Un système déclaratif de types pour PROLOG

Lan Nguyen Phuong

THESE

présentée par

Lan NGUYEN PHUONG

pour obtenir le grade de **DOCTEUR**

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

(arrêté ministériel du 23 novembre 1988)

Spécialité : INFORMATIQUE

==========

UN SYSTEME DECLARATIF DE TYPES POUR PROLOG

==========

Date de soutenance : 23 Septembre 1992

Composition du jury :	Président	Jacques Mossière
	Rapporteurs	Daniel Herman
		Jean-François Pique
	Examinateurs	Yann Rouzaud
		Laurent Trilling

Thèse préparée au sein du Laboratoire de Génie Informatique

Résumé :

Cette thèse présente un système de types pour le langage Prolog, intégrant la notion de mode, c'est-à-dire le degré d'instanciation des termes. Le type d'un prédicat est caractérisé par les types d'appel et de retour de ses arguments, ce qui permet d'en spécifier plus finement le comportement.

Le système est déclaratif : l'utilisateur doit fournir les types de tous les prédicats. Un langage de types est défini, permettant d'exprimer la notion de polymorphisme paramétrique, ainsi que la relation d'inclusion entre types.

Une condition suffisante de bon typage est présentée et validée théoriquement. Cette condition permet de vérifier qu'un programme a un comportement compatible avec les spécifications de types de ses prédicats.

Un prototype du système a été réalisé, permettant une validation pratique de la condition de bon typage.

Mots clés : Prolog, types, polymorphisme, modes.

TABLE DES MATIERES

Introduction

Il est communément admis que la notion de type est une notion essentielle aux langages de programmation, pour une exécution efficace des programmes et une meilleure sûreté dans l'utilisation des constructions d'un langage. La description des types de données utilisées fait par ailleurs partie de la documentation d'un programme.

Il est donc naturel d'envisager de typer Prolog, et ce thème a fait l'objet de nombreux travaux ces dernières années. Cependant, un consensus est loin d'avoir été trouvé, sans doute parce que les bonnes notions n'ont pas encore été découvertes. En effet, tous les travaux ont une caractéristique commune : ils ne traitent qu'une partie du langage, et leur extension pose une série de problèmes non résolus à ce jour.

Le travail de cette thèse a été consacré à l'étude d'un système déclaratif de types polymorphes pour le langage Prolog et au développement d'un prototype de ce système.

Nous avons particulièrement étudié ici une notion rarement abordée dans le contexte de Prolog : le polymorphisme inclusif. Grossièrement, il s'agit de permettre à un terme d'appartenir à plusieurs types non nécessairement disjoints. Cela permet de définir plusieurs profils pour le même prédicat. Par exemple, l'addition peut s'effectuer sur des entiers ou des entiers naturels.

Cette notion ne s'introduit cependant pas de façon immédiate en Prolog : l'étude d'un seul exemple (cf. 1.4.2) montre l'interaction avec une caractéristique du langage, la non-directionnalité des arguments d'un prédicat. Nous avons ainsi été amenés à prendre en compte la notion de mode (le degré d'instanciation d'un terme) qui permet de spécifier les flot de données des prédicats. L'originalité de notre approche est que nous avons intégré les notions de mode et de type, plutôt que de les considérer classiquement comme indépendantes. Il en résulte une plus grande finesse des modes, et une plus grande richesse des types. Il nous est par exemple facile de décrire l'ensemble des listes *semi-closes* (cf. 2.3.1) : leur nombre d'éléments est instancié (connu), mais les éléments ne le sont pas nécessairement.

Nous divisons ce rapport de thèse de la manière suivante :

- Le chapitre 1 présente les concepts fondamentaux d'un système de types pour Prolog : la notion de type et les différentes approches de typage. Des systèmes

de types concrets sont présentés pour illustrer ces approches, montrer les problèmes classiques d'un système de types Prolog et les solutions possibles. Le rôle de la notion de mode pour le langage et dans les systèmes de types est également justifié. Le chapitre est terminé par une discussion sur le choix des méthodes de notre système.

- Dans le chapitre 2, nous décrivons notre système de types du point de vue de l'utilisateur : le langage de types, les différents types qui peuvent être définis et utilisés, les formes de déclaration des prédicats, la relation d'ordre principale entre les types : relation "plus modée", et le principe de vérification du bon typage d'un programme. Plusieurs exemples sont montrés pour illustrer les capacités du système dans la détection d'erreurs.

- La fondation théorique du système est présentée dans le chapitre 3 : la notion de type, ses propriétés principales, la relation "plus modée" entre les types, les conditions du bon typage, et le théorème central, qui affirme la correction de types des programmes bien typés, sont définis et prouvés théoriquement.

- Le chapitre 4 décrit l'implantation du prototype de ce système. L'organisation interne des déclarations, les algorithmes et les techniques utilisées pour mettre en œuvre les concepts théoriques du système sont présentés.

- Le dernier chapitre présente un bilan général du système de types réalisé, et les développements envisageables.

Dans toute cette thèse, la syntaxe Prolog Edinbourgh est utilisée. Pour le lecteur plus familiarisé avec Prolog II ([Colmerauer 82]), rappelons que les constantes commencent par une minuscule et les variables par une majuscule. Par ailleurs, les listes s'écrivent entre crochets, par exemple, [], [1,2,3], [a | L]. Enfin, la notation p/n désigne un prédicat n-aire.

Chapitre 1

TYPER PROLOG

Dans ce chapitre d'introduction aux systèmes de types Prolog, après avoir présenté le rôle des types pour le langage, nous décrivons la notion de type communément définie dans les systèmes de types actuels avec ses propriétés les plus caractéristiques. Les différentes approches de typage sont ensuite analysées et illustrées par des systèmes de types concrets. La dernière partie est consacrée à une discussion qui conduit à la conception de notre système.

1.1 Avantages de la notion de type pour Prolog

La notion de type est largement utilisée dans les langages de programmation. Un type est tout d'abord un ensemble de valeurs que l'on sait représenter efficacement en machine. Mais c'est aussi un moyen de contrôler l'utilisation des constructions d'un langage. L'exemple des langages fortement typés montre la bénéfice d'un contrôle de type effectué statiquement, à la compilation : de nombreuses erreurs sont détectées plus tôt et plus systématiquement.

Etant un langage faiblement typé, Prolog ne facilite pas la mise au point de ses programmes. Les seules erreurs de type dans les programmes proviennent d'un mauvais usage des prédicats prédéfinis. Un programme ne contient aucune information redondante sur les constructions utilisées. Tout argument d'un appel de prédicat appartient à l'univers des termes Prolog, tout appel est correct : en particulier ni les noms des prédicats, ni leur nombre d'arguments ne sont vérifiés.

Cependant, chaque prédicat est toujours défini par rapport à une spécification éventuellement implicite (l'intention du programmeur), sur ce qu'il doit faire et sur la structure de ses données. Cette spécification implique donc, pour chacun des arguments, un type qu'il doit satisfaire dans tous les appels du prédicat.

Les incohérences de type, ayant lieu quand un argument d'un prédicat n'est pas de la structure désirée, existent donc toujours. Elles ont pour effet des comportements non désirés, des résultats inattendus des programmes.

Considérons, par exemple, l'addition des entiers naturels, où *zéro* représente l'entier *0* et *s(N)*, l'entier *n+1*, où *n* est l'entier représenté par *N*.

```
add(zéro, N, N).
add(s(X), Y, s(Z))  :–  add(X, Y, Z).
```

L'addition est ici implicitement spécifiée sur les entiers, et cependant le prédicat peut réussir lorsque le deuxième et le troisième argument ne sont pas des entiers, comme dans le but *:– add(s(zéro), ab, s(ab)).*

Les incohérences de type, parfois conséquences de simples fautes de frappe (donnant lieu à des noms incorrects de foncteur, de variable), ne sont pas détectées, ni à la compilation, ni à l'exécution. La mise au point des programmes en devient parfois vraiment difficile.

Examinons maintenant le programme suivant, où *rev/2* sert à mettre une liste donnée dans l'ordre inverse :

```
rev([], []).
rev([A | L], R)  :–  rev(L, R1), append(r1, [A], R).

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

Une faute de frappe a remplacé la variable *R1* par la constante *r1*, dans la deuxième clause de *rev/2*. Le sous-but *append(r1, [A], R)* ne peut jamais réussir, et la question *:– rev([1,2], [2,1])*, par exemple, échoue.

Ou encore soit le programme, qui définit le prédicat *sous_ens/2*, supposé vérifier l'inclusion de deux ensembles représentés sous forme de listes :

```
sous_ens([], _).
sous_ens([E | Es1], Es2)  :–  élément(Es2, E), sous_ens(Es1, Es2).

élément(X, [X | _]).
élément(X, [_ | L])  :–  élément(X, L).
```

Ici, le prédicat *élément/2* est défini pour vérifier l'appartenance du premier argument à une liste d'éléments. L'utilisation incorrecte de *élément/2* dans la deuxième clause de *sous_ens/2* (les arguments ont été permutés), fait échouer la question *:– sous_ens([a, b], [a, b, c])*, qui est en réalité vraie par rapport à la spécification du prédicat.

L'utilisateur peut éviter ces incohérences en écrivant, par exemple pour le prédicat *add/3*, la version suivante, cette fois conforme à la spécification implicite :

```
add(zéro, N, N) :– nat(N).
add(s(X), Y, s(Z)) :– add(X, Y, Z).
nat(zéro).
nat(s(N)) :– nat(N).
```

Cette version est cependant inefficace, puisque la vérification que *N* est un entier est effectuée pendant l'exécution.

Définir une notion de type pour Prolog devrait permettre d'effectuer des contrôles statiques sur un programme, sans nuire à son exécution efficace : dans l'exemple précédent, le prédicat *nat/1* sert justement à effectuer ce contrôle de type. Ces contrôles détectent des incohérences des données par rapport à la spécification du programme et facilitent donc le travail de mise au point de l'utilisateur.

L'introduction de la notion de type a un autre avantage dans la correction des programmes en Prolog standard. Une implantation concrète de Prolog contient, à côté de Prolog pur, de nombreux prédicats prédéfinis : prédicats arithmétiques (*is/2*, *</2*, *>/2*, …), prédicats extra-logiques (*read/1*, *write/1*, *name/2*, …). Les données, pour cette partie du langage, sont typées. Il existe donc des types primitifs : *integer*, *atom*, …, et certains de ces prédicats ne traitent strictement que des données d'un type primitif déterminé. Par exemple, le prédicat *name/2* exige que son premier argument soit un atome ou un nombre, le prédicat *is/2* exige comme deuxième argument une expression arithmétique. Les erreurs de type dans ce cas sont détectées pendant l'exécution du programme. Elles produisent des messages d'erreur du système, et l'exécution du programme s'arrête.

L'existence de types en Prolog permettrait encore de résoudre partiellement le problème de sécurité pour le développement de gros programmes sur la base de prédicats déjà construits. Les spécifications des types d'argument servent en effet d'interface entre les prédicats. Les mauvaises utilisations des prédicats deviendraient moins fréquentes.

Enfin, les spécifications de type des prédicats rendraient possible une génération de code exécutable optimisé, surtout en utilisant des structures déjà connues des arguments pour choisir l'algorithme d'unification spécifique correspondant.

Les tentatives de définition de type pour le langage Prolog sont donc bien justifiées. Pourtant, définir un système de types efficace, mais qui ne limite pas

les capacités du langage, et qui n'alourdit pas trop la tâche de l'utilisateur, demande de résoudre plusieurs problèmes : quelle notion de type choisir, comment typer, comment acquérir les spécifications de type et comment vérifier la cohérence entre un programme et ses spécifications de type. Les sections qui suivent présentent ces problèmes et les solutions possibles proposées dans les systèmes de types existants.

1.2 Notion de type de Prolog

Comme dans d'autres langages, le type d'un argument d'un prédicat Prolog spécifie le domaine de termes que cet argument peut prendre pendant l'exécution du programme, quand le prédicat est appelé. D'après [Cardelli 85], où un type est considéré comme un ensemble de valeurs, un type dans le langage Prolog est un ensemble de termes que ses programmes manipulent. Un terme est du type τ, s'il appartient à τ.

Pourtant, tout ensemble de termes ne peut pas être un type. Un type d'un langage doit satisfaire de plus des contraintes définies par la sémantique de ce langage.

Cette partie présente les propriétés les plus caractéristiques de la notion de type utilisée dans les systèmes de types Prolog actuels : la fermeture par généralisation d'un ensemble de termes clos et la fermeture par produit cartésien. Ensuite, la notion d'ensemble régulier choisie pour représenter les types Prolog est définie sous la forme d'équations récursives. Une propriété très recherchée des systèmes de types est le polymorphisme qui permet à un argument de prédicat ou de procédure d'avoir différents types. Les deux sortes de polymorphisme déjà traitées dans les systèmes de types Prolog : le polymorphisme paramétrique et le polymorphisme inclusif, sont aussi présentées et illustrées par des exemples.

1.2.1 Qu'est-ce qu'un type pour Prolog?

La première caractéristique d'un type Prolog provient du fait que la sémantique du langage est basée sur l'Univers de Herbrand, qui ne contient que des termes clos. La plupart des systèmes de types Prolog définissent donc les types comme des ensembles de termes clos. Pour un terme non clos, il est dit bien typé par rapport au type τ s'il peut être instancié à un terme clos de τ.

Le système de types que nous proposons admet une notion de mode, et dans ce système un type peut contenir des termes non clos. Nous avons donc choisi de ne pas présenter les travaux classiques à l'aide des notations usuelles se limitant aux termes clos. Dans notre présentation, tout terme bien typé par rapport à un type est aussi considéré comme appartenant à ce type. Par conséquent, un type dans les systèmes classiques est constitué d'un ensemble de termes clos et des termes plus généraux que (instanciables à) les éléments de cet ensemble.

La deuxième propriété importante des types Prolog : la fermeture par produit cartésien, est demandée pour l'efficacité et le déterminisme des opérations principales de l'inférence ou de la vérification de types : calcul de l'intersection de deux types, vérification de l'inclusion d'un type dans un autre, dérivation du type d'un sous-terme à partir de celui du terme ([Mishra 85]).

Un ensemble de termes E est dit **fermé par produit cartésien**, si pour tous x, y de E tels que :

x contient un n-uplet $(x_1, \ldots, x_n)$,

$y = x\,[(x_1, \ldots, x_n) / (y_1, \ldots, y_n)]$,

($x\,[x_1 / x_2]$ signifie le remplacement de x_1 dans x par x_2),

tout ensemble de termes :

$\{x\,[(x_1, \ldots, x_n) / (z_1, \ldots, z_n)] \mid z_i = x_i$, ou $z_i = y_i$, $1 \le i \le n\}$

(fermeture cartésienne de {x, y})

est inclus dans E.

Par exemple *{f(a, a), f(a, b), f(b, a), f(b, b)}* est fermé par produit cartésien, mais *{f(a, a), f(b, b)}* ne l'est pas.

La fermeture par produit cartésien exprime l'indépendance entre les composants des termes d'un type. Par exemple, $\{f(x_1, x_2) \mid x_1, x_2 \in nat\}$ est un type, mais $\{f(x, x) \mid x \in nat\}$ n'en est pas un.

Ensembles réguliers de termes

La notion d'ensemble régulier de termes est une extension de celle d'expression régulière définie classiquement en théorie des langages. Par exemple, l'ensemble des entiers naturels *nat = {zéro, s(zéro), s(s(zéro)), …}*, est un ensemble régulier. Les ensembles réguliers satisfont les deux propriétés des types présentées dans la partie précédente, et forment de plus une famille d'ensembles fermée par application de fonction, par union disjointe et par récursivité ([Mishra 85]).

Plusieurs notations équivalentes peuvent être utilisées pour présenter les ensembles réguliers. L'ensemble *nat*, par exemple, peut être donné par l'équation récursive suivante :

nat = zéro | s(nat).

Il peut aussi être défini par le prédicat unaire :

nat(zéro).
nat(s(X)) :– nat(X).

Et enfin, d'après la définition de [Mishra 84], il peut être donné par l'arbre régulier suivant :

nat = **fix** *X.(zéro + s(X)),*

où **fix** est l'opérateur utilisé pour définir des ensembles infinis, et *X* la variable de récursion.

Dans la plupart des systèmes de types et surtout dans les systèmes déclaratifs, la notation équationnelle a été choisie pour définir les types, car cette notation est proche de celle des langages typés classiques. Nous allons donc nous aussi utiliser cette notation dans tout le reste de ce document.

Soit Var - l'ensemble de variables,
Func - l'ensemble de foncteurs (symboles de fonction avec arité),
Pred - l'ensemble de prédicats (symboles de prédicat avec arité),
utilisés pour construire les programmes Prolog.

Soit τName un ensemble de symboles, différents de ceux de Var, Func, Pred, utilisés pour désigner les ensembles réguliers.

Un **ensemble régulier** est l'ensemble défini par une équation de forme :

$$\tau = C_1 \mid \ldots \mid C_m$$

où :

- τ est un élément de τName qui désigne l'ensemble à définir,
- la partie droite est une énumération de composants C_i de la forme $f(E_1, \ldots, E_n)$:
  - . f/n est un foncteur Prolog (de Func),
  - . $E_1, \ldots, E_n$ sont des noms de τName ou des énumérations de composants de la forme $f(E_1, \ldots, E_n)$,
- les foncteurs de C_i sont différents.

Par exemple : *t1 = a | b,*
t2 = f(a, b) | g(a | b),
t = a | b | f(t),
est un système d'équations qui définissent les ensembles réguliers *t1, t2, t.*

Soit E un ensemble de termes. La **fermeture par généralisation** de E est l'ensemble : gen(E) = {x | ∃ x' ∈ E, ∃ une substitution σ, σx = x'}.

L'ensemble de termes représenté par un ensemble régulier est déterminé de la façon suivante :

- [[τ]] est l'ensemble de termes désigné par τ,
- [[τ]] = [[C_1]] ∪ … ∪ [[C_m]], si τ = C_1 | … | C_m,
- [[$f(\tau_1, \ldots, \tau_1)$]] = gen {$f(c_1, \ldots, c_n)$ | c_i clos ∈ [[τ_i]], 1≤i≤n},
- Pour les équations récursives ou mutuellement récursives, les ensembles [[τ]] à définir, sont les plus petits ensembles qui satisfont toutes ces équations.
  On démontre que ces ensembles existent toujours et sont uniques pour chaque système d'équations récursives.

Les ensembles définis par les équations de l'exemple présenté sont donc les suivants :

t1 = {a, b} ∪ Var,
t2 = {f(a, b), g(a), g(b)} ∪
{g(X), f(a, X), f(X, b), f(X, Y) | X, Y ∈ Var, X ≠ Y} ∪ Var,
t = {a, b, X, f(a), f(b), f(X), f(f(a)), f(f(b)), f(f(X)), … | X ∈ Var}.

On démontre que l'ensemble de termes clos des ensembles ainsi définis est fermé par produit cartésien. Cette propriété est déduite de la condition imposant que les foncteurs des composants C_i d'une énumération doivent être distincts.

On peut voir maintenant que cette propriété permet de dériver, à partir du type d'un terme, le type de chacun de ses sous-termes d'une façon déterministe. Ainsi, si le terme *f(x, y)* est de type $\tau = f(a \mid b, c \mid d)$, on peut déduire indépendamment de la valeur de *y*, le type $\tau_1 = a \mid b$ pour *x*, et indépendamment de la valeur de *x* le type $\tau_2 = c \mid d$ pour *y*.

1.2.2 Polymorphisme

Dans les systèmes de types, le polymorphisme exprime le fait qu'un prédicat ou une procédure peut être utilisé sur différents types de données. Il est clair que c'est une propriété importante et les systèmes de types polymorphes sont donc très appréciés. Pour le langage Prolog, deux sortes de polymorphisme ont été déjà pris en compte.

a/ Le polymorphisme paramétrique

Traduit par la présence de variables dans les expressions de type d'un argument, ce polymorphisme capture l'idée qu'un prédicat fonctionne uniformément sur certaines de ses données, sans utiliser leurs structures particulières. Cela permet de ne pas préciser le type de ces éléments et de l'exprimer par une variable de type.

Exemple : Dans les prédicats suivants

length([], 0).
length([A | L], N) :– length(L, N1), N is N1+1.

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

où *length/2* détermine le nombre d'éléments d'une liste et *append/3* sert à concaténer deux listes, la structure des éléments des listes n'est précisée dans aucune des clauses, et ces prédicats peuvent être utilisés avec des listes d'entiers, des listes de listes, … L'important pour le prédicat *append/3* est que les éléments soient de même nature. Il est donc souhaitable de spécifier le type de l'élément de la liste par une variable de type *T*, et le type d'un argument qui représente une liste, dépend de cette variable *T*.

Un tel type peut être représenté par un nom contenant des paramètres qui sont utilisés dans l'équation le définissant, par exemple :

list(T) = [] | [T | list(T)].

Un type polymorphe est interprété comme une fonction sur les types. Un prédicat ayant un type d'argument polymorphe peut accepter comme argument effectif correspondant, un terme d'un type qui est une instance du type polymorphe précisé. L'utilisation de variables de type permet en plus de contraindre différents arguments à prendre leurs valeurs dans le même type, ce qui permet d'exprimer la compabilité des variables dans un programme. Par exemple, la spécification :

append(list(T), list(T), list(T))

exprime bien le fait que les éléments des listes représentées par les arguments de *append/3* doivent être d'un même type.

b/ Le polymorphisme inclusif

La définition de la relation d'inclusion entre deux types permet à un terme d'appartenir en même temps à plusieurs types, et au système de types de reconnaître un ensemble plus large de clauses bien typées. Ce traitement est différent des systèmes de types issus des travaux sur le langage ML où deux types doivent nécessairement être disjoints.

Exemple :

Soit le prédicat *dérivée(E, V, D)*, qui calcule la dérivée *D* d'une expression arithmétique *E* par rapport à une variable *V* (pouvant être *x* ou *y*) :

```
dérivée(N, V, 0) :– number(N).        % number/1 est prédéfini
dérivée(V, V, 1).
dérivée(E, V, 0) :– variable(E), différent(E, V).
dérivée(E1 + E2, V, D1 + D2) :–  dérivée(E1, V, D1),
                                 dérivée(E1, V, D1).

variable(x).
variable(y).

différent(x, y).                      % Les constantes x, y représentent
différent(y, x).                      % les variables formelles
```

Il est clair que les nombres et les variables formelles *x*, *y* sont des expressions et les types utilisés devraient refléter ces inclusions, par exemple en étant définis par :

var = x | y,
exp = number | var | exp + exp.

Et le système devrait ensuite accepter comme bien typée la clause :

dérivée(V, V, 1).

en autorisant les appels comme :– *dérivée(x, x, 1)*, où *x* est considéré comme une variable formelle dans le deuxième argument, et une expression dans le premier.

Nous verrons cependant que l'inclusion de types ne peut pas être correctement traitée sans la notion de mode. Nous étudions ce problème au paragraphe 1.4.

1.3 Les approches de typage

Les différents rôles de la notion de type dans le langage, ainsi que les multiples méthodes pour acquérir les spécifications de types, ont amené plusieurs approches de typage pour Prolog. Cette partie propose une classification générale des systèmes de types actuels afin de présenter ces approches et les problèmes communs à tout système de types Prolog. Deux systèmes de types les plus représentatifs de deux approches différentes : déclarative et inférentielle, sont présentés, ce qui permet de détailler leurs méthodes de typage et de donner une comparaison générale entre ces deux approches.

1.3.1 Erreur de type et Echec

Dans les systèmes de types, un programme est dit “bien typé” si les appels de prédicats pendant l’exécution sont en conformité avec les types spécifiés. Un programme est dit “mal typé” s’il n’est pas bien typé. Il existe deux points de vue différents sur la corrélation entre les types et les solutions des programmes.

Dans les systèmes de [Mycroft 84], [Dietrich 88], [Lakshman 91], le fait qu’un programme soit bien typé ou non ne dépend pas de sa réussite ou son échec. Il existe dans cette approche une notion d“erreur de type” qui sert à marquer une incohérence de type dans les appels de prédicats pendant l’exécution. Il en résulte donc qu“un programme bien typé ne produit pas d’erreur de type à l’exécution”.

Certains, ([Xu 88]), ont reproché à Mycroft-O’Keefe de ne pas avoir redéfini la sémantique non typée du langage et d’utiliser donc les types comme des propriétés extra-logiques des programmes. Pourtant ce n’est qu’un aspect formel qui n’affecte pas la valeur pratique du système dans la détection des incohérences de types.

En reconstruisant le système de [Mycroft 84] pour prendre en considération les types dans la définition de la sémantique du langage, Lakshman a prouvé la consistance sémantique de ce système. Dans le travail de Lakshman, un nouveau langage : “Prolog typé”, est donc défini avec la sémantique non typée de Prolog reformulée pour tenir compte en plus des clauses définissant les prédicats, et les types déclarés. Un programme mal typé est maintenant rejeté par le système parce qu’il ne fait sûrement pas partie du langage. Par exemple, le programme :

