MASTER EN SCIENCES INFORMATIQUES

Étude d'un modèle d'interprétation abstraite des langages logiques concurrents avec contraintes

Cordier, Catherine

Award date:
1994

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'informatique
Rue Grandgagnage, 21
B-5000 NAMUR

Etude d'un modèle d'interprétation abstraite des langages logiques concurrents avec contraintes.

Catherine Cordier

Mémoire présenté en vue d'obtenir le grade de
Licencié et Maître en Informatique

Promoteur: Baudouin Le Charlier

Année académique 1993-1994

Résumé

Ce mémoire aborde l'interprétation abstraite des langages logiques concurrents avec contraintes. D'une part, nous présentons ces langages de façon détaillée. Brièvement, plusieurs agents sont exécutés en parallèle et interagissent en ajoutant des contraintes dans un ensemble global de contraintes, et en testant si une contrainte donnée est impliquée par cet ensemble. D' autre part, nous présentons un modèle d'interprétation abstraite pour ces langages. L'interprétation abstraite est une méthode d'analyse qui consiste à exécuter un programme sur un domaine particulier, dit abstrait car il abstrait seulement certaines propriétés pertinentes du domaine concret d'exécution. Le modèle que nous présentons est basé sur l'abstraction des contraintes d'un programme. Nous discutons des possibilités du modèle et examinons de plus près son application au cas de l'analyse de blocage.

Abstract

This work deals with abstract interpretation for concurrent constraint languages. First, we present these languages in detail. Briefly, multiple agents run concurrently and interact by adding constraints in a global set of constraints, and by checking wether a given constraint is entailed by this set. Secondly, we present a framework for abstract interpretation for these languages. Abstract interpretation is a general scheme for analysis that consists in executing a program on some special domain, called abstract because it abstracts only some relevant properties of the concrete domain of computation. The framework we present is based on abstracting the constraints of programs. We discuss the framework's possibilities and examine its application to deadlock detection.

Au terme de ce travail je voudrais remercier les personnes qui m'ont apporté leur aide et leur soutient durant cette année.

Je remercie tout particulièrement monsieur Le Charlier pour ses nombreux conseils et le temps consacré à suivre ce travail.

Je remercie monsieur Codognet pour son accueil à l'INRIA et sa contribution à ce travail.

Je remercie également l'ensemble des chercheurs du projet CHLOE pour leur accueil à l'INRIA et leur collaboration.

Enfin, merci à tout ceux qui m'ont apporté leur soutient, à mes compagnons d'étude et à mes proches.

Table des matières

Introduction

La programmation logique est apparue au début des années 70. Elle est issue de recherches sur la démonstration automatique de théorèmes. Il est apparu que la logique du premier ordre, qui était alors essentiellement utilisée dans un but déclaratif, pouvait être utilisée comme langage de programmation. Et le premier interpréteur Prolog (PROgramming in LOGic) fut implémenté. Ensuite nous avons vu apparaître de nombreuses versions et extensions de Prolog. Parmi celles-ci, nous trouvons en particulier les langages logiques avec contraintes (CLP) et les langages logiques concurrents. Les premiers étendent Prolog avec un mécanisme de résolution de contraintes qui offre un langage plus souple et élargit l'éventail des domaines d'applications. Les seconds introduisent des notions de concurrence, d'exécution parallèle, et offrent des mécanismes de synchronisation. Aujourd'hui nous voyons apparaître les langages logiques concurrents avec contraintes (Langages CC). Ces derniers mêlent à la fois contraintes et concurrence. Ainsi, ils offrent à la fois la souplesse et la puissance de la résolution de contraintes, et les mécanismes de parallélisme et de synchronisation des langages concurrents. C'est à ces langages que nous nous intéressons ici.

L'interprétation abstraite est un modèle général pour l'analyse statique des programmes. Elle consiste essentiellement à exécuter le programme sur un domaine particulier, dit abstrait, car il abstrait seulement certaines propriétés pertinentes du domaine concret d'exécution. Bon nombre de travaux ont été effectués dans ce domaine, en particulier pour l'analyse des programmes logiques. Aussi, nous nous intéressons à l'application de ces idées au cas des langages concurrents avec contraintes.

Ce travail est essentiellement le résultat de nombreuses lectures. D' une part nous y faisons le point sur de nombreux articles qui, pour la plupart, sont très théoriques et très peu détaillés. Aussi, la présentation qui en est faite ici est le fruit d'un long travail. A l'aide d'explications, parfois d'interprétations, d'exemples et de commentaires tout à fait personnels, nous espérerons avoir rendu ces théories plus abordables.

D' autre part, nous rapportons également dans ce travail les aboutissements du stage effectué auprès de monsieur P. Codognet à l' Institut National de Recherche en Informatique et Automatique (INRIA) à Rocquencourt en France. Ce stage consistait à étudier le modèle d'interprétation abstraite des langages logiques concurrents avec contraintes que nous présentons dans le dernier chapitre de ce travail. Cette étude devait se diriger plus particulièrement vers l' application de ce modèle au cas de l'analyse de blocage. Finalement, cette étude a soulevé plus de problèmes que de solutions. Nous exposons les résultats de nos recherches sur ce sujet dans la suite de ce même chapitre.

Ce travail est divisé en trois chapitres. Le premier est introductif. Il consiste en l'exposé des définitions préliminaires et en un rappel sur la programmation logique. Le second aborde les langages logiques concurrents avec contraintes. Il consiste en une présentation détaillée de la sémantique de ces langages. Le troisième chapitre traite de l'interprétation abstraite de ces langages. Il consiste plus particulièrement en la présentation et la mise en oeuvre du modèle d'interprétation abstraite auquel nous nous sommes intéressé pendant le stage.

Chapitre 1: La programmation logique

Dans ce premier chapitre, nous introduisons les langages de programmation logique et définissons les notions de bases. D'abord, nous donnons une série de définitions relatives à la logique du premier ordre sur base de l'ouvrage de J. W. Lloyd dans [9]. Ensuite nous présentons brièvement le langage Prolog sur base des notes du cours de B. Le Charlier [7]. Puis nous présentons brièvement les langages logiques avec contraintes (CLP) sur base de l'article [5] de C. Lassez. Nous terminons avec quelques définissons complémentaires relatives aux points fixes à nouveau basées sur [9].

1. La logique du premier ordre:

1.1. Théorie du premier ordre:

Une théorie du premier ordre est composée de:

- un alphabet
- un langage du premier ordre
- un ensemble d'axiomes
- un ensemble de règles d'inférences.

Un langage du premier ordre consiste en l'ensemble des formules bien formées de la théorie. Les axiomes consistent en un sous-ensemble de formules du langage. Les règles d'inférences sont des mécanismes qui permettent de dériver des théorèmes à partir des axiomes de la théorie.

Dans ce qui suit, nous introduisons la syntaxe des formules bien formées d'une théorie du premier ordre.

Un alphabet consiste en sept classes de symboles:

- les variables
- les constantes
- les symboles de fonctions
- les symboles de prédicats
- les connecteurs
- les quantificateurs
- les symboles de ponctuations.

Les trois dernières classes sont les mêmes pour tout alphabet, alors que les autres peuvent varier. Par convention de notation, les constantes débutent toujours par une minuscule et les variables par une majuscule. Les connecteurs sont $\neg$, $\wedge$, $\vee$, $\Rightarrow$ et $\Leftrightarrow$. Les quantificateurs sont $\forall$ et $\exists$. Et les symboles de ponctuations sont '(', ')', et ','.

Un terme est soit

- une variable
- une constante
- une construction de la forme $f(t_1, ..., t_n)$ où f est un symbole de fonction d'arité n et $t_1, ..., t_n$ sont des termes.

Une formule du premier ordre est soit

- une construction de la forme $p(t_1, ..., t_n)$ où p est un symbole de prédicat d'arité n et $t_1, ..., t_n$ sont des termes. Une telle formule est appelée un atome.
- une construction de la forme $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$ ou $(F \Leftrightarrow G)$, où F et G sont des formules.
- une construction de la forme $(\forall X. F)$ ou $(\exists X. F)$, où X est une variable et F est une formule.

Un langage du premier ordre donné par un alphabet consiste en l'ensemble des formules construites à partir des symboles de l'alphabet.

La portée de $\forall$ X (resp. $\exists$ X) dans $\forall$ X. F (resp. $\exists$ X. F) est F. Une occurrence liée d'une variable dans une formule est une occurrence qui suit immédiatement un quantificateur ou une occurrence dans la portée d'un quantificateur qui a la même variable immédiatement après lui. Toute autre occurrence d'une variable est libre. Par exemple, dans la formule $(\forall X. p(X, Y)) \wedge (q(X))$, les deux premières occurrences de X sont liées, la troisième occurrence de X et l'occurrence de Y sont libres.

Une formule close est une formule avec aucune occurrence libre de ses variables.

1.2. Interprétations et modèles:

Nous abordons maintenant la sémantique d'un langage du premier ordre. Pour parler de la valeur de vérité d'une formule, il faut d'abord donner une signification à chaque symbole de la formule. Les quantificateurs et les connecteurs ont des significations fixées mais le sens attaché aux constantes, aux symboles de fonction et aux symboles de prédicat peut varier.

Une interprétation d'un langage du premier ordre L consiste en

- un domaine de valeurs D
- l'assignation d'un élément du domaine à chaque constante de L
- l'assignation d'une fonction de $D^n \rightarrow D$ à chaque symbole de fonction de L d'arité n.
- l'assignation d'une fonction de $D^n \rightarrow \{true, false\}$ à chaque symbole de prédicat de L d'arité n.

Il reste à donner une valeur aux variables. Une assignation de variable V pour une interprétation I d'un langage du premier ordre L est une assignation d'un élément du domaine de I à chaque variable de L.

Un terme est assigné à une valeur pour une interprétation I et une assignation de variable V. La valeur d'un terme est définie comme suit:

(i). Chaque variable reçoit son assignation selon V.

(ii). Chaque constante reçoit son assignation selon I.

(iii). La valeur assignée à $f(t_1, ..., t_n)$ est donnée par $f'(t'_1, ...t'_n)$, où f' est la fonction assignée à f par I et $t'_1, ...t'_n$ sont les valeurs assignées aux termes $t_1, ..., t_n$ par I et V.

Une formule est vraie ou fausse pour une interprétation I et une assignation de variable V. La valeur de vérité d'une formule est définie comme suit:

(i). Si la formule est un atome de la forme $p(t_1, ..., t_n)$, alors sa valeur de vérité est donnée par $p'(t'_1, ...t'_n)$ où p' est la fonction assignée à p dans I et $t'_1, ..., t'_n$ sont les valeurs assignées aux termes $t_1, ...t_n$ par I et V.

(ii). Si la formule est de la forme $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$ ou $(F \Leftrightarrow G)$, alors sa valeur de vérité est donnée par la table suivante en fonction des valeurs de vérité de F et G:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \Rightarrow G$	$F \Leftrightarrow G$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

(iii). Si la formule est de la forme $\exists X.$ F, alors sa valeur de vérité est 'true' si il existe un élément d du domaine de I tel que F a la valeur 'true' pour I et V{X/d}, où V{X/d} est V excepté que d est assigné à X; sinon, sa valeur de vérité est 'false'.

(iv). Si la formule est de la forme $\forall X.$ F, alors sa valeur de vérité est 'true' si, pour tout élément d du domaine de I, F a la valeur 'true' pour I et V{X/d}, où V{X/d} est V excepté que d est assigné à X; sinon, sa valeur de vérité est 'false'.

En fait, la valeur de vérité d'une formule close ne dépend pas de l'assignation de variable. Nous disons alors que la formule est vraie si sa valeur de vérité est 'true', et qu'elle est fausse si sa valeur de vérité est 'false'.

En particulier, pour une formule close F d'un langage du premier ordre L, un modèle de F est une interprétation I de L telle que F est vraie pour I.

Pour un ensemble de formules closes d'un langage du premier ordre L, un modèle de S est une interprétation I de L telle que I est un modèle pour chaque formule de S. Un tel ensemble est dit satisfaisable si il possède un modèle et insatisfaisable sinon. Et pour une formule F de L, F est dite consistante avec S si $F \cup S$ est satisfaisable, F est dite inconsistante avec S sinon.

Finalement, nous définissons la notion de conséquence logique. Pour un ensemble S de formules closes et F une formule close d'un langage du premier ordre L, nous disons que F est une conséquence logique de S si, tout modèle de S est un modèle de F.

1.3. Interprétations et modèles de Herbrand:

En programmation logique, nous sommes amenés à considérer plus particulièrement les interprétations de Herbrand définies comme suit:

Nous appelons terme clos (ground), un terme qui ne contient pas de variable.

Pour un langage du premier ordre L, l'univers de Herbrand H_L de L est l'ensemble de tout les termes clos qui peuvent être formés avec les constantes et les symboles de fonctions de L.

Pour un langage du premier ordre L, une interprétation de Herbrand de L est une interprétation de L telle que:

- le domaine de valeur est l'univers de Herbrand H_L de L.
- les constantes sont assignées à elles-mêmes dans H_L.
- un symbole de fonction f de L d'arité n, est assigné à la fonction f' de $(H_L)^n \rightarrow H_L$ définie par $f'(t_1, ..., t_n) = f(t_1, ..., t_n)$.

Donc, dans une interprétation de Herbrand, l'assignation des constantes et des symboles de fonctions est fixée de telle façon que un terme est assigné à lui même. Pour fixer une interprétation de Herbrand, il reste à définir l'assignation des symboles de prédicats.

Pour un langage du premier ordre L et un ensemble S de formules closes de L, un modèle de Herbrand de S est une interprétation de Herbrand de L qui est un modèle de S.

2. La programmation logique:

La programmation logique est issue de travaux sur la démonstration automatique de théorèmes. Le langage Prolog (Programming in Logic) peut être vu comme un sous-ensemble de la logique du premier ordre avec pour seule règle d'inférence, la règle de résolution.

Un programme Prolog consiste un ensemble d'axiomes d'une forme particulière (les clauses). Et une exécution consiste à démontrer qu'une formule donnée (un but) est une conséquence logique des axiomes du programme. En fait, dans le cas de programmes logiques, il apparaît suffisant de ne considérer que les interprétations de Herbrand.

La règle de résolution est une généralisation du modus ponens qui se prête bien au traitement automatique par ordinateur:

$$\frac{\begin{array}{c} A \\ B \Rightarrow C \\ A\,\theta = B\,\theta \end{array}}{C\,\theta}$$

où l'application de θ consiste à instancier les variables pour particulariser les formules. θ est appelé une substitution et les instanciations sont appelées des liens.

Par exemple:

$$\frac{\begin{array}{c} p(X, g(a)) \\ p(b, Y) \Rightarrow q(Y, c) \\ p(X, g(a))\ \theta \equiv p(b, Y)\ \theta \equiv p(b, g(a)) \end{array}}{q(g(a), c)}$$

Ici, l'application de θ revient à remplacer X par b et Y par g(a).

L'interpréteur Prolog travaille par dérivation de but. Pour démonter qu'un but B est une conséquence logique d'un programme P, il cherche une formule $B' \Rightarrow C$ de P telle que il existe θ tel que $B\,\theta = B'\,\theta$ et recommence avec le nouveau but $C\,\theta$. Ainsi de suite jusqu'à ce que soit le nouveau but est vide, il répond alors 'yes' à la question ' B est-elle une conséquence logique de P ? ', soit le nouveau but n'est pas vide mais il n'est plus possible de dériver, il répond alors 'no' à la même question.

D'un point de vue langage de programmation, nous sommes plus intéressés par les liens obtenus sur les variables que par le fait de savoir si un but est oui ou non une conséquence logique du programme. Ces liens constituent le résultat du programme.

