USTL

LIFL Laboratoire d'Informatique Fondamentale de Lille

Numéro d'ordre : 1248

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Christophe Lecoutre

INTERPRETATION ABSTRAITE EN PROGRAMMATION LOGIQUE AVEC CONTRAINTES

Thèse soutenue le 4 Février 1994, devant le jury composé de :

Patrick COUSOT	Président
Gilberto FILE	Rapporteur
Baudouin LE CHARLIER	Rapporteur
Philippe DEVIENNE	Examinateur
Jean-Paul DELAHAYE	Examinateur
Gérard FERRAND	Examinateur
Laurent FRIBOURG	Examinateur
Rémi GILLERON	Examinateur
Patrick LEBEGUE	Examinateur

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.47.24 Fax. 20.43.65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Remerciements

Je tiens à remercier :

◦ Patrick Cousot pour l'honneur qu'il me fait de présider cette thèse malgrè un emploi du temps délicat.

◦ Gilberto Filè et Baudouin Le Charlier qui ont accepté de rapporter cette thèse. Leur gentillesse et leurs remarques pertinentes ont été un encouragement.

◦ Gérard Ferrand, Laurent Fribourg et Rémi Gilleron pour avoir accepté de participer au jury.

◦ Jean-Paul Delahaye pour m'avoir accueilli dans son équipe.

◦ Philippe Devienne et Patrick Lebègue pour leur encadrement, leur soutien, leurs conseils et leur amitié. Sans eux, ce travail n'aurait pu voir le jour.

◦ les collègues du labo pour la bonne humeur générale.

◦ les gens sur lesquels je sais pouvoir compter dans les moments faciles comme dans les moments difficiles.

Merci aussi à ...

Table des matières

Partie I
La logique en Programmation

Chapitre 1 : Programmation logique

Chapitre 2 : Programmation logique avec contraintes

Partie II
Interprétation Abstraite

Chapitre 3 : Aspects Théoriques

Chapitre 4 : Application à la Programmation Logique

Partie III
Un modèle générique d'interprétation abstraite

Chapitre 5 : Extension du domaine

Chapitre 6 : Abstraction du calcul

Partie IV
Une Inférence de Types

Chapitre 7 : Types et Contraintes Ensemblistes

Chapitre 8 : Inférence de Types avec CLP($\mathcal{FT}$+$\mathcal{SC}$)

Introduction

L'interprétation abstraite [Cousot et Cousot 77] est une technique d'analyse statique qui permet d'étudier le comportement dynamique d'un programme au seul vu de son code source. Cette technique est généralement utilisée pour l'optimisation du code objet d'un programme (e.g., élimination du code inutile), la transformation d'un programme (e.g., évaluation partielle), ou encore l'établissement de certaines preuves de programme (e.g., preuve d'arrêt). Si une sémantique est considérée comme un calcul sur un domaine, alors une interprétation abstraite est la donnée de deux sémantiques et d'une relation [Cousot et Cousot 92c, Marriott 93]. L'une des deux sémantiques est dite concrète (ou standard) tandis que l'autre est dite abstraite (ou non standard). La relation (dite d'approximation) est définie entre le domaine concret et le domaine abstrait. Une interprétation abstraite est consistante si et seulement si le résultat du calcul concret à partir d'une donnée concrète d est approchée par le résultat du calcul abstrait à partir d'une donnée abstraite qui approche d.

Partant de la donnée d'une sémantique concrète, la conception d'une interprétation abstraite consiste à établir successivement :

- le domaine abstrait
- la relation d'approximation entre le domaine concret et le domaine abstrait
- le calcul abstrait

Il est possible d'être plus précis, en particulier pour l'élaboration du calcul abstrait. Une bonne démarche est la suivante : d'abord, induire le calcul abstrait en imitant le calcul concret de façon opérationnelle, et ensuite, forcer (ou accélérer) la terminaison du calcul abstrait à l'aide d'opérateurs de widening, i.e., d'opérateurs qui effectuent des généralisations [Cousot et Cousot 76,77]. On peut considérer dans ce cas qu'une interprétation abstraite se compose de deux phases :

- une phase d'abstraction du domaine
- une phase d'abstraction du calcul

La phase d'abstraction du domaine revient à définir le domaine abstrait, définir la relation d'approximation et induire le calcul abstrait à partir du calcul concret. La phase d'abstraction du calcul revient à forcer la terminaison du calcul abstrait. On sépare ainsi en quelque sorte l'aspect déclaratif (la manière dont on définit le domaine abstrait) de l'aspect calculatoire (la manière dont on obtient un calcul fini). Cette démarche correspond, pour l'essentiel, à l'approche de l'interprétation abstraite basée sur la combinaison d'une connexion de Galois et d'opérateurs de widening et narrowing [Cousot et Cousot 92b].

Cahier des charges

L'intérêt de l'interprétation abstraite a été clairement démontré pour des paradigmes aussi variés que la programmation impérative, fonctionnelle et logique. Pour notre part, nous situons notre recherche dans le cadre de la programmation logique avec contraintes. En effet, nous proposons un modèle d'interprétation abstraite qui puisse être appliqué à la classe des CLP-langages [Jaffar et Lassez 86]. Cette classe de langages présente l'avantage de fournir un cadre homogène qui soit adapté à la définition de différentes sémantiques (concrètes et abstraites). Avant toute chose, il est nécessaire de formuler le cahier des charges du modèle d'interprétation abstraite que nous souhaitons définir. Ce modèle doit être :

- générique,
  i.e., utilisable pour l'ensemble des CLP-langages.

- souple,
  i.e., permettre de définir aussi bien des analyses de flux de données (inférence de mode, inférence de type, ...) que des analyses opérationnelles (étude de la terminaison, du déterminisme, ...).

- simple,
  i.e., permettre de concevoir facilement une analyse et de prouver la consistance de celle-ci.

- précis,
  i.e., permettre de concevoir des analyses dont le résultat soit précis.

Solutions retenues

A partir du cahier des charges, nous devons effectuer certains choix importants. Le premier concerne l'approche utilisée pour calculer la sémantique abstraite. Trois approches sont principalement proposées dans le domaine de l'interprétation abstraite :

- l'approche opérationnelle
i.e., une approche consistant à imiter le déroulement du calcul concret par le calcul abstrait.

- l'approche de point fixe
i.e., une approche consistant à définir la sémantique abstraite en terme de point fixe.

- l'approche dénotationnelle
i.e., une approche où la sémantique abstraite est définie par des équations sémantiques.

Dans notre cahier des charges, nous avons indiqué notre souhait de pouvoir établir des analyses opérationnelles à partir de notre modèle. Or, plus la sémantique abstraite est proche de la sémantique concrète, plus l'information opérationnelle disponible est grande. De plus, CLP se prête tout à fait naturellement à l'abstraction du calcul concret. En effet, pour obtenir une interprétation abstraite, il suffit de définir un CLP-langage abstrait apte à simuler un CLP-langage concret, et d'introduire une relation d'approximation entre les domaines des deux langages. Ceci constitue l'idée maîtresse de l'approche opérationnelle de [Codognet et Filè 92]. Les prérequis de généricité, souplesse et simplicité sont alors satisfaits. Reste le problème de la précision du résultat des analyses ... mais nous y reviendrons plus loin.