```
append([], L, L).
append([A | L], M, [A | N])  :–  append(L, M, N).
:–  append([], a, X).
```

ne sera pas accepté, car *a* dans la question n'est pas du type déclaré. Cette approche, séduisante par ses liens avec la logique typée, n'a cependant pas encore permis de traiter la notion d'inclusion entre les types.

Dans [Mishra 84], [Kanamori 85], [Kluzniak 87], [Zobel 87], [Hanus 89], qui suivent un autre point de vue, une incohérence de type conduit nécessairement à l'échec du programme. "Un programme mal typé échoue donc toujours".

Ce principe de typage est réalisé dans le système de Hanus par la reconstruction du mécanisme d'exécution des programme : l'unification standard a été remplacée par une unification typée. Ainsi, dans le programme présenté ci-dessus de *append/3*, le but :– *append([], a, X)* ne provoque pas d'erreur de type, mais les unifications de ce but avec les deux têtes des clauses échouent quand il est détecté que *a* n'est pas de type *list(T)*.

Dans [Mishra 84], [Kanamori 85], [Kluzniak 87], [Zobel 87], les types des prédicats sont inférés à partir du programme et sont des sur-ensembles de leurs solutions. Les types sont donc une notion syntaxique utilisée pour décrire les propriétés sémantiques du langage.

1.3.2 Typer un programme

Typer un programme Prolog signifie déterminer les types de tous ses objets : les prédicats, les foncteurs et les variables.

- Nous supposons les notations suivantes pour la suite de notre exposé :

τVar	l'ensemble de variables de type,
τFunc	l'ensemble de foncteurs de type,
T	variable de type (de τVar), appelée *τ-variable*,
τ	type,
a, b, c, …	atomes,
x, y, z, …	termes Prolog,
X, Y, Z, …	variable Prolog (de Var),
f	foncteur Prolog (de Func),
p	prédicat Prolog (de Pred).

- Les définitions ci-dessous sont essentielles pour présenter les systèmes de types.

Le type d'un prédicat, noté $p^{\tau_1,\ldots,\tau_n}$, où p/n est le nom/arité du prédicat, est l'assertion que les arguments du prédicat sont respectivement de type τ_1, …, τ_n, à son appel et à son retour. Le type d'un prédicat est dit **correct** s'il est vrai pendant toute l'exécution du programme.

Le type d'une variable, noté X : τ, où X est une occurrence de variable dans le programme, et appelé **assignation** de X, est l'assertion que le terme représenté par X, est de type τ. Une assignation est dite **consistante** si elle est vraie pendant toute l'exécution du programme.

Un programme est dit **typé** si tous ses prédicats ont une spécification de type. Un propramme typé est dit **bien typé** si les types de tous ses prédicats sont corrects.

1.3.3 Systèmes déclaratifs et inférentiels

On peut aussi classifier les systèmes de types suivant leur façon d'acquérir les spécifications de type dans les programmes Prolog.

- Dans les systèmes déclaratifs : les types des objets du programme sont définis par l'utilisateur.
- Dans les systèmes inférentiels : les types sont inférés automatiquement à partir du programme.

Pour les systèmes déclaratifs : [Mycroft 84], [Dietrich 88], [Hanus 89], les déclarations de types, données par l'utilisateur, sont des contraintes sur le type d'argument des prédicats. Les systèmes disposent de règles permettant de vérifier statiquement si un programme est bien typé.

En pratique, l'utilisateur ne spécifie dans un programme que le type des prédicats. A partir de ces déclarations et des règles de vérification du bon typage, qui expriment une relation entre les types des prédicats et ceux des variables dans une clause, le système dérive pour chaque variable son assignation. Si pour chaque variable est trouvée une assignation unique, le programme est bien typé. Dans ce cas, ces assignations sont consistantes.

Ainsi du point de vue de l'utilisateur, un tel système ressemble à celui d'un langage typé classique, où les déclarations permettent d'exprimer l'intention de l'utilisateur dans la définition des prédicats : leur spécification. Toutes les incohérences de types seront détectées par le compilateur. Pourtant ces

déclarations sont à la charge de l'utilisateur. Une solution à ce problème a été réalisé dans [Lakshman 91], où l'utilisateur ne spécifie que les types de certains prédicats clé, et le système est capable de reconstruire les types des autres prédicats d'après les types déclarés.

Dans les systèmes [Mishra 84], [Kanamori 85], [Kluzniak 87], [Zobel 87], par contre, les spécifications des prédicats sont calculées par le système à partir du programme, où à partir du programme et des types déjà définis comme dans le système de Kanamori et Horiuchi. Le principe de l'inférence des types d'un programme consiste à effectuer une analyse statique de toutes ses clauses, en collectant, pour chaque prédicat, les structures possibles que chacun de ses arguments peut avoir chaque fois que ce prédicat réussit. Les conditions nécessaires au succès des prédicats sont considérées pendant le calcul comme des contraintes de types. Le type inféré pour chaque argument de prédicat est le plus petit type qui contient tous les termes des structures trouvées. Ce type inclut donc l'ensemble des solutions du prédicat, défini par son modèle de Herbrand. Un programme est mal typé si pour un prédicat est inféré un type vide, ou s'il est impossible d'inférer les types qui satisfont les contraintes de types.

1.3.4 Un système déclaratif

Cette partie présente le système [Mycroft 84], qui, étant le premier système de types polymorphes, a abouti à plusieurs systèmes de types pour Prolog : [Dietrich 88] en a fait une extension avec traitement des inclusions des types, [Lakshman 91] l'a reconstruit avec reformulation de la sémantique du langage.

Déclaration d'un programme

Un programme avec déclarations doit contenir les parties suivantes :
- les définitions de types,
- les déclarations de prédicats,
- les clauses définissant les prédicats.

Le langage de type utilisé dans ce système est une version du langage équationnel des ensembles réguliers présenté dans le paragraphe 1.2.1 avec la participation des variables de type.

Une définition de type est de la forme :

type $t(T_1, \ldots, T_k) = C_1 \mid \ldots \mid C_m$

où :

- t/k est un foncteur de type (de τFunc),
  $T_1, \ldots, T_k$ sont des τ-variables,
  C_i est de la forme $f(\tau_1, \ldots, \tau_n)$, f/n est un foncteur Prolog (de Func), $\tau_1, \ldots, \tau_n$ sont des noms de type : τ-variables, ou foncteurs de type appliqués sur un n-uplet de noms de type.
- Les foncteurs de C_i sont différents,
- Toutes les τ-variables de la partie droite doivent être présentes dans la partie gauche.

Dans chaque définition de type, le nom à gauche désigne le type, qui est défini par l'expression linéaire des foncteurs Prolog et des constructeurs de type à droite.

Une déclaration de prédicat est de la forme :

pred $p(\tau_1, \ldots, \tau_n)$

où p/n est un prédicat de Pred,
$\tau_1, \ldots, \tau_n$ sont des noms de type,
et qui signifie que τ_i est le type de l'argument i de p/n.

Exemple :

```
type    list(T) = [] | [T | list(T)].
type    t  = a | b.
pred    append(list(T), list(T), list(T)).
pred    rev(list(T), list(T)).
pred    exlist(list(t)).
pred    ex(list(t), list(t)).

append([], L, L).                                   (1)
append([A | L], M, [A | N])  :–  append(L, M, N).   (2)
rev([], []).                                        (3)
rev([A | L], R)  :–  rev(L, R1), append(R1, A, R).  (4)
exlist([a, b, a]).                                  (5)
ex(Li, Lf)  :– exlist(Li), append([a], Lf, Li).     (6)
```

est un programme avec toutes les déclarations nécessaires de ses objets.

Vérificateur de type

La vérification d'un programme consiste à vérifier le bon typage de chaque clause d'après les règles établies dans le système. Cette vérification est locale : chaque clause est traitée indépendamment des autres.

Dans chaque clause, à partir de ces règles et des types déclarés des prédicats et des foncteurs, le système dérive pour chaque occurrence de variable, les types qu'elle devrait avoir : son assignation consistante (cf. 1.3.2). Le type de la tête de clause doit être aussi général que la déclaration du prédicat. Le type des littéraux du corps de la clause doit être une instance de leur déclaration.

Si pour une variable sont dérivés deux types, ils doivent être unifiables. Leur instance la plus générale, sera son assignation finale. L'instanciation de τ-variables, produite par cette unification, ne peut donc être faite que si ces τ-variables n'apparaissent que dans le corps de la clause.

Si le résultat de l'analyse donne à chaque variable un type unique, la clause est bien typée.

Exemple :

La vérification de la clause (2) par rapport à la déclaration de *append/3* déduit les assignations :

- dans la tête, $A : T_1$, $L : list(T_1)$, $M : list(T_1)$, $N : list(T_1)$,
- dans le sous-but, $L : list(T_2)$, $M : list(T_2)$, $N : list(T_2)$.

Les variables *L, M, N* se voient assignées deux types : $list(T_1)$ et $list(T_2)$ qui sont unifiables, et la clause est donc bien typée.

Dans la clause (4), l'erreur de type du deuxième argument du sous-but *append/3* est détectée, car la vérification de cette clause par rapport à la déclaration de *rev/2* assigne à *A* deux types T_1 et $list(T_1)$ qui ne sont pas unifiables.

Remarques sur les systèmes déclaratifs

Il est clair que les déclarations de types peuvent exprimer très précisément l'intention du programmeur et donc la spécification du programme. Un très grand nombre d'erreurs de concordance de type des arguments, de nombre et d'ordre des paramètres, de fautes de frappes, … peuvent être détectés statiquement par le vérificateur de type.

De plus, l'algorithme de vérification est en général simple, son implantation, dans [Mycroft 84] par exemple, n'est pas coûteuse, car l'analyse de types est effectuée localement sur chaque clause.

L'inconvénient reproché aux systèmes déclaratifs est que le programmeur voit son travail alourdi par la charge de déclarer les type des prédicats.

1.3.5 Un système inférentiel

Le système de types exposé dans cette partie engendre, pour un programme donné, les types monomorphes, représentés par des ensembles réguliers. Les contraintes de type sont exprimées sous forme d'inclusion du type d'un sous-but dans le type inféré du prédicat de ce sous-but. Cet algorithme a été réalisé par [Mishra 84].

L'algorithme consiste à construire pour chaque argument d'un prédicat son type inféré, qui est la fermeture cartésienne de la somme de tous les termes qui représentent cet argument dans les têtes de toutes les clauses de ce prédicat.

Initialement chaque variable est remplacée par un type inconnu.

Dans l'étape suivante, le système déduit pour chaque sous-but de toutes les clauses, les inéquations qui expriment l'inclusion du type de ses arguments dans le type inféré correspondant du prédicat d'appel.

La dernière étape est la résolution du système d'inéquations obtenues pour trouver la solution des types inconnus.

Exemple : Pour le programme

sub(suc(zero), zero).
sub(suc(suc(X)), suc(Y)) :– sub(suc(X), Y).

- Le type inféré pour le premier argument de *sub/2* est la fermeture cartésienne de *suc(zero)* | *suc(suc(T_1))*, donc : *suc(zero* | *suc(T_1))*,

  Le type inféré pour le deuxième argument de *sub/2* est la fermeture cartésienne de *zero* | *suc(T_2)*, donc : *zero* | *suc(T_2)*.

- Les contraintes du programme (obtenues de la deuxième clause) sont :
  $suc(T_1) \subseteq suc(zero \mid suc(T_1)) \Rightarrow T_1 \subseteq zero \mid suc(T_1)$,
  $T_2 \subseteq zero \mid suc(T_2)$.

- Les solutions des inéquations sont : $T_1 = t$, $T_2 = t$,
  où *t* est le type défini par l'équation : *t = zero | suc(t)*.

- Le type inféré pour ce prédicat est donc *sub(st, t)*, où :
  t = zero | suc(t), st = suc(t).

Remarques sur les systèmes inférentiels

Le plus grand avantage des systèmes inférentiels est que le travail de l'utilisateur reste inchangé. Le système peut calculer automatiquement les types des prédicats et vérifier en même temps le bon typage du programme : une incohérence de type conduit nécessairement à un type inféré vide, ou à une contrainte d'inclusion de types qui ne peut pas être satisfaite.

Parfois, le type inféré est compliqué et non pertinent. Par exemple, le type inféré (par un système polymorphe) pour le prédicat *append/3* est :

append(list(T1), T2, list(T1, T2)),

où *list(T) = [] | [T | list(T)]*,
list(T1, T2) = T2 | [T1 | list(T1, T2)],
(listes d'éléments de type *T1* se terminant par un élément de type *T2*).

La complexité de ce type se propage aux types des prédicats qui utilisent *append/3*, ce qui aboutit rapidement à des résultats illisibles ([Pyo 89]).

Enfin, le calcul pour inférer les types d'un programme doit être effectué globalement sur tout le programme, ce qui est plus coûteux que le cas des systèmes déclaratifs.

1.4 Les modes et leur relation avec les types

1.4.1 Intérêt des modes

Les modes sont utilisés pour exprimer une autre spécification des prédicats : le degré d'instanciation de chaque argument d'un prédicat à son appel et son retour.

Les modes les plus courants, utilisés dans [Warren 77], [Mellish 81], [Debray 85] sont :

c l'argument est toujours instancié,
f l'argument est toujours non-instancié (variable),
d l'argument est parfois instancié, parfois non.

Les modes structurés, introduits par [VanRoy 86], qui sont de la forme $f(\mu_1, \ldots, \mu_n)$, où f est un foncteur n-aire, et $\mu_1, \ldots, \mu_n$ des modes de base (c, f, d), peuvent être considérés comme une intégration de la notion de type dans les modes. Ces modes permettent de spécifier le degré d'instanciation des sous-termes d'un argument.

Les modes sont utilisés principalement pour rendre l'exécution des programmes plus efficace :

- La connaissance des modes des arguments d'une unification permet de remplacer l'unification générale par une unification spécifique : affectation, substitution, …, moins coûteuse. Cette application avec les modes définis dans [Warren 77], a été réalisée dans la plupart des compilateurs du marché.

- Une utilisation importante des modes consiste à une meilleure gestion de l'espace mémoire, car la distinction des variables des autres termes, plus ou moins instanciés, permet, dans certains cas, de récupérer facilement l'espace, occupé par un argument.

- L'information sur le degré d'instanciation des arguments en appel, peut être utilisée pour réordonner les sous-buts dans une clause, afin d'éviter les calculs infinis ([Oudot 87]).

- Les modes, comme spécifications des prédicats, peuvent aussi être utilisés pour vérifier la correction des programmes. Par exemple, le programme :

```
rev([], []).
rev([A | L], R)  :–  rev(R1, L), append(R1, [A], R).
:–  rev(L, [a,b]).
```

boucle avec la stratégie de résolution en profondeur d'abord, car cette définition de *rev/2* doit être utilisé pour résoudre des questions où le premier argument est clos. Une spécification de modes *rev(c, d)*, connue par le compilateur ou l'interprète, est utile pour détecter l'incohérence dans la question.

Dans [Dietrich 88], [Bruynooghe 82], [Somogyi 87], les modes sont utilisés avec les types pour vérifier la cohérence des programmes.

Comme pour les types, deux approches de la spécification des modes sont possibles :

- l'approche déclarative, où les modes sont déclarés explicitement par le programmeur;
- l'approche inférentielle, où les modes sont calculés automatiquement à partir du programme.

La première approche exige une vérification qui teste si toutes les déclarations de modes données par l'utilisateur sont correctes sur l'ensemble de clauses des prédicats.

L'inférence automatique des modes consiste à analyser statiquement et globalement tout le programme, pour calculer, pour chaque argument de prédicat, le mode qui englobe tous les termes qu'il représente dans tous les appels.

Il existe pour les systèmes de modes, ainsi que pour les systèmes de types avec sous-types, un problème qui peut conduire à des résultats inconsistants. C'est le problème d'***aliasing***, réalisé par des occurrences multiples de variables et qui peut avoir comme conséquence des instanciations incontrôlables des variables logiques. Ces dernières peuvent changer brusquement la propriété de l'argument qui représente cette variable.

Exemple :

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

Soit *p(f, f)* le mode d'appel du prédicat *p/2*. Une propagation non soigneuse de ce mode dans la clause *p/2* pourrait donner à *Y* le mode de retour *f*, qui n'est pas consistante, car en réalité, après le succès de *r(X)*, les deux variables *X* et *Y* sont instanciées à *a*, et le mode de *Y* devrait donc être *c*.

Des traitements spéciaux ont été réalisés dans les systèmes de modes pour prévenir les erreurs d'aliasing.

1.4.2 Les modes dans un système de types avec sous-types

Exploiter la relation d'inclusion des types a pour but d'augmenter la précision de vérification, car dans un système de types où cette relation n'est pas prise en considération, il peut y avoir des clauses qui sont cohérentes par rapport aux déclarations mais qui ne passent pas la vérification de types du système.

Revenons à l'exemple de *dérivée/3* (cf. 1.2.2.b) :

```
dérivée(N, V, 0) :– number(N).
dérivée(V, V, 1).
dérivée(E, V, 0) :– variable(E), différent(E, V).
dérivée(E1 + E2, V, D1 + D2) :–   dérivée(E1, V, D1),
                                  dérivée(E1, V, D1).
variable(x).
variable(y).
différent(x, y).
différent(y, x).
```

Il est clair que ce prédicat doit être utilisé avec une expression en premier argument et une variable formelle en deuxième. Cette spécification peut être donnée dans le système de [Mycroft 84] par des déclarations suivantes :

```
type    var = x | y.
type    exp = number | x | y | exp+exp.
pred    dérivée(exp, var, exp).
```

Pourtant la clause :

