(e->element)) return 1;
    e=e->next; }

return 0;
}

void buildparties(type_ensemble *e,classe *k)
{ if (!est_dans_ens(*e,k))
     { type_ensemble j;
       ajoute_dans_ens(e,k);
       for (j=k->forme.arguments;j;j=j->next) buildparties(e,j->element); }
}
```

```
********************************************************************************************** o_access.h **********
int dansparties(classe *i,classe *k);
int accesvar(classe *t);
void buildparties(type_ensemble *e,classe *k);
```

```
********************************************************************************** o_couple.c **********
#include <stdlib.h>
#include <stdio.h>

#include "messtruc.h"

#include "o_couple.h"

void *trvcori(void *i,struct ptrcouple *p,int n)
{
int k;
for (k=0;k<n;k++)
  if (p[k].j==i) return p[k].i;
return NULL;
}

void *trvcorj(void *i,struct ptrcouple *p,int n)
{
int k;
for (k=0;k<n;k++)
  if (p[k].i==i) return p[k].j;
return NULL;
}

int corresp(void *i,void *j,struct ptrcouple *p,int k)
{
int l;
for (l=0;l<k;l++)
   if (p[l].i==i && p[l].j==j) return 2;
   else if (p[l].i==i || p[l].j==j) return 1;
return 0;
}

int est_dans_ens_couples_ptrs(type_ensemble e,void *i, void *j)
{
type_ensemble ie;
struct ptrcouple *p;
for (ie=e;ie;ie=ie->next)
   { p=ie->element;
     if (p->i==i && p->j==j) return 1; }
return 0;
}

struct ptrcouple *buildcouple(void *i, void *j)
{
struct ptrcouple *p=(struct ptrcouple *) safe_malloc(sizeof(struct ptrcouple));
p->i=i;p->j=j;
return p;
}

struct ptrcouple *est_dans_liste_couple_(type_ensemble ti,void *i,int k)
{
type_ensemble ie;
struct ptrcouple *p;
for (ie=ti;ie;ie=ie->next)
  { p=ie->element;
    if (k=1 && p->i==i) return p;
    if (k=2 && p->j=i) return p; }
```

```
return NULL;
}
```

```
********************************************************************************************** o_couple.h **********
void *trvcori(void *i,struct ptrcouple *p,int n);
void *trvcorj(void *i,struct ptrcouple *p,int n);
int corresp(void *i,void *j,struct ptrcouple *p,int k);
int est_dans_ens_couples_ptrs(type_ensemble e,void *i, void *j);
struct ptrcouple *buildcouple(void *i, void *j);

#define est_dans_liste_couple_1(a,b) est_dans_liste_couple_ (a,b,1)
#define est_dans_liste_couple_2(a,b) est_dans_liste_couple_ (a,b,2)

struct ptrcouple *est_dans_liste_couple_(type_ensemble ti,void *i,int k);
```

```
********************************************************************************** o_operat.c **********
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#include "messtruc.h"

#include "defines.h"

#include "ensemble.h"

#include "sortie.h"
#include "o_types.h"
#include "o_ps.h"
#include "o_sv.h"
#include "o_access.h"
#include "stdfunct.h"
#include "o_couple.h"
#include "managest.h"

#include "o_operat.h"

int memeforme(classe *c1, classe *c2)
{
if (!c1->forme.functor && !c2->forme.functor)
   return taille_ensemble(c1->forme.arguments)==taille_ensemble(c2->forme.arguments);
else
   if ((c1->forme.functor && !c2->forme.functor) || (!c1->forme.functor && c2->forme.functor))
      return 0;
   else
      return (!strcmp(c1->forme.functor,c2->forme.functor) && taille_ensemble(c1->forme.arguments)==taille_ensemble(c2-
>forme.arguments));
}

void remover(classe *z,classe *z2,type_ensemble esv,type_ensemble *t,type_ensemble *eps)
{
type_sv *ptr;
type_ensemble ptrc1,ptrc12;
paire *p;
classe *c;
/* z>z2; enlever z;*/
while (ptr=est_dans_sv_droite(esv,z)) ptr->sv=z2;
for (ptrc1=*t; ptrc1; ptrc1=ptrc1->next)
   for (ptrc12=((classe *)ptrc1->element)->forme.arguments; ptrc12; ptrc12=ptrc12->next)
      if (ptrc12->element==z) ptrc12->element=z2;
ptrc1=*eps;
while (p=getfirstps(*eps,z,&c))
    if (ps(z2,(c==z)?z2:c,*eps))
       enleve_de_ens(eps,p);
    else
      { if (p->e1==z) p->e1=z2;
        if (p->e2==z) p->e2=z2; }
enleve_de_ens(t,z);
kill_classe(z);
}

void restr1(int *v,int n,type_ensemble *sv,type_ensemble *t,type_ensemble *eps,int flag)
```

```
/* flag = 1 si changement de variable */
{
int i,j;
type_sv *getsv;
type_ensemble tpr,zut,nsv,parties=nsv=NULL;
for (i=0;i<n;i++)
  { if (!(getsv=est_dans_sv_gauche(*sv,v[i]))) { printf("Erreur : variable non trouvee, x%d.\n",v[i]);
                                                 afftabs2(0,125,*t,*sv,*eps);
                                                 if (parties) kill_ensemble(parties);
                                                 return; }
    else
      { for (j=0;j<i;j++)
          if (v[i]==v[j])
            { printf("Erreur : variable presente deux fois dans la restriction, x%d\n",v[i]);
              if (parties) kill_ensemble(parties); return; }
        buildparties(&parties,getsv->sv);
      }
  }

for (i=0;i<n;i++)
   addsv(&nsv,v[i],est_dans_sv_gauche(*sv,v[i])->sv);
destroy_ensemble(*sv);
*sv=nsv;

tpr=*t;
while (tpr)
 {
  zut=tpr->next;
  if (!est_dans_ens(parties,tpr->element))
     { kill_class_element(t,tpr->element);
       kill_ps_elements(eps,tpr->element); }
  tpr=zut;
 }

if (flag) chvar(v,n,*sv);

kill_ensemble(parties);
}

void extension(int n,type_ensemble *sv,type_ensemble *t,type_ensemble *eps)
{
type_ensemble *tmp,ouaccrocher;
classe *triple;
type_sv *new_sv;
int k;

tmp=t;

while (*tmp && est_dans_sv_droite(*sv,(*tmp)->element))
       tmp=&((*tmp)->next);

ouaccrocher=(*tmp);

for (k=1;k<=n;k++)
    if (!est_dans_sv_gauche(*sv,k))
       { paire *zut=(paire *)safe_malloc(sizeof(paire));
         triple=(classe *) safe_malloc (sizeof(classe));
         triple->tp=VAR;triple->forme.functor=NULL;triple->forme.arguments=NULL;
```

```
        zut->e1=zut->e2=triple;
        ajoute_dans_ens(eps,zut);
        new_sv=(type_sv *) safe_malloc (sizeof(type_sv));
        new_sv->xi=k; new_sv->sv=triple;
        append(sv,new_sv);
        *tmp=(type_element *)safe_malloc(sizeof(type_element));
        (*tmp)->element=triple;
        tmp=&((*tmp)->next); }
(*tmp)=ouaccrocher;
}

void empiltppr(piletpprim **ptrpile,classe *k,type_ensemble l,type_tp *z,int indice)
{
piletpprim *p=(piletpprim *)safe_malloc(sizeof(piletpprim));
p->next=*ptrpile;
*ptrpile=p;
(*ptrpile)->k=k;
(*ptrpile)->l=l;
(*ptrpile)->z=z;
(*ptrpile)->indice=indice;
}

void depiltppr(piletpprim **ptrpile,classe **k,type_ensemble *l,type_tp **z,int *indice)
{
piletpprim *p=*ptrpile;
*ptrpile=(*ptrpile)->next;
*k=p->k;
*l=p->l;
*z=p->z;
*indice=p->indice;
free(p);
}

void tpprim(classe *i,classe *j,classe *k,paire_ptr_tp *tprime,type_ensemble eps_et)
{ int etat=0,tmp;
  type_tp *z,res,type;
  piletpprim *ptrpile=NULL:
  type_ensemble l;
  int indice;
if (get_tp_from_ptr_tp(tprime,k)==-1)
 while (etat < 3)
    switch(etat)
      { case 0:{ /* begin */
                 etat=2;

                 if (!ps(i,k,eps_et) && !ps(j,k,eps_et)) res = k->tp;

                 else if (i==k || j==k) res=uat(i->tp,j->tp);

                     else if (undefined(k))

                            if ((type = get_tp_from_ptr_tp(tprime,i)) != -1) res=iat2(k->tp,type);

                            else res=iat(k->tp);

                          else { etat=1; l = k->forme.arguments;
                                 tmp = taille_ensemble(l);
                                 indice=0;
```