⇒ *nous considérons une approche opérationnelle.*

Ensuite, il est indispensable de se positionner par rapport à l'interprétation. Suivant [Marriott et Sondergaard 89b], les interprétations abstraites définies pour la programmation logique se rangent essentiellement dans deux classes distinctes : les interprétations top-down et les interprétations bottom-up. Une interprétation top-down tient compte d'un programme et d'un but (ou état initial) et caractérise par chaînage arrière l'ensemble des appels possibles tandis qu'une interprétation bottom-up tient compte d'un programme et de faits (ou états finaux) et caractérise par chaînage avant l'ensemble des solutions possibles. La plupart des travaux consacrés à l'interprétation abstraite en programmation logique correspondent à une interprétation top-down [Mellish 87 , Kanamori et Kawamura 90 , Bruynooghe 91 , Marriott et Sondergaard 90 , Le Charlier et al. 91] mais dans le cadre des interprétations bottom-up, on trouve notamment les travaux de [Marriott et Sondergaard 88,92]. Pour tenir compte des conditions "réelles" de l'exécution d'un programme, il est nécessaire d'utiliser une interprétation top-down.

⇒ *nous considérons une interprétation top-down.*

Il est à noter qu'un interpréteur Prolog (concret) top-down a parfois un comportement infini (là où un interpréteur bottom-up a un comportement fini), aussi est-il nécessaire d'introduire un "principe bottom-up" lors de l'élaboration d'un interpréteur abstrait top-down, c'est à dire une technique qui prend en compte ce qui a déjà été calculé. Il peut s'agir de la tabulation [Tamaki et Sato 86] ou d'un calcul de point fixe.

⇒ *nous considérons l'utilisation de la tabulation.*

Pour revenir au problème de la précision du modèle, il est important de noter la manière dont est conçue l'approximation. Les différents travaux en interprétation abstraite se rangent par rapport à :

- l'approximation par combinaison

ou

- l'approximation par abstraction,

mais aucun (à notre connaissance) ne se range explicitement par rapport à :

- l'approximation par extension.

Le modèle d'approximation par combinaison consiste à définir la sémantique abstraite à partir d'un programme concret et d'un but abstrait. A chaque étape élémentaire, il est donc nécessaire de combiner information abstraite et information concrète. Ce modèle est le plus répandu [Mellish 87 , Bruynooghe 91 , Marriott et Sondergaard 90 , Kanamori et Kawamura 90 , Le Charlier et al. 91]. Le modèle d'approximation par abstraction consiste à définir la sémantique abstraite à partir d'un programme abstrait et d'un but abstrait. [Hermenegildo et al. 92] utilisent le terme de compilation abstraite. Du fait de l'abstraction immédiate (avant le début de l'exécution) et dans le cas où cette abstraction est syntaxique, les analyses fondées sur ce modèle peuvent se révéler moins précises (pour un domaine abstrait équivalent). Ce modèle est utilisé par [Codognet et Filè 92 , Warren 92 , Codish et Daemon 93], et nous rappelons que c'est celui sur lequel nous nous basons. Toutefois, nous nous plaçons dans un cadre particulier, à savoir, le domaine abstrait doit être une extension du domaine concret. C'est pourquoi nous appelons ce modèle le modèle d'approximation par extension. L'avantage est qu'aucune abstraction n'est faite à priori, et qu'en conséquence, les analyses peuvent se révéler très précises.

⇒ *nous considérons une approximation par extension.*

Pour assurer la terminaison de l'analyse, il est nécessaire d'introduire des opérateurs de widening.

⇒ *nous considérons l'utilisation d'opérateurs de widening.*

Signalons que le choix d'un modèle d'approximation par extension va dans le sens d'une approche de l'interprétation abstraite basée sur une phase d'abstraction (ou plutôt extension) du domaine suivie d'une phase d'abstraction du calcul.

Un modèle générique d'interprétation abstraite

Etant donné un CLP-langage concret, la phase d'extension du domaine consiste à définir un nouveau CLP-langage (abstrait) via l'adjonction de nouvelles contraintes au domaine concret. La relation d'approximation se résume alors à l'implication logique définie sur le domaine concret et étendue au domaine abstrait. De cette manière, nous arrivons à prouver en particulier la monotonie de la résolution SLD pour tout CLP-langage. Par exemple, avec CLP(IR) et CLP(IR+IB) il est possible de construire une analyse de l'assignation unique ("definiteness analysis" ou analyse des variables qui sont contraintes pendant la résolution à une assignation unique) en adaptant les résultats de [Marriott et Sondergaard 89a] et [Cortesi et al. 91]. Pour le CLP-langage abstrait obtenu, le problème de l'arrêt de la résolution reste entier. La tabulation est un moyen d'éviter certaines phases de calcul redondantes. L'intégration de cette technique à la résolution OLD donne la résolution OLDT et a été formalisée par [Tamaki et Sato 86] pour Prolog. Nous la généralisons pour tout CLP-langage. Dans certains cas (i.e, pour certains CLP-langages), la tabulation n'est pas suffisante pour assurer la terminaison du calcul. Il est alors nécessaire d'abstraire le résolution OLDT : la résolution OLDT abstraite est appelée résolution AOLDT. Celle-ci est composée de deux étapes. La première étape appelée étape de widening a pour objectif de fournir une approximation finie de la résolution OLDT et la seconde étape appelée étape de narrowing celui de corriger (améliorer) sensiblement cette approximation. Nous montrons que la résolution AOLDT est finie et qu'elle est complète vis à vis de la résolution OLD par rapport aux appels et solutions atomiques apparaissant lors de la résolution.

Un système d'inférence de types

Pour illustrer le modèle d'interprétation abstraite générique proposé ci-dessus nous proposons un système d'inférence de types pour Prolog. Prolog est un CLP-langage que l'on désigne alors par CLP($\mathcal{FT}$). Nous commençons par étendre le domaine de CLP($\mathcal{FT}$) en y intégrant des contraintes ensemblistes. Le langage obtenu par cette intégration est noté CLP($\mathcal{FT}$+$\mathcal{SC}$)[†]. Les contraintes de ce nouveau langage portent à la fois sur les termes (ou arbres finis) et sur les ensembles. L'intérêt de cette combinaison est que les contraintes sur les termes

---

[†] $\mathcal{FT}$ est mis pour "finite trees" et $\mathcal{SC}$ est mis pour "set constraints"

permettent de coder les dépendances entre les variables et que les contraintes ensemblistes permettent de coder les structures récursives et non déterministes [Heintze 92]. Les contraintes de CLP($\mathcal{FT}$+$\mathcal{SC}$) sont de la forme $c = (\mathcal{ST},\mathcal{SS})$ où $\mathcal{ST}$ représente un système d'équations sur les termes et $\mathcal{SS}$ représente un système d'équations ensemblistes. Nous montrons que lorsque de telles contraintes sont placées sous forme résolue, la satisfiabilité et l'implication sont deux propriétés décidables. Ces décisions sont essentielles à l'utilisation de CLP($\mathcal{FT}$+$\mathcal{SC}$) comme base de calcul abstrait. De fait, en utilisant la tabulation et en introduisant des opérateurs de widening adaptés, nous obtenons un algorithme d'inférence de types non trivial. [Heintze et Jaffar 92] concilient également contraintes ensemblistes et information sur la dépendance entre variables mais leur approche est dénotationnelle.

Plan de la thèse

Partie I

La première partie est consacrée à l'utilisation de la logique en programmation. Le chapitre 1 traite exclusivement du langage Prolog et le chapitre 2 des CLP-langages, i.e., des langages de programmation logique avec contraintes. Il s'agit en quelque sorte d'une approche "historique".

Partie II

La seconde partie concerne l'interprétation abstraite. Dans le chapitre 3 sont présentés essentiellement les aspects théoriques de l'interprétation abstraite et dans le chapitre 4 les différents modèles d'interprétation abstraite proposés pour la programmation logique.

Partie III

La troisième partie présente notre modèle d'interprétation abstraite générique. Celui-ci consiste en une phase d'extension du domaine, décrite au chapitre 5, suivie d'une phase d'abstraction du calcul, décrite au chapitre 6.