```
dérivée(V, V, 1).
```

est mal typée dans ce système, car les types déduits pour la variable *V* sont *exp* et *var*, qui ne sont pas unifiables (cf. 1.3.4.Vérificateur de type). La remarque que les termes du type *var* sont aussi du type *exp*, donc *var* est inclus dans *exp*, conduit à l'idée d'accepter cette clause, en autorisant deux occurrences d'une même variable d'avoir des types différents, si l'un est inclus dans l'autre.

Cependant une simple acceptation de la clause peut conduire à des erreurs de type pendant l'exécution. La question :– *dérivée(x+y, X, 1),* par exemple,

retourne la réponse *X = x+y*, qui n'est pas du type *var*. Cette erreur provient d'une mauvaise instanciation des arguments de la question : le deuxième argument est non instancié (utilisé comme paramètre *résultat*), le premier est instancié (utilisé comme paramètre *donnée*), et donc le transfert de données est effectué d'un type plus grand à un type plus petit. Dans le programme de *dérivée/2*, cette propriété des termes n'est pas spécifiée pour des raisons d'efficacité : La version correcte de la clause en question devrait être *dérivée*(*V, V, 1*) :– *variable*(*V*).

Pourtant une spécification explicite des modes d'argument du prédicat, qui serait vérifiée à la compilation, peut aussi rendre le programme de l'exemple correct et non moins efficace.

Cette remarque met donc en évidence une relation étroite entre le flot de données, dérivé des modes des arguments dans une clause, et la correction des types quand il y a des sous-types. Cette information sur le sens du transfert de données permet de vérifier les inclusions de type.

Les inclusions entre types ont été traitées dans [Dietrich 88], par une extension du système de Mycroft et O'Keefe [Mycroft 84]. Un type, considéré comme ensemble de termes, peut être inclus dans un autre type. Un prédicat peut être appelé avec des arguments d'un type inclus dans son type déclaré.

La présentation des types, le mécanisme du polymorphisme, le principe de typage sont sensiblement similaires à ceux de Mycroft-O'Keefe.

Une autre information : le flot de donnée, fournie par le système de modes, est utilisée, pour exprimer les conditions de bon typage des programmes.

Modes. Flot de données

L'ensemble de **modes** utilisés dans le système [Dietrich 88] est M = {*i, o, io*}, où :

- *i*(input) l'argument n'est pas plus instancié après l'exécution du prédicat;
  (**Remarque** : l'argument n'est pas forcément clos)
- *o*(output) l'argument est toujours une variable quand le prédicat est appelé;
- *io*(in_out) rien n'est spécifié sur le degré d'instanciation de l'argument

Le mode d'un prédicat p/n est de forme p(μ_1, …, μ_n), où μ_1, …, μ_n sont des éléments de M. Par exemple, *dérivée(o, i, io), élément(o, i), append(i, i, o).*

Le **mode d'une variable** dans une clause est défini de la façon suivante :

- Une occurrence de variable est du mode *i*, si elle appartient à une donnée de la clause :
- soit à un argument *i* de la tête (une donnée d'appel)
- soit à un argument *o* d'un littéral du corps (un résultat d'un sous-but)

- Une occurrence de variable est du mode *o*, si elle appartient à un résultat de la clause :
- soit à un argument *o* de la tête
- soit à un argument *i* d'un littéral du corps

- Dans tous les autres cas, la variable est en mode *io*.

Le **flot de données** est donc établi entre deux occurrences d'une même variable, et dirigé de la position *i* ou *io* à la position *o* ou *io*.

Il doit exister, dans chaque clause, un flot de données de l'occurrence O_1 à l'occurrence O_2 d'une même variable X. Si O_1 est en mode *i* ou *io*, et O_2 est en mode *io* ou *o*, le type assigné à O_1 doit être inclus dans le type assigné à O_2.

Exemple :

- Avec le mode *dérivée(io, i, io)*, la clause *dérivée(X, X, 1)* est bien typée. On a *X : exp* en mode *io*, *X : var* en mode *i*, et *var* est inclus dans *exp*.

- Avec le mode *dérivée(i, io, io)*, la clause *dérivée(X, X, 1)* est mal typée. On a *X : exp* en mode *i*, *X : var* en mode *io*, le flot de données est donc dirigé de la première occurrence de X vers sa deuxième occurrence, mais *exp* n'est pas inclus dans *var*.

Remarques sur le système de Dietrich

Ce travail a ainsi réussi à introduire dans le système de types Prolog le polymorphisme inclusif. Le système est capable de filtrer plus précisément les clauses cohérentes par rapport aux déclarations de types. L'algorithme de vérification reste simple, car l'analyse est effectuée localement sur chaque clause.

Il est pourtant à noter sur ce système qu'il doit comprendre une vérification sur les déclarations de mode pour assurer la sécurité du système.

1.5 Notre approche dans les système de types

Le but étant de concevoir un système de types polymorphes pour Prolog avec l'inclusion des types, nous présentons dans cette partie une discussion qui conduit à notre choix de solutions et méthodes aux problèmes posés.

1.5.1 Système déclaratif

Sans aller aussi loin que [Naish 87] : "Spécification = Programme + Types", nous pensons que l'approche déclarative, qui permet d'exprimer l'intention du programmeur sur les types des arguments, est plus intéressante pour la spécification des programmes. L'inférence de types, qui calcule le sur-ensemble de solutions d'un prédicat, donne parfois des résultats non pertinents, comme le montre l'exemple de *append/3* dans le paragraphe 1.3.5, ou des résultats avec une grande imprécision. La recherche de types inférés les plus précis peut aboutir à des notations de moins en moins compréhensibles pour l'utilisateur [Pyo 89].

Ensuite, l'approche déclarative avec la notion d'erreur de type permet de mieux localiser les erreurs, de distinguer les erreurs typographiques des erreurs de conception, ce qui facilite plus la mise au point des programmes.

Du point de vue de l'utilisateur, un système inférentiel a plus d'avantages qu'un système déclaratif, car il n'exige aucune déclaration du programme. Dans un système déclaratif, le programmeur est tenu de définir les types et déclarer les prédicats. Cependant, un bon programmeur connaît la structure de ses données et les décrit souvent à l'aide de commentaires. Il ne lui est alors pas difficile de formaliser ces commentaires.

En ce qui concerne la complexité, une inférence de types doit être effectuée globalement sur l'ensemble des clauses du programme. La vérification de la cohérence entre le programme et les déclarations est, par contre, effectuée localement sur chacune des clauses, ce qui est évidemment moins coûteux.

1.5.2 Intégration de la notion de mode dans les types

Le plus grand problème à résoudre dans un système avec sous-types est l'incorrection des types, produite par une acceptation naïve des inclusions des types pour les occurrences multiples d'une même variable. L'un des cas a été

déjà présenté dans l'exemple *dérivée/3* de Dietrich (cf. 1.4.2). Le problème est résolu dans ce système par l'introduction d'une notion de mode. Cependant un système de calcul de modes doit aussi prendre en compte des traitements spéciaux pour ne pas avoir de résultats inconsistants dus au phénomène d'alias des variables logiques (cf. 1.4.1).

La connaissance des modes est nécessaire pour garantir la correction des types quand l'inclusion est prise en considération. Pourtant nous choisissons une autre approche à ce problème en remarquant que le problème d'aliasing dont l'origine provient toujours des variables logiques, existe aussi dans un système de types. En faisant la distinction entre les types qui peuvent contenir des variables non instanciées et ceux qui n'en contiennent pas, nous ne considérons plus les modes comme une notion séparée, mais une partie intégrée de la notion de type. La relation d'ordre entre les types n'est pas une simple inclusion, mais une propriété sur le degré d'instanciation des termes est ajoutée. Les exemples présentés ci-dessous illustrent notre proposition.

Exemple 1 :

type $t = a.$
type $t_1 = f(t).$
type $t_2 = b \mid f(t).$
pred $p(t_1).$
pred $q(t_2).$
pred $r(t_1).$

p(X) :– q(X), r(X).
q(b).
q(f(_)).

Dans la clause de *p/1*, le seul type possible pour X est t_1, qui n'est pas consistant (cf. 1.3.2) : l'acceptation de cette clause conduit à des erreurs de type pendant l'exécution. En effet, la résolution du but *:– p(Y),* qui est bien du type d'appel de *p/1* car Y est variable, donc du type t_1, conduit à l'instanciation de Y à *b* après l'effacement de *q(Y).* Le terme que Y représente n'est plus du type t_1, or Y est l'argument de *r/1*, d'où l'incohérence de type à l'exécution.

Dans le système [Dietrich 88], l'acceptation de cette clause dépend du flot de données spécifié. Si la déclaration :

mode *p(o), q(o), r(i)*

est donnée, il y a dans la clause de *p/1* un flot de données de l'occurrence de X

dans *q/1* à celle dans la tête de la clause, ce qui implique que le type de *X* dans *q/1* doit être inclus dans le type de *X* dans la tête. La clause est donc mal formée et sera refusée.
Certes, avec la déclaration :

mode *p(i), q(i), r(i)*

cetta clause sera acceptée, mais cette fois c'est le but :– *p(Y)* qui est mal formé et donc refusé.

Nous remarquons que si *p/1* est appelé avec l'argument de forme *f(Z)*, c.à.d. un terme bien typé par t_1 mais qui n'est pas une variable libre, cette erreur ne se produit plus, car avec toute substitution de *q/1* qui l'instancie à un terme de t_2, son instance obtenue est toujours dans t_1.

Cette remarque nous a amenés à introduire la définition de type suivante :

type $t_1 = + f(a)$.

où "+" signifie qu'un terme bien typé par t_1 doit être de la forme *f(x)* à l'exclusion d'une variable libre. Avec cette définition de type, l'assignation $X : t_1$ trouvée dans la clause de *p/1* est bien consistante.

t_1 est appelé dans notre système un type restreint, car il ne contient pas l'ensemble de variables Var.

De telles définitions de type représentent une intégration des modes dans les types : l'information sur le degré d'instanciation des termes est directement contenue dans les types. On remarque que pour ces définitions de types, la fermeture par généralisation n'est plus vraie.

Exemple 2 : Variables multiples dans un appel de prédicat

type $t_1 = a \mid b$.
type $t_2 = a$.
pred $p(t_1, t_2)$.
pred $q(t_1)$.
pred $r(t_2)$.

p(X, Y) :– *q(X), r(Y).* *(1)*
q(b). *(2)*

L'analyse de *(1)* donne les seules assignations suivantes aux variables *X* et *Y* :

$p(X, Y)$:– $q(X)$, $r(Y)$.

$X:t_1$ $X:t_1$

$Y:t_2$ $Y:t_2$

qui ne sont pas consistantes. En effet, dans la résolution de :– $p(Z, Z)$, les deux variables X, Y sont liées et l'instanciation à b de X après l'effacement de $q(X)$ entraîne aussi celle de Y, qui n'est donc plus du type t_2. Dans ce cas, l'exécution de $q(X)$ pouvait instancier Y (par l'intermédiaire de X), à un terme de t_1 qui n'appartient pas à t_2.

On remarque que t_2 est inclus dans t_1, et que si le deuxième argument de l'appel à *p/2* est un terme qui reste toujours dans t_2 après toute substitution dans t_1, cette incorrection n'a plus lieu. Un terme ayant cette propriété est dit dans notre système **suffisamment instancié dans *t2* pour *t1***.

Cette propriété serait satisfaite si on définissait dans cet exemple :

type $t_2 = +a.$

Les appels de *p/2* se restreindraient alors à la forme :– $p(X, a)$ et l'inconsistance n'aurait plus lieu.

Exemple 3 : Variables multiples dans une tête de clause

type $t_1 = a.$
type $t_2 = a \mid b.$

pred $p(t_1, t_2).$
pred $q(t_1, t_2).$
pred $r(t_2).$
pred $s(t_1).$

$p(X, Y)$:– $q(X, Y)$, $r(Y)$, $s(X)$. *(1)*
$q(X, X)$. *(2)*
$q(a, b)$. *(3)*
$r(b)$. *(4)*

Les solutions de q sont bien de leur type déclaré. Mais l'alias, réalisé par la répétition de X dans *(2)*, conduit la résolution de :– $p(X, Y)$ à l'instanciation

à *b* de *X* à la suite de celle de Y, après l'exécution de *r(Y)*. Dans cet exemple, l'erreur vient aussi du degré d'instanciation des termes. Etant variable, *X* n'est pas dans ce cas suffisamment instanciée dans t_1 pour t_2.

Cette propriété des termes d'être suffisament instanciés dans un type pour un autre, a été choisie dans notre système comme condition pour déterminer les assignations inconsistantes. Elle garantit qu'une variable est toujours instanciée à un terme correspondant à son type, même si elle est liée à une autre variable. Autrement dit, une variable de clause représente toujours un terme du type qui est lui assigné, ce type est donc consistant. La relation d'ordre établie dans notre système entre les types devient la relation "**plus modé**" : Le type τ_1 est plus modé que le type τ_2, si τ_1 est inclus dans τ_2 et tous les termes de τ_1 sont suffisamment instanciés dans τ_1 pour τ_2.

1.5.3 Conclusion

Ainsi, notre système de types est un système déclaratif acceptant deux formes de polymorphisme : paramétrique et inclusif. Les modes sont pris en compte pour assurer la consistance des inclusions entre les types. Pourtant la notion de mode est intégrée dans la notion de type par l'introduction des types restreints qui ne sont pas fermés par généralisation de variables. La relation d'ordre entre les types est la relation "plus modé" qui est l'inclusion plus la propriété d'instanciation suffisante des termes.

Le reste de ce rapport de thèse présente la réalisation concrète de ces nouvelles notions et leur utilisation pour la validité du système.

Chapitre 2

PRESENTATION DU SYSTEME DE TYPES REALISE

2.1 Introduction

Le travail principal réalisé dans cette thèse est l'étude théorique et l'implantation du prototype d'un système de types suivant les objectifs que nous avons présenté dans le chapitre précédent (cf. 1.5.3). Les caractéristiques essentielles de ce système sont les suivantes.

• C'est un système déclaratif : le programmeur fournit les déclarations du programme et le système vérifie leur cohérence.

• Un type est un ensemble de termes non nécessairement clos, basé sur un ensemble de termes clos. Il est possible de définir un degré d'instanciation pour tous les termes d'un type.

• Deux formes de polymorphisme sont acceptés : le polymorphisme paramétrique et l'inclusion.

• Le profil d'un prédicat est la donnée du type de ses arguments avant et après l'exécution de ses appels : le profil d'appel est une précondition requise au moment de l'effacement du but, tandis que le profil de retour est une propriété garantie par le système, sous réserve que le but réussise.

• Les prédicats d'ordre supérieurs *call, setof, assert, not, …*, sont pris en compte, mais sous une forme restreinte.

Le traitement d'un programme par le système comprend :
- L'analyse des déclarations indépendamment du programme et leur enregistrement dans une base interne.
- La vérification, appelée par une commande de l'utilisateur, des types de toutes les clauses du programme par rapport à ces déclarations.

Le principe de la vérification de types est le suivant :
- La vérification de chaque clause est locale : le résultat de la vérification d'une clause n'affecte pas l'analyse des autres.
- Le bon typage d'un programme est le bon typage de toutes ses clauses.

Ce chapitre expose le système du point de vue de l'utilisateur. Dans la première partie, nous donnons la définition syntaxique des déclarations et la description de leur sémantique, ce qui permet de présenter notre notion de type. Ensuite nous introduisons une autre définition centrale du système : La relation d'ordre "plus modé" entre les types, qui permet d'exprimer les conditions du bon typage. Enfin, l'algorithme de la vérification de types d'une clause et les exemples permettent de présenter les conditions de bon typage du système.

2.2 Syntaxe du langage de type

Pour permettre aux utilisateurs de définir les types et déclarer les profils de prédicat, le système propose un langage de type qui contient les déclarations suivantes :

<déclaration de type> ::= **type** <déf_type> {, <déf_type>} '.'

<déf_type> ::= <nom_param> '=' <énumération>
<nom_param> ::= <atom> | <atom>'('<variable>{, <variable>}')'
<énumération> ::= <énum_simple> | <mode><énum_simple>

<mode> ::= '–' | '+' | '*' | '#'
<énum_simple> ::= <expression> | <énum_simple> '|' <expression>

<expression> ::= <exp_simple> | '{'<exp_simple>'}'
<exp_simple> ::= <variable> | <atom> |
<atom>'('<énumération>{,<énumération>}')'

<déclaration de prédicat> ::= **profil** <profil> {, <profil>} '.'

<profil> ::= <atom> |
<atom>'('<type_argument>{, <type_argument>}')'
<type_argument> ::= <énumération>'→'<énumération> |
<énumération> | '→' <énumération>

2.3 Sémantique des définitions de type

Ainsi, un type, désigné par un nom, est défini par une énumération des expressions, modée (par '–', '#', '+', '∗'), ou sans mode. Chaque type est l'ensemble de termes spécifié par son énumération. Un terme x est dit du type τ si x appartient à l'ensemble de termes de τ.

Définition :
Une énumération sans mode ou modée par '–', est dite complète. Elle définit un type ***complet***. Une énumération modée par '+', '∗', '#', est dite restreinte. Elle définit un type ***restreint***.

Par défaut, un type est donc complet.

2.3.1 Signification des énumérations

Enumération complète :
L'ensemble de termes, défini par une énumération complète, est la fermeture par généralisation, (cf. 1.2.1), de l'union des ensembles, définis par les expressions de l'énumération simple.

Enumération modée par "+" :
L'ensemble de termes, défini par une énumération modée par "+", est l'union des ensembles, définis par les expressions de l'énumération simple, sans l'ensemble des variables.

Expression simple :

- Une expression simple peut être le nom d'un type défini dans la même déclaration, ou dans une déclaration antérieure (cf. 2.3.3.*Contrainte d'ordre de définition*). Elle définit alors l'ensemble de termes de ce type.

- Une expression simple de la forme $f(E_1, \ldots, E_n)$, où f est un foncteur n-aire Prolog, et les E_i $(1 \leq i \leq n)$ des énumérations, est appelée ***expression élémentaire***. Elle définit l'ensemble contenant les termes $f(x_1, \ldots, x_n)$ instanciables à un terme clos $f(c_1, \ldots, c_n)$, où x_i et c_i $(1 \leq i \leq n)$ sont du type défini par E_i.

Exemples :

- **type** $t_1 = a \mid f(a \mid b)$,
  $t_2 = g(c, d)$.

  définissent : $t_1 = gen\ \{a, f(a), f(b)\}$,
  $t_2 = gen\ \{g(c, d)\}$.

- **type** $t_0 = a \mid b$,
  $t_1 = +\ a \mid f(t_0)$,
  $t_2 = +\ g(+c, d)$.

  définissent :
  $t_0 = gen\ \{a, b\}$,
  $t_1 = \{a\} \cup \{f(x) \mid x \in gen\{a, b\}\}$, t_1 ne contient pas de variables,
  $t_2 = \{g(c, x) \mid x \in gen\{d\}\}$,
  t_2 ne contient pas de variables, le 1er argument de *g/2* n'est pas variable.

Remarques :

- Un type complet contient des termes non instanciés (variables), un type restreint ne contient que des termes instanciés (partiellement).

- Dans une énumération restreinte, les arguments d'une expression élémentaire peuvent eux-mêmes être restreints : il est ainsi possible de préciser finement le degré d'instanciation des termes.

- Il n'existe pas de type ne contenant que des variables : un type est toujours basé sur un ensemble de termes clos.

Les définitions de type peuvent aussi être récursives ou mutuellement récursives. Les types ainsi définis, appelés types récursifs, sont des ensembles infinis, déterminés de la façon suivante.

Signification des types récursifs : Soit les définitions de type :

$t_1 = E_1$
……
$t_n = E_n$

où chaque énumération E_i ($1 \le i \le n$) dépend (directement ou indirectement) de tous les noms $t_1, \ldots, t_n$.

L'ensemble de termes de type t_i ($1 \le i \le n$) est itérativement calculé par $\tau_i = \tau_{i0} \cup \tau_{i1} \cup \tau_{i2} \cup \ldots$, où :

τ_{i0}, $1 \le i \le n$, sont des ensembles obtenus respectivement de E_i en remplaçant t_1 par Ø, …, t_n par Ø,
τ_{i1}, $1 \le i \le n$, sont des ensembles obtenus respectivement de E_i en remplaçant t_1 par τ_{10}, …, t_n par τ_{n0},
τ_{i2}, $1 \le i \le n$, sont des ensembles obtenus respectivement de E_i en remplaçant t_1 par τ_{11}, …, t_n par τ_{n1},
……

Exemples :

- **type** *lista = [] | [a | lista].*

définit *lista = gen {[], [a], [a, a], [a, a, a], …},*

- **type** $t_1 = a \mid f(t_2)$,
$t_2 = b \mid g(t_1)$.

définissent :
$t_1 = gen\ \{a, f(b), f(g(a)), f(g(f(b))), \ldots\}$,
$t_2 = gen\ \{b, g(a), g(f(b)), g(f(g(a))), \ldots\}$.

- **type** *inlista = + [] | [a | inlista].*

définit *inlista* $= \{[], [x_1], [x_1, x_2], [x_1, x_2, x_3], \ldots \mid x_i \in gen\{a\}\}$,
dont les termes sont des listes "semi-closes" avec des éléments qui ne sont pas nécessairement clos.

Il est possible de définir un type par union d'autres types et d'expressions élémentaires. Par exemple,

type $t_1 = a \mid b \mid c,\ t_2 = f(a) \mid t_1$.

définissent : $t_1 = gen\ \{a, b, c\}$,
$t_2 = gen\ \{f(a), a, b, c\}$.

On dit que *f*/n est un **foncteur principal** de l'ensemble de termes E si E contient des termes de ce foncteur.

2.3.2 Autres éléments de l'énumération

Les définitions de type peuvent aussi contenir des noms de types prédéfinis. Nous utilisons dans ce système, les types prédéfinis, dérivés de C-Prolog :

integer	l'ensemble des entiers et des variables
number	l'ensemble des entiers, des réels et des variables
dbreference	l'ensemble des "références" et des variables
atom	l'ensemble des atomes et des variables
primitive	l'ensemble des entiers, réels, références et des variables
atomic	l'ensemble des atomes, des entiers et des variables

Accolades :
Les accolades dans les énumérations sont utilisées pour distinguer les cas où l'une des unités lexicales suivantes du langage de type : "–", "+", "∗", "#", est utilisée comme foncteur dans une expression élémentaire.

Par exemple dans la définition :
type *int_exp = integer | {int_exp+int_exp}.*
\+ est un foncteur Prolog, mais n'est pas le symbole de mode d'une énumération.

Mode "–" :
Le mode "–" permet de mettre en évidence la fermeture par généralisation de l'énumération qui le suit.

Par exemple :
type *t1 = + g(+c, d), 2 = – t1.*
définit *t2 = gen(t1) = gen {g(c, d)}.*

Ce mode est pratique pour réutiliser un type restreint dans la définition d'un type complet.

Les symboles "∗", "#", sont des notations supplémentaires, qui facilitent les définitions des types restreints.

Mode "∗" :
Un type, défini par une énumération, précédée par "∗", ne contient que les termes clos de l'ensemble spécifié par cette énumération.

Par exemple, $*\, f(a \mid b) = \{f(a), f(b)\}$.

Ce mode est utile pour définir des types qui ne contiennent que des termes clos.

Mode "#" :
Le mode "#" est utilisé pour définir un type récursif restreint, à partir d'un type récursif complet. Appliqué à un nom de type t, il définit une énumération modée, obtenue de celle du type t, en remplaçant toutes les occurrences du nom t par un nouveau nom, et ajoutant les "+" si nécessaire.

Exemples :

- Le type *inlista = + [] | [a | inlista]* peut aussi être défini à partir du type *lista* par :
  type *lista = [] | [a | lista],*
  inlista = # lista.

- La définition du type *t* dans la déclaration :
  type $t_c = a \mid f(b,\ g(t_c))$,
  $t = \#\ t_c$.
  a le même résultat que $t = + a \mid f(b,\ +g(t))$.

2.3.3 Contraintes dans les définitions de type

Les définitions de type doivent respecter certaines restrictions dont l'effet est, d'une part, de satisfaire certaines contraintes sémantiques, et d'autre part, de faciliter l'analyse des déclarations par le système.

Contrainte d'union disjointe :
Deux expressions d'une énumération doivent représenter des ensembles n'ayant aucun foncteur principal commun.

Par exemple, l'énumération *f(a,b) | f(c,d)* est interdite.

Cette contrainte garantit la fermeture d'un type par produit cartésien (cf. 3.1.Types dérivés).

Contrainte de mode :
Une énumération complète sans mode ne peut pas contenir une sous-énumération restreinte, ou un nom d'un type restreint.

Par exemple, la définition *t = a | f(a,+b)* est rejetée.
Cette contrainte a pour but d'obliger l'utilisateur à exprimer explicitement le fait qu'un type complet est fermé par généralisation.

Un cas particulier est cependant autorisé : le renommage d'un type restreint. Ainsi, la déclaration :

type *t1 = + a | b,*
t2 = t1.

est correcte, et *t2* est un type restreint, identique à *t1*.

Contrainte d'ordre des définitions :
Un type ne peut utiliser dans sa définition que des noms de type déjà définis, ou définis dans une même déclaration.

Cette contrainte a pour rôle de distinguer, dans les énumérations, un foncteur Prolog d'un nom de type. Les types mutuellement récursifs doivent donc être définis dans une seule déclaration. Par exemple :

type *pair = zéro | s(impair),*
impair = s(pair).

Il est à noter que le même foncteur peut appartenir à plusieurs types.

2.4 Définition des types polymorphes

Les τ-variables dans les expressions de type permettent de définir des types polymorphes. Une définition de type contenant des τ-variables est un schéma qui définit un type en fonction des valeurs de ses paramètres. Ce polymorphisme paramétrique permet à un prédicat d'avoir des arguments de types différents. Un prédicat déclaré d'un profil polymorphe, a un nombre infini de profils.

Exemples :

- Le prédicat *append/3* peut être déclaré dans ce système par :
  type *list(T) = [] | [T | list(T)].*
  profil *append(list(T), list(T), list(T)).*
  append([], L, L).
  append([X | L], M, [X | N]) :– append(L, M, N).
- Le type polymorphe *list(T)* restreint est obtenu par la définition :
  type *inlist(T) = + [] | [T | inlist(T)].*

Restrictions des définitions paramétriques :
- Les τ-variables utilisées à gauche et à droite dans une définition doivent être les mêmes.
- Une seule τ-variable est acceptée comme expression d'une énumération.

Par exemple, *list(T1, T2) = [] | [T1 | list(T1)]*, *t(T1, T2) = a | T1 | T2*, ne sont pas des définitions correctes.

Variable inclusive :
Si une expression d'une énumération est une τ-variable, on a une inclusion paramétrique. Cette τ-variable s'appelle ***variable inclusive***.

Un type paramétriquement inclusif permet de spécifier les types, contenant un type défini.

Exemples :

- **type** *t(T) = a | f(a) | T.*

  spécifie les types, qui ont un sous-type défini par *a | f(a)*.

- **type** *integer(T) = integer | T.*

  spécifie les types contenant le type *integer*.

Par conséquence de la contrainte d'union disjointe (cf. 2.3.3), une variable inclusive ne peut pas être remplacée par un type contenant les foncteurs de l'énumération. La variable inclusive sert donc, dans un type paramétrique, à séparer le reste du type de sa partie spécifiée.

Exemple :

type *list(T) = [] | [T | list(T)],*
tree(T) = list(tree(T)) | T.

Dans le type *tree(T)*, *T* exprime le type des termes dont le foncteur n'est pas *[]* ou *[_|_]*, c.à.d. le type des "feuilles d'un arbre".

Contraintes d'exclusion :
Chaque τ-variable T, qui est le paramètre d'un type polymorphe τ, doit respecter des contraintes d'exclusion qui sont déterminées par les positions de cette variable dans l'énumération de τ :

- Si T est la variable inclusive de τ, T ne doit pas contenir les foncteurs des termes de τ.
- Si T est le paramètre d'un type τ', qui se trouve dans la définition de τ, T doit satisfaire les contraintes d'exclusion définies pour lui dans τ'.

Exemple :

type *list(T) = [] | [T | list(T)],*
t(T) = + b | h(list(T)) | integer | T.

Dans *list(T)*, il n'y a pas de contrainte d'exclusion pour le paramètre *T*.

Dans *t(T)*, les contraintes d'exclusion de *T* sont les foncteurs *b*, *h/1* et tous les foncteurs de *integer*.

2.5 Relation "plus modé" entre types

2.5.1 Définition structurelle des types

Quelle que soit la forme syntaxique utilisée, un type peut toujours être défini par une énumération de composants élémentaires qui sont :
- soit une expression élémentaire $f(\tau_1, \ldots, \tau_n)$,
- soit un type primitif,
- soit une variable inclusive.

Ces composants sont ceux de la déclaration initiale du type, ainsi que ceux des sous-types présents dans l'énumération.

Par exemple, de la déclaration :

type $t_1 = a \mid b \mid c,$
$t_2 = f(a \mid d) \mid t_1.$

t_1 est défini par $a \mid b \mid c$, t_2 est défini par $f(a \mid d) \mid a \mid b \mid c$.

Les composants élémentaires d'un type spécifient la structure de ses éléments. Cette définition structurelle des types est donc utilisée dans la suite pour définir des relations entre les types comme ensembles de termes.

2.5.2 Inclusion des types

Un type τ_1 est inclus dans un type τ_2, si l'ensemble de termes de τ_1 est un sous-ensemble de celui de τ_2.

Exemple :

type $t_1 = a \mid b,$
$t_2 = t_1 \mid f(a),$
$t_3 = a \mid b \mid f(a \mid b).$

définissent :
$t_1 = gen\ \{a, b\},$
$t_2 = gen\ \{a, b, f(a)\},$
$t_3 = gen\ \{a, b, f(a), f(b)\},$
et impliquent que t_1 est inclus dans t_2, t_2 est inclus dans t_3.

2.5.3 Relation "plus modé"

La relation d'ordre des types utilisée pour exprimer les conditions de bon typage, est la relation partielle "plus modé" qui est définie par l'inclusion entre les types plus une propriété d'instanciation de leurs termes.

Définition :
Soit deux types τ_1 et τ_2, τ_1 est inclus dans τ_2. Le terme x de τ_1 est dit ***suffisamment instancié*** dans τ_1 pour τ_2, si toute instance de x (σx, pour toute substitution σ), appartenant à τ_2 appartient aussi à τ_1.

Exemple :

f(X) est suffisamment instancié dans $t_1 = + a \mid f(a)$ pour $t_2 = a \mid b \mid f(a)$.
f(X) n'est pas suffisament instancié dans t_1 pour $t_3 = a \mid b \mid f(a \mid b)$.

Définition :
Un type τ_1 est dit ***plus modé*** qu'un type τ_2, (τ_1 « τ_2), si τ_1 est inclus dans τ_2, et tout terme de τ_1 est suffisamment instancié dans τ_1 pour τ_2.

Remarque :

- Un type complet ne peut pas être plus modé qu'un type restreint, car il n'est pas inclus dans ce dernier.

- Si τ_1 est plus modé que τ_2 et τ_1, τ_2 sont complets, alors τ_1 doit être égal à τ_2. Par exemple, $t_1 = a \mid b$ ne peut pas être plus modé que $t_2 = a \mid b \mid c$, car une variable X de t_1 peut être instanciée à c qui appartient à t_2 mais pas à t_1.

Caractérisation constructive de la relation "plus modé" entre deux types

La relation "plus modé" entre deux types peut être détectée par les règles suivantes qui sont déduites de la définition de cette relation (cf. 3.2.Propriété1).

• Soit τ_1, τ_2 deux types monomorphes. τ_1 est plus modé que τ_2 si l'une des conditions suivantes est satisfaite :

1/ $\tau_1 = \tau_2$, ou

2/ τ_1 est restreint et :

- pour chaque composant $f(\tau_{11}, \ldots, \tau_{1n})$ de τ_1, il existe un composant $f(\tau_{21}, \ldots, \tau_{2n})$ de τ_2 tel que $\tau_{1i} \ll \tau_{2i}$, $1 \leq i \leq n$,
- pour chaque composant primitif Π_1 de τ_1, il existe un composant primitif Π_2 de τ_2 tel que $\Pi_1 \subseteq \Pi_2$.

Exemples :

- $t_1 = + a \mid f(a)$ est plus modé que $t_2 = f(a) \mid a$.

- $t_3 = + g(t_1) \mid integer$ est plus modé que $t_4 = g(t_2) \mid number$, car $t_1 \ll t_2$ et $integer \subseteq number$.

- $t_5 = + a \mid f(a \mid b)$ n'est pas plus modé que $t_6 = a \mid b \mid f(a \mid b \mid c)$, car dans la comparaison de $f(a \mid b)$ et $f(a \mid b \mid c)$, $a \mid b$ n'est pas plus modée que $a \mid b \mid c$.

• Pour les types polymorphes :

- $T \ll T$, pour tout T.
- $T \ll \tau \mid T$ pour tout T restreint.
- $T \ll \tau$ pour $T = \tau'$ tel que $\tau' \ll \tau$.
- $\tau \ll T$ pour $T = \tau'$ tel que $\tau \ll \tau'$.

Exemples :

- $t_1(T) = + a \mid f(T)$ est plus modé que $t_2(T) = a \mid b \mid f(T)$ pour tout T, car la comparaison de $f(T)$ et $f(T)$ amène à $T \ll T$ qui est vrai pour tout T.

- $t_1(T) = + a \mid f(a \mid b)$ est plus modé que $t_3 = a \mid b \mid f(T)$ pour $T = a \mid b$, car la comparaison de $f(a \mid b)$ et $f(T)$ amène à $a \mid b \ll T$ qui est vrai pour $T = a \mid b$.

- $t_6 = + a \mid b$ est plus modé que $t_5(T) = integer \mid T$ pour $T = a \mid b \mid T1$, car $t_5(T)$ doit contenir les composants a et b.

• Pour les types récursifs :
Si $\tau_1 = \tau'_1 \mid f(\tau_1)$, $\tau_2 = \tau'_2 \mid f(\tau_2)$, et $\tau'_1 \ll \tau'_2$, alors $\tau_1 \ll \tau_2$.

Exemple :

$t_1 = + a \mid f(t_1)$ est plus modé que $t_2 = a \mid b \mid f(t_2)$.

2.6 Les profils

Chaque profil est composé du foncteur d'un prédicat et des spécifications de type de ses arguments. Une spécification définit, pour l'argument correspondant, son type exigé dans chaque appel, et le type de son instance en retour si cet appel se termine avec succès. Un prédicat peut avoir plusieurs profils.

Les différentes formes de ces spécifications et leur signification sont les suivantes :

$E_1 \rightarrow E_2$:	E_1 est le type d'appel, E_2 est le type de retour de l'argument
$\rightarrow E$:	gen(E) est le type d'appel, E le type de retour de l'argument
E :	E est le type d'appel et de retour de l'argument

La distinction entre le type de l'argument avant et après l'effacement de chaque appel permet d'exprimer la particularité des prédicats dont le rôle est d'effectuer un test d'appartenance à un ensemble de valeurs. Par exemple, les prédicats prédéfinis *integer/1*, *atom/1*, …, ou le prédicat suivant :

