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

researchportal.unamur.be

UNIVERSITÉ DE NAMUR

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Étude et Comparaison de deux Modèles d'Interprétation Abstraite

Nélis, Patrick

Award date:
1992

Awarding institution:
Universite de Namur

Link to publication

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

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

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

Download date: 23. Jun. 2020

Facultés
Universitaires
Notre - Dame de la Paix
Namur

Institut d' Informatique

Etude et Comparaison de deux Modèles d' Interprétation Abstraite

par Patrick Nélis

Promoteur :
B. Le Charlier

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

Année académique 1991- 1992

ERRATA

NUMERO DE PAGE - NUMERO DE LIGNE - VERSION CORRIGEE

Abstract, L.19 : Finally, we propose a program that...

Avant-propos, L.4 : ...qui par ses conseils et sa méthode m'**a guidé**
L.10 : ...pour m'avoir **intégré** ...

p.13, L.1 : S1 et S2 sont équivalents par rapport à la substitution σ **vu l'égalité obtenue dans l'exemple page 12.**

p.15, L.15 : Append([X4 | X5], X2, [X4 | X6]) :- append(X5, X2, X6).

p.16, L.10 : **Elle** devient vrai...

p.16, L.20 : ...en ne s'intéressant qu'à certaines propriétés

p.20, L.16 : ⊕ : permet de faire des compositions

p.26, L.7 : Nous donnons ensuite cet algorithme ...

p.27, L.9 : [A,c] est équivalent à [A',c'] noté **A≡ A'** , ssi **c ⊕(A=A')≡ c' ⊕(A=A').**

p.28, L.14 : Les solutions trouvées...

p 32, L.4 : L'arbre et la table sont décrits...

p.43, L.9 : Cette sémantique **manipule**...

p.57, L.17 : Nous trouvons ainsi...
L.30 : ... où sont définis tous les prédicats....

p.59, L.4 : certaines conjonctions

p.69, L.14 : **Dans le modèle 2**, β' porte sur les variables de la tête de p, donc {X}

L.15 : **Dans le modèle 1**, les patterns de sortie portent sur les variables de la clause p, donc {X, Y, Z, W }.

p.71, L.15 : Or une **relation** d'ordre ...

p.72, L.12 : ... permet d'assurer que cette fonction de composition peut **jouer** les trois rôles.

p.78, L.13 : 1. le traitement d'un noeud solution : procédure développe_noeud_**solution.**

L.16 : 3. le traitement d'un noeud spécial : procédure **développe_noeud_spécial.**

Annexes, L.9 : Modification 1 : Il a été nécessaire de modifier les déclarations des procédures car **celles-ci** ...

Résumé

Ce travail concerne l'interprétation abstraite. Nous présentons et comparons deux modèles d'interprétation abstraite.

Le premier modèle est basé sur une sémantique opérationnelle. Nous reprenons ce modèle en détaillant l'algorithme, notamment l' utilisation faite de la table et la manière de construire l'arbre. Enfin, nous proposons un programme de manière à comparer son efficacité par rapport à celui de l'autre modèle.

Le deuxième modèle est basé sur une sémantique de point fixe. Nous reprenons ce modèle en le comparant au premier modèle. La comparaison porte sur les résultats et la méthode de calcul d'une interprétation abstraite. Nous discutons de l' implémentation déjà réalisée pour ce deuxième modèle en présentant quelques tests réalisés sur des programmes Prolog.

Abstract

This work concerns abstract interpretation. We present and compare two models of abstract interpretation.

The model one is based on an operational semantic. We describe the algorithm of the model. The description focuse on the use of the table and the way the tree is built. Finally, we propose a program than can be implemented in order to compare the efficacity of the method use towards the model two.

The model two is based on an fixpoint semantic. We present and compare this model towards the model one. We speak about the result and the way the computation is done for the two models of abstract interpretation. We describe a program made for this model and give some tests realised on Prolog programs.

Avant- propos

La réalisation de ce mémoire est le fruit d'une collaboration. Je tiens à remercier les personnes qui ont contribué à celle-ci.

L'idée de ce mémoire revient à mon promoteur, le professeur B. Le Charlier, qui par ses conseils et sa méthode m'à guider durant toute la rédaction de ce travail. Je le remercie vivement.

Ma recherche dans le domaine de l'interprétation abstraite a été réalisée pour une grande part à l'Université de Padoue. Je remercie G. Filé, mon maître de stage, pour sa grande disponibilité. Ses idées précieuses m'ont beaucoup aider dans ma quête à la compréhension de l'interprétation abstraite. Je remercie les doctorants de cette université pour m'avoir intégrer à leur univers de recherche.

J'associe à mes remerciements les membres de l'équipe de recherche FOLON pour leur amabilité et leur conseil .

Que toutes les personnes qui m'ont aidé de quelque manière que ce soit peuvent trouver dans ces quelques lignes l'expression de ma reconnaissance.

Je remercie également les étudiants, habitués du pool de Macintosh du 2ème, qui ont pu créer une ambiance stimulante et "anti-glucose" indispensable au maintien de notre équilibre psycho-neuro-physio-informatique lors de nos longues journées de travail acharné.

Je remercie mes proches pour tous les témoignages d'encouragement qu'ils m'ont apporté durant toutes ces années académiques.

P.N.

Table des matières

Introduction 1

Première partie : Généralités sur l'interprétation abstraite 2

Introduction 2
Chapitre 1 : L'interprétation abstraite 3
Chapitre 2 : Programmes logiques 5
2.1. Interprétation et programmes logiques 5
2.2. Définition d'un programme logique 6
Chapitre 3 : Le domaine abstrait 8
3.1. Concept 8
3.2. Le domaine abstrait Prop 11
Chapitre 4 : Le point fixe 14
Chapitre 5 : Un exemple d' interprétation abstraite 15

Deuxième partie : Modèles d'interprétation abstraite 17

Introduction 17
Chapitre 6 : Un modèle d'interprétation abstraite selon une sémantique opérationnelle 19

6.1. Le modèle de base 19
6.1.1. Notations. 19
6.1.2. Le système de calcul C 20
6.1.3. Définition d'un programme dans C 21
6.1.4. Définition d'une exécution d'un programme dans C 21

6.1.5. C' : Simulation du système de calcul C 23
6.1.6. Définition d'un programme dans le système de calcul C' 24
6.2. Le modèle avec table 26
6.2.1. Définitions 26
6.2.2. La condition d'équivalence entre une interprétation abstraite avec et sans table 27
6.2.3. Les trois scénarii 28
6.2.4. Un algorithme d'interprétation 38
6.2.5. Un exemple d'interprétation abstraite 39
Chapitre 7 : Un autre modèle d'interprétation abstraite selon une sémantique de point fixe 43

7.1. Le modèle d'interprétation 43
7.1.1. Le principe 43
7.1.2. Sémantique concrète et abstraite 44
7.1.3. Sémantique abstraite de point fixe 44
7.1.4. Définition des opérations 45
7.1.5. Le point fixe 46
7.2 L' implémentation 49
7.2.1. L'algorithme 49
7.2.2. Eléments d'optimisation 52
7.2.3. Description des procédures 53
7.2.5. L' implémentation de Prop 57
7.3. Les corrections 59
7.4. Exemples d' exécutions 59
7.4.1. Queens 59
7.4.2. SORT 65
7.4.2. QUICKSORT 67
Chapitre 8 : Comparaison des deux modèles d'interprétation abstraite. 68

8.1. Les résultats des interprétations abstraites 68
8.2. La méthode de calcul des interprétations abstraites 70
8.3. Similitude et différence entre les opérations dans les deux modèles 71

8.3.1. Les opérations de mise à jour de la table SAT 71
8.3.2. La fonction de projection π 72
8.3.3. La fonction de composition $\oplus$ 72

Troisième partie : Implémentation 73

Introduction 73
Chapitre 9: un algorithme d'interprétation abstraite 74

9.1. Modifications par rapport à l'algorithme développé au point 6.2.4. 74
9.2 Structures des données 75
9.2.1. L' arbre 75
9.2.2. Les noeuds 75
9.2.3. La table 77
9.3 Spécifications de l'algorithme 77
9.3.1. Terminologie 77
9.3.2. Les spécifications des procédures 77
9.4 Algorithme 82

Conclusions 88

Bibliographie 90

Annexes 1

Modification 1
Modification 2
La procédure CHGVAR 2
Modification 3
La procédure EXTEND 5
Modification 4
La procédure compareBeta 7
Modification 5
La Procédure Adjust. 11

Introduction

Ce travail se compose de trois parties. La première partie introduit les notions de base relative à l'interprétation abstraite. Nous tentons de familiariser le lecteur au langage propre de l'interprétation abstraite. Nous définissons l'interprétation abstraite et les applications possibles. La deuxième partie présente deux modèles d'interprétation abstraite. Le premier modèle est présenté au chapitre 6 d'un point de vue théorique. Nous développons l'algorithme d'interprétation sur un exemple concret et sur un exemple plus général. Nous terminons ce chapitre par un exemple d'interprétation abstraite. Le chapitre 7 est consacré à la présentation du deuxième modèle d'interprétation abstraite. Celui-ci a fait l'objet d' une implémentation que nous présentons. Nous donnons ensuite les résultats de l'exécution de cette algorithme sur quelques programmes Prolog. La comparaison entre ces deux modèles est écrite au chapitre 8. Elle compare les résultats donnés par l'interprétation abstraite ainsi que la méthode de calcul. La troisième partie présente un programme et ses spécifications en vue d'une implémentation de l'algorithme d'interprétation abstraite donné au chapitre 6. Enfin , en annexes, nous trouvons les corrections apportées au programme trouvé dans (J-P Nelissen, 1991).

Première partie

Généralités sur l'interprétation abstraite de Prolog

Introduction

Cette partie sera consacrée à définir l' interprétation abstraite et à donner le contexte dans lequel elle s'applique. Nous introduisons ensuite les notions importantes sous-jacentes à l'interprétation abstraite. Nous parlons d'abord du domaine abstrait en tant que concept général. Ensuite, nous nous

intéressons à un domaine particulier, Prop. Nous terminons par la notion de point fixe et la présentation d'un schéma général de l'interprétation abstraite.

Chapitre 1

L'interprétation abstraite

L'interprétation abstraite s' inscrit dans le cadre de l'analyse statique de programmes. Cette analyse nous donne des réponses quant à savoir si l'exécution d'un programme, ou mieux, si l'ensemble de ses exécutions respectent certaines propriétés données. "Les informations ainsi obtenues serviront soit à transformer le programme pour l'optimiser, soit à vérifier qu'il remplit certains critères de correction." (B. Le Charlier, 1991, p.1) "Une manière de connaître de telles informations serait d'exécuter à la compilation le programme pour toutes les données en entrée. Cette approche directe est infaisable en pratique car un programme a un nombre illimité d'exécution." (D.S.Warren, 1992, p. 102)

L'interprétation abstraite ne consiste pas à exécuter le programme en ajoutant par exemple, au sein de son code quelques tests pour rechercher les propriétés de son exécution . Au contraire, cette méthode effectue, à partir d'un domaine abstrait choisi en fonction des propriétés à examiner, un traitement sur le code du programme. Dans le but de réaliser ce traitement, on généralise le programme en question en créant un nouveau programme qui est une approximation de l' original. "Ce nouveau programme préserve la

structure de l'original mais opère sur des objets abstraits. " (D.S.Warren, 1992, p. 102)
"L'idée de base de l'interprétation abstraite est la suivante : l'analyse statique d'un programme consiste à exécuter une analyse sur un domaine spécial appelé domaine abstrait parce qu'il abstrait seulement les propriétés désirées du domaine concret. " (P. Codognet, G. Filé , 1992, p.2)

"L'idée de base de l'interprétation abstraite est de réaliser une approximation de certaines propriétés (souvent indécidables) en utilisant un domaine abstrait à la place du domaine concret de calcul." (B. Le Charlier, P. Van Hentenryck , 1992a, p.2),

L'interprétation abstraite se sert d'un domaine abstrait; nous voyons deux raisons à l'utilisation d'un domaine abstrait. D'une part, il s'agit de prendre en compte uniquement les propriétés à examiner lors de l'analyse. D'autre part , il est inacceptable d'examiner les propriétés sur le domaine concret de calcul. (B. Le Charlier , 1991, p.5) "Les opérations de base (sur le domaine concret) doivent être remplacées par des opérations ,qui sont des approximations consistantes, sur le domaine abstrait." (B. Le Charlier, P. Van Hentenryck , 1992a, p.2) Nous constatons qu'il faut, premièrement, établir une relation entre domaine concret et domaine abstrait; deuxièmement, qu'il faut créer, à partir des opérations opérant sur le domaine concret, des opérations consistantes pour le domaine abstrait.

Nous trouvons dans (B. Le Charlier , 1991, p.5) la relation suivante entre domaine abstrait et concret :

"Soit C un domaine concret, soit A un domaine abstrait,

1.$\forall c \in C : \gamma(\alpha(c)) \supset c$

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

où α : $C \rightarrow A$, fonction monotone d'abstraction

γ : $A \rightarrow C$, fonction monotone de concrétisation

La fonction d'abstraction α associe à chaque ensemble de valeur, sa meilleure approximation. La fonction de concrétisation γ associe à chaque élément de A, l'ensemble des valeurs qu'il représente."
La fonction d'abstraction et de concrétisation nous définissent la relation entre les éléments du domaine concret et ceux du domaine abstrait.

Les interprétations abstraites de programmes s'exécutent au moyen d'un algorithme générique. "Un algorithme , d'interprétation abstraite, générique est un algorithme indépendant du domaine abstrait qui peut faire l' objet d'une spécialisation afin de construire un algorithme traitant les propriétés à évaluer." (B. Le Charlier, K. Musumbu, P. Van Hentenryck , 1991, p.1) .
Une interprétation abstraite est réalisée grâce à un algorithme générique spécialisé en fonction du domaine abstrait choisi.

Chapitre 2

Programmes logiques

2.1. Interprétation et programmes logiques

L'interprétation abstraite de programmes a de multiples applications. Une de celles-ci est d'optimiser le code généré lors de la compilation des programmes logiques. En programmation logique, les programmes sont définis comme des relations entre objets. Ils ne sont pas définis en fonction du calcul à réaliser. Le Charlier, 1991, p.3 indique que les langages logiques présentent la caractéristique de "multidirectionnalité". Cette propriété dit que tout paramètre d'une procédure peut être utilisé indifféremment comme donnée ou comme résultat. " Le code généré pour une procédure totalement multidirectionnelle doit être soit très général soit composé d'un grand nombre de codes distincts correspondants à toutes les utilisations possibles." (Le Charlier, 1991, p.3) L'auteur insiste sur le fait que toutes les directions

possibles ne sont pas utilisées en pratique. Il propose ainsi une application de l'interprétation abstraite qui est de détecter les utilisations faites du programme et d'en déduire les optimisations nécessaires dans le code du programme.

2.2. Définition d'un programme logique

Le vocabulaire de la programmation logique est composé de faits, de régles, de questions, de substitutions, de programmes logiques, de procédures, de clauses, de termes,... Nous reprenons à présent la définition de ces termes.

"Un terme est une constante, une variable ou un terme complexe" (L. Sterling, E. Shapiro ,1986, p.17)

"Un fait est un moyen d'établir un lien entre des objets" (L. Sterling, E. Shapiro ,1986, p.2)

Exemple: père(Jean, Joseph).
Ce fait dit que Jean est le père de Joseph.

"Une question est une conjonction de la forme $A_1,...,A_n$? , $n > 0$, où A_n sont des buts."(L. Sterling, E. Shapiro ,1986, p.17)

"Une substitution est un ensemble de paire de la forme X=t, où X est une variable et t, un terme; aucune paire n'a la même variable comme membre de gauche." (L. Sterling, E. Shapiro ,1986, p.17)

Nous pouvons appliquer une substitution à un terme e. Pour cela, il faut remplacer chaque occurrence X_i dans le terme e par t_i.

"Un programme logique est un ensemble fini de clauses. Une clause ou régle est une phrase logique quantifiée de manière universelle de la forme

$$A \leftarrow B_1,B_2,...,B_k \quad k \geq 0$$

où A et B_i sont des buts." (L. Sterling, E. Shapiro ,1986, p.17)

Il existe deux manières de lire une clause : une manière déclarative et une manière opérationnelle. L. Sterling, E. Shapiro ,1986, p.17 donne la lecture déclarative :" A est impliqué par les conjonctions B_i."; et la lecture opérationnelle : "Pour répondre à la question A, il faut répondre à la conjonction de questions B_1,B_2,...,B_k".

"Une collection de régles ou de clauses avec le même prédicat dans la tête est appelé une procédure." (L. Sterling, E. Shapiro ,1986, p.11)

"Le calcul d'un programme logique consiste à trouver une instance de la question donnée, qui peut se déduire logiquement de P. Un but G peut se déduire de P s'il existe une instance A, de G, où $A \leftarrow B_1,B_2,...,B_n$, $n \leq 0$, est une instance ground d'une clause de P, et les B_i's peuvent se déduire de P".
(L. Sterling, E. Shapiro ,1986, p.17)

L'exécution d'un programme logique est différente de l'exécution de programme non logique. La manière de calculer importe peu. Il s'agit plutôt de trouver des déductions logiques à partir de nos connaissances.

Les programmes logiques sont donc un terrain d'application idéal pour l'interprétation abstraite, étant donné les multiples optimisations qui peuvent être trouvées et appliquées.

Chapitre 3

Le domaine abstrait

Nous présentons d'une part le concept que recouvre le domaine abstrait. Nous utilisons pour ce dessein le domaine abstrait SIGNE. Nous présentons d'autre part le domaine abstrait PROP utilisé dans (J-P Nelissen, 1991) et présenté de manière plus consistante dans (A. Cortesi, G. Filé, W. Winsborough, 1991). Ce domaine abstrait sera utilisé dans les chapitres suivants.

3.1. Concept

Nous l'avons vu, le domaine abstrait permet de cerner les propriétés à évaluer. Nous aurons pour chaque propriété ou ensemble de propriétés un domaine abstrait qui lui correspond.

Voici comment K. Musumbu, 1990, p. I/15 définit un domaine abstrait.

"On définit un domaine abstrait en associant à tout ensemble fini de variables de programme, soit D ,un ensemble inductif, noté $Asub_D$. Les éléments de cet ensemble sont appelés substitutions abstraites. Un ensemble inductif est un ensemble ordonné possédant un plus petit élément et tel que toute suite croissante possède une borne supérieure. Un ensemble ordonné fini,

possédant un plus petit élément, est inductif. On notera par le même symbole $\leq$ les différentes relations d'ordre des différents ensembles inductifs $Asub_D$. De même, on notera $\perp$ leur élément minimal."

Examinons d'abord l'exemple suivant pour mieux comprendre. Il est inspiré de D.S.Warren ,1992, p.102. Soit le domaine concret : l'ensemble des entiers N. Nous examinons la régle des signes. Nous pouvons créer le domaine abstrait qui divise N en trois parties :

neg : les entiers négatifs
nul : les entiers nuls
pos : les entiers positifs ;
entier : abstrait N.

Nous appelons ce domaine abstrait SIGNE: {neg,nul,pos,entier}. Nous pouvons maintenant redéfinir les opérations classiques d' addition, de multiplication et de soustraction sur SIGNE. Nous obtenons les opérations abstraites suivantes :

Tableau 3.1. L'opération de multiplication :

*	neg	nul	pos	entier
neg	pos	nul	neg	entier
nul	nul	nul	nul	nul
pos	neg	nul	pos	entier
entier	entier	nul	entier	entier

Tableau 3.2. L'opération d' addition :

+	neg	nul	pos	entier
neg	neg	neg	entier	entier
nul	neg	nul	pos	entier
pos	entier	pos	pos	entier
entier	entier	entier	entier	entier

Tableau 3.3. L'opération de soustraction :

-	neg	nul	pos	entier
neg	entier	neg	neg	entier
nul	pos	nul	neg	entier
pos	pos	pos	entier	entier
entier	entier	entier	entier	entier

La fonction de concrétisation Cc est :

$Cc(pos) = \{ x \mid x \text{ est un entier et } x > 0 \}$

$Cc(nul) = \{ x \mid x \text{ est un entier et } x = 0 \}$

$Cc(neg) = \{ x \mid x \text{ est un entier et } x < 0 \}$

$Cc(entier) = \{ x \mid x \text{ est un entier} \}$

Programme Bidon
Z := X * Y
W := Z * Z
U := Z-W
V := Z + W

schéma 3.1. : Programme bidon.

Considérons le programme BIDON du schéma 3.1. Si nous donnons une valeur, respectivement à X et à Y, l'exécution du programme nous donnera des valeurs pour Z, W, U, V.

Exemple: X=4, Y= -5

=> Z= -20, W=400, U= -420, V=380.

Nous pouvons extraire quelques informations concernant la liaison entre les entrées (X et Y) et les sorties (Z, W, U et V) en utilisant la règle des signes. Le processus d'interprétation abstraite pour le programme Bidon, en utilisant le

domaine abstrait SIGNE et les opérations sur SIGNE définies par les tableaux 3.1 à 3.3 , est décrit dans le tableau 3.4.

Tableau 3.4 Résultats de l'interprétation :

X	*Y*	Z	W	U	V
neg	*neg*	pos	pos	entier	pos
pos	*neg*	neg	pos	neg	entier
pos	*pos*	pos	pos	entier	pos
neg	*pos*	neg	pos	neg	entier

Nous pouvons tirer de ce tableau l'information suivante: quel que soit la valeur de X et Y, la valeur de W sera toujours positive. Cette information donne au compilateur la possibilité de choisir une meilleure représentation pour la variable W.