Partie IV

La quatrième partie est une illustration de la puissance du modèle proposé. Le chapitre 7 présente les contraintes ensemblistes et leur adéquation au typage. Le chapitre 8 décrit un système d'inférence de types basée sur l'utilisation de contraintes ensemblistes.

Avertissement au lecteur

Les deux premières parties présentent les définitions et résultats essentiels en programmation logique (avec contraintes) et en interprétation abstraite. Dans cette présentation, nous fixons de nombreuses notations. Malgré cela, le lecteur qui est coutumier des domaines évoqués peut survoler ces deux parties quitte à utiliser, en cas de besoin, l'index et le rappel des notations placés en fin de document. La troisième partie correspond à un travail de recherche personnel portant sur la conception d'un modèle générique d'interprétation abstraite appliqué à la programmation logique avec contraintes. Pour terminer avec la quatrième partie, le chapitre 7 représente une sorte d'état de l'art portant sur les contraintes ensemblistes et le typage en programmation logique et le chapitre 8 correspond à un travail de recherche personnel portant sur la conception d'un système d'inférence de types.

Partie I

La Logique en Programmation

L'histoire des relations entre la logique et le calcul est issue d'une longue recherche sur la démonstration automatique de théorèmes commencées dans les années 30 avec les travaux de Herbrand. Au début des années 70, à partir des travaux de [Robinson 65] sur la résolution, [Kowalski 74] montra qu'un ensemble de formules logiques pouvait avoir une interprétation procédurale et [Colmerauer et al. 73] créèrent le langage Prolog. Cette épopée est retracée notamment par [Cohen 88], [Kowalski 88] et [Robinson 93]. Puis, au milieu des années 80, [Colmerauer 82] proposa une extension de Prolog, appelée Prolog II, et [Jaffar et Lassez 86] introduisirent un modèle générique de programmation logique avec contraintes, appelé CLP, dont Prolog et Prolog II étaient des instances. La section 1 est consacrée à la programmation logique (Prolog) et la section 2 est consacrée à la programmation logique avec contraintes (CLP). Dans cette thèse, nous ne débattons ni des langages concurrents [Saraswat et Rinhard 91], ni des langages basés sur une logique d'ordre supérieure à 1 [Miller et Nadathur 86].

Chapitre 1

Programmation Logique

1. Introduction

Cette partie est consacrée au rappel des concepts et définitions élémentaires de la programmation logique (Prolog). Pour la rédiger, nous nous sommes inspirés essentiellement des travaux de [Lloyd 85] et de [Apt 90]. Des résultats classiques sont énoncés sans démonstration, le lecteur peut les retrouver dans [Huet 76], [Lloyd 85], [Delahaye 86], [Lassez et al. 87] et [Apt 90].

Les aspects syntaxiques et sémantiques de la programmation logique sont successivement étudiés. La syntaxe rend compte de la manière dont sont construits les programmes logiques tandis que la sémantique rend compte de la signification qui leur est donnée. La classe des programmes logiques à laquelle nous nous intéressons est la classe des programmes définis. Un programme défini est une conjonction de formules appelées clauses définies de la forme $h \leftarrow b_1 \wedge \ldots \wedge b_n$. Le sens naturel (logique) de chaque clause définie est l'implication logique. $h \leftarrow b \wedge \ldots \wedge b_n$ est naturellement interprétée comme : si la condition $b_1 \wedge \ldots \wedge b_n$ est vrai alors la conséquence h est vrai. Cette interprétation est dite logique (ou déclarative). Il est possible d'associer à chaque programme l'ensemble de ses conséquences logiques (les clauses inconditionnelles sont des conséquences logiques triviales). Cet ensemble représente la sémantique logique d'un programme.

Il existe un moyen de calculer cette sémantique mais ce calcul s'avère en pratique souvent infini. Aussi, au lieu de calculer l'ensemble des conséquences logiques d'un programme, va-t-on simplement chercher à démontrer qu'une formule atomique est une conséquence logique d'un programme. Une clause $h \leftarrow b_1 \wedge \ldots \wedge b_n$ est alors interprétée de façon procédurale : prouver h revient à prouver b_1,

puis ..., puis prouver b_n. Pour chercher à prouver qu'une formule est conséquence logique d'un programme, on utilise la négation de cette formule comme but. On cherche alors à prouver ce but. Si la clause vide est obtenue, on déduit une contradiction du but, et par conséquent on déduit que la formule initiale est une conséquence logique du programme. Dans un cadre particulier (celui de l'espace de Herbrand), on parle de succès plutôt que de conséquence logique.

Le mécanisme général de cette interprétation procédurale est appelé résolution SLD. A tout programme, on peut associer l'ensemble des succès déduits par la résolution SLD. Cet ensemble représente la sémantique de l'ensemble des succès SLD d'un programme. Il est à noter que la sémantique logique exprime ce qu'un programme doit théoriquement pouvoir déduire tandis que la sémantique de l'ensemble des succès SLD exprime ce qu'un programme permet de déduire effectivement via la résolution SLD. Sous certaines conditions, ces deux sémantiques sont équivalentes.

2. Syntaxe

Un programme logique est un ensemble de formules particulières tirées d'un langage du premier ordre. Un langage du premier ordre est un ensemble de formules construites à partir d'un alphabet du premier ordre. Un alphabet du premier ordre est constitué de sept catégories de symboles : les symboles de variable, les symboles de fonction, les symboles de prédicats, les constantes propositionnelles, les connecteurs, les quantificateurs et les symboles de ponctuation.

- $\mathcal{V}$ représente un ensemble infini dénombrable de symboles de variable.
- $\mathcal{F}$ représente un ensemble fini de symboles de fonction.
- $\mathcal{P}$ représente un ensemble fini de symboles de prédicat.
- Les constantes propositionnelles sont : vrai et faux.
- Les cinq connecteurs sont la conjonction ($\wedge$), la disjonction ($\vee$), l'implication ($\rightarrow$), l'équivalence ($\leftrightarrow$) et la négation ($\neg$).
- Les deux quantificateurs sont le quantificateur existentiel ($\exists$) et le quantificateur universel ($\forall$).
- Les symboles de ponctuation utilisées sont "(", ")" et ",".

Chaque alphabet du premier ordre est entièrement déterminé par la donnée de $\mathcal{F}$ et $\mathcal{P}$ car les autres catégories de symboles sont invariantes. Un langage du premier ordre est l'ensemble de toutes les formules bien formées construites à partir des symboles de l'alphabet. $\mathcal{T}erm(\mathcal{V},\mathcal{F})$ désigne l'ensemble des termes (libres) et $\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P})$ désigne l'ensemble des atomes (libres), ensembles définis à l'annexe A.

$\mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}))$ représente l'ensemble des formules bien formées construites à partir de l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P})$ des formules atomiques (atomes). Une formule bien formée est définie inductivement comme suit :

Définition 1.1

- $\forall\ a \in \mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}) : a \in \mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}))$.
- $\forall\ f \in \mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P})) : \neg f \in \mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}))$.
- $\forall\ (f,g) \in \mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}))^2 : f \rightarrow g \in \mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}))$
- $\forall\ f \in \mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}))$, $\forall\ X \in \mathcal{V} : \forall X\ f \in \mathcal{F}orm(\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P}))$.