```
est_liste([]).
est_liste([_| L]) :– est_liste(L).
```

qui teste si un terme est une liste. Le type de leur solution est plus petit que le type qu'ils peuvent accepter pour l'argument en entrée. L'information sur la propriété de l'argument en sortie est alors plus précise qu'en entrée. Ces prédicats peuvent avoir dans notre système les déclarations suivantes :

profil *integer(integer | T → +integer), atom(atom | T → +atom).*

profil *est_liste(list(T)→inlist(T)).*

Il est possible de spécifier finement des profils. Par exemple des profils possibles pour *append/3* sont :

profil *append(inlist(T), inlist(T), →inlist(T)),*
append(→inlist(T), →inlist(T), inlist(T)).

Ces profils expriment le fait qu'il suffit d'avoir en entrée soit les deux premiers arguments semi-clos, soit le troisième semi-clos, pour obtenir des listes semi-closes. Noter que les éléments des listes ne sont pas nécessairement instanciés.

Les conditions de correction des types exigent que seuls peuvent être déclarés comme types d'appel et de retour d'un argument, les types qui satisfont la contrainte suivante :

Contrainte d'entrée-sortie :
Le type de retour est plus modé que le type d'appel.

Remarque : τ est toujours plus modé que τ.
τ est toujours plus modé que gen(τ).

2.7 Vérification de bon typage d'une clause

Cette section présente les conditions de bon typage des programmes avec déclarations. La validité de ces conditions est donnée brièvement dans le principe de bon typage du système. L'algorithme de vérification des conditions de bon typage d'une clause est décrite après les définitions du type déduit d'une occurrence de variable et de l'intersection des types. Les exemples qui suivent montrent le rôle de cette vérification dans la détection des incohérences de type. La fin de la section présente des problèmes du traitement des prédicats d'ordre supérieur et le traitement concret du prédicat d'unification.

2.7.1 Principe de bon typage

La vérification de type dans ce système et la consistance de cette vérification sont exprimées par le principe de bon typage suivant :

- Un programme est dit **bien typé** si tous les prédicats sont bien typés. Un prédicat est dit **bien typé** si pour chacun de ses profils déclarés, toutes ses clauses sont bien typées.
- La résolution d'un but bien typé par un programme bien typé est cohérente par rapport aux profils déclarés des prédicats : Un prédicat est toujours appelé avec les arguments du type d'appel, et s'il réussit, les solutions sont toujours du type de retour.

En général, la vérification des conditions de bon typage d'une clause consiste à inférer, pour chaque variable de la clause, avant et après chaque sous-but, une assignation consistante (cf. 1.3.2), c.à.d un type :
- qui est non vide, déduit pour la variable à partir des déclarations des prédicats qui l'utilisent, et
- tel que lors de toute exécution de la clause, le terme représenté par la variable en est un élément.

Les calculs nécessaires pour cette inférence des types des variables dans une clause sont définis dans les deux parties suivantes, et l'algorithme plus concret de la vérification de type d'une clause est donné dans 2.7.4.

2.7.2 Déduction des types pour les occurrences de variable

Le calcule des types des positions de variables à partir des profils est une suite de déductions des types des sous-termes à partir du type d'un terme. Cette déduction est effectuée de la façon suivante : le type τ peut être déduit pour les sous-termes d'un terme $f(x_1, \ldots, x_n)$ si τ a un composant $f(\tau_1, \ldots, \tau_n)$, et le type déduit pour chaque x_i sera τ_i, $1 \leq i \leq n$.

Par exemple, si $f(X, g(X, Y))$ est du type $\tau = a \mid f(+a, g(a, a))$, les types déduits pour les deux occurrences de X sont $+a$ et a, pour l'occurrence de Y est a.

2.7.3 Intersection des types

L'intersection de deux types n'est pas forcément un type (cf. 3.1). Dans la suite de cette thèse nous utiliserons le terme "intersection" pour désigner le plus grand type inclus dans tous les deux types donnés.

Définition :
L'intersection de deux types τ_1 et τ_2, (notée τ_1 & τ_2), est l'ensemble des termes x appartenant à $\tau_1 \cap \tau_2$, instanciables à un terme clos élément de $\tau_1 \cap \tau_2$.

Exemples :

Si $t_1 = a \mid b \mid f(a \mid b)$, $t_2 = a \mid g(b) \mid f(a)$, alors t_1 & $t_2 = a \mid f(a)$.

Si $t_3 = + a \mid f(+a \mid b)$, $t_4 = a \mid b \mid f(a \mid b)$, alors t_3 & $t_4 = + a \mid f(+a \mid b)$.

Si $t_5 = a \mid b$, $t_6 = f(a \mid b)$, alors t_5 & $t_6 = \emptyset$.

2.7.4 Bon typage d'une clause et d'un but

Clause bien typée
La vérification du bon typage d'une clause par rapport à un profil donné d'un prédicat est effectuée de la façon suivante :

Au début, les types des variables de la tête sont déduits du type d'appel du prédicat. Si pour les différentes occurrences d'une variable sont déduits des types différents, il doit y en avoir un qui est le plus modé. Ce type est assigné à la variable.

Ensuite, après chaque sous-but du corps de la clause, les types assignés aux variables sont réduits à l'intersection de leur dernier type assigné et de leur type de retour dans ce sous-but. Le profil utilisé dans les sous-buts doit être une instance de leur déclaration.

Les types assignés aux variables de la clause doivent respecter la contrainte suivante :

- Dans chaque sous-but, pour une variable dont c'est la première occurrence dans la clause, les types d'appel déduits pour toutes ses positions doivent être complets et égaux. Pour une variable qui apparaissait déjà auparavant, son dernier type assigné doit être plus modé que les types d'appel déduits pour ses positions dans ce sous-but.

- A la fin de la clause, le type assigné d'une variable de la tête doit être plus modé que son type de retour déduit du profil donné.

Si une déduction de type ne peut pas être effectuée, ou une contrainte n'est pas satisfaite, la clause est mal typée.

But bien typé
Un but est bien typé s'il existe une instance d'un profil de son prédicat tel que les types déduits pour toutes les positions d'une variable sont complets et égaux.

Les définitions complètes de bon typage, d'intersection des types et de type d'une position de variable sont présentées dans le chapitre 3. L'algorithme précis de la vérification et de l'inférence de types des variables de la clause est donné dans le chapitre 4.

Exemple 1 :

type $t_1 = a \mid c,\quad t_2 = a \mid b,\quad t_3 = +a,$
$t_4 = f(t_1),\quad t_5 = f(t_2),$
profil $p(t_1, t_5),\ q(t_1 \rightarrow t_3),\ r(T, T).$

$p(X, Y)\ :-\ q(X),\ r(f(X), Y).$

Le résultat de la vérification effectuée sur cette clause et le profil $p(t_1, t_5)$, est le suivant :

$p(X, Y)\ :-\ q(X),\quad r(f(X), Y)$
$\uparrow \qquad\quad \uparrow \qquad\quad \uparrow$
$X : t_1 \quad X : t_3 \qquad X : t_3$
$Y : t_5 \quad Y : t_5 \qquad Y : t_5$

Après l'exécution de $q(X)$ le nouveau type de X est t_1 & $t_3 = t_3$, car t_3 est inclus dans t_1.

Pour le sous-but $r(f(X), Y)$, $T = t_5$ définit une instance de son profil polymorphe $r(T, T)$ qui satisfait sa contrainte d'appel : le type d'appel de Y est t_5 et $t_5 « t_5$.

Les types assignés à X et Y satisfont la contrainte de sortie de la tête. La clause est donc bien typée.

Exemple 2 :

type $t_1 = f(a), t_2 = t_1 \mid b.$
profil $p(t_1), q(t_2), r(t_1).$

p(X) :– q(X), r(X).
q(b).
q(f(a)).

Les clauses de *q/1* sont bien typées. La clause de *p/1* n'est pas bien typée, car le type assigné à *X* dans la tête, t_1, n'est pas plus modé que t_2, le type déduit pour *X* dans le sous-but *q(X)*.

En effet, la résolution du but :– *p(Y),* qui est bien du type d'appel de *p/1* car *Y* est variable, donc du type t_1, conduit à l'instanciation de *Y* à *b* après l'effacement de *q(Y)*. Le terme que *Y* représente n'est plus du type t_1, donc ne peut pas être l'argument de *r/1*.

2.7.5 Clause de profil polymorphe

Puisque dans la vérification du bon typage d'une clause, les conditions de bon typage doivent être satisfaites pour tous ses profils, dans le cas d'une clause d'un profil polymorphe, les τ-variables doivent être quantifiées universellement : ne doivent pas être instanciées.

Exemple 3 : Soit les déclarations du prédicat *append/3*

type *list(T) = [] | [T | list(T)],*
inlist(T) = + [] | [T | inlist(T)].

profil *append(list(T), list(T), list(T)),* *(1)*
append(inlist(T), inlist(T), list(T)→inlist(T)). *(2)*

L'analyse de la clause *append([A | L], M, [A | N]) :– append(L, M, N)* sur le profil *append(list(T), list(T), list(T)),* doit être faite de la façon suivante : trouver, pour tout *T* de ce profil appliqué sur la tête de la clause, une instance de T_1 du profil *(1)*, ou une instance de T_2 du profil *(2)* pour le sous-but, telle que ses arguments sont du type attendu.

L'instanciation $T_1 = T$ donne aux variables de la clause les assignations *A : T, L : list(T), M : list(T), N : list(T),* qui satisfont les conditions de la vérification.

De la même façon, l'analyse de cette clause sur le profil (2) conduit aux assignations *A : T, L : inlist(T), M : inlist(T), N : list(T).*

Cette clause est donc bien typée avec les déclaration données.

Au contraire, la clause *append([a | L], M, [a | N]) :– append(L, M, N)* n'est pas bien typée avec ces déclarations, car l'atome *a* n'appartient pas à *T* pour tout *T*.

2.7.6 Occurrences multiples de variables

Dans l'analyse d'une clause, la relation "plus modé" est nécessaire pour la comparaison entre deux types déduits pour une même variable. Elle permet d'accepter ou de refuser les répétitions de variables dans la tête, ou un sous-but d'une clause, quand les occurrences d'une variable correspondent à différents types.

Exemple 4 : Soit les déclarations suivantes et une clause du prédicat *dérivée/3* (cf.1.2.2) :

type *var = + x | y | z,*
exp = x | y | z | integer | {exp+exp}.
profil *dérivée(exp, var, exp).*

dérivée(X, X, 1).

Cette clause est bien typée par rapport au profil déclaré : les types déduits pour les deux positions de X sont *exp* et *var*, et *var* est plus modé que *exp*. La contrainte d'entrée est donc vérifiée, et évidemment la contrainte de sortie, car les types d'appel et de retour de ce profil sont les mêmes.

La question *:– dérivée(x+y, X, 1)* n'est pas légale, car le type déduit pour *X* est *var*, qui n'est pas complet et ce but n'est donc pas bien typé.

Exemple 5 :

type $t_1 = a,\ t_2 = a \mid b.$
profil $p(t_1, t_2),\ q(t_1, t_2),\ r(t_2),\ s(t_1).$

```
p(X, Y) :– q(X, Y), r(Y), s(X).
q(X, X).
q(_, b).
r(b).
```

La clause *q(X, X)* est mal typée par rapport à son profil $q(t_1, t_2)$, car les types déduits pour les deux positions de *X* sont t_1 et t_2, et aucun des deux n'est plus modé que l'autre.

En effet, le but :– *p(X, Y)* conduit à la substitution *X/Y* dans *q(X, Y)*, puis *Y/b* après *r(Y)*; en conséquence, *X*, instancié à *b*, n'est pas du type attendu de *s/1*.

2.7.7 Clause contenant des disjonctions et des conditions

Ces clauses sont transformées sous une forme qui ne contient que des conjonctions de sous-buts. La vérification d'une clause devient la vérification de plusieurs clauses.

Une clause de forme :

p :– q, (r ; s), t

où **;** signifie une disjonction, est transformée en deux clauses :

p :– q, r, t.
p :– q, s, t.

Une clause de forme :

p :– q, (r –> s ; t), u

où (r –> s ; t) signifie **if** r **then** s **else** t, est transformée en deux clauses :

p :– q, r, s, u.
p :– q, t, u.

2.7.8 Echec de la vérification et messages d'erreur

La vérification du bon typage peut échouer pour les raisons suivantes :
- la clause contient des erreurs syntaxiques, des fautes de frappes
- la clause est vraiment incohérente par rapport à son profil
- la clause est cohérente par rapport à son profil, mais ne satisfait pas les conditions de bon typage du système qui ne sont que des conditions suffisantes.

Les messages d'erreur fournis par le système pendant l'analyse d'une clause ont pour but d'aider l'utilisateur à trouver l'origine de l'échec et mettre à jour le programme. Ces messages sont donnés dans les cas suivant :

- Le foncteur d'un terme n'est pas le foncteur principal de son type attendu. Le terme n'appartient donc absolument pas à ce type.

- A différentes occurrences d'une variable de la tête sont déduits différents types incomparables par la relation "plus modé".

- Le prédicat d'un sous-but n'est pas encore déclaré.

- Le type assigné à une variable n'est pas plus modé que le type déduit pour cette variable dans un sous-but.

Exemple 6 : Soit le programme :

type *list(T) = [] | [T | list(T)],*
inlist(T) = + [] | [T | inlist(T)].
profil *append(inlist(T), inlist(T), →inlist(T)),*
reverse(inlist(T), →inlist(T)).

La vérification de la clause
reverse([A | L], R) :– reverse(L1, R1), append(R1, [A], R).
donne le message :