Dans la suite, nous présentons brièvement l'algorithme d'unification, la syntaxe et la sémantique opérationnelle de Prolog (sans cut ni négation).

2.1. Unification:

A la base d'un système Prolog, se trouve un algorithme d'unification, utilisé pour calculer les liens entre les variables. Nous introduisons ici la notion d'unificateur et présentons l'algorithme d'unification sous forme de règles de transitions.

2.1.1. Substitution:

Une substitution est un ensemble fini de la forme $\{X_1 / t_1, ..., X_n / t_n\}$ où $X_1, ..., X_n$ sont des variables distinctes, $t_1, ..., t_n$ sont des termes et chaque t_i est différent de X_i. Chaque élément X_i / t_i est appelé un lien sur X_i.

Par exemple, $\theta = \{X / b, Y / X\}$ est une substitution.

2.1.2. Application d'une substitution:

Pour une substitution $\theta = \{X_1 / t_1, ..., X_n / t_n\}$, et un terme t, l'instance de t par θ, est le terme $t\theta$ obtenu en remplaçant simultanément toutes les occurrences de X_i par t_i, pour tout i.

Par exemple, l'instance du terme t = p(X, Y, f(a)) par la substitution $\theta = \{X / b, Y / X\}$ est $t\theta$ = p(b, X, f(a)).

2.1.3. Composition de substitutions:

Pour deux substitutions $\theta = \{X_1 / t_1, ..., X_n / t_n\}$ et $\sigma = \{Y_1 / s_1, ..., Y_m / s_m\}$, la composition $\theta\sigma$ de θ et σ est la substitution obtenue à partir de l'ensemble:

$$\{X_1 / t_1\sigma, ..., X_n / t_n\sigma, Y_1 / s_1, ..., Y_m / s_m\}$$

en supprimant tout lien $X_i / t_i\sigma$ pour lequel $X_i = t_i\sigma$ et tout lien Y_i / s_i pour lequel $Y_i \in \{X_1, ..., X_n\}$.

Par exemple la composition des substitutions $\theta = \{X / f(Y), Y/Z\}$ et $\sigma = \{X / a, Y / b, Z / Y\}$ est $\theta\sigma = \{X / f(b), Z / Y\}$.

La substitution définie par l'ensemble vide est notée ε. Nous avons: $\varepsilon\, t = t$, pour tout terme t.

Remarquons que soient θ, σ et γ des substitutions, nous avons:

(i). $\theta\, \varepsilon = \varepsilon\, \theta = \theta$.
(ii). pour tout terme t : $(t\, \theta)\, \sigma = t\, (\theta\, \sigma)$
(iii). $(\theta\, \sigma)\, \gamma = \theta\, (\sigma\gamma)$

2.1.4. Unificateur:

Nous nous intéressons particulièrement aux substitutions qui unifient deux termes, c'est à dire qui les rendent syntaxiquement identiques.

Deux termes t_1 et t_2 sont unifiables ssi il existe une substitution σ telle que $t_1\, \sigma = t_2\sigma$. Dans ce cas, σ est un unificateur de t_1 et t_2.

Par exemple p(f(X), Z)) et p(Y, a) sont unifiables, car $\sigma = \{Y / f(a), X / a, Z /a\}$ est un unificateur possible.

2.1.5. Unificateur le plus général:

Un unificateur θ de deux termes t_1 et t_2 est un unificateur le plus général (mgu) de t_1 et t_2 si pour tout unificateur σ de t_1 et t_2, il existe une substitution γ telle que $\sigma = \theta\, \gamma$.

Par exemple, $\theta = \{Y / f(X), Z /a\}$ est un unificateur le plus général de p(f(X), Z)) et p(Y, a). Remarquons que $\sigma = \theta\ \{X /a\}$ pour l'exemple précédent.

Il est possible de montrer que l'unificateur le plus général est unique à un renommage près.

Pour t, un terme et V, l' ensemble des variables apparaissant dans t, une substitution de renommage est une substitution $\{X_1/Y_1, ..., X_n/Y_n\}$ telle que $\{X_1, ..., X_n\} \subseteq V$, les Y_i sont distincts et $(V / \{X_1, ..., X_n\}) \cap \{Y_1, ..., Y_n\} = \varnothing$.

2.1.6. Algorithme d'unification:

Pour un ensemble d'équations E = { $t_1 = s_1$, ... $t_n = s_n$}, où t_1, ..., t_n, s_1, ..., s_n sont des termes, nous disons que σ est un unificateur le plus général (mgu) de E si $t_1\sigma = s_1\sigma$ et ... $t_n\sigma = s_n\sigma$.

L'algorithme d'unification prend un ensemble d'équations E = { $t_1 = s_1$, ... $t_n = s_n$}, où t_1, ..., t_n, s_1, ..., s_n sont des termes, et une substitution σ en entrée. Il renvoie 'fail' si E n'est pas unifiable, et $\sigma\sigma'$, où σ' est un mgu de E sinon.

Nous notons $t_1 = s_1$ :: E pour désigner l'ensemble composé de l'équation ($t_1 = s_1$) et des équations de E. Un état de transition intermédiaire est défini par une paire (σ, E). Un état final est soit une substitution, soit 'fail'. L'algorithme d'unification est défini par les sept règles de transitions suivantes:

(1). $(\sigma, \{\ \}) \rightarrow \sigma$

(2). $$\frac{X \neq t,\ X \notin var(t)}{(\sigma,\ X = t :: E) \rightarrow (\sigma\{X/t\},\ E\{X/t\})}$$

(3). $$\frac{X \neq t,\ X \in var(t)}{(\sigma,\ X = t :: E) \rightarrow fail}$$

(4). $(\sigma, X = X :: E) \rightarrow (\sigma, E)$

(5). $(\sigma, t = X :: E) \rightarrow (\sigma, X = t :: E)$ si $X \neq t$

(6). $$\frac{f \neq g \text{ ou } n \neq m}{(\sigma,\ f(t_1,\ldots,t_n) = g(s_1, \ldots, s_m) :: E) \rightarrow fail}$$

(7). $(\sigma, f(t_1, ..., t_1) = g(s_1, ...,s_n) :: E) \rightarrow (\sigma, t_1 = s_1 :: ... :: t_n = s_n :: E)$

Dans (2) et (3), var(t) désigne l'ensemble des variables contenues dans t. Le test pour savoir si une variable X est contenue dans un terme t est connu sous le nom de test d'occurrence (occur check). La plupart des systèmes Prolog l'ignorent cependant car il est très lourd et en fait rarement utile.

En particulier, pour calculer un unificateur le plus général de deux termes t_1 et t_2, il suffit d'exécuter l'algorithme avec E = $\{t_1 = t_2\}$ et la substitution vide en entrée.

Par exemple, l'unification de p(a, X, h(g(Z))) et p((Z, h(Y), h(Y)) donne la trace suivante:

$(\varepsilon, p(a, X, h(g(Z))) = p((Z, h(Y), h(Y)))$

$\rightarrow (\varepsilon, a = Z, X = h(Y), h(g(Z)) = h(Y))$

$\rightarrow (\varepsilon, Z = a, X = h(Y), h(g(Z)) = h(Y))$

$\rightarrow (\{Z / a\}, X = h(Y), h(g(a)) = h(Y))$

$\rightarrow (\{Z / a, X / h(Y)\}, h(g(a)) = h(Y))$

$\rightarrow \{Z / a, X / h(Y)\}, g(a) = Y)$

$\rightarrow (\{Z / a, X / h(Y)\}, Y = g(a))$

$\rightarrow (\{Z / a, X / h(Y), Y = g(a)\})$

Ce qui donne l'unificateur $\{Z / a, X / h(Y), Y = g(a)\}$.

2.2. Syntaxe:

Un programme Prolog est composé d'un ensemble de clauses. Clause, fait et but sont définis comme suit:

Une clause est une formule de la forme

$\forall X_1, ..., X_n (A \Leftarrow B_1 \wedge ... \wedge B_n)$

notée A :- $B_1, ..., B_n$.

où $A, B_1, ..., B_n$ sont des atomes et $X_1, ..., X_n$ sont toutes les variables apparaissant dans ces atomes. A est appelé la tête de la clause et $B_1, ..., B_n$ est appelé le corps de la clause.

Un fait est une clause de la forme

$\forall X_1, ..., X_n (A \Leftarrow true)$.

notée A .

Un but est une clause de la forme

$\forall X_1, ..., X_n (false \Leftarrow A)$.

notée ? :- A.

Dans un programme Prolog, parmi les termes de la forme $f(t_1, ..., t_n)$, on trouve en particulier les listes:

Une liste est soit
- la liste vide: nil notée [ ].
- une construction de la forme cons(t, l) où t est un terme et l est une liste. Une liste cons (t, l) est notée [t | l] et une liste cons(a, cons(b, cons(...))) est notée [a, b, ...].

Pareillement, le prédicat = est toujours interprété comme l'égalité syntaxique. C'est à dire que deux termes sont égaux ssi ils s'écrivent de la même façon.

Par exemple le programme habituel pour Append s'écrit comme suit:

Append ([ ], X, X).
Append ([X | Y], Y, [Z, W]) :- Append (Y, Z, W).

Pour les listes X, Y, Z en entrée, le programme Append (X, Y, Z) renvoie X, Y et Z telles que Z est la concaténation de X et Y.

2.3. Sémantique:

Nous présentons la sémantique sémantique opérationnelle de Prolog exposée dans [9]. L'exécution d'un programme Prolog est définie comme une séquence de buts dérivés. D'abord, nous définissons la notion de SLD-dérivation. Ensuite, celle de procédure de dérivation.

2.3.1. SLD-dérivation:

Soit P, un programme et B, un but. Une SLD-dérivation de $P \cup \{B\}$ consiste en un séquence $<B_1, ..., B_n>$ de buts où $B = B_0$ et si $B_i = A_1, ..., A_m, ..., A_k$ alors:

(i). il existe une clause $A:- A'_1, ..., A'_q$ qui est une clause renommée de P
(ii). θ est un mgu de A et A_m
(iii). $B_{i+1} = (A_1, ..., A_{m-1}, A'_1, ..., A'_q, A_{m+1}, ..., A_k)\theta$

On dit que B_{i+1} est dérivé de B_i et que A_m est l'atome sélectionné. La sélection des atomes dans une dérivation est définie par une fonction arbitraire appelée règle de sélection. On parle alors de dérivation via une règle de sélection.

Chaque clause utilisée est une variante d'une clause de P, de telle façon qu'elle ne contienne pas de variables qui apparaissent déjà dans la dérivation jusque B_i . Cela peut être réalisé par exemple en indiçant les variables de B par 0 et les variables de la clause qui sert à dériver B_i par i. Ce renommage est nécessaire pour éviter une confusion entre les variables de B_i et les variables présentes dans les différentes clauses. Par exemple, il serait impossible d'unifier le but p(X) et la tête de clause p(f(X)), alors que la présence de X dans la clause est un hasard.

Les SLD-dérivations peuvent être finies ou infinies. Une SLD-dérivation finie peut réussir ou échouer. Une SLD-dérivation réussit si son dernier but est vide. Une SLD-dérivation échoue si son dernier but n'est pas vide et si l'atome sélectionné de ce but ne peut être unifié avec la tête d'aucune des clauses du programme (fail).

Pour un programme P et un but B, une solution θ de $P \cup \{B\}$ est la substitution obtenue en restreignant la composition des θ_1, .., θ_n aux variables de B, où θ_1, .., θ_n est la séquence des mgu's utilisés dans une SLD-dérivation réussie de $P \cup \{B\}$.

Il est possible de monter que quelle que soit la règle de sélection choisie, si $P \cup \{B\}$ possède une solution, alors il est possible de trouver une dérivation réussie de $P \cup \{B\}$, avec la règle de sélection donnée. Cette propriété est connue comme l'indépendance de la règle de sélection.

Par exemple, pour le programme Append donné précédemment, considérons le but Append ([a], [b, c], R). Nous avons la SLD-dérivation suivante:

(0). B_0 = Append ([a], [b, c], R_0).

(1). C_1 = Append ([X_1 | Y_1], Z_1, [X_1, W_1]) :- Append (Y_1, Z_1, W_1).

$\theta_1 = \{X_1$ / a, Y_1 / [ ], Z_1 / [b, c], R_0 / [X_1 , W_1]}.

B_1 = Append ([ ], [c, d], W_1).

(2). C_2 = Append ([ ], X_2, X_2).

$\theta_2 = \{X_2$ / [b, c], W_1 / $X_2\}$.

Alors {R / [a, b, c]} est une solution. Dans ce cas c'est en fait la seule solution.

Dans d'autres cas, il peut il y avoir plusieurs solutions. Par exemple, considérons le but Append (X, Y, [a, b]). Nous avons la SLD-dérivation suivante:

(0). B_0 = Append (X_0, Y_0, [a, b]).

(1). C_1 = Append ([ ], X_1, X_1).

θ_1 = {X_0 / [ ], Y_0 / X_1, X_1 = [a, b]}.

Alors {X/ [ ], Y = [a, b]} est une solution. Ce n'est pas le seule, il y a d'autres SLD-dérivations possibles. Par exemple en prenant la seconde clause au lieu de la première pour C_1:

(1). C_1 = Append ([X_1 | Y_1], Z_1, [X_1, W_1]) :- Append (Y_1, Z_1, W_1).

θ_1 = {X_0 / [X_1 | Y_1], Y_0 / Z_1, X_1 / a, W_1 / b}

B_1 = Append (Y_1, Z_1, [b]).

(2). C_2 = Append ([ ], X_2, X_2)

θ_2 = {Y_1 / [ ], Z_1 / X_2, X_2 / [b]}

Ce qui donne la solution {X / [a], Y/ [b]}. A nouveau, nous pouvons prendre la seconde clause au lieu de la première pour C_2:

(2). C_2 = Append ([X_2 | Y_2], Z_2, [X_2, W_2]) :- Append (Y_2, Z_2, W_2).

θ_2 = {Y_1 / [X_2 | Y_2], Z_1 / Z_2, X_2 / [b], W_2 / [ ]}

B_2 = Append (Y_2, Z_2, [ ]).

(3). C_3 = Append ([ ], X_3, X_3).

θ_3 = {Y_2 / [ ], X_3 / Z_2, X_3 / [ ]).

Ce qui donne finalement la solution {X / [a, b], Y/ [ ]}.

2.3.2. Procédure de dérivation:

Plusieurs stratégies sont possibles pour réaliser la recherche des solutions. L'espace de recherche peut être décrit sous forme d'un arbre, appelé arbre de dérivation.

Soit P, un programme et B, un but. Un arbre de dérivation pour $P \cup \{B\}$est un arbre tel que:

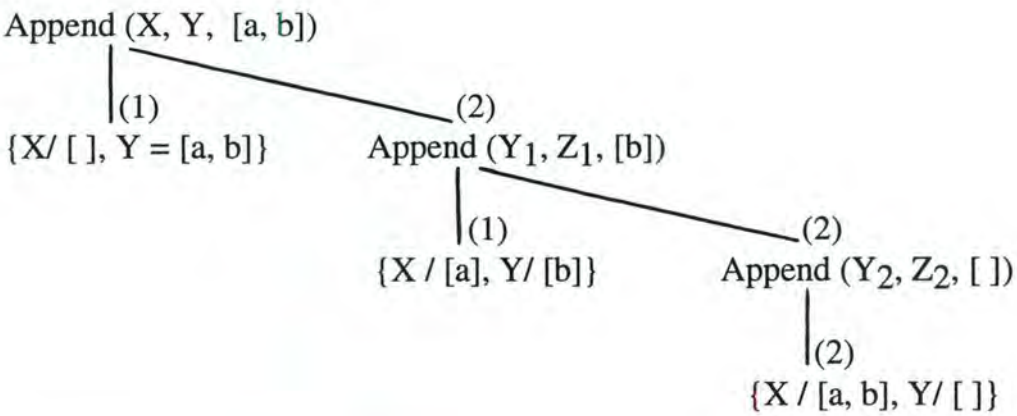
(i). Chaque noeud est un but, la racine est B.
(ii). Soit un noeud $B_i = A_1, ..., A_m, ..., A_k$ et A_m l'atome sélectionné. Alors pour chaque clause $A:- A'_1, ..., A'_q$ telle que A_m et A sont unifiables avec le mgu θ, le noeud a un fils $B_{i+1} = (A_1, ..., A_{m-1}, A'_1, ..., A'_q, A_{m+1}, ..., A_k)\theta$.
(iii). Un noeud qui est un but vide n'a pas de fils.