Il est à noter que les symboles '$\exists$', '$\wedge$', '$\vee$' et '$\leftrightarrow$' sont souvent utilisés comme raccourcis d'écriture :

- $\exists X\ f$ représente $\neg\ (\forall X\ \neg f)$
- $f \vee g$ représente $\neg f \rightarrow g$
- $f \wedge g$ représente $\neg\ (f \rightarrow \neg g)$
- $f \leftrightarrow g$ représente $(f \rightarrow g) \wedge (g \rightarrow f)$

Soit f une formule, $\mathrm{Var}(f)$ désigne l'ensemble des variables apparaissant dans f. Si une variable X de f est précédée d'un quantificateur, alors X est liée, sinon X est libre[†]. $\mathrm{Var}_{\mathrm{bound}}(f)$ et $\mathrm{Var}_{\mathrm{free}}(f)$ désignent respectivement l'ensemble des variables liées et l'ensemble des variables libres de f. On a : $\mathrm{Var}(f) = \mathrm{Var}_{\mathrm{bound}}(f) \cup \mathrm{Var}_{\mathrm{free}}(f)$. On peut aussi distinguer les variables liées existentiellement et universellement. On note alors : $\mathrm{Var}_{\mathrm{bound}}(f) = \mathrm{Var}_{\exists}(f) \cup \mathrm{Var}_{\forall}(f)$.

La clôture existentielle (resp. universelle) d'une formule consiste à quantifier existentiellement (resp. universellement) chaque variable libre de cette formule. Notons auparavant que pour une formule f et un ensemble de variables $V = \{X_1,\ldots,X_n\}$, $\forall X_1 \ldots \forall X_n\ f$ peut se coder par $\forall V\ f$ et $\exists X_1 \ldots \exists X_n\ f$ par $\exists V\ f$.

[†] on supposera, sans perte de généralité, qu'un symbole de variable ne peut être à la fois libre et lié dans une formule, ni même à la fois lié existentiellement et universellement.

Une clause est une formule particulièrement importante en théorie du premier ordre car toute formule peut s'écrire sous la forme d'un ensemble (une conjonction) de clauses. Un littéral est un atome ou la négation d'un atome. Un littéral positif est un atome. Un littéral négatif est la négation d'un atome.

Définition 1.2

Une clause est une formule de la forme $\forall V\ (L_1 \vee L_2 \vee ... \vee L_n)$ où chaque L_i est un littéral et V l'ensemble des variables apparaissant dans la formule.

La clause vide est une clause sans littéraux. Nous ne nous intéressons qu'à une sous-catégorie de l'ensemble des clauses : les clauses de Horn, i.e., les clauses comportant au plus un littéral positif. Cette restriction aux clauses de Horn entraîne une perte de la puissance d'expression (aspect déclaratif) mais pas de la puissance de calcul (aspect calculatoire).

Définition 1.3

Une clause définie est une clause ne comportant qu'un seul littéral positif, c'est à dire une clause (de Horn) de la forme $h \vee \neg b_1 \vee ... \vee \neg b_n$ où h, b_1, ... b_n sont des atomes.

Une clause définie se note généralement : $h \leftarrow b_1,...,b_n$.

h est appelé la tête de la clause et $b_1,...,b_n$ est appelé le corps de la clause. Si le corps de la clause est vide, alors la clause est appelée un fait.

Définition 1.4

Un programme défini est un ensemble (une conjonction) de clauses définies.

On considère par la suite que les clauses d'un programme défini sont numérotées.

Définition 1.5

Un but défini est une clause ne comportant aucun littéral positif, c'est à dire une clause (de Horn) de la forme $\neg b_1 \vee ... \vee \neg b_n$ où b_1, ... b_n sont des atomes.

Un but défini se note généralement : $\leftarrow b_1,...,b_n$.

Pour la suite, on ne considère que des clauses, programmes et buts définis.

3. Sémantique

Nous commençons par définir la sémantique d'un langage du premier ordre (la programmation logique étant basé sur cette théorie). La sémantique d'un langage du premier ordre associe une interprétation mathématique à chaque symbole de l'alphabet du premier ordre.

L'interprétation des symboles de fonction et des symboles de prédicat est donnée par une Σ-algèbre (homogène) $\mathcal{A}$ avec $\Sigma = (S, \mathcal{F}, \mathcal{P})$ et S = {term}. Etant donnée une Σ-algèbre $\mathcal{A}$ (voir l'annexe A) de domaine $\mathcal{D} = \mathcal{A}(\text{term})$, toute formule close (sans variables libres) de l'ensemble $\mathcal{F}orm(\mathcal{A}tom(\mathcal{V}, \mathcal{F}, \mathcal{P}))$ obtient une valeur de vérité suivant les règles suivantes :

- si la formule est atomique, sa valeur de vérité est donnée par $\mathcal{A}$
- si la formule a la forme $\neg f$ ou $f \rightarrow g$, sa valeur de vérité est donnée par la table 1.6
- si la formule a la forme $\forall X\, f$, sa valeur de vérité est vraie ssi pour tout élément d de $\mathcal{D}$, la valeur de vérité de la formule obtenue à partir de f en remplaçant X par d est vraie.

f	g	$\neg f$	$f \rightarrow g$
faux	faux	vrai	vrai
faux	vrai	vrai	vrai
vrai	faux	faux	faux
vrai	vrai	faux	vrai

Table 1.6

On pose pour la suite une Σ-algèbre $\mathcal{A}$ (généralement appelée interprétation dans la littérature concernant la programmation logique) de domaine $\mathcal{D}$.

Définition 1.7

$\mathcal{V}al$ désigne l'ensemble des applications de $\mathcal{V}$ vers $\mathcal{D}$. Tout élément θ de $\mathcal{V}al$ est appelée une $\mathcal{A}$-assignation de variables.

Soit f une formule (de l'ensemble $\mathcal{F}orm(\mathcal{A}tom(\mathcal{V}, \mathcal{F}, \mathcal{P}))$) et θ une $\mathcal{A}$-assignation de variables, $\theta(f)$ désigne la formule f où chaque variable libre X de f a été remplacée par θ(X). Si $\theta(f)$ est évaluée à vrai par rapport à $\mathcal{A}$, alors θ est dit $\mathcal{A}$-solution de f. $\mathcal{A}$ est un modèle de f ssi toute $\mathcal{A}$-assignation de variables θ est une $\mathcal{A}$-solution de f. Si f possède un modèle, f est dit satisfiable ou consistant. Si f ne possède aucun modèle, f est dit insatisfiable ou inconsistant. Un ensemble de formules $S = \{f_1, \ldots, f_n\}$ est interprétée dans ce qui suit comme la formule $f_1 \wedge \ldots \wedge f_n$. Une formule f est une conséquence logique d'un ensemble de formules S ssi tout modèle de S est aussi un modèle de f ssi $S \cup \{\neg f\}$ est insatisfiable.

La sémantique logique d'un programme P, notée $\overline{P}$, est définie comme étant l'ensemble des conséquences logiques (atomiques et closes) de P.

Définition 1.8

Soit P un programme,

$\dot{P}$ = $\{a \in \mathcal{A}tom(\mathcal{F},\mathcal{P}) : a$ est une conséquence logique de P$\}$.

En appliquant la propriété précédente à la programmation logique, on remarque que si P est un programme (défini) et que a est un atome, alors prouver que a est une conséquence logique de P revient à prouver que P $\cup \{\neg a\}$ est insatisfiable sachant que $\neg a$ représente un but (défini). Il faut donc montrer qu'il n'existe aucun modèle de P $\cup \{\neg a\}$. Cependant, il s'avère qu'il suffit de considérer une catégorie d'interprétations (algèbres) beaucoup plus petite pour montrer l'insatisfiabilité d'un ensemble de clauses : la catégorie des interprétations de Herbrand.

Soit P un programme, U_P et B_P désignent respectivement l'univers et la base de Herbrand de P, c'est à dire l'ensemble des termes et des atomes clos construits à partir des symboles de fonction et de prédicat (pour B_P) apparaissant dans P. Une interprétation de Herbrand est un sous-ensemble de B_P dont les éléments sont interprétés à vrai (par défaut les autres éléments sont interprétés à faux).

Propriété 1.9

Soit S un ensemble de clauses. Si S a un modèle alors S a un modèle de Herbrand et S est insatisfiable ssi S n'a pas de modèle de Herbrand.