3.2. Le domaine abstrait Prop

Prop (pour proposition) est un domaine abstrait qui met en évidence la propriété de groundness des variables d'un programme et d'équivalence entre variables, ou ensembles de variables. "Il est composé de classes d'équivalence de formules propositionnelles dont les variables correspondent aux variables du programme (à analyser) tel qu'une assignation vraie (de la variable) indique que la variable du programme est ground." (A. Cortesi, G. Filé, W. Winsborough, 1991, p.1)

Considérons $V = \{x_i \mid i \in N\}$ un ensemble de variables propositionnelles, PF l'ensemble de toutes les formules propositionnelles construites sur les variables de V avec les connectifs $\vee, \wedge, \leftrightarrow$ et $\neg$

Commençons par définir la propriété ground d'une variable. Il faut replacer le mot ground dans son contexte. En effet, ground est un des résultats possibles lors de l'analyse de mode des variables d'un programme.

"Un terme ground est un terme ne contenant pas de variables. De manière similaire, un atome ground est un atome ne contenant pas de variables".(J. W. Lloyd ,1987, §3 , p.15)

Poursuivons en donnant la définition d'une substitution ground.

"$\sigma = \{v_1/t_1,\dots,v_n/t_n\}$ est une substitution ground si tous les t_i sont des termes ground's." (J. W. Lloyd ,1987, §4 , p.20)

Une variable est ground si elle est liée à un terme ne contenant pas de variables. Pour définir l'assignation vraie d'une variable, nous avons besoin d'une fonction **r** qui se définit comme suit :

" r: V -> {0,1} où 1 signifie assignation vraie et 0 assignation faux."

(A. Cortesi, G. Filé, W. Winsborough, 1991, p.2)

L'équivalence se définit comme suit :

"Deux ensembles de variables S_1 et S_2 sont équivalents respectivement à un ensemble de substitution Σ.

$$\text{Si } \sigma \in \Sigma \Rightarrow \bigcup_{x \in S1} var(\sigma x) = \bigcup_{x \in S2} var(\sigma x)$$

où var(t) est l'ensemble de variables qui apparait dans t".
(A. Cortesi, G. Filé, W. Winsborough, 1991, p.2)

exemple $\sigma = \{ x_1/ y_1, x_2/ y_1, x_3/ y_4\}$
$S_1 = \{x_1, x_3\}$
$S_2 = \{x_2, x_3\}$

$$\bigcup_{x \in S1} var(\sigma S1) = \bigcup_{x \in S2} var(\sigma S2)$$

$$\{y_1, y_4\} = \{y_1, y_4\}$$

S1 et S2 sont équivalents par rapport à la substitution σ vu que var(S1) = var(S2). "Le domaine Prop se consiste de classes d'équivalence de formules propositionnelles basées sur les connectifs $\vee,\wedge,\leftrightarrow$." (A. Cortesi, G. Filé, W. Winsborough, 1991, p.4) Avec Prop, nous pouvons exprimer les propriétés des variables d'un programme sous forme de proposition. Ici la propriété est la groundness.

Prenons un exemple : Append(X,Y,Z). La proposition $(X \wedge Y \wedge Z)$ exprime que X, Y et Z sont ground's. De même, $(X \wedge Y \leftrightarrow Z)$ exprime que X et Y devient ground quand Z devient ground et réciproquement.

Dans (J-P Nelissen, 1991), Prop est utilisé durant tout le processus d'interprétation, premièrement, pour décrire l'état initial et final des variables du programme avant et après l'exécution de l'algorithme générique sur ce programme, deuxièmement, les opérations abstraites manipulent les éléments du domaine abstrait Prop.

Chapitre 4

Le point fixe

La notion de point fixe est une notion importante si l'on veut comprendre comment se termine le processus d'interprétation abstraite d'un programme. Les définitions et propositions suivantes vont nous aider à comprendre la notion de point fixe.

"Un ensemble avec un ordre partiel est un treillis complet si le "lub"(x) et "glb"(x) existe pour tout sous-ensemble x de L."
" Soit L un treillis complet et T: L → L un mapping. On dit que a ∈ L est le plus petit point fixe de T, si a est un point fixe (T(a) = a), et pour tous les points fixes b de T, nous avons a ≤ b. De manière similaire, on définit le plus grand point fixe."
" Soit L un treillis complet et T: L→ L monotone, alors T a un plus petit point fixe "lfp"(T) et un plus grand point fixe "gfp"(T).

De plus $\quad lfp(T) = glb\{ x: T(x)= x \} = glb\{ x: T(x) \leq x \}$
$\quad glp(T) = lub \{x : T(x) = x \} = lub \{ x : x \leq T(x) \}$ "

(J. W. Lloyd, 1987, §5, p.26-30)

La notion de point fixe se porte principalement sur les procédures et fonctions récursives. Les programmes que nous analysons au moyen de l'interprétation

abstraite sont souvent définis de manière récursive. Pour les fonctions non récursives, le point fixe n'a pas de sens puisque le calcul réalisé par ces fonctions est immédiat. Nous pouvons retenir que le point fixe est atteint quand T(a)=a, c'est à dire quand le calcul de la fonction sur la donnée est égal à cette donnée.

Chapitre 5

Un exemple d' interprétation abstraite

Nous présentons tout d'abord le schéma général valable pour toute interprétation abstraite concernant l'analyse de groundness des variables. Nous allons brièvement décrire une interprétation abstraite. Nous choisissons l'analyse de groundness de variables. Il s'agit de déterminer si les variables X1, X2, X3 du programme APPEND , après son exécution, sont ground's.

Le programme append est défini comme suit.

Append ([],X1,X1).
Append (X4| X5],X2,[X4| X6]):- append(X4,X2,X6).
Nous allons dans une première étape normaliser/transformer ce programme.
Nous obtenons :
Append(X1,X2,X3):-X1=[],X2=X3.
Append(X1,X2,X3):-X1=[X4| X5],X3=[X4|X6],append(X5,X2,X6).

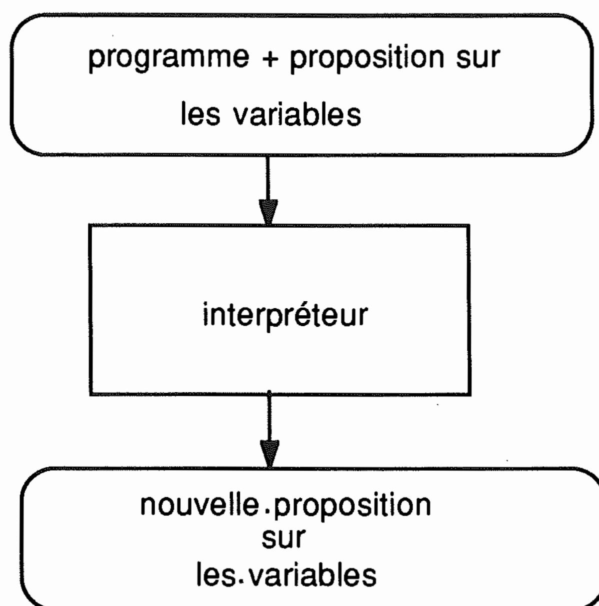


Schéma 5.1: Plan général de l' interprétation abstraite

Nous avons choisi Prop comme domaine abstrait. Nous décidons que nous n'avons aucune information sur les variables. Le tableau 1.6. donne une vue de l'interprétation abstraite. Si nous nous basons sur le modèle d'interprétation abstraite décrit au chapitre 3, nous pouvons brièvement décrire l'interprétation abstraite de la manière suivante.

Pour chaque opération concrète tel que l'assignation, l'appel d'une procédure, nous avons une opération abstraite Ces opérations abstraites manipulent des éléments abstraits. Il s'agit, en fait, de modifier l'assignation de la variable. Il devient vrai si l'opération a pour but de rendre la variable ground .

Pour chaque procédure du programme nous pouvons conserver les informations au sujet du mode des variables. Durant le processus d'interprétation, nous pouvons comparer les nouvelles informations reçues au sujet du mode des variables d'une procédure avec les anciennes. Si celles ci apportent une information nouvelle, nous mettons à jour nos informations. L'interprétation abstraite se termine quand nous obtenons le point fixe dans le processus d'interprétation. Nous ne pouvons plus préciser nos informations.

L'interprétation abstraite simule d'une certaine manière une exécution concrète d'un programme en ne s'intéressant qu' à une certaines propriétés concernant le programme; d'où l'importance du choix du domaine abstrait qui représente ces propriétés.

Deuxième partie

Modèles d'interprétation abstraite

Introduction

Cette deuxième partie sera consacrée à la présentation, d'une part au chapitre 6, d'une méthode d'interprétation abstraite décrite dans (P. Codognet, G. Filé, 1992). Celle-ci sera développée avec l'aide de quelques schémas qui nous aideront à comprendre les principaux mécanismes. Nous pouvons résumer la méthode de la manière suivante.

Nous avons d'une part un système de calcul C et un autre appelé C'. Nous prenons un programme p du système de calcul C pour lequel nous pouvons définir un programme abstrait p' tel que l'exécution de p' réalise l'interprétation abstraite de p. Un interpréteur avec table est proposé pour rendre le calcul fini et moins grand.

Nous présentons, d'autre part au chapitre 7, un autre modèle d'interprétation abstraite décrit dans (B. Le Charlier, K. Musumbu, P. Van Hentenryck, 1990), (B. Le Charlier, K. Musumbu, P. Van Hentenryck, 1991) et (B. Le Charlier, P. Van Hentenryck, 1992a). Ce modèle a fait l'objet d'une implémentation décrite dans (J-P. Nelissen, 1992). Nous reprenons les éléments principaux de la méthode. Nous présentons ensuite l' implémentation du modèle et du domaine Prop au point 7.2. Nous donnons ensuite les améliorations apportées au code de cette implémentation. Quelques exemples d'interprétation abstraite sont proposés pour conclure ce chapitre.

Le chapitre 8 sera consacré à la comparaison des deux méthodes proposées.

Chapitre 6

Un modèle d'interprétation abstraite selon une sémantique opérationnelle

6.1. Le modèle de base

Le modèle d'interprétation abstraite que nous allons présenter est basé sur le principe suivant. Le programme à interpréter est décrit dans un système de calcul C. L'interprétation de p nécessite la construction d'un programme p' décrit dans un système de calcul C'. Son exécution produit l'interprétation abstraite de p. Ce processus peut se répéter plusieurs fois. Il suffit de définir un nouveau système de calcul C" et de construire p". Dans ce dernier cas, il s'agit de réaliser l'interprétation abstraite du programme p' en exécutant p" : nous réalisons ainsi l'interprétation abstraite de l'interpréteur.

6.1.1. Notations.

P= ensemble des prédicats

F= ensemble des fonctions

$\Sigma = F \cup P$

Soit V= Un ensemble fini de variables

P contient le prédicat "=" et la constante faux

$L\Sigma$ = l'ensemble des formules bien-formées sur Σ et V.

6.1.2. Le système de calcul C

Un système de calcul C est défini par un tuple de cinq composants $(\Sigma^*, I, D, \oplus, \pi)$ où

Σ^* est un sous-ensemble fini de Σ.
I est une interprétation des symboles de Σ^*
D est le domaine de calcul
$\oplus$ est une opération associative de D x D dans D, appelée composition
π est une fonction de D x 2^V dans D, appelée projection.

I : "Une interprétation définit une signification pour chaque symbole de la formule." (J. W. Lloyd, 1987, §3, p.10) Ici, il s'agit de donner une interprétation pour tous les éléments de $L\Sigma$. Celle-ci est utile quant il s'agit de connaître la valeur (vraie ou fausse) de la formule.

D : Il s'agit du domaine abstrait. (cfr le point 1.2) On peut mettre en correspondance D avec un sous-ensemble de $L\Sigma*$.

$\oplus$: permet de faire des composition avec les éléments du domaine. Dans Prop, prenons la proposition $[A \wedge B]$ qui signifie que A et B sont ground's et d' autre part la proposition suivante $[(A \leftrightarrow X) \wedge (A \leftrightarrow Y)]$, nous pouvons faire une composition de ces deux propositions et nous obtenons

$$[A \wedge B \wedge (A \leftrightarrow X) \wedge (A \leftrightarrow Y)]$$
$$= [A \wedge B \wedge X \wedge Y],$$

ce qui signifie que X et Y sont devenus ground's. Des deux représentations du résultat, celle de droite du signe de l'égalité est une simplification de celle de gauche.

π : Cette fonction permet d'exprimer les éléments du domaine en fonction de certaines variables. Si nous projetons la formule propositionnelle $[A \wedge B \wedge X \wedge Y]$ sur (X,Y), nous obtenons $\pi([A \wedge B \wedge X \wedge Y], (X,Y)) = [X \wedge Y]$. Remarquons que si nous prenons un exemple plus complexe, $\pi([X \leftrightarrow (Y \wedge Z)], \{X\})$, nous perdons de l'information.

Nous pouvons résumer la description d'un système de calcul C de la manière suivante. C nous donne des informations sur les objectifs de l'exécution d'un programme. Nous connaissons les fonctions et les prédicats utilisés ainsi que leurs interprétations. Nous pouvons composer les éléments du domaine grâce à l'opérateur $\oplus$. Chaque système de calcul est défini à partir des mêmes éléments mais les éléments du tuple $(\Sigma^*, I, D, \oplus, \pi)$ sont différents d'un système à l'autre.

6.1.3. Définition d'un programme dans C

Nous pouvons maintenant définir un programme dans un système de calcul C. Un programme dans le système de calcul $C = (\Sigma^*, I, D, \oplus, \pi)$ où un programme dans C est un ensemble de clauses de la forme

$$p(\overline{X}) \leftarrow c, q_1(\overline{Y_1}), \ldots, q_n(\overline{Y_n})$$

où $\overline{X}$, $\overline{Y_1}$, $\overline{Y_n}$ sont des vecteurs de variables différentes prises dans V,

$c \in D$

et $\{p, q_1, \ldots, q_n\} \in K$ où K est un ensemble de prédicats disjoint de Σ.

$p(\overline{X})$ est la tête de la clause et $c, q_1(\overline{Y_1}), \ldots, q_n(\overline{Y_n})$ le corps de la clause.

Les éléments du domaine D sont regroupés au début du corps de la clause. Quand n=0, la clause est appelée une unit clause étant donné que son corps est vide. Un but est une clause où la tête de la clause est vide.

Un but sera de la forme $\leftarrow c, A_1, \ldots A_k$ où A_i sont des atomes.

Un but de la forme $\leftarrow c$ est un but vide. Il est aussi appelé but terminal.

6.1.4. Définition d'une exécution d'un programme dans C

L'exécution du programme sera comme en programmation logique la résolution d'une suite de buts. Nous allons maintenant définir une exécution d'un programme p dans C avec un but b.

Une exécution de p dans le système de calcul C avec b, une suite de but $<b_0,b_1,...,b_n,...>$, se définit comme suit :

Si $b_i = \leftarrow c',A_0(\overline{X_0}),A_1(\overline{X_1}),...,A_k(\overline{X_k})$

alors

[1.] Il existe une clause $A_0(\overline{Y_0}) \leftarrow c_1,G_1(\overline{Y_1}),...,G_n(\overline{Y_n})$ qui est une clause de p.

[2.] Soit m, l'arité de A_0, et c' se calcule de la manière suivante:

$$c'=c \oplus c_1 \oplus (\overline{X_0}(1) = \overline{Y_0}(1)),\oplus...\oplus(\overline{X_0}(1) = \overline{Y_0}(m)) \text{ et } c'<>faux.$$

Nous notons c=faux lorsque les contraintes c, c_1 et les contraintes issues de l' unification, pour le passage de paramètres, de$(\overline{X_0}(1) = \overline{Y_0}(1)),\oplus...\oplus(\overline{X_0}(1) = \overline{Y_0}(m))$ sont incompatibles. Dans le cas contraire, c<>faux, il est possible de créer une nouvelle contrainte c à partir des autres contraintes car elles sont compatibles.

$\overline{X_0}(1), \overline{Y_0}(1)$ sont respectivement les premières composantes des vecteurs $\overline{X_0}$ et $\overline{Y_0}$.

exemple :
c= $[X \wedge Y]$, c_1= $[S \wedge T]$, $\oplus = \wedge$ et
$(X_0(1)=Y_0(1)),\oplus...\oplus(X_0(m)=Y_0(m))$ équivalent à $[(X \leftrightarrow T) \wedge (Y \leftrightarrow R) \wedge (Z \leftrightarrow S)]$.
c'= $[X \wedge Y \wedge Z \wedge R \wedge S \wedge T]$.

[3.] $b_{i+1} = \leftarrow c', G_1(\overline{Y_1}),..., G_n(\overline{Y_n}), A_1(\overline{X_1}),...,A_k(\overline{X_k})$

A l'étape 1, on cherche une clause de p qui permet de résoudre le but. A l'étape 2, nous calculons c', un élément du domaine et à l'étape 3, nous définissons un nouveau but à résoudre b_{i+1} dans lequel nous avons développé A_0. Chaque transition de b_i à b_{i+1} est un pas de l'exécution. Une exécution réussit si le dernier but est de la forme $b_n = \leftarrow c'$, et échoue dans le cas contraire.

Nous pouvons définir la sémantique du programme p à partir de tous les buts engendrés par le but initial. Nous définissons ANS(p,b)= { c ∈ D | ←c est le but terminal pour une résolution réussie d'un but }.

Nous présentons au schéma 6.1., sous forme d'arbre, la suite des buts engendrés à partir de $\leftarrow c_0$, append(A,B,C) le but initial avec $c_0 = \{(A \wedge B)\}$. Nous ne donnons pas les explications complètes concernant le contenu des noeuds de cet arbre. Celles-ci seront données au point 6.2.

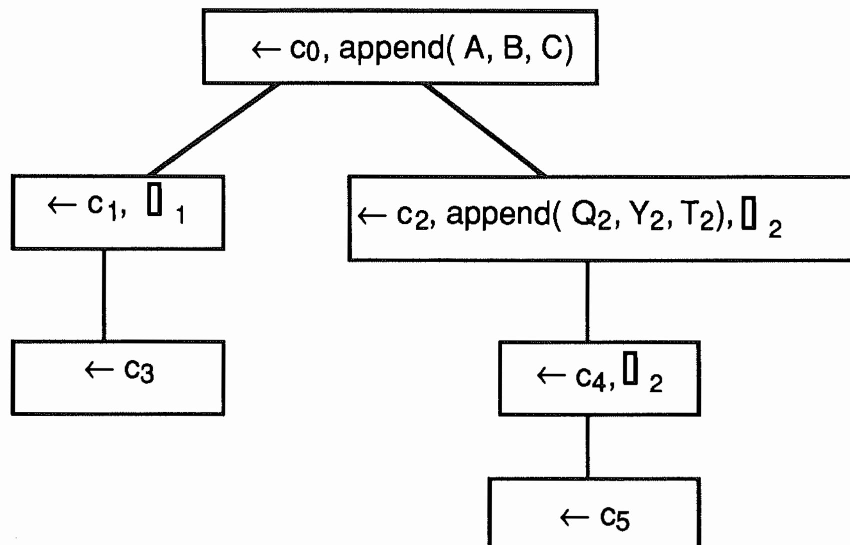


Schéma 6.1.: l'arbre de dérivation des buts à partir de $\leftarrow c_0$, append(A,B,C)
ANS(append(A,B,C),c_0) = $\{c_3,c_5\}$.

6.1.5. C' : Simulation du système de calcul C

Le système de calcul C' se définit de la même manière que C. Le tuple de cinq éléments reste le même mais ses éléments peuvent être différents.
L'objectif dans C' est que l'exécution de tout programme p' transformé selon les règles décrites ci-dessous à partir d'un programme p de C produit une interprétation de p.
Nous pouvons considérer que le système de calcul C correspond au système concret que nous trouvons dans d'autres méthodes d'interprétation. Ensuite, le système de calcul C', dans le même ordre d'idée, est considéré comme le système abstrait. De plus, dans cette méthode d' interprétation, nous pouvons définir plusieurs niveaux d'abstraction. Le principe pour passer d'un niveau donné d'abstraction à un niveau plus élevé est le même quel que soit le niveau.
Considérons deux systèmes d' exécution $C = (\Sigma, I, D, \oplus, \pi)$ et $C' = (\Sigma', I', D', \oplus', \pi')$.

C' simule C par rapport à une fonction γ de D' vers 2^D ssi

1. γ est monotone
2. $\forall$ c $\in$ D, ¡ c'$\in$ D': c$\in$ γ(c')
3. $\forall$ c'_1, $c'_2 \in$ D', $\forall$ c_1, $c_2 \in$ D:
$c_1 \in \gamma(c'_1)$, $c_2 \in \gamma(c'_2)$ implique $c_1 \oplus c_2 \in \gamma(c'_1 \oplus c'_2)$

Le domaine D joue un rôle important puisque les règles de passage d'un système de calcul à l'autre font intervenir le domaine. Il existe aussi une fonction d'abstraction α qui forme avec γ une insertion de Galois de D' vers 2^D.

α et γ satisfont les deux conditions suivantes

(a) $\forall$ c'$\in$ D', $\alpha(\gamma(c')) = c'$

(b) $\forall$ c $\in$ D, $\gamma(\alpha(c)) \supset c$

6.1.6. Définition d'un programme dans le système de calcul C'

Nous allons maintenant décrire comment construire un programme dans C' à partir d'un programme décrit dans C.

Si C' simule C, alors pour tout programme p de C avec un but b, on peut construire un programme p' de C' abstrait correspondant avec un but b' tel que l'exécution de p' avec b' dans le système de calcul C' simule l'exécution de p avec b dans le système de calcul C.