En fait, chaque branche de l'arbre de dérivation de $P \cup \{B\}$ correspond à une dérivation de $P \cup \{B\}$.

Une procédure de dérivation est spécifiée par une règle de sélection et une règle de recherche. Une règle de recherche est une stratégie pour parcourir l'arbre de dérivation à la recherche de solutions.

Prolog utilise une règle de sélection qui sélectionne toujours l'atome le plus à gauche dans un but et effectue une recherche en profondeur d'abord en utilisant l'ordre textuel des clauses dans le programme comme l'ordre fixé dans lequel elles doivent être essayées. Remarquons qu'une recherche en profondeur d'abord peut boucler sur une branche infinie et ne pas donner de solution. Cela peut être géré par l'utilisation d'une opération appelée 'cut', expressément ajoutée à Prolog dans ce but.

Par exemple, la recherche des solutions du programme Append donné précédemment pour le but Append (X, Y, [a, b]) est décrite par l'arbre suivant:



Les branches sont étiquetées par le numéro de la clause qui a servi à dériver. Pour les buts vides, nous donnons les solutions qui leurs correspondent.

3. La programmation logique avec contraintes:

La programmation logique avec contraintes (CLP) étend la programmation logique à une réduction basée sur la résolution de contraintes. Elle est issue d'une volonté d'ajouter à Prolog une certaine théorie de l'égalité (autre que l'égalité syntaxique) et d'utiliser l'arithmétique. L'introduction de contraintes permet d'exprimer des propriétés directement liées au domaine d'application au lieu de devoir les coder dans des termes Prolog. Les contraintes permettent également de représenter des propriétés explicitement et pas comme des liens sur les variables. Cela offre des langages plus souples et plus expressifs.

Typiquement nous voulons résoudre des problèmes sur des domaines comme les ensembles, les graphes, les expressions booléennes, les réels... Ces domaines ont des opérations algébriques comme l'intersection d'ensemble, de graphe, la disjonction, la multiplication... Ils ont aussi des prédicats privilégiés comme l'égalité d'ensembles, l'isomorphisme de graphes, différentes formes d'inégalités comme l'inclusion ensembliste, <, >, ... Ce sont ces prédicats que nous appelons contraintes. Les différentes formes d'égalités ne sont rien de plus qu' un type particulier de contrainte.

Les langages CLP admettent des contraintes dans le corps des clauses et dans les buts. L'exécution d'un programme CLP est basée sur la détermination de la solvabilité d'un ensemble de contraintes dans le domaine d'application et non plus sur l'unification. L'algorithme d'unification qui est au centre de la sémantique de Prolog apparaît comme un cas particulier de résolution de contraintes. Il nous dit si deux termes comme f(X, a) et f(b, Y) peuvent être rendu identiques par une instanciation particulière des variables (ici X = a et Y = b). En d'autres mots il nous dit si l'équation f(X, a) = f(b, Y) est solvable.

Contrairement à Prolog, le domaine d'application du modèle CLP est générique. Chaque instance du modèle CLP est définie par un domaine d'application et un mécanisme de résolution de contraintes. La sémantique est cependant définie au niveau du domaine et est héritée par chaque instance. En particulier, Prolog est considéré comme une instance du modèle CLP sur le domaine de Herbrand.

Dans la suite, nous présentons brièvement la syntaxe et la sémantique opérationnelle du modèle CLP.

3.1. Syntaxe:

Le modèle CLP utilise la même syntaxe que Prolog mais admet en plus des contraintes dans les buts et les corps de clauses. Une clause CLP est une de la forme:

$$A :- c_1, ..., c_k , B_1, ..., B_n.$$

où A, B_1, ..., B_n sont des atomes et c_1, ..., c_k sont des contraintes. Ces contraintes sont des formules formées des termes du domaine d'application selon les opérations (fonctions et prédicats) prédéfinies sur ce domaine.

Par exemple, Prolog est considéré comme un CLP sur le domaine de Herbrand. Les seules contraintes admises sont les équations entre termes.

Dans CLP(H), le programme habituel pour Append peut s'écrire:

```
Append(X, Y, Z) :- X = [ ], Y = Z.
Append(X, Y, Z) :- X = [A|B], Z = [A|C], Append(B, Y, C).
```

Un autre exemple est CLP(R), défini sur les Réels. Les contraintes admises sont formées à partir des opérations arithmétiques usuelles.

Dans CLP(R), le programme habituel pour Fact peut s'écrire:

```
Fact (N, F) :- N  > 0, F = N * M , Fact ((N-1), M).
Fact (0, 1).
```

Pour N positif en entrée, Fact(N, F) renvoie la factorielle de N dans F.

3.2. Sémantique:

Dans le modèle CLP, l'interpréteur est toujours basé sur le principe de résolution mais le concept d'unification est remplacé par la satisfaction de contraintes dans le domaine d'application.

La sémantique de CLP est calquée sur celle de Prolog. D'une manière générale, l'algorithme d'unification est remplacé par un algorithme de résolution de contraintes. Dans CLP(R), il s'agit d'une version modifiée du simplexe.

Au lieu des substitutions, nous trouvons des ensembles de contraintes. Et pour un but $B_i = c, A_1, ..., A_m, ..., A_k$ et une clause A:- $c', A'_1, ..., A'_q$, nous ne cherchons plus à ce que A et A_m soient unifiables, mais que $c'' = \{c, c', \{A = A_m\})$ soit solvable. Dans ce cas, le but dérivé de B_i est $B_{i+1} = (c'', A_1, ..., A_{m-1}, A'_1, ..., A'_q, A_{m+1}, ..., A_k)$.

La dérivation est réussie lorsque le dernier but ne contient plus que des contraintes. Ces contraintes constituent alors le résultat du programme. La dérivation échoue lorsque le dernier but contient encore des atomes qui ne peuvent être réduits avec aucunes clauses du programme.

Par exemple, l'exécution du programme Fact sur CLP(R), donné précédemment, avec le but $B = \{fact (X, 6)\}$ donne:

(0). D'abord, nous renommons le but initial de la façon suivante:

$B_0 = \{Fact (X_0, 6)\}$

(1). Puis, nous prenons la première clause de Fact renommée pour C_1 et nous rassemblons les contraintes d'unification $\{X_0 = N_1, 6 = F_1\}$ et les contraintes de C_1 dans c_1. Nous résolvons c_1 et obtenons le nouveau but B_1 en prenant c_1 et les atomes de C_1.

$C_1 = Fact (N_1 , F_1)$:- $N_1 > 0, F_1 = N_1 * M_1 , Fact ((N_1 - 1), M_1)$

$c_1 = \{X_0 = N_1, 6 = F_1, N_1 > 0, F_1 = N_1 * M_1\} = \{X_0 > 0, 6 = X_0 * M_1\}$

$B_1 = c_1 , Fact ((X_0 - 1), M_1)$.

(2). A nouveau, nous prenons la première clause de Fact renommée pour C_2 et nous rassemblons les contraintes d'unification $\{X_0 - 1 = N_2, M_1 = F_2\}$, les contraintes de C_2 et de B_1 dans c_2. Nous résolvons c_2 et obtenons le nouveau but B_2 en prenant c_2 et les atomes de C_2.

$C_2 = Fact (N_2 , F_2)$:- $N_2 > 0, F_2 = N_2 * M_2 , Fact ((N_2 - 1), M_2)$.

$c_2 = \{ X_0 - 1 = N_2, M_1 = F_2, X_0 > 0, 6 = X_0 * M_1, N_2 > 0, F_2 = N_2 * M_2 \} =$
$\{ X_0 > 1, 6 = X_0 * (X_0 - 1) * M_2\}$

$B_2 = c_2, Fact ((X_0 - 2), M_2)$.

(3). A nouveau, nous prenons la première clause de Fact renommée pour C_3 et nous rassemblons les contraintes d'unification $\{X_0 - 2 = N_3, M_2 = F_3\}$, les contraintes de C_3 et de B_2 dans c_3. Nous résolvons c_3 et obtenons le nouveau but B_3 en prenant c_3 et les atomes de C_3.

C_3 = Fact (N_3, F_3) :- $N_3 > 0$, $F_3 = N_3 * M_3$, Fact $((N_3-1), M_3)$.

$c_3 = \{X_0 - 2 = N_3, M_2 = F_3, X_0 > 1 , 6 = X_0 * (X_0 - 1) * M_2, N_3 > 0, F_3 = N_3 * M_3\} = \{X_0 > 2, 6 = X_0 * (X_0 - 1) * (X_0 - 2) * M_3\}$

$B_3 = c_3$, Fact $((X_0 -3), M_3)$.

(4). Finalement, nous prenons la seconde clause de Fact renommée pour C_4 et nous rassemblons les contraintes d'unification $\{X_0 - 3 = 0, M_3 = 1\}$, les contraintes de B_3 dans c_4. Nous résolvons c_4 et obtenons le nouveau but B_4 en prenant c_3. Et B_4 ne contient plus d'atomes.

(4). C_4 = Fact (0, 1).

$c_4 = \{X_0 - 3 = 0, M_3 = 1, X_0 > 2, 6 = X_0 * (X_0 - 1) * (X_0 - 2) * M_3\} = \{X_0 = 3\}$

$B_4 = c_4 = (X_0 = 3)$.

Ce qui donne la solution: X = 3.

4. Points fixes:

Une relation R sur un ensemble S est un sous-ensemble de $S \times S$. Il est d'usage de noter x R y pour $(x, y) \in R$.

Un ordre partiel sur un ensemble S est une relation R sur cet ensemble telle que:

(i). x R x pour tout $x \in S$.

(ii). x R y et y R x implique $x = y$, pour tout $x, y \in S$.

(iii). x R y et y R z implique x R z pour tout $x, y, z \in S$.

Par exemple, pour un ensemble S, l'inclusion $\subseteq$ est un ordre partiel sur l'ensemble 2^S des sous-ensembles de S.

Un ordre partiel est plus souvent noté $\leq$.

Pour un ensemble S avec un ordre partiel $\leq$, $a \in S$ est une borne supérieure d'un sous-ensemble X de S si $x \leq a$, pour tout $x \in X$. Pareillement, $b \in S$ est une borne inférieure d'un sous-ensemble X de S si $b \leq x$, pour tout $x \in X$.

Pour un ensemble S avec un ordre partiel $\leq$, $a \in S$ est la plus petite une borne supérieure d'un sous-ensemble X de S si a est une borne supérieure de X et pour toute borne supérieure a' de X, nous avons $a \leq a'$. Pareillement, $b \in S$ est la plus grande borne inférieure d'un sous-ensemble X de S si b est une borne inférieure de X et pour toute borne inférieure b' de X, nous avons $b' \leq b$.

La plus petite borne supérieure d'un ensemble X est unique, si elle existe, et est notée lub(X) (least upper bound). Pareillement, la plus grande borne inférieure d'un ensemble X est unique, si elle existe, et est notée glb(X) (greatest lower bound).

Un ensemble L, muni d'un ordre partiel, est un treillis complet si lub(X) et glb(X) existent pour tout sous-ensemble X de L.

Lub(L) est l'élément supérieur (Top) et glb(L) est l'élément inférieur (Bottom) du treillis, noté $\perp$.

Pour l'exemple précédent, 2^S muni de $\subseteq$ est treillis complet. En effet, la plus petite borne supérieure d'une collection de sous-ensemble de S est donnée par leur union et la plus grande borne inférieure est leur intersection. L'élément 'top' est S et l'élément 'bottom' est l'ensemble vide.

Pour un treillis complet L, une fonction T: L $\rightarrow$ L est monotone si T(x) $\leq$ T(y) lorsque x $\leq$ y.

Pour un treillis complet L, un sous-ensemble X de L est dirigé si chaque sous-ensemble fini de X possède une borne supérieure dans X.

Pour un treillis complet L, une fonction T: L $\rightarrow$ L est monotone si T(lub(X)) = lub(T(X)) pour tout sous-ensemble dirigé X de L.

Pour un treillis complet L, une fonction T: L $\rightarrow$ L, a $\in$ L est un point fixe de T si T(a) = a. Et a est le plus petit point fixe de T si pour tout point fixe b de T, nous avons a $\leq$ b.

Enfin, pour un treillis complet L et une fonction T: L $\rightarrow$ L, si T est monotone alors T possède un plus petit point fixe. De plus si T est continue alors le plus petit point fixe de T est égal à la limite de la suite croissante:

$$x_0 \leq x_1 \leq \ldots \leq x_i \leq \ldots$$

où $x_0 = \bot$ et $x_{i+1} = T(x_i)$.

Chapitre 2: Les langages logiques concurrents avec contraintes

Les langages logiques concurrents avec contraintes (Langages CC) étendent les langages logiques avec contraintes (CLP) avec des notions issues du domaine de la programmation logique concurrente.

La programmation logique concurrente est issue de considérations sur l'exécution parallèle de programmes logiques. Ces langages considèrent un but comme un système de processus concurrents. Un pas d'exécution consiste en la réduction d'un processus en un système de processus et les variables partagées sont vues comme des canaux de communications entre processus. L'exécution parallèle simple mène très rapidement à des calculs non-déterministes inutiles. Certains buts non-déterministes peuvent être lancés avant d'autres qui auraient pu arrêter la recherche immédiatement. Aussi les langages logiques concurrents introduisent des opérations permettant de suspendre l'exécution de certains buts jusqu'à ce que certains liens soient obtenus.

L'idée essentielle du modèle CC est celle d'une communication basée sur les contraintes. Les langages CC introduisent une condition de suspension basée sur le test qu'une contrainte est impliquée par les contraintes déjà accumulées au cours de la résolution. Un but est considéré comme un système de processus concurrents communiquant en posant et testant des contraintes dans un espace partagé.

Dans ce chapitre, nous présentons d'abord le paradigme de base des langages CC. Ensuite nous donnons une définition de la notion de système de contraintes. Enfin, nous présentons la syntaxe et la sémantique des langages CC. Pour plus de clarté, nous présentons d'abord les langages CC déterministes que nous étendons ensuite aux langages CC non-déterministes. Nous terminons ce chapitre en donnant quelques exemples de programmes CC.

1. Paradigme de base:

Les langages CC constituent une classe de langages logiques concurrents qui repose sur l'utilisation de contraintes comme moyen de communication et de contrôle entre plusieurs agents (processus) exécutés en concurrence. Un programme CC est conceptualisé comme un système d'agents exécutés simultanément et interagissant par le biais d'un ensemble global de contraintes, appelé store. Chaque agent peut essentiellement ajouter une contrainte au store (opération Tell) ou tester si une contrainte est impliquée par le store (opération Ask).