Toute intersection d'un ensemble de modèles de Herbrand pour un programme P est encore un modèle de Herbrand. On sait donc qu'il existe un plus petit modèle de Herbrand (par intersection de tous les modèles de Herbrand). Ce plus petit modèle est noté : M_P.

La sémantique logique d'un programme P est égale à son plus petit modèle de Herbrand. Ce résultat est du à [van Emden et Kowalski 76]. De plus, il existe un moyen de calculer cette sémantique.

Définition 1.10

Soit P un programme, l'application $T_P \in \wp(B_P) \rightarrow \wp(B_P)$ est définie par :

$T_P(I) = \{\theta(h) \in B_P$ tel que

- $h \leftarrow b_1, ..., b_n$ est une clause de P
- θ est une assignation de variables définie de $\mathcal{V}$ vers U_P
- $\{\theta(b_1),...,\theta(b_n)\} \subseteq I$ $\}$

Propriété 1.11

T_P est un opérateur continu.

Le plus petit point fixe de l'opérateur T_P, noté lfp T_P, est calculé en au plus ω étapes et est égal à M_P. On résume l'ensemble de ces équivalences par le théorème suivant :

Théorème 1.12 [van Emden et Kowalski 76]
Si P est un programme défini alors :
$\llbracket P \rrbracket = M_P = \text{lfp } T_P = T_P^{\omega}(\emptyset)$.

Ce théorème indique donc qu'il est possible de calculer la sémantique logique d'un programme. Malheureusement, ce calcul est la plupart du temps infini. Par conséquent, en pratique, il n'est pas possible d'utiliser directement ce théorème.

Au lieu de calculer l'ensemble des conséquences logiques d'un programme, on va simplement se limiter à prouver qu'un atome *a* est ou n'est pas conséquence logique d'un programme. Pour ce faire, on considère la négation de *a* (qui est un but) et en utilisant une règle d'inférence appelée règle de résolution, on tente de déduire la clause vide (ce qui représente une contradiction). Dans ce cas, *a* est une conséquence logique du programme.

En fait, on ne s'intéresse pas uniquement au but lui-même, mais à toutes les instances de ce but. La résolution est utilisée pour chercher l'ensemble des instances du but qui sont conséquences logiques du programme. La substitution et l'unification en sont les éléments fondamentaux.

4. Algèbre $\mathcal{T}erm(\mathcal{V},\mathcal{F})$

Dans cette partie, nous introduisons les opérations classiques définies sur l'algèbre $\mathcal{T}erm(\mathcal{V},\mathcal{F})$, en particulier les opérations nécessaires à la description de la règle de résolution. Ces opérations se généralisent sans problèmes pour $\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P})$.

4.1. Substitution

Définition 1.13
Une substitution est une application définie de $\mathcal{V}$ sur $\mathcal{T}erm(\mathcal{V},\mathcal{F})$.

Deux ensembles de variables sont donnés par rapport à toute substitution σ, le domaine noté $Dom(\sigma)$ et le codomaine noté $Ran(\sigma)$:
- $Dom(\sigma) = \{X \in \mathcal{V} : X \neq \sigma(X)\}$.
- $Ran(\sigma) = \cup \{Var(\sigma(X)) : X \in Dom(\sigma)\}$.

Une substitution σ est entièrement définie par la donnée de $Dom(\sigma)$ et des images de $Dom(\sigma)$. Toutes les substitutions que nous allons manipuler ont un ensemble $Dom(\sigma)$ fini, aussi utilisera-t-on généralement la notation suivante : $\sigma = \{X_1=\sigma(X_1), ..., X_n=\sigma(X_n)\}$ où $Dom(\sigma) = \{X_1,...,X_n\}$.

Définition 1.14

Soient une substitution σ et un ensemble de variables V, la projection (ou la restriction) de σ sur V est la substitution $\sigma\downarrow V$ telle que

- $Dom(\sigma\downarrow V) = Dom(\sigma) \cap V$
- $\forall X \in Dom(\sigma\downarrow V), \sigma\downarrow V(X) = \sigma(X)$.

Par exemple, si σ = {W=b, X=s(a), Y=Z} et V = {Y} alors $\sigma\downarrow V$ = {Y/Z}.

Définition 1.15

Soient un terme t et une substitution σ, $\sigma(t)$ désigne l'instance de t par σ, c'est-à-dire le terme obtenu en remplaçant en parallèle chaque variable X de Var(t) par σ(X).

L'application d'une substitution sur un terme induit un pré-ordre sur l'ensemble des termes. Soient t et s deux termes, $t \leq s$ (t est moins général que s) ssi il existe une substitution σ telle que $t = \sigma(s)$, et $t \approx s$ (t est aussi général que s) ssi $t \leq s$ et $s \leq t$. Par exemple, si t = f(a,g(b,W)) et s = f(X,g(Y,Z)) alors $t \leq s$ car $t = \sigma(s)$ avec σ = {X=a, Y=b, Z=W}.

Définition 1.16

$\mathcal{R}en$ désigne l'ensemble des bijections de $\mathcal{V}$ sur $\mathcal{V}$. Tout élément ρ de $\mathcal{R}en$ est appelée une substitution de renommage (ou permutation).

Notons que deux termes t et s sont aussi généraux l'un que l'autre ssi il existe une substitution de renommage ρ telle que $\rho(t) = s$. Dans ce cas, t et s sont appelées des variantes. Par exemple, f(X,Y) et f(Y,X) sont des variantes mais f(X,X) et f(Y,X) ne sont pas des variantes.

La composition o de substitutions est définie de manière classique. Soient σ et σ' deux substitutions, pour toute variable X, $\sigma o\sigma'(X) = \sigma(\sigma'(X))$. On sait que l'application $I\delta$ est une substitution et que o est une loi de composition interne. Un sous-ensemble de substitutions est souvent considéré pour ses bonnes propriétés : les substitutions idempotentes.

Définition 1.17

$\forall \sigma \in \mathcal{S}ubs$, σ est dite idempotente ssi $\sigma o\sigma = \sigma$.

Pour finir, le pré-ordre introduit sur les termes est étendu aux substitutions. Soient σ et σ' deux substitutions, $\sigma \leq \sigma'$ (σ est moins générale que σ') ssi il existe une substitution β telle que $\sigma = \beta o\sigma'$, et $\sigma \approx \sigma'$ (σ est aussi générale que σ') ssi $\sigma \leq \sigma'$ et $\sigma' \leq \sigma$. Par exemple, si σ = {W=b, X=s(a), Y=Z} et σ' = {X=s(X'), Y=Z} alors $\sigma \leq \sigma'$ car $\sigma = \beta o\sigma'$ avec β = {W=b, X'=a}.

4.2. Unification

L'unification de deux termes t et s consiste à calculer un terme r appelé plus grand unifié. Un terme r est un unifié de deux termes t et s ssi il existe une substitution σ appelée unificateur telle que $r = \sigma(t) = \sigma(s)$. L'unification de deux termes n'est pas toujours possible, par exemple f(a) et f(b) ne sont pas unifiables. Un plus grand unifié est unique au nom des variables près. En pratique, calculer t nécessite de calculer σ. La substitution σ calculée est alors un plus grand unificateur de t et s. Si on se restreint à l'ensemble des substitutions idempotentes, elle est unique au nom des variables près.

Les notions d'unifié et de minorant ne sont pas équivalentes. Cependant lorsque les termes t et s ne possèdent aucune variable en commun, elles coïncident. Ceci est le cas qui nous intéresse car lors de la résolution, chaque fois qu'une clause est considérée, une variante de cette clause est générée. Dans ce contexte, le calcul d'un plus grand unifié est donc égal au calcul d'un plus grand minorant. A ce propos, [Eder 85] parle d'unification faible.