```
                        z = (type_tp *)safe_malloc(tmp*sizeof(type_tp)); }

              if (etat != 1) break; }
       case 1:{ /* empiler */
              if (l)
                if ((type = get_tp_from_ptr_tp(tprime,l->element)) != -1)
                  { z[indice++]=type;l=l->next;etat=1;break; }
                else
                  { empiltppr(&ptrpile,k,l,z,indice);
                    k=l->element;etat=0;break; }
              else { res=mat(k->tp,k->forme.functor,z,indice);
                     if (z) free(z);
                     etat=2; }
            }
       case 2:{ /* cloture */
              set_tp_of_ptr_tp(tprime,k,res);
              if (ptrpile) { depiltppr(&ptrpile,&k,&l,&z,&indice);
                             z[indice++]=res; l=l->next;etat=1; }
              else etat=3; }
     }
}

void uact1(classe *i,classe *j,type_ensemble sv,type_ensemble *t,type_ensemble *eps)
{
if (i!=j)
  {
  int k=0,te=taille_ensemble(*t);
  type_ensemble ps1=NULL,cl;
  paire_ptr_tp *tprime=(paire_ptr_tp *) safe_malloc (taille_ensemble(*t)*sizeof(paire_ptr_tp));
  cl=*t;
  while (cl)
    { tprime[k].ptr  = cl->element;
      tprime[k++].tp = -1;
      cl = cl->next; }
  ps1=ps_et(*t,*eps);
  tpprim(i,j,i,tprime,ps1);
  if (get_tp_from_ptr_tp(tprime,j)==-1) tpprim(i,j,j,tprime,ps1);
  for (cl=*t;cl;cl=cl->next)
     if (get_tp_from_ptr_tp(tprime,(classe *)cl->element)==-1) tpprim(i,j,(classe *)cl->element,tprime,ps1);
  for (k=0;k<te;k++) tprime[k].ptr->tp=tprime[k].tp;
  if (tprime) free(tprime);
  destroy_ensemble(ps1);
  if (ng(i->tp))
    { ps1=buildmistotrans(*eps,j,i);
      enleve_selection1_de_ps(eps,ps_ng);
      *eps=union_disj(*eps,ps1); }
  else
      enleve_selection1_de_ps(eps,ps_ng);
  remover((classe *)ptrmax(i,j),(classe *)ptrmin(i,j),sv,t,eps);
  }
}

int egalbottom(type_tp *ti,int n)
{
int i;
for (i=0;i<n;i++) if (ti[i]==BOTTOM) return 1;
return 0;
}
```

```
int specat(classe *i,classe *j,type_ensemble *t,type_ensemble *eps)
{
type_ensemble ind_ens,e,e2,eps_tr;
classe *k;
type_tp *ti,tmp;
paire_ptr_tp *tprime;
int m,n=taille_ensemble(j->forme.arguments),l=estpluspetit(VAR,i->tp),te=taille_ensemble(*t),bot;
if (((bot=egalbottom(ti=extr(i->tp,j->forme.functor,n),n)) && !l) || dansparties(i,j))
   { printf("t[%x].tp=%s\n",i,(i->tp==-1)?"indefini":typestring(i->tp));
     printf("l=%d\n",l);
     for (m=0;m<n;m++)
          printf("ti[%d]=%s\n",m,((ti[m]==-1)?"indefini":typestring(ti[m])));
     printf("getparties %d\n",dansparties(i,j)); return -1; }

tprime=(paire_ptr_tp *) safe_malloc (te * sizeof(paire_ptr_tp));

for (m=0,ind_ens=*t;ind_ens;ind_ens=ind_ens->next)
    { tprime[m].ptr  = ind_ens->element;
      tprime[m++].tp = -1; }

eps_tr=ps_et(*t,*eps);

for (ind_ens=*t;ind_ens;ind_ens=ind_ens->next)
 if (get_tp_from_ptr_tp(tprime,k=ind_ens->element)==-1)
  { piletpprim *mapile= NULL; int ind,etat=0; type_tp *z, res; type_ensemble narg;
   while (etat < 3)
    {
    switch(etat)
      { case 0:{ /* begin */
                 etat=1;
                 if (!ps(i,k,eps_tr) || dansparties(k,j)) res = k->tp;
                 else
                     if (i==k)
                       { type_tp type=mat(i->tp,j->forme.functor,ti,n);
                         if (l)
                             { type_tp *ti2=(type_tp *) safe_malloc(n*sizeof(type_tp));
                               for (m=0;m<n;m++) ti2[m]=VAR;
                               res=lub(type,cons(j->forme.functor,ti2,n));
                               if (ti2) free(ti2); }
                         else res=type; }
                     else
                       if (!undefined(k))
                         { narg=k->forme.arguments;
                           m=taille_ensemble(narg);
                           z=(type_tp *) safe_malloc (m*sizeof(type_tp));
                           ind=0;
                           etat=2; break; }
                       else if (l)
                            { type_tp *ti2=(type_tp *) safe_malloc(n*sizeof(type_tp));
                              for (m=0;m<n;m++) ti2[m]=VAR;
                              res=iat2(k->tp,cons(j->forme.functor,ti2,n));
                              if (ti2) free(ti2); }
                          else res=k->tp;
            }

        case 1:{ /* cloture et depilage */
                 set_tp_of_ptr_tp(tprime,k,res);
```

```
                if (mapile) { depiltppr(&mapile,&k,&narg,&z,&ind);
                              z[ind++]=res; narg=narg->next;
                              etat=2; }
                else { etat=3;break; } }

        case 2:/* testboucle et empilage*/
               if (narg)
                   if ((tmp=get_tp_from_ptr_tp(tprime,k))!=-1) { z[ind++]=tmp; narg=narg->next; }
                   else { empiltppr(&mapile,k,narg,z,ind);
                          k=narg->element; etat=0; }
               else { res=mat(k->tp,k->forme.functor,z,taille_ensemble(k->forme.arguments));
                      if (z) free(z); etat=1; }
      }
    }
  }

destroy_ensemble(eps_tr);
for (m=0;m<te;m++) tprime[m].ptr->tp=tprime[m].tp;

free(tprime);

i->forme.functor=strdup(j->forme.functor);
i->forme.arguments=NULL;

e2=NULL;

for (m=0;m<n;m++)
     { e=append(t,safe_malloc(sizeof(classe)));
       if (m==0) e2=e;
       k=e->element;
       k->tp = l ? lub(ti[m],VAR) : ti[m];
       append(&(i->forme.arguments),(void *)k);
       k->forme.functor=NULL;
       k->forme.arguments=NULL; }

for (ind_ens=*eps;ind_ens;ind_ens=ind_ens->next)
   { paire *p=ind_ens->element;
     classe *k;
     /* calcul de Ps2 */
     if (p->el!=p->e2)
        { k=NULL;
          if (p->el==i) k=p->e2;
          else if (p->e2==i) k=p->el;
          if (k && ng(k->tp))
             /* Ps(i,k) && ng(tp(k)) */
             for (e=e2;e;e=e->next)
                 if (ng(((classe *) e->element)->tp)) ajoute_dans_ens(eps,buildcouple(k,e->element));
        }
   }
enleve_i_de_ps(eps,i);
if (l || !bot)
   for (e=e2;e;e=e->next) ajoute_dans_ens(eps,buildcouple(e->element,e->element));
if (!bot && e2)
   for (e=e2->next;e;e=e->next)
      for (ind_ens=e2;ind_ens!=e;ind_ens=ind_ens->next)
         ajoute_dans_ens(eps,buildcouple(e->element,ind_ens->element));
                   /* si var <= ti ou !bot, ng est trivial */
return 0;
```

```
}

int undefined(classe *i)
{
/* i est une forme, undefined renvoie vrai si cette forme est indefinie : pas de foncteur, 0 args */
return (i->forme.functor==NULL) && (i->forme.arguments==NULL);
}

void majcples(type_ensemble e,classe *k,classe *l)
{ type_ensemble ie;
  void *maxij,*minij;
  struct ptrcouple *p;
  maxij=ptrmax(k,l); minij=ptrmin(k,l);
  for (ie=e;ie;ie=ie->next)
      if (p=ie->element)
        { if (p->i==maxij) p->i= minij;
          if (p->j==maxij) p->j= minij; }
}

void libere(uactpile *pile)
{
uactpile *p;
while (pile)
  {
   p=pile;
   pile=pile->next;
   free(p);
  }
}

int uactgen(classe *i,classe *j,type_ensemble sv,type_ensemble *t,type_ensemble *eps,type_ensemble *liste)
{
type_ensemble *e,ei,ii,ji;
type_element *elt;
if (i!=j)
   if (undefined(i) && undefined(j))
      { uact1(i,j,sv,t,eps);
        free((*liste)->element);
        enleve_de_ens(liste,(*liste)->element);
        majcples(*liste,i,j);
      }
   else
      {
       if (undefined(i))
         { if (specat(i,j,t,eps)==-1) { printf("Unification ratee  specat\n");
                                        return -1; }
         }
       else
         if (undefined(j))
           { if (specat(j,i,t,eps)==-1) { printf("Unification ratee  specat\n");
                                          return -1; }
           }
         else
           if (!memeforme(i,j))
             { printf("Unification ratee  forme\n"); return -1; }

       for (e=&ei,ii=i->forme.arguments,ji=j->forme.arguments;ii;ii=ii->next, ji=ji->next)
          {
```

```
            elt=(type_element *) safe_malloc(sizeof(type_element));
            elt->element=buildcouple(ii->element, ji->element);
            elt->next=NULL;
            *e=elt;
            e=&(elt->next);
           }
       elt=(type_element *) safe_malloc(sizeof(type_element));
       elt->element=NULL;
       *e=elt;
       elt->next=*liste;
       *liste=ei;
      }
else
   enleve_de_ens(liste,(*liste)->element);

return 0;
}

int ualct(type_ensemble *i,type_ensemble sv,type_ensemble *t,type_ensemble *eps)
{
struct ptrcouple *p;
while (*i)
  { p=(*i)->element;
    if (p)
      { if (uactgen(p->i,p->j,sv,t,eps,i)==-1)
        { destroy_ensemble(*i); *i=NULL; return -1; }
      }
    else
      { enleve_de_ens(i,(*i)->element);
        p=(*i)->element;
        remover((classe *)ptrmax(p->i,p->j),(classe *)ptrmin(p->i,p->j),sv,t,eps);
        majcples(*i,p->i,p->j);
        free(p);
        enleve_de_ens(i,p);
      }
  }
return 0;
}

int uact(classe *i,classe *j,type_ensemble sv,type_ensemble *t,type_ensemble *eps)
{
type_ensemble e=NULL;
ajoute_dans_ens(&e,buildcouple(i,j));
return ualct(&e,sv,t,eps);
}

void empiluact(uactpile **pile,classe *i,classe *j,int l)
{
uactpile *p=(uactpile *)safe_malloc(sizeof(uactpile));
p->next=*pile;
*pile=p;
(*pile)->i=i;
(*pile)->j=j;
(*pile)->l=l;
}

void depiluact(uactpile **pile,classe **i,classe **j,int *l)
{
```

```
uactpile *p=*pile;
*pile=(*pile)->next;
*i=p->i;
*j=p->j;
*l=p->l;
free(p);
}
```