Conceptuellement, une contrainte peut être définie comme un sous-ensemble de l'espace des valeurs possibles des variables d'un programme donné. En d'autres mots, une contrainte constitue une information partielle sur la valeur des variables. Par exemple la contrainte X = [a|Y] nous dit que Y est une liste et X est une liste dont le premier élément est a et le second est la liste Y. Cela constitue une information partielle sur la valeur de X et Y dans le sens où ces valeurs ne sont pas complètement définies par la contrainte. Tout couple de valeurs (X, Y) tel que Y est une liste et X est une liste dont le premier élément est a et le second est égal à la liste Y est possible. Au cours de l'exécution, les contraintes sont accumulées dans le store et constituent une information de plus en plus précise. La situation dans laquelle une variable est contrainte à une valeur unique apparaît comme un cas particulier.

Une opération Tell consiste à ajouter une contrainte à celles déjà présentes dans le store. Poser une contrainte sur une variable revient à réduire l'ensemble des valeurs possibles pour cette variable. C'est à dire que, l'ensemble des valeurs possibles dans le store final est l'intersection de l'ensemble des valeurs possibles dans le store initial et l'ensemble des valeurs définies par la contrainte ajoutée. Une variable ne change pas de valeur, mais certaines valeurs qui étaient possibles initialement, ne le sont plus. Il est important de remarquer que le store est restreint de façon monotone. Toutes les contraintes qui étaient impliquées par le store initial, le sont encore par le store final.

Une opération Ask consiste à tester une contrainte par rapport au store. Elle réussit si les contraintes contenues dans le store impliquent la contrainte donnée. La synchronisation entre agents est simplement réalisée par un Ask bloquant: un agent qui exécute un Ask bloque (suspend) si le store ne peut pas impliquer la contrainte donnée. C'est à dire que la contrainte est consistante avec le store mais celui-ci ne contient pas encore assez d'information pour décider. L'agent reste alors bloqué jusqu'à ce qu'éventuellement un autre agent exécuté en concurrence fournisse au store l'information nécessaire pour prendre la décision.

Intuitivement, un système de contraintes peut être considéré comme un système d'information partielle sur les variables d'intérêt. En fait pour fixer un langage, il suffit de donner l'ensemble des contraintes admises par le système et la relation d'implication qui les relie. La notion de système de contraintes permet la définition d'une sémantique générique, chaque instance constituant un langage avec son propre système de contraintes.

Le paradigme CC offre de puissantes possibilités de communications. Les variables sont utilisées comme canaux de communications entre différents agents exécutés en concurrence. Par exemple, deux agents partagent une variable X. Supposons que l'un veut transmettre le message m à l'autre. Il lui suffit de mettre la contrainte X = m dans le store. L'autre agent reçoit ainsi le message via X.

A partir de cette idée, il est facile de générer un flux de messages à partir d'une variable partagée entre deux agents. En plus du message, l'agent émetteur transmet une nouvelle variable qui servira à émettre le message suivant. Par exemple, dans la situation précédente, l'émetteur met la contrainte X = f (m, Y) dans le store, où Y est une nouvelle variable. L'agent récepteur reçoit ainsi le message et Y, qui peut servir à son tour de variable partagée entre les deux agents. Cette technique permet d'établir un réel dialogue entre les agents. Elle repose sur la possibilité d'instancier partiellement les variables.

2. Systèmes de contraintes:

Le modèle CC est défini sur un système de contraintes générique. Il s'agit ici de définir une classe de systèmes de contraintes auxquels le modèle s'applique.

D'abord nous définissons la notion de système de contraintes. Ensuite nous présentons le système de contraintes de Herbrand, sous-jacent aux langages de programmation logique proprement dits, sur lequel nous baserons nos exemples par la suite.

2.1. Définition d'un système de contraintes:

La façon la plus simple de définir les contraintes est de les considérer comme des formules de premier ordre interprétées sur une structure particulière, pour tenir compte des propriétés particulières du domaine d'application. Néanmoins, une formalisation générale des systèmes de contraintes a été récemment proposée par V. Saraswat dans [12]. Les systèmes de contraintes sont définis à la manière de systèmes de déduction, ce qui semble mieux adapté aux concepts de base du modèle CC. Cette formalisation met en évidence la définition de la relation d'implication entre contraintes qui suffit à définir tout le système de contraintes:

Un système de contraintes est une paire (D, |-) telle que:

(i). D, le domaine de contraintes, est un ensemble de formules du premier ordre sur un alphabet contenant le prédicat =, closes sous la conjonction et la quantification existentielle. Ces formules sont appelées les contraintes de D.

(ii). |- est une relation d'implication sur D entre un ensemble fini de formules et une formule telle que:

$$(1).\ \Gamma, d \vdash d \qquad (4).\ \frac{\Gamma_1 \vdash d \quad \Gamma_2, d \vdash e}{\Gamma_1, \Gamma_2 \vdash e}$$

$$(2).\ \frac{\Gamma, d, e \vdash f}{\Gamma, d \wedge e \vdash f} \qquad (5).\ \frac{\Gamma \vdash d \quad \Gamma \vdash e}{\Gamma \vdash d \wedge e}$$

$$(3).\ \frac{\Gamma, d \vdash \mathrm{e}}{\Gamma, \exists X.d \vdash \mathrm{e}} \qquad (6).\ \frac{\Gamma \vdash d[t/X]}{\Gamma \vdash \exists X.d}$$

où Γ représente un ensemble de formules; d, e et f des formules, t un terme et X une variable. En fait, ces règles sont celles de la logique intuitive pour la quantification existentielle et la conjonction. Dans (3), nous supposons X non libre dans Γ,e.

(iii). |- est générique: $\Gamma[t / X] \vdash d[t / X]$ lorsque $\Gamma \vdash d$ pour tout terme t. La généricité exprime essentiellement que les variables de chaque côté de l'implication sont universellement quantifiées.

L' implication est étendue à une relation sur $2^D \times 2^D$ avec: $(u \vdash v) \Leftrightarrow (\forall d \in v. u \vdash d)$. C'est à dire qu'un ensemble de contraintes u implique un ensemble de contraintes v ssi u implique chaque contrainte d de v.

Un système de contraintes peut néanmoins être défini directement à partir d'une théorie du premier ordre T. Il suffit de considérer l'ensemble des formules du langage de T, closes sous la conjonction et la quantification existentielle, comme domaine de contraintes D. Et qu'un ensemble de contraintes Γ implique une contrainte d ssi d est une conséquence logique de Γ et des axiomes de T.

Remarquons que puisque un domaine de contraintes est défini comme clos sous la conjonction, il aurait été possible de définir un store comme une contrainte consistant en la conjonction des contraintes accumulées au cours de l'exécution plutôt que comme l'ensemble de ces contraintes. Le sens aurait été tout à fait pareil. Le choix de cette définition est essentiellement une question de style. La notion de store comme ensemble de contraintes semble mieux adaptée à la définition des concepts de bases du modèle CC. Le domaine de contraintes est néanmoins défini comme clos sous la conjonction pour des facilités de définitions.

Pour un système de contraintes (D,|-), et un ensemble fini de variables V (celles d'un programme donné), nous appelons store un ensemble de contraintes clos sous |-, et nous notons D_V l'ensemble des stores de contraintes de D sur les variables de V, c'est à dire qui n'utilisent que des variables de V.

Intuitivement, deux stores sont équivalents s' ils contiennent la même information. c'est à dire qu'ils s'impliquent l'un, l'autre. Aussi, pour éviter de manipuler des ensembles de contraintes différents mais qui contiennent la même information, nous considérons des ensembles de contraintes clos sous l'implication. C'est à dire que un ensemble {c} est assimilé à $\{c' : c \vdash c'\}$. Par exemple, si les seules contraintes admises sont de la forme $\leq$ alors $\{X \leq 3\}$ est assimilé à $\{X \leq 3, X \leq 4, X \leq 5, ...\}$.

L'implication défini un ordre partiel sur D_V : $c \geq d \Leftrightarrow c \vdash d$. Intuitivement un store est plus grand qu'un autre s' il contient plus d'information que lui.

Il est alors possible de munir (D_V, $\vdash$) d'une structure de treillis complet en prenant la fermeture de l'union comme plus petite borne supérieure (notée $\cup$). Et l'intersection comme plus grande borne inférieure. Intuitivement prendre la fermeture de l'union revient à rassembler l'information de plusieurs stores. Et prendre l'intersection revient à extraire l'information commune à plusieurs stores.

L'élément inférieure du treillis est true = $\{s: \varnothing \vdash s\}$. Et l'élément supérieure est D, qui est noté false.

2.2. Le système de contraintes de Herbrand:

Par exemple, nous considérons le système de contraintes de Herbrand qui est sous-jacent aux langages de programmation logique proprement dits.

Soit un alphabet du premier ordre A, avec le prédicat d'égalité =. Les contraintes (c) admises par le système sont définies comme suit:

$$c ::= s = t \mid \exists X.\, c$$

où s et t représentent des termes construits sur A, X une variable de A et $\exists$ la quantification existentielle.

Les contraintes sont interprétées sur l'univers de Herbrand. Et l'égalité est interprétée comme l'égalité syntaxique. C'est à dire que s = t est satisfaisable ssi il existe une assignation de valeurs aux variables de s et t qui rende deux termes identiques.

Par exemple f (a, X) et f (Y, b) peuvent être rendu identiques en prenant X = a et Y = b. Bien souvent, il y a plusieurs assignations possibles. Par exemple: X et f (a, Y) peuvent être rendu identiques en prenant X = f (a, b) et Y = b, ou X = f (a, a) et Y = a.

Alors, un ensemble de contraintes (c_1, ... c_n) implique une contrainte c si toute assignation de valeur aux variables qui satisfait chacune des c_i satisfait c.

Les algorithmes de satisfiabilité et l'implication de ce système de contraintes peuvent être résolus par un algorithme de mgu du premier ordre, comme celui présenté dans le chapitre précédent.

3. Les langages CC déterministes:

Les langages CC déterministes traitent les opérations Tell et Ask, la conjonction, la quantification existentielle et la récursion. Comme nous l'avons déjà introduit, l'opération Tell consiste à ajouter une contrainte au store et l'opération Ask revient à exécuter un agent donné à la condition qu' une contrainte donnée soit impliquée par le store. La conjonction consiste en l'exécution parallèle de plusieurs agents. Elle correspond en fait à l'exécution parallèle des différents atomes d'un même but en Prolog (et pas à la réduction d'un atome avec différentes clauses). La quantification existentielle consiste à introduire des variables locales à un agent, c'est à dire cachées au store global partagé. Elle est en fait une réplique au renommage des variables en Prolog. La récursion consiste en un appel de prédicat, à la manière d'un appel de procédure classique.

Les programmes CC déterministes sont caractérisés par le fait qu'ils produisent un résultat (store final) unique. Remarquons en effet que ces programmes n'offrent pas l'équivalent du choix entre plusieurs clauses en Prolog. Cet équivalent est introduit par la disjonction dans les langages CC non-déterministes. Il n'y a aucun choix dans un programme CC déterministe si ce n'est l'ordre dans lequel les contraintes sont ajoutées au store, ce qui n'a pas d'influence pas sur le résultat final.

D'abord, nous présentons la syntaxe et la sémantique opérationnelle des langages CC déterministes sur base des règles de transitions proposées par V. Saraswat, M Rinard et P. Panangaden dans [10]. Ensuite, nous donnons la sémantique dénotationnelle des langages CC déterministes, sur base des dénotations proposées par C. Codognet et P. Codognet dans [2]. Il est sans doute inutile de considérer les deux sémantiques. Néanmoins la sémantique opérationnelle semble plus abordable et permet de mieux comprendre ensuite la sémantique dénotationnelle qui est à la base du modèle d'interprétation abstraite donné dans le chapitre suivant.

3.1. Syntaxe :

Soit un système de contraintes (D, |-). Programme (P), déclaration (D) et agent (A) sont définis comme suit:

$$P ::= D.A$$

$$D ::= \varepsilon \mid p(\overline{X}) :: A \mid D.D$$

$$A ::= c \mid c \rightarrow A \mid A \wedge A \mid \exists X.\ A \mid p(\overline{X})$$

où c représente une contrainte de D, p un symbole de prédicat non interprété et $\overline{X}$ un vecteur de variables .

Un programme est composé d'un ensemble de déclarations et d'un agent initial. Une déclaration associe un symbole de prédicat à un agent (statique), à la manière d'une définition de procédure classique. Un agent est soit une contrainte primitive (Tell), un agent conditionnel (Ask), une conjonction (composition parallèle), une quantification existentielle ou un appel de prédicat.

Par convention, la conjonction d'agents $A_1 \wedge A_2 \wedge, ..., A_n$ est plus souvent notée $A_1, A_2, ..., A_n$. Remarquons que la conjonction est définie à la fois au niveau des agents et au niveau des contraintes. Il est en de même pour la quantification existentielle.

Par exemple, sur le système de contraintes de Herbrand, considérons les programmes Append et Reverse écrits comme suit:

Append (X, Y, Z) : :
    X = [ ] → Y = Z,
    cons(X) → ∃ A ∃ B ∃ C. (X = [A | B], Z = [A | C], Append(B, Y, C)).

Reverse (X, Y) : :
    X = [ ] → X = Y,
    cons(X) → ∃ A ∃ B ∃ C. (X = [A | B], Reverse(B, C), Append(C, [A], Y)).

où cons(X) est équivalent à ∃ A ∃ B. X = [A | B].

La procédure Append concatène les deux listes X et Y et renvoie le résultat dans Z. Si la liste X est vide, alors Y est égale à Z. Si la liste X n'est pas vide, X est décomposé en [A | B]; Z est égalé à [A | C]; et un appel récursif est ensuite lancé avec B, Y et C.

La procédure Reverse inverse les éléments de la liste X et renvoie le résultat dans Y. Si la liste X est vide, alors Y est égale à X. Si la liste X n'est pas vide, X est décomposé en [A | B]; un appel récursif est lancé avec B et C; et [A] est ensuite concaténé à C pour former Y.

Il faut remarquer que, malgré une ressemblance trompeuse, le programme Append donné ci-dessus ne se comporte pas exactement comme le programme Append habituel en Prolog (il en est de même pour le programme Reverse). Par exemple, contrairement au programme Prolog, le programme CC ne donnera aucun résultat si X n'est pas instancié. Les tests X = [ ] et Cons(X) vont suspendre indéfiniment. De plus les tests n'instancient pas. Par exemple, après avoir testé Cons(X), il faut répéter $\exists$ A $\exists$ B. X = [A | B] contrairement au programme Prolog.

De plus, c'est parce que les tests X = [ ] et Cons(X) sont mutuellement exclusifs que nous pouvons utiliser une conjonction entre les deux agents correspondant aux deux clauses du programme Append de Prolog. En effet aucun appel n'exécutera les deux agents. Dans le cas de tests non exclusifs, il faudrait utiliser une disjonction et non une conjonction sans quoi les agents, qui devraient être exécutés indépendamment, interagiraient sur le même store. Avec la disjonction, nous pourrions également réaliser une traduction exacte du programme Prolog en un programme CC (voir plus loin).

3.2. Sémantique opérationnelle:

Soit un système de contraintes (D, |-), V l'ensemble des variables d'un programme donné et Env le domaine des environnements. Nous appelons environnement d'un programme l'ensemble des associations symboles de prédicats / agents définies par les déclarations du programme. La sémantique opérationnelle des langages CC déterministes est définie par sept règles de transitions définies sur Env $\times$ (D_V $\times$ A) $\times$ (D_V $\times$ A). Un état de transition est constitué d'un store de D_V et d'un agent.