Définition 1.18

$\forall (t,s) \in \mathcal{T}erm(\mathcal{V},\mathcal{F})^2$. Une substitution σ est appelée un unificateur de t et s ssi $\sigma(t) = \sigma(s)$. Un unificateur σ de t et s est appelé un plus grand unificateur de t et s ssi pour chaque unificateur β de t et s, β est moins général que σ.

Deux termes t et s sont dits unifiables ssi il existe un unificateur de t et s. Par exemple, si t = h(f(X),g(Y),Z) et s = h(Y,g(f(a),Z) alors t et s sont unifiables. β = {X=a, Y=f(a), Z=a} est un unificateur de t et s et σ = {X=a, Y=f(a)} est un plus grand unificateur de t et s.

Le théorème suivant, dû à [Robinson 65], indique en outre qu'il est possible de calculer un plus grand unificateur.

Théorème 1.19 [Robinson 65]

Il existe un algorithme (appelé algorithme d'unification) qui pour deux termes t et s retourne (en un temps fini)

- un plus grand unificateur de t et s si t et s sont unifiables
- un échec sinon.

Nous présentons maintenant un algorithme d'unification. Cet algorithme est basé sur l'algorithme d'unification original de Herbrand et fut tout d'abord présenté par [Martelli et Montanari 82]. Le principe de l'algorithme est de résoudre un système composé d'un ensemble fini d'équations de termes. Unifier t et s équivaut à réduire le système d'équations Equ = $\{t = s\}$. Réduire le système d'équations Equ consiste à appliquer tant que cela est possible l'une des transformations suivantes sur l'une quelconque des équations de Equ :

$f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)$	$\Rightarrow$ remplacer cette équation par $t_1 = s_1, \ldots, t_n = s_n$
$f(t_1,\ldots,t_n) = g(s_1,\ldots,s_n)$	$\Rightarrow$ arrêt avec échec
$X = t$ $(X \notin Var(t))$	$\Rightarrow$ remplacer X par t dans toute autre équation
$X = t$ $(X \in Var(t))$	$\Rightarrow$ arrêt avec échec
$t = X$	$\Rightarrow$ remplacer cette équation par $X = t$
$X = X$	$\Rightarrow$ supprimer cette équation

Algorithme 1.20 d'unification

Cet algorithme d'unification se termine toujours. Un échec indique que les deux termes ne sont pas unifiables et un succès indique qu'ils le sont. Le système réduit d'équations représente le plus grand unificateur de t et s. Celui-ci est idempotent. Ce résultat va au delà du "simple" problème d'unification puisque il souligne que tout système d'équations possède une forme normale qui est celle d'une substitution idempotente. Dorénavant, nous ne considérerons plus que de telles substitutions. L'algorithme d'unification s'étend sans problèmes aux substitutions.

4.3. Anti-unification

L'anti-unification de deux termes t et s consiste à calculer un plus petit majorant r (parfois appelé anti-unifié) et noté $lub(t,s)$. L'anti-unification de deux termes est toujours possible. Un plus petit majorant est unique au nom des variables prés. Il est à noter que cela ne serait plus le cas si on considérait l'ensemble des termes rationnels au lieu de l'ensemble des termes finis.

Propriété 1.21

L'ensemble (au nom des variables prés) des majorants d'un terme est fini.

Théorème 1.22

Il existe un algorithme (appelé algorithme d'anti-unification) qui pour deux termes t et s retourne (en un temps fini) un plus petit majorant de t et s.

Nous présentons maintenant l'algorithme d'anti-unification. Cet algorithme a été introduit pour la première fois par Plotkin et Reynolds en 1970. Nous donnons ici la version de [Huet 76] que l'on peut retrouver dans [Lassez et al. 87].

Une bijection θ de $\mathcal{T}erm(\mathcal{V},\mathcal{F})^2$ vers $\mathcal{V}$ est donnée. Anti-unifier t et s revient à appliquer ψ sur le couple (t,s) où ψ est une application de $\mathcal{T}erm(\mathcal{V},\mathcal{F})^2$ vers $\mathcal{T}erm(\mathcal{V},\mathcal{F})$ définie comme suit :

$\psi(t,s) = f(\psi(t_1,s_1),\ldots,\psi(t_n,s_n))$ si $t = f(t_1,\ldots,t_n)$ et $s = f(s_1,\ldots,s_n)$
$\psi(t,s) = \theta(t,s)$ sinon

Algorithme 1.23 d'anti-unification

Par exemple, si t = h(f(X),g(Y),Z) et s = h(Y,g(f(a),Z) alors h(U,V,W) est un majorant (anti-unifié) de t et s et h(U,g(V),Z) est un plus petit majorant (anti-unifié) de t et s. L'algorithme d'anti-unification s'étend sans problèmes aux substitutions.

5. Résolution SLD

5.1. Description

La résolution SLD est fondée sur le principe de résolution de [Robinson 65] dont une stratégie d'utilisation particulière est la SL résolution, ce qui pour les clauses Définies donne la SLD résolution. La SL résolution désigne une résolution Linéaire (à chaque étape de dérivation, on utilise le résultat de l'étape précédente) avec une fonction de Sélection qui indique l'atome à dériver à chaque étape. Nous commençons par définir les notions de résolvant et d'étape de dérivation SLD.

Définition 1.24

Soient P un programme et G : $\leftarrow a_1,\ldots,a_n$ un but. Etant sélectionnés un atome a_i de G et une clause C_j de P, considérons une variante $h \leftarrow b_1,\ldots,b_m$ de C_j dont les variables sont nouvelles. Si a_i et h sont unifiables alors le but H : $\leftarrow \sigma(a_1,\ldots,a_{i-1},b_1,\ldots,b_m,a_{i+1},\ldots,a_n)$ est appelé le (i,j)-résolvant de (P,G) sachant que σ représente un plus grand unificateur de a_i et de h.

Le (i,j)-résolvant de (P,G) est unique au nom des variables près. Une étape de dérivation SLD consiste à calculer un résolvant. Un arbre SLD est un objet qui est défini à partir d'un couple (P,G) constitué d'un programme P et d'un but G et qui mémorise une séquence d'étapes de dérivation SLD. Chaque noeud de cet arbre est étiqueté par un but (et un entier i dont nous ne tenons pas compte ici) et chaque arc de cet arbre est étiqueté par une substitution et un entier j. Un noeud étiqueté par la clause vide est appelé un noeud vide.

Un arbre SLD est dépendant d'une fonction de sélection, c'est à dire une fonction qui pour chaque noeud de l'arbre sélectionne un atome de l'étiquette de ce noeud. Etant donnée une fonction de sélection fs, un arbre SLD T est obtenu à partir d'un couple (P,G) comme suit :

- la racine de T est étiquetée par G

- si v est un noeud (non vide) de T étiqueté par H : $\leftarrow a_1,\ldots,a_n$ et si a_i est l'atome sélectionné par fs, alors pour toute clause C_j de P, on considère une variante $h \leftarrow b_1, \ldots, b_m$ de C_j (dont les variables sont nouvelles). Si a_i et h sont unifiables alors on ajoute un noeud fils w à v tel que w est étiqueté par le (i,j)-résolvant de (P,H) et on ajoute un arc reliant v à w étiqueté par (σ,j) où σ est un plus grand unificateur de a_i et de h. Les différents noeuds sont ordonnés suivant l'ordre des clauses utilisées pour les obtenir.

Une interprétation SLD désigne la construction d'un arbre SLD, et de ce fait est spécifié par une méthode de résolution (méthode d'interprétation), c'est à dire par :

- une fonction de sélection fs qui détermine le choix de l'arbre SLD à construire.

- une stratégie de parcours sp qui détermine l'ordre dans lequel sont considérés les noeuds de l'arbre SLD.