Nous avons C et C' deux systèmes de calcul tel que C' simule C avec une fonction de concrétisation γ et un programme p décrit dans C.

Un programme p' décrit dans C' correspondant à p est un programme formé avec une clause

$T(Y_0) \leftarrow c', C_1(Y_1), \ldots, C_n(Y_n)$

pour chaque clause $T(Y_0) \leftarrow c, C_1(Y_1), \ldots, C_n(Y_n)$ de p

tel que $c \in \gamma(c')$

Un but b' décrit dans C' se définit de la même manière à partir de b décrit dans C.

Nous remarquons que pour construire c', nous prenons un c' tel que $c \in \gamma(c')$. L'élément c' n'est pas nécessairement la meilleure abstraction possible. Pour l'obtenir, il faut prendre $c' = \alpha(c)$.

exemple :
$c = \{X/[\,], Y/Z\}$.
$c'=\{(Y \leftrightarrow Z)\}$ tel que $c \in \gamma(c')$.
La meilleure abstraction est $c'=\{X \wedge (Y \leftrightarrow Z)\}$.

Nous donnons à présent un exemple de formation d'un but et d'un programme b' et p' à partir de b et p.
Prenons le but suivant ← append([d,e,f],[g,h,j],L). Il s'agit de produire une liste L qui est la concaténation de [d,e,f] et [g,h,j].

Le programme APPEND est défini comme suit :

append([ ], X, X).
append([H|X], Y, [H|Z]) ← append(X, Y, Z).

Nous définissons les programmes p et p' comme suit en prenant Prop comme domaine abstrait.

p = append(X, Y, Z)←{X/[ ], Y/Z}.
append(X,Y,Z)←{X/[H|Q],Z/[H|T]}, append(Q,Y,T).

p' = append(X', Y', Z')←{X'∧(Y'↔ Z')}.
append(X',Y',Z')
←{(X'↔(H∧Q))∧(Y'↔Y)∧(Z'↔(H∧T))},append(Q,Y,T).

Nous définissons les buts b et b'comme suit :

b = ←{A/[d,e,f],B/[g,h,j]},append(A,B,L).

b'= ←{A∧B},append(A,B,L)

On a bien $\{X/[\,], Y/Z\} \in \gamma([X \wedge (Y \leftrightarrow Z)])$.
$\{X/[H|Q], Z/[H|T]\} \in \gamma([(X \leftrightarrow (H \wedge Q)) \wedge (Z \leftrightarrow (H \wedge T))])$
et $\{A/[d,e,f], B/[g,h,j]\} \in \gamma([A \wedge B)]$

6.2. Le modèle avec table

Nous avons présenté jusqu' à présent le coeur de la méthode d'interprétation abstraite décrit au point 6.1. Nous allons maintenant présenter une manière d'implémenter la méthode d'interprétation. Nous donnons quelques définitions et notations utiles. Ensuite, nous présentons la condition d'équivalence entre une interprétation avec et sans table. Nous détaillons l'algorithme à l'aide de trois scénarii. nous donnons ensuite cet algorithme et un exemple d'interprétation abstraite termine ce chapitre.

6.2.1. Définitions

Un but généralisé est un but de la forme $\leftarrow c, A_1, \ldots, A_k$

où - A_i est soit un atome,
soit un marqueur de sortie

- c est une valeur abstraite qui porte sur les variables de $A_1, \ldots, A_k$.

L'ensemble VARCL(A) est un ensemble de variables de la clause ou de du but b qui contient l'atome A. Pour le but $\leftarrow c, A_1, \ldots, A_k$, VARCL(A_1) = l'ensemble des variables de $A_1, \ldots, A_k$.

Un littéral d'appel est un couple [A,c]

où - A est un atome,

- c une valeur abstraite qui porte sur les variables de A.

On note lg(B), le littéral d'appel le plus à gauche d'un but généralisé b

Soit b= $\leftarrow c, A_1, \ldots, A_k$; on a lg(B)=[$A_1$,c]

Un marqueur de sortie est un littéral spécial noté [A=H,c]•

où - A est un prédicat d'appel,
- H est la tête de la clause de A utilisée avec les variables renommées,
- c est une valeur abstraite qui porte sur varcl(A).

Nous avons un ensemble de procédures définies chacune par un ensemble de clauses. Une clause contient une tête et un corps. H représente la tête.

Un marqueur de sortie rappelle la place, dans le résolvant, où la clause utilisée avec un littéral A a été introduite. Quand il devient le littéral le plus à gauche d'un but généralisé, cela signifie que A a été complètement résolu et que la solution trouvée peut être placée dans la table.

Equivalence de littéraux d'appel :

soit deux littéraux d'appel [A,c] et [A',c'] de C tel que A est $p(X_1,\dots,X_n)$ et A' est $p(Y_1,\dots,Y_n)$;

A=A' signifie que $(X_1=Y_1,\dots, X_n=Y_n)$.

[A,c] est équivalent à [A',c'] noté $A \neq A'$, ssi $c \oplus(A=A') \neq c' \oplus(A=A')$.

Exemple: soit les littéraux d'appel [append(X,Y,Z), $X \wedge Y$] et [append(R,S,T), $R \wedge S$].

A= append(X,Y,Z), $c=X \wedge Y$, A'=append(R,S,T), $c'=R \wedge S$. Montrons qu'ils sont équivalents. Le domaine choisi est Prop.

$$c \oplus(A=A') = [(X \wedge Y) \wedge (X \leftrightarrow R) \wedge (Y \leftrightarrow S) \wedge (Z \leftrightarrow T)] = [(X \wedge Y \wedge R \wedge S)]$$

$$c' \oplus(A=A') = [(R \wedge S) \wedge (X \leftrightarrow R) \wedge (Y \leftrightarrow S) \wedge (Z \leftrightarrow T)] = [(X \wedge Y \wedge R \wedge S)]$$

6.2.2. La condition d'équivalence entre une interprétation abstraite avec et sans table

P. Codognet, G. Filé, 1992, p.10 donnent la condition, qu'un système de calcul doit satisfaire, pour que l'interprétation avec table soit équivalente à celle sans table.

Cette condition porte en fait sur la fonction de projection π. En effet, dans le modèle sans table, il n'existe pas de projection. Cette condition permet d'assurer l'équivalence entre les deux modèles.

" Condition(X) : nous considérons la suite de buts $< G_0, G_1, \dots, G_k >$

où

- $G_0 = \leftarrow c_0, A_1, A_2, \dots, A_n$,
- $G_1 = \leftarrow c_1, B_1, \dots, B_m , A_2, \dots, A_n$,
- la clause utilisée pour le pas $G_0 \to G_1$ est

$$cs = H \leftarrow c', B_1, \dots, B_m,$$

- $G_k = \leftarrow c_k, A_2, \dots, A_n$.

c'_k est la contrainte calculée localement dans la clause cs et porte sur les variables de cette clause. Les deux point suivants doivent être vérifiés:

1. $\pi(c_1, Var(cs)) = \pi(c' \oplus c_0 \oplus (A_1=H), Var(cs))$

2. $\pi(c_k, Var(G_0)) = \pi(c_0 \oplus c'_k \oplus (H= A_1), Var(G_0))$

Le point 1 assure que nous pouvons oublier une partie des contraintes obtenues jusqu'à présent lorsqu' on rencontre un nouvel atome. Il suffit en fait de garder les contraintes sur les variables de la tête de la clause.

Le point 2 indique lorsque le calcul de la clause sera terminé, il suffira de combiner les contraintes calculées localement , en ne gardant que celles de la tête de la clause, avec les contraintes connues avant ce calcul.

6.2.3. Les trois scénarii

Le modèle d'interprétation abstraite avec table utilise deux éléments "nouveaux". Il s'agit d'une table et d'un arbre. La table contient la liste des littéraux d'appel rencontrés ainsi que les solution trouvées pour ces derniers. Les solutions pour un même littéral sont différentes.

littéraux d'appel	liste des solutions
[A,c]	$c_1, c_2, c_3, c_4, \ldots$
....	.....

Schéma 6.2. : La table des littéraux d'appel et de leurs solutions.

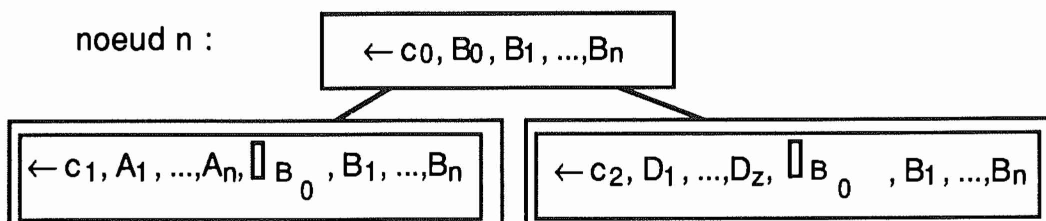
Les littéraux d'appel jouent le rôle de clé pour cette table; en effet, tous les littéraux d'appel sont différents. L'équivalence des littéraux d'appel est défini au point 6.2.1. Les solutions sont des valeurs abstraites qui portent sur les variables de varcl(A).
L' arbre est composé d'un ensemble de sommets étiquetés par un but généralisé. Le principe de de ce modèle est le suivant. Nous collectons dans une table tous les littéraux d'appel rencontrés lors de la résolution du but initial. Les littéraux d'appel seront en fait les littéraux d'appel les plus à gauche dans un but. Nous avons alors trois scénarii possibles. Le premier

consiste à traiter le cas où le littéral d'appel n'est pas présent dans la table. Le deuxième consiste au traitement du cas où le littéral d'appel est présent dans la table. Le troisième concerne le traitement du marqueur de sortie.

Dans les schémas qui vont illustrer les principes du modèle avec table, l'arbre sera représenté par un ensemble de rectangles correspondant aux sommets reliés entre eux par des arêtes. Nous présenterons chaque fois la situation initiale et la situation résultante de l'exécution de l'action déduite du principe décrit. Les sommets doublement encadrés représenteront les nouveaux sommets. Dans la table, le format gras sera utilisé pour mettre en évidence les éléments nouveaux. Nous utiliserons la procédure append comme exemple. Celle-ci est définie au point 6.1.6. Nous nous plaçons également d'un point de vue général.

Scénario 1: le traitement du littéral d'appel non présent dans la table.

Supposons que notre but initial est $\leftarrow c_0$, append(A,B,C) avec $c_0 = \{(A \wedge B)\}$ Dans le cas général, nous avons le but initial suivant $\leftarrow c_0, B_0, B_1, \ldots B_n$. La table ne contient aucun élément. Remarquons que les variables de c_0 portent sur les buts $B_0, B_1, \ldots B_n$. Ce but initial est le sommet de notre arbre. Nous considérons le littéral d'appel le plus à gauche [append(A,B,C), c_0] pour notre exemple, ou [B_0, c_0] pour notre cas général; c_0 est restreint aux variables de B_0. La table ne contient pas de littéral d'appel équivalent puisqu'elle est vide. En conséquence, le noeud de l'arbre qui contient ce but est appelé un noeud solution. Nous insérons dans la table le littéral d'appel [append(A,B,C), c_0] dans le cas général ou [B_0, c_0] dans notre exemple. Nous recherchons les clauses qui définissent la procédure append ou B_0. Pour chaque clause, nous devons créer un nouveau noeud dans l'arbre; ce seront les noeuds fils du noeud solution.



$\square_{B_0} = [B_0 = H, c_0]$

Schéma 6.3. : Cas général : l'arbre dans le scénario 1.

Calculons maintenant le corps du noeud fils sur le schéma 6.3., en utilisant la clause cs= $H \leftarrow c_q, A_1, A_2 \ldots, A_n$. Les autres sommets se calculent de manière similaire. Le noeud fils se compose d'une première partie où nous retrouvons les éléments du domaine. Cette valeur se calcule selon la formule suivante: $c_1 = \pi(c_q \oplus c_0 \oplus (B_0 = H), var(cs))$ .La deuxième partie se compose du corps de la clause utilisée en se limitant à prendre les atomes, suivie d'un marqueur de sortie pour B_0 et du reste du but initial $B_1, B_2 \ldots, B_n$

littéraux d'appel	liste des solutions
[B_0,c_0]	

Schéma 6.4. : Cas général: la table.dans le scénario 1.

Dans notre exemple, la procédure append est définie par deux clauses. Nous devons créer deux noeuds fils pour le noeud initial en utilisant chacune des deux clauses renommées.

La première clause renommée est append(X'_1, Y'_1, Z'_1)$\leftarrow$ $\{(X'_1 \wedge (Y'_1 \leftrightarrow Z'_1))\}$.

Calcul de c_1 :

$c_1 = \pi(\{(X'_1 \wedge (Y'_1 \leftrightarrow Z'_1)) \wedge (A \wedge B) \wedge ((A \leftrightarrow X'_1) \wedge (B \leftrightarrow Y'_1) \wedge (C \leftrightarrow Z'_1))\}$, var(clause 1)) où var (clause 1) est l'ensemble des variables de la clause 1 renommée.

$= \pi(\{(X'_1 \wedge Y'_1 \wedge Z'_1) \wedge (A \wedge B \wedge C)\}, \{X'_1, Y'_1, Z'_1\})$

$= [X'_1 \wedge Y'_1 \wedge Z'_1]$

Le marqueur de sortie est

$\square_1$ = [append(A,B,C) = append (X'_1,Y'_1,Z'_1), c_0]

Le corps du noeud fils utilisant la première clause de append est composé de c_1, suivi du marqueur de sortie.

La deuxième clause renommée est

append(X'_2,Y'_2,Z'_2)

$\leftarrow \{(X'_2 \leftrightarrow (H_2 \wedge Q_2)) \wedge (Y'_2 \leftrightarrow Y_2) \wedge (Z'_2 \leftrightarrow (H_2 \wedge T_2))\}, append(Q_2, Y_2, T_2).$

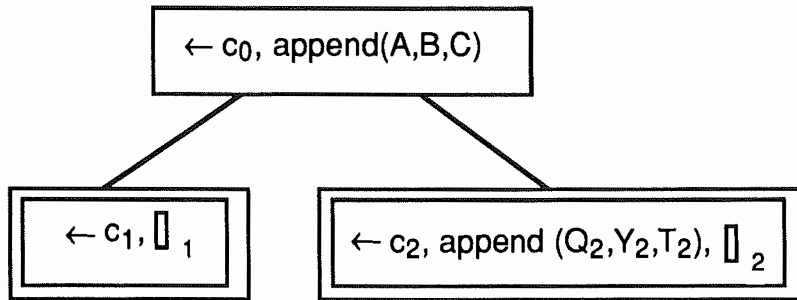


Schéma 6.5. : Notre exemple: l'arbre dans le scénario 1.

Calcul de c_2 :

$c_2 = \pi(\{(X'_2 \leftrightarrow (H_2 \wedge Q_2)) \wedge (Y'_2 \leftrightarrow Y_2) \wedge (Z'_2 \leftrightarrow (H_2 \wedge T_2) \wedge (A \wedge B) \wedge ((A \leftrightarrow X'_2) \wedge (B \leftrightarrow Y'_2) \wedge (C \leftrightarrow Z'_2))\}$, var(clause2)) où var(clause2) est l'ensemble des variables de la clause 2 renommée.

$= \pi(\{X'_2 \wedge Y'_2 \wedge Y_2 \wedge H_2 \wedge Q_2 \wedge Z'_2 \leftrightarrow (H_2 \wedge T_2) \wedge A \wedge B \wedge C \leftrightarrow Z'_2\}, \{X'_2, Y'_2, Z'_2, Y_2, H_2, Q_2, T_2\})$

$= [X'_2 \wedge Y'_2 \wedge Y_2 \wedge H_2 \wedge Q_2 \wedge Z'_2 \leftrightarrow (H_2 \wedge T_2)]$

Le marqueur de sortie est

$[]_2$ = [append(A,B,C) = append (X'_2, Y'_2, Z'_2), c_0]

littéraux d'appel	liste des solutions
[append(A,B,C), (A∧B)]	

Schéma 6.6. : Notre exemple: la table.dans le scénario 1.

Le corps du noeud fils utilisant la deuxième clause de append est composé de c_2 suivi du but append(Q_2,Y_2,T_2) trouvé dans le corps de la clause et du marqueur de sortie. Nous insérons dans la table le littéral d'appel [append(A,B,C),(A∧B)]

Résumé du scénario 1 : le littéral d'appel ne se trouve pas dans la table.Nous ajoutons à la table ce littéral d'appel ainsi qu'une liste vide dans la liste de ses solutions. Nous ajoutons dans l'arbre autant de noeuds fils qu'il existe de clauses qui définissent le prédicat du littéral.

Scénario 2 : le traitement du littéral d'appel connu.

Prenons maintenant la situation où nous sommes en train d'examiner le noeud n. Le noeud n possède son littéral d'appel le plus à gauche équivalent à un littéral d'appel présent dans la table. L'arbre et la table sont décrit dans les schémas 6.7. et 6.8.

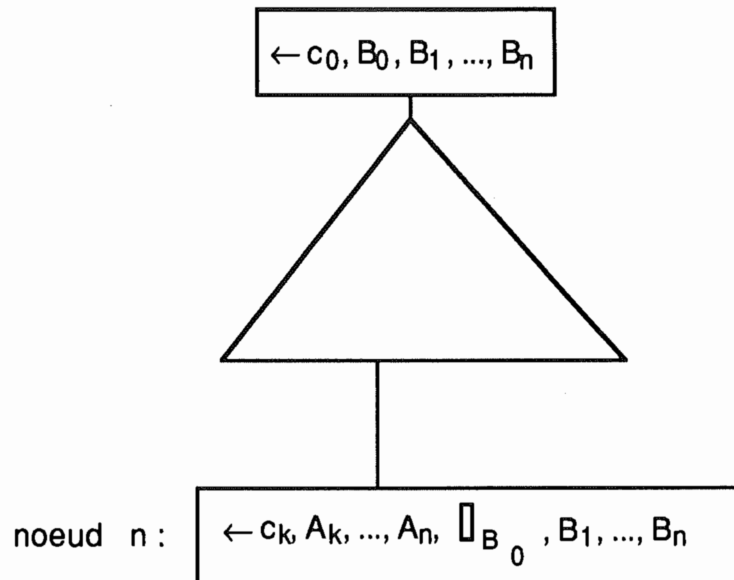


Schéma 6.7 : Cas général: l'arbre dans le scénario 2.

littéraux d'appel	liste des solutions
...	...
$[B_0,c_0]$	c_a, c_b, c_c, c_d
...	...

Schéma 6.8. : Cas général: la table dans le scénario 2.

Dans le cas général, nous nous intéressons au littéral d'appel le plus à gauche du noeud n. Il s'agit de $[A_k,c_k]$. Nous avons dans la table un littéral d'appel $[B_0,c_0]$ équivalent à $[A_k,c_k]$. Nous appelons ce noeud, un noeud de référence, parce que nous utilisons, pour ce littéral, les solutions qui se trouvent dans la table afin de créer les noeuds fils.

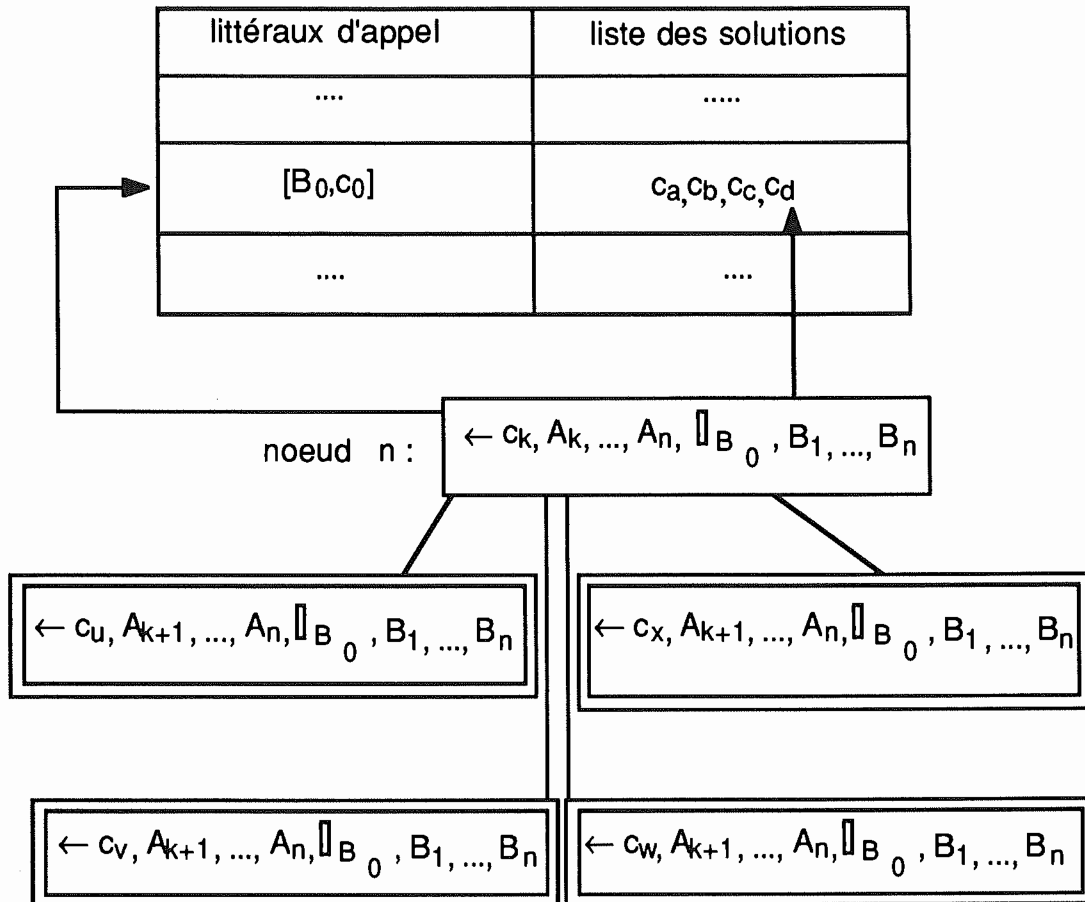


Schéma 6.9.: Quatre sommets fils sont créés.
Ils correspondent chacun respectivement à une des quatres solutions trouvées dans la table. Deux références partent de l'arbre vers la table; une pour pointer vers l'entrée utilisée et une vers la dernière solution utilisée.