```
********************************************************************************** o_operat.h **********
void empiluact(uactpile **pile,classe *i,classe *j,int l);
void depiluact(uactpile **pile,classe **i,classe **j,int *l);
int memeforme(classe *c1, classe *c2);
void remover(classe *z,classe *z2,type_ensemble esv,type_ensemble *t,type_ensemble *eps);
void restr1(int *v,int n,type_ensemble *sv,type_ensemble *t,type_ensemble *eps,int flag);
void extension(int n,type_ensemble *sv,type_ensemble *t,type_ensemble *eps);
void empiltppr(piletpprim **ptrpile,classe *k,type_ensemble l,type_tp *z,int indice);
void depiltppr(piletpprim **ptrpile,classe **k,type_ensemble *l,type_tp **z,int *indice);
void tpprim(classe *i,classe *j,classe *k,paire_ptr_tp *tprime,type_ensemble eps_et);
void uact1(classe *i,classe *j,type_ensemble sv,type_ensemble *t,type_ensemble *eps);
int egalbottom(type_tp *ti,int n);
int specat(classe *i,classe *j,type_ensemble *t,type_ensemble *eps);
int undefined(classe *i);
void majcples(type_ensemble p,classe *k,classe *l);
void libere(uactpile *pile);
int uactgen(classe *i,classe *j,type_ensemble sv,type_ensemble *t,type_ensemble *eps,type_ensemble *liste);
int ualct(type_ensemble *i,type_ensemble sv,type_ensemble *t,type_ensemble *eps);
int uact(classe *i,classe *j,type_ensemble sv,type_ensemble *t,type_ensemble *eps);
```

```
********************************************************************************** o_ps.c **************
#include <stdlib.h>
#include <stdio.h>

#include "messtruc.h"

#include "ensemble.h"
#include "o_couple.h"
#include "o_operat.h"
#include "stdfunct.h"
#include "o_access.h"
#include "o_types.h"
#include "sortie.h"

#include "o_ps.h"

int F_PS1(paire *p1)
{ return !ps_ng(p1); }

void enleve_selection1_de_ps(type_ensemble *ensemble,int (*fonction)())
{ type_ensemble *ptr=ensemble,ptr2,ens=*ensemble;
  paire *zut;
  while (ens)
    {
        zut=ens->element;
        if (fonction(zut))
          { ptr2=ens;
            *ptr=ens->next;
            free(ens->element);
            free(ptr2);
            ens=*ptr; }
        else
          { ptr=&(ens->next);
            ens=ens->next; }
    }
}

void enleve_i_de_ps(type_ensemble *ensemble,classe *i)
{ type_ensemble *ptr=ensemble,ptr2,ens=*ensemble;
  paire *zut;
  while (ens)
    {
        zut=ens->element;
        if (zut->e1==i || zut->e2==i)
          { ptr2=ens;
            *ptr=ens->next;
            free(ens->element);
            free(ptr2);
            ens=*ptr; }
        else
          { ptr=&(ens->next);
            ens=ens->next; }
    }
}

type_ensemble missing_to_transitivity(type_ensemble p)
{
paire pa1,pa2;
```

```
type_ensemble p1,p2,p3=NULL;

for (p1=p;p1;p1=p1->next)
   for (p2=p;p2;p2=p2->next)
      if (p2!=p1)
        {
         pa1 = *((paire *)p1->element);
         pa2 = *((paire *)p2->element);

         if (pa1.e1=pa2.e1)
            { if (!ps(pa1.e2,pa2.e2,p) && !ps(pa1.e2,pa2.e2,p3))
                ajoute_dans_ens(&p3,buildpaire(pa1.e2,pa2.e2)); }
         else
         if (pa1.e1=pa2.e2)
            { if (!ps(pa1.e2,pa2.e1,p) && !ps(pa1.e2,pa2.e1,p3))
                ajoute_dans_ens(&p3,buildpaire(pa1.e2,pa2.e1)); }
         else
         if (pa1.e2=pa2.e1)
            { if (!ps(pa1.e1,pa2.e2,p) && !ps(pa1.e1,pa2.e2,p3))
                ajoute_dans_ens(&p3,buildpaire(pa1.e1,pa2.e2)); }
         else
         if (pa1.e2=pa2.e2)
            { if (!ps(pa1.e1,pa2.e1,p) && !ps(pa1.e1,pa2.e1,p3))
                ajoute_dans_ens(&p3,buildpaire(pa1.e1,pa2.e1)); }
        }
return p3;
}

int paire_not_acces_var(paire *p)
{
  return !(accesvar(p->e1) && accesvar(p->e2));
}

void removpsinut(type_ensemble *eps)
{
  enleve_selection1_de_ps(eps,paire_not_acces_var);
}

void removpsfaux(type_ensemble *eps)
{
  enleve_selection1_de_ps(eps,paire_defined);
}

int ij(paire *p,classe *i,classe *j,classe **k)
{
if (p->e1=i || p->e1=j) { *k=p->e2; return 1; }
if (p->e2=i || p->e2=j) { *k=p->e1; return 1; }
return 0;
}

int paire_defined(paire *p)
{
  return !(undefined(p->e1) && undefined(p->e2));
}

void replace_paires(type_ensemble *e,void *j,void *i)
{
  void *p1,*p2;
```

```
  type_ensemble z=*e,w;
  while(z)
    { if (((paire *) z->element)->e1 == j || ((paire *) z->element)->e2=j)
        {
         if ((p1=((paire *) z->element)->e1)=j) p1=i;
         if ((p2=((paire *) z->element)->e2)=j) p2=i;
         if (ps(p1,p2,*e))
           {
            w=z;
            z=z->next;
            enleve_elt_de_ens(e,w);
           }
         else
           {
            ((paire *) z->element)->e1=p1;
            ((paire *) z->element)->e2=p2;
            z=z->next;
           }
        }
      else z=z->next;
    }
}

type_ensemble buildmistotrans(type_ensemble p, classe *i, classe *j)
{
classe *k,*l;
type_ensemble p1,p2,p3=NULL;

for (p1=p;p1;p1=p1->next)
    if (ij(p1->element,i,j,&k))
       for (p2=p;p2;p2=p2->next)
           if (p2!=p1 && ij(p2->element,i,j,&l) && !ps(k,l,p) && !ps(k,l,p3))
             ajoute_dans_ens(&p3,buildpaire(k,l));
return p3;
}

type_ensemble buildtransitivity(type_ensemble p)
{
paire pa1,pa2;
type_ensemble p1,p2,p3=NULL;
int added;
for (p1=p;p1;p1=p1->next)
   {
    ajoute_dans_ens(&p3,safe_malloc(sizeof(paire)));
    *((paire *) p3->element)=*((paire *) p1->element);
   }

do
  {
   added=0;
   for (p1=p3;p1;p1=p1->next)
     for (p2=p3;p2;p2=p2->next)
         if (p2!=p1)
           {
            pa1 = *((paire *)p1->element);
            pa2 = *((paire *)p2->element);

            if (pa1.e1==pa2.e1)
```

```
            { if (!ps(pa1.e2,pa2.e2,p3))
                { ajoute_dans_ens(&p3,buildpaire(pa1.e2,pa2.e2));
                  added++; }
            }
          else
            if (pa1.e1==pa2.e2)
              { if (!ps(pa1.e2,pa2.e1,p3))
                  { ajoute_dans_ens(&p3,buildpaire(pa1.e2,pa2.e1));
                    added++; }
              }
            else
              if (pa1.e2==pa2.e1)
                { if (!ps(pa1.e1,pa2.e2,p3))
                    { ajoute_dans_ens(&p3,buildpaire(pa1.e1,pa2.e2));
                      added++; }
                }
              else
                if (pa1.e2==pa2.e2)
                  { if (!ps(pa1.e1,pa2.e1,p3))
                      { ajoute_dans_ens(&p3,buildpaire(pa1.e1,pa2.e1));
                        added++; }
                  }
        }
 } while (added);

for (p1=p3;p1;p1=p1->next)
  {
   pa1 = *((paire *)p1->element);

   if (!ps(pa1.e1,pa1.e1,p3) && accesvar(pa1.e1)) addps(&p3,pa1.e1,pa1.e1);
   if (pa1.e1 != pa1.e2)
      if (!ps(pa1.e2,pa1.e2,p3) && accesvar(pa1.e2)) addps(&p3,pa1.e2,pa1.e2);
  }

return p3;
}

void kill_ps_elements(type_ensemble *eps,classe *elt)
{ type_ensemble *ptr=eps,ptr2,i=*eps;
  paire *zut;
  while (i)
    {
       zut=i->element;
       if (zut->e1==elt || zut->e2==elt)
         { ptr2=i;
           free(i->element);
           i=*ptr=i->next;
           free(ptr2); }
       else
         { ptr=&(i->next);
           i=i->next; }
    }
}

paire *getfirstps(type_ensemble eps,classe *z,classe **c)
{
while(eps)
  { if (((paire *)(eps->element))->e1==z)
```