Un interpréteur SLD est un programme qui effectue des interprétations SLD et qui est caractérisé par une méthode de résolution. En Prolog, on considère un petit nombre de méthodes de résolution. La fonction de sélection considérée est la fonction qui associe à chaque étiquette du noeud d'un l'arbre l'atome le plus à gauche. On parle alors de résolution OLD (i.e., résolution linéaire ordonnée). Par ailleurs, la stratégie de parcours considérée est généralement soit une stratégie de parcours en profondeur d'abord, soit une stratégie de parcours en largeur d'abord.

A partir de maintenant, nous utilisons la notation SLD_{sp} (resp. OLD_{sp}) pour annoter tout objet en rapport avec une interprétation SLD (resp. OLD) utilisant une stratégie de parcours sp. On note sp = df ("depth-first") la stratégie de parcours en profondeur d'abord et sp = bf ("breadth-first") la stratégie de parcours en largeur d'abord.

Définition 1.25

Soit un couple (P,G) constitué d'un programme défini P et d'un but défini G et soit T un arbre SLD construit par une interprétation SLD de (P,G). Tout chemin de T est appelé un chemin (de dérivation) SLD. Tout chemin SLD partant de la racine de T et se terminant par un noeud vide est appelé une réfutation SLD. Soit m un chemin SLD et soit $\sigma_1,\ldots,\sigma_n$ les substitutions étiquetant les arcs successifs de m. La substitution étiquetant m est la composition $\sigma = \sigma_n o \ldots o \sigma_1$. Lorsque m est une réfutation SLD, σ est appelée une substitution réponse.

5.2. Sémantique opérationnelle SLD

La sémantique opérationnelle d'un programme P est une description de l'ensemble des exécutions possibles de P. Il est à noter qu'à cause des branches infinies apparaissant dans certains arbres SLD, il n'est pas toujours possible d'accéder (en un temps fini) à chaque noeud d'un arbre. Cela dépend de la stratégie de parcours utilisée. Ceci nous amène à considérer la définition suivante :

Définition 1.26
Un arbre SLD partiel est obtenu à partir d'un arbre SLD T en supprimant un nombre arbitraire de sous-arbres de T (tout en conservant la racine de T).

A partir de maintenant, l'interprétation SLD_{sp} d'un couple (P,G) constitué d'un programme P et d'un but G désigne la construction d'un arbre SLD partiel où n'apparaissent que les ω premiers noeuds visités par la stratégie de parcours sp. La sémantique opérationnelle SLD_{sp} d'un programme P est alors caractérisée par l'ensemble des arbres SLD partiels construits à partir de P par une interprétation SLD_{sp}. De cette sémantique opérationnelle sont issues d'autres sémantiques opérationnelles qui peuvent être qualifiées de partielles car obtenues après sélection de certaines propriétés du comportement opérationnel. La sémantique de l'ensemble des succès SLD est l'une d'entre elles. Notons que par défaut, la stratégie de parcours bf est sous-entendue à partir de maintenant.

Définition 1.27
Soit un programme P, l'ensemble des succès SLD_{sp} de P, noté $SS_{(SLD_{sp})}\llbracket P \rrbracket$, est l'ensemble des atomes a appartenant à B_P tel qu'il existe une réfutation SLD dans l'arbre SLD partiel construit à partir de $(P, \neg a)$ par un interpréteur SLD_{sp}.

Le résultat suivant établit que la sémantique logique et la sémantique de l'ensemble des succès SLD sont équivalentes puisque la sémantique déclarative a déjà été montrée égale à son plus petit modèle de Herbrand (M_P). Il est dû à [Apt et van Emden 82].

Théorème 1.28 [Apt et van Emden 82]
Soit un programme P, $SS_{(SLD)}\llbracket P \rrbracket = M_P$.

Le théorème reste valable si on restreint l'ensemble des arbres SLD à ceux qui sont construits à partir d'une fonction de sélection particulière. En particulier, $SS_{(OLD)}\llbracket P \rrbracket = M_P$.

Le théorème suivant montre que la sémantique de l'ensemble des succès OLD_{bf} est un sur-ensemble de la sémantique de l'ensemble des succès OLD_{df}. Cela

signifie que les interpréteurs OLD_{df} ne trouvent pas assez de succès à cause de la stratégie de parcours en profondeur d'abord.

Théorème 1.29
Soit un programme P, $SS(OLD_{df})\lfloor P \rfloor \subseteq SS(OLD_{bf})\lfloor P \rfloor = M_P$.

6. Exemple

On donne un exemple pour lequel un interpréteur OLD_{df} se trouve "piégé" par une branche infinie.

Soit le programme P suivant :

reach(X,Y) ← reach(X,Z), edge(Z,Y).
reach(X,X).
edge(a,b).
edge(a,c).
edge(b,a).
edge(b,d).

Soit le but G suivant :

← reach(a,X). .

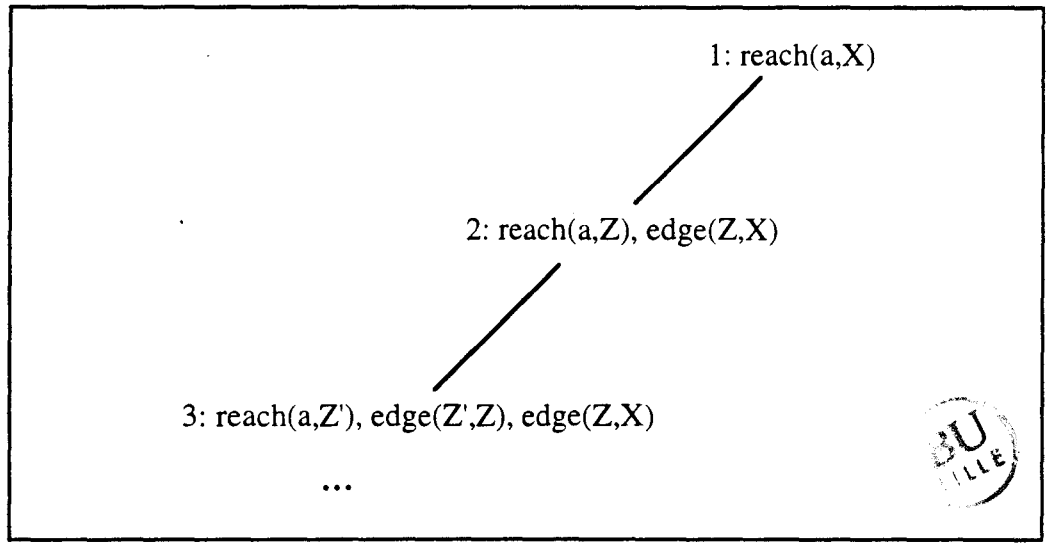

Figure 1.30

L'interprétation OLD_{df} de (P,G) fournit l'arbre OLD partiel de la figure 1.30 tandis que l'interprétation OLD_{bf} de (P,G) fournit l'arbre OLD de la figure 1.31.