```
Profil : reverse(inlist(T), list(T)→inlist(T))
*****   assignation L1 : _13  n'est pas plus modé que inlist(T)
```

(`_13` correspond à l'affichage d'une variable non instanciée sous l'interprète C-Prolog)

Il est clair qu'il s'agit ici d'une faute de frappe (*L* est remplacé par *L1*).

2.7.9 Prédicats d'ordre supérieur

La plupart des prédicats prédéfinis sont spécifiés par des profils suivant leur sémantique. Ces profils (cf. Annexe) sont déclarés par le système et mis à disposition de l'utilisateur. Certains arguments de ces prédicats ne peuvent être caractérisés que par le type le plus général : *any*, qui n'apporte malheureusement aucune information sur les termes. C'est le cas des prédicats *read/1*, *recorded/3*, …, et des prédicats *arg/3*, *functor/3*. Les types restreints nous permettent pourtant de spécifier certains de ces prédicats par des profils plus précis. Par exemple, le profil de *arg/3* est *arg(+integer, any, any)*. les profils du prédicat *functor/3* sont : *functor(+T, →+atom, →+integer)*, *functor(→+any, +atom, +integer)*.

Les prédicats : *assert, call, setof, not, …*, ont des arguments qui sont eux-même des clauses ou des appels de prédicat, et non des termes Prolog. Notre notion de type n'exprime donc pas les propriétés de ces arguments, et ces prédicats doivent être traités à part par des algorithmes *ad hoc*. Les arguments qui sont des appels de prédicat, par exemple, sont analysés par rapport à leurs profils d'après l'algorithme de traitement d'un sous-but.

Un cas particulier constitue celui du prédicat *not.* Soit un profil $p(\tau_1 \rightarrow \tau'_1, \ \ldots, \ \tau_n \rightarrow \tau'_n)$ du prédicat p/n. La vérification du sous-but not $p(x_1, \ \ldots, \ x_n)$ consiste à l'analyser par rapport à not $p(\tau_1, \ \ldots, \ \tau_n)$, ce qui est vraiment exigé pour ses arguments en entrée. Pour avoir un type de sortie plus précis, il faut des évaluations délicates qui peuvent donner des informations sur les solutions de P et qui dépassent donc le travail de cette thèse.

Exemple 7 :

type $t_1 = + a \mid b, \ t_2 = + a, \ t_3 = + b.$
profil $p(t_1), \ q(t_1 \rightarrow t_2), \ r(t_1).$

p(X) :– not q(X), r(X).

Le profil utilisé pour analyser le sous-but *not q(X)* est *not* $q(t_1)$, et *X* obtient donc, après ce sous-but, l'assignation t_1 qui satisfait le profil de *r/1*. La clause est bien typée.

Par contre, avec les profils :
profil $p(t_1), \ q(t_1 \rightarrow t_2), \ r(t_3)$
cette assignation de *X* ne satisfait plus le profil de *r/1*. La clause est mal typée.

Pour certains prédicats prédéfinis, nous permettons des déclarations plus précises :

profil *not atom(+ atom | T → +T),*
not atomic(+ atomic | T → +T),
not db_reference(+ db_ref | T → +T),
not integer(+ integer | T → +T),
not number(+ number | T → +T),
not primitive(+ primitive | T → +T).

2.7.10 Prédicat d'unification

D'après la sémantique de l'unification et les conditions de consistance de la vérification, les profils de ce prédicat prédéfini sont de la forme :

$$(Type_1 \rightarrow Type_1 \,\&\, Type_2) = (Type_2 \rightarrow Type_1 \,\&\, Type_2)$$

où il existe un type $Type_0$ tel que $Type_1 \ll Type_0$ et $Type_2 \ll Type_0$. $(*)$

La contrainte d'entrée-sortie de ces profils (cf. 2.6) :
$(Type_1 \,\&\, Type_2) \ll Type_1$ et $(Type_1 \,\&\, Type_2) \ll Type_2$,
est déduite directement de la contrainte $(*)$ et la définition de la relation "plus modé".

Ces profils englobent tous les cas particuliers suivants du prédicat :

$(Type_1 \rightarrow Type_2) = Type_2$	quand l'argument à gauche est instancié à l'argument à droite
$Type_2 = (Type_1 \rightarrow Type_2)$	quand l'argument à droite est instancié à l'argument à gauche
Type = Type	quand un test d'égalité des deux arguments est effectué.

A la différence des autres prédicats, où le nombre de profils à essayer, y compris les profils polymorphes, est fini, $Type_1$ et $Type_2$ doivent être calculés à partir des arguments du sous-but, des contraintes entre les assignations et des contraintes entre $Type_1$ et $Type_2$ eux-mêmes. Ces profils ne peuvent donc pas être déclarés d'une facon officielle comme les autres profils. L'analyse d'un sous-but de l'unification, soit $x_1 = x_2$, consiste donc à :

- Trouver T_1 et T_2 tels que la contrainte d'assignations des variables d'un sous-but (cf. 2.7.4) soit satisfaite.
- Vérifier qu'il existe un type T tel que $T_1 \ll T$ et $T_2 \ll T$.
- Assigner à chaque variable du sous-but l'intersection de sa dernière assignation et de son type de retour déduits de $T_1 \,\&\, T_2$.

Exemple 8 :

type *oper = {+} | {–} | {∗},*
var = a | b | c,
terme = oper | var,
inlist(T) = + [] | [T | inlist(T)].
profil *analyse(+oper, inlist(terme)),*
éval(+oper, inlist(terme)).

analyse(Op, [Prefix | Exp]) :– *Op = Prefix, éval(Prefix, Exp).*

Dans le sous-but *Op=Prefix*, les instances trouvées sont T_1 = + *oper*, T_2 = *terme*, $T = T_2$ car + *oper « terme*. Le type de retour du sous-but et donc de *Prefix*, est +*oper*, et le sous-but suivant est aussi bien typé.

Exemple 9 : Si maintenant *analyse/2* est déclaré par

profil *analyse(oper, inlist(terme)).*

cette clause n'est plus bien typée : T_1 = *oper*, et T_2 = *terme*, et il n'existe donc aucun type *T* tel que $T_1 « T$ et $T_2 « T$.

Nous espérons que ces exemples ont bien montré notre approche de typage et la méthode de vérification de types dans le système. Le chapitre suivant donne le fondement théorique de cette vérification et des notions utilisées.

Chapitre 3

BASE THEORIQUE DU SYSTEME

Cette partie présente la construction formelle du système de type proposé. Les concepts de base : notion de type, relation “plus modé”, sont définis de façon ensembliste, indépendamment de tout langage d’expression de type. Nous décrivons ensuite les conditions de bon typage, qui sont des conditions suffisantes pour la consistance des types d’un programme. Ce résultat, exprimé souvent dans les systèmes de types pour Prolog par le principe: “Un programme bien typé se comporte toujours correctement” (ne produit jamais d’erreur de type à l’exécution), est donné dans le théorème central de notre système de types.

3.1 Définition de la notion de type

La notion de type a été conçue à partir des propriétés nécessaires au langage Prolog et de l’approche choisie dans notre système (cf. 1.5). Plus concrètement :

- Puisque l’Univers de Herbrand est le domaine d’interprétation des programmes, chaque élément d’un type doit pouvoir trouver dans le même type au moins une instance close.
- Un type peut contenir une information de mode.
- L’exécution d’un programme ne peut qu’aumenter le degré d’instanciation de tout terme. Il est donc nécessaire que si x et y appartiennent à un type, tout terme z tel que $x \leq z \leq y$, appartient aussi au même type.
- Pendant une exécution “sûre” du programme, un terme doit rester toujours dans son type; l’unification de deux termes d’un même type donne un terme de ce type.

Définition :
Soit x, x' deux termes. $x \leq x'$ (x est ***plus général*** que x'), si $\exists\ \sigma$, $\sigma x = x'$.
$x \lessapprox x'$ (x est ***plus général par variables*** que x'), si $\exists\ \sigma$, $\sigma x = x'$, et $\forall\ V \in Var$, $\sigma V \in Var$.

Remarque : Si $x \lessapprox x'$, x n’a des répétitions de variable que s’il y en a dans x'. Par exemple $f(X, Y) \lessapprox f(Z, Z)$, $f(V, V) \lessapprox f(Z, Z)$.

Définition :
Soit E un ensemble de termes.
prod(E) = $\{x \in E \mid \exists\, x_c \text{ clos} \in E,\ x \leq x_c\}$,
gen(E) = $\{x \mid \exists\, x' \in E,\ x \leq x'\}$.
prod(E) est le ***noyau productif*** de E, gen(E) est la ***fermeture par généralisation*** de E.

Soit x un terme. Vars(x) dénote l'ensemble des variables ayant une occurrence dans x.

Définition :
Soit x, x' deux termes. éch(x, x'), l'***échange d'arguments*** de x et x', est l'ensemble de termes défini par :
Si $x = f(x_1, \ldots, x_n)$, $x' = f(x'_1, \ldots, x'_n)$, et $Vars(x) \cap Vars(x') = \emptyset$,
$éch(x, x') = \{f(x''_1, \ldots, x''_n) \mid x_i'' \in éch(x_i, x'_i)\}$
Sinon, éch(x, x') = {x, x'}.

Exemples :
$x_1 = f(a, Y)$, $x_2 = f(X, b)$
$\Rightarrow \quad éch(x_1, x_2) = \{f(a, Y), f(a, b), f(X, b), f(X, Y)\}$
$x'_1 = f(X, X)$, $x'_2 = f(b, Y)$
$\Rightarrow \quad éch(x'_1, x'_2) = \{f(X, X), f(X, Y), f(b, X), f(b, Y)\}$

Définition :
Un ensemble de termes E est appelé un ***type*** si :
1/ E est fermé par généralisation de variables : $\forall x \in E, \forall x' \lessapprox x, x' \in E$.
2/ E est continu : $\forall x_1, x_2 \in E, \forall x$ tel que $x_1 \leq x \leq x_2,\ x \in E$.
3/ E est productif : $\forall x \in E, \exists x'$ clos $\in E$ tel que $x \leq x'$.
4/ E est fermé par échange d'arguments : $\forall x, x' \in E$, $éch(x, x') \subseteq E$.

Exemples :
$\tau_1 = \{f(X) \mid X \in Var\} \cup \{f(a)\}$ est un type,
qui peut être défini dans le langage de types (cf. 2.3.1), par $+f(a)$.

$\tau_2 = Var \cup \{f(X) \mid X \in Var\} \cup \{f(a)\}$ est un type,
qui peut être défini dans le langage de types par $f(a)$.

La fermeture par généralisation de variables (et pas la généralisation totale), permet d'avoir des types qui ne contiennent pas des variables, par exemple τ_1. Ces types sont dits ***restreints***. Un type fermé par la généralisation totale, comme τ_2 dans l'exemple, est dit ***complet***.

La continuité correspond à l'évolution du degré d'instanciation des termes pendant l'exécution du programme. C'est aussi une condition nécessaire pour que l'instance la plus générale de deux termes d'un type reste dans ce type.

La productivité sert à ne pas changer la sémantique du langage pour les programmes typés. Elle permet de trouver toujours une interprétation de Herbrand correspondante à un programme bien typé.

La fermeture par échange d'arguments est une généralisation de la fermeture par produit cartésien présentée dans 1.2.1.

Types dérivés :
Soit $\tau_1, \ldots,\tau_n$ des types, f un foncteur n-aire. On définit :

- $\tau_1 \,\&\, \tau_2 = \text{prod}(\tau_1 \cap \tau_2)$
- $+f(\tau_1, \ldots,\tau_n) = \text{prod}(\{f(x_1, \ldots, x_n) \mid x_i \in \tau_i\})$
- $f(\tau_1, \ldots,\tau_n) = \text{gen}(+f(\tau_1, \ldots,\tau_n))$
- $\tau_1 \mid \tau_2 = \tau_1 \cup \tau_2$, si τ_1 et τ_2 sont tous complets (ou restreints), et τ_1 et τ_2 n'ont aucun foncteur principal (cf. 2.3.1) commun.

On montre facilement que ces ensembles sont des types.

Remarques :

- L'intersection (ensembliste) de deux types n'est pas forcément un type. Par exemple, $a \cap b$ ne contient aucun terme clos; $f(a, a \mid b) \cap f(a \mid b, b)$ contient un seul terme clos $f(a, b)$ et aussi $f(X, X)$ qui n'est pas instanciable à $f(a, b)$.

- L'union (ensembliste) de deux types n'est pas toujours un type (par exemple, $f(a, b) \cup f(c, d)$ ou $a \cup +f(+a)$. La contrainte d'union disjointe (cf. 2.3.3) et la factorisation du symbole de mode de l'énumération (cf. 2.3.1), assurent que l'opération "|" du langage de type du chapitre 2 a la même signification définie ici et que les énumérations représentent bien des types.

- Pourtant notre définition de type ne se limite pas à ce langage. Par exemple, $\{s^n(0) \mid n \text{ premier}\}$ est un type, où $s^n(0) \equiv \underbrace{s(\ldots s}_{n}(0)\ldots)$.

A partir de la définition générale de type, on peut prouver le lemme suivant, qui est à la base de notre théorie.

Lemme 1 (Lemme de Décomposition) :
Soit τ un type, f/n un foncteur principal de τ. Il existe de façon unique un ensemble E, et des types $\tau_1, \ldots, \tau_n$ tels que $\tau = +\, f(\tau_1, \ldots, \tau_n) \cup E$, et f/n n'est pas foncteur principal de E.

Démonstration :
Prenons $\tau_f = \{f(x_1, \ldots, x_n) \in \tau\}$, $E = \tau \setminus \tau_f$, et
$\tau_i = \{x \mid \exists\, f(y_1, \ldots, y_{i-1}, x, y_{i+1}, \ldots, y_n) \in \tau_f\}$, $1 \le i \le n$.
Prouvons que $\tau_f = +\, f(\tau_1, \ldots, \tau_n)$ et τ_i, $1 \le i \le n$, sont des types.

- $\tau_f \subseteq +\, f(\tau_1, \ldots, \tau_n)$ est déduite de la définition de τ_i.
  Soit $f(x_1, \ldots, x_n) \in +\, f(\tau_1, \ldots, \tau_n) \Rightarrow x_i \in \tau_i$, $1 \le i \le n$. Par définition de τ_i et par échange d'arguments de τ_f, on prouve que $f(x_1, \ldots, x_n) \in \tau_f$.
  Donc $\tau_f = +\, f(\tau_1, \ldots, \tau_n)$ et il est évident que cette décomposition de τ en E et $\tau_1, \ldots, \tau_n$, est unique.
- Montrons la continuité de τ_i, $1 \le i \le n$: Soit $x, u \in \tau_i$, $1 \le i \le n$ et $x \le z \le u \Rightarrow$
  $\exists\, f(y_1, \ldots, y_{i-1}, x, y_{i+1}, \ldots, y_n), f(v_1, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_n) \in \tau$.
  Par généralisation de variables et échange d'arguments de τ, on peut déduire que $f(y_1, \ldots, y_{i-1}, u, y_{i+1}, \ldots, y_n) \in \tau \Rightarrow f(y_1, \ldots, y_{i-1}, u, y_{i+1}, \ldots, y_n) \in \tau$ par continuité de $\tau \Rightarrow$ CQFD.
  La productivité, la fermeture par généralisation de variables, la fermeture par échange d'arguments de τ_i, sont déduites directement de la même propriété de τ. Donc τ_i, $1 \le i \le n$, sont des types.

Remarque : E, le "reste" des termes de τ, n'est pas nécessairement un type.

3.2 Relation "plus modé" entre types

Définition :
Soit deux types τ_1 et τ_2 tels que $\tau_1 \subseteq \tau_2$, et $x \in \tau_1$. x est dit ***suffisamment instancié dans*** τ_1 ***pour*** τ_2, si $\forall\, \sigma$ substitution, $(\sigma x \in \tau_2 \Rightarrow \sigma x \in \tau_1)$.

Exemple :
Soit $\tau_1 = f(a, a \mid b)$, $\tau_2 = f(a \mid b, a \mid b)$. Alors :
$x = f(a, X)$ est suffisamment instancié dans τ_1 pour τ_2,
$y = f(Y, Y)$ ne l'est pas : $\{Y/b\}y = f(b, b)$ appartient à τ_2 mais pas à τ_1.

Définition de la relation "plus modé", notée «.
Soit deux types τ_1, τ_2. τ_1 est dit ***plus modé*** que τ_2, (notation : $\tau_1 \ll \tau_2$), si $\tau_1 \subseteq \tau_2$, et $\forall x \in \tau_1$, x est suffisamment instancié dans τ_1 pour τ_2.

Exemples : *+ f(a) « f(a) | b, + f(a) ¬« f(a | b), +f(+a) « f(a | b).*

Remarque : Cette relation est un ordre partiel.

Propriété 1 :
1/ $+f(\tau_1, \ldots, \tau_n) \mid \tau \ll +f(\tau'_1, \ldots, \tau'_n) \mid \tau'$ ssi $\tau_i \ll \tau'_i$, $1 \le i \le n$, et $\tau \ll \tau'$
2/ $\tau \mid \tau_1 \ll \tau \mid \tau_2$ ssi $\tau_1 \ll \tau_2$
3/ Si τ_1 , τ_2 satisfont les équations : $\tau_1 = f(\tau_1) \mid \tau'_1$, $\tau_2 = f(\tau_2) \mid \tau'_2$, alors $\tau_1 \ll \tau_2$ ssi $\tau'_1 \ll \tau'_2$.

Démonstration :
1/ Prouvons que $+f(\tau_1, \ldots, \tau_n) \ll +f(\tau'_1, \ldots, \tau'_n)$ ssi $\tau_i \ll \tau'_i$, $1 \le i \le n$.
- Soit $+f(\tau_1, \ldots, \tau_n) \ll +f(\tau'_1, \ldots, \tau'_n)$ et soit $x \in \tau_i$, $1 \le i \le n$.
  Par productivité de τ_i, on peut montrer que $\exists$ c_j clos, $j \neq i$, tels que $f(c_1, \ldots, x, \ldots, c_n) \in +f(\tau_1, \ldots, \tau_n)$.
  $\forall \sigma$, $\sigma x \in \tau'_i \Rightarrow \sigma f(c_1, \ldots, x, \ldots, c_n) \in +f(\tau'_1, \ldots, \tau'_n) \Rightarrow$ Par hypothèse, $\sigma f(c_1, \ldots, x, \ldots, c_n) \in +f(\tau_1, \ldots, \tau_n) \Rightarrow \sigma x \in \tau_i \Rightarrow \tau_i \ll \tau'_i$, $1 \le i \le n$.
- ($\Leftarrow$) est déduit facilement de la définition de la relation "plus modé".

2/ Déduite directement de la définition de la relation "plus modé".

3/ Cette propriété peut être prouvée par induction structurelle.

Conséquence : Ces propriétés prouvent que la détermination de la relation "plus modé" des types, décrite dans 2.5.3, est correcte.

Définition :
La ***position*** d'un sous-terme y d'un terme x est un élément de $\mathbb{N}^*$ défini par :
- ε, si $y = x$
- i.u, si $x = f(x_1, \ldots, x_n)$ et y est un sous-terme de x_i en position *u*.

Par exemple : Dans f(a, g(a)), *a* se trouve aux positions 1 et 2.1.

Définition :
Soit τ un type, x un terme de gen(τ). Le ***type d'une position*** u de x dans τ, noté p(τ, x, u), est défini par :

$p(\tau, V, \varepsilon) = \tau$

$p(+f(\tau_1, \ldots, \tau_n) \cup E, f(x_1, \ldots, x_n), \varepsilon) = +f(\tau_1, \ldots, \tau_n)$

$p(+f(\tau_1, \ldots, \tau_n) \cup E, f(x_1, \ldots, x_n), i.u) = p(\tau_i, x_i, u)$

Exemple : Soit $\tau = +f(g(a), +b)$, $x = f(g(X), Y)$.
Alors : $p(\tau, x, 1) = +g(a)$, $p(\tau, x, 1.1) = a$, $p(\tau, x, 2) = +b$.

Lemme 2 : Soit τ_1, τ_2 deux types. $\tau_1 \ll \tau_2$ ssi $\tau_1 \subseteq \tau_2$ et $\forall x \in \tau_1$, $\forall V \in$ Vars(x), $\forall$ u position de V dans x, $p(\tau_1, x, u) = p(\tau_2, x, u)$.

Démonstration :
Par récurrence sur la longueur des positions de variables dans les termes de τ_1.

- Soit $u = \varepsilon$. On a $x = V \in \tau_1$ donc τ_1 et τ_2 sont complets et $\tau_1 \ll \tau_2 \Leftrightarrow \tau_1 = \tau_2$ et $p(\tau_1, x, u) = \tau_1$, $p(\tau_2, x, u) = \tau_2 \Rightarrow$ CQFD.
- Supposons que le lemme est correct pour toutes les position u de longeur k. Considérons toute position i.u de V dans x, u de longeur k.
  On a $x = f(x_1, \ldots, x_n)$, V se trouve dans la position u de x_i et $\tau_1 = +f(\tau'_1, \ldots, \tau'_n) \mid E'$, $\tau_2 = +f(\tau''_1, \ldots, \tau''_n) \mid E''$ d'après le lemme de Décomposition.
  Par propriété 1, $\tau_1 \ll \tau_2 \Leftrightarrow \tau'_i \ll \tau''_i$, $1 \le i \le n \Leftrightarrow \forall u$, $p(\tau'_i, x_i, u) = p(\tau''_i, x_i, u)$ par hypothèse de récurrence.
  $p(\tau'_i, x_i, u) = p(\tau_1, x, u)$, $p(\tau''_i, x_i, u) = p(\tau_2, x, u) \Rightarrow$ CQFD.

Exemple :
Les types $\tau_1 = +f(a \mid b, b)$, $\tau_2 = f(a \mid b, a \mid b)$ ne satisfont pas les conditions du lemme : $\tau_1 \subseteq \tau_2$, mais pour $x = f(a, X) \in \tau_1$, $p(\tau_1, x, 2) \neq p(\tau_2, x, 2)$.
Et en effet, $\tau_1 \neg\ll \tau_2$ car $\{X/a\}f(a, X) = f(a, a) \notin \tau_1$

3.3 Assignations déduites dans un terme

Les notions présentées dans cette partie : la relation déduite, la relation déduite modée et l'assignation déduite modée d'un type sur un terme, sont les moyens principaux pour former les conditions de bon typage des programmes Prolog. Elles expriment précisément les règles d'inférence de type pour les variables d'une clause à partir des déclarations utilisées dans la vérification de types (cf. 2.7.3) du système.

Une remarque importante est qu'un type n'est pas fermé par unification. Par exemple, si $\tau = +f(a, a \mid b)$, on a $x = f(X, X)$, $x' = f(Y, b) \in \tau$, mais l'unification $x = x'$ donne $f(b, b) \notin \tau$.

Cette particularité constitue une origine des problèmes d'aliasing. Pour trouver les conditions du bon typage des programmes, on est conduit tout d'abord à chercher des conditions suffisantes pour qu'une substitution transforme un terme à un autre dans son type et que deux termes d'un type s'unifient en un terme du même type. Ces conditions sont présentées dans les propriétés 2.1, 2.2, 3.1, 3.2, et le lemme d'unification.

Définition :
Soit E un ensemble de types. $\min E = \{\tau \in E \mid \forall \tau' \in E, \tau' \ll \tau \Rightarrow \tau' = \tau\}$.
min E est l'ensemble des minimaux de E pour $\ll$.

Soit Θ l'ensemble de tous les types.

Définition :
Soit τ un type et x un terme $\in gen(\tau)$.

- $r(\tau, x) \subseteq Var \times \Theta$, la ***relation déduite*** de τ sur x, est définie par :
  $r(\tau, x) = \{V : p(\tau, x, u) \mid V \in Vars(x), u \text{ position de } V \text{ dans } x\}$
  $r(\tau, x, V) = \{\tau' \mid V : \tau' \in r(\tau, x)\}$

- $m(\tau, x)$, la ***relation déduite modée*** de τ sur x, est définie par :
  $m(\tau, x) = \{V : \tau_V \mid \tau_V \in \min r(\tau, x, V)\}$
  $m(\tau, x, V) = \{\tau_V \mid V : \tau_V \in m(\tau, x)\}$

- $a(\tau, x)$, l'***assignation déduite modée*** de τ sur x, est définie par :
  $a(\tau, x) = \{V : \& r(\tau, x, V)\}$
  $a(\tau, x, V) = \& r(\tau, x, V)$

Exemples :

- Soit $\tau_1 = +f(+a, a \mid b, a \mid c)$, $x_1 = f(X, X, X)$.
  $r(\tau_1, x_1) = \{X : +a, X : a \mid b, X : a \mid c\}$
  $m(\tau_1, x_1) = \{X : +a\}$
  $a(\tau_1, x_1) = \{X : +a\}$

- Soit $\tau_2 = f(a, b)$, $x_2 = f(X, X)$.
  $r(\tau_2, x_2) = m(\tau_2, x_2) = \{X : a, X : b\}$, $a(\tau_2, x_2) = \{X : \emptyset\}$

Définition :
$r(\tau, x)$ est dite ***fonctionnelle*** si $\forall V \in Vars(x)$, $r(\tau, x, V)$ est un singleton.
$m(\tau, x)$ est dite ***fonctionnelle*** si $\forall V \in Vars(x)$, $m(\tau, x, V)$ est un singleton.

Par exemple, $m(\tau_1, x_1)$ est fonctionnelle, mais $r(\tau_1, x_1)$ ne l'est pas.

Il en résulte que :
- Pour $x \in gen(\tau)$, $a(\tau, x, V) \neq \emptyset$, $\forall V \in Vars(x)$.
- $x \in \tau$ ssi $\forall V \in Vars(x)$, $a(\tau, x, V)$ est complet.
- Si $x \in \tau_1$, $\tau_1 \ll \tau_2$, alors $r(\tau_1, x) = r(\tau_2, x)$.

Les propriétés qui suivent concernent la conservation de type d'un terme par substitution, et sont donc des résultats de base pour prouver le lemme d'unification et le théorème central du système.

Propriété 2.1 :
Soit $x \in gen(\tau)$. Si σ est une substitution telle que $\sigma x \in gen(\tau)$, alors :
1/ $\forall V \in Vars(x)$, $\sigma V \in gen(a(\tau, x, V))$,
2/ $r(\tau, \sigma x) = \bigcup_{V \in Vars(x)} \bigcup_{\tau_V \in r(\tau, x, V)} r(\tau_V, \sigma V)$

Propriété 2.2 :
Soit $x \in gen(\tau)$.
Si σ est une substitution telle que $\sigma t \in \tau$, alors $\forall V \in Vars(x)$, $\sigma V \in a(\tau, x, V)$.

Propriété 3.1 :
Soit $x \in gen(\tau)$, σ une substitution telle que :
1/ $\forall V \in Vars(x)$, $\sigma V \in gen(a(\tau, x, V))$,
2/ $\forall X \in Vars(\sigma x)$, & $R(X) \neq \emptyset$, où :
$R = \bigcup_{V \in Vars(x)} \bigcup_{\tau_V \in r(\tau, x, V)} r(\tau_V, \sigma V)$, $R(X) = \{\tau_X \mid X : \tau_X \in R\}$

Alors $\sigma x \in gen(\tau)$.

Propriété 3.2 :

Soit $x \in gen(\tau)$, σ une substitution telle que :

1/ $\forall V \in Vars(x), \sigma V \in a(\tau, x, V)$,

2/ $R = \cup_{V \in Vars(x)} \cup_{\tau_V \in r(\tau,x,V)} r(\tau_V, \sigma V)$ est fonctionnelle

Alors $\sigma x \in \tau$.

Ces propriétés expriment en fait les conditions nécessaires (2.1 et 2.2) et suffisantes (3.1 et 3.2) pour qu'un terme x substitué par σ reste dans son type τ (ou sa fermeture par généralisation). Ces conditions consistent à ce que chaque variable V de x est substituée par un terme appartenant à tous les types déduits de τ pour les positions de V dans x. La condition suffisante est renforcée par la contrainte que les types déduits de τ pour les positions d'une variable dans σx sont les mêmes (ou croisés).

Exemples : Soit $\tau = +f(+g(a), g(b))$, $x = f(X, Y) \Rightarrow$
$a(\tau, x, X) = +g(a)$, $a(\tau, x, Y) = g(b)$

- Pour $\sigma = \{X/g(U), Y/g(V)\}$: $\sigma X \in gen(a(\tau, x, X))$, $\sigma Y \in gen(a(\tau, x, Y))$
  $R = r(+g(a), g(U)) \cup r(g(b), g(V)) \Rightarrow \& R(U) \neq \emptyset$, $\& R(V) \neq \emptyset$
  Les conditions de 3.1 sont satisfaites et en effet :
  $\sigma x = f(g(U), g(V)) \in gen(\tau)$.

- Pour $\sigma = \{X/g(Z), Y/g(Z)\}$:
  Pour Z sont déduits deux types a et b, qui ne sont pas croisés. Les conditions de 3.1 ne sont pas satisfaites et en effet :
  $\sigma x = f(g(Z), g(Z)) \notin gen(\tau)$.

- Pour $\sigma = \{X/g(U), Y/V\}$:
  $\sigma X \in a(\tau, x, X)$, $\sigma Y \in a(\tau, x, Y)$, $R = \{U : a, V : b\}$ est fonctionnelle.
  Les conditions de 3.2 sont satisfaites et en effet $\sigma x = f(g(U), V) \in \tau$.

- Pour $\sigma = \{X/U, Y/g(V)\}$:
  $\sigma X = U \notin a(\tau, x, X)$, donc les conditions de 3.2 ne sont pas satisfaites et en effet $\sigma x = f(U, g(V)) \notin \tau$.

Lemme 3 (Lemme d'Unification) :

Soit τ un type et x, y des termes qui satisfont les conditions suivantes :

- $x \in gen(\tau)$, $y \in \tau$
- $m(\tau, x) \cup r(\tau, y)$ est fonctionnelle
- $\exists \theta = pgu(x, y)$

Alors :

- $\theta x = \theta y \in \tau$

- $r(\tau, \theta x)$ est fonctionnelle

Exemples :

- Soit $x = f(X, X)$, $y = f(a, Y)$, $\tau = f(+a, a \mid b)$.
  Les conditions du lemme sont satisfaites avec :
  $m(\tau, x) \cup r(\tau, y) = \{X : a, Y : a \mid b\}$, et $\theta = \{X/a, Y/a\}$,
  et en effet :
  $\theta x = \theta y = f(a, a) \in \tau$, $r(\tau, \theta x) = \{\}$ est fonctionnelle.

- Soit $x = f(X, X)$, $y = f(Y, b)$, $\tau = f(a, a \mid b)$.
  $m(\tau, x) = \{X : a, X : a \mid b\}$ n'est pas fonctionnelle, et $\theta x = \theta y = f(b, b) \notin \tau$.

Nous allons prouver ce lemme par récurrence en suivant l'algorithme général de recherche du pgu θ de x, y unifiables [Lloyd 84] que nous rappelons ici :

1. $k = 0$: $\sigma_0 = \varepsilon$ (identité)
2. Soit $x_k = \sigma_k x$, $y_k = \sigma_k y$. Si $x_k = y_k$, <u>stop</u> : $\theta = \sigma_k$
   Sinon, soit $D_k = \{V, x, [u]\}$, une différence de position entre x_k, y_k, où V est variable, $x \neq V$ et V, x se trouvent l'un à x_k, l'autre à y_k, dans la même position u. (Il peut y avoir plusieurs D_k). $\sigma_{k+1} = \{V/x\} \bullet \sigma_k$.

Démonstration du lemme d'Unification :
Montrons par récurrence que $\forall$ k, $x_k = \sigma_k x \in gen(\tau)$,
$y_k = \sigma_k y \in \tau$,
$m(\tau, x_k) \cup r(\tau, y_k)$ est fonctionnelle,
ce qui conduit au résultat du lemme quand l'algorithme s'arrête.

* **k = 0**

$\sigma_0 = \varepsilon \Rightarrow$ C'est évident

* **k/ $\Rightarrow$ k+1/**

Parmi les différences entre x_k et y_k, choisissons $D_k = \{V, x, [u]\}$ telle que :
Si $V \in Vars(x_k)$, $p(\tau, x_k, u) = a(\tau, x_k, V)$ qui est le seul élément de $m(\tau, x_k, V)$;
Si $V \in Vars(y_k)$, $r(\tau, x_k)$ est fonctionnelle.
(Si $\exists$ des différences où $V \in x_k$, choisir celle dans la position de $m(\tau, x_k, V)$.
Si $\forall$ différences, $V \in y_k \Rightarrow \forall X \in x_k$, $\forall$ position u de X, $p(\tau, x_k, u)$ est un type complet. Mais $m(\tau, x_k)$ est fonctionnelle $\Rightarrow$ $r(\tau, x_k)$ fonctionnelle)
Soit $\sigma = \{V/x\}$.

- Prouvons $\forall U \in Vars(x_k)$, $\sigma U \in gen(a(\tau, x_k, V))$,

$$\forall U \in Vars(y_k),\ \sigma U \in a(\tau, y_k, V) \qquad (1)$$

- Si $V \in Vars(x_k) \Rightarrow \sigma V$ = x sous-terme de y_k à position u $\Rightarrow$ $x \in p(\tau, y_k, u)$
  $p(\tau, y_k, u) = p(\tau, x_k, u)$ par définition du type déduit par position
  $p(\tau, x_k, u) = a(\tau, x_k, V)$ par définition de D_k $\Rightarrow$ $\sigma V \in a(\tau, x_k, V)$

- Si $V \in Vars(y_k) \Rightarrow \sigma V = x \in gen(p(\tau, x_k, u)) = gen(p(\tau, y_k, u))$
  $V \in p(\tau, y_k, u) \Rightarrow p(\tau, y_k, u)$ complet $\Rightarrow gen(p(\tau, y_k, u)) = p(\tau, y_k, u)$
  $\Rightarrow \sigma V \in p(\tau, y_k, u) = a(\tau, y_k, V)$

  $r(\tau, x_k)$ est fonctionnelle par définition de D_k $\Rightarrow p(\tau, x_k, u) = a(\tau, x_k, V)$
  $\Rightarrow$ $\sigma V \in gen(a(\tau, y_k, V))$

- Pour les autres variables U de x_k et y_k, $\sigma U = U \Rightarrow$ CQFD

- $R_x = \bigcup_{U \in Vars(x_k)} \bigcup_{\tau_U \in r(\tau, x_k, U)} r(\tau_U, \sigma U) = \bigcup_{U \neq V} \bigcup_{\tau_U \in r(\tau, x_k, U)} r(\tau_U, U)) + \bigcup_{\tau_V \in r(\tau, x_k, V)} r(\tau_V, x)$

  $\forall \tau_V \in r(\tau, x_k, X), a(\tau, x_k, X) \ll \tau_V \Rightarrow r(\tau_V, x) = r(\tau_u, x) \subseteq r(\tau, y_k) \Rightarrow$

  $R_x = \{U : \tau_U \in r(\tau, x_k, U) \mid U \neq V\} + r(\tau_u, x) = r(\tau, x_k) / [U \neq V] + r(\tau_u, x)$
  Par hypothèse de récurrence et définition de D_k, on peut obtenir :
  $\min R_x \subseteq m(\tau, x_k) \cup r(\tau, y_k)$ fonctionnelle $\Rightarrow$
  $\forall X \in Vars(\sigma x_k), \& R_x(X) \neq \emptyset \qquad (2)$

- $R_y = \bigcup_{U \in Vars(y_k)} \bigcup_{\tau_U \in r(\tau, y_k, U)} r(\tau_U, \sigma U) = \bigcup_{U \in Vars(y_k)} r(\tau_U, \sigma U)$, $\tau_U = r(\tau, y_k, U)$ singleton
  $= \bigcup_{U \neq V} r(\tau_U, U) + r(\tau_V, x) = r(\tau, y_k) / [X \neq V] + r(\tau_V, x)$
  Par le mêmes raisonnement que pour R_x, on obtient :
  $R_y \subseteq m(\tau, x_k) \cup r(\tau, y_k)$ fonctionnelle (3)

- (1), (2) $\Rightarrow x_{k+1} = \sigma x_k \in gen(\tau)$ par propriété 3.1
  (1), (3) $\Rightarrow y_{k+1} = \sigma y_k \in \tau$ par propriété 3.2
  Par propriété 2.1 :
  $r(\tau, \sigma x_k) = R_x \Rightarrow m(\tau, \sigma x_k) = \min R_x \subseteq m(\tau, x_k) \cup r(\tau, y_k)$
  $r(\tau, \sigma y_k) = R_y \subseteq m(\tau, x_k) \cup r(\tau, y_k)$
  $\Rightarrow m(\tau, x_{k+1}) \cup r(\tau, y_{k+1}) \subseteq m(\tau, x_k) \cup r(\tau, y_k)$ fonctionnelle.

CQFD.

3.4 Condition de bon typage d'un programme

Cette partie présente la définition formelle de bon typage sur laquelle est construite la vérification de type décrite dans 2.7.4.

Pour la suite nous utilisons les notations :
- x pour le n-uplet $(x_1, \ldots, x_n)$,
- p(x) pour le prédicat n-aire p, et le n-uplet $(x_1, \ldots, x_n)$,
- $r(\tau, x)$, où $\tau = (\tau_1, \ldots, \tau_n)$, $x = (x_1, \ldots, x_n)$ pour la relation déduite étendue sur les n-uplets, définie par : $r(\tau, x) = \bigcup_i r(\tau_i, x_i)$.

3.4.1 Typage d'un programme

Soit **P** un programme Prolog, Pred l'ensemble de ses prédicats. **P** est dit **typé** par Π s'il est donné l'ensemble :

$\Pi = \{p^{\tau \to \tau'} \mid p \text{ prédicat n-aire}, \tau = (\tau_1, \ldots, \tau_n), \tau' = (\tau'_1, \ldots, \tau'_n), \tau' \ll \tau\}$

et $\forall p \in Pred, \exists p^{\tau \to \tau'} \in \Pi$.

Chaque $p^{\tau \to \tau'}$ s'appelle un ***profil*** du predicat p, τ est le ***type d'appel***, τ' est le ***type de retour*** des arguments de p, τ_i et τ'_i, $1 \le i \le n$, correspondant à l'argument i.

Un profil d'un prédicat signifie que si ce prédicat est appelé avec des arguments du type d'appel, en cas de succès, il va avoir comme solutions des termes du type de retour. Le type d'appel caractérise donc les arguments des appels du prédicat avant l'exécution, le type de retour caractérise leurs instances quand l'exécution réussit.

Un bon typage doit assurer que toute exécution du programme est cohérente par rapport à ces profils : les arguments d'un but sont toujours du type d'appel de son prédicat, et ses solutions, s'il réussit, sont de son type de retour.

3.4.2 Conditions de bon typage

Une clause typée par Π de **P** est de la forme :

$$p_0(x_0)^{\tau_0 \to \tau'_0} :- p_1(x_1)^{\tau_1 \to \tau'_1}, \ldots, p_n(x_n)^{\tau_n \to \tau'_n}$$

où $p_0(x_0) :- p_1(x_1), \ldots, p_n(x_n)$ une clause de **P**,

$p_i{}^{\tau_i \to \tau'_i} \in \Pi$, $0 \le i \le n$.

De façon intuitive, l'exécution de cette clause est la suivante. Après l'unification avec un but du même profil, les variables de la tête obtiennent leurs types déduits de τ_0 sur leurs positions. Avec les autres variables du corps, elles s'instancient dans les sous-buts et obtiennent les nouveaux types déduits de τ'_i pour leurs positions.

Informellement, pour que cette exécution soit cohérente par rapport aux profils déclarés, il faut que :

1/ Pour chaque sous-but, les types déduits des variables, en tenant compte des types de retour des sous-buts déjà effacés, satisfassent son type d'appel.
2/ Les types déduits des variables de la tête, en tenant compte des types de retour des sous-buts, satisfassent le type de retour du prédicat.

Nous ajoutons la troisième condition qui exprime notre solution pour la correction des assignations des variables.

3/ La suite des types déduits pour une variable soit décroissante pour la relation «.

Ces conditions sont établies formellement dans la définition suivante.

Définition :
La clause $p_0(x_0)$:– $p_1(x_1)$, …, $p_n(x_n)$ du programme **P** est dit ***bien typée*** par rapport à Π si $\forall\ p_0{}^{\tau_0 \to \tau'_0} \in \Pi$, $\exists\ p_i{}^{\tau_i \to \tau'_i} \in \Pi$, $1 \le i \le n$, tels que dans la clause typée par Π obtenue :

$p_0(x_0)^{\tau_0 \to \tau'_0}$:– $p_1(x_1)^{\tau_1 \to \tau'_1}$, …, $p_n(x_n)^{\tau_n \to \tau'_n}$

les conditions suivantes soient satisfaites :

C1/ $x_i \in gen(\tau'_i)$, $0 \le i \le n$, $m(\tau_0, x_0)$ est fonctionnelle,
$\forall$ V, $(m = \min \{j \mid V \in Vars(x_j)\} \wedge m \neq 0)$
$\Rightarrow r(\tau_m, x_m, V)$ est fonctionnelle et contient un type complet

C2/ Soit $A_0 = \{V : a(\tau_m, x_m, V) \mid m = \min \{j \mid V \in Vars(x_j)\}\}$
$A_i = A_{i-1}$ & $a(\tau'_i, x_i)$, $1 \le i \le n$
Alors :
$\forall\ V \in Vars(x_i)$, $1 \le i \le n$, $\forall\ \tau_V \in r(\tau_i, x_i, V)$, $A_{i-1}(V) \ll \tau_V$

$\forall V \in Vars(x_0), \forall \tau_V \in r(\tau'_0, x_0, V), A_n(V) « \tau_V$

Dans cette définition les ensembles A_i contiennent les types déduits pour chaque variable de la clause à chaque étape de son exécution. Ces types, appelés assignations de variable (cf. 1.3.2), sont calculés à partir des déclarations des prédicats de la tête et des sous-buts. Il en résulte la propriété suivante des assignations dans une clause bien typée.

Propriété 4 : $\forall V \in x_0, \ldots, x_n, A_i(V) « A_{i-1}(V), 1 \leq i \leq n$.

La condition C2/ permet à un prédicat d'accepter comme arguments des termes d'un type plus petit que son type d'entrée et comme solution des termes d'un type inclus dans son type de sortie.

L'obligation de la relation "plus modé" entre l'assignation d'une variable et le type déduit pour elle dans un appel, a pour but de satisfaire la propriété, exprimée dans la condition 3/, d'une exécution cohérente d'un programme typé.

Définition :
- Un but p(y) d'un programme typé par Π est dit ***bien typé*** par rapport à Π, s'il existe un profil $p^{\tau \to \tau'} \in \Pi$ tel que $y \in \tau$ et $r(\tau, y)$ est fonctionnelle.
- Un programme typé par Π est dit ***bien typé***, si son but est bien typé et pour tout prédicat du programme, toutes ses clauses sont bien typées par rapport à Π.

3.5 Correction d'un programme typé

Ainsi, un programme typé par Π est un programme avec des assertions pour ses prédicats. Ces assertions sur les propriétés de leurs arguments ne peuvent pas être exprimées par la sémantique déclarative du langage Prolog. Leur correction doit donc être prouvée, et le mécanisme essentiel utilisé pour les démontrer est une démonstration par induction sur la résolution.

3.5.1 Méthode inductive de preuves d'Assertions dans les programmes logiques

Cette méthode, établie par [Drabent 87], constitue un moyen de prouver que les assertions supposées sur chacun des prédicats, avant et après son appel, sont correctes dans toute exécution du programme, où le prédicat réussit. Cette

correction, qui ne garantit pas le succès, est donc partielle.

Une assertion est définie composée d'une pré-condition unaire π et une post-condition binaire π'. Par exemple l'assertion :

$\pi = p : x \in list(T)$, $\pi' = p : (x \in list(T), \sigma x \in list(T))$

signifie que :

- $\exists$ *T1* tel que l'argument du prédicat *p* est de type *list(T1)* à chaque appel de *p*, et
- si l'argument est de type *list(T2)* à l'appel de *p* et si l'exécution de cet appel réussit, la réponse est de type *list(T2)*.

Programme correct :
Un programme **P** avec assertions est dit ***correct***, si pour toute résolvante $R = Q_1, \ldots, Q_n$ d'une résolution de **P** :

- Q_1 satisfait la pré-condition de son prédicat
- Si R' est la première résolvante dans la résolution après R telle que $R' = \sigma(Q_2, \ldots, Q_n)$, alors $(Q_1, \sigma Q_1)$ satisfait la post-condition du prédicat de Q_1.

Soit une clause $p_0(x_0)$:– $p_1(x_1), \ldots, p_n(x_n)$, et un but :– $p_0(y)$. On appelle ***séquence d'évaluation*** pour cette clause et ce but, les substitutions $\rho_0, \ldots, \rho_n$ telle que :
$\rho_0 = pgu(x_0, y_0)$,
$\rho_i = \sigma_i \bullet \rho_{i-1}$, où σ_i est la substitution utilisée pour effacer $p_i(\rho_{i-1}x_i)$, $1 \le i \le n$.

La méthode est exprimée dans le théorème suivant.

Théorème de Drabent :
Pour qu'un programme **P** avec assertions soit correct, il est suffisant que pour toute clause $p_0(x_0)$:– $p_1(x_1), \ldots, p_n(x_n)$, où (π_i, π'_i) est l'assertion de p_i, $0 \le i \le n$, pour tout but :– $p_0(y_0)$ tel que y_0 satisfait π_0 et pour toute séquence d'évaluation $\rho_0, \ldots, \rho_n$, les conditions suivantes soient satisfaites :

1/ $\rho_0 x_1$ satisfait π_1
2/ Si $(\rho_0 x_1, \rho_1 x_1)$ satisfait π'_1, …, $(\rho_{k-1}x_k, \rho_k x_k)$ satisfait π'_k, alors $\rho_k x_{k+1}$ satisfait π_{k+1}, pour $1 \le k \le n-1$.
3/ Si $(\rho_0 x_1, \rho_1 x_1)$ satisfait π'_1, …, $(\rho_{n-1}x_n, \rho_n x_n)$ satisfait π'_n, alors $(\rho_0 y_0, \rho_n y_0)$ satisfait π'_0.

3.5.2 Un programme bien typé ne produit jamais d'erreur de type à l'exécution

Nous allons prouver dans cette partie le principe de bon typage du système : un programme bien typé appelle toujours un prédicat avec des arguments de son type d'appel.

Le lemme suivant, donnant les propriété des variables dans une clause bien typée, est le résultat principal qui conduit à la démonstration de la méthode.

Lemme 4 :
Soit :

- $p_0(x_0)^{\tau_0 \to \tau'_0} :- p_1(x_1)^{\tau_1 \to \tau'_1}, \ldots, p_n(x_n)^{\tau_n \to \tau'_n}$ une clause bien typée
- $:- p_0(y)^{\tau_0 \to \tau'_0}$ un but bien typé
- $\rho_0, \rho_1, \ldots, \rho_n$ une séquence d'évaluation pour ce but et cette clause.

Notons :

$P_k \equiv \rho_{k-1}x_k \in \tau_k$, $r(\tau_k, \rho_{k-1}x_k)$ est fonctionnelle
$P'_k \equiv \rho_k x_k \in \tau'_k$, $r(\tau'_k, \rho_k x_k)$ est fonctionnelle

Alors $\forall k, 0 \le k \le n$:

$(P_1, P'_1), \ldots, (P_k, P'_k) \Rightarrow$ Q1/ $\forall V \in Vars(x_0) \cup . . \cup Vars(x_k), \rho_k V \in A_k(V)$
Q2/ $\cup_{V \in Vars(x_0) \cup . . \cup Vars(x_k)} r(A_k(V), \rho_k V)$ est fonctionnelle

Démonstration : Par récurrence sur k.

* **k = 0 :**

- En utilisant le lemme d'unification sur x_0, y_0 et $\rho_0 = pgu(x_0, y_0)$, on obtient $\rho_0 x_0 \in \tau_0$ et $r(\tau_0, \rho_0 x_0)$ est fonctionnel

- $\rho_0 x_0 \in \tau_0 \Rightarrow \forall V \in Vars(x_0), \rho_0 V \in a(\tau_0, x_0, V)$ par propriété 2.2
  $a(\tau_0, x_0, V) \in m(\tau_0, x_0, V) = A_0(V)$ par définition de la clause bien typée
  $\Rightarrow$ Q1/ est prouvé

- $r(\tau_0, \rho_0 x_0) = \cup_{V \in Vars(x_0)} \cup_{\tau_V \in r(\tau_0, x_0, V)} r(\tau_V, \rho_0 V)$ par propriété 2.1
  $= \cup_{V \in Vars(x_0)} r(A_0(V), \rho_0 V)$ car $A_0(V) \ll \tau_V, \forall \tau_V \in \min r(\tau_0, x_0, V)$

$r(\tau_0, \rho_0 x_0)$ est fonctionnelle
$\Rightarrow$ Q2/ est prouvé

* **$k \rightarrow k+1$:**

Soit $y_{k+1} = \rho_k x_{k+1}$, $\rho_{k+1} = \sigma\rho_k$.

- Prouver que $\forall V \in Vars(x_0) \cup \ldots \cup Vars(x_{k+1}), \sigma\rho_k V \in A_k(V)$.
- Par hypothèse de récurrence pour $V \in Vars(x_0) \cup \ldots \cup Vars(x_k)$, et par définition de la clause bien typée pour $V \notin Vars(x_0) \cup \ldots \cup Vars(x_k) \Rightarrow$

$$\forall V \in Vars(x_0) \cup \ldots \cup Vars(x_{k+1}), \rho_k V \in A_k(V) \quad (1)$$

- En utilisant les propriétés 2.1, 2.2 sur $r(\tau_{k+1}, \rho_k x_{k+1})$ et l'hypothèse de récurrence Q2/ pour k, on obtient :

$$\forall X \in Vars(\rho_k V), \sigma X \in a(A_k(V), \rho_k V, X) \quad (2)$$

- En utilisant la propriété 2.1 sur $r(\tau_{k+1}, \sigma y_{k+1})$ fonctionnelle par $P'_{k+1} \Rightarrow$

$$R_V = \bigcup_{X \in Vars(\rho_k V)} \bigcup_{\tau_X \in r(A_k(V), \rho_k V, X)} r(\tau_X, \sigma X) \text{ est fonctionnelle} \quad (3)$$

- (1), (2), (3) $\Rightarrow$ Par 3.2,

$$\forall V \in Vars(x_0) \cup \ldots \cup Vars(x_{k+1}), \sigma\rho_{k+1} V \in A_k(V) \Rightarrow \text{CQFD}$$

- Si $V \notin Vars(x_{k+1})$, alors $A_{k+1}(V) = A_k(V) \Rightarrow \rho_{k+1} V = \sigma\rho_{k+1} V \in A_{k+1}(V)$
  Si $V \in Vars(x_{k+1})$, alors $\rho_{k+1} x_{k+1} \in \tau'_{k+1}$ par P_{k+1}
  $\Rightarrow$ $\rho_{k+1} V \in a(\tau'_{k+1}, x_{k+1}, V)$ par 1.2
  $\Rightarrow$ $\rho_{k+1} V \in A_k(V)$ & $a(\tau'_{k+1}, x_{k+1}, V) = A_{k+1}(V)$

  $\Rightarrow$ Q1/ est prouvé.

- Par propriété 3 des assignations $\Rightarrow$
  $\forall V \in Vars(x_0) \cup \ldots \cup Vars(x_{k+1}), A_{k+1}(V) \ll A_k(V) \Rightarrow$

$$\bigcup_{V \in Vars(x_0) \cup \ldots \cup Vars(x_{k+1})} r(A_{k+1}(V), \rho_{k+1} V) = \bigcup_{V \in Vars(x_0) \cup \ldots \cup Vars(x_k)} r(A_k(V), \rho_{k+1} V)$$

En utilisant 2.2 pour $\sigma\rho_k V \in A_k(V)$, $V \in Vars(x_0) \cup \ldots \cup Vars(x_{k+1})$, on obtient :

$$\bigcup_{V \in Vars(x_0) \cup \ldots \cup Vars(x_k)} r(A_k(V), \sigma\rho_k V) \subseteq r(\tau_{k+1}, \sigma y_{k+1}) + r(A_k(V), \rho_k V) / [X \notin \sigma y_{k+1}]$$

qui est fonctionnelle par P'_{k+1} et hypothèse de récurrence $\Rightarrow$

$$\bigcup_{V \in Vars(x_0) \cup \ldots \cup Vars(x_{k+1})} r(A_{k+1}(V), \rho_{k+1} V) \text{ est fonctionnelle}$$

$\Rightarrow$ Q2/ est prouvé.

Conséquence :
Un programme bien typé par rapport à Π est correct par rapport aux assertions (P, P') définie par :

$P = \quad y \in \tau, \quad r(\tau, y)$ est fonctionnelle
$P' = \quad \sigma y \in \tau', r(\tau', \sigma y)$ est fonctionnelle

pour tout $p \in Pred$, $p^{\tau \to \tau'} \in \Pi$, où y est un argument d'un appel de p, σ est la substitution utilisée pour l'effacement total de p(y).

Démonstration : Par méthode de Drabent.

Soit :

- $p_0(x_0)^{\tau_0 \to \tau'_0} :- p_1(x_1)^{\tau_1 \to \tau'_1}, \ldots, p_n(x_n)^{\tau_n \to \tau'_n}$ une clause bien typée
- $:- p_0(y)^{\tau_0 \to \tau'_0}$ un but bien typé
- $\rho_0, \rho_1, \ldots, \rho_n$ une séquence de substitutions qui conduit à l'effacement total de ce but à partir de cette clause.

- $P_k \equiv \quad \rho_{k-1}x_k \in \tau_k, \quad r(\tau_k, \rho_{k-1}x_k)$ est fonctionnelle
  $P'_k \equiv \quad \rho_k x_k \in \tau'_k, \quad r(\tau'_k, \rho_k x_k)$ est fonctionnelle

- Soit $(P_1, P'_1), \ldots, (P_k, P'_k)$, $k \geq 0$.
  $\forall V \in Vars(x_{k+1}), \rho_k V \in a(\tau_{k+1}, x_{k+1}, V)$:
  Si $V \in Vars(x_0) \cup \ldots \cup Vars(x_k)$, alors $\rho_k V \in A_k(V)$ par lemme-Q1/
  $A_k(V) \ll \tau_V, \forall \tau_V \in r(\tau_{k+1}, x_{k+1}, V) \Rightarrow A_k(V) \subseteq a(\tau_{k+1}, x_{k+1}, V) \Rightarrow$ CQFD

  Si $V \notin Vars(x_0) \cup \ldots \cup Vars(x_k)$, alors $\rho_k V = V \Rightarrow$ CQFD

- $\bigcup_{V \in Vars(x_{k+1})} \bigcup_{\tau_V \in r(\tau_{k+1}, x_{k+1}, V)} r(\tau_V, \rho_k V) = \bigcup_{V \in Vars(x_0) \cup \ldots \cup Vars(x_k)} r(A(V), \rho_k V) + \bigcup_{V \notin Vars(x_0) \cup \ldots \cup Vars(x_k)} r(\tau_V, V)$
  qui est fonctionnelle par lemme-Q2/
  $\Rightarrow \rho_k x_{k+1} \in \tau_{k+1}$, par 3.2 et $r(\tau_{k+1}, \rho_k x_{k+1})$ est fonctionnelle, par 2.1
  $\Rightarrow P_{k+1}$

(1) Si k = 0, on obtient : $p_1(\rho_0 x_1)^{\tau_1 \to \tau'_1}$ est bien typée ou satisfait P_1.
(2) Si $k \geq 1$, on obtient : $(P_1, P'_1), \ldots, (P_k, P'_k) \Rightarrow P_{k+1}$.
(3) Par le même raisonnement que pour $(P_1, P'_1), \ldots, (P_k, P'_k)$,
$(P_1, P'_1), \ldots, (P_n, P'_n) \Rightarrow \rho_n x_0 \in \tau'_0, r(\tau'_0, \rho_n x_0)$ est fonctionnelle.
$\Rightarrow P'_0$.

Par application directe du théorème de Drabent, le théorème suivant est prouvé.

Théorème :
Pour tout but p(y) appelé dans toute exécution d'un programme Π bien typé, il existe un profil $p^{\tau\to\tau'}$ tel que $y \in \tau$, $r(\tau, y)$ est fonctionnelle, et pour toute solution σy de ce but, $\sigma y \in \tau'$ et $r(\tau', \sigma y)$ est fonctionnelle.

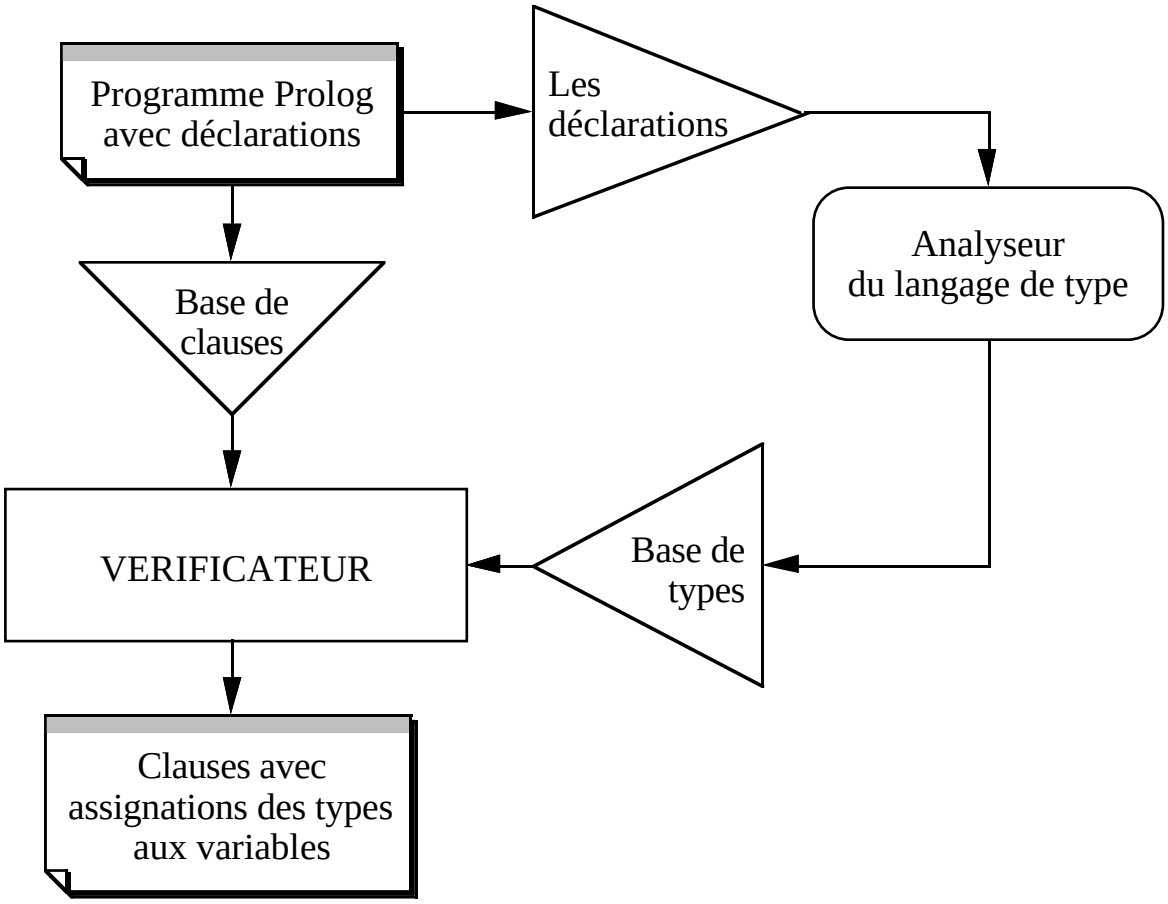
Avec ce théorème, est donc prouvée la validité du principe de typage du système : *un programme bien typé ne produit jamais d'erreur de type à l'exécution.*

Chapitre 4

MISE EN ŒUVRE DU SYSTEME

Dans les chapitres précédents, notre système de types a été présenté et prouvé du point de vue théorique. La mise en œuvre d'un prototype de ce système permet d'évaluer en pratique les profits qu'il apporte à la programmation en Prolog. Ce prototype est composé d'un analyseur du langage de type et d'un vérificateur du bon typage des programmes, construit sur la base de la théorie établie. Il a été écrit en Prolog, testé sur l'interprète C-Prolog et peut donc servir d'un analyseur statique de type pour les programmes Prolog.

Le fonctionnement du prototype peut être décrit par le schéma suivant :



Ce chapitre présente les algorithmes de vérification et les mécanismes essentiels utilisés pour les réaliser, ainsi que l'organisation interne du système.

4.1 Base de types du vérificateur

4.1.1 Traduction des déclarations

Les déclarations sont écrites par l'utilisateur dans la syntaxe décrite dans le chapitre 2. Elles sont analysées et traduites par le système en clauses Prolog. Le fonctionnement de l'analyseur du langage consiste à :

- Vérifier que toute déclaration définit bien un type.
- Traduire chaque définition de type en :
  - une clause de définition de ce type
  - des clauses définissant les expressions élémentaires qui le composent (cf. 2.3.1)
- Traduire chaque profil de prédicat en une clause de profil.
- Engendrer des clauses de définition pour les sous-énumérations utilisées dans les expressions des types et des profils déclarés.
- Engendrer des clauses, mémorisant les relations d'ordre "plus modé" entre les types, obtenues pendant l'analyse.

4.1.2 Enregistrements internes des définitions de type

Les clauses obtenues après le traitement des déclarations constituent, avec celles de types primitifs, la base de données du vérificateur de types. Cette base de clauses contient :

- les clauses des types primitifs
- les clauses de définition de type
- les clauses d'appartenance d'un foncteur à un type
- les clauses de profil de prédicat
- les clauses des relations entre les types : plus modé, inclus

Représentation interne des types

La dénotation interne des types dans le système est effectuée de la façon suivante :

- Tout type, utilisé dans les expressions de type, est représenté par un nom (et non par une énumération de ses composants)
- La définition d'un type est l'énumération des composants élémentaires (cf. 2.5.1), (les sous-types rencontrés dans une déclaration sont remplacés par leurs composants élémentaires)

Cette représentation des types permet d'exprimer d'une façon simple et directe les algorithmes de vérification de la relation "plus modé" et d'intersection des types. Dans cette représentation, les types récursifs et les types prédéfinis, ne peuvent être désignés que par un nom. Le fait de donner un nom à tous les types permet un traitement homogène dans la base de types. De plus, il n'y a plus aucune ambiguïté entre nom de type et nom de foncteur.

Exemple :

type $t_1 = f(a)$,
$t_2 = t_1 \mid b$.

a la dénotation interne $t_1 = f(t'_1)$, $t'_1 = a$, $t_2 = f(t'_1) \mid b$.

Clause de définition de type

Chaque type utilisé dans le système, a une clause de la forme suivante, qui définit tous ses éléments :

def_type(Nom, Mode, Foncteurs, Primitif, Inclusif, Domaine).

Nom : Le nom du type, formé d'un foncteur et des paramètres τ-variables

Mode : Le mode du type :
– si le type est complet
+ si le type est restreint
∗ si le type est clos

Primitif : L'union des sous-types primitifs du type :
- soit un nom de type primitif
- soit *vide* si le type n'en contient pas

Foncteurs : La liste des expressions élémentaires (cf. 2.3.1) du type, qui ne sont pas incluses dans Primitif.
Ces expressions sont classées par ordre alphabétique de leur foncteur, et leurs arguments sont des noms de type.

Inclusif :
- Soit la variable inclusive du type (cf. 2.4)
- Soit *vide*, si le type n'en a pas

Domaine : Les contraintes d'exclusion (cf. 2.4) pour toutes les τ-variables du type.
Le terme a la même structure que Nom du type et à la place de chaque argument se trouve ses contraintes.

L'ordre alphabétique des foncteurs permet des recherches plus rapides dans les traitements des types.

Les contraintes d'exclusion d'une définition de type servent à vérifier si une utilisation du type, quand ses paramètres sont remplacés par des expressions de type, est correcte.

Clause d'appartenance d'un foncteur à un type non primitif

élément(Foncteur, Type).

Type : un nom de type avec les paramètres τ-variables
Foncteur : un foncteur, dont les arguments sont des noms de type

Cette clause signifie que Foncteur est une expression élémentaire de Type.
Elle permet, lorsqu'il faut instancier une τ-variable à un type contenant un foncteur donné, de retrouver plus facilement les types contenant ce foncteur.

Clauses des types primitifs

Les types primitifs sont : *integer*, *float*, *dbreference*, *atom*, et toutes leurs unions possibles, par exemple : *number* (*integer* ∪ *float*), *atomic*(*atom* ∪ *number*).

Le système possède des faits suivants sur les types primitifs :

primitif(Nom). <1>

union(Nom1, Nom2, Nom). <2>

intersection(Nom1, Nom2, Nom). <3>

Nom, Nom1, Nom2 sont des noms de type primitif

<1> définit tous les noms des types primitifs
<2> définit l'union de deux types primitifs disjoints
<3> définit l'intersection non vide de deux types primitifs non inclus

Sur ces faits ont été construites d'autres clauses qui définissent l'union, l'intersection, la différence, l'inclusion, la relation "disjoint" (intersection vide), entre deux types primitifs quelconques.

Exemple : Les faits de types primitifs peuvent être