Nous notons une transition e : (c, A) $\rightarrow$ (d, B) pour signifier qu' un agent A exécuté sur un store c dans l'environnement e, augmente le store à d et se comporte ensuite comme B. L'environnement e est omis lorsqu'il n'intervient pas.

3.2.1. Tell:

L'exécution d'un agent c consiste à ajouter la contrainte c au store en un seul pas:

$$(d, c) \rightarrow (d \cup c, true) \quad \text{si } c \neq true \quad (1)$$

Par exemple, l'agent X = a exécuté sur le store vide donne le store {X = a}:

$(\{\ \}, X = a) \rightarrow (\{X = a\}, true)$.

L'agent true est celui qui ne fait rien, la règle ne s'y applique pas pour assurer la terminaison du programme comme elle est définie dans la suite. Remarquons que nous pourrions définir la terminaison autrement en disant que nous nous arrêtons si les seules transitions encore possibles sont de la forme (d, true) $\rightarrow$ (d, true).

3.2.2. Ask:

Un agent $c \rightarrow A$, se réduit à l'agent A si la contrainte c est impliquée par le store.

$(d, c \rightarrow A) \rightarrow (d, A)$ si $d \vdash c$ (2)

Par exemple, l'agent $Cons(X) \rightarrow p(X)$ exécuté sur le store $\{X = [a| Y]\}$ réussit car $\{X = [a| Y]\}$ implique Cons(X):

$(\{X = [a| Y]\}, Cons(X) \rightarrow p(X)) \rightarrow (\{X = [a| Y]\}, p(X))$.

Nous disons que l'agent $Cons(X) \rightarrow p(X)$ exécuté sur le store $\{X = a\}$ échoue car $\{X = a\}$ est inconsistant avec Cons(X). Nous disons que le même agent exécuté sur le store vide $\{\ \}$ suspend car $\{\ \}$ n' implique pas Cons(X) mais est consistant avec Cons(X). Dans les deux cas, la règle de transition (2) ne s'applique pas.

Nous ajoutons à cet effet la règle suivante:

$(d, c \rightarrow A) \rightarrow (d, true)$ si $d \vdash (\neg c)$ (2')

C'est à dire qu'un agent $c \rightarrow A$, se réduit à l'agent true si la négation de la contrainte c est impliquée par le store, c est alors inconsistante avec le store. Ainsi, un agent qui échoue est enlevé du système. Nous pouvons dès lors distinguer une situation où un agent est suspendu d'une situation où un agent échoue.

3.2.3. Conjonction:

L'exécution d'un agent composé A ∧ B est définie en terme de l'exécution de ses constituants:

$$\frac{(c,A) \rightarrow (d, A')}{\begin{array}{l}(c, A \wedge B) \rightarrow (d, A' \wedge B) \\ (c, B \wedge A) \rightarrow (d, B \wedge A')\end{array}} \qquad (3)$$

La règle reflète le fait que A et B ne communiquent jamais de façon synchrone dans A ∧ B. Un agent ajoute de l'information dans le store pour que l'autre l'utilise.

Par exemple, l'agent (X = a ∧ Y = b) exécuté sur le store vide { } donne le store {X = a, X= b}. Les deux séquences suivantes sont possibles selon que X = a ou Y = b est exécuté en premier:

$$(\{\ \}, X = a \wedge Y = b) \rightarrow (\{X = a\}, Y = b) \rightarrow (\{X = a, Y = b), true\}$$

$$(\{\ \}, X = a \wedge Y = b) \rightarrow (\{X = b\}, Y = a) \rightarrow (\{X = a, Y = b), true\}$$

Remarquons que le résultat est pareil dans les deux cas.

Par exemple, l'agent (X = [ ]) ∧ (Cons(X) → Y = Z) exécuté sur le store vide donne le store {X = [ ], Y = Z}. Ici une seule séquence est possible car l'agent (Cons(X) → Y = Z) attend que le store implique Cons(X), ce qui est le cas après l'exécution de l'agent X = [ ]:

$$(\{\ \}, (X = [\ ]) \wedge (Cons(X) \rightarrow Y = Z) \rightarrow (\{\ X = [\ ]\}, Cons(X) \rightarrow Y = Z)$$

$$\rightarrow (\{\ X = [\ ]\ \}, Y = Z) \rightarrow (\{X = [\ ], Y = Z\}, true)$$

3.2.4. Quantification existentielle:

La quantification existentielle permet la définition de variables locales à un agent. Un même identificateur peut dès lors représenter à la fois une variable globale et une variable locale à un agent.

L'exécution d'un agent $\exists X.$ A est définie par l'exécution de l'agent A avec une variable X locale. C'est à dire que toutes les interactions de A sur X sont cachées aux autres agents concurrents.

Pour définir cette transition il faut malheureusement introduire des agents de la forme $\exists X.$ (d,A) où d est un store local contenant des informations sur la variable X. La quantification existentielle d'un store est définie par $\exists X.(c_1, ..., c_n) = (\exists X. c_1 \wedge ... \wedge c_n)$.

$$\frac{(\exists X.c,\ A) \rightarrow (d,B)}{(c,\ \exists X.A) \rightarrow (c \cup \exists X.d,\ \exists X(d,B))} \qquad (4)$$

L'agent A est exécuté le store $\exists X.c$ de telle façon que la variable X globale lui soit cachée; le store est augmenté à d et l'agent A est réduit à B. Le store global est mis à jour avec $\exists X.d$ de telle façon que les interactions de A avec la variable X locale lui soit cachées. Par contre d est conservé dans le store local à l'agent B.

La règle suivante concerne les agents avec un store local:

$$\frac{(d \cup \exists X.c,\ A) \rightarrow (d',\ B)}{(c,\ \exists X(d,A)) \rightarrow (c \cup \exists X.d',\ \exists X(d',\ B))} \qquad (5)$$

Le store local d est combiné au store $\exists X.c$ de telle façon que la variable X globale lui soit cachée; le store est augmenté à d' et l'agent A est réduit à B. Le store global est mis à jour avec $\exists X.d'$ de telle façon que les interactions de A avec la variable X locale lui soit cachées. Par contre d' est conservé dans le store local à l'agent B.

Remarquons que la quantification existentielle est une réplique au renommage utilisé en Prolog. Le renommage consiste à renommer les variables des clauses avant de les utiliser dans la réduction pour éviter d'introduire de nouvelles variables portant un nom déjà utilisé par d'autres et ainsi éviter les confusions que cela pourrait provoquer. Pareillement, la quantification existentielle des variables permet l'introduction de nouvelles variables mais ici, plutôt que d'utiliser de nouveaux noms de variables, nous manipulons le store (en cachant tour à tour les anciennes et les nouvelles variables) pour éviter les confusions que cela pourrait provoquer.

Par exemple, considérons l'exécution de l'agent $\exists$ X. (Z = [Y| X], p(X)) sur le store c = {X = a, Y = b}. Nous utilisons la règle (4):

$(\{X = a, Y = b\}, \exists X. (Z = [Y| X], p(X)))$

$\rightarrow (\{X = a, Y = b, \exists X. Z = [b |X]\}, \exists X. (\{Y = b, Z = [b| X]\}, p(X)))$

D'abord, X est caché à c. Cela revient à exécuter l'agent (Z = [Y| X], p(X)) sur le store $\exists X. c = \{Y = b\}$.

Supposons que (Z = [Y| X]) est exécuté en premier, cela revient à exécuter l'agent p(X) sur le store $d = \{Y = b, Z = [b| X]\}$.

Ensuite, X est caché à d et le résultat est ajouté à c. Cela revient à exécuter l'agent $\exists X. (d, p(X))$ sur le store $c \cup \exists X. d = \{X = a, Y = b, \exists X. Z = [b |X]\}$.

Pour poursuivre, nous utilisons la règle (5). Considérons l'exécution de l'agent $\exists X. (d, p(X))$, où $d = \{Y = b, Z = [b| X]\}$, sur le store $c = \{X = a, Y = b, \exists X. Z = [b |X]\}$.

$(\{X = a, Y = b, \exists X. Z = [b |X]\}, \exists X. (\{Y = b, Z = [b| X]\}, p(X)))$

$\rightarrow (\{X = a, Y = b, Z = [b| [ ]]\}, true)$

D'abord, X est caché à c et le résultat est ajouté à d. Cela revient à exécuter l'agent p(X) sur le store $d \cup \exists X. c = \{Y = b, Z = [b| X]\}$.

Supposons que p(X) est X = [ ]. La réduction de p(X) donne le store $d' = \{Y = b, Z = [b| [ ]], X = [ ]\}$ et l'agent résiduel true.

Ensuite, X est caché à d' et le résultat est ajouté à c. Cela donne l'état composé du store $c \cup \exists X. d' = \{X = a, Y = b, Z= [b| [ ]]\}$ et de l'agent true.

Remarquons qu'avec un renommage du X local en X' nous aurions simplement:

$(\{X = a, Y = b\}, \exists X. (Z = [Y| X], p(X)))$

$\rightarrow (\{X = a, Y = b, Z = [b |X']\}, p(X'))$

$\rightarrow (\{X = a, Y = b, Z = [b| [ ]]\}, true)$

3.3.5. Appel de prédicat:

Un appel de prédicat $p(\overline{X})$ est réalisé en utilisant la définition du prédicat dans l'environnement:

$$e : (d, p(\overline{X})) \rightarrow (d, \exists\, \alpha.\, (\overline{X} = \overline{\alpha} \wedge e(p)) \qquad (6)$$

où e(p) est l'agent associé à p dans l'environnement e.

Les variables $\overline{\alpha}$ sont utilisées pour réaliser le passage de paramètre. Elles sont réservées à cet usage et interdites dans un programme utilisateur. Dans e(p), elles sont systématiquement égalées aux paramètres formels (voir (8) plus loin). Lors d'un appel $p(\overline{X})$, les variables $\overline{X}$ sont à leur tour égalées aux $\overline{\alpha}$. Les variables $\overline{X}$ sont ainsi égalées aux paramètres formels.

Par exemple, considérons l'agent p(X) et la déclaration p(Y) :- Y = a. Par (8), nous avons e[p] = $\exists$Y. (Y = α, Y = a).

Considérons l'exécution de p(X) sur le store d = {Y = b}. Cela donne la transition suivante:

$$(\{Y = b\}, p(X)) \rightarrow (\{Y = b\}, \exists\alpha.\, (X = \alpha, e[p]))$$

Poursuivons l'exécution:

$$(\{Y = b\}, \exists\alpha.\, (X = \alpha, e[p])) \rightarrow \ldots \rightarrow (\{Y = b, X = a\}, true)$$

D'abord, α est caché à d. Cela revient à exécuter (X = α, e[p]) sur le store $\exists\alpha$. d = {Y = b}.

Supposons que X = α est exécuté en premier, cela donne le store d' = {Y = b, X = α} et l'agent résiduel e[p] = $\exists$Y. (Y = α, Y = a).

De la même façon, Y est caché à d'. Cela revient à exécuter (Y = α, Y = a) sur le store $\exists$Y. d' = {X = α}.

L'exécution de Y = α et Y = a donne le store d'' = {X = Y= α = a} et l'agent résiduel true.

Ensuite, Y puis α sont cachés de d'' et le résultat est ajouté à d. Cela donne finalement l'état composé du store $d \cup \exists\ \alpha.(\exists\ Y.\ d'') = \{Y = b, X = a\}$ et de l'agent true.

3.2.6. Déclaration:

Les définitions suivantes permettent d'extraire un environnement à partir des déclarations d'un programme:

$$R(\varepsilon)\ e = e \qquad (7)$$

$$R(p(\overline{Y}) :: A)\ e = e\ [p \mapsto \exists \overline{Y}\ ((\overline{Y} = \overline{\alpha}) \wedge A)] \qquad (8)$$

$$R(D.D)\ e = R(D)\ (R(D)\ e) \qquad (9)$$

La règle (8) signifie que l'association du prédicat p à l'agent $\exists \overline{Y}\ ((\overline{Y} = \overline{\alpha}) \wedge A)$ est ajoutée à l'environnement. Le passage par les variables $\overline{\alpha}$ permet essentiellement d'éliminer l'emploi arbitraire des variables $\overline{Y}$ dans la déclaration de p.

3.2.7. Programme:

L'exécution d'un programme sur un store initial c est définie par une c-séquence de transitions. Une c-séquence de transitions pour un programme D.A est une séquence (finie ou infinie) d'états de transitions (c_i, A_i) telle que $c_0 = c$ et $A_0 = A$ et pour tout i :

$$R(D)\ e_0 : (c_i, A_i) \rightarrow (c_{i+1}, A_{i+1})$$

où e_0 représente l'environnement vide.

Une telle séquence de transitions est dite terminale si elle est finie de longueur n et que l'agent A_{n-1} est bloqué dans c_{n-1}. C'est à dire que plus aucune règle de transition n'est applicable. Dans ce cas, c_{n-1} est appelé le store final.

Le lemme suivant exprime une propriété fondamentale des agents déterministes:

Lemme : Si un agent déterministe a plus d'une transition possible dans un store donné, alors elles commutent. C'est à dire qu' en les combinant dans n'importe quel ordre, le résultat obtenu est pareil.

Le théorème suivant peut être prouvé en faisant appel à la commutativité:

Théorème de confluence: pour tout store c et pour tout programme déterministe D.A, si D.A possède une c-séquence de transitions terminale avec un store final d, alors D.A ne possède pas de c-séquence de transitions infinie. De plus, toutes les c-séquences de transitions ont le même store final.

C'est à dire qu'un programme CC déterministe produit toujours le même store final pour un store initial donné.

3.3. Sémantique dénotationnelle:

Soit un système de contraintes arbitraire (D, |-) et V l'ensemble des variables d'un programme donné. Nous avons vu que l'exécution d'un programme CC déterministe donne un store final unique. Pour un agent A, considérons la fonction f : $D_V \rightarrow D_V$ qui associe un store c au store d obtenu en exécutant A sur c, et à 'false' si l'exécution est infinie.

La sémantique dénotationnelle des langages CC déterministes consiste à définir les agents en termes des fonctions sur $D_V \rightarrow D_V$ qui leur correspondent. Il apparaît que ces fonctions sont des opérateurs de fermetures. Un opérateur de fermeture sur un ordre partiel est une fonction extensive, monotone et idempotente.

En effet, soit une fonction f correspondant à un agent A:

> (i) La seule façon pour A d'affecter un store est d'y ajouter de l'information; f est donc extensive.
> (ii) Si l'exécution de A sur un store c se termine avec un store d, alors d est un point fixe de f, c'est à dire que l'exécution ne peut pas continuer sur d (sinon, l'exécution ne se serait pas terminée sur d); f est donc idempotente.
> (iii) Finalement, supposons deux stores en entrée s_1 et s_2 tels que $s_1 \leq s_2$. Si f correspond à ajouter une contrainte, alors puisque s_1 et s_2 reçoivent la même information, l'ordre est préservé sur les stores résultants; f est donc monotone.

La propriété la plus importante d'un opérateur de fermeture f est qu'il est totalement défini par l'ensemble de ses points fixes. En effet, l'idempotence signifie que le rang de f est égal à l'ensemble de ses points fixes; l'extensivité, que chaque point du domaine est associé à un point fixe plus grand que lui; et la monotonie que ce dernier est le plus petit de tels points fixes. Beaucoup de

définitions importantes sur les opérateurs de fermeture ont des définitions assez simples en termes de points fixes.