En effet, il faut créer autant de noeuds fils qu'il existe de solutions dans la table pour ce littéral. Ensuite, une référence vers la table doit être donnée au noeud de référence, ainsi qu'un pointeur vers la dernière solution utilisée. Nous avons besoin de cette référence et de ce pointeur car ils indiquent si un noeud de référence a bien utilisé toutes les solutions de la table. De nouvelles solutions peuvent être ajoutées à la table dans la suite de l'interprétation. Dans ce cas, il faudra ajouter au noeud de référence un nouveau fils utilisant cette valeur.
Nous allons maintenant calculer les sommets fils. La première partie se calcule selon la formule suivante: $c_u = (c_a \oplus (A_k=B_0))$. La deuxième partie est la copie du reste du but du noeud n, en retirant l'atome A_k.

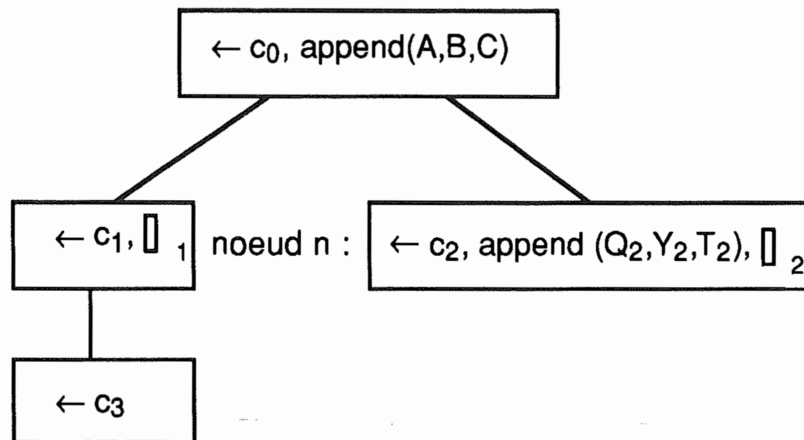


Schéma 6.10. : Notre exemple : l'arbre de départ dans le scénario 2

Dans notre exemple, la situation initiale est représentée par les schémas 6.10. et 6.11. Le noeud-n est le noeud de référence puisque le littéral d'appel de ce noeud [append(Q_2,Y_2,T_2), ($Q_2 \wedge Y_2$)] est équivalent au littéral [append(A,B,C), ($A \wedge B$)] présent dans la table.

littéraux d'appel	liste des solutions
[append(A,B,C), ($A \wedge B$)]	**($A \wedge B \wedge C$)**

Schéma 6.11. : Notre exemple : la table de départ dans le scénario 2.

Nous devons créer autant de noeuds fils pour le noeud n qu'il existe de solutions dans la table pour le littéral [append(A,B,C), ($A \wedge B$)] équivalent. Dans notre cas, il se trouve seulement une solution [$A \wedge B \wedge C$]. Nous créons donc un fils en utilisant cette solution.

Calcul de c_4 :

$$c_4 = (A \wedge B \wedge C) \wedge (Q_2 \leftrightarrow A) \wedge (Y_2 \leftrightarrow B) \wedge (T_2 \leftrightarrow C)$$
$$= [A \wedge B \wedge C \wedge Q_2 \wedge Y_2 \wedge T_2]$$

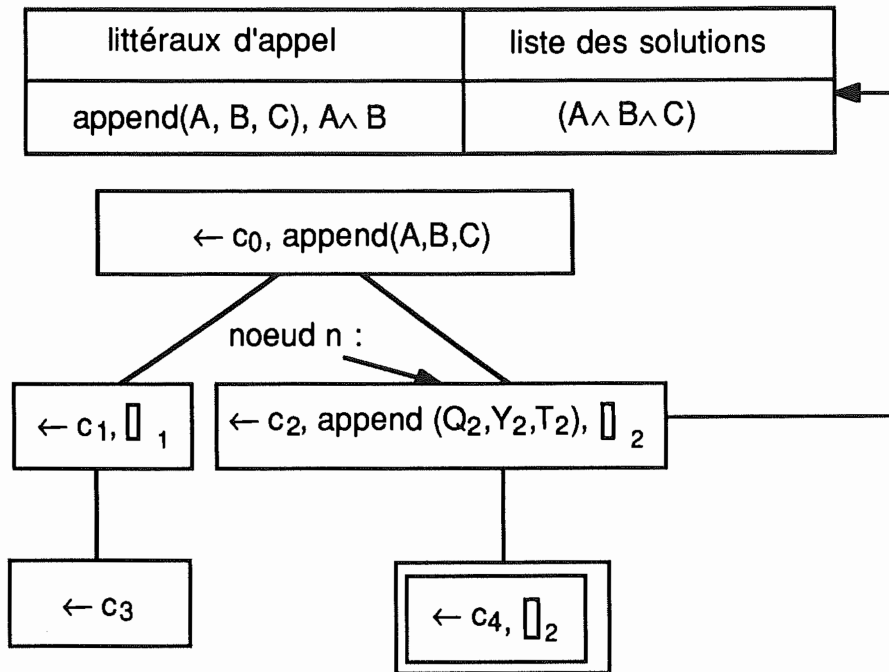


Schéma 6.12.:

Notre exemple : L'arbre et la table après la création du noeud fils.

Le corps du noeud fils est composé de c_4 et du reste du but qui est le marqueur de sortie du noeud n. Une référence est donnée au noeud n vers l'entrée utilisée dans la table et un pointeur indique la dernière solution utilisée.

<u>Résumé du scénario 2 :</u> Nous traitons un littéral connu. En effet, il en existe un équivalent dans la table. Nous utilisons les solutions trouvées jusqu'à présent pour ce littéral. Nous créons autant de noeuds fils qu' il existe de solutions. Nous donnons une référence au noeud n de l'entrée utilisée dans la table et de la dernière solution utilisée. Cette référence sera utile quand une nouvelle solution sera ajoutée à la table. La table n'est pas modifiée.

Scénario 3: Le traitement du littéral de sortie.

Nous allons examiner maintenant le cas où le littéral d'appel le plus à gauche est un marqueur de sortie.

Dans le schéma 6.13., le littéral le plus à gauche du noeud n, est un marqueur de sortie ; celui de B_0. Cela signifie que B_0 est complètement

résolu et qu'on peut placer la solution trouvée dans la table. Le marqueur de sortie contient le littéral d'appel pour lequel nous avons trouvé une solution avec deux jeux de variables ; celles de l'appel et celles de la clause utilisée. Le marqueur de sortie contient aussi une substitution sur les variables du but qui comprend le littéral d'appel. Il suffit de retrouver le littéral d'appel $[B_0,c_0]$ dans la table et d'ajouter à sa liste de solutions, la solution trouvée c_l en la projetant sur les variables de B_0 selon la formule $c_l = \pi(c_k \oplus (B_0=H), varcl(B_0))$. Il faut aussi créer un nouveau noeud qui sera, soit un noeud avec un but terminal ou vide si $B_1...B_n$ est vide, soit un noeud avec un but généralisé si $B_1...B_n$ n'est pas vide.

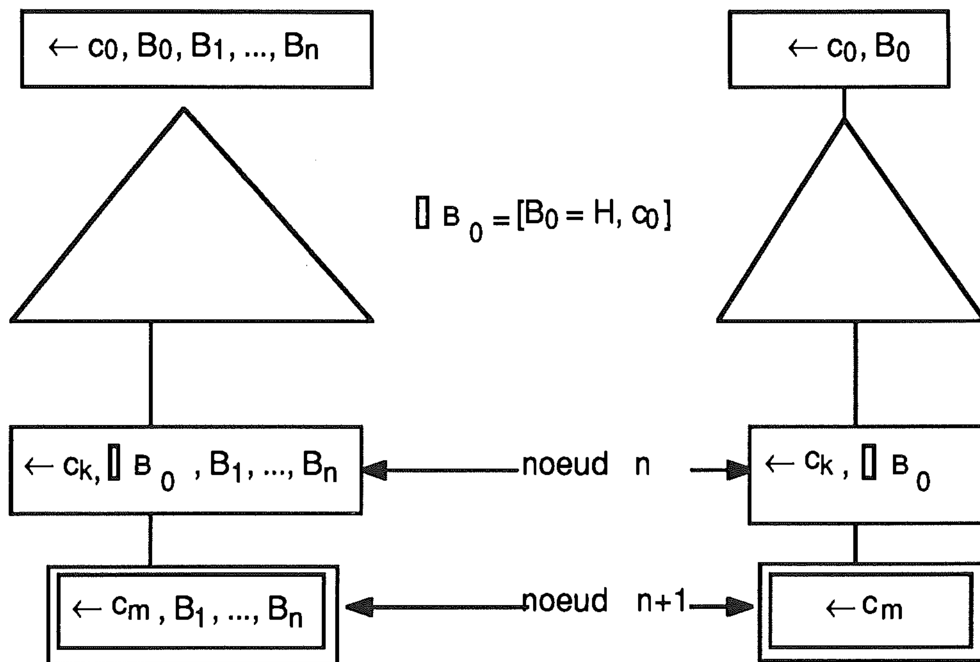


schéma 6.13. : Cas général : l'arbre dans le scénario 3 :
Deux cas sont possibles selon que le corps du but est composé de plusieurs atomes ou d'un seul. Dans les deux cas, on crée un fils pour le noeud comprenant le littéral de sortie.

littéraux d'appel	liste des solutions
...	...
$[B_0,c_0]$	$c_a,c_b,c_c,c_d,\mathbf{c_l}$
...	...

schéma 6.14.: Cas général: la solution $\mathbf{c_l}$ est reportée dans la table.

Nous allons maintenant calculer les sommets fils. La première partie se calcule selon la formule suivante: $c_m = \pi(c_k \oplus c_0 \oplus (B_0{=}H),\ \mathrm{varcl}(B_0))$.La deuxième partie est la copie du reste du but du noeud n en retirant le littéral de sortie.

Reprenons notre exemple. Nous présentons au schéma 6.15. l'arbre avec un noeud n tel que le littéral de gauche est un marqueur de sortie. Nous rappelons que $c_1 = [X'_1 \wedge Y'_1 \wedge Z'_1]$, $c_0 = [A \wedge B]$ et que le marqueur de sortie est [append(A,B,C) =append(X'_1,Y'_1,Z'_1), c_0)].

Calcul de c_3:

$$c_3 = \pi([c_1 \wedge c_0 \wedge (A \leftrightarrow X'_1) \wedge (B \leftrightarrow Y'_1) \wedge (C \leftrightarrow Z'_1)], \{A,B,C\})$$
$$= [A \wedge B \wedge C]$$

Le corps du noeud fils est composé de c_3 . Le reste du but est vide puisqu'il n'existe aucun but, suivant le marqueur de sortie pour le noeud n. On ajoute à la table le résultat $\pi([c_1 \wedge (A \leftrightarrow X'_1) \wedge (B \leftrightarrow Y'_1) \wedge (C \leftrightarrow Z'_1)], \{A,B,C\}) = (A \wedge B \wedge C)$ trouvé pour le littéral d'appel [append(A,B,C),c_0].On accède à la bonne entrée dans la table grâce au marqueur de sortie qui comprend le littéral d'appel se trouvant dans la table.

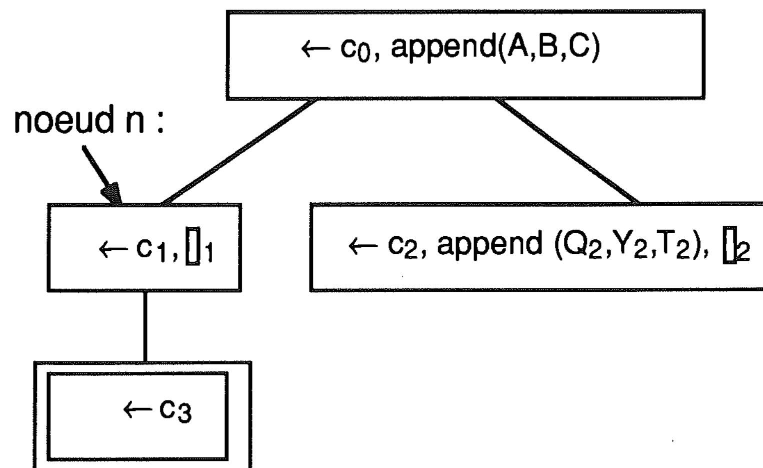


Schéma 6.15. : Notre exemple : L'arbre dans le scénario 3.

<u>Résumé du scénario 3:</u> Le littéral que nous traitons est un marqueur de sortie. Il indique que le but contenu dans le marqueur de sortie est résolu. Nous devons placer la solution trouvée dans la table. Il faut aussi créer un noeud fils pour le noeud qui contient ce marqueur de sortie.

littéraux d'appel	liste des solutions
[append(A,B,C), (A∧B)]	**(A∧B∧C)**

Schéma 6.16. : Notre exemple : La table dans le scénario 3.

6.2.4. Un algorithme d'interprétation

Nous présentons maintenant l'algorithme d'interprétation avec table :

Soit un programme p décrit dans C, un but b décrit dans C, l'interpréteur produit une suite peut-être infinie de couples arbre-table < (t_0,T_0), (t_1,T_1) ,....., (t_n,T_n),...> appelée une interprétation avec table

où - t_0 est la racine étiquetée par b et T_0 est vide,

- (t_{i+1},T_{i+1}) se dérive de (t_i,T_i) en appliquant le pas d'expansion suivant:

choisir une feuille n de t_i; elle est étiquetée $\leftarrow c,A_1,\ldots,A_k$.

Deux cas se proposent:

1. A_1 est un atome.

a. T_i ne possède pas d'entrée avec la clé équivalente à $[A_1,c]$ avec c restreint aux variables de A_1.

Dans ce cas, le noeud-n est un noeud solution, ti+1 est dérivé de ti comme suit:

Pour chaque clause (renommée) cs' = $H \leftarrow c_1,B_1,\ldots,B_m$ de P tel que $c \oplus c_1 \oplus (A_1{=}H)$ est différent de faux , ajouter un fils n' à n dans t_{i+1}, étiquetée $\leftarrow c'',B_1,\ldots,B_m,[A_1{=}H,c]^{\bullet},A_2,\ldots,A_k$ où $c'' = \pi(c \oplus c_1 \oplus (A_1{=}H), var(cs'))$.

T_{i+1} est dérivé de T_i en ajoutant une nouvelle entrée avec la clé $[A_1,c]$ et une liste vide de solutions.

b. T_i possède une entrée e avec la clé $[A',c']$ équivalente à $[A_1,c]$; donc le noeud n est un noeud de référence. Dans t_{i+1}, une référence vers la dernière solution utilisée de la liste des solutions à l'entrée e est donnée à n.

Pour chaque solution c'' de cette liste, il faut ajouter un fils n' à n étiqueté comme suit :
$\leftarrow c'' \oplus (A_1=A'),A_2,....,A_k$

2. A1 est un marqueur de sortie [A=H,c'].

Pour obtenir t_{i+1}, ajouter un fils n' à n , étiqueté comme suit :
$\leftarrow c'', A_2,...,A_k$ où $c''= \pi(c \oplus c' \oplus (A=H),varcl(A))$.

T_{i+1} est obtenu en ajoutant dans la liste à l'entrée e contenant la clé [A=H,c'], la solution c'', pour autant qu'elle ne soit pas présente. Il faut aussi ajouter, pour chaque noeud qui pointe vers l'entrée e, un nouveau noeud fils qui utilise cette nouvelle solution.

Nous avons présenté un méthode d'interprétation abstraite. Nous voyons que le modèle général dans ce cas , a fait l'objet d'une optimisation dans l'utilisation d'une table. Ce mécanisme permet de réutiliser les résultats obtenus précédemment et évite ainsi de recommencer le même traitement plusieurs fois. En effet, les solutions recueillies dans la table sont réutilisées lorsque nous rencontrons des noeuds de référence. Dans ce cas, nous utilisons les solutions de la table pour continuer le traitement, au lieu de recommencer un traitement déjà réalisé auparavant, lorsque nous considérons ces noeuds comme des noeuds solutions.

6.2.5. Un exemple d'interprétation abstraite

Nous donnons maintenant un exemple d'interprétation abstraite pour le but $B_0 = \leftarrow\{A \wedge B\}$, append(A,B,C) Le formalisme utilisé est inspiré de (A. Cortesi, 1992, p. 76 à79).
Les états successifs de l'arbre et de la table seront représentés respectivement par A_0, A_1, ...A_n.et T_0, T_1, ...T_n. Les noeuds de l'arbre seront étiquetés par B_0, B_1, ...B_m.

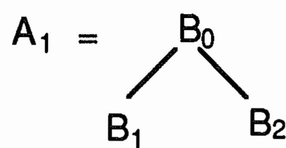
Situation de départ :

$A_0= B_0$ $\quad\quad$ $B_0 = \leftarrow c_0$, append(A,B,C).
$T_0= \emptyset$. $\quad\quad$ $c_0=\{A \wedge B\}$.

Premier pas : Le noeud de A_0 est un noeud solution. Nous appliquons le scénario 1.

La première clause renommée est append(X'_1, Y'_1, Z'_1)$\leftarrow\{(X'_1 \wedge (Y'_1 \leftrightarrow Z'_1))\}$.

La deuxième clause renommée est
append(X'_2,Y'_2,Z'_2)
$\leftarrow\{(X'_2 \leftrightarrow (H_2 \wedge Q_2)) \wedge (Y'_2 \leftrightarrow Y_2) \wedge (Z'_2 \leftrightarrow (H_2 \wedge T_2))\}$,append($Q_2$,$Y_2$,$T_2$).



$B_0 = \leftarrow c_0$, append(A,B,C)
$B_1 = \leftarrow c_1, \square_1$
$B_2 = \leftarrow c_2$,append(Q_2,Y_2,T_2), $\square_2$

c_1 , $\square_1$, c_2, $\square_2$ sont définis comme suit :

$c_1 = \pi(\{(X'_1 \wedge (Y'_1 \leftrightarrow Z'_1)) \wedge (A \wedge B) \wedge ((A \leftrightarrow X'_1) \wedge (B \leftrightarrow Y'_1) \wedge (C \leftrightarrow Z'_1))\}, var(c_1))$
$= \pi(\{(X'_1 \wedge Y'_1 \wedge Z'_1) \wedge (A \wedge B \wedge C)\}, \{X'_1, Y'_1, Z'_1\})$
$= [X'_1 \wedge Y'_1 \wedge Z'_1]$

$\square_1$ = [append(A,B,C)= append(X'_1, Y'_1, Z'_1), c_0]

$c_2 = \pi(\{(X'_2 \leftrightarrow (H_2 \wedge Q_2)) \wedge (Y'_2 \leftrightarrow Y_2) \wedge (Z'_2 \leftrightarrow (H_2 \wedge T_2) \wedge (A \wedge B) \wedge ((A \leftrightarrow X'_2) \wedge (B \leftrightarrow Y'_2) \wedge (C \leftrightarrow Z'_2))\}, var(c_2))$

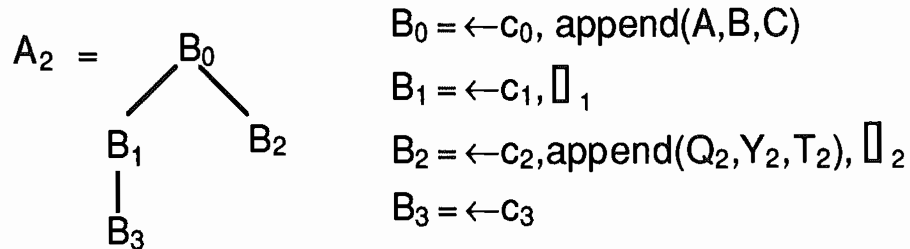
$= \pi(\{(X'_2 \wedge Y'_2 \wedge Y_2 \wedge H_2 \wedge Q_2 \wedge Z'_2 \leftrightarrow (H_2 \wedge T_2)) \wedge (A \wedge B \wedge (C \leftrightarrow Z'_2))\}, \{X'_2, Y'_2, Z'_2, Y_2, H_2, Q_2, T_2\})$

$= [X'_2 \wedge Y'_2 \wedge Y_2 \wedge H_2 \wedge Q_2 \wedge Z'_2 \leftrightarrow (H_2 \wedge T_2)]$

$\square_2$ = [append(A,B,C)= append(X'_2, Y'_2, Z'_2), c_0]

T_1	
littéraux d'appel	liste des solutions
[append(A,B,C), $(A \wedge B)$]	

Deuxième pas : Nous traitons le noeud B_1. Il s'agit d'un noeud où le littéral de gauche est un marqueur de sortie. Nous appliquons le scénario 3.