```
      { *c= ((paire *)(eps->element))->e2;
        return (paire *) eps->element; }
    if (((paire *)(eps->element))->e2==z)
      { *c= ((paire *)(eps->element))->e1;
        return (paire *) eps->element; }
    eps=eps->next; }
return NULL;
}

int nbps(classe *i,type_ensemble e)
{
int cpt=0;
paire *w;
while (e)
  { w=e->element;
    if (w->e1==i || w->e2==i) cpt++;
    e=e->next; }
return cpt;
}

void addpsver(type_ensemble *eps,void *i, void *k)
{
if (!ps(i,k,*eps))
   ajoute_dans_ens(eps,buildpaire(i,k));
}

void addps(type_ensemble *eps,void *i, void *k)
{
ajoute_dans_ens(eps,buildpaire(i,k));
}

int ps_ng(paire *p)
{
return !(ng(((classe *) p->e1)->tp) && ng(((classe *) p->e2)->tp));
}

void psrecover(type_ensemble t,type_ensemble *eps)
{
type_ensemble ie,je;
int i,j,lig,nbvars,nbfunctors=nbvars=0,sw=0;
classe **functs;
unsigned char **tab;

for (ie=t;ie;ie=ie->next)
   if (undefined(ie->element)) nbvars++;
   else if (taille_ensemble(((classe *) ie->element)->forme.arguments)) nbfunctors++;
if (nbfunctors && nbvars)
   { tab=(unsigned char **) safe_malloc (nbfunctors*sizeof(unsigned char *));
     functs=(classe **) safe_malloc (nbfunctors*sizeof(classe *));
     for (i=0;i<nbfunctors;i++)
        { tab[i]=(unsigned char *) safe_malloc ((taille_ensemble(t)+7)>>3);
          for (j=0;j<((taille_ensemble(t)+7)>>3);j++) tab[i][j]=0x00; }
     for (lig=i=0,ie=t;lig<nbfunctors;ie=ie->next,i++)
        if (taille_ensemble(je=((classe *) ie->element)->forme.arguments))
          { functs[lig]=ie->element;
            for (j=0;je;je=je->next,j++) setparties(getpos(t,je->element)-1,tab[lig]);
            lig++; }
     for (i=0,ie=t;ie;ie=ie->next,i++)
```

```
      /*        if (undefined(ie->element)) */
            for (lig=0;lig<nbfunctors;lig++)
              if (getparties(i,tab[lig]))
                 for (j=lig+1;j<nbfunctors;j++)
                     if (getparties(i,tab[j]))
                        if (!ps(functs[lig],functs[j],*eps))
                           { if (!sw) { sw=1; afftabs2(148,148,t,NULL,*eps); }
                             printf ("ps added between %d & %d\n",functs[lig],functs[j]); addps(eps,functs[lig],functs[j]);
}
     for (i=0;i<nbfunctors;i++) free(tab[i]);
     free(tab);
     free(functs); }
if (sw) afftabs2(148,148,t,NULL,*eps);
}

type_ensemble ps_clot_trans(type_ensemble t)
{
return buildtransitivity(t);
}

void removps(type_ensemble *ps,classe *i,classe *j)
{
enleve_paire_de_ens(ps,i,j);
}

int ps(classe *i,classe *j,type_ensemble p)
{
return est_dans_ens_paires(p,i,j);
}

type_ensemble ps_et(type_ensemble t, type_ensemble eps)
{
type_ensemble ps_res=NULL,p1,p2,*eo_ps_res=&ps_res;
classe *c;
paire cps;

for (p1=eps;p1;p1=p1->next)
   {
    append(eo_ps_res,safe_malloc(sizeof(paire)));
    *((paire *) (*eo_ps_res)->element)=*((paire *) p1->element);
    eo_ps_res=&((*eo_ps_res)->next);
   }

for (p1=ps_res;p1;p1=p1->next)
   {
    cps=*((paire*)p1->element);
    for (p2=t;p2;p2=p2->next)
       {
        c=p2->element;
        if (est_dans_ens(c->forme.arguments,cps.e1) && !ps(c,cps.e2,ps_res))
           { append(eo_ps_res,buildcouple(c,cps.e2));
             eo_ps_res=&((*eo_ps_res)->next); }
        if (est_dans_ens(c->forme.arguments,cps.e2) && !ps(c,cps.e1,ps_res))
           { append(eo_ps_res,buildcouple(c,cps.e1));
             eo_ps_res=&((*eo_ps_res)->next); }
       }
   }
```

```
psrecover(t,&ps_res);
return ps_res;
}
```

```
********************************************************************************** o_ps.h **************
#define egale_ps(a,b) ((a)==(b))

int nbps(classe *i,type_ensemble e);
int ps_ng(paire *p);
void addpsver (type_ensemble *t,void *i,void *j);
void addps(type_ensemble *t,void *i,void *j);
type_ensemble ps_clot_trans(type_ensemble t);
void removps(type_ensemble *t,classe *i,classe *j);
int ps(classe *i,classe *j,type_ensemble t);
type_ensemble ps_et(type_ensemble t, type_ensemble eps);
paire *getfirstps(type_ensemble ps,classe *z,classe **c);
void enleve_selection1_de_ps(type_ensemble *ensemble,int (*fonction)());
int F_PS1(paire *p1);
void enleve_i_de_ps(type_ensemble *eps,classe *i);
void kill_ps_elements(type_ensemble *ps,classe *elt);
type_ensemble missing_to_transitivity(type_ensemble e);
type_ensemble buildtransitivity(type_ensemble p);
int ij(paire *p,classe *i,classe *j,classe **k);
type_ensemble buildmistotrans(type_ensemble p,classe *i,classe *j);
int paire_not_acces_var(paire *p);
void removpsinut(type_ensemble *eps);
void removpsfaux(type_ensemble *eps);
int paire_defined(paire *p);
void replace_paires(type_ensemble *e,void *j,void *i);
void psrecover(type_ensemble t, type_ensemble *eps);
```

```
********************************************************************************** o_sv.c **************
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "stdfunct.h"

#include "messtruc.h"
#include "ensemble.h"
#include "o_sv.h"

pilechvar *pilchvar=NULL;

void *est_dans_ens_sv(type_ensemble ens,type_sv element)
{
while (ens)
  { if (egale_sv(*((type_sv *)ens->element),element))
       return ens->element;
    ens=ens->next;
  }
return NULL;
}

void *copy_sv(type_sv elt)
{
type_sv *res=(type_sv *) safe_malloc (sizeof(type_sv));
*res=elt;
return res;
}

void ajoute_dans_sv(type_ensemble *ensemble,type_sv element)
{
type_element *w=(type_element *)safe_malloc(sizeof(type_element));
w->element=copy_sv(element);
w->next=*ensemble;
*ensemble=w;
}

void enleve_de_sv(type_ensemble *ensemble,type_sv element)
{ type_ensemble tmp=*ensemble, *ptr=ensemble,ptr2;
  while (tmp)
       if (egale_sv(*((type_sv *) tmp->element),element))
         { ptr2=tmp;
           *ptr=tmp->next;
           free(tmp->element);
           free(ptr2);
           return; }
       else
         { ptr=&(tmp->next);
           tmp=tmp->next; }
}

type_sv *est_dans_sv_droite(type_ensemble esv,classe *sv)
{
while (esv && ((type_sv *) esv->element)->sv != sv) esv = esv->next;
return esv ? esv->element : NULL;
}
```

```
type_sv *est_dans_sv_gauche(type_ensemble esv,int xi)
{
while (esv && ((type_sv *) esv->element)->xi != xi) esv = esv->next;
return esv ? esv->element : NULL;
}

void addsv(type_ensemble *sv,int xi,classe *v)
{
type_ensemble z=*sv;
type_element *new=(type_element *) safe_malloc (sizeof(type_element));
type_sv *ntype   =(type_sv *)      safe_malloc (sizeof(type_sv));

new->next=NULL;
new->element=ntype;
ntype->xi=xi; ntype->sv=v;

if (!*sv) *sv=new;
else { while (z->next) z=z->next; z->next=new; }
}

classe *trouvsv(type_ensemble sv,int var)
{
while (sv && ((type_sv *) sv->element)->xi != var) sv=sv->next;
return (sv ? (classe *) (((type_sv *) sv->element)->sv) : NULL);
}

void renum(type_ensemble sv)
{
int i;
for (i=1;sv;sv=sv->next,i++) ((type_sv *) sv->element)->xi=i;
}

void depilchvar(type_ensemble sv)
{
if (pilchvar)
     { int *p,l;
       type_ensemble k;
       pilechvar *s=pilchvar;
       printf("\nChangement de variable :\n");
       p=pilchvar->valeur;
       if (*p!=taille_ensemble(sv) && taille_ensemble(sv)) { printf("Erreur nombre d'arguments de depilchvar\n"); return; }
       for (k=sv;k;k=k->next)
           { ((type_sv *) k->element)->xi=p[l=((type_sv *)k->element)->xi];
             printf("x%d->x%d\n",l,p[l]); }
       free(pilchvar->valeur);
       pilchvar=pilchvar->prev;
       free (s); }
}

void chvar(int v[],int n,type_ensemble sv)
{
pilechvar *s;
int *k,i;
type_ensemble ie;

if (n==taille_ensemble(sv))
{ for (i=0;i<n;i++)
    if (!trouvsv(sv,v[i])) { printf("Erreur : Variable %d inconnue\n",v[n]); exit(1); }
```

```
  s=(pilechvar *) safe_malloc(sizeof(pilechvar));
  (*s).prev=pilchvar;
  pilchvar=s;
  k=(*pilchvar).valeur = (int *) safe_malloc((n+1)*sizeof(int));
  *(k++)=n;
  for (i=0,ie=sv;i<n;i++,ie=ie->next)
    { *(k++)=v[i];
      ((type_sv *)ie->element)->xi=ascan(v,n,((type_sv *)ie->element)->xi)+1; }
}
else
  printf("Erreur : nombre de variables incorrect\n");
}
```