Cependant, dans la suite, nous ne définissons pas les dénotations des opérateurs de fermetures en termes de leurs points fixes, mais directement comme des fonctions sur des stores car cela peut suggérer une implémentation directe de la sémantique:

Soit Env, le domaine des environnements. Agent (A), déclaration (D) et programme (P) sont définis par les dénotations suivantes:

$$[A] : Env \rightarrow D_V \rightarrow D_V$$

$$[D]_{Dec} : Env \rightarrow Env$$

$$[P]_{Prog} : D_V$$

3.3.1. Tell:

Un agent c est défini par la fonction qui associe un store s au store $s \cup c$:

$$[c]\, e = \lambda s.\ s \cup c \qquad (1)$$

3.3.2. Ask:

Un agent $c \rightarrow A$ se comporte comme l'agent A si le store implique c, sinon il ne fait rien. Un tel agent peut être décrit par la fonction:

$$[c \rightarrow A]\, e = \lambda s.\ \text{if } s \vdash c \text{ then } [A]\, e\, s \text{ else } s. \qquad (2)$$

3.3.3. Conjonction:

L'exécution parallèle de deux agents A et B peut être décrite comme suit: A et B sont exécutés simultanément sur un store initial c; A produit un store [A] e c et B produit un store [B] e c. Il se peut maintenant que l'information apportée par l'un des agents permette à l'autre de redémarrer et ainsi de suite. Le système s'arrête lorsque à la fois les deux agents s'arrêtent.

$$[A \wedge B]\, e = \lambda s.\ \mu c.\ (\,[A]\, e\, c) \cup (\,[B]\, e\, c) \cup s \qquad (3)$$

La principale propriété exploitée ici est la possibilité de redémarrer un processus déterministe. Supposons un agent A exécuté sur un store initial c, qui produit un store d et s'arrête laissant un agent résiduel B. En fait, pour déterminer l'effet de l'agent B sur un store e ≥ d, il suffit d'exécuter l'agent initial A sur e. Il n'est pas nécessaire de maintenir une représentation de B dans la dénotation de A.

3.3.4. Quantification existentielle:

L'application d'un agent $\exists X.$ A sur un store s est définie par l' application la dénotation de A sur $\exists X.s$, de telle façon que la variable X globale soit cachée à A; ensuite X est à nouveau caché du résultat, de telle façon que les interactions de A sur la variable X locale soient cachées à s.

$$[\exists X.\ A]\ e = \lambda s.\ s \cup \exists X.\ [A]\ e\ (\exists X.\ s) \qquad (4)$$

3.3.5. Appel de prédicat:

Un appel de prédicat $p(\overline{X})$ sur un store s est défini par l'application de la dénotation de l'agent associé à p dans l'environnement sur le store $((\overline{X} = \overline{\alpha}) \cup s)$, de telle façon que les variables $\overline{\alpha}$ soient égalées aux paramètres $\overline{X}$; les variables $\overline{\alpha}$ sont ensuite cachées du résultat.

$$[p(X)]\ e = \lambda s.\ (\ \exists \alpha\ [e(p)]\ e\ ((X = \overline{\alpha})\ \cup s)) \qquad (5)$$

où e(p) désigne l'agent associé à p dans l'environnement e.

Les variables $\overline{\alpha}$ sont utilisées pour réaliser le passage de paramètre. Elles sont réservées à cet usage et interdites dans un programme utilisateur. Dans (7), elles sont systématiquement égalées aux paramètres formels. Ainsi, lors de l'appel ces derniers sont égalés aux $\overline{X}$.

3.3.6. Déclaration:

La déclaration vide ε est définie par le neutre sur Env:

$$[\varepsilon]_{\,Dec}\ e = e \qquad (6)$$

Une déclaration $p(\overline{Y}) :: A$ sur un environnement e est définie par la fonction qui ajoute la définition de p à e:

$$[p(\overline{Y}) :: A]_{Dec}\, e = e\,[p : \exists \overline{Y}\ ((\overline{Y} = \overline{\alpha}) \wedge A)] \qquad (7)$$

$e(p) = \exists \overline{Y}\ ((\overline{Y} = \overline{\alpha}) \wedge A)$ de telle façon que les paramètres formels $\overline{Y}$ soient égalés aux variables $\overline{\alpha}$.

Une déclaration $D_1 .D_2$ est définie par la composition des dénotations de D_1 et D_2:

$$[D_1 .D_2]_{Dec}\, e = [D_1]_{Dec}\, e \cup [D_2]_{Dec}\, e \qquad (8)$$

3.3.7. Programme:

Un programme D. A est défini par l'application de la dénotation de A sur l'environnement obtenu en appliquant la dénotation de D à l'environnement vide [ ] et le store vide { }:

$$[D.\ A]_{Prog} = [A]\ ([D]_{Dec}\ [\,])\ \{\,\} \qquad (9)$$

4. Les langages CC non-déterministes:

Les langages CC non-déterministes étendent les langages CC déterministes avec l'opérateur de disjonction. La disjonction consiste en l'exécution indépendante de plusieurs agents, sur des stores différents. Elle correspond en fait au choix entre plusieurs clauses en Prolog .

Les programmes CC non-déterministes produisent un ensemble de résultats. En effet, il y a maintenant plusieurs choix possibles dans un programme et chacun d'eux donne lieu à un résultat.

D'abord, nous présentons la syntaxe et la sémantique opérationnelle des langages CC non-déterministes sur base de [11] de V. Saraswat. Ensuite, nous présentons la sémantique dénotationnelle des langages CC non-déterministes sur base des dénotations proposées par C. Codognet et P. Codognet dans [2]. A nouveau, nous présentons les deux sémantiques dans un but explicatif.

4.1. Syntaxe:

Les langages CC non-déterministes admettent en plus des langages CC déterministes des agents de la forme:

$$A ::= A \vee A$$

C'est à dire qu'un agent peut également être une disjonction d'agents.

Par exemple, sur le système de contraintes de Herbrand, un processus constructeur de liste peut s'écrire:

$$p(X) :: (X = [\,]) \vee \exists Y.\, (X = [a \mid Y], p(Y))$$

La procédure p construit une liste de constantes a. X est soit égalé à la liste vide, soit égalé à la liste X = [a | Y] et dans ce cas un appel récursif est ensuite lancé sur Y.

4.2. Sémantique opérationnelle:

Soit un système de contraintes (D, |-), V l'ensemble des variables d'un programme donné et Env le domaine des environnements. Pour définir la sémantique opérationnelle des langages CC non-déterministes, nous prenons cette fois

comme état de transition un ensemble de paires constituées d'un store de D_V et d'un agent. Intuitivement, un état de transition représente un ensemble de sous-systèmes indépendants.

La relation de transition est facilement étendue à Env $\times 2^{D_V} \times 2^{D_V}$ en prenant:

$$\{(c_1, A_1), ..., (c_i, A_i), ..., (c_n, A_n)\} \rightarrow \{(c_1, A_1), ..., (c'_i, A'_i), ..., (c_n, A_n)\}$$

ssi $(c_i, A_i) \rightarrow (c'_i, A'_i)$

C'est à dire que le calcul se poursuit indépendamment sur chaque sous-système.

4.2.1. Disjonction:

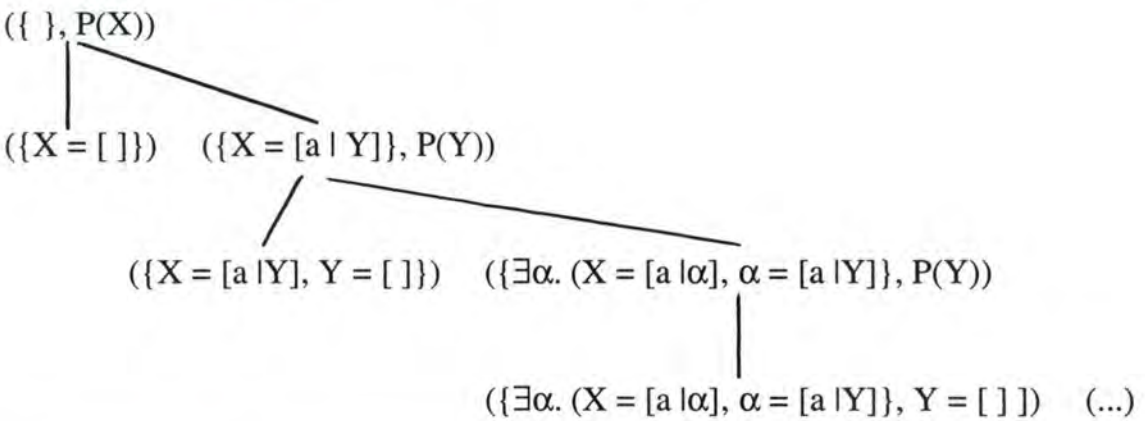
La disjonction est définie par la règle suivante:

$$\{(c, A \vee B)\} \rightarrow \{(c, A), (c, B)\} \quad (10)$$

C'est à dire qu'un agent $A \vee B$ exécuté sur un store c se divise en deux sous-systèmes (c, A) et (c, B) exécutés indépendamment.

Pratiquement, le traitement de la disjonction revient à effectuer une recherche avec backtracking. Plutôt qu'une séquence de transitions pour un programme, nous avons maintenant un arbre de transitions dont chaque branche correspond à un choix possible dans le programme. Lorsque le store d'une branche devient inconsistant, elle disparaît. Le résultat de l'exécution est l'ensemble des stores obtenus sur chaque branche.

Par exemple, l'exécution du programme p(X), donné dans le paragraphe précédent, sur le store initial vide est donnée par l'arbre suivant:



Ce qui donne les résultats: {X = [ ]}, {X = [a | [ ]]}, {X = [a | a |[ ]]}, ...

4.3. Sémantique dénotationnelle:

Soit un système de contraintes (D, |-), V l'ensemble des variables d'un programme donné et Env le domaine des environnements. Pour définir la sémantique dénotationnelle des langages CC non-déterministes, nous considérons les agents comme des fonctions prenant des ensembles de stores, et non plus des stores, en entrée et en sortie.

Les agents non-déterministes peuvent à nouveau être définis comme un certain type d'opérateurs de fermetures, appelés opérateurs linéaires non-déterministes. Ces opérateurs peuvent également être représentés par l'ensemble de leurs points fixes mais ici il n'est pas garanti qu'il y a un point fixe minimal plus grand que chaque entrée. Il faut parler d'un ensemble de points fixes minimal plus grand qu'une entrée.

La dénotation des agents est étendue à:

$$[A] : Env \rightarrow 2^{D_v} \rightarrow 2^{D_v}$$

Les dénotations données précédemment sont généralisées à 2^{D_v} en prenant $S \cup S' = \{s \cup s'.\ s \in S \text{ et } s' \in S'\}$, excepté pour le Ask, où il faut considérer:

$$[c \rightarrow A]\ e = \lambda S.\ \cup_{s \in S}\ (\text{ if } s \vdash c \text{ then } [A]\ e\ s \text{ else } s\ ) \quad (2)$$

C'est à dire que le calcul se poursuit indépendamment sur chaque store s de S.

4.3.1. Disjonction:

La disjonction est définie par:

$$[A_1 \vee A_2]\ e = [A_1]\ e \cup [A_2]\ e \quad (10)$$

C'est à dire que la dénotation d'un agent $A_1 \vee A_2$ consiste à prendre l'ensemble des résultats de l'application des dénotations des agents A_1 et A_2.

5. Exemples de programmes:

5.1. Nombres de Fibonnaci:

Le programme fib(N, NS) renvoie dans NS la liste des nombres de Fibonacci plus petits ou égaux à N. Remarquons que dans fib', nous pouvons à nouveau utiliser une conjonction plutôt qu'une disjonction entre les agents car les tests sont mutuellement exclusifs.

fib (N, NS) :: fib' (N, 0, 1, NS).

fib' (N, N1, N2, NS) ::
(N < N1) → (NS = [ ]),
(N ≥ N1) → ∃ NS' ∃ N3. (NS = [N1 | NS'],
N3 = N1 + N2, fib' (N, N2, N3, NS')).

5.2. Processus producteur / consommateur:

Le programme suivant schématise une interaction producteur / consommateur entre deux agents.

producteur (X) :: (X = [ ]) ∨
∃ Y. (X = [m | Y], producteur(Y))

consommateur (X) :: (X = [ ] → true) ∨
∃ Y. ∃ M. (X = [M | Y] → consommateur(Y))

La variable X sert de buffer. L'agent producteur instancie X avec une liste de messages m. L'agent consommateur lit les éléments de X via une variable M. Remarquons que pour chaque message, le consommateur suspend jusqu'à ce que le producteur instancie le début de la liste X.

5.3. Mélange non-déterministe:

Dans le programme suivant, les agents p(X) et q(Y) renvoient respectivement une liste de constantes a et b dans X et Y et l'agent merge (In1, In2, Out) renvoie dans Out une liste qui est un mélange des éléments des listes In1 et In2.

```
p(X) :: (X = []) ∨ (∃X'. X = [a | X'], p(X'))

q(Y) :: (Y = []) ∨ (∃Y'. Y = [b | Y'], q(Y'))

merge(In1, In2, Out) ::
        (In1 = [ ] → In2 = Out)
        ∨ (In2 = [ ] → In1 = Out)
        ∨ (cons(In1) → ∃ X ∃ In1' ∃ Out'. ( In1 = [ X | In1'] ,
                                Out = [ X | Out'], merge(In1', In2, Out'))
        ∨ (cons(In2) → ∃ X ∃ In2' ∃ Out'. ( In2 = [ X | In2'] ,
                                Out = [ X | Out'], merge(In1, In2', Out')).
```

Considérons l'exécution de l'agent merge (In1, In2, Out). Si In1 = [ ], alors Out est égalé à In2. Pareillement si In2 = [ ], alors Out est égalé à In1. Si In1 est une liste non vide, alors Out est égalé à la liste constituée du premier élément de In1 et d'une nouvelle variable Out', et un appel récursif est lancé avec (In1', In2, Out') où In1' est la liste obtenue en enlevant X à In1. Pareillement, si In2 est une liste non vide, alors Out est égalé à la liste constituée du premier élément de In2 et d'une nouvelle variable Out', et un appel récursif est lancé avec (In1, In2', Out') où In2' est la liste obtenue en enlevant X à In2.

5.4. Processus serveur de compte:

Le programme suivant schématise une interaction client / serveur dans le cas d'un serveur de compte. Celui ci reçoit une liste de commandes sur le compte d'un client et les exécute une à une.

```
counter(In) :: counter'(In, 0).

counter'(In, C) ::
        ∃ In'. (In = [clear| In'] → counter'(In', 0)),
        ∃ In'. (In = [add| In'] → ∃ C'.((C' = C + 1), counter'(In', C'))),
        ∃ In'. (In = [read(X)| In'] → (X = C), counter'(In', C)),
        In = [ ] → true.
```

L'agent counter(In) reçoit dans In une liste de commandes de la forme clear, add ou read(X). L'agent counter(In) initialise le compte C à 0 et lance l'agent counter'(In, C). Si In = [clear| In'] alors un appel récursif est lancé avec (In', 0). Si In = [add| In'] alors une variable C' est égalée à C + 1 et un appel récursif est lancé avec (In', C'). Et si In = [read(X)| In'] alors la variable X est égalée à C et ainsi communiquée au client et un appel récursif est lancé avec (In', C)).