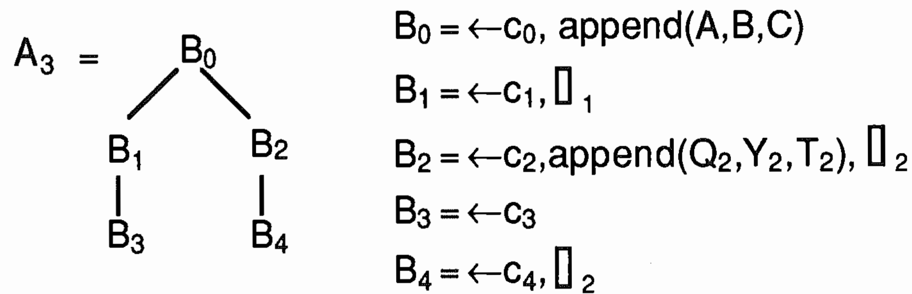
Calcul de c_3:

$c_3 \quad = \pi([c_1 \wedge c_0 \wedge (A \leftrightarrow X'_1) \wedge (B \leftrightarrow Y'_1) \wedge (C \leftrightarrow Z'_1)], \{A,B,C\})$

$\quad = [A \wedge B \wedge C]$

T2	
littéraux d'appel	liste des solutions
[append(A,B,C), (A∧B)]	(A∧B∧C)

Troisième pas : Nous traitons le noeud B_2 qui est un noeud de référence. Nous appliquons le scénario 2.



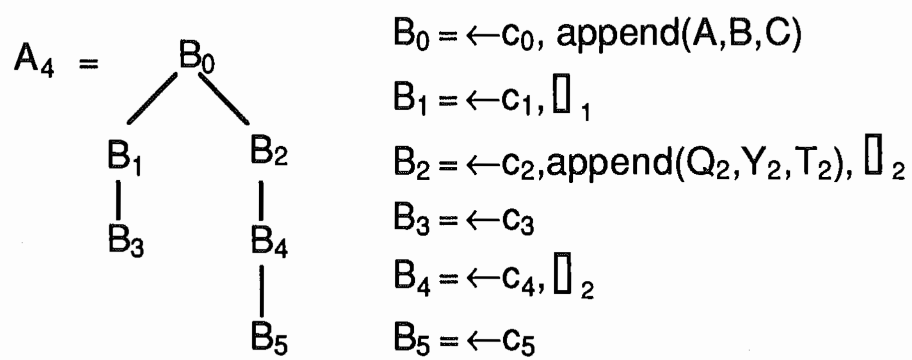
Calcul de c_4 :

$c_4 \quad = (A \wedge B \wedge C) \wedge (Q_2 \leftrightarrow A) \wedge (Y_2 \leftrightarrow B) \wedge (T_2 \leftrightarrow C)$

$\quad = [A \wedge B \wedge C \wedge Q_2 \wedge Y_2 \wedge T_2]$

$T_3 = T_2$

Quatrième et dernier pas : Nous traitons le noeud B_4. Il s'agit d'un noeud où le littéral de gauche est un marqueur de sortie. Nous appliquons le scénario 3.



Calcul de c_5 :

$c_5 \quad = \pi([c_4 \wedge c_0 \wedge (X'_2 \leftrightarrow A) \wedge (Y'_2 \leftrightarrow B) \wedge (Z'_2 \leftrightarrow C)], \{A,B,C\})$

$\quad = [A \wedge B \wedge C]$

T_2	
littéraux d'appel	liste des solutions
[append(A,B,C), ($A \wedge B$)]	($A \wedge B \wedge C$)

Chapitre 7

Un autre modèle d'interprétation abstraite selon une sémantique de point fixe

7.1. Le modèle d'interprétation

7.1.1. Le principe

Le coeur de cette méthode est défini par la sémantique de point fixe donnée au point 7.1.3. Cette sémantique manipulent des substitutions, des représentations de prédicat et un ensemble appelé sat qui contient des tuples composés d'une substitution ß, d'un prédicat p et d'une autre substitution ß'.

Le principe de cette méthode peut se résumer de cette manière. Nous donnons d'une part un prédicat p et une substitution ß sur les variables de p à l' interpréteur. Celui-ci dispose également des définitions des procédures nécessaires à la définition de p. Un ensemble fini de définitions de

procédures est ainsi créé. L'interprétation abstraite de (ß, p) selon la sémantique donnée calcule un ensemble de substitution α' pour tous les tuples (α, q) tel que α est une une substitution sur les variables du prédicat q et q est défini dans l'ensemble des définitions de procédures de p. En particulier l'interprétation abstraite calcule une nouvelle substitution ß' sur les variables de p pour le tuple (ß, p). L'ensemble sat contient les tuples (α, q, α') construit lors de l'interprétation.

7.1.2. Sémantique concrète et abstraite

La réalisation d'une interprétation part dans cette méthode d'une sémantique concrète. Pour manipuler les objets - des ensembles de substitutions concrètes - , nous créons des opérations tel que UNION, AIVAR, AIFUNC, etc pour lesquels nous donnons une définition plus loin. Une interprétation manipulant des substitutions concrètes n' est pas réalisable vu la complexité du calcul. Nous nous tournons alors vers l'interprétation abstraite. A ce dessein, nous définissons une sémantique abstraite en établissant une correspondance entre les éléments concrets et abstraits grâce à une fonction de concrétisation et d'abstraction. Les opérations décrites pour manipuler les substitutions concrètes sont transposées pour manipuler les substitutions abstraites. Le principe dans cette méthode est de rester indépendant de quelque domaine que ce soit. Dans cette méthode l'interprétation abstraite peut être exécutée au moyen d'un algorithme générique qui respecte la sémantique de point fixe donnée et qui utilise les opérations abstraites.

7.1.3. Sémantique abstraite de point fixe

Nous présentons la sémantique abstraite de point fixe donnée par (B. Le Charlier, K. Musumbu, P. Van Hentenryck, 1990, p 14). Cette sémantique abstraite travaille avec des tuples abstraits. Un tuple abstrait est un élément de la forme ($ß_{in}$, p, $ß_{out}$) où $ß_{in}$ et $ß_{out}$ sont deux substitutions abstraites dont la portée est l'ensemble D. D=$\{x_1,...,x_n\}$ est l'ensemble des variables de p; n est l'arité de p, le nom d'une procédure.

"Sémantique abstraite de point fixe :

$TSAT(sat) = \{ (ß, p, ß') : (ß, p) \in U D$ et $ß' = T_p(ß, p, sat) \}$.

$T_p(ß, p, sat) = UNION(ß_1,....,ß_n)$

où $ß_i = T_c(ß, c_i, sat)$
$c_i,....,c_n$ sont les clauses de p.

$T_c(ß, c, sat) = RESTRC(c, ß')$

où $ß' = T_b(EXTC(c,ß),b, sat)$,
b est le corps de c.

$T_b(ß,<>, sat) = ß$.

$T_b(ß,g.gs, sat) = T_b(ß, gs, sat)$

où $ß_3 = EXTG(g,ß,ß_2)$,

$ß_2 =$ $sat(ß_1, p)$ si g est p(...)

$AI_VAR(ß_1)$ si g est $x_i = x_j$

$AI_FUNC(ß_1, f)$ si g est $x_i = f(...)$

$ß_1 = RESTRG(g, ß)$."

7.1.4. Définition des opérations

1. $UNION(ß_1,....,ß_n)$

Pour chaque clause, nous obtenons les substitution $ß_1,....,ß_n$ comme résultat de $T_c(ß, c_i, sat)$ pour les clauses $c_1,....,c_n$ de p. Cette opération prend le "lub" de tous les $ß_i$ et produit un nouvel ß.

2. RESTRC (c, ß')
Le but de cette opération est de restreindre la substitution ß' qui porte sur l'ensemble des variables de la clause c aux variables de la tête de c.

3. EXTC(c, ß)

Nous avons une substitution ß exprimée sur les variables de la tête de la clause. Cette opération va prendre en compte les variables du corps de la clause c et exprimer la substitution sur toutes les variables de la clause c.

4. EXTG(g,ß,$ß_2$)

Nous avons une substitution ß exprimée sur les variables du but g.gs, une substitution $ß_2$ sur les variable de g renommée . Cette opération étend $ß_2$ à ß en prenant en compte le résultat sur les variables correspondantes.

5. RESTRG(g,ß)

Il s'agit de restreindre ß aux variables de g en renommant ces variables .

6. AI_VAR($ß_1$)

Le but de cette opération est d'unifier deux variables en prenant le mgu. Nous obtenons ainsi une substitution qui porte sur ces deux variables.

7. AI_FUNC($ß_1$, f)

L'unification porte ici sur une variable avec un ensemble de variable. Nous prenons aussi le mgu et nous obtenons une substitution sur l'ensemble des variables.

7.1.5. Le point fixe

TSAT : Il s'agit de calculer le meilleur ß' pour tous les tuples (ß,p) $\in$ sat en utilisant T_p jusqu'à l'obtention du point fixe.

T_p(ß,p,sat) : calcule un ß' en fonction de (ß,p) et du sat existant. Cette fonction fait appel à la fonction T_c pour calculer les $ß_i$. Elle exécute ensuite l'opération UNION avec les $ß_i$.trouvés.

T_c(ß,c,sat) : calcule un ß' en fonction de ß , c et de sat. On étend, par la fonction EXTC, ß aux variables du corps b de la clause. On calcule une ß' en appliquant T_b. On restreint ensuite ß' aux variables de la tête de c.

T_b(ß,<>,sat)= ß

T_b(ß,g.gs,sat)

On restreint ß ,par la fonction RESTRG, aux variables de g et on obtient $ß_1$. On traite le but g. Trois cas sont possibles.

1. g est du type p(...). Dans ce cas, nous prenons le ß' existant dans le sat. C'est la meilleure approximation trouvée jusqu'à présent.

2. g est du type $x_i = x_j$. Il s' agit d'exécuter l'opération d'unification AI_VAR.

3. g est du type x_i = f(...) Il s'agit d' exécuter l'opération d'unification AI_FUNC.

Dans les trois cas, on produit une nouvelle substitution $ß_2$. Enfin, on étend par l'opération EXTG $ß_2$ à ß. On applique de nouveau la fonction T_b pour le reste du but gs.

L'interprétation consiste donc à exécuter l'algorithme décrit au point 7.1.3. jusqu'à l'obtention du point fixe. Le point fixe porte sur la fonction TSAT. Il faut s'arrêter quand TSAT(sat) =sat. L'exécution de l'algorithme s' arrête quand il n'est plus possible de trouver un α' plus précis pour un des tuples (α, q) de sat en supposant que l'ensemble sat contienne tous les tuples (α, q) nécessaire à l'interprétation abstraite de p avec ß. B. Le Charlier, K. Musumbu, P. Van Hentenryck, 1990, p.15 et 16 présentent le point fixe de TSAT de la manière suivante. "Le point fixe de TSAT peut être calculé par un ensemble de sat (sat_0, sat_1, ..., sat_n) tel que

$sat_0 = \perp$;

$sat_{i+1} = TSAT(sat_i)$ ($i \geq 0$);

$sat_n = TSAT(sat_n)$.

Le but de l'algorithme est de fournir à chaque pas dans son exécution un sat plus précis. Nous pouvons ainsi ordonner les sat : $sat_0 \leq sat_1 \ldots sat_{n-1} \leq sat_n$. Nous avons $sat_0 = \{(\beta, p, \perp)\}$ et

sat_{i+1} est obtenu à partir de sat_i en sélectionnant un nouveau tuple (α, q, α') tel que

(α, q) n'est pas basé dans sat_i

ou

$T_p(\alpha, q, sat_i) \leq sat_i(\alpha, q)$ n'est vrai.

sat_n est tel que pour tout $(\alpha, q) \in sat_n$, $T_p(\alpha, q, sat_n) \leq sat_n(\alpha, q)$ et (α, q) est basé dans sat_n."

"(β,p) est basé dans sat ssi (β,c) est basé dans sat pour toutes les clauses c qui définissent p."

"(β,c) est basé dans sat ssi $(EXTC(c,\beta),sg)$ est basé dans sat ou sg est le corps de c."

" (β,sg) est basé dans sat ssi

1. sg est une séquence vide.
2. sg est de la forme g.sg' et
   - (a) $var(sg) \supset dom(\beta)$;
   - (b) (β_3,sg) est basé dans sat;
   - (c) $(\beta_1,p) \in dom(sat)$ si g est de la forme p(...);

où $\beta_3 = EXTG(g,\beta,\beta_2)$,

$\beta_2 =$ $sat(\beta_1, p)$ si g est p(...)

$AI_VAR(\beta_1)$ si g est $x_i = x_j$

$AI_FUNC(\beta_1, f)$ si g est $x_i = f(...)$

$\beta_1 = RESTRG(g, \beta)$".

(B. Le Charlier, K. Musumbu, P. Van Hentenryck, 1990, p.15)

7.2 L' implémentation

7.2.1. L'algorithme

Nous trouvons plusieurs versions de l'algorithme d'interprétation. Chacune d'elles correspond à une optimisation. Nous donnons la version 3 de l'algorithme trouvé dans (B. Le Charlier, K. Musumbu, P. Van Hentenryck, 1990, p. 64 et suivantes. Il s'agit des procédures solve, solve_goal, solve_procedure, solve_allclauses, solve_clause.

Algorithme d'interprétation abstraite :

1. La procédure SOLVE

```
procedure solve(   in ßin , p; out ßout )
begin
        same-sat := true;  use := ø;
        sat := ø;  def := ø ;
        solve_goal (ßin, p, ø, use, same_sat, sat, def);
        ßout := sat (ßin, p)
end
```

2. La procédure SOLVE_GOAL

```
procedure solve_goal(    in ßin, p, suspended,
                         inout use,same_sat, sat, def);

begin
        if (ßin, p) ∈ suspended then
                use := use ∪ {(ßin, p)}
        else if (ßin, p) ∉ def then
```

```
        begin
                if (ßin, p) ∉ dom(sat) then
                begin
                        same-sat := false;
                        sat := extend ( ßin, p, sat )
                end;
                solve_procedure ( ßin, p, suspended, useaux,
                                                        same_sat, sat, def );
                if useaux = ø then
                        def := def ∪ {(ßin,p)}
                else
                        use := use ∪ useaux
        end
end
```

3. La procédure SOLVE_PROCEDURE

```
procedure solve_procedure (     in ßin, p, suspended, out use;
                                inout same_sat, sat , def )

begin
        repeat
                solve_allclauses ( ßin, p, suspended ∪ {(ßin, p)} , use,
                                        same_sat, sat , def )
                same_sat := same_sat ∧ same_sataux
        until same_sataux ;
        use := use \ {(ßin,p)}
end
```

4. La procédure SOLVE_ALLCLAUSES

```
procedure solve_allclauses (    in ßin, p, suspended ;
                                out use, same_sat; inout sat, def )

begin
        ßout := ⊥;
        use := ø;
        same_sat := true ;
```

```
        for i :=1 to m with c1, ..., cm clauses-of p do
        begin
                solve_clause ( ßin, ci , suspended, ßaux, same_sat, sat, def )
                ßout := union ( ßout , ßaux)
        end;
        if ¬ ( ßout ≤ sat(ßin, p)) then
        begin
                sat := adjust(ßin, p, ßout, sat);
                same_sat := false
        end
end
```

5. La procédure SOLVE_CLAUSE (in β_{in}, c , suspended, out β_{aux}, inout same_sat, sat, def)

```
begin
        ßext :=EXTC(c,ßin );
        for i :=1 to with b1,...,bm do
        begin
                ßaux :=RESTRB(bi,ßext );
                switch (bi) of

                case xj =xj :
                        ßaux :=AI_VAR(ßaux)

                case xj = f(...) :
                        ßaux :=AI_FUNC(ßaux, f)


                case p(...) :

                        solve_goal(ßaux, p, suspended, use, same_sat, sat, def);
                        ßaux := sat(ßaux , p)

                end;

                ßext := EXTB(bi, ßext, ßaux)
        end;
```

β_{out} := RESTRC(c, β_{ext})

end;

7.2.2. Eléments d'optimisation

Nous présentons les éléments d'optimisation présents dans cette version de l'algorithme.

1. EXTEND

Cette opération place un nouveau tuple (β_{in}, p, β_{out}) dans sat avec (β_{in}, p) défini. Elle calcule le β_{out} pour ce nouveau tuple selon la formule suivante :

$$\beta_{out} = \text{lub}\,\{sat(\beta'_{in}, p) : \beta'_{in} \leq \beta_{in} \text{ et } (\beta'_{in}, p) \in \text{dom}\,(sat)\}$$

L'optimisation consiste à ne pas mettre l'élément bottom comme β_{out} mais à calculer un β_{out} en fonction du sat existant. On pourra gagner une ou plusieurs itérations dans la procédure solve_allclauses.

2 . ADJUST

Cette opération améliore le β_{out} d'un tuple (β_{in}, p) existant dans sat. La procédure solve_allclauses peut nous donner un β_{out} plus grand que celui existant dans sat pour (β_{in}, p). Dans le même ordre d'idée que l'opération EXTEND, nous prenons le lub entre β_{out} et ceux existants dans sat pour les tuple dont le β_{in} est plus grand que le β_{in} de ce tuple.

3. SAME_SAT

Ce booléen indique si le sat à été modifié. S'il est à vrai l'ensemble sat n'a pas été modifié. Nous évitons ainsi une comparaison entre deux ensembles: l'ancien sat et le nouveau sat.

4. SUSPENDED

Suspended est un ensemble qui contient des tuples (β_{in}, p). Plus précisément suspended contient les tuples dont on a suspendu le calcul pour réaliser le calcul d'un autre tuple.

5 DEF

Cet ensemble contient l'ensemble des tuples (β_{in}, p) pour lequel on a trouvé un β_{out} définitif, qu'il n'est plus possible d'amélioré.

6. USE

Cet ensemble est un sous ensemble de suspended. Il contient les tuples (α_{in}, q) de suspended qui ont été utilisé pour le calcul d'un tuple (β_{in}, p).

7.2.3. Description des procédures

1. procédure solve

Elle recoit comme donnée en entrée le tuple (β_{in}, p). Elle calcule un β_{out} tel que le tuple (β_{in}, p, β_{out}) $\in$ au point fixe de TSAT.

2. procédure SOLVEGOAL

Elle recoit comme donnée en entrée le tuple (β_{in}, p), l'ensemble suspended, use, same-sat, sat et def . Dans sat, se trouve la meilleure valeur trouvée jusqu'à présent pour (β_{in}, p).

Elle calcule une nouvelle valeur pour (β_{in}, p), si ce tuple n'appartient pas à suspended, en faisant un appel à solve_procedure. Dans le cas contraire, elle prend la valeur présente dans sat.
Si le tuple appartient à DEF, alors la valeur pour ce tuple est dans sat et la meilleure possible. Un appel à solve_procedure ne sert à rien.

Dans le cas d'un appel à solve_procedure, on ajoute le tuple (β_{in}, p) à SAT si nécessaire par la procédure EXTEND.

A près l'appel à solve_procedure, si l'ensemble use en sortie est vide, cela signifie qu'on a utilisé aucune valeur approximative pour calculer le β_{out} du

tuple ($ß_{in}$, p) et donc c'est la meilleure valeur possible. on ajoute à DEF le tuple ($ß_{in}$, p) .

3. procédure SOLVE_PROCEDURE

Cette procédure a les mêmes paramètres que la procédure solve_goal. Elle répète les appels à la procédure solve_allclauses jusqu'à l'obtention de la meilleure approximation pour ($ß_{in}$, p). A la sortie de la procédure, on retire de USE le tuple ($ß_{in}$, p).

4. procédure SOLVE_ALLCLAUSES

Cette procédure implémente la fonction T_p. Elle calcule un $ß_{out}$ pour toutes clauses qui définissent p pour le tuple ($ß_{in}$, p). Elle prend l'UNION de tous les $ß_{out}$ trouvés et insère le nouveau $ß_{out}$ dans sat si nécessaire par la procédure ADJUST.

5. procédure SOLVE_CLAUSES

Cette procédure implémente la fonction T_c etT_b. La fonction sat($ß_{in}$, p) est remplacé par un appel à solve_goal. En effet il faut connaître un valeur pour un tuple dont on ne connaît pas encore sa meilleure approximation ou qui ne se trouve pas dans sat.

7.2.4. Exemple d'interprétation abstraite

Nous prenons l'interprétation abstraite de append(X_1, X_2, X_3) avec X_1 et X_2 ground, noté $X_1 \wedge X_2$. Le Schéma 7.1. illustre le première appel de la procédure solve_allclauses. Le schéma 7.2. illustre le deuxième et dernier appel de la procédure solve_allclauses.