```
********************************************************************************** o_sv.h **************
#define egale_sv(a,b) ((a).xi=(b).xi && (a).sv=(b).sv)

void addsv(type_ensemble *sv,int xi,classe *v);
classe *trouvsv(type_ensemble sv,int var);
void renum(type_ensemble sv);
void depilchvar(type_ensemble sv);
void chvar(int v[],int n,type_ensemble sv);
type_sv *est_dans_sv_gauche(type_ensemble esv,int xi);
type_sv *est_dans_sv_droite(type_ensemble esv,classe *sv);
void *est_dans_ens_sv(type_ensemble ens,type_sv element);
void enleve_de_sv(type_ensemble *ensemble,type_sv element);
void *copy_sv(type_sv elt);
void ajoute_dans_sv(type_ensemble *ensemble,type_sv element);
```

```
********************************************************************************** o_types.c ***********
#include <string.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "defines.h"

#include "messtruc.h"

#include "stdfunct.h"

#include "o_types.h"

int ng(type_tp c)
{
return c!=BOTTOM;
}

type_tp lub(type_tp i1,type_tp i2)
/* renvoie le plus petit ancetre commun de deux noeuds */
{
  return i1 | i2;
}

type_tp glb(type_tp i1,type_tp i2)
/* renvoie le plus grand descendant commun de deux noeuds */
{
  return (i1 & i2);
}

type_tp uat(type_tp i1,type_tp i2)
{
  type_tp i3,i4;

  if (i1==BOTTOM || i2==BOTTOM)
    return BOTTOM;

  if (i1==VAR)
    return i2;

  if (i2==VAR)
    return i1;

  if (i1==NLV && i2==NLV)
    return NOVAR;

  if (i1==LIST || i2==LIST)
    return LIST;

  if (i1==ANY && i2==ANY)
    return ANY;

  if (estpluspetit(i1,i2) || i1==NLV) swaptp(&i1,&i2);

  trouvt3t4(i1,&i3,&i4);

  return lub(uat(i3,i2),uat(i4,i2));
```

```
}

void swaptp(type_tp *i1,type_tp *i2)
{
type_tp z=*i1;
*i1=*i2;
*i2=z;
}

int estpluspetit(type_tp i,type_tp j)
{
  return (i&j)==i;
}

void trouvt3t4(type_tp t,type_tp *t3,type_tp *t4)
{
  utype_tp ut=t,tmp;

  tmp=TOPTREE>>1; /* 1000>>1 */
  while (!(tmp & ut)) tmp>>=1;
  (*t3)=(type_tp)(ut & ~tmp);

  tmp=1;
  while (!(tmp & ut)) tmp<<=1;
  (*t4)=(type_tp)(ut & ~tmp);
}

type_tp iat2(type_tp t1,type_tp t2)
{ type_tp t3,t4;

  if (t1==BOTTOM || t2==BOTTOM)
    return BOTTOM;

  if (t1==LIST)
    return LIST;

  if (t1==NLV && (t2==NLV || t2==VAR))
    return NLV;

  if (t1==NLV && t2==LIST)
    return NOVAR;

  if (t1==VAR)
    return lub(VAR,t2);

  if (t1==ANY && t2==ANY)
    return ANY;

  if (t2 != 1 && t2 != 2 && t2 != 4)
      { trouvt3t4(t2,&t3,&t4);
        return lub(iat2(t1,t3),iat2(t1,t4)); }

  trouvt3t4(t1,&t3,&t4);
  return lub(iat2(t3,t2),iat2(t4,t2));
}


type_tp iat(type_tp p)
```

```
( if (p==BOTTOM || p==LIST || p==NOVAR) return p;
  if (p==NLV) return NOVAR;
  return ANY; )

type_tp mat(type_tp t,char *f,type_tp *ti,int n)
{
return glb(iat(t),cons(f,ti,n));
}

type_tp cons(char *foncteur,type_tp *args,int n)
{
int i;
for (i=0;i<n;i++) if (args[i]==BOTTOM) return BOTTOM;
if (n==0 && !strcmp(foncteur,"nil"))
  return LIST;
if (n==2 && !strcmp(foncteur,"cons"))
  if (args[1]==LIST) return LIST;
return NLV;
}

type_tp *extr(type_tp type,char *foncteur,int n)
{
type_tp *res;
if (!n) return NULL;
res=(type_tp *) safe_malloc(n*sizeof(type_tp));
if (type==VAR || (strcmp(foncteur,"cons") && type==LIST))
   while (n--) res[n]=BOTTOM;
else if (!strcmp(foncteur,"cons") && type==LIST && n==2)
      { res[0]=ANY;res[1]=LIST; }
    else while (n--) res[n]=ANY;
return res;
}
```

```
********************************************************************************************** o_types.h ***********
void init(void);
int ng(type_tp c);
type_tp lub(type_tp i1,type_tp i2);
type_tp glb(type_tp i1,type_tp i2);
type_tp uat(type_tp i1,type_tp i2);
void swaptp(type_tp *i1,type_tp *i2);
int estpluspetit(type_tp i,type_tp j);
void trouvt3t4(type_tp t,type_tp *t3,type_tp *t4);
type_tp iat2(type_tp t1,type_tp t2);
type_tp iat(type_tp p);
type_tp mat(type_tp t,char *f,type_tp *ti,int n);
type_tp cons(char *foncteur,type_tp *args,int n);
type_tp *extr(type_tp type,char *foncteur,int n);
```

```
********************************************************************************** solve.c *************
#define TRUE      1
#define FALSE     0

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "messtruc.h"

#include "defines.h"
#include "ensemble.h"
#include "stdfunct.h"
#include "abi.h"
#include "ext2.h"
#include "o_operat.h"
#include "o_sv.h"
#include "o_ps.h"
#include "majlub.h"
#include "betaordr.h"
#include "managest.h"
#include "sortie.h"
#include "solve.h"

extern type_ensemble E_clauses;
extern int iter_count;

char *get_info_pred(int *arite,type_ensemble e)
{
Nnoeud_ex *ptr=e->element;
*arite=taille_ensemble(ptr->ne.app.e);
return ptr->ne.app.v;
}

beta fsat(beta bin,type_ensemble p,type_ensemble sat)
{
type_ensemble i;
int arite;
char *z=get_info_pred (&arite,p->element);
tsat *s;
for (i=sat;i;i=i->next)
   { s=i->element;
     if (arite == s->arite && strcmp(z,s->p)==0)
        if (cmpbbeta(&bin,&(s->bin)))
            return s->bout; }
for (i=sat;i;i=i->next) printf("sat concerne %s d'arite %d\n",s->p,s->arite);
printf("Recherche pour %s d'arite %d\n",z,arite);
printf("Erreur fsat\n");exit(1);
}

beta fsat_susp(beta bin,type_ensemble p,type_ensemble sat,type_ensemble suspended)
{
type_ensemble i;
int arite;
char *z=get_info_pred (&arite,p->element);
tsat *s;
for (i=sat;i;i=i->next)
   { s=i->element;
```

```
    if (arite = s->arite && strcmp(z,s->p)=0)
      if (cmpbbeta(&bin,&(s->bin)))
          return s->bout; }
for (i=suspended;i;i=i->next)
   { s=i->element;
    if (arite = s->arite && strcmp(z,s->p)=0)
      if (cmpbbetatrafique(&bin,&(s->bin)))
          return s->bout; }
for (i=sat;i;i=i->next) printf("sat concerne %s d'arite %d\n",s->p,s->arite);
printf("Recherche pour %s d'arite %d\n",z,arite);
printf("Erreur fsat\n");exit(1);
}

beta solve(beta bin,type_ensemble p)
{
   type_ensemble suspended,sat=suspended=NULL;
   int same_sat=TRUE;
   iter_count=0;
   solve_goal(&bin,p,&suspended,&same_sat,&sat);
   return fsat(bin,p,sat);
}

int in_suspended(beta *bin,type_ensemble p,type_ensemble suspended)
{
type_ensemble i;
int arite;
char *z=get_info_pred (&arite,p->element);
tsat *s;
for (i=suspended;i;i=i->next)
   { s=i->element;
    if (arite = s->arite && strcmp(z,s->p)=0)
      if (cmpbbetatrafique(bin,&(s->bin))) return 1; }
return 0;
}

int in_dom(beta b,type_ensemble p,type_ensemble sat)
{
type_ensemble i;
tsat *s;
int arite;
char *z=get_info_pred (&arite,p->element);
for (i=sat;i;i=i->next)
    { s=i->element;
    if (arite = s->arite && strcmp(z,s->p)=0)
       if (cmpbbeta(&b,&(s->bin))) return 1; }
return 0;
}

void ADJUST(beta b,type_ensemble p, beta bprime, type_ensemble sat)
{
type_ensemble i;
tsat *s;
int arite;
char *z=get_info_pred (&arite,p->element);
for (i=sat;i;i=i->next)
   { s=i->element;
    if (arite = s->arite && strcmp(z,s->p)=0)
      if (betaorder(b,s->bin))
```