5.5. Processus d'exclusion mutuelle:

Le programme suivant schématise un processus d'exclusion mutuelle simpliste qui gère l'accès à une ressource non-partageable.

$$p(X, Y) :: (X = lock(granted) \rightarrow Y = p) \vee (X = lock(denied) \rightarrow true).$$

$$q(X, Y) :: (X = lock(granted) \rightarrow Y = q) \vee (X = lock(denied) \rightarrow true).$$

$$mutex(X) :: \exists X'. (X = [lock(Reply) \mid X'] \rightarrow Reply = granted, mutex'(X')).$$

$$mutex'(X') :: \exists X'. (X = [lock(Reply) \mid X'] \rightarrow Reply = denied, mutex'(X')) \vee (X = [\,] \rightarrow true).$$

$$faster(X) :: \exists R1 \exists R2 \exists L. (p(lock(R1), X), q(lock(R2), X), merge ([lock(R1)], [lock(R2)], L), mutex (L)).$$

L'agent mutex(X) reçoit une liste de demande de lock(Reply) dans X. Il répond à la première demande en égalant Reply à granted et lance l'agent mutex'(X') où X' est obtenue en enlevant la première demande de X. L'agent mutex'(X') répond tour à tour aux autres demandes en égalant Reply à denied.

L'agent faster(X) lance mutex avec une demande de lock pour les agents p et q. Ceux-ci attendent la réponse de l'agent mutex via l'instanciation de leur variable X. Ensuite celui qui reçoit le granted égale la variable X de faster avec son nom.

Chapitre 3 : Interprétation abstraite des langages CC

L'interprétation abstraite est un modèle général pour l'analyse statique de programmes. L'analyse statique couvre l'ensemble des traitements que l'on peut appliquer à un programme en dehors de son exécution, par exemple au cours de la compilation. Les informations obtenues serviront à l'optimiser ou à vérifier s'il rempli certains critères de corrections.

Le terme interprétation abstraite a été introduit par P. Cousot et R. Cousot (voir [1]). L'idée essentielle de l'interprétation abstraite est de réaliser l'analyse statique d'un programme en l'exécutant sur un domaine particulier, dit abstrait car il abstrait seulement certaines propriétés pertinentes du domaine concret d'exécution. A partir de cette idée, beaucoup de modèles d'interprétation abstraite ont été développés pour différents types langages et plus particulièrement pour les langages logiques. Ceux-ci offrent en effet de nombreuses opportunités d'optimisation. Par exemple, nous citons les analyses de mode (groundness), de type et de détection de procédures déterministes. Dans le cas des langages CC, il est également intéressant d'analyser les communications entre agents et en particulier de détecter les cas d'interblocage.

Dans ce chapitre, nous illustrons d'abord brièvement la notion d'interprétation abstraite. Ensuite, nous présentons le modèle d'interprétation abstraite des langages CC proposé par C. Codognet et P. Codognet dans [2]. Puis, indépendamment de toute analyse, nous exposons les problèmes rencontrés lors de la mise en oeuvre de ce modèle et nous présentons brièvement le domaine des abstractions de niveau k. Finalement, nous nous penchons plus particulièrement sur l'application du même modèle au cas de l'analyse de détection de blocage des programmes CC.

1. L'interprétation abstraite:

Comme nous l'avons déjà introduit, le principe de base de l'interprétation abstraite est de réaliser l'analyse statique d'un programme en l'exécutant sur un domaine particulier, dit abstrait car il abstrait seulement certaines propriétés voulues du domaine concret d'exécution.

C'est à dire que le programme est exécuté avec des représentations de données plutôt qu'avec les données elles-mêmes. Et les opérations qui servent à exécuter le programme sont remplacées par des opérations, sur les représentations des données, qui les approximent de façon consistante. C'est à dire que le résultat obtenu de ces opérations doit être une représentation du résultat obtenu des opérations normales. Ainsi, le résultat obtenu de cette exécution est lui-même une approximation du résultat de l'exécution normale.

Certaines règles de calcul bien connues sont des applications de ces idées. Par exemple la règle des signes pour la multiplication algébrique revient à réduire l'ensemble des nombres à l'ensemble abstrait des signes {+, −}. La multiplication des nombres peut être représentée par la règle des signes:

*	+	−
+	+	−
−	−	+

Cette règle permet de calculer le signe d'un produit sans effectuer réellement ce produit.

Un autre exemple est donné par la preuve par 9 des opérations arithmétiques. En raisonnant sur le reste de la division par 9 des nombres, nous pouvons calculer le reste du résultat et détecter une erreur avec une probabilité de 8/9.

Considérons l'opération $x * y = z$. Chaque nombre x est représenté par $x' = x \bmod 9$. Et la multiplication sur les valeurs abstraites est définie par $x' *' y' = (x' * y') \bmod 9$. Nous pouvons facilement calculer $x' *' y' = z'$. Alors, si $z \bmod 9 \neq z'$, il y a une erreur.

Pour résumer, l'abstraction d'une opération consiste à abstraire les opérandes et appliquer l'opération abstraite pour obtenir un résultat qui est une abstraction du résultat concret.

Un modèle d'interprétation abstraite est une méthode générale, c'est à dire indépendante d'un type d'analyse, pour réaliser des interprétations abstraites d'un langage. Plusieurs démarches sont possibles pour définir de tels modèles.

La démarche habituelle pour développer un modèle d'interprétation abstraite consiste à partir de la sémantique du langage considéré, appelée sémantique standard, qui défini les opérations utilisées pour l'exécution des programmes dans ce langage. L'interprétation abstraite est obtenue de cette sémantique en interprétant ces fonctions sur un domaine abstrait. Donc une interprétation abstraite constitue une sémantique non-standard du langage. Certaines fonctions sont souvent ajoutées pour garantir la terminaison de l'exécution abstraite.

D'autres modèles d'interprétation abstraite reposent sur l'idée que l'interprétation abstraite doit simplement mimer l'exécution concrète sur le domaine abstrait. Le programme à analyser est d'abord systématiquement traduit en un programme abstrait qui exécute l'interprétation abstraite du premier. Le modèle que nous présentons dans la suite est de ce type. Dans [3], P. Codognet et G. Filé montrent comment l'analyse de mode du programme CLP(H) Append peut être réalisée en exécutant un programme CLP(Boole) avec un mécanisme de tabulation pour assurer la terminaison. Ces idées peuvent être étendues au cas des langages CC.

2. Abstraction des programmes CC:

Nous présentons le modèle d'interprétation abstraite des langages CC proposé par C. Codognet et P. Codognet dans [2]. Ce modèle est défini sur la sémantique dénotationnelle des langages CC présentée dans le chapitre précédent. Remarquons cependant que le modèle d'interprétation abstraite suit, par contre, une démarche opérationnelle.

L'interprétation abstraite des langages CC est définie de façon similaire à ce qui a été fait pour les langages logiques avec contraintes (CLP) dans [4], en introduisant la notion d'abstraction entre systèmes de contraintes. Si (D',|-') abstrait (D,|-), alors pour tout programme P sur (D,|-), P peut être syntaxiquement transformé en un programme P' sur (D',|-') de telle façon que l'exécution de P' est une approximation de celle de P et que donc l'analyse statique peut être effectuée en exécutant P'.

Cette idée de réaliser l'interprétation abstraite par la dérivation d'un programme abstrait explicite a d'abord été proposée dans la cadre de l'analyse de mode de programmes Prolog sur un domaine fini. Un mécanisme de tabulation avait été utilisé pour assurer la terminaison de l'exécution. En effet, lors de l'exécution du programme abstrait, il peut arriver qu'un nombre fini de valeurs soient indéfiniment répétées à cause de l'abstraction. La tabulation consiste à construire une table dans laquelle sont gardées à tout moment les valeurs déjà calculées et permet ainsi de détecter un tel cas de non terminaison.

Dans le cas les langages avec contraintes (CLP et CC), tout domaine abstrait peut être redéfini en terme de système de contraintes. De plus la sémantique est définie de façon générale, sur un système de contraintes générique et peut donc être utilisée à la fois pour l'exécution concrète et l'exécution abstraite.

Nous disons qu' un système de contraintes (D',|-') abstrait un système de contraintes (D, |-) ssi il existe une fonction γ: $D' \rightarrow 2^D$, dite fonction de concrétisation, telle que les quatres conditions suivantes soient remplient:

(i). γ est monotone : $\forall\, c_1, c_2 \in D'$: $(c_1 \vdash' c_2 \Rightarrow \gamma(c_1) \subseteq \gamma(c_2))$

C'est à dire que plus une contrainte abstraite est forte, plus sa concrétisation est petite. Par exemple, c |-' true implique que $\gamma(c) \subseteq \gamma(\text{true})$.

(ii). $\forall c \in D, \exists c' \in D' : (c \in \gamma(c'))$

C'est à dire que toute contrainte concrète doit posséder une représentation abstraite.

(iii). $\forall c'_1, c'_2 \in D', \forall c_1, c_2 \in D :$
$(c_1 \in \gamma(c'_1) \wedge c_2 \in \gamma(c'_2) \Rightarrow (c_1 \wedge c_2) \in \gamma(c'_1 \wedge c'_2))$

C'est à dire que si la contrainte abstraite c'_1 représente la contrainte c_1 et que la contrainte abstraite c'_2 représente la contrainte c_2, alors la contrainte abstraite $(c'_1 \wedge c'_2)$ représente la contrainte $(c_1 \wedge c_2)$.

(iv). $\forall S' \subseteq D', \forall c' \in D' :$
$(S' \vdash' c' \Rightarrow (\forall S \subseteq \gamma(S'), \forall c \in \gamma(c') : S \vdash c))$

C'est à dire que si un ensemble S' de contraintes abstraites implique une contrainte abstraite c', alors tous les ensembles S_i de contraintes concrètes, que S' représente et toutes les contraintes concrètes c_j, que c' représente sont tels que chaque S_i implique chaque c_j.

Soient deux systèmes de contraintes $(D, \vdash)$ et $(D', \vdash')$ tels que $(D', \vdash')$ abstrait $(D, \vdash)$ avec une fonction de concrétisation γ et un programme P sur $(D, \vdash)$. Nous construisons un programme abstrait P' sur $(D', \vdash')$ correspondant à P en remplaçant toutes les contraintes c dans P par des contraintes c' telles que $c \in \gamma(c')$.

Remarquons qu'il y a a priori plusieurs choix possibles pour remplacer les contraintes c par des contraintes c' telles que $c \in \gamma(c')$. Il s'agit de faire le choix le plus judicieux.

Il est facile de voir que l'exécution d'un programme abstrait P' abstrait l'exécution de P dans le sens où les contraintes résultant de l'exécution de P sont contenues dans la concrétisation des contraintes résultant de l'exécution de P':

Théorème: Soit $(D', \vdash')$ un système de contraintes qui abstrait un système de contraintes $(D, \vdash)$. Soit P un programme sur $(D, \vdash)$ et P' un programme abstrait correspondant sur $(D', \vdash')$. Alors:

$\forall s \in [\, P \,]_{Prog} \exists s' \in [\, P' \,]_{Prog}$ tel que $\exists \sigma \in \gamma(s') : s \vdash \sigma$.

Preuve: Les seuls points où les exécutions de P et de P' diffèrent sont lorsque les contraintes apparaissent: lors d'un Tell ou d'un Ask.

Pour le Tell, soit $c \in D$ et $c' \in D'$ telles que $c \in \gamma(c')$ et $s \in 2^{D_V}$ et $s' \in 2^{D'_{V'}}$ tels que $\exists \sigma \in \gamma(s')$: $s \vdash \sigma$. Considérons l'exécution de c sur le store s pour P et l'exécution de c' sur s' pour P'. Comme $c \in \gamma(c')$ et $\sigma \in \gamma(s')$, on a $\sigma \cup c \in \gamma(s' \cup c')$ par (iii) de la définition de l'abstraction. Et comme $s \vdash \sigma$, on a $s \cup c \vdash \sigma \cup c$ par monotonie du Tell. Donc on a $\exists \sigma \cup c \in \gamma(s' \cup c')$: $s \cup c \vdash \sigma \cup c$.

Pour le Ask, soit $c \in D$ et $c' \in D'$ telles que $c \in \gamma(c')$ et $s \in 2^{D_V}$ et $s' \in 2^{D'_{V'}}$ tels que $\exists \sigma \in \gamma(s')$: $s \vdash \sigma$. Considérons l'exécution de $c \rightarrow A$ sur le store s pour P et l'exécution de $c' \rightarrow A'$ sur le store s' pour P'. Par (iv) de la définition de l'abstraction, il se peut que $s \vdash c$ et pas $s' \vdash' c'$, c'est à dire que le Ask de P réussi et celui de P' suspend. Supposons que l'exécution de $c \rightarrow A$ sur le store s donne s''. Par monotonie du Ask, on a $s'' \vdash s$. Et comme $s \vdash \sigma$, on a $\exists \sigma \in \gamma(s')$: $s'' \vdash \sigma$ par transitivité de $\vdash$.

3. Le problème du Ask:

Dans ce qui suit, nous présentons les problèmes soulevés lors de la mise en oeuvre du modèle d'interprétation abstraite exposé précédemment. Pour les résoudre, nous devons malheureusement nous écarter quelque peu de ce modèle. Nous expliquons pourquoi et proposons plusieurs solutions.

3.1. Définition d'un système de contraintes abstrait:

Avant tout, remarquons que les programmes CC peuvent être analysés de deux façons; soit en examinant les agents au cours de l'exécution, soit en examinant uniquement le résultat du programme, c'est à dire le store final. La démarche que nous suivons est la seconde. En fait, dans le cas d'une analyse de détection de blocage, il semble que la première aurait été plus praticable.

Pratiquement, soit (D,|-) le système de contraintes concret, le problème qui se pose est de définir un système de contraintes abstrait (D',|-') et une fonction de concrétisation γ tels que (D',|-') abstrait (D,|-) et que les renseignements que nous puissions puisse tirer d'une exécution sur (D',|-') soient intéressants pour l'analyse considérée. Indépendamment de toute analyse, il apparaît difficile de trouver une traduction de contraintes qui satisfait à la fois les conditions (iii) et (iv) de la définition de l'abstraction.

Ces conditions garantissent respectivement la consistance de l'abstraction des opérations Tell et Ask, c'est à dire que la concrétisation du résultat un Tell abstrait doit comprendre le résultat de tous les Tell concrets qui lui correspondent. Et un Ask abstrait ne peut réussir que si tous les Ask concrets qui lui correspondent réussissent; sinon il doit échouer ou suspendre.

Par exemple, considérons le système de contraintes de Herbrand et l'abstraction qui consiste à examiner la fermeture (groundness) des termes. Un terme est clos (ground) si il ne contient pas de variable. Par exemple d'une contrainte X = a, nous retenons que X est clos. D'une contrainte X = Y, nous retenons que X est clos si Y est clos et inversement.

Dans le cas du Tell, nous pouvons imaginer de traduire, par exemple:

(X = a) par (ground(X))

En effet {X = A} appartient à la concrétisation de {ground(X)}.

Dans le cas du Ask, il apparaît qu'une traduction similaire n'est pas consistante, par exemple supposons que nous remplacions:

$(X = a) \rightarrow Q(X)$ par $(ground(X)) \rightarrow Q(X))$

Alors, il se peut que Ask (ground(X)) réussisse et que Ask (X = a) échoue, par exemple si X = b.

D'une manière générale, une traduction qui convient pour le Tell ne convient pas pour le Ask. Intuitivement, pour le Tell, il faut remplacer une contrainte par une contrainte plus générale. Pour le Ask, c'est le contraire, il faut remplacer une contrainte par une contrainte plus restrictive.

Dès lors, nous envisageons de traduire différemment les contraintes selon le fait qu'elles apparaissent dans un Tell ou dans un Ask pour vérifier à la fois la consistance du Tell et la consistance du Ask.

Intuitivement, il faut traduire un Ask par un Ask qui réussi moins ou aussi souvent pour une approximation correcte. Avec des contraintes sur les Réels, nous pourrions imaginer de traduire, par exemple:

$X \geq 5 \rightarrow Q(X)$ par $X \geq 10 \rightarrow Q(X)$

Mais le choix d'une telle traduction apparaît pour le moins délicat et l'intérêt peu évident. Aussi semble-t-il préférable pour la clarté de prendre l'option de ne pas traduire les contraintes qui apparaissent dans les Ask, la condition (iv) de la définition de l'abstraction est ainsi radicalement vérifiée. L'abstraction revient alors simplement à approximer les stores.