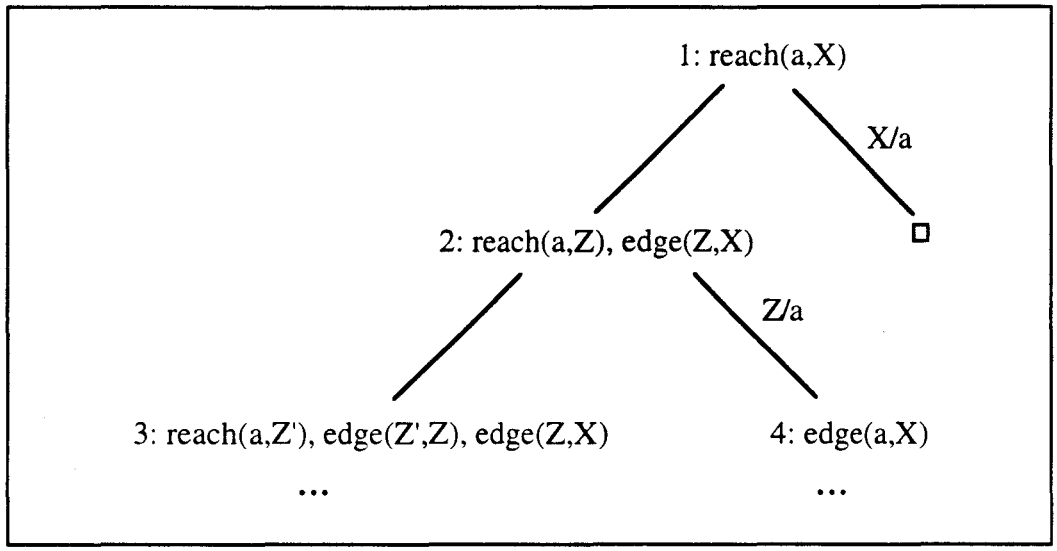

Figure 1.31

Cet exemple montre bien qu'un interpréteur OLD_{df} peut ne pas donner toutes les réponses souhaitées. Un interpréteur OLD_{bf} semble donc préférable. Malheureusement, celui-ci construit un arbre infini alors que le nombre des solutions est borné. Une réponse à ce problème de la non-terminaison de la résolution est en partie donnée par une technique de tabulation (voir chapitre 6).

Chapitre 2

Programmation Logique avec Contraintes

1. Introduction

Cette partie est consacrée à la programmation logique avec contraintes, et plus précisément au modèle générique de programmation logique avec contraintes introduit par [Jaffar et Lassez 86] sous le terme de CLP. La rédaction de cette partie est inspirée essentiellement des travaux de [Jaffar et Lassez 86], [Jaffar et Lassez 87] et [Maher 92]. Une introduction générale à la programmation logique avec contraintes est donnée par [Van Hentenryck 89], [Cohen 90] et [Fruhwirth et al. 92].

CLP, modèle générique de programmation logique avec contraintes, a été proposé par [Jaffar et Lassez 86] de manière à intégrer un mécanisme de calcul basé sur la manipulation de contraintes à l'intérieur du cadre habituel de la programmation logique. La généricité de CLP signifie qu'il est possible de définir différentes instances de CLP (i.e., différents CLP-langages) en associant au domaine de calcul du modèle une algèbre (ou une théorie) particulière. Tout CLP-langage est ainsi caractérisé par une sémantique algébrique. CLP conserve en outre les bonnes propriétés sémantiques de la programmation logique, à savoir l'équivalence des sémantiques opérationnelles, de point fixe et de la théorie des modèles.

Les contraintes manipulées par un CLP-langage représentent toute la puissance du langage. Savoir décider de la satisfiabilité d'un ensemble de contraintes est déterminant en programmation logique avec contraintes car la sémantique opérationnelle est basée sur un test de satisfiabilité. Un CLP-langage est donc

avant tout construit sur la base d'un univers de contraintes (muni d'une interprétation) et d'un solveur de contraintes (algorithme permettant de décider de la satisfiabilité d'un ensembles de contraintes). De nombreuses techniques, dites de consistance, ont été conçues pour optimiser cette décision [Van Hentenryck 89]. Par exemple, Prolog est un CLP-langage particulier où le test de satisfiabilité se limite à l'unification de termes. Les CLP-langages qui ont été proposées, les plus connus étant Prolog III [Colmerauer 90], CLP(IR) [Jaffar et Michaylov 87], CHIP [Dincbas et al. 88] et CAL [Aiba et al. 88] , intègrent des domaines tels que les entiers, les rationnels, les réels, les booléens et les domaines finis. Il est à noter que tous ces langages n'ont pas été formulés explicitement à l'origine dans le cadre de CLP.

Le paradigme de la résolution par contraintes permet une représentation concise et naturelle de problèmes complexes [Jaffar et Lassez 87] car 1) les contraintes sont directement gérées par rapport au domaine du calcul plutôt que codées sous forme de termes Prolog et 2) les contraintes ont la capacité de représenter de manière encore plus symbolique l'information. L'introduction des contraintes améliore donc la puissance de modélisation des programmes logiques sans dégrader leur propriétés sémantiques. Les CLP-langages ouvrent la voie vers des systèmes de programmation déclaratif et efficace.

[Hofeld et Smolka 88] ont généralisé le modèle de programmation logique avec contraintes de [Jaffar et Lassez 86] en permettant la prise en compte simultanée de plusieurs interprétations du domaine. Ce nouveau modèle a été conçu pour répondre aux exigences du monde de la représentation des connaissances. Bien qu'il soit séduisant, nous n'envisagerons pas ce modèle par la suite.

Un CLP-langage $C\mathcal{L}$ est défini par un triplet $(\Sigma,\mathcal{L},\mathcal{A})$ où Σ, $\mathcal{L}$ et $\mathcal{A}$ désignent respectivement une signature, un langage de contraintes (appelé le domaine de $C\mathcal{L}$) et une Σ-algèbre (appelée l'interprétation de $C\mathcal{L}$). Le comportement opérationnel de tout programme écrit dans un CLP-langage est décrit par le mécanisme de la résolution SLD où la gestion de contraintes remplace la gestion de substitutions pour Prolog.

2. Syntaxe

On considère un ensemble $\mathcal{P}$ infini dénombrable de symboles de prédicat et un ensemble $\mathcal{V}$ infini dénombrable de symboles de variable. Les ensembles $\mathcal{P}$ et $\mathcal{V}$ sont considérés partagés par l'ensemble des CLP-langages. Par ailleurs, $\mathcal{R}en$ désigne l'ensemble des substitutions de renommage (applications bijectives) définies de $\mathcal{V}$ vers $\mathcal{V}$.

Syntaxiquement, un CLP-langage $\mathcal{CL}$ est défini à partir d'un couple $(\Sigma,\mathcal{L})$. $\Sigma = (S,\mathcal{F},C)$ est une signature composée d'un ensemble de sortes S, un ensemble $\mathcal{F}$ de symboles de fonction et un ensemble C de symboles de contraintes. Les symboles de contraintes sont des symboles de prédicat particuliers. Pour un CLP-langage donné, on supposera toujours que le symbole de contrainte = appartient à C (en fait, on suppose qu'il existe un symbole de contrainte $=_{s\times s}$ dans C par sorte s de S), de même que deux constantes $\bot$ et $\top$. Les contraintes atomiques sont les éléments de $\mathcal{A}tom(S,\mathcal{V},\mathcal{F},C)$. $\mathcal{L}$ désigne un sous-ensemble de l'ensemble des formules du premier ordre construites à partir de $\mathcal{A}tom(S,\mathcal{V},\mathcal{F},C)$. $\mathcal{L}$ est un langage de contraintes et est appelé le domaine de $\mathcal{CL}$. Si X est un symbole de variable et s une sorte, alors on considère que X_s est une contrainte atomique et que X_s appartient à $\mathcal{L}$. On suppose par ailleurs que :

- $\mathcal{L}$ contient les constantes $\bot$ et $\top$.

- $\mathcal{L}$ est clos par renommage,
i.e., si $c \in \mathcal{L}$ et $\rho \in \mathcal{R}en$ alors $\rho(c) \in \mathcal{L}$

- $\mathcal{L}$ est clos par conjonction,
i.e., si $c_1 \in \mathcal{L}$ et $c_2 \in \mathcal{L}$ alors $c_1 \wedge c_2 \in \mathcal{L}$

- $\mathcal{L}$ est clos par quantification existentielle,
i.e., si $c \in \mathcal{L}$ et $X \in \mathcal{V}$ alors $\exists X\