```
          { beta z=s->bout;
            s->bout=betalub(s->bout,bprime);
            printf("%cGot it\n",7);
            liberebeta(z); }
   }
}

void EXTEND(beta b,type_ensemble p,type_ensemble *sat)
{
type_ensemble i;
beta T;
int arite;
char *z=get_info_pred (&arite,p->element);
tsat *new=(tsat *) safe_malloc (sizeof(tsat)),*s;
new->bin=copybeta(b);
new->p=strdup(z);
new->arite=arite;
new->bout.sv=new->bout.t=new->bout.eps=NULL;
for (i=*sat;i;i=i->next)
   { s=i->element;
     if (new->arite==s->arite && strcmp(new->p,s->p)==0)
        if (betaorder(s->bin,b))
           { T=betalub(s->bin,new->bout);
             liberebeta(new->bout);
             new->bout=T; }
   }
append(sat,new);
}

void solve_goal(beta *bin,type_ensemble p,type_ensemble *suspended,int *same_sat,type_ensemble *sat)
{
  printf("solve_goal\n");fflush(stdout);
  if (!in_suspended(bin,p,*suspended))
       { if (!in_dom(*bin,p,*sat))
            { *same_sat=FALSE;
              EXTEND(*bin,p,sat); }
         solve_procedure(bin,p,suspended,same_sat,sat);
       }
}

type_ensemble satscan_bin_p(beta *b,type_ensemble p,type_ensemble sat)
{
 type_ensemble i;
 tsat *s;
 int arite;
 char *z=get_info_pred (&arite,p->element);
 for (i=sat;i;i=i->next)
    { s=i->element;
      if (arite = s->arite && strcmp(z,s->p)==0)
          if (cmpbbeta(b,&(s->bin))) return i; }
 return NULL;
}

void unionsuspended(beta b,type_ensemble *s,type_ensemble p,type_ensemble sat)
{
  printf("Unionsuspended\n");
  ajoute_dans_ens(s,satscan_bin_p(&b,p,sat)->element);
}
```

```
void solve_procedure(beta *bin,type_ensemble p,type_ensemble *suspended,int *same_sat,type_ensemble *sat)
{
  type_ensemble tmp;
  int same_sat_aux;
  unionsuspended(*bin,suspended,p,*sat);
  printf("solve_procedure\n");fflush(stdout);
  do { iter_count++;
       same_sat_aux=solve_all_clauses(bin,p,suspended,sat);
       *same_sat = *same_sat && same_sat_aux; } while (!same_sat_aux);
  tmp=(*suspended)->next;
  free(*suspended);
  *suspended=tmp;
}

int solve_all_clauses(beta *bin,type_ensemble p,type_ensemble *suspended,type_ensemble *sat)
{
  type_ensemble ie;
  int same_sat=TRUE;
  beta bout,baux,boutp,zut;
  printf("solve_all_clauses\n");
  bout.sv=bout.t=bout.eps=NULL;
  for (ie=p;ie;ie=ie->next)
    { baux=solve_clause(bin,(type_ensemble) ie->element,suspended,&same_sat,sat);
      printf("Avant lub\n");
/*      afftabs2(0,186,bout.t,bout.sv,bout.eps); */
      afftabs2(1,186,baux.t,baux.sv,baux.eps);
      boutp=betalub(bout,baux);
      liberebeta(baux);
      liberebeta(bout);
      bout=boutp;
/*      puts("resultat lub");
      afftabs2(2,186,bout.t,bout.sv,bout.eps); */ }

/*    printf("Apres lub :\n"); */
/*    puts("before betaorder");
    afftabs2(3,186,bout.t,bout.sv,bout.eps); */
    zut=fsat(*bin,p,*sat);
/*  afftabs2(4,186,zut.t,zut.sv,zut.eps); */
    if (!betaorder(bout,zut))
    { printf("Resultat du test : OK\n");
      puts("sat avant adjust");affsat(*sat);
/*      printf("\n\n ADJUST : bin :\n");
      afftabs2(5,186,(*bin).t,(*bin).sv,(*bin).eps); */
      ADJUST(*bin,p,bout,*sat);
      puts("sat apres adjust");affsat(*sat);
      same_sat=FALSE; }
  else printf("Resultat du test : KO\n");
  liberebeta(bout);
/*  printf("fin_solve_all_clauses%c\n",same_sat?7:20); */
/*       printf("sat\n");
             { type_ensemble i; tsat *s;
               for (i=*sat;i;i=i->next)
                 { s=i->element;
                   printf("bin\n"); afftabs2(6,186,s->bin.t, s->bin.sv, s->bin.eps);
                   printf("p=%s,arite=%d\n",s->p,s->arite);
                   printf("bext\n");afftabs2(7,186,s->bout.t,s->bout.sv,s->bout.eps);
                   printf("_____\n");fflush(stdout);   } } */
```

```
  return same_sat;
}

beta EXTC(type_ensemble p,beta *bin)
{
beta res=copybeta(*bin);
if ((*bin).sv || (*bin).t || (*bin).eps)
   {
    int n=0;
    type_ensemble ie,je;
    Nnoeud_ex *ne;
    for (ie=p;ie;ie=ie->next)
       {
        ne=ie->element;
        switch(ne->type)
           { case NABI: { n=max(n,max(ne->ne.abi1.v1,ne->ne.abi1.v2)); break; }
             case NABI2:{ n=max(n,ne->ne.abi2.v1);
                          for (je=ne->ne.abi2.e;je;je=je->next)
                              n=max(n,((Nnoeud_ex *)je->element)->ne.v.v);
                          break; }
             case PROC: { for (je=ne->ne.app.e;je;je=je->next)
                              n=max(n,((Nnoeud_ex *)je->element)->ne.v.v);
                          break; }
           }
       }
    extension(n,&(res.sv),&(res.t),&(res.eps));
   }

return res;
}

void setNLV(type_ensemble c)
{
for (;c;c=c->next)
   ((classe *)c->element)->tp=NLV;
}

beta solve_clause(beta *bin,type_ensemble p,type_ensemble *suspended,int *same_sat,type_ensemble *sat)
{
   type_ensemble ie;
   beta bext=EXTC(p,bin),baux;
   printf("solve_clause\n");
   for (ie=p->next;ie;ie=ie->next)
     { beta z;
       Nnoeud_ex *pe=ie->element;
       if (bext.sv==NULL && bext.t==NULL && bext.eps==NULL) return bext;
       switch(pe->type)
        { case NABI: (printf("x%d%cx%d\n",pe->ne.abi1.v1, pe->ne.abi1.typ_egal, pe->ne.abi1.v2); break; }
          case NABI2:(printf("x%d%c%s\n",pe->ne.abi2.v1, pe->ne.abi2.typ_egal, pe->ne.abi2.v2); break; }
          case PROC: (printf("%s\n",pe->ne.app.v); break; }
        }
/*        printf("avant restriction\n");
       afftabs2(0,193,bext.t,bext.sv,bext.eps); */
       baux=RESTRB(pe,bext);
/*        printf("apres restriction\n");
       afftabs2(1,193,baux.t,baux.sv,baux.eps); */
       switch(pe->type)
         {
```

```
        case NABI:  { if (pe->ne.abi1.typ_egal!='=') { setNLV(baux.t); afftabs2(2,193,baux.t,baux.sv,baux.eps); }
                      else betabi(&baux); break; }
        case NABI2: { if (pe->ne.abi2.typ_egal!='=') setNLV(baux.t);
                      else betabi2(&baux,strdup(pe->ne.abi2.v2)); break; }
        case PROC:  { printf("recursive call\n");fflush(stdout);
/*                      if (strcmp(pe->ne.app.v,"append")==0)
                                { puts("\n\nAPPEND\n\n"); afftabs2(3,193,baux.t,baux.sv,baux.eps); inkey(); } */
                      solve_goal(&baux,pe->ne.app.matching,suspended,same_sat,sat);
/*                      if (strcmp(pe->ne.app.v,"append")==0)
                                { puts("\n\nBACK\n\n"); inkey(); } */
                      printf("back of recursive call\n");
                      z=fsat_susp(baux,pe->ne.app.matching,*sat,*suspended);
                      liberebeta(baux);
                      baux=copybeta(z); }
      }
    printf("avant ext2\n");
    afftabs2(4,193,baux.t,baux.sv,baux.eps);
    printf("bext\n");
    afftabs2(5,193,bext.t,bext.sv,bext.eps);
    depilchvar(baux.sv);
    bext=EXTB(bext,baux);
    printf("apres extb\n");
    afftabs2(6,193,bext.t,bext.sv,bext.eps);
/*    psrecover(&bext);
    puts("after psrecover de solve.c");
    afftabs2(7,193,bext.t,bext.sv,bext.eps); */
/*    printf("sat\n");
          { type_ensemble i; tsat *s;
            for (i=*sat;i;i=i->next)
              { printf("bin\n");
                s=i->element;
                afftabs2(8,193,s->bin.t, s->bin.sv, s->bin.eps);
                printf("p=%s,arite=%d\n",s->p,s->arite);
                printf("bext\n");
                afftabs2(9,193,s->bout.t,s->bout.sv,s->bout.eps);
                printf("_____\n");fflush(stdout);   ) ) */
/*          { int c=0; while (c!=65) c=getchar(); ) */
  }
  return RESTRC(p,bext);
}

beta RESTRB(Nnoeud_ex *c,beta b)
{
beta res=copybeta(b);
if (res.sv || res.t || res.eps)
 {
  int n,*v,i;
  type_ensemble t;
  switch(c->type)
    {
     case NABI: { n=2; v=(int *) safe_malloc(2*sizeof(int)); v[0]=c->ne.abi1.v1; v[1]=c->ne.abi1.v2; break; }
     case NABI2:{ v=(int *) safe_malloc((n=taille_ensemble(c->ne.abi2.e)+1)*sizeof(int)); v[0]=c->ne.abi2.v1;
                  for (i=1,t=c->ne.abi2.e;i<n;i++,t=t->next) v[i]=((Nnoeud_ex *)t->element)->ne.v.v; break; }
     case PROC: { n=taille_ensemble(c->ne.app.e);v=(int *) safe_malloc(n*sizeof(int));
                  for (i=0,t=c->ne.app.e;i<n;i++,t=t->next) v[i]=((Nnoeud_ex *)t->element)->ne.v.v; break; }
    }

  restr1(v,n,&(res.sv),&(res.t),&(res.eps),1);
```

```
   if (n) free(v);
  }

return res;
}

beta RESTRC(type_ensemble p,beta bin)
{
puts("restrc");
if (bin.sv || bin.t || bin.eps)
   {
    int n=taille_ensemble(((Nnoeud_ex *)p->element)->ne.app.e),i,*v=(int *) safe_malloc(n*sizeof(int));
    for (i=0;i<n;i++) v[i]=i+1;
    restr1(v,n,&(bin.sv),&(bin.t),&(bin.eps),0);
   }
return bin;
}

beta EXTB(beta b1,beta b2)
{
beta res;
if (b1.sv==NULL && b1.t==NULL && b1.eps==NULL) return b1;
if (b2.sv==NULL && b2.t==NULL && b2.eps==NULL) return b2;
if (extension2(b1.sv,b2.sv,b1.t,b2.t,b1.eps,b2.eps,&(res.sv),&(res.t),&(res.eps))==-1)
                { liberebeta(res);
                  res.eps=res.sv=res.t=NULL; }
return res;
}
```