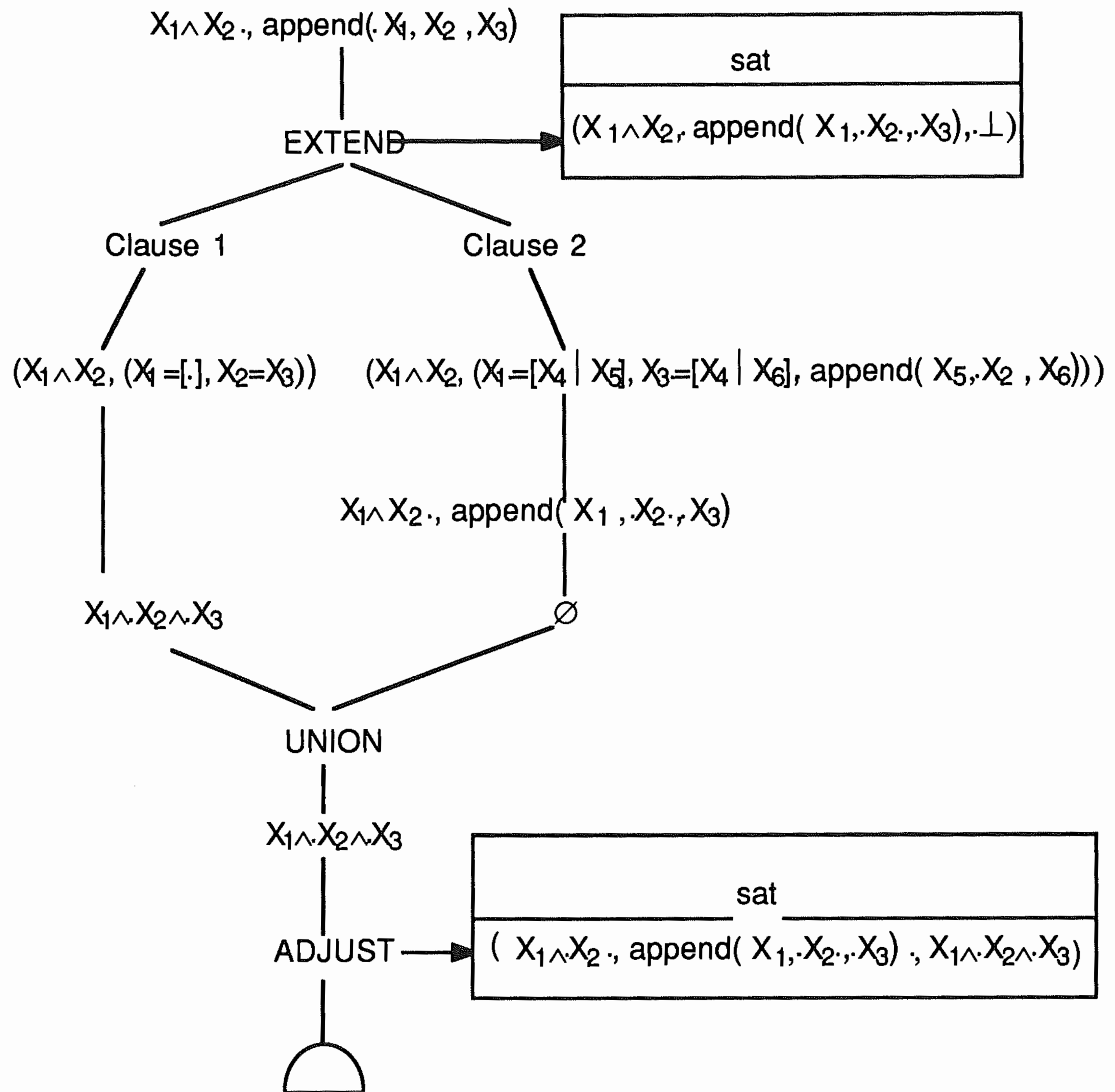


Schéma 7.1 :Premier appel de la procédure solve_allclauses

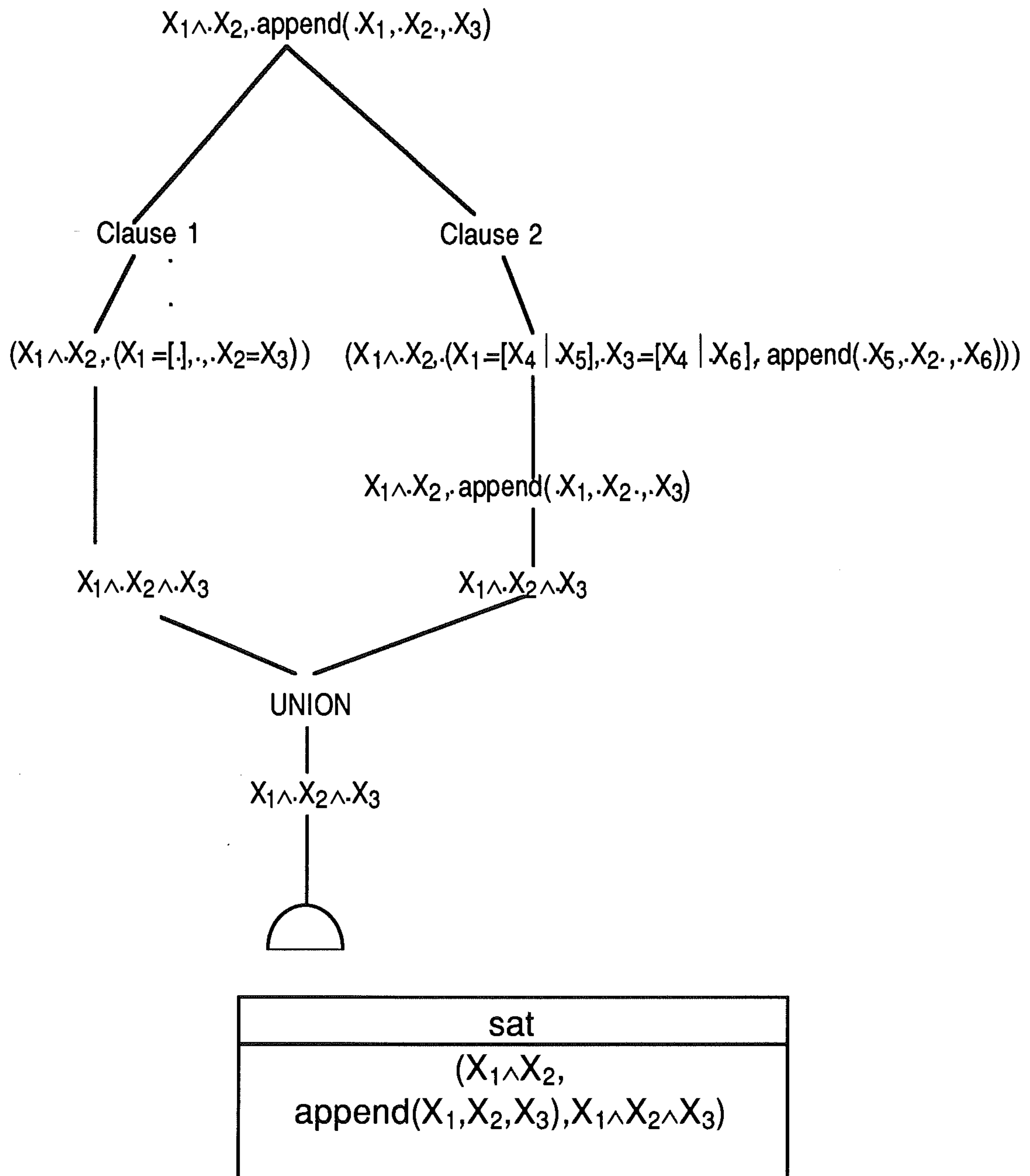


Schéma 7.2 : Deuxième et dernier appel de la procédure solve_allclauses

Lors du premier appel à la procédure solve_allclauses(schéma 7.1.), nous ajoutons à la table le tuple $((X_1 \wedge X_2)$, append(X_1,X_2,X_3), $\perp)$ avec EXTEND. L' exécution de solve_clause pour la première clause nous donne comme résultat $(X_1 \wedge X_2 \wedge X_3)$. Lors de l'exécution de solve_clause pour la deuxième clause, nous retrouvons le but initial. Nous utilisons alors la solution $\perp$ de la table. L'opération UNION prend le "lub" des deux résultats obtenus pour les deux clauses. Nous modifions ce tuple à la sortie de solve_allclauses en remplaçant $\perp$ par $(X_1 \wedge X_2 \wedge X_3)$ avec ADJUST.
Lors du deuxième appel à la procédure , nous exécutons de nouveau le but initial pour essayer de trouver une nouvelle solution supérieure à celle déjà trouvée. Dans notre cas , il n'est pas possible de trouver une meilleure solution puisque $(X_1 \wedge X_2 \wedge X_3)$ est la valeur la plus grande. Ce deuxième appel permet d'obtenir le point fixe. Nous trouvons pour les deux clauses le même résultat $(X_1 \wedge X_2 \wedge X_3)$. L'opération d' UNION nous donne ce résultat. L' opération d'ADJUST n'est pas exécutée puisque ce résultat $(X_1 \wedge X_2 \wedge X_3)$ n'est pas supérieur à celui présent dans la table mais est égal à celui- ci. nous trouvons ainsi le point fixe et l'exécution se termine.

7.2.5. L' implémentation de Prop

Nous avons présenté au point 3.2. le domaine abstrait Prop. Nous trouvons une implémentation du domaine dans (J-P Nelissen, 1991, p.25 à 36). A partir des éléments d' implémentation notamment la structure de données, nous détaillons de manière précise cette implémentation.

Le domaine se compose de tuples (ß, p, ß'). L'élément d' implémentation représentant ce tuple s'appelle TUPLE3.
"Tuple3 : PC[beta1: formule, p: INT, beta2: formule] "
(J-P Nelissen, 1991, p.34).

L' implémentation du tuple consiste à trouver une représentation pour ß et ß' deux substitutions abstraites et p un appel de procédure. Les substitutions sont représentées sous formes de formules et p est représenté par un entier qui sert entrée dans une table où sont défini tous les prédicats

L' élément d' implémentation représentant les substitutions abstraites s'appelle FORMULE.

"Formule : PC[nbXCjct: INT, nbCjct: INT, djct: SEQ[Cjct]".
(J-P Nelissen, 1991, p.29).

La formule est composée de trois éléments:

1. une liste linéaire de conjonctions
2. une variable qui indique la taille de cette liste
3. la taille d'une conjonction (nombre de variables)

La conjonction : notée Cjct est représentée comme suit :

"Cjct: SEQ [{0,1}]
Si la variable X_i apparait positivement dans la conjonction
alors $c_i = 1$
sinon $c_i = 0$"
(J-P Nelissen, 1991, p.26).

Nous représentons une conjonction par une séquence de {0,1}. Lorsque la variable apparaît positivement, elle est représentée par un 1 et par un 0 lorsqu'elle apparaît négativement. Une conjonction est donc la représentation binaire d'une entier.

Dans cette implémentation des formules, un choix est pris d'utiliser uniquement les connectifs $\wedge$, $\vee$ et $\neg$. Une transformation est donc nécessaire pour passer d'une représentation à l'autre.

forme 1.	forme 2.
$X_1 \wedge (X_2 \leftrightarrow X_3)$	$X_1 \wedge \neg X_2 \wedge \neg X_3$
	$\vee$
	$X_1 \wedge X_2 \wedge X_3$

Exemple 7.1.

La forme 2 s'appelle la forme normale disjonctive. Cette transformation n'est que théorique; le programme utilise les formules uniquement sous la forme 2.

Cette représentation est ingénieuse car elle exprime tous les résultats possibles au sujet de la groundness des variables. Lors de l'interprétation

abstraite, il suffit de supprimer les conjonctions dans la formule qui ne concordent pas. Toutes les opérations sur les formules se résumeront soit à la création ou soit à la modification de formule. Dans ce cas, il suffit de supprimer certaine conjonctions de la formule.

7.3. Les corrections

Nous avons tenté de corriger un programme trouvé dans (J-P Nelissen, 1991). Ce programme est l' implémentation de l'algorithme décrit au point 7.2.1. Nous avons procédé à cinq corrections que nous donnons en annexes.

7.4. Exemples d' exécutions

7.4.1. Queens

Nous donnons maintenant le contenu du fichier resultat.iab après exécution de l'algorithme trouvé dans (J-P Nelissen, 1991) après ajout dans le programme des corrections données au point 7.3. , avec comme inputs le programme queens et $(X1 \wedge X2) \vee (X1 \wedge \neg X2)$ comme donnée.

1. Programme QUEENS

```
queens (X1, X2) :-
        perm (X1, X2), safe (X2).

perm (X1, X2) :-
        X1=[], X2=[].

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

delete (X1, X2, X3) :-
        X2=[X1|X3].
```

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

safe (X1) :-
        X1=[].

safe (X1) :-
        X1=[X2|X3], X4= [],
        noattack (X2, X3, X4),
        safe(X3).

noattack (X1, X2, x3) :-
        X2=[].

noattack (X1, X2, x3) :-
        X2=[X4|X5],
        inegal(X1,X4), plus(X6, X4, X3),
        inegal(X1,X6), plus(X7, X1, X3),
        inegal(X4,X7), X10=[], plus(X9, X3, X10),
        inegal(X8,X9),
        noattack(X1, X5, X8).

inegal (X1, X2) :-
        X1=[], X2=[].

plus (X1, X2, X3) :-
        X1=[], X2=[], X3=[].
```

2. Trace

Contenu de sat après corrections :	Contenu de sat avant corrections :
sat en sortie Tuple3 de symbole #2 : queens beta1 1 10 3 11 beta2 3 11 Tuple3 de symbole #3 : perm beta1 1 10 3 11 beta2 3 11 Tuple3 de symbole #3 : perm beta1 2 01 3 11 beta2 3 11 Tuple3 de symbole #5 : delete beta1 2 010 3 110 6 011 7 111 beta2 7 111	sat en sortie Tuple3 de symbole #2 : queens beta1 1 10 3 11 beta2 1 10 3 11 Tuple3 de symbole #3 : perm beta1 1 10 3 11 beta2 1 10 3 11 Tuple3 de symbole #3 : perm beta1 formule vide de nbx = 2 beta2 formule vide de nbx = 2 Tuple3 de symbole #5 : delete beta1 2 010 3 110 6 011 7 111 beta2 6 011

Tuple3 de symbole #5 : delete
beta1
1 100
3 110
5 101
7 111
beta2
1 100
7 111
Tuple3 de symbole #4 : safe
beta1
formule vide de nbX=1
beta2
formule vide de nbX=1
Tuple3 de symbole #4 : safe
beta1
1 1
beta2
1 1

Tuple3 de symbole #6 : noattack
beta1
7 111
beta2
7 111
Tuple3 de symbole #6 : noattack
beta1
formule vide de nbX=3
beta2
formule vide de nbX=3

Tuple3 de symbole #5 : delete
beta1
formule vide de nbx = 3
beta2
formule vide de nbx = 3
Tuple3 de symbole #4 : safe
beta1
formule vide de nbX=1
beta2
formule vide de nbX=1
Tuple3 de symbole #4 : safe
beta1
0 0
1 1
beta2
0 0
1 1
Tuple3 de symbole #4 : safe
beta1
1 1
beta2
1 1
Tuple3 de symbole #6
noattack
beta1
formule vide de nbX=3
beta2
formule vide de nbX=3

Tuple3 de symbole #6 :
noattack
beta1
3 110
7 111
beta2
3 110
7 111

Tuple3 de symbole #7 : inegal
beta1
formule vide de nbX=2
beta2
formule vide de nbX=2
Tuple3 de symbole #7 : inegal
beta1
3 11
beta2
3 11
Tuple3 de symbole #7 : inegal
beta1
1 10
3 11
beta2
3 11
Tuple3 de symbole #7 : inegal
beta1
2 01
3 11
beta2
3 11

Tuple3 de symbole #6 :
noattack
beta1
4 001
5 101
6 011
7 111
beta2
6 011
7 111
Tuple3 de symbole #6 :
noattack
beta1
7 111
beta2
7 111

Tuple3 de symbole #7 : inegal
beta1
formule vide de nbX=2
beta2
formule vide de nbX=2

Tuple3 de symbole #7 : inegal
beta1
0 00
1 10
2 01
3 11
beta2
3 11
Tuple3 de symbole #7 : inegal
beta1
3 11
beta2
3 11

Tuple3 de symbole #8 : plus beta1 6 011 7 111 beta2 7 111 Tuple3 de symbole #8 : plus beta1 formule vide de nbX=3 beta2 formule vide de nbX=3 Tuple3 de symbole #8 : plus beta1 2 010 3 110 6 011 7 111 beta2 7 111 ... 17 tuples dans la liste ... and the winner is ... (X1^X2)	Tuple3 de symbole #8 : plus beta1 formule vide de nbX=3 beta2 formule vide de nbX=3 Tuple3 de symbole #8 : plus beta1 6 011 7 111 beta2 7 111 ... 16 tuples dans la liste ... and the winner is ... (X1^¬X2) | (X1^X2)

7.4.2. SORT

Nous donnons maintenant la trace du fichier resultat.iab pour l'exécution du même algorithme pour le programme sort avec $(X1 \wedge X2) \vee (X1 \wedge \neg X2)$ comme donnée.

1. Programme SORT

```
sort (X1, X2) :-
        X3=[], quicksort (X1, X2, X3).

quicksort (X1, X2, X3) :-
        X1=[], X3=X2.

quicksort (X1, X2, X3) :-
        X1=[X4|X5],
        partition (X5, X4, X6, X7),
        quicksort (X7, X8, X3),
        X9=[X4|X8],
        quicksort (X6, X2, X9).

partition (X1, X2, X3, X4) :-
        X1=[], X3=[], X4=[].

partition (X1, X2, X3, X4) :-
        X1=[X5|X6], X3=[X5|X7], inegal(X5, X2),
        partition (X6, X2, X7, X4).

partition (X1, X2, X3, X4) :-
        X1=[X5|X6], X4=[X5|X7], inegal(X5, X2),
        partition (X6, X2, X3, X7).