```
primitif(atom).
primitif(integer).
union(integer, float, number).
```

Clause de profil de prédicat

pred_profil(Entrée, Sortie, Contrainte).

Entrée : Le type d'appel, formé du foncteur de prédicat et des arguments qui sont des noms de type

Sortie : Le type de retour, formé du foncteur de prédicat et des arguments qui sont des noms de type

Contrainte : La liste des contraintes d'éxclusion pour toutes les τ-variables, utilisées dans Entrée, Sortie.
Chaque élément de la liste est un couple : une τ-variable avec la liste de ses contraintes.

Le paramètre Contrainte facilite la vérification d'une utilisation, dans une clause, d'un profil polymorphe, pendant la vérification de cette clause.

Clauses de relation entre les types

plus_modé(Type1, Type2). <1>

ss_types(Type, SsTypes). <2>

Type, Type1, Type2 sont des noms de type.
Type est un nom de type
SsTypes est une liste des noms de type

<1> signifie que Type1 est plus modé que Type2.
Elle est engendrée pendant la vérification de la relation "plus_modé" entre les types, et utilisée pour les autres vérifications.

<2> signifie que les types dans SsTypes sont inclus dans Type.
Elle est créée pendant l'analyse des définitions de type et sert à définir le mode # sur les types.

Exemple : Les déclarations

type *t1(T) = f(b, T),*
t2(T) = a | t1(T) | integer | T.
type *list(T) = [] | [T | list(T)],*
inlist(T) = + [] | [T | inlist(T)].
profil *p(a | b → +a),*
q(t1(T), t2(T)).
profil *append(inlist(T), inlist(T), list(T) → inlist(T)).*

sont transformées en clauses suivantes :

```
def_type($syst1,-,[b],vide,vide,$syst1).
def_type(t1(T),-,[f($syst1,T)],vide,vide,t1([])).
élément(f($syst1,T),t1(T)).

def_type(t2(T),-,[a,f($syst1,T)],integer,T,
        t2([integer,a,f(_,_)])).
élément(a,t2(T)).
élément(f($syst1,T),t2(T)).

def_type(list(T),-,[[],[T | list(T)]],vide,vide,list([])).
élément([],list(T)).
élément([T | list(T)],list(T)).

def_type(inlist(T),+,[[],[T|inlist(T)]],vide,vide,inlist([])).
élément([],inlist(T)).
élément([T | inlist(T)],inlist(T)).

def_type($syst2,-,[a,b],vide,vide,$syst1).
def_type($syst3,+,[a],vide,vide,$syst2).
pred_profil(p($syst1),p($syst2),[]).

pred_profil(q(t1(T),t2(T)),q(t1(T),t2(T)),
           [T-([integer,a,f(_,_)])]).

pred_profil(append(inlist(T),inlist(T),list(T)),
           append(inlist(T),inlist(T),inlist(T)),[T-([])]).
plus_modé(inlist(T),list(T)).
plus_modé($syst3,$syst2).

ss_types(t2(T),[t1(T)]).
```

`$syst1`, `$syst2`, `$syst3` sont des noms de type, créés par le système pour identifier respectivement les types définis par *b*, *a | b*, *+a*.

4.2 Le vérificateur de type

Cette partie du système a pour tâche de vérifier, conformément aux définitions présentées dans le chapitre 3, le bon typage des clauses d'un programme par rapport à tous les profils déclarés.

L'ordre d'exécution de la vérification totale d'un programme est le suivant :