```
********************************************************************************** solve.h *************
beta fsat(beta bin,type_ensemble p,type_ensemble sat);
beta fsat_susp(beta bin,type_ensemble p,type_ensemble sat,type_ensemble suspended);
beta solve(beta bin,type_ensemble p);
int in_suspended(beta *bin,type_ensemble p,type_ensemble suspended);
int in_dom(beta b,type_ensemble p,type_ensemble sat);
void ADJUST(beta b,type_ensemble p, beta bprime, type_ensemble sat);
void EXTEND(beta b,type_ensemble p,type_ensemble *sat);
void solve_goal(beta *bin,type_ensemble p,type_ensemble *suspended,int *same_sat,type_ensemble *sat);
type_ensemble satscan_bin_p(beta *b,type_ensemble p,type_ensemble sat);
void unionsuspended(beta b,type_ensemble *s,type_ensemble p,type_ensemble sat);
void solve_procedure(beta *bin,type_ensemble p,type_ensemble *suspended,int *same_sat,type_ensemble *sat);
int testclause(int p,int c,char *string);
int trouvclause (int p, int c, int *j);
int solve_all_clauses(beta *bin,type_ensemble p,type_ensemble *suspended,type_ensemble *sat);
beta EXTC(type_ensemble p,beta *bin);
beta solve_clause(beta *bin,type_ensemble p,type_ensemble *suspended,int *same_sat,type_ensemble *sat);
beta RESTRB(Nnoeud_ex *c,beta b);
beta RESTRC(type_ensemble p,beta bin);
beta EXTB(beta b1,beta b2);
```

```
********************************************************************************** sortie.c ************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "messtruc.h"
#include "ensemble.h"

#include "defines.h"

#include "sortie.h"

char buf[1000];
extern pilechvar *pilchvar;

void afftree2(type_ensemble tree)
{
type_ensemble z;
for (z=tree;z;z=z->next,putchar(' '))
   affclause2(z->element);
putchar('\n');
}

void affclause2(Nnoeud_ex *p)
{
  type_ensemble t;
  switch(p->type)
   { case NABI: { printf("x%d%cx%d",p->ne.abi1.v1,p->ne.abi1.typ_egal,p->ne.abi1.v2);
                  break; }
     case NABI2:{ printf("x%d%c%s(",p->ne.abi2.v1,p->ne.abi2.typ_egal,p->ne.abi2.v2);
                  for (t=p->ne.abi2.e;t;t=t->next)
                      { affclause2(t->element);
                        if (t->next) putchar(','); }
                  putchar(')');
                  break; }
     case PROC:{  printf("%s(",p->ne.app.v);
                  for (t=p->ne.app.e;t;t=t->next)
                      { affclause2(t->element);
                        if (t->next) putchar(','); }
                  putchar(')');
                  break; }
     case NVAR:  { printf("x%d",p->ne.v.v);
                  break; }
     case ABI3:{  printf("%s(",p->ne.fun.v1);
                  for (t=p->ne.fun.e1;t;t=t->next)
                      { affclause2(t->element);
                        if (t->next) putchar(','); }
                  printf(")%c%s(",p->ne.fun.typ_egal,p->ne.fun.v2);
                  for (t=p->ne.fun.e2;t;t=t->next)
                      { affclause2(t->element);
                        if (t->next) putchar(','); }
                  putchar(')');
                  break; }
     default:{ printf("type  : %d\n",(int)p->type); }
   }
}

char *typestring(type_tp val)
```

```
{
  switch (val) {
  case 0: { strcpy(buf,"bottom");  break; } /* 000 */
  case 4: { strcpy(buf,"list");    break; } /* 100 */
  case 2: { strcpy(buf,"nlv");     break; } /* 010 */
  case 1: { strcpy(buf,"var");     break; } /* 001 */
  case 6: { strcpy(buf,"novar");   break; } /* 110 */
  case 5: { strcpy(buf,"lv");      break; } /* 101 */
  case 3: { strcpy(buf,"nolist");  break; } /* 011 */
  case 7: { strcpy(buf,"any");     break; } /* 111 */
  default :
       { char *p=buf+6;
         type_tp i=1;
         strncpy(buf,"union(",6);
         for (i=1;i!=TOPTREE;i<<=1)
            if (i & val)
              { switch(i) {
                   case 4: { strcpy(p,"list"); p+=4; break; } /* 100 */
                   case 2: { strcpy(p,"nlv");  p+=3; break; } /* 010 */
                   case 1: { strcpy(p,"var");  p+=3; break; } /* 001 */
                         }
                *p++=','; }
         strcpy(p-1,")"); }
  }
  return buf;
}

void affsat(type_ensemble sat)
{
int i;
tsat *elt;
for (i=0;sat;sat=sat->next,i++)
  { printf("bin[%d] :\n_______\n",i);
    elt=sat->element;
    afftabs(elt->bin.sv,elt->bin.t,elt->bin.eps);
    printf("\np=%s,arite=%d\nbout[%d] :\n_______\n",elt->p,elt->arite,i);
    afftabs(elt->bout.sv,elt->bout.t,elt->bout.eps);
    printf("\n\n\n"); }
}

void affensemble(char *p,type_ensemble a)
{
int i=1;

if (p) puts(p);

while(a) { printf("%d o elt : %x, elt :%x\n",i++,a,a->element); a=a->next; }
puts("_____");
}

void afftabs2(int k,int l,type_ensemble t,type_ensemble sv,type_ensemble eps)
{
int i;
type_ensemble ind,ind2;
classe *z;
printf ("** %d,%d **\n",k,l);
if (sv==NULL && t==NULL && eps==NULL) { printf("bottom\n"); return; }
for (;sv;sv=sv->next)
```

```
    printf("sv : x%d, %d\n",((type_sv *)(sv->element))->xi,getpos(t,((type_sv *)(sv->element))->sv));
for (i=1,ind=t;ind;ind=ind->next,i++)
   {
     z=ind->element;
     printf("%d %s\n",i,typestring(z->tp));
     if (z->forme.functor) { printf("%s",z->forme.functor); if (!z->forme.arguments) putchar('\n'); }
     if (ind2=z->forme.arguments)
                      { printf("(");
                        for (;ind2;ind2=ind2->next)
                            printf(ind2->next ? "%d,":"%d",getpos(t,ind2->element));
                        printf(")\n"); }
   }
for (i=1,ind=eps;ind;ind=ind->next,i++)
   printf(i%6==0?"(%d,%d)\n":"(%d,%d)   ",getpos(t,((paire *)(ind->element))->e1),getpos(t,((paire *)(ind->element))->e2));
if (i%6!=1) putchar('\n');
}

void afftabs(type_ensemble sv,type_ensemble t,type_ensemble eps)
{
int i;
pilechvar *s=pilchvar;
type_ensemble ind,ind2;
classe *z;
if (sv==NULL && t==NULL && eps==NULL) { printf("bottom\n"); return; }
for (;sv;sv=sv->next)
    printf("sv : x%d, %d\n",((type_sv *)(sv->element))->xi,getpos(t,((type_sv *)(sv->element))->sv));
for (i=1,ind=t;ind;ind=ind->next,i++)
   {
     z=ind->element;
     printf("%d %s\n",i,typestring(z->tp));
     if (z->forme.functor) { printf("%s",z->forme.functor); if (!z->forme.arguments) putchar('\n'); }
     if (ind2=z->forme.arguments)
                      { printf("(");
                        for (;ind2;ind2=ind2->next)
                            printf(ind2->next ? "%d,":"%d",getpos(t,ind2->element));
                        printf(")\n"); }
   }
for (i=1,ind=eps;ind;ind=ind->next,i++)
   printf(i%6==0?"(%d,%d)\n":"(%d,%d)   ",getpos(t,((paire *)(ind->element))->e1),getpos(t,((paire *)(ind->element))->e2));
if (i%6!=1) putchar('\n');
if (s)
printf("\nChangements de variables :\n");
i=1;
while (s)
     { int *p,k,j;
       printf("\n%d) \n",i++);
       p=s->valeur+1;
       j=*(s->valeur);
       for (k=0;k<j;k++) printf(" x%d-->x%d\n",*(p+k),k+1);
       s=s->prev;
     }
}

void printtabs(type_ensemble sv,type_ensemble t,type_ensemble eps,int sw)
{
FILE *fich=fopen("printer","at");
int i;
type_ensemble ind,ind2;
```