inegal (X1, X2) :-
        X1=[], X2=[].
```

2. Trace

sat en sortie

Tuple3 de symbole #2 : sort
beta1
1 10
3 11
beta2
3 11
Tuple3 de symbole #3 : quicksort
beta1
5 101
7 111
beta2
7 111
Tuple3 de symbole #3 : quicksort
beta1
formule vide de nbX=3
beta2
formule vide de nbX=3

Tuple3 de symbole #4 : partition
beta1
3 1100
7 1110
11 1101
15 1111
beta2
15 1111
Tuple3 de symbole #4 : partition
beta1
formule vide de nbX=4
beta2
formule vide de nbX=4
Tuple3 de symbole #5 : inegal
beta1
3 11
beta2
3 11

Tuple3 de symbole #5 : inegal
beta1
formule vide de nbX=2
beta2
formule vide de nbX=2 ... 7 tuples dans la liste
.. and the winner is ...
(X1^X2)

7.4.2. QUICKSORT

Nous présentons le résumé des exécutions de l'algorithme avec comme donnée le programme QUICKSORT.

1. X_1 est ground.
On obtient $X_1 \wedge \neg X_2 \wedge \neg X_3$ ou $X_1 \wedge X_2 \wedge X_3$

2. X_2 est ground.
On obtient $\neg X_1 \wedge X_2 \wedge X_3$ ou $X_1 \wedge X_2 \wedge X_3$

3 X_3 est ground.
On obtient $\neg X_1 \wedge \neg X_2 \wedge X_3$ ou $X_1 \wedge \neg X_2 \wedge X_3$ ou $X_1 \wedge X_2 \wedge X_3$

4. X_1 et X_2 sont ground
On obtient $X_1 \wedge X_2 \wedge \neg X_3$ ou $X_1 \wedge X_2 \wedge X_3$

5. X_2 et X_3 sont ground
On obtient $\neg X_1 \wedge X_2 \wedge X_3$ ou $X_1 \wedge X_2 \wedge X_3$

6 X_1 et X_2 et X_3 sont ground
On obtient $X_1 \wedge X_2 \wedge X_3$

7 rien n'est ground.
On obtient $\neg X_1 \wedge \neg X_2 \wedge \neg X_3$ ou $X_1 \wedge \neg X_2 \wedge \neg X_3$
ou $\neg X_1 \wedge \neg X_2 \wedge X_3$ ou $X_1 \wedge \neg X_2 \wedge X_3$
ou $X_1 \wedge X_2 \wedge X_3$

8. X1 et X3 sont ground.
On obtient (X1^X2^X3)

Chapitre 8

Comparaison des deux modèles d'interprétation abstraite

Nous avons d'une part le modèle d'interprétation abstraite avec table basé sur une sémantique opérationnelle décrit au chapitre 6. Nous l'appelons désormais le modèle 1. D'autre part, le modèle d'interprétation abstraite basé sur une sémantique de point fixe décrit au chapitre 7 sera appelé le modèle 2.
Nous comparons ces deux modèles au niveau des résultats calculés. L'information produite par l'exécution d'un interpréteur pour chacun de ces deux modèles est -elle identique ? Ensuite, nous nous intéressons à caractériser la méthode de calcul d'un point de vue de l'efficacité. Nous terminons en donnant les opérations d'un modèle pour lesquelles nous trouvons une opération correspondante dans l'autre modèle.

8.1. Les résultats des interprétations abstraites

Une interprétation abstraite complètement terminée, pour le modèle 1, nous donne

1. un arbre composé de l'ensemble des buts engendrés à partir du but initial

2. une table comprenant tous les littéraux d'appel rencontrés et leurs solutions.
Le modèle 2 nous donne comme résultat une table appelée SAT comprenant un ensemble de tuples de la forme (β, p, β').

Les deux premiers éléments du tuple (β, p, β') (soient(β, p)) correspondent au littéral d'appel de la table du modèle 1. La troisième partie β' est une substitution sur les variables du prédicat dans le modèle 2. Dans le modèle 1, nous trouvons un ensemble de patterns de sortie portant sur les variables d'un but.

exemple :

p(X) ← q(X, Y), b(Z,W)
Dans le modèle 1, β' porte sur les variables de la tête de p, donc {X}

Dans le modèle 2, les patterns de sortie portent sur les variables de la clause p, donc {X, Y, Z, W }.

Les résultats ne sont donc pas les mêmes. Les informations collectées dans le modèle 1 correspondent le mieux à ce que fait le programme réellement. En effet, pour chaque appel d'une clause, nous conservons dans la table et l'arbre l'information à l'entrée et à la sortie.
Nous gardons ainsi les informations complètes sur toutes les exécutions possibles pour toutes les clauses des procédures du programme choisi. L'arbre représente toutes ces exécutions et la table collectionne les informations avant exécutions et après exécutions sur les clauses des procédures. Nous pouvons affirmer que l'information est complète dans ce modèle parce que toutes les informations sont conservées. Néanmoins, cette information reste fragmentée au niveau des clauses. Si une procédure est définies par 5 clauses, alors nous avons 5 solutions qui sont générées pour cette procédure.

Dans le modèle 2, nous ne retrouvons pas l'arbre. Nous collectons les littéraux d'appel dans SAT et une solution unique pour chaque littéral

d'appel. Chaque solution globalise les patterns du modèle 1 pour ce même littéral d'appel. En effet, dans le modèle 2, il existe une opération UNION qui produit à partir des patterns de sortie pour une procédure une solution unique. L'information que nous trouvons dans la table est globalisée au niveau procédure.

8.2. La méthode de calcul des interprétations abstraites

Nous observons deux approches différentes proposées par les deux modèles. L'approche opérationnelle est suivie pour le modèle 1 tandis que l'approche du point fixe est suivie dans le modèle 2.

"L'approche opérationnelle suit l'idée que l'interprétation abstraite doit simplement simuler l'exécution concrète sur un domaine abstrait." (B.Le Charlier, P.Van Henteryck, 1992b, p.7)

L' approche de point fixe est définie dans (B.Le Charlier, P.Van Henteryck, 1992b, p.7). :

"1. Une sémantique de point fixe est utilisée pour caractériser l'information qui doit être calculée.

2. les algorithmes qui calculent le point fixe sont conçus de manière indépendante par rapport à la sémantique originale."

Le modèle 2 calcule un ensemble d'informations mémorisées dans SAT. Le calcul est réalisé selon une sémantique de point fixe. Cette sémantique opère un choix parmi l'ensemble des calculs possibles Certains calculs sont réexécutés jusqu'à l'obtention du point fixe.

Dans le modèle 1, nous gardons à tout moment l'ensemble des calculs réalisés pour les littéraux d'appel et leurs solutions.

Le modèle 2 opère différemment. Le modèle 2 calcule une table appelé SAT dans une première étape. Cette information peut servir ensuite pour

construire une information complémentaire. Ainsi lors de la compilation d' un programme, nous pouvons réaliser les optimisations grâce à l'information ainsi obtenue. Comme la solution pour un littéral d'appel est unique et vraie, nous pouvons l'utiliser directement et en déduire les optimisations nécessaires.

Le modèle 2 est moins fin mais est susceptible d'être plus efficace en temps et en espace. Des tests comparatifs doivent être réaliser pour confirmer cette affirmation.

8.3. Similitude et différence entre les opérations dans les deux modèles

8.3.1. Les opérations de mise à jour de la table SAT

Deux opérations manipulent la table. Il s'agit de Extend et Adjust. Extend ajoute un tuple (β, p, β') à la table. L'opération adjust modifie le β' des tuples de la table. Dans les deux opérations, un β' est modifié dans un tuple de la table. Or une realtion d' ordre sur le domaine abstrait doit être respectée. Celle-ci dit que pour tout (β, p, β'), (β_1, p, β_1'), ..., (β_n, p, β_n') appartenant à la table, si $\beta_1 > \beta$ alors $\beta_1' > \beta'$. Lorsque nous modifions β', il faut donc recalculer les β_i' des autres tuples qui sont comparables au tuple modifié.

Dans le modèle 1, cette relation d'ordre sur le domaine n'est pas utilisée. Les tuples sont ajoutés à la table ainsi que leur solutions sans ètablir de lien entre les tuples.

En utilisant cette relation d'ordre, nous pouvons gagner une ou plusieurs itérations dans le cacul.

Les deux modèles utilisent la table dans leurs calculs. Ils utilisent les solutions des littéraux d'appel, équivalents à ceux à résoudre, présentes dans la table quand ces littéraux sont rencontrés de nouveau par l'interpréteur.

8.3.2. La fonction de projection π

Dans le modèle 1, nous trouvons une seule fonction de projection. Cette fonction a pour objectif de changer la portée d'une substitution.
Dans le modèle 2, cette fonction de projection est spécialisée. Ainsi nous trouvons trois fonctions de projection : RESTRC, EXTC, RESTRG.

8.3.3. La fonction de composition $\oplus$

La fonction de composition est utilisée pour combiner ensemble des contraintes portant sur un ensemble de variable. Dans le modèle 1, il existe une seule fonction de composition. Dans le modèle 2, cette fonction de composition est spécialisée. Il en existe trois: AI_VAR, AI_FUNC, EXTG. La condition X définie au point 6.2.2. permet d'assurer que cette fonction de composition peut jour les trois rôles.

Troisième partie

Implémentation

Introduction

Dans cette partie, nous donnons et décrivons un algorithme pour le modèle d'interprétation abstraite décrit au chapitre 6. Nous donnons d'abord les modifications apportées à la méthode. Nous utilisons quelques éléments d' implémentation du modèle décrit au chapitre 7 pour lesquels l'équivalence avec les fonctions de composition et projection du modèle décrit au chapitre 6 est discutée au chapitre 8. Nous donnons ensuite la structure de données et les spécifications des procédures. Enfin, nous trouvons au point 9.4. l' algorithme et ses procédures.

Chapitre 9

un algorithme d'interprétation abstraite

9.1. Modifications par rapport à l'algorithme développé au point 6.2.4.

Nous proposons un ordre de parcours de l'arbre non arbitraire postfixé. Une liste linéaire de noeuds est créée où tout noeud de l'arbre est plus à gauche que son père; en ce qui concerne les fils, le premier créé est le plus à gauche, le deuxième créé le second à gauche, et ainsi de suite. Le sommet est le noeud le plus à droite.

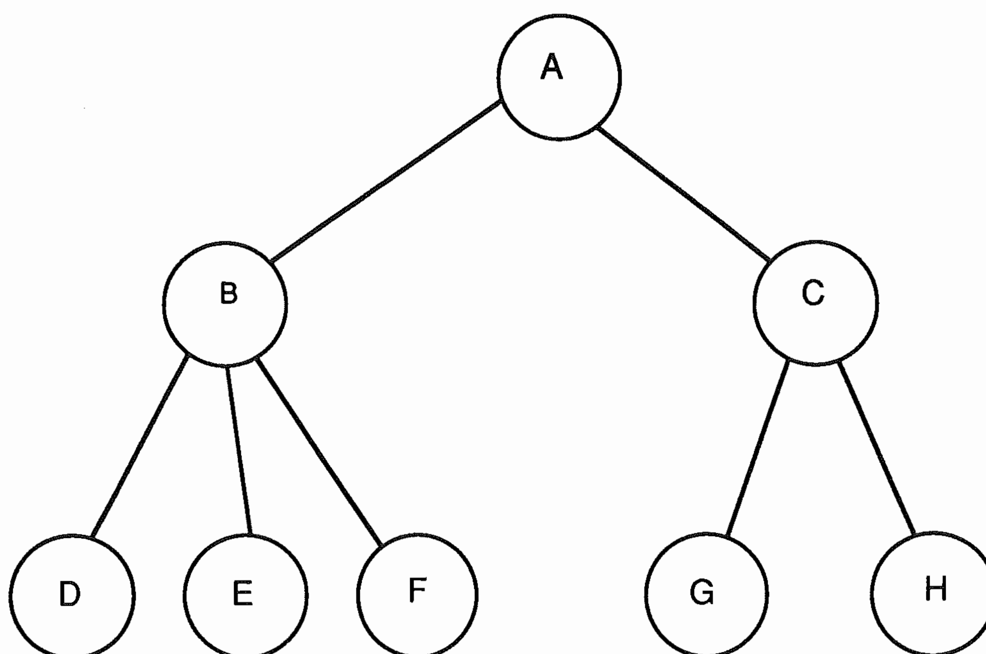


Schéma 9.1. : L'ordre de traitement des noeuds dans l'arbre : DEFBGHCA.

9.2 Structures des données

9.2.1. L' arbre

L'arbre est une liste de noeuds. Ils sont classés dans l'ordre du traitement. Cet ordre permet une recherche séquentielle du prochain noeud à examiner dans l'arbre.

Schéma 9.2. : La représentation de l'arbre sous forme de liste de noeuds

9.2.2. Les noeuds

Le noeud représente un but généralisé B dans lequel il faut pouvoir isoler le littéral le plus à gauche, le lg(B). Le noeud est composé de plusieurs items :

Nom de l' item n° 1: TYPE

Quatre type de noeud sont possibles :

- "noeud solution" : noeud pour lequel le lg(B), le littéral d'appel le plus à gauche, est un littéral d'appel non présent dans la table.

- "noeud de référence" : noeud pour lequel le lg(B) est un littéral d'appel présent dans la table.

- "noeud spécial" : noeud pour lequel le lg(B) est un marqueur de sortie.

- "noeud terminal" : noeud pour lequel le lg(B) est un but terminal.

Nom de l' item n° 2 : EXAMINE :

Il s'agit d'un booléen de visite.

Lorsque le noeud est créé, il prend la valeur 0 pour les noeuds "solution" et "spécial". Il passe à 1 lorsque le noeud a été examiné. Il est mis à 1 lors de la création du "noeud terminal" car celui-ci n'a plus besoin d'aucun traitement. Pour les noeuds "de référence" le booléen ne sert à rien puisque ceux-ci peuvent être examinés plusieurs fois.

Nom de l' item n° 3 : NOMBRE_SOLUTION_UTILISEE :

Cet item nous donne le nombre de solutions utilisées pour les noeuds "de référence."

Cet item ne sert que pour les noeuds de référence. En effet, il faut connaître le nombre de valeurs utilisées par ce type de noeud quand on examine celui-ci. En comparant ce nombre avec le nombre de valeur dans la liste à l'entrée référencée, nous pouvons connaître le nombre de nouveaux noeuds fils qu' il faut encore créer et qui correspondent aux valeurs non encore utilisées dans la table à l'entrée référencée par l'item 4 du noeud : Référence.

Nom de l' item n° 4 : REFERENCE :

Cet item est un pointeur vers un élément dans la table des solutions. Il sert uniquement pour les noeuds "de référence" et les noeuds "spécial".

Nom de l' item n°5 : VALEUR_ABSTRAITE :

Cet item représente la valeur abstraite c_1 du but $\leftarrow c_1,B_1,B_2,\ldots,B_n$.

Nom de l' item n°6 : BUT : une liste de de buts qui peuvent être un atome, un built-in, un marqueur de sortie

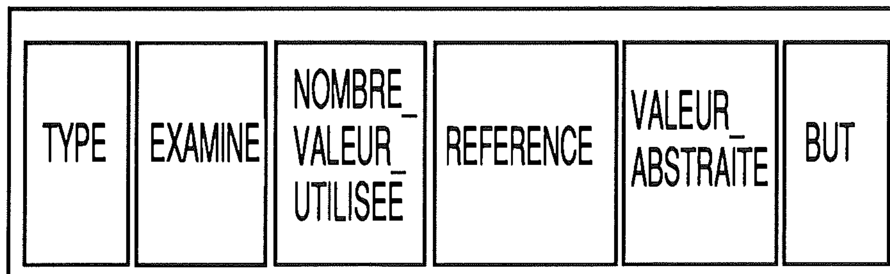


Schéma 9.3. : Représentation d'un noeud avec ses items.

9.2.3. La table

La table est une liste de littéraux d'appel auxquels une liste de valeurs abstraites, solutions pour ces littéraux d'appel, est associée. Un élément de cette table est donc un littéral d'appel avec sa liste de solutions.

9.3 Spécifications de l'algorithme

9.3.1 Terminologie.

Soit n : un noeud

1. lgbut(n.but) prend le but le plus à gauche de l'item but du noeud n.

2. forme(b) : fonction qui nous donne la forme d'un but. Quatre formes sont possibles :

   "built-in", "prédicat" , "vide", "terminal".

3 nombre_valeur(n.référence) : cette fonction donne le nombre de solutions de la table pour l' élément référencé par n.référence.

4. marqueur_de_sortie(p) = (p,c_a,c_b) ou p est un atome et c_a une valeur abstraite sur les variables de p lors de l'appel et c_b une valeur abstraite sur les variables de p utilisée pour la clause choisie.(cfr scénario 3 du point 6.2.3).
5. but_suivant (b) : fonction qui nous donne le but suivant de b.

9.3.2. Les spécifications des procédures

1. procédure solve(In: c_0,p; out : table,arbre)

La procédure solve reçoit en entrée le but initial qui est un littéral d' appel $[p,c_0]$. Cette procédure construit l'arbre qui contient la suite des buts engendrés à partir du but initial. Chaque but engendré donne naissance à un

noeud dans l'arbre. Cette procédure construit également la table décrite au point 9.2.3. Elle fait appel à la procédure solve_with_tabulation pour créer la table et l'arbre.

2. procédure solve_with_tabulation(Inout : table, arbre)

A partir d'un but généralisé initial placé dans le sommet de l'arbre, cette procédure développe ce but et développe la table des solutions ainsi que l'arbre. On détermine d'abord le prochain noeud à examiner avec la procédure détermine_noeud_courant. Elle applique un traitement au noeud courant tant qu'il existe un noeud courant dans l'arbre. Quand le noeud courant est vide, la procédure se termine. Le traitement à appliquer au noeud courant est fonction du type du noeud courant. Les traitements possibles sont

1. le traitement d'un noeud solution : procédure développe_noeud_courant;

2. le traitement d'un noeud de référence : procédure développe_noeud_de_référence.

3. le traitement d'un noeud spécial : procédure développe_noeud_de_spécial.

3. Procédure détermine_noeud_courant(In: arbre; Out: noeud_courant).

Cette procédure recherche le premier noeud à examiner dans l'arbre. Les noeuds étant classés par ordre de traitement de gauche à droite dans la liste, il suffit de parcourir la liste de manière séquentielle pour trouver le prochain noeud à examiner. Celui-ci doit respecter les propriétés suivantes pour être choisi comme noeud courant :

noeud.examine=0
pour un noeud "solution", un noeud "spécial"
ou
noeud.nombre_valeur_utilisée est < nombre_valeur (noeud.référence)
pour un noeud "de référence",

Si la liste est parcourue sans trouver un noeud courant, alors il ne reste plus de noeud à examiner et nous prenons la convention de dire que le noeud courant est vide.

4. Procédure développe_built_in(In :c_0; Inout: b)

Cette procédure prend en compte tous les built_in's successifs du corps b d'une clause. Elle calcule une nouvelle substitution c_0 à partir de c_0 et de c_1 trouvé en appliquant AI_VAR ou AIFUNC selon le type du built_in.

Nous restreignons c_0 aux variables du built_in avec la fonction RESTRG. Nous appliquons ensuite la fonction AIVAR ou AI_FUNC. Nous trouvons une nouvelle valeur c_1. Nous appliquons la fonction EXTG à c_0 et c_1. Nous lisons le but suivant du corps b. Si celui-ci est un built_in, nous recommençons le traitement. Dans le cas contraire, le traitement se termine. Nous sortons de cette procédure avec une nouvelle substitution c_0 et forme(b) $\in$ {"prédicat", "vide"}.

5. Procédure développe_noeud_solution.(in noeud_courant; Inout: arbre; Out: table)

Cette procédure applique un traitement au noeud courant qui est un noeud solution. Ce traitement est une mise en oeuvre du scénario 1 décrit au point 6.2.3.

Le littéral d'appel du noeud solution est
[noeud_courant.but, noeud_courant.valeur_abstraite]
Dans c_0, on trouve le résultat de l' application de la fonction RESTRG avec comme argument le littéral d'appel du noeud courant.
Nous plaçons ensuite le littéral d'appel dans la table et comme pattern de sortie, une liste vide avec la procédure étendre_la_table.
Il faut créer autant de fils dans l'arbre qu'il existe de clauses pour ce littéral d'appel. Pour chaque fils, il faut déterminer c_1 selon les règles décrites dans le scénario 1. Nous utilisons ici une fonction d'extension EXTC.

Il faut ensuite déterminer le type des noeuds fils. On lit le premier but de la clause et forme(b) nous donne la forme du but. Deux cas sont possibles :

- forme(b) = "built-in" , alors on exécute la procédure développe_built_in(c_1,b) qui nous donne une nouveau c_1 et b tel que forme(b) $\in$ {"prédicat", "vide"}.

Dans le cas "vide", il faut créer un noeud spécial avec la procédure crée_noeud_spécial.

Dans le cas "prédicat", il faut donc créer un fils de type "noeud solution" ou " noeud de référence " , avec c_1 comme valeur abstraite, selon forme(b) obtenu après l'exécution de développe_built_in(c_1,b). Pour le choix entre un "noeud solution" et " noeud référence", il suffit d'appliquer le cas où forme(b) ="prédicat".

- forme(b) = "prédicat", alors il faut créer un fils de type "noeud solution" ou " noeud de référence" avec c_1 comme valeur abstraite. On détermine le type en testant l'appartenance du littéral d'appel [b,c_2] à la table où c_2 est la restriction de c_1 aux variables du prédicat b.

Rem : On utilise les procédures crée_noeud_solution, crée_noeud_de_référence, crée_noeud_spécial, pour ajouter à l'arbre les nouveaux noeuds. Ceci est aussi valable pour les procédures procédure développe_noeud_de_référence et procédure développe_noeud_spécial.

.

Procédure développe_noeud_de_référence

Le littéral d'appel du noeud de référence est [noeud_courant.but, noeud_courant.valeur_abstraite]
Nous appliquons le scénario 2 du point 6.2.3.
Ce littéral d'appel à un littéral équivalent dans la table à l'entrée référencée par noeud_courant.référence.

Il faut créer autant de noeud fils qu'il existe de solutions non utilisées dans la table pour le littéral d'appel.
Pour chaque fils, il faut déterminer la valeur abstraite c_0 . La valeur c_0 = noeud_courant.valeur_abstraite; on détermine c_0 en étendant avec la procédure extg la valeur abstraite c_1 trouvée dans la liste à l'entrée noeud_courant.référence aux variables de c_0 . (cfr scénario 2 du point 2.2.3).

Il faut ensuite déterminer le type du noeud à créer. On lit le but suivant du but généralisé et forme(b) nous donne la forme du but.

Trois cas sont possibles :

- forme(b) = “built-in” alors voir Procédure développe_noeud_solution en utilisant c_0 à la place de c_1.
- forme(b) = “prédicat”, alors voir Procédure développe_noeud_solution en utilisant c_0 à la place de c_1.
-forme(b) = “de référence” , alors il faut créer un fils de type “ noeud de référence “ avec c_0 comme valeur abstraite.

7. Procédure développe_noeud_spécial.

Le littéral du noeud spécial est [noeud_courant.but, noeud_courant.valeur_abstraite]

Nous appliquons le scénario 3.

Il faut d'abord calculer la valeur abstraite à placer dans la table. Il faut restreindre la valeur abstraite (noeud_courant.valeur_abstraite) trouvée pour le but qu'on vient de résoudre aux variables de la tête de la clause utilisée. On obtient c_0. On peut alors placer cette valeur abstraite dans la table.

Cette procédure crée un noeud fils qui peut être un noeud "terminal", un noeud "solution", un noeud de "référence" ou un noeud "spécial".

Il faut maintenant définir la valeur abstraite du fils. Il suffit d'étendre et de faire correspondre c_0 à c_1 , pour revenir dans l' ensemble de variable de l'appel du prédicat pour lequel on a trouvé une valeur abstraite. Il faut ensuite déterminer le type du noeud fils. On lit le but suivant b et forme(b) nous donne la forme de b.

Quatre cas sont possibles :

- forme(b) = "terminal" , alors il faut créer un noeud terminal avec c_1.
- forme(b) = "built-in" alors alors voir Procédure développe_noeud_solution en utilisant c_0 à la place de c_1.
- forme(b) = "vide" , alors il faut créer un noeud spécial avec c_1.

- forme(b) = "prédicat" alors voir Procédure développe_noeud_solution en utilisant c_0 à la place de c_1.
- forme(b) = "de référence" alors voir Procédure développe_noeud_solution en utilisant c_0 à la place de c_1.

Nous mettons à 1 le booléen noeud_courant.examine.

9.4 Algorithme

```
procédure solve (c0,p,table,arbre)
begin
        table ← ø
        crée_sommet_de_l_arbre(c0,p,arbre)
        solve_with_tabulation(table,arbre)
end


procédure solve_with_tabulation (table ,arbre)
begin
        détermine_noeud_courant(noeud_courant,arbre)
        while (noeud_courant <> "vide")
        do begin
                case noeud_courant.type= "noeud solution":
                                développe_noeud_solution
                case noeud_courant.type= "noeud de référence" :
                                développe_noeud_de_référence
                case noeud_courant.type= "noeud spécial" :
                                développe_noeud_spécial
                end
                détermine_noeud_courant(noeud_courant,arbre)
        end
end
```

```
développe_built_in(c0,b)
begin
        while (forme(b) = "built-in" )
        do begin
                var_de_built_in ←restrg (b ,c0)
                if forme(b).type = "Xj = Xk "
                then c1←aivar(var_de_built_in)

                if forme(b).type = "Xj = f(...)"
                then c1← aifunc(var_de_built_in,f)
                c0← extg(b,c0,c1)
                b ← but_suivant(b)
        end
end

développe_noeud_solution
begin
        c0 ← restrg(lgbut(noeud_courant.but),noeud_courant.valeur_abstraite)
        étendre_la_table ([lgbut(noeud_courant.but),c0] ,table)
        clause← lire_clause(lgbut(noeud_courant.but))
        while (lire_clause <> vide) do
        begin
                c1← extc(clause, c0))
                b ← premier_but(clause)
                if forme(b) = "built-in" then
                begin
                        c1 ← développe_built_in (c1 , b )
                        if forme(b) ="vide"
                        then créer noeud spécial(c1 , b )
                end
                if forme(b) = "prédicat" then
                begin
                        c2 <-restrg(b,c1)
                        if littéral d'appel[b,c2] est une entrée de table
                        then crée_noeud_de_référence(c1,b)
                        else crée_noeud_solution(c1,b)
                end
                clause← lire_clause(lgbut(noeud_courant.but))
```