```
Pour chaque prédicat P faire
début
   Soit Ps l'ensemble de profils de P
   Pour chaque clause Clause de P faire
   début
      Pour chaque profil Profil dans Ps faire
         verif_clause(Profil, Clause)
   fin
fin
```

Les algorithmes principaux du vérificateur sont :

- La vérification du bon typage d'une clause par rapport à un profil :
  verif_clause(Profil, Clause) qui réussit si la clause est bien typée et échoue dans le cas contraire.

- Le calcul de la relation déduite d'un type à un terme :
  relation_déduite(Type, Terme, R) qui réussit et calcule R = r(TypI, Terme) s'il existe une instance TypI de Type telle que Terme ∈ gen(TypI), et échoue sinon.

- La vérification de la relation "plus modé" entre deux types :
  plus_modé(Type1, Type2) qui réussit s'il existe des instances TypI1, TypI2 de Type1, Type2 telles que TypI1 « TypI2, et échoue sinon.

- Le calcul de l'intersection de deux types :
  inter_type(Type1, Type2, Type) qui calcule Type = Type1 & Type2.

4.2.1 Les variables de type

Pour les profils polymorphes, qui contiennent des paramètres, la vérification de bon typage d'une clause doit distinguer deux traitements différents sur les τ-variables.

D'après la définition, les conditions du bon typage d'une clause par rapport à un profil $p^{\tau\rightarrow\tau'}$ doivent être vérifiées pour toute instance de ce profil. Donc si τ est polymorphe, les τ-variables dans τ doivent être ***universellement quantifiées*** :
- Elles ne peuvent pas correspondre à des termes non variables de la clause.
- Une assignation contenant une τ-variable universelle, ne peut pas correspondre à une position dont le type est moins général que cette assignation.

Cela entraîne que les τ-variables du profil de la tête ne peuvent être instanciées à aucun type concret pendant la propagation des assignations des variables de la clause dans les sous-buts. Dans l'algorithme de la vérification d'une clause, les τ-variables du profil de la tête sont *figées* par des termes de la forme $T(K), où K est un nombre entier, et ne sont donc plus des variables Prolog dans les traitements ultérieurs.

Par contre, pour les sous-buts, il faut qu'il existe des instances de leurs profils satisfaisant les conditions de bon typage. Ainsi, si dans un profil $p_i^{\tau_i\rightarrow\tau'_i}$ d'un sous-but, τ_i est polymorphe, ses τ-variables sont ***existentiellement quantifiées***, et la vérification de la clause consiste à en trouver une valeur concrète. Dans notre algorithme d'analyse d'une clause, les solutions pour ces τ-variables sont déduites des termes utilisés dans les sous-buts de la clause, et des contraintes de la relation entre les assignations calculées.

D'un autre côté, chaque paramètre d'un type polymorphe doit avoir un domaine de définition, déterminé par les contraintes d'exclusion (cf. 2.4). Les valeurs trouvées pour les τ-variables pendant l'analyse d'une clause doivent donc satisfaire leurs contraintes d'exclusion. Cette vérification est effectuée, pour des raisons d'efficacité, une seule fois pour chaque sous-but de la clause, et non pas après chaque instanciation de τ-variable.

4.2.2 Vérification du bon typage d'une clause

L'analyse du typage d'une clause par rapport à un profil consiste à :
- trouver des profils pour ses sous-buts
- calculer les assignations des variables de clause
- vérifier les contraintes de la tête et des sous-buts

Les techniques effectuées dans l'algorithme pour réaliser ces tâches sont les suivantes :

• L'assignation A_i qui est l'ensemble $\{X_i : \tau_i \mid X_i$ variable de clause$\}$, est représentée par le paramètre de forme $a(T_1, \ldots, T_n)$, où *n* est le nombre de variables de la clause, T_i est l'assignation de la variable numéro *i*.

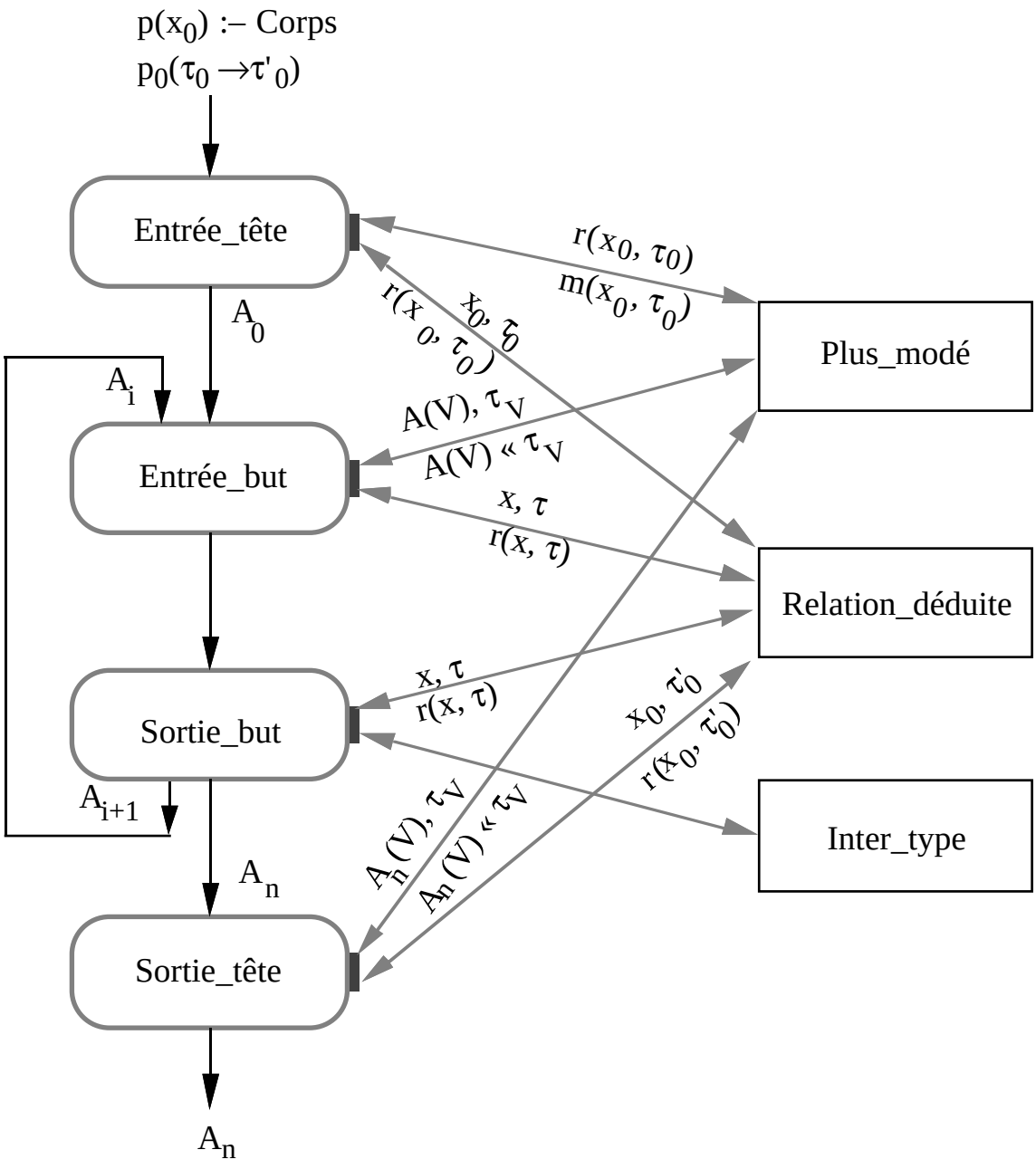
• La relation déduite d'un type sur un terme est représentée par une liste [i–τ, …], où *i*, qui est un entier, représente la variable numéro *i* et τ représente le type déduit pour une position de cette variable. Cette liste est triée sur *i* afin d'être exploitée plus efficacement.

• Pour les profils polymorphes des sous-buts, qui conduisent à essayer chacune des solutions possibles des τ-variables, et pour les sous-buts, qui ont plusieurs profils, le mécanisme de retour-arrière Prolog est utilisé. Ces retour-arrières déclenchés sur un échec de la vérification peuvent avoir lieu dans :

- la recherche des profils,
- la décomposition d'une τ-variable sur un terme,
- la vérification de la relation "plus modé" entre deux types

• L'aplatissement des structures syntaxiques "disjonctions", "if-then-else" (cf. 2.7.7), est effectué en même temps que l'analyse du corps d'une clause.

L’organisation générale de la vérification d’une clause est la suivante :



Algorithme : verif_clause(Profil, Clause)

Données : Profil = $p_0(\tau_0 \rightarrow \tau'_0)$
Clause = $(p_0(x_0)$:– Corps)

1 . Initialisation :

Numéroter les variables de la clause.
Créer l'assignation initiale Ass telle que pour toute variable V de la clause, Ass(V) est une τ-variable non instanciée.
Figer les τ-variables dans τ_0.

2 . Contrainte d'entrée de la tête :

Vérifier que la relation déduite modée $m(\tau_0, x_0)$ est fonctionnelle et calculer l'assignation des variables après la tête :
Si relation_déduite(τ_0, x_0, R) réussit, et
$\forall$ V, $\exists$ (V : τ_V) $\in$ R, plus_modé(τ_V, τ), $\forall$ (V : τ) $\in$ R, alors Ass(V) := τ_V
Sinon : **Echec**

3 . Vérifier le corps :

Transformer le corps, qui peut contenir des disjonctions, et des "conditions", en un ou plusieurs corps, qui ne contiennent qu'une conjonction des sous-buts.
Pour une conjonction des sous-buts, les vérifier séquentiellement par ordre.

Pour tout sous-but q(x), soit Ass l'assignation des variables juste avant q(x) :
- Si *q* est un prédicat d'ordre supérieur : *call*, *assert*, *setof*, *not*, …, ou le prédicat prédéfini "=", effectuer son algorithme de vérification spécifique avec *x* et Ass.
- Si *q* est un prédicat standard (utilisateur ou prédéfini) :

Pour tout profil $q(\tau \rightarrow \tau')$ non essayé,

Vérifier la contrainte d'entrée du sous-but :
relation_déduite(τ, x, R) réussit,
$\forall$ (V : τ) $\in$ R, plus_modé(Ass(V), τ).
Sinon : **Echec**

Calculer la nouvelle assignation Ass' des variables après q(x) :
Si relation_déduite(τ', x, R') réussit, alors
$\forall$ V, (V : τ_i) $\in$ R', 1≤i≤n, Ass'(V) := ($\&_i \tau_i$) & Ass(V)
Pour toutes les autres variables U, Ass'(U) := Ass(U)
Sinon : **Echec**

4. Vérifier la contrainte de sortie de la tête :

Si relation_déduite(τ'_0, x_0, R') réussit,
$\forall$ (V : τ) $\in$ R, plus_modé(Ass(V), τ), alors : **Succès**
Sinon : **Echec**

4.2.3 Calcul de la relation déduite

L'exécution de l'algorithme relation_déduite(Type, Terme, R) consiste à vérifier que Terme appartient à gen(Type), puis calculer r(Type, Terme), la relation déduite du type Type, sur le terme Terme (cf. 3.3). Donc si Type contient des τ-variables, l'algorithme doit chercher à instancier ces τ-variables.

Algorithme : relation_déduite(Type, Terme, R)

- Si Terme = V variable, R := {V : Type}
- Si Terme n'est pas une variable :
  - Si Type = T τ-variable, décomposer T sur Terme :
    Instancier T à TypeT non essayé, qui contient le foncteur principal de Terme, et que relation_déduite(TypeT, Terme, R) réussit.
  - Si Type est un type primitif, vérifier que Terme est de ce type primitif.
  - Si Type = *any*, Terme = $f(x_1, \ldots, x_n)$:
    Pour 1≤i≤n, relation_déduite(*any*, x_i, R_i).
    R est l'union de $R_1, \ldots, R_n$.
  - Si Type est défini dans la base, Terme = $f(x_1, \ldots, x_n)$:
    Vérifier que Type a un composant élémentaire $f(Type_1, \ldots, Type_n)$.
    Pour 1≤i≤n, relation_déduite($Type_i$, x_i, R_i).
    R est l'union de $R_1, \ldots, R_n$. **Succès**

- Pour tous les autres cas : **Echec**

Décomposition d'une τ-variable

D'après la définition de la fermeture par généralisation d'un type, si un terme *x* appartient au type gen(τ), alors :
- soit x = X une variable,
- soit x = $f(x_1, \ldots, x_n)$ et τ a un composant $f(\tau_1, \ldots, \tau_n)$ tel que x_1 appartient à τ_i, $1 \leq i \leq n$.

Donc pour décomposer la τ-variable T sur un terme dont le foncteur est *f*/n, la méthode du système consiste à prendre comme valeurs possibles de T les types suivants :

T1. tous les types Type, définis dans la base de types, qui contiennent le foncteur principal *f*/n, et les gen(Type).

T2. les types $+f(T_1, \ldots, T_n) \mid T$, et $f(T_1, \ldots, T_n) \mid T$, où T, T_1, …, T_n sont des τ-variables.

T3. les types +*any* et *any*.

Exemple : Avec la base de types, obtenue à partir des définitions

type $t_1 = a \mid f(t_1)$,
$t_2 = a \mid b$,
$t_3 = +f(+integer) \mid t_2$.

la relation déduite R = r(*T*, *f(X)*) donne des solutions :
T = t_1, *R* = {*X* : t_1},
T = t_3, *R* = {*X* : +*integer*},
T = *gen*(t_3), *R* = {*X* : *integer*},
T = *$syst1*($T_1$, T_2), *R* = {*X* : T_1},
T = *gen*(*$syst1*($T_1$, T_2)), *R* = {*X* : *gen*(T_1)},
où *$syst1*($T_1$, T_2) est le nom de type créé par le système pour identifier le type + *f*(T_1) | T_2.
T = +*any*, *R* = {*X* : +*any*}.
T = *any*, *R* = {*X* : *any*}.

La solution **T1** permet d'instancier la τ-variable T aux types dont les relations avec les autres ont été déjà établies. Les clauses d'appartenance d'un foncteur à un type, décrites dans § 4.2, ont pour but de réaliser cet algorithme. L'inconvénient est que la recherche sera longue, surtout si le foncteur du terme x appartient à beaucoup de types. Par exemple, dans la base ci-dessus, r(*T*, *a*) donne à *T* des solutions : *T* = t_1, *T* = t_2, *T* = *atom*, *T* = *a* | *T'*, *T* = *any*.

D'un autre côté, les types de **T2** et **T3** englobent toutes les solutions possibles de T. Cette méthode de recherche est donc suffisante pour contenir tous les cas de bon typage. Il en résulte aussi que l'algorithme du calcul de la relation déduite est correct.

4.2.4 Vérification de la relation "plus modé"

Les règles suivantes, déduites de la définition (cf. 3.2), permettent de vérifier la relation "plus modé" entre deux types. La définition des type par une énumération de composants élémentaires (cf. 2.5.1), est utilisée. Pour traiter les types récursifs, la vérification est accompagnée d'un ensemble H qui contient les paires de noms de type rencontrées pendant la vérification. Si la vérification ramène à deux types qui sont déjà rencontrés auparavant, la relation entre eux est considérée justifiée.

Règles de vérification
Soit τ_1 et τ_2 deux types définis respectivement par les énumérations Es_1 et Es_2. $\tau_1 \ll \tau_2$, si $Es_1 \ll Es_2 /_H$ (*Es_1 est plus modé que Es_2 par rapport à H*), où $H = \{(\tau_1,\tau_2)\}$.

Règle 1 : $(Es_1 \mid E_1 \ll Es_2 \mid E_2) /_H$ si $(Es_1 \ll Es_2) /_H$ et $(E_1 \ll E_2) /_H$
$(+E_1 \ll Es_2 \mid E_2) /_H$ si $(E_1 \ll E_2) /_H$

Règle 2 : $(E \ll E) /_H$
$(+E \ll E) /_H$

Règle 3 : $(f(\tau_{11}, \ldots, \tau_{1n}) \ll f(\tau_{21}, \ldots, \tau_{2n})) /_H$ si pour $1 \le i \le n$:
- soit (τ_{1i}, τ_{2i}) se trouve dans H,
- soit $(\tau_{1i} \ll \tau_{2i}) /_{H'}$, où $H' = H \cup \{(\tau_{1i}, \tau_{2i})\}$

L'algorithme plus_modé($Type_1$, $Type_2$) réalisé dans notre prototype consiste à vérifier la relation $Type_1 \ll Type_2$. Puisque les τ-variables universellement quantifiées sont figées (cf. 4.2.1), les τ-variables trouvées dans $Type_1$ et $Type_2$ sont existentiellement quantifiées et elles seront instanciées à des types respectant la relation.

La définition interne des types étant l'énumération de leurs composants élémentaires, cet algorithme est une réalisation directe des règles de vérification

ci-dessus. Il utilise une liste L qui joue le rôle de l'ensemble H dans ces règles. Dans L sont mémorisées les relations à justifier à partir de la relation initiale.

- Si la vérification initiale conduit à celle d'une relation qui se trouve déjà dans L, cette dernière est considérée correcte et ne sera plus traitée.
- Dans le cas contraire, elle est mémorisée dans L et sera effectivement vérifiée.

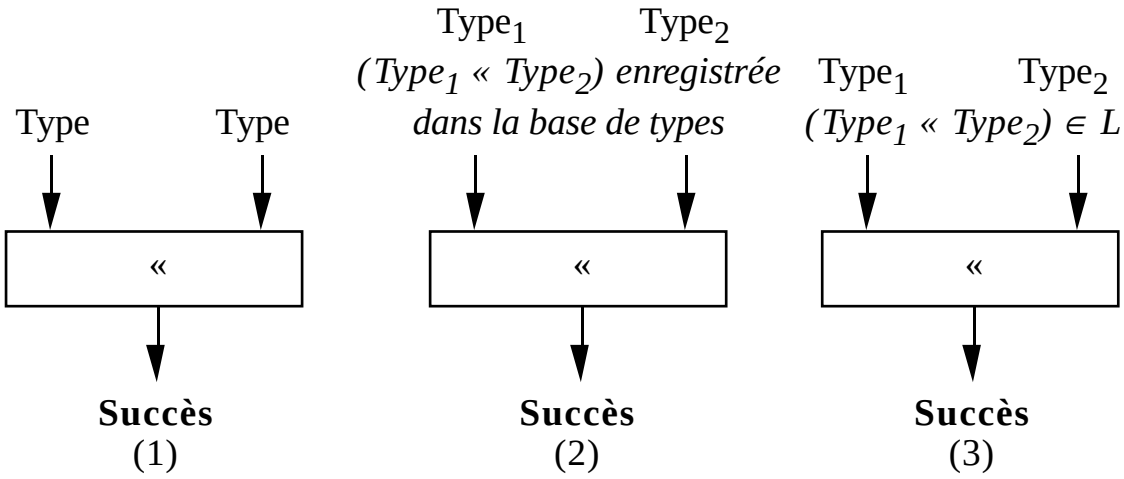
Cette liste joue, dans le cas des types récursifs, le rôle de retenir le nom des types à vérifier, et donc assure la terminaison de l'algorithme. Dans le cas des types non récursifs, elle permet de réutiliser les relations correctes déjà traitées.

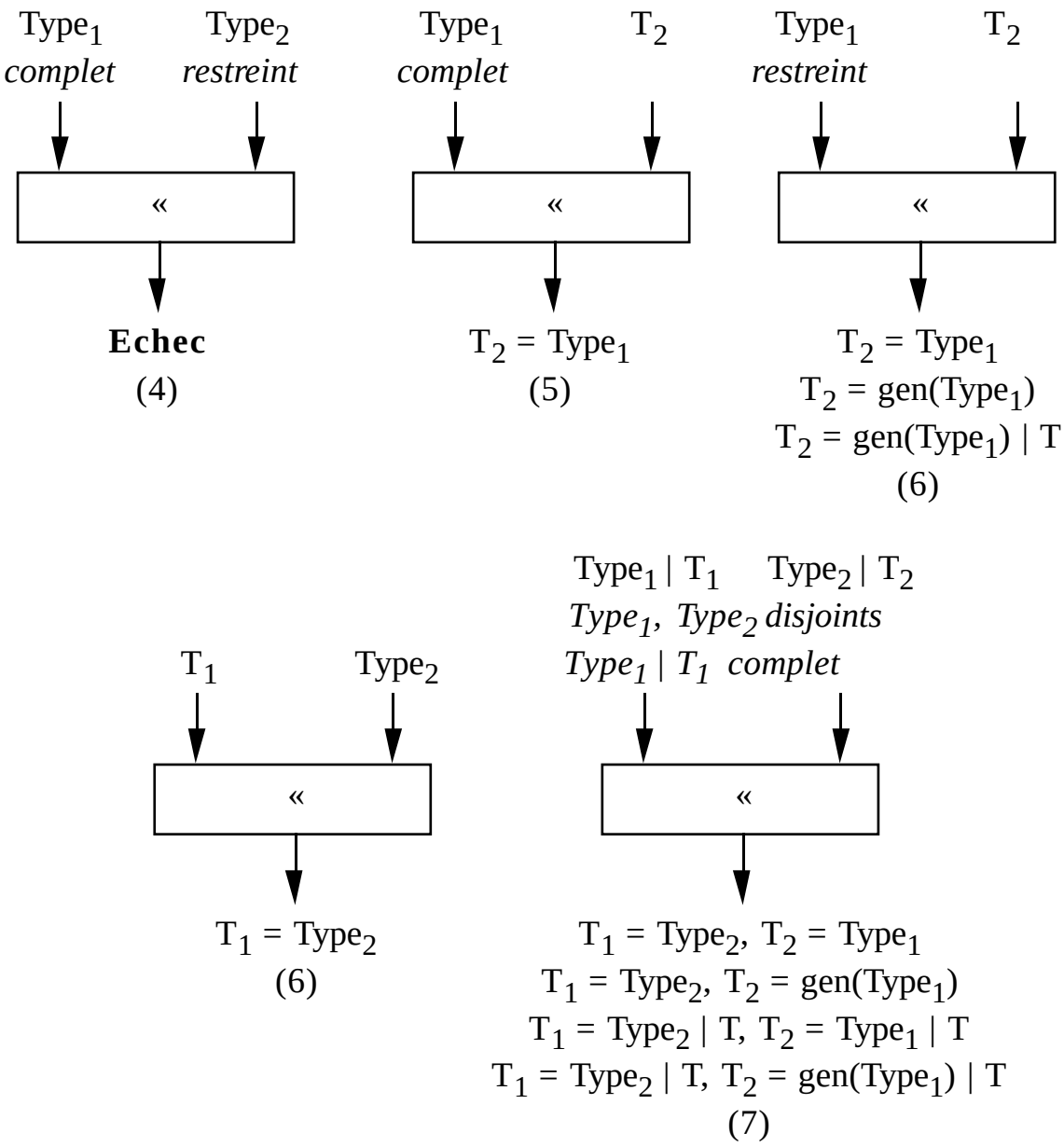
Initialement, L = [] et l'exécution de plus_modé($Type_1$, $Type_2$) déclenche celle de plus_modé($Type_1$, $Type_2$, L).

Algorithme : plus_modé($Type_1$, $Type_2$, L)

- Soit Es_1 l'énumération interne de $Type_1$ et Es_2 l'énumération interne de $Type_2$:
- Si pour tout $f(Type_{11}, \ldots, Type_{1n})$ de Es_1 tel qu'il y a $f(Type_{21}, \ldots, Type_{2n})$ de Es_2 : plus_modé($Type_{i1}$, $Type_{i2}$, L'), $1 \le i \le n$,
  L' = [plus_modé($Type_1$, $Type_2$) | L],
  les enlever de Es_1, Es_2.
  Sinon : **Echec**
- Enlever de Es_1 et Es_2, le type primitif commun le plus grand.
- Enlever de Es_1 et Es_2, leurs variables inclusives, si elles sont identiques.

- Les cas particuliers sont résolus par les règles suivantes :





Exemple : Soit les définitions

type $t_1 = a,$
$t_2 = + f(t_1, t_1) \mid a,$
$t(T) = atom \mid f(T, t_1).$

type $t_3 = + f(t_1, t_1) \mid g(t_1),$
$t_4(T) = f(t_1, t_1) \mid atom \mid T.$

La vérification plus_modé(t_2, $t(T)$) réussit et retourne $T = t_1$.
La verification plus_modé(t_3, $t_4(T)$) réussit et retourne $T = g(t_1)$.

Dans le cas (6) de l'algorithme, la τ-variable est instanciée à une seule parmi des solutions possibles de l'inéquation si $Type_2$ est restreint. Pourtant, provenant toujours d'un sous-but précédent ayant comme profil T → +T et comme argument une variable non instanciée, les inéquations de ces cas n'existent pas en réalité : Ces sous-buts ne réussissent jamais. Dans les autres cas, toutes les solutions possibles des τ-variables sont prises. L'algorithme "plus_modé" répond donc parfaitement au modèle souhaité de cette relation.

4.2.5 Intersection de deux types

Par définition (cf. 3.1), l'intersection de deux types définis respectivement par les énumérations Es_1 et Es_2, est le type défini par l'énumération Es, ($Es = Es_1 \& Es_2$), telle que :

- Es est restreinte ssi Es_1 ou Es_2 est restreinte.
- Es contient $f(\tau_1, \ldots, \tau_n)$ ssi Es_1 contient $f(\tau_{11}, \ldots, \tau_{1n})$, Es_2 contient $f(\tau_{21}, \ldots, \tau_{2n})$, et $\tau_i = \tau_{1i} \& \tau_{2i}$, $1 \leq i \leq n$.
- Es contient le type primitif Π ssi Es_1 contient Π_1, Es_2 contient Π_2, et $\Pi = \Pi_1 \cap \Pi_2$.
- Es contient la variable inclusive T ssi Es_1 et Es_2 contiennent T.

L'algorithme inter_type/3 calcule l'intersection de deux types donnés. Il utilise une liste L pour mémoriser les noms des intersections calculées pendant le calcul de l'intersection initiale. L'intersection de deux types qui se trouvent déjà dans L, sera remplacée par le nom assigné. Le rôle essentiel de cette liste est de traiter les types récursifs. En conséquence, le calcul de leur intersection ne va pas boucler, car le nombre d'intersections à calculer est fini. Initialement, L = [] et l'exécution de cet algorithme déclenche celle de inter_type($Type_1$, $Type_2$, Type, L).

Algorithme : inter_type($Type_1$, $Type_2$, Type, L)

- Si $Type_1$ et $Type_2$ sont le même type : Type := $Type_1$.
- Si $Type_1 \& Type_2 = Type_m$ est mémorisée dans la liste L : Type := $Type_m$.
- Si plus_modé($Type_1$, $Type_2$) est un fait de la base de types : Type := $Type_1$.
  Si plus_modé($Type_2$, $Type_1$) est un fait de la base de types : Type := $Type_2$.

- Soit Es_1 l'énumération interne de $Type_1$, Es_2 l'énumération interne de $Type_2$.
  Calculer l'énumération Es de Type :
  Si Es_1 et Es_2 sont complètes, Es est complète, sinon Es est restreinte.
  Es ne contient que les éléments suivants :
  - Tout $f(Type'_1, \ldots, Type'_n)$, où :
    $f(Type_{11}, \ldots, Type_{1n})$ est de Es_1, $f(Type_{21}, \ldots, Type_{2n})$ est de Es_2,
    intersection($Type_{1i}$, $Type_{2i}$, $Type'_i$, L'), $1 \leq i \leq n$,
    L' = [($Type_1$ & $Type_2$ = $Type_m$) | L].
  - Π, l'intersection des types primitifs de Es_1, Es_2, si $\Pi \neq \emptyset$.
  - La variable inclusive T, si Es_1 et Es_2 ont la même variable inclusive.

Exemple : Soit les définitions

type $t = a$,
$t_1 = a \mid b \mid f(t_2, t_1)$,
$t_2 = b \mid c \mid f(t_1, t)$.

intersection(t_1, t_2, *Type*) retourne $Type = t_{res}$, où $t_{res} = b \mid f(t_{res}, t)$.

4.2.6 Traitement du prédicat prédéfini "="

Les profils choisis dans le système pour le prédicat d'unification étant :

$(Type_1 \rightarrow Type_1 \,\&\, Type_2) = (Type_2 \rightarrow Type_1 \,\&\, Type_2)$

satisfaisant la contrainte qu'il existe un type $Type_0$ tel que $Type_1 \ll Type_0$ et $Type_2 \ll Type_0$, les valeurs de $Type_1$ et $Type_2$ ne peuvent être connues que dans chaque appel de ce prédicat. Elles sont déduites des arguments de l'appel au moment de son analyse. Les appels de ce prédicat doivent donc avoir un traitement spécifique.

Algorithme :
Données : But = ($x_1 = x_2$)
Assignation = Ass
Résultat : Profil = ($T_1 \rightarrow T_1 \& T_2$, $T_2 \rightarrow T_1 \& T_2$)
Assignation = Ass'

- Calculer la relation déduite $r(T_1, x_1) \cup r(T_2, x_2)$:
  relation_déduite((T_1, T_2), (x_1, x_2), R).
  Résoudre la contrainte d'entrée :
  $\forall (V : \tau) \in R$, plus_modé(Ass(V), τ).

  (Ces calculs instancient T_1 et T_2 qui étaient des τ-variables)

- Vérifier la contrainte du profil :
  plus_modé(T_1, T),
  plus_modé(T_2, T).
  Sinon : **Echec**

  (Cette vérification cherche à instancier la τ-vatiable T à un type qui satisfait ces relations)

- Calculer le type de retour du profil qui est l'intersection de T_1 et T_2 :
  inter_type(T_1, T_2, T_{retour}),

- Calculer la nouvelle assignation Ass' des variables après le sous-but :
  Si relation_déduite((T_{retour}, T_{retour}), (x_1, x_2), R') réussit, alors :
  $\forall$ V, $(V : \tau_i) \in R'$, $1 \leq i \leq n$, Ass'(V) := $(\&_i \tau_i)$ & Ass(V),
  pour toutes les autres variables U, Ass'(U) := Ass(U).
  Sinon : **Echec**

Exemple :

type $t_1 = a \mid b$,
$t_2 = + a$.
profil $p(t_1), q(t_2)$.

$p(X) :- X = a, q(X)$.

L'analyse du sous-but $X = a$ permet d'instancier T_1 à t_1, T_2 à t_2. X peut donc obtenir la nouvelle assignation t_2, qui satisfait ensuite le profil de *q/1*.

En conclusion de ce chapitre, signalons que les exemples testés sur le système implanté ont donné des résultats satisfaisants. Mis à part certains traitements qui limitent son efficacité, l'installation de ces algorithmes a pu permettre de mettre en œuvre un prototype qui répond bien à notre modèle théorique du système de types.

Chapitre 5

CONCLUSION

La présentation du système de types dans les chapitres précédents a illustré et précisé notre point de vue sur les types en Prolog, les solutions choisies pour résoudre les problèmes principaux d'un système de types, les concepts théoriques de base du système et les techniques utilisées pour l'implantation du prototype. Le but de ce dernier chapitre est de donner un bilan comparatif de notre système par rapport aux autres systèmes de types Prolog existants pour mettre en évidence les méthodes classiques que nous avons utilisées et nos solutions personnelles. Les deux concepts principaux d'un système de types : le principe de typage et la notion de type, sont tout d'abord analysés. Ensuite nous présentons certaines approches particulières de notre système, des problèmes qui ne sont pas encore complètement résolus, et des travaux envisagés dans le but d'améliorer le système.

5.1 Principe du typage

Notre système de types peut être classé parmi les systèmes déclaratifs : les spécifications de types sont entièrement fournies par l'utilisateur. Les incohérences du programme par rapport aux déclarations sont classifiées *erreurs de type* et sont donc indépendantes de la notion d'échec du programme. Cette distinction entre les deux notions permet de mieux localiser les erreurs dans un programme et de séparer les erreurs typographiques des erreurs de programmation.

Le bon typage défini dans notre système est une condition suffisante pour qu'un programme ne produise pas d'erreurs de type à l'exécution.

La donnée des profils de tous les prédicats permet une vérification locale à chaque clause, ce qui rend le système d'un coût raisonnable. Un système avec un minimum de reconstruction des spécifications (par exemple [Lakshman 91]) demande toujours une analyse globale des clauses.

Un point commun se trouve entre notre système et les systèmes [Mycroft 84] et [Dietrich 88] : nous n'avons pas redéfini la sémantique de Prolog. Les types dans notre système ne sont donc que des notions syntaxiques, optionnelles,

utilisées pour documenter le programme. Pourtant, comme nous l'avons déjà remarqué dans le §1.3.1, cela ne concerne que l'aspect formel et n'affecte pas le rôle du système dans la pratique. Le système peut détecter la plupart des erreurs de "bas niveau", si pénibles à débusquer en Prolog standard : les fautes de frappe qui déforment le nom d'une variable, d'un foncteur, d'un prédicat, l'oubli ou l'ajout d'un argument, … Il peut être utilisé comme frontal d'un compilateur pour exploiter les informations de type, et comme moyen de spécifier le fonctionnement des programmes. Il constitue une partie indépendante de la compilation et ne s'exécute que quand l'utilisateur le désire. Les déclarations ne conduisent à aucun changement dans l'exécution des programmes. L'exécution d'un programme est donc identique dans les deux cas : avec ou sans la vérification de types.

5.2 Notion de type

Pour traiter les inclusions des types, nous considérons qu'il existe un Univers de tous les termes, chaque type est un sous-ensemble de cet Univers et un terme peut appartenir à plusieurs types. La relation d'inclusion et les autres opérations sur les types comme sur les ensembles de termes, sont reconnues et exploitées. L'équivalence entre les types est déterminée par leur structure, à la différence du système [Mycroft 84], où il y a équivalence par noms, réalisée par l'unification des termes de type. L'utilisation de l'équivalence structurelle a deux avantages. Pour l'utilisateur, la définition des types à utiliser n'est pas très compliquée. Il est possible de donner des noms différents à un même type, et il ne faut pas se soucier de préciser les inclusions, qui peuvent être utilisées. Dans le système, cette équivalence permet de détecter toutes les inclusions possibles entre les types.

Il est établi que la vérification de types dans un système avec sous-types ne peut être effectuée qu'avec la connaissance du degré d'instanciation des termes. Les différents exemples ont montré cette liaison étroite entre la notion de type et la notion de mode. Dans un programme typé et correct, un argument doit être rester toujours dans son type et être de plus en plus instancié. Le mode est donc une information nécessaire pour justifier la correction d'un programme.

Dans son système de types, [Dietrich 88] a utilisé la notion de flot de données, pour définir le bon typage. Pourtant cette approche a un inconvénient : il faut un système de modes séparé pour garantir la correction des déclarations de modes utilisées.

Nous avons ainsi été amenés à introduire dans le système la notion de mode. Cependant la notion de mode dans notre système n'est pas séparée de la notion

de type comme dans [Dietrich 88] : elle est intégrée à la notion de type. Les types complets et les types restreints sont distingués. Les types restreints correspondent aux termes qui sont instanciés (partiellement ou entièrement). Les types complets correspondent aux termes qui peuvent être de tout degré d'instanciation. Par conséquent, à la différence de la plupart des auteurs, qui définissent un type par un ensemble de termes clos et des règles de dérivation pour les autres termes (cf. par exemple [Mycroft 84]), nous définissons directement un type comme un ensemble de termes, non nécessairement clos. Pour les types complets, ces deux définitions sont équivalentes, mais cela nous permet de définir les types restreints.

D'après nous, cette solution correspond mieux à l'approche déclarative choisie pour le système. Les modes sont ainsi déclarés dans les types par l'utilisateur et vérifiés par le système. Ils permettent de décrire plus précisément l'intention du programmeur. Si modes et types sont des notions indépendantes, il est impossible de décrire des ensembles de termes utiles, comme les listes semi-closes d'éléments non nécessairement clos (cf. 2.3.1).

La relation d'ordre entre les types, établie et utilisée dans le système pour vérifier la correction des déclarations, est la relation "plus modé", qui est la relation d'inclusion renforcée par une condition suffisante pour que le sous-type accepté soit consistant. Cette condition permet d'éliminer les erreurs d'inférence causées par des alias de variables, tout en conservant une analyse locale sur chaque clause. Le même problème est résolu, par exemple dans [Debray 88], par un algorithme d'analyse globale.

5.3 Variable inclusive. Négation.

La notion de variable inclusive dans les définitions de types est une particularité de notre système par rapport aux autres, qui permet d'augmenter le pouvoir d'expression du langage de types du système.

Une variable inclusive est par définition un paramètre sous-type d'un type polymorphe. Par exemple, dans les définitions de types suivantes, *T* et *T2* sont des variables inclusives :