```
fputs(sw ? "apres\n":"avant\n",fich);

if (sv==NULL && t==NULL && eps==NULL) { fputs("bottom\n",fich); return; }
for (;sv;sv=sv->next)
    fprintf(fich,"sv : x%d, %d\n",((type_sv *)(sv->element))->xi,getpos(t,((type_sv *)(sv->element))->sv));
for (i=1,ind=t;ind;ind=ind->next,i++)
     printf("%d %s\n",i,typestring(((classe *)(ind->element))->tp));
fputs("frm   ",fich);
for (i=1,ind=t;ind;ind=ind->next,i++)
   if (((classe *) (ind->element))->forme.functor || ((classe *) ind->element)->forme.arguments)
    { fprintf(fich,"%d  ",i);
      if (((classe *)(ind->element))->forme.functor)
        fprintf(fich,"%s  ",((classe *)(ind->element))->forme.functor);
      for (ind2=((classe *)(ind->element))->forme.arguments;ind2;ind2=ind2->next)
        fprintf(fich,"%d  ",getpos(t,ind2->element));
      fprintf(fich,"\n"); }
for (i=1,ind=eps;ind;ind=ind->next,i++)
   printf(i%6==0?"(%d,%d)\n":"(%d,%d)   ",getpos(t,((paire *) ind->element)->e1),getpos(t,((paire *) ind->element)->e2));
if (i%6!=1) putchar('\n');
fclose(fich);
}

void prsusp(beta *bin, int p, int c, type_ensemble s)
{
FILE *fich=fopen("susp","at");
int i,l,sw;
type_ensemble ind,ind2,ind3;
fprintf(fich,"BIN :\n____\n");
for (ind=(*bin).sv;ind;ind=ind->next)
   fprintf(fich,"x%d  %d\n",((type_sv *)(ind->element))->xi,getpos((*bin).t,((type_sv *)(ind->element))->sv));
for (i=1,ind=(*bin).t;ind;i++,ind=ind->next)
   fprintf(fich,"%d  %s\n",i,typestring(((classe *)(ind->element))->tp));
for (sw=0,i=1,ind=(*bin).t;ind;ind=ind->next,i++)
  {
   classe *z=ind->element;
   if (z->forme.functor || z->forme.arguments)
    { if (sw++==0) fputs("frm   ",fich);
      fprintf(fich,"%d  ",i);
      if (z->forme.functor) fprintf(fich,"%s  ",z->forme.functor);
      for (ind2=z->forme.arguments;ind2;ind2=ind2->next)
        fprintf(fich,"%d  ",getpos((*bin).t,ind2->element));
      fprintf(fich,"\n"); } }
for (i=1,ind=(*bin).eps;ind;ind=ind->next,i++)
   fprintf(fich,i%6==0?"(%d,%d)\n":"(%d,%d)   ",getpos((*bin).t,((paire *) ind->element)->e1),getpos((*bin).t,((paire *) ind-
>element)->e2));
if (i%6!=1) fputc('\n',fich);
fprintf(fich,"p=%d,c=%d\n\n=%d\n",p,c);
for (l=0,ind3=s;ind3;ind3=ind3->next,l++)
 {
  fprintf(fich,"suspended b[%d] :\n____\n",l);
  for (ind=((tsat *)(s->element))->bin.sv;ind;ind=ind->next)
    fprintf(fich,"x%d  %d\n",((type_sv *)(ind->element))->xi,getpos(((tsat *)(s->element))->bin.t,((type_sv *)(ind-
>element))->sv));
  for (i=1,ind=((tsat *)(s->element))->bin.t;ind;ind=ind->next,i++)
    fprintf(fich,"%d  %s\n",i,typestring(((classe *)(ind->element))->tp));
  for (sw=0,i=1,ind=((tsat *)(s->element))->bin.t;ind;ind=ind->next,i++)
    {
```

```
      classe *z=ind->element;
      if (z->forme.functor || z->forme.arguments)
        { if (sw++==0) fputs("frm   ",fich);
          fprintf(fich,"%d  ",i);
          if (z->forme.functor) fprintf(fich,"%s  ",z->forme.functor);
             for (ind2=z->forme.arguments;ind2;ind2=ind2->next)
                 fprintf(fich,"%d  ",getpos(((tsat *)(s->element))->bin.t,ind2->element));
          fprintf(fich,"\n"); } }
   for (i=1,ind=((tsat *)(s->element))->bin.eps;ind;ind=ind->next,i++)
      fprintf(fich,i%6==0?"(%d,%d)\n":"(%d,%d)   ",getpos(((tsat *)(s->element))->bin.t,((paire *)(ind->element))-
>e1),getpos(((tsat *)(s->element))->bin.t,((paire *)(ind->element))->e2));
   if (i%6!=1) fputc('\n',fich);
   fprintf(fich,"\np=%s\n\n",((tsat *)(ind3->element))->p); }
fprintf(fich,"_____________________________________________________________________________\n");
fclose(fich);
}
```

```
********************************************************************************** sortie.h ************
char *typestring(type_tp val);
void affsat(type_ensemble sat);
void affensemble(char *p,type_ensemble a);
void afftabs2(int k,int l,type_ensemble t,type_ensemble sv,type_ensemble eps);
void afftabs(type_ensemble sv,type_ensemble t,type_ensemble eps);
void printtabs(type_ensemble sv,type_ensemble t,type_ensemble eps,int sw);
void prsusp(beta *bin, int p, int c, type_ensemble s);
void afftree2(type_ensemble t);
void affclause2(Nnoeud_ex *p);
```

```
********************************************************************************** stdfunct.c **********
#define NEWLINE 10

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "stdfunct.h"

extern char * safe_malloc(int i);

void swapptrs(void **i,void **j)
{
void *k=*j;
*j=*i;
*i=k;
}

void *ptrmin(void *i,void *j)
{
return (i>j) ? j : i;
}

void *ptrmax(void *i,void *j)
{
return (i>j) ? i : j;
}

int readln(FILE *fich,char *buf)
{
int i=0,c=fgetc(fich);
while (c!=EOF && c!=NEWLINE)
  {
   buf[i++]=(char) c;
   c=fgetc(fich);
  }
buf[i]=NULL;
return ((c==EOF && i==0) ? -1 : 0);
}

void swapint(int *i1,int *i2)
{
int j=*i2;*i2=*i1;*i1=j;
}

int partition(int i,int j,int tab[])
{
 int x=tab[i]; /*ig*/
 ig: if (i==j) { tab[i]=x;return i; }
     if (tab[j]>=x) { j--;goto ig; }
     else tab[i++]=tab[j];
 id: if (i==j) {tab[i]=x;return i; }
     if (tab[i]<=x) { i++;goto id; }
     else {tab[j--]=tab[i];goto ig; }
}

void quiksort(int tab[],int u,int v)
{
```

```
 int s,ptrpile=0,pile1[10],pile2[10];
 debut:
 if (u<v) { s=partition(u,v,tab);
          if (s-u < v-s)
             { pile1[ptrpile]=s+1;
               pile2[ptrpile++]=v;
               v=s-1; goto debut; }
          else
             { pile1[ptrpile]=u;
               pile2[ptrpile++]=s-1;
               u=s+1; goto debut; }
        }
 if (ptrpile) { u=pile1[--ptrpile];
                v=pile2[ptrpile];
                goto debut;         }
}

void setparties(int i,unsigned char *parties)
{ parties+=(i>>3);*parties|= (unsigned char)(0x80>>(i%8)); }

int getparties(int i,unsigned char *parties)
{ parties+=(i>>3);return ((*parties) & ((unsigned char)(0x80>>(i%8)))); }

int ascan(int *array,int n,int value)
{
int i;
for (i=0;i<n;i++) if (array[i]==value) return i;
return -1;
}

int nbchif(int i)
{
if (i<10) return 1;
else if (i<100) return 2;
else if (i<1000) return 3;
else return 4;
}

int ascanstring(char **array, int n, char *valeur)
{
int i;
for (i=0;i<n;i++) if (strcmp(array[i],valeur)==0) return i;
return -1;
}

char *monstrcat(char *deb,int l1,char *suit,int l2)
{
  char *res=(char *) safe_malloc(l1+l2+1);
  strncpy(res,deb,l1);
  strncpy(res+l1,suit,l2);
  res[l1+l2]=NULL;
  return res;
}

char *monstcpy(char *fin,char *deb)
{
char *res=(char *)safe_malloc((fin-deb+2)*sizeof(char));
strncpy(res,deb,fin-deb+1);
```

```
res[fin-deb+1]=NULL;
return res;
}

#ifndef min
int min(int a,int b)
{
return a>b ? b : a;
}
#endif

#ifndef max
int max(int a,int b)
{
return a<b ? b : a;
}
#endif

int empty(char *p)
{
while (*p) if (*p++>0x20) return 0;
return 1;
}

void ains(int **tab,int *n,int val)
{
if ((*n)++==0)
   { *tab=(int *)safe_malloc(sizeof(int));**tab=val; }
else
   { int i;
     *tab=(int *)realloc(*tab,*n*sizeof(int));
     for (i=0;i<*n-1 && (*tab)[i]<val;) i++;
     if (i==*n-1) *tab[i]=val;
     else
       { int j;
         for (j=*n-2;j>=i;j--) (*tab)[j+1]=(*tab)[j];
         (*tab)[i]=val; }
   }
}
```

```
********************************************************************************************** stdfunct.h **********
void swapptrs(void **i,void **j);
void *ptrmin(void *i,void *j);
void *ptrmax(void *i,void *j);
int readln(FILE *fich,char *buf);
void swapint(int *i1,int *i2);
int partition(int i,int j,int tab[]);
void quiksort(int tab[],int u,int v);
void setparties(int i,unsigned char *parties);
int getparties(int i,unsigned char *parties);
int ascan(int *array,int n,int value);
int nbchif(int i);
int ascanstring(char **array, int n, char *valeur);
char *monstrcat(char *deb,int l1,char *suit,int l2);
char *monstcpy(char *fin,char *deb);
#ifndef min
int min(int a,int b);
#endif
#ifndef max
int max(int a,int b);
#endif
int empty(char *p);
void ains(int **tab, int *n, int val);
```