3.2. Précision du système de contraintes abstrait:

En fait, il faut admettre qu'il y a un conflit entre l'idée même d'approximer les stores et l'opération de Ask. Considérons par exemple, l'opération:

$(X = a) \rightarrow Q(X)$ sur le store $\{ground(X)\}$

Le test ne pourra jamais réussir car l'information nécessaire à la décision est perdue dans l'approximation. C'est à dire qu'il y aura toujours suspension. L'approximation d'un Ask concret par un Ask abstrait qui suspend a pour effet

d'ignorer certains agents (Q(X) notamment) qui auraient pu être lancés à la suite du Ask concret, et donc d'ignorer leur contribution au store résultant.

Un tel résultat peut être satisfaisant pour certaines analyses. Nous pourrions alors également envisager de définir le système de contraintes abstrait de telle façon qu'un Ask abstrait comme celui donné plus haut échoue ou réussisse mais ne lance pas Q(X). Nous pourrions imaginer de traduire par exemple:

$(X = a) \rightarrow Q(X)$ par $ground(X) \rightarrow true$

De cette façon, Q(X) n'affecte en aucun cas le store et l'opération est consistante. Cela permet d'exprimer qu' aucun Ask concret correspondant au Ask abstrait ne suspend si X est clos.

Il faut remarquer que pour d'autres analyses, comme la détection de blocage où nous nous intéressons au comportement du programme plutôt qu'au résultat, nous ne pouvons pas procéder ainsi. Pour la correction de l'analyse, le programme abstrait doit mimer exactement le programme concret et ne peut ignorer le lancement éventuel de certains agents à la suite d'un Ask.

Nous devons cependant admettre que l'exécution d'un programme abstrait avec l'approximation envisagée ici ne nous donnera pas beaucoup d'information car les Ask abstraits sont condamnés à suspendre. Pour une analyse valable, il faut utiliser un domaine abstrait qui garde assez d'information pour répondre aux Ask abstraits lorsque il y a lieu. La précision nécessaire à une bonne analyse dépendra non seulement du type d'analyse à effectuer mais également de la nature des Ask du programme à analyser, s'ils portent sur la forme des termes, un intervalle de valeur, une valeur, etc.

Par exemple, considérons des contraintes sur les Réels et l'abstraction qui consiste à examiner le signe des variables. Nous avons par exemple les deux Ask abstraits suivants:

$X = 1 \rightarrow Q(X)$ est condamné à suspendre

$X \geq 0 \rightarrow Q(X)$ réussit si positif(X)

L'approximation ne condamne pas le second Ask. Ce type d'abstraction peut donc être suffisante pour certains programmes.

3.3. Une abstraction du Ask plus complexe:

Une autre solution pour abstraire les Ask serait d'envisager les différents cas qui se présentent. Nous pourrions imaginer de traduire par exemple:

$$(X = a) \rightarrow Q(X) \text{ par } ground(X) \rightarrow (X = a \wedge Q(X)) \vee (X \neq a)$$

C'est à dire que nous faisons une fois l'hypothèse que X = a et une fois l'hypothèse que X ≠ a et nous poursuivons l'exécution. De cette façon nous obtenons des informations sur chaque cas. Il faut prendre note d'une façon quelconque que X = a et X ≠ a sont des hypothèses et les ajouter au store courant immédiatement.

Cette solution permet d'obtenir des informations sur des tests comme celui-la qui sont condamné à suspendre à cause de l'approximation des stores. Nous pouvons alors raisonnablement travailler avec une abstraction comme celle de la fermeture (groundness) des termes. Cependant, cette solution s'écarte beaucoup du modèle initial et nécessite un traitement complexe.

Il apparaît donc assez difficile de concilier approximation du store et décision au niveau du Ask abstrait. Une approximation qui ne permette à aucun Ask abstrait de réussir quand il y a lieu, nous conduirait à une analyse qui ignore toutes les communications entre agents, qui sont l'enjeu même des langages CC. Pour une analyse valable, il faut soit utiliser un domaine abstrait qui garde assez d'information pour répondre aux Ask abstraits lorsque il y a lieu, soit utiliser des mécanismes plus complexes pour gérer l'abstraction du Ask. Nous poursuivons sur la première solution en présentant un domaine abstrait plus précis que celui de la fermeture (groundness) des termes.

4. Le domaine des abstractions de niveau k:

Nous avons vu précédemment que des domaines abstraits comme celui traitant de la fermeture des termes ne conviennent pas pour approximer les stores car l'approximation est trop forte.

Nous présentons ici brièvement le domaine des abstractions de niveau k introduit par Sato et Tamaki. Ce type d'abstraction est plus précis et donne une approximation de store qui permet à l'exécution abstraite de poursuive relativement loin.

Intuitivement, prendre l'abstraction de niveau k d'un terme revient couper le terme à la profondeur k. Par exemple:

pour k = 1: f(a, f(a)) est abstrait par $\exists X.\ f(X)$

pour k = 2: f(a, f(a)) est abstrait par $\exists X.\ f(a, X)$

Soit (D,|-), le système de contraintes concret, nous considérons le système de contraintes abstrait (D', |-') et la fonction de concrétisation γ tels que:

(i) D' est l'ensemble obtenu en coupant les termes de D au niveau k.

(ii) |-' est identique à |-.

(iii) γ est la fonction qui associe un terme à l'ensemble des termes clos que on peut obtenir en instanciant les variables qu'il contient de toutes les façons possibles.

Sur ce domaine, l'exécution abstraite ignore l'instanciation des termes de profondeur supérieure à k. Donc, tout Ask abstrait portant sur ces valeurs suspendra. Par contre aucune information n'est perdue sur l'instanciation des termes de profondeur inférieure à k, ce qui permet aux Ask portant sur ces valeurs de réussir lorsque il y a lieu et de poursuivre l'exécution abstraite jusqu'à un certain point. La précision de l'analyse augmente avec k.

5. Analyse de détection de blocage:

L'analyse de détection de blocage (deadlock) dans les programmes CC est sans doute l'une des plus intéressantes et des plus complexes. Elle consiste à détecter qu'un ou plusieurs agents restent suspendus indéfiniment lorsque ils sont exécutés en concurrence avec d'autres. L'exemple typique du blocage est celui d'agents bloqués en attente d'information l'un de l'autre. Plus exactement, l'analyse que nous faisons, nous dit qu'un programme donné ne bloque pas.

5.1. Abstraction:

Comme nous l'avons déjà suggéré, dans le cas de l'analyse de blocage, il faut poser une condition supplémentaire sur la fonction de concrétisation pour une analyse correcte:

Soit le système de contraintes concret (D, |-), le système de contraintes abstrait (D', |-') et γ la fonction de concrétisation, il faut:

$$\forall S' \subseteq D', \forall c' \in D'.$$
$$(S' \vdash' (\neg c') \Rightarrow \forall S \subseteq \gamma(S'), \forall c \in \gamma(c'). S \vdash (\neg c))$$

C'est à dire que non seulement un Ask abstrait ne peut réussir que si tous les Ask concrets qu'il approxime réussissent mais un Ask abstrait ne peut également échouer que si tous les Ask concrets qu'il approxime échouent. En effet, de la réussite ou de l'échec dépend la suite du calcul et de l'analyse. Il faut prendre en compte les agents éventuellement exécutés à la suite d'un Ask. On peut alors dire que lorsque il n'y a pas d'agents qui suspendent indéfiniment dans l'exécution du programme abstrait, alors le calcul abstrait mime exactement le calcul concret. Et nous pouvons conclure que le programme initial est sans blocage.

Il est facile de vérifier que le système de contraintes basé sur le domaine des abstractions de niveau k, défini précédemment, vérifie ces conditions. En effet, au dessous de k, le calcul abstrait correspond exactement au calcul concret, et au delà de k, il peut suspendre. Mais cette suspension systématique au delà de k ne permettra pas toujours de bonnes analyses, notamment dans les cas récursifs.

Dans certains cas, il est possible de réaliser une analyse de détection de blocage plus fine en utilisant également de l'information sur la fermeture (groundness) des termes. Par exemple, considérons l'agent:

$X = a \rightarrow Q(X)$ sur le store {ground(X)}

Nous pouvons dire que lorsque X est clos, quelque soit sa valeur, le Ask sur X = a ne suspendra pas. Dans le cas où Q(X) ne suspend pas lui-même, si il ne contient pas de Ask par exemple, nous pouvons l'ignorer et considérer que le Ask échoue ou réussi mais ne lance pas Q(X). En effet, le fait qu'il réussisse ou échoue nous importe peu en lui -même.

5.2. Enregistrement du blocage:

La sémantique dénotationnelle des langages CC ne nous permet pas de déterminer si tel ou tel agent est suspendu pendant l'exécution. Nous utilisons à cet effet des variables particulières pour enregistrer dynamiquement la suspension des agents.

Pour chaque agent conditionnel P(X), nous ajoutons une variable $X_{P(X)}$ de telle façon que:

$X_{P(X)}$ = suspendible $\Rightarrow$ P(X) suspend, réussi ou échoue.

$X_{P(X)}$ = success-or-fail $\Rightarrow$ P(X) réussi ou échoue.

Nous considérons l'ordre: suspendible $\leq$ success-or-fail.

Soit P(X) :- c $\rightarrow$ A un agent du programme à analyser et $c \in \gamma(c')$. Nous construisons l'agent abstrait P'(X) de la manière suivante:

$$P'(X) :- (X_{P(X)} = \text{suspendible}) \wedge (c' \rightarrow (X_{P(X)} = \text{success-or-fail}) \wedge A)) \wedge (\neg c' \rightarrow (X_{P(X)} = \text{success-or-fail})))$$

ou

$$P'(X) :- (X_{P(X)} = \text{suspendible}) \wedge (c' \rightarrow (X_{P(X)} = X_A) \wedge A)) \wedge (\neg c' \rightarrow (X_{P(X)} = \text{success-or-fail}))) \text{ si } X_A \text{ est défini.}$$

L'analyse du blocage revient alors à examiner la valeur finale de ces variables. Lorsque aucune n'a la valeur suspendible, nous pouvons conclure que le programme est sans blocage.

5.3. Exemple:

Considérons, par exemple, le programme d'exclusion mutuelle présenté dans le chapitre précédent:

$$p(X, Y) :: (X = lock(granted) \rightarrow Y = p) \vee (X = lock(denied) \rightarrow true).$$

$$q(X, Y) :: (X = lock(granted) \rightarrow Y = q) \vee (X = lock(denied) \rightarrow true).$$

$$mutex(X) :: \exists X'. (X = [lock(Reply) \mid X'] \rightarrow Reply = granted, mutex'(X')).$$

$$mutex'(X') :: \exists X'. (X = [lock(Reply) \mid X'] \rightarrow Reply = denied, mutex'(X')) \vee (X = [\,] \rightarrow true).$$

$$faster(X) :: \exists R1\ \exists R2\ \exists L.\ (p(lock(R1), X), q(lock(R2), X), merge([lock(R1)], [lock(R2)], L), mutex(L)).$$

Ce programme présente un schéma de communication assez complexe. Considérons l'agent initial faster(X). Dans un premier temps, les agents p, q et mutex suspendent. L'agent merge instancie alors L avec [lock(R1), lock(R2), [ ]] ou [lock(R2), lock(R1), [ ]] selon le cas. Puis, l'agent mutex instancie R1 et R2 avec 'granted' ou 'denied' selon le cas. Les agents p et q sont alors libérés.

Nous considérons une abstraction de niveau k. Pour que l'exécution abstraite ne se termine pas avec des agents suspendus, nous devons prendre k = 4 de telle façon que R1 et R2 ne soient pas coupés et que les agents abstraits correspondants à p et q soient libérés. Nous pouvons imaginer que, une fois libérés, ces derniers se lancent dans de lourds calculs qui eux seront coupés. Peu importe, nous pouvons conclure que le programme ne bloque pas.

Conclusion

D'abord, nous nous sommes intéressés aux langages logiques concurrents avec contraintes. Le modèle CC définit une classe de langages qui recouvre à la fois les langages logiques proprement dits et les langages logiques avec contraintes. Il ouvre la porte à l'implémentation de langages logiques concurrents très puissants. Cette puissance réside dans les mécanismes de communications inter-processus de haut niveau, qu'il offre. De plus, par la présence de contraintes, il offre une certaine souplesse et un éventail de domaines d'applications très large.

Ensuite, nous avons étudié un modèle d'interprétation abstraite pour ces langages, basé sur leur sémantique dénotationnelle et sur la notion d'abstraction de système de contraintes. Ce modèle apparaît comme une généralisation de ce qui avait été fait pour l'interprétation abstraite des langages logiques avec contraintes.

Malheureusement, l'application de ce modèle a soulevé plus de problèmes que de solutions. Nous n'avons pas de solution générale et immédiate comme dans le cas de l'interprétation abstraite des langages logiques avec contraintes. Le problème majeur réside dans l'abstraction du Ask. En effet, pour une analyse valable, nous devons recourir à un domaine abstrait relativement précis pour éviter que l'information nécessaire au test ne soit perdue dans l'abstraction. Il semble également prometteur de développer un traitement plus complexe de l'abstraction du Ask qui permette d'obtenir de l'information cas par cas.

En particulier, l'application du modèle au cas de l'analyse de détection de blocage n'est pas des plus faciles. L'abstraction est d'autant plus délicate que l'analyse porte sur le comportement des agents et non sur le résultat du programme. De plus, sa mise en oeuvre est très lourde car elle nécessite l'utilisation de variables particulières pour enregistrer la suspension des agents au cours de l'exécution.

Bibliographie

[1] P. Cousot et R. Cousot. Abstract Interpretation and Application to Logic Programs. 1992.

[2] C. Codognet et P. Codognet : Abstract Interpretation for Concurrent Constraint Languages. Draft. 1993.

[3] C. Codognet, P. Codognet et MM Corsini : Abstract Interpretation for Concurrent Logic Langages. In proceedings North American Conference on Logic Programming, Austin, Texas, MIT Press 1990.

[4] P. Codognet et G. Filé: Computations, Abstractions and Constraints in Logic Programs. In proceedings IEEE International Conference on Computer Languages. IEEE Press. 1992

[5] C. Lassez. Constraint Logic Programming. 1987.

[6] B. Le Charlier. L'analyse statique des programmes par l'interprétation abstraite.

[7] B. Le Charlier. Notes du cours de Computational Logic. Institut d'informatique. FUNDP. Année 1992-1993.

[8] B. Le Charlier and P. Van Hentenryck: Reexecution in Abstract Interpretation of Prolog. In proceedings Joint International Conefrence and Symposium on Logic Programming, Washington, MIT Press 1992.

[9] J.W.Lloyd. Foundations of logic Programming. Springer-Verlag.

[10] V.A Saraswat, M. Rinard et P. Panangaden: Semantic Foundations of Concurrent Constraint Programming. In Proceedings of Ninth ACM Symposium on Principles of Programming Languages, Orlando, FL, January 1991.

[11] V.A Saraswat : Concurrent Constraint Programming : A Brief Survey. Augustus 1992.

[12]. V.A Saraswat : The Category of Constraint Systems is Cartesian-Closed. In Proceedings LICS'92, Logic In Computer Science, IEEE Press 1992.

[13] V.A Saraswat et Evan Tick: A Retrospective Look at Concurrent Constraint Programming. June 1993.