```
type    t(T) = f(a) | T,
        surlist(T1, T2) = list(T1) | T2.
```

Puisqu'une variable inclusive est unique dans une énumération (cf. 2.4), un type τ contenant une variable inclusive est donc composé de deux parties : la variable

inclusive et une énumération déterminée. Par les contraintes d'exclusion (cf. 2.4), la variable inclusive est le complément de cette énumération. Par exemple, *T* est le complément de *f(a)* dans *t(T)*, *T2* est le complément de *list(T1)* dans *surlist(T1, T2)*.

Les variables inclusives permettent tout d'abord d'exprimer le profil des prédicats de test d'appartenance d'un terme à un type restreint τ. Ces profils acceptent en entrée des termes susceptibles d'être de type τ et ne retournent que des termes de type τ. Par exemple, les prédicats suivants :

```
is_f(f(_)).

is_list([]).
is_list([_|_]).
```

peuvent être déclarés par les profils :

```
profil    is_f( f(a) | T → +f(a) ),
          is_list( list(T) | T'→ +list(T) ).
```

Le prédicat *integer/1* peut être déclaré par :

```
profil    integer(integer | T → +integer).
```

Ce profil permet la spécification restreinte suivante de *flatten/2* (aplatissement d'un arbre d'entiers) :

```
type      inlist(T) = + [] | [T | inlist(T)],
          tree(T) = + inlist(tree(T)) | T.
profil    append(inlist(T), inlist(T), →inlist(T)),
          integer(integer | T → +integer),
          flatten(tree(integer), →inlist(integer)).

flatten([], []).
flatten([T| Ts], L)  :–  flatten(T, L1), flatten(Ts, L2), append(L1, L2, L).
flatten(A, L)  :–  integer(A), L = [A].
```

Puisque la variable inclusive peut exprimer le complément d'un type, elle peut servir à exprimer le profil de la négation d'un prédicat. Par exemple, on sait que *integer(X)* réussit si *X* est instancié à un entier, et donc *not integer(X)* réussit si *X* est une variable ou instancié à un terme qui n'est pas un entier. On peut donc exploiter le profil spécialisé suivant pour *not integer/1* :

profil *not integer(+integer | T → +T).*

qui signifie que si l'argument effectif de ce prédicat est un terme instancié, et si l'exécution réussit, cet argument doit appartenir à la partie du type qui ne contient pas *integer*. (Pour les paramètres effectifs qui peuvent être variables, le seul profil utilisable est *not integer(T)*).

Pourtant, ces spécifications de la négation ne peuvent être exploitées qu'avec des prédicats prédéfinis. Pour les prédicats définis par le programmeur, le seul traitement effectué actuellement sur un sous-but not p(x), étant donné que le profil de p est $p(\tau \rightarrow \tau')$, est de vérifier p(x) par rapport à τ.

Pour pouvoir donner des profils plus précis à la négation, une idée serait de vérifier que le type de retour de la négation est le complément du type de retour du prédicat en cours. Ce problème n'est résolu totalement dans aucun système, mais l'est partiellement dans des cas restreints.

Nous pensons qu'un travail possible dans ce système est de traiter la négation de l'unification not _=_. Ce développement promet une amélioration de la précision du système, car il permettrait la spécification polymorphe suivante de *flatten/2* :

```
type    inlist(T) = + [] | [T | inlist(T)],
        tree(T) = + inlist(tree(T)) | T.
profil  append(inlist(T), inlist(T), →inlist(T)),
        is_list(list(T) | T1 → +list(T)),
        flatten(tree(T), →inlist(T)).

flatten([], []).
flatten(A, L) :– not is_list(A), L = [A].
flatten([T| Ts], L) :– flatten(T, L1), flatten(Ts, L2), append(L1, L2, L).

is_list([]).
is_list([_|_]).
```

Ceci est actuellement impossible, faute du profil :

not is_list(list(T) | T1 → +T1).

5.4 Transfert d'instanciations. Effet d'aliasing.

La notion de type restreint permet non seulement de prouver la correction des programmes quand il y a explicitement des inclusions de types, mais aussi de spécifier le transfert des instanciations entre les différentes parties de la clause. Ce transfert d'instanciations est illustré dans les exemples 1 et 2. Les exemples 3 et 4 montrent, par contre, des programmes que l'on voudrait considérer comme bien typés, mais qui sont rejetés par le système par manque d'un traitement plus adéquat des modes.

Le transfert d'instanciations peut être exprimé explicitement dans les profils, ou implicitement par le prédicat d'unification prédéfini "=".

Exemple 1 :

```
type      inlist(T) = + [] | [T | inlist(T)].
profil    member(–T→T, inlist(T)),
          load(atom, inlist(+atom)),
          load_file(+atom).

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

load(F, L)  :–  member(F, L), load_file(F).
```

Dans le profil de *member/2*, *–T→T* signifie qu'en entrée le premier argument du prédicat peut être une variable non instanciée et qu'au retour, il sera de type *T* qui peut être restreint si les éléments du deuxième argument sont instanciés. Il y a donc un transfert de l'instanciation du deuxième argument au premier dans ce prédicat. Le transfert d'instanciations de *L* à *F* est ainsi effectué dans le sous-but *member(F, L)*.

Exemple 2 :

```
type      data = f(data) | a | b | c,
          simple = + a | b | c,
          struct = + f(data).
profil    analyse(data, simple), process1(simple), process2(struct).

analyse(X, M)  :–  X = M, process1(X).
analyse(X, _)  :– X = f(_), process2(X).
```

Le transfert d'instanciations de *M* à *X* dans la première clause permet d'assigner à *X* le type *simple* et puis d'accepter le sous-but *process1(X)* comme bien typé.

Le transfert d'instanciations de *f(_)* à *X* dans la deuxième clause permet d'assigner à *X* le type *struct* et puis d'accepter le sous-but *process2(X)* comme bien typé.

Ces possibilités d'expression ne peuvent pas couvrir tous les cas de transfert d'instanciations. Les alias entre les variables de la clause pendant l'exécution peuvent aussi conduire à ce phénomène, car le changement du type d'une variable implique celui des variables en alias avec elle. Pourtant cet *effet de l'aliasing* n'est pas pris en compte.

Exemple 3 :

type $t_1 = a \mid b \mid c,$
$t_2 = + a \mid b,$
$t_3 = + a.$
pred $p(t_1, t_2), q(t_2 \rightarrow t_3), r(t_3).$

$p(X, Y) :- X = Y, q(Y), r(X).$

Au début, *X* et *Y* sont unifiées. Après le retour de *q(Y)*, *Y* obtient le type t_3 et implicitement *X* est aussi de ce type. Pourtant, faute d'un traitement adéquat, le type assigné à *X* reste toujours t_2 et la clause est rejetée.

Le problème d'exploiter l'effet des alias ne se pose pas dans les systèmes de types non modés. Mais dans notre système, comme dans les systèmes de modes, les propriétés des arguments évoluent au fur et à mesure de l'exécution. L'information sur les alias entre les variables de clause permet donc de déduire des types plus précis des arguments. Cependant, cela nécessite une analyse globale d'un programme et se traduit par un allongement non négligeable du temps d'exécution du système.

C'est aussi pour cette raison que les structures incomplètes de données ne peuvent pas avoir toujours un typage utile dans les systèmes avec les modes. Un exemple intéressant est le suivant (utilisation de listes différentielles).

Exemple 4 :

```
type    list(T) = [] | [T | list(T)],
        inlist(T) = + [] | [T | inlist(T)],
        tree(T) = + inlist(tree(T)) | T.
profil  integer(integer | T → +integer).

flatten(Tree, L) :– flatten_df(Tree, L–[]).

flatten_df([], L–L).
flatten_df(Int, [Int | L]–L) :– integer(Int).
flatten_df([Left | Right], L1–L2) :– flatten_df(Left, L1–L),
                                     flatten_df(Right, L–L2).
```

est la plus optimale version du prédicat *flatten/2* qui utilise des listes différentielles. Il serait intéressant de pouvoir typer ce *flatten/2* par :

```
profil  flatten(tree(integer), list(integer)→inlist(integer)).
```

qui spécifie le deuxième argument par la liste semi-close. Ce profil exige la spécification suivante de *flatten_df/2* :

```
profil  flatten_df(tree(integer), list(integer)–list(integer)),
        flatten_df(tree(integer), list(integer)–inlist(integer) →
                                  inlist(integer)–inlist(integer)).
```

Mais le dernier profil ne respecte pas les conditions de bon typage pour la troisième clause de *flatten_df/2* : *L1* est du type *list(integer)* et reste de ce type quel que soit les profils choisis pour les sous-buts. En fin de clause, *L1* n'est donc pas du type *inlist(integer)*.

Il faudrait raisonner sur les trois clauses pour se rendre compte que *L1* et *L* sont liées après le premier but de la troisième clause de telle sorte que lorsque *L* est instancié dans le deuxième but, *L1* aussi.

5.5 Perspectives

Le prototype réalisé est encore incomplet, concernant la qualité et la pertinence des messages d'erreur. Ceux-ci ne sont parfois compréhensibles qu'aux utilisateurs ayant une bonne connaissance de l'algorithme de vérification. Une phase d'expérimentation serait souhaitable, permettant de constater les réactions d'un utilisateur en train de développer une application.

Plusieurs prolongements à ce travail sont possibles : une combinaison de déclarations et d'inférence et le traitement des prédicats d'ordre supérieur (cf. 2.7.9), en particulier la négation.

- Un problème qui commence à être étudié dans plusieurs systèmes, est de combiner la déclaration et l'inférence de types. Dans [Lakshman 91], les types des prédicats déclarés sont vérifiés par le système, les types des autres prédicats sont calculés à partir du programme source et les types définis.

Sur la base théorique construite, nous pourrions envisager un système, où l'utilisateur déclarerait les types de certains foncteurs et certains prédicats et le système ferait le reste, en déduisant les types des autres foncteurs et prédicats. Ce système exigerait une analyse globale du programme.

- Un autre problème est la spécification du type des objets d'ordre supérieur permettant une analyse homogène de tous les prédicats prédéfinis dans le système ou définis par l'utilisateur, standards ou d'ordre supérieur.

Ce problème n'a été réellement traité dans aucun système de types. Seul [Hanus 89] y apporte une solution partielle. Dans notre système, nous avons des algorithmes *ad hoc* pour ces prédicats. Ces algorithmes détectent si l'argument est une clause ou un but et effectuent la vérification correspondante.

Bibliographie

[**Bansai 88**]
Bansai, A.K. and Sterling, L.
"An Abstract Interpretation Scheme for Logic Programs based on Type Expression"
Proc. of International Conference on Fifthth Generation Computer Systems, Tokyo, 1988, p. 422-429.

[**Bruynooghe 82**]
Bruynooghe, M.
“Adding redundancy to obtain more reliable and more readable Prolog Programs”
Proc. of 1st International Conference on Logic Programming, Marseille, France, 1982, p. 129-133.

[**Bruynooghe 87a**]
Bruynooghe, M.
"A framework for the Abstract Interpretation of Logic Programs"
Report CW 62, Dept. of Computer Science, K.U. Leuven, Octobre 1987.

[**Bruynooghe 87b**]
Bruynooghe, M.
"Abstract Interpretation : Towards the global optimisation of PROLOG Programs"
Proc. of 4th Symposium on Logic Programming, San Francisco, USA, 1987, p. 192-204.

[**Cardelli 85**]
Cardelli, L. and Wegner, P.
“On understanding Types, Data Abstraction, and Polymorphism”
ACM Computing Survey, Vol 17, N° 4, December 1985, p. 471-522.

[**Colmerauer 82**]
Colmerauer, A.
“Prolog II : Manuel de référence et modèle théorique”
Groupe Intelligence Artificielle ERA CNRS 363, Mars 1982.

[**Debray 88**]
Debray, S. and Warren, D.
“Automatic mode inference for Logic Programs”
Journal of Logic Programming, N° 5, 1988, p. 207-229.

[**Dietrich 88**]
Dietrich, R.
“Modes and Types for Prolog”
Proc. of 2nd European Symposium on Programming, 1988, LNCS N° 300, Springer-Verlag, p. 79-93.

[**Drabent 87**]
Drabent, W. and Maluszynski, J.
“Inductive Assertion Method for Logic Programs”
Proc. TAPSOFT, Pisa, Italy, March 1987, LNCS N° 250, Springer-Verlag, p. 167-181.

[**Fruhwirth 88**]
Fruhwirth, T.W.
"A Type Language for Prolog and its Application to Type Inference"
Computational Intelligence 88, Milan, Italy, September 1988, p. 27-39

[**Fujita 88**]
Fujita, H.
"Abstract Interpretation and Partial Evaluation of Prolog Programs"
ICOT Technical Memorandum TM-0484, March 1988.

[**Hanus 89**]
Hanus, M.
“Horn Clause Programs with Polymorphic Types : Semantics and Resolution”
TAPSOFT 89, LNCS N° 352, Springer-Verlag, p. 225-240.

[**Kanamori 85**]
Kanamori, T. and Horiuchi, K.
"Type Inference in PROLOG and its Application"
Proc. of 9th International Joint Conference on Artificial Intelligence, Los Angeles California, 1985, p. 704-707.

[**Kanamori 87**]
Kanamori, T. and Kawamura, T.
"Analyzing Succes Patterns of Logic Programs by Abstract Hybrid Interpretation"
ICOT Technical Report TR-279, 1987.

[**Kluzniak 87**]
Kluzniak, F.
“Type Synthesis for Ground Prolog”
Proc. of 4th International Conference on Logic Programming, Melbourne, Australia, 1987, p. 788-816.

[**Lakshman 91**]
Reddy, U. and Lakshman, T.K.
“Typed Prolog : A Semantic Reconstruction of the Mycroft-O’Keefe Type System”
Proc. of International Symposium on Logic Programming, San Diego, USA, 1991, p. 202-217.

[**Landais 88**]
Landais, G.
“Contribution à la compilation de Prolog par évaluation partielle”
Thèse, Université de Rennes I, France, December 1988.

[**Lloyd 87**]
Lloyd, J.W.
“Foundations of Logic Programming”
Second, Extended Edition, New York, Springer-Verlag, 1987.

[**Mellish 86**]
Mellish, C.S
"Abstract Interpretation of Prolog Programs"
Proc. of 3rd International Conference on Logic Programming, London, UK, July 1986, LNCS N° 225, Springer-Verlag, p. 463-474.

[**Milner 78**]
Milner, R.
“A theory of type polymorphism in programming”
Journal of Computer and System Science, December 1978, Vol. 17, p. 348-375.

[**Mishra 84**]
Mishra, P.
“Towards a Theory of Types in Prolog”
Proc. IEEE International Symposium on Logic Programming, Atlanta City, USA, 1984, p. 289-298.

[**Mishra 85**]
Mishra, P. and Reddy, U. S.
“Declaration-free type checking”
12th ACM Symposium on Principles of Programming Languages, New Orleans Louisana, USA, January 1985, p. 7-21.

[**Mycroft 84**]
Mycroft, A. and O’Keefe, R.A.
“A Polymorphic Type System for Prolog”
Artificial Intelligence Vol. 23, N° 3, August 1984, p. 295-307.

Egalement dans : Research Paper N° 211, Dept. of Artificial Intelligence, University of Edinburgh, 1983.

[**Naish 87**]
Naish, L.
“Specification = Program + Types”
Proc. of 7th Conference on Foundations of Software Technology and Theoretical Computer Science, Pune, India, December 1987, p. 326-339.

[**Oudot 87**]
Oudot, O.
“Utilisation des modes directionnels dans la résolution”
Thèse, LGI-IMAG, Grenoble, France, November 1987.

[**Pyo 89**]
Pyo, C. and Reddy, U.
“Inference of polymorphic types for logic programs”
Proc. of NACLP, 1989, p. 1115-1134.

[**Somogyi 87**]
Somogyi, Z.
"A system of precise modes for Logic Programs"
Proc. of 4th International Conference on Logic Programming, Melbourne, Australia, June 1987, p. 769-787.

[**Van Roy 86**]
Van Roy, P.
“Improving the execution speed of compiled Prolog with modes, clauses selection, and determinism”
CW Report N° 51, Dept. of Computer Science, November 1986.

[**Warren 77**]
Warren, D.H.
"Implementing Prolog-Compiling Predicate Logic Programs"
R.R. 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, Mai 1977.

[**Xu 88**]
Xu, J. and Warren, D.S.
"A type inference system for Prolog"
Proc. of 5th International Conference on Logic Programming, Seatlle, USA, 1988, p. 604-619.

[**Yardeni 87**]
Yardeni, J. and Shapiro, E.
"A type system for logic programs"
In E. Shapiro editor, Concurrent Prolog, Vol 2, p. 211-244, MIT Press 1987.

[**Zobel 87**]
Zobel, J.
"Derivation of Polymorphic Types for Prolog Programs"
Proc. of 4th International Conference on Logic Programming, Melbourne, Australia, June 1987, p. 817-838.

Annexe

LES PROFILS PREDEFINIS

```
type  t_atom(T) = atom / T.
type  t_atomic(T) = atomic / T.
type  t_int(T) = integer / T.
type  t_number(T) = number / T.
type  t_prim(T) = primitive / T.
type  t_dbref(T) = dbreference / T.
type  t_float(T) = float / T.
type  int_expr = (int_expr+int_expr) / (int_expr-int_expr)
              / (int_expr*int_expr)
              / {int_expr/int_expr} / (int_expr mod int_expr)
              / (int_expr/\int_expr) / (int_expr\/int_expr)
              / (int_expr<<int_expr) / (int_expr>>int_expr)
              / {+int_expr} / (-int_expr) / integer.
type  list(T) = [] / [T | list(T)].
type  inlist(T) = + [] / [T | inlist(T)].
type  couple(K,T) = (K-T).
type  expanse = off / on.
type  t_expanse(T) = expanse / T.
type  ord_alphabet = '<' / '=' / {'>'}.
type  leashing_mode = + full / half / loose / off / tight / integer.
type  pred_specif = + {atom/integer} / atom.
type  op_type = + fx / fy / xf / xfx / xfy / yf / yfx.

profil      abolish(+atom, +integer).
profil      abort.
profil      arg(+integer, any, any).
profil      atom(atom > +atom).
profil      atom(t_atom(T) > +atom).
profil      \+ atom(+t_atom(T) > +T).
profil      atomic(atomic > +atomic).
profil      atomic(t_atomic(T) > +atomic).
profil      \+ atomic(+t_atomic(T) > +T).
profil      break.
profil      clause(any, any).
profil      clause(any, any, dbreference > +dbreference).
profil      close(+atom).
profil      compare(ord_alphabet > +ord_alphabet, T, T).
profil      consult(+atom).
profil      current_atom(atom > +atom).
profil      current_functor(atom > +atom, any).
profil      current_predicate(atom > +atom, any).
profil      db_reference(dbreference > +dbreference).
profil      db_reference(t_dbref(T) > +dbreference).
profil      \+ db_reference(+t_dbref(T) > +T).
profil      debug.
profil      debugging.
```

```
profil      display(T).
profil      erase(+dbreference).
profil      erased(+dbreference).
profil      expanded_exprs(expanse, expanse).
profil      expanded_exprs(t_expanse(T), expanse).
profil      expand_term(any, any).
profil      exists(atom > +atom).
profil      fail.
profil      fileerrors.
profil      functor(any > +any, +atom, +integer).
profil      functor(+T, atom > +atom, integer > +integer).
profil      get(integer > +integer).
profil      get(t_int(T) > +integer).
profil      get0(integer > +integer).
profil      get0(t_int(T) > +integer).
profil      halt.
profil      instance(+dbreference, any).
profil      integer(integer > +integer).
profil      integer(t_int(T) > +integer).
profil      \+ integer(+t_int(T) > +T).
profil      (t_int(T) > +integer) is +int_expr.
profil      (integer > +integer) is +int_expr.
profil      keysort(inlist(couple(K,T)), >inlist(couple(K,T))).
profil      leash(leashing_mode).
profil      length(list(T), integer > +integer).
profil      listing.
profil      listing(pred_specif).
profil      name(atom > +atom, inlist(+integer)).
profil      name(+atom, >inlist(+integer)).
profil      nl.
profil      nodebug.
profil      nofileerrors.
profil      nonvar(T > +T).
profil      nospy.
profil      nospy pred_specif.
profil      number(number > +number).
profil      number(t_number(T) > +number).
profil      \+ number(+t_number(T) > +T).
profil      op(op_type, +integer, +atom).
profil      primitive(primitive > +primitive).
profil      primitive(t_prim(T) > +primitive).
profil      \+ primitive(+t_prim(T) > +T).
profil      print(T).
profil      prompt(atom > +atom, +atom).
profil      put(+int_expr).
profil      read(any).
profil      reconsult(+atom).
profil      recorda(+any, any, dbreference > +dbreference).
profil      recorda(+any, any, t_dbref(T) > +dbreference).
profil      recorded(+any, any, dbreference > +dbreference).
profil      recorded(+any, any, t_dbref(T)> +dbreference).
profil      recordz(+any, any, dbreference > +dbreference).
profil      recordz(+any, any, t_dbref(T) > +dbreference).
profil      rename(+atom, +atom).
profil      repeat.
```

```
profil      retract(any).
profil      save(+atom).
profil      see(+atom).
profil      seeing(atom > +atom).
profil      seen.
profil      sh.
profil      skip(+int_expr).
profil      sort(inlist(T), >inlist(T)).
profil      spy pred_specif.
profil      statistics.
profil      system(+atom).
profil      tab(+int_expr).
profil      tell(+atom).
profil      telling(atom > +atom).
profil      told.
profil      trace.
profil      true.
profil      ttyget0(integer > +integer).
profil      ttyget0(t_int(T) > +integer).
profil      var(T).
profil      write(T).
profil      writeq(T).
profil      'LC'.
profil      'NOLC'.
profil      !.
profil      +number < +number.
profil      +number =< +number.
profil      +number > +number.
profil      +number >= +number.
profil      (any > +any) =.. inlist(any).
profil      +any =.. (>inlist(any)).
profil      T==T.
profil      T\==T.
profil      T@<T.
profil      T@=<T.
profil      T@>T.
profil      T@>=T.
```