```
        end
        noeud_courant.examine = 1
end


développe_noeud_de_référence
begin
        b ← lgbut(noeud_courant.but)
        while (noeud_courant.nombre_valeur_utilisée)
                        < nombre_valeur(noeud_courant.référence))  do
        begin
                noeud_courant.nombre_valeur_utilisée =
noeud_courant.nombre_valeur_utilisée          +1
                c0 ← noeud_courant.valeur_abstraite
                c1 ← valeur_abstraite_de(noeud_courant.nombre_valeur_utilisée )
                c0← extg(b,c0,c1)
                b ← but_suivant (noeud_courant.but)
                if forme(b) = "built-in"
                then c0← développe_built_in(c0,b)
                if forme(b) = "vide"
                then crée_noeud_special(c0,b)
                else if forme(b) = "prédicat"  then
                        begin
                                c1 ← restrg(b,c0)
                                if littéral d'appel[b,c1] est une entrée de table
                                then crée_noeud_de_reference(c0,b)
                                else crée_noeud_solution(c0,b)
                        end
        end
end


développe_noeud_special
begin
        c0←restrc(lgbut(noeud_courant.but).p,
                                        noeud_courant.valeur_abstraite)
        ajouter_a_la_table(noeud_courant.entrée,c0)
        c1← extg(c1,c0)
```

```
        b ← but_suivant(noeud_courant.but)
        if forme(b) = "terminal"
        then crée_noeud_terminal(c1,b)
        else begin
                        if forme(b) = "built-in"
                        then c1← développe_built_in(c1,b)
                        if forme(b) = "vide"
                        then crée_noeud_de_spécial(c1,b)
                        else if forme(b) = "prédicat" then
                         begin
                                c2← restrg(b,c1)
                                if littéral d'appel[b,c2] belong to table
                                then crée_noeud_de_référence   (c1,b)
                                else crée_noeud_solution(c1,b)
                        end
                end
        noeud_courant.examine = 1
end

procédure détermine_noeud_courant(noeud_courant,arbre)
begin
        noeud_courant ← vide
        courant ← faux
        noeud lu ← lire arbre
        fin arbre ← faux
        while (not courant && not fin arbre) do
        begin
                case noeud lu.type =        "noeud solution" ou
                                            "noeud spécial" :
                        if  noeud lu.examiné =0
                        then begin
                                        courant ← true
                                        noeud_courant ← noeud lu
                                end
                case noeud lu.type = "noeud de référence  " :
                        if noeud lu.nombre_valeur_utilisée
<nombre_valeur(noeud_courant.entrée))
```

```
                then begin
                        courant ← true
                        noeud_courant ← noeud  lu
                    end
            case noeud  lu.type = "noeud terminal":
                rien
            end
            if not courant
            then noeud lu ← lire arbre
        end
end


crée_noeud_solution(c,b)
begin
        noeud.type = "noeud solution"
        noeud.examine = 0
        noeud.référence = ∅
        noeud.nombre_valeur_utilisée =0
        noeud.valeur_abstraite = c
        noeud.but = (b,marqueur_de_sortie(b),reste(b))
        ajoute_noeud(noeud,arbre)
end
crée_noeud_de_référence(c,b)
begin
        noeud.type = "noeud_de_référence"
        noeud.examine = 0
        noeud.référence =  recherche_entrée(b,restrg(b,c))
        noeud.nombre_valeur_utilisée = vide
        noeud.valeur_abstraite = c
        noeud.but = (b,reste(b))
        ajoute_noeud(noeud,arbre)
end
```

```
crée_noeud_special(c,b)
begin
        noeud.type = "noeud spécial"
        noeud.examine = 0
        noeud.référence = recherche_entrée(p,ca)
        noeud.nombre_valeur_utilisée = 0
        noeud.valeur_abstraite = c
        noeud.but = (b.reste(b))
        ajoute_noeud(noeud,arbre)
end
crée_noeud_terminal(c,b)
begin
        noeud.type = "noeud terminal"
        noeud.examine = 1
        noeud.référence = ∅
        noeud.nombre_valeur_utilisée = 0
        noeud.valeur_abstraite = c
        noeud.but = ∅
        ajoute_noeud(noeud,arbre)
end
```

Remarque : On pourrait construire une liste contenant les noeuds à traiter. Chaque fois qu' une nouvelle solution est trouvée pour un noeud solution, nous ajoutons à la liste des noeuds à traiter l'ensemble des noeuds de référence qui utilisent cette solution. nous avons donc besoin de pointeur dans la table qui référence les noeuds de référence.

Conclusions

Ce travail est une étude de deux modèles d'interprétation abstraite. Nous avons d'abord, au travers des modèles rencontrés, étudié l'interprétation abstraite. Nous savons que l'interprétation abstraite est une méthode qui permet de rechercher des informations sur des programmes afin de les optimiser. Les programmes logiques sont principalement concernés par ces optimisations. En effet, ceux-ci sont écrits indépendamment de la manière de calculer le résultat.

L'interprétation abstraite peut nous aider à optimiser ces programmes, en utilisant les informations qu'elle nous donne, lors de la compilation du programme Prolog.

Nous avons étudié plus particulièrement deux modèles d'interprétation abstraite. Le premier modèle étudié est décrit dans (P. Codognet, G. Filé , 1992). Ce premier modèle présente la particularité de définir une interprétation abstraite par une simulation d'une exécution concrète. Le problème de ce type d'interprétation est le nombre élevé de calcul à réaliser. Un mécanisme à été introduit dans ce modèle pour rendre le calcul moins grand. Il s'agit d'utiliser les résultats de la table.

Le second modèle est décrit dans (B. Le Charlier, K. Musumbu, P. Van Hentenryck, 1990). Un travail d' implémentation a été réalisé dans (J-P Nelissen, 1991). Nous avons testé le programme proposé sur quelques programmes Prolog avec un domaine abstrait décrit dans (A. Cortesi, G. Filé, W. Winsborough, 1991). Ce domaine abstrait permet d'étudier l'analyse de mode des variables.

La première idée triviale était de comparer ces deux modèles. Nous avons d'une part un travail de comparaison théorique à réaliser et d'autre part un programme à réaliser. Le but était de trouver tout ce qui pouvait être réutilisé de l' implémentation du deuxième modèle pour l' implémentation du premier modèle. Nous avons pu établir un lien entre les résultats que nous donne une interprétation abstraite d'un programme dans les deux modèles. Ensuite nous avons trouvé les fonctions et opérations qui pouvaient être comparées. Nous pouvons, du point de vue théorique, retenir quelques enseignements précieux. Le premier modèle nous donne des informations au niveau des clauses tandis que le deuxième modèle nous donne des informations au niveau des procédures.

Il serait maintenant intéressant de réaliser un interpréteur pour le premier modèle. Nous pourrons ainsi comparer son efficacité sur divers domaines. Nous avons, à ce dessein, raffiné l'algorithme pour ce modèle d'une part d'un point de vue théorique au chapitre 6. Nous avons proposé ensuite un algorithme qui utilise les fonctions implémentées dans le deuxième modèle. Il reste maintenant à implémenter cet algorithme.

Bibliographie

P. Codognet, G. Filé (1992), "Computations, Abstractions and Constraints in Logic Programs" , *In Proc ICCL 92.*

A. Cortesi (1992). *Domini astratti per l analisi statica di programmi logici,* Thèse de doctorat, Universita degli Studi di Padova.

A. Cortesi, G. Filé, W. Winsborough (1991), "Prop revisited: Propositional Formula as Abstract Domain for Groundness Analysis", *In proc LICS 91, IEEE symposium on Logic in Computer Science, Amsterdam .*

B. Le Charlier (1991), "L'analyse statique des programmes par l'interprétation abstraite", Institut d'Informatique, F.U.N.D.P. Namur.

B. Le Charlier, K. Musumbu, (1992), "Une sémantique opérationnelle instrumentale pour Prolog et son application à la preuve de consistance d'un modèle d'interprétation abstraite.", *In J-P Delahaye, editor, Actes des journées francophones de programmation logique (JFFL 92) .*

B. Le Charlier, K. Musumbu, P. Van Hentenryck (1990), "Efficient and Accurate Algorithms for the Abstract Interpretation of Prolog Programs", *Rapport interne,* Institut d'Informatique, F.U.N.D.P. Namur.

B. Le Charlier, K. Musumbu, P. Van Hentenryck (1991), "A Generic Abstract Interpretation Algorithm and its complexity analysis", *In Proc ICLP 91.*

B. Le Charlier, P. Van Hentenryck (1992a), "Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog" , *In Proc ICCL 92,* San Francisco.

B. Le Charlier, P. Van Hentenryck (1992b), "On the design of generic abstract Interpretation frameworks", *in W.S.A.'92,* Bordeaux.

J. W. Lloyd (1987), *Foundations of logic Programming,* Springer Verlag.

K. Musumbu (1990), *Interprétation abstraite des programmes Prolog,* Thèse de doctorat (Phd), Institut d'Informatique, F.U.N.D.P. Namur.

J-P Nelissen (1991), *Interprétation Abstraite : Analyse de Mode au moyen de Formules Logiques,* mémoire, Institut d'Informatique, F.U.N.D.P. Namur.

L. Sterling, E. Shapiro (1986), *The art of prolog : Advanced programming techniques,* The MIT press.

D.S.Warren (1992), "Memoing for Logic Programs", *Communications of the ACM,* Vol 35 n° 3, p.94 -111.

o

Annexes

Nous présentons les modifications apportées à l'algorithme donné et commenté dans (J-P Nelissen, 1991). Nous donnons chaque fois la version avant correction que nous appelons version originale ainsi que la version corrigée. En gras se trouve la ligne concernée par la modification. Entre parenthèses, nous trouvons le nom du fichier dans lequel se trouve la procédure. Une justification est donnée pour chaque modification.

Modification 1.:

Il a été nécessaire de modifier les déclarations des procédures car ceux-ci contenaient les déclarations de types , dans les fichiers avec suffixe c et h.

exemple: La déclaration de la procédure SOLVE est écrite dans la version originale comme ceci: ptrBeta SOLVE(ptrBeta betaIn, ptrProc p). Dans la version corrigée, elle devient :

```
ptrBeta SOLVE(betaIn,p)
ptrBeta betaIn;
ptrProc p;
```

<u>Modification 2.: La procédure CHGVAR</u>
(primitives.c)

Version Originale :

```
ptrFormule CHGVAR(ptrFormule beta0)
{/*     pre :
        post : on étend beta0 aux variables de gamma
        qu'on supprime.     on renvoie beta1
*/
        int iX= 0, j;
        ptrElement pcBeta0, pcGamma;
        ptrDjct plcBeta0= beta0->pDjct;
        ptrDjct plcGamma;
        ptrCjct pc;
        ptrFormule beta1;

        plcGamma= pGamma->pDjct;
        beta1= nouvelleFormule(pGamma->nbXCjct);
        pc= nouvelleCjct(); /* à libérer */
/* pour toute Cjct de beta0 */
        for(pcBeta0= plcBeta0->tete->suivant;
                        pcBeta0!= plcBeta0->fin ;
                        pcBeta0= pcBeta0->suivant) {
                effaceCjct(pc
                );
/* pour toute Cjct de gamma */
        for(iX= 1,pcGamma= plcGamma->tete->suivant;
                pcGamma!= plcGamma->fin ;
                iX++, pcGamma= pcGamma->suivant)
        {
                if (j= premierXiCjct(pcGamma->cle, iX)) {
                        if (vraiXiCjct(pcBeta0->cle, j)) {
                                affirmeXiCjct(pc, iX);
                        }
                }
                }
                insereCFormule(beta1,pc);
```

```
        }
        return(beta1);
}
```

Version Corrigée :

```
ptrFormule CHGVAR( beta0)
ptrFormule beta0;

{/*     pre :
        post : on étend beta0 aux variables de gamma
        qu'on supprime.     on renvoie beta1
*/
        int iX= 0, j;
        ptrElement pcBeta0, pcGamma;
        ptrDjct plcBeta0= beta0->pDjct;
        ptrDjct plcGamma;
        ptrCjct pc;
        ptrFormule beta1;

        plcGamma= pGamma->pDjct;
        beta1= nouvelleFormule(pGamma->nbXCjct);
        pc= nouvelleCjct(); /* à libérer */
/* pour toute Cjct de beta0 */
        for(pcBeta0= plcBeta0->tete->suivant;
                        pcBeta0!= plcBeta0->fin ;
                        pcBeta0= pcBeta0->suivant) {
                effaceCjct(pc
                );
/* pour toute Cjct de gamma */
        for(iX= 1,pcGamma= plcGamma->tete->suivant;
                pcGamma!= plcGamma->fin ;
                iX++, pcGamma= pcGamma->suivant)
        {
                if (j= premierXiCjct(
                pcGamma->cle,pGamma->nbXCjct)) {
                        if (vraiXiCjct(pcBeta0->cle, j)) {
```

```
                    affirmeXiCjct(pc, iX);
                }
            }
            }
            insereCFormule(beta1,pc);
        }
        return(beta1);
}
```

La variable nbXCjct représente le nombre de variables dans une conjonction de variables. Celui-ci est constant. La variable iX est un compteur qui compte le nombre de conjonctions présentes dans la disjonction de conjonctions. Il faut donc remplacer iX par nbXCjct puisque nbXCjct a la même signification que le deuxième argument de la procédure premierXiCjct dans lequel il se trouve.

Modification 3 : La procédure EXTEND (primitives.c)

Version Originale:

```
ptrListeTuple3 EXTEND
        (ptrBeta beta, ptrProc p, ptrListeTuple3 sat)
{/*     pre :
        post :  modificateur de sat
renvoie l'union de sat et de (beta,p,beta2) où beta2
est le plus grand des betaOut des tuples trouvés
quand on recherche dans sat s'il existe des tuples3
de même p et de betaIn <= à beta
sinon c'est bottom, une beta vide
*/
        ptrElement pe;
/* contient un tuple3 dans sa clé */
        ptrTuple3 pt, ptMax;
        ptrBeta pbMax;
/* pbMax contient bottom, soit la liste vide */
        pbMax= nouvelleFormule(beta->nbXCjct);
/* pour tout tuple de sat */
        for(pe= sat->tete->suivant;
                pe != sat->fin ;pe= pe->suivant) {
                pt= (ptrTuple3) pe->cle;
/* les tuples ont même pointeur de procédure */
        if ( (p == pt->p) &&
/* les betaIn des tuples sont <= à beta */
                (compareBeta(beta, pt->beta1) >=0)) {
/* si le betaOut est > au maximum actuel */
        if  (compareBeta(pbMax,  pt->beta2)>0)
                        pbMax= pt->beta2;
                }
        }
        ptMax= nouveauTuple3(beta, p,                   copieFormule(pbMax));
        insereElement(sat, ptMax, compareDomTuple3);
        return sat;
}
```

Version Corrigée:

```
ptrListeTuple3 EXTEND(beta,p,sat)
ptrBeta beta;
ptrProc p;
ptrListeTuple3 sat;
{/*     pre :
        post :  modificateur de sat
renvoie l'union de sat et de (beta,p,beta2) où beta2
est le plus grand des betaOut des tuples trouvés quand on recherche
dans sat si il existe des tuples3 de même p et de betaIn <= à beta
sinon c'est bottom, une beta vide
*/
        ptrElement pe;
/* contient un tuple3 dans sa clé */
        ptrTuple3 pt, ptMax;
        ptrBeta pbMax;
/* pbMax contient bottom, soit la liste vide */
        pbMax= nouvelleFormule(beta->nbXCjct);
/* pour tout tuple de sat */
        for(pe= sat->tete->suivant;
                 pe != sat->fin ;pe= pe->suivant) {
                pt= (ptrTuple3) pe->cle;
/* les tuples ont même pointeur de procédure */
        if ( (p == pt->p) &&
/* les betaIn des tuples sont <= à beta */
                (compareBeta(beta, pt->beta1) >=0)) {
/* si le betaOut est > au maximum actuel */
        if  (compareBeta(pt->beta2,  pbMax)  >0)
                        pbMax= pt->beta2;
                }}
        ptMax= nouveauTuple3(beta, p,                copieFormule(pbMax));
        insereElement(sat, ptMax, compareDomTuple3);
        return sat;
}
```

pbMax est initialise avec bottom; donc il est le plus petit des beta's. Le test ne sera jamais verifié. Il faut donc inverser les arguments, ce qui verifie les spécifications pour extend.

Modification 4 : La procédure compareBeta (formulation.c)

Version Originale :

```
int compareBeta(beta1, beta2)
ptrBeta beta1 beta2;
{/*     pre : beta1 et beta2 sont déclarés et triés
                ou pointent vers NULL
        post : un beta vide est plus petit que tout
                        >0 si beta1>beta2 ou beta1<>vide et beta2= vide
                        <0 si beta1<beta2 ou beta1= vide et beta2<>vide
                        =0 si beta1=beta2
*/
        ptrElement pe1,pe2;
        ptrDjct pd1,pd2;
        enum {EGALE,PREMIERE,SECONDE} betaLong;

        assert(beta1 != NULL);
        assert(beta2 != NULL);
        assert(beta1->nbXCjct = beta2->nbXCjct);
        if (beta1->nbCjct > beta2->nbCjct)
                betaLong= PREMIERE;
        else if (beta1->nbCjct < beta2->nbCjct)
                betaLong= SECONDE;
        else
                betaLong= EGALE;
        pd1= beta1->pDjct;
        pd2= beta2->pDjct;
        pe1= pd1->tete->suivant;
        pe2= pd2->tete->suivant;
        while( (pe1 != pd1->fin) && (pe2 != pd2->fin) ) {
                if ( !compareCjct(pe1->cle, pe2->cle) ) {
                        pe1= pe1->suivant; pe2= pe2->suivant;
                } else
                        switch(betaLong) {
                                case(EGALE):        assert("betas
incomparable"==0);        /* incomparable */
```

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

Version Corrigée.:

```
int compareBeta(beta1,beta2)
ptrBeta beta1;
ptrBeta beta2;
{/*     pre : beta1 et beta2 sont declares et tries
                ou pointent vers NULL
        post : un beta vide est plus petit que tout

                  =1 si beta1>beta2 ou beta1<>vide et beta2= vide

                    =0 si beta1=beta2

                    -1 si beta1<beta2 ou beta1= vide et beta2<>vide
```

```
                    -2 si beta1->nbCjct = beta2->nbCjct et beta1 et bea2
incomparable
                    -3 si beta1->nbCjct > beta2->nbCjct et beta1 et beta2
incomparable
                    -4 si beta1->nbCjct < beta2->nbCjct et beta1 et beta2
incomparable

*/

	ptrElement pe1,pe2;
	ptrDjct pd1,pd2;
	int continuer=1;
	enum {EGALE,PREMIERE,SECONDE} betaLong;
	assert(beta1 != NULL);
	assert(beta2 != NULL);
	assert(beta1->nbXCjct = beta2->nbXCjct);
	if (beta1->nbCjct > beta2->nbCjct)
		betaLong= PREMIERE;
	else if (beta1->nbCjct < beta2->nbCjct)
		betaLong= SECONDE;
	else
		betaLong= EGALE;
	pd1= beta1->pDjct;
	pd2= beta2->pDjct;
	pe1= pd1->tete->suivant;
	pe2= pd2->tete->suivant;
	while( (pe1 != pd1->fin) && (pe2 != pd2->fin) && (continuer == 1) ) {
		if ( !compareCjct(pe1->cle, pe2->cle) ) {
			pe1= pe1->suivant; pe2= pe2->suivant;
		} else
			switch(betaLong) {
				case(EGALE): continuer= 0;
				case(PREMIERE):  pe1= pe1->suivant;
				case(SECONDE):  pe2= pe2->suivant;
			}
	}
	switch(betaLong) {
```

```
		case(EGALE):
			if (continuer == 1)

			  { assert( (pe1==pd1->fin)&&(pe2==pd2->fin));
			    /* egalite */
			    return 0;}
		      else return -2;

		case(PREMIERE):
			if (pe2==pd2->fin)
				return 1;
			else
				return -3;
		case(SECONDE):
			if (pe1==pd1->fin)
				return -1;
			else
				return -4;
	}
}
```

Lorque deux beta's sont incomparables il faut le signaler. Le fait de placer des assert's arrête l'exécution du programme. La procédure renvoit un négatif quand deux beta's sont incomparables. Il faut donc modifier la procédure Adjust qui utilise la procédure comparebeta avec un test negatif. Il suffit de le transformer par un test positif en inversant l'ordre des arguments.

Modification 5: La Procédure Adjust. (primitives.c)

Version Originale:

```
ptrListeTuple3 ADJUST(beta,  p, betaPrime, sat)
ptrBeta beta;
ptrProc p;
ptrBeta betaPrime;
ptrListeTuple3 sat;


{/*     pre:
        post:  modificateur de sat
               on remplace le beta2 des tuples de même domaine
               par le max de betaPrime et leur valeur existante
*/
        ptrTuple3 pt;
        ptrElement pe;
int i;

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

Version Corrigée :

```
ptrListeTuple3 ADJUST(beta,p,betaPrime,sat)
```

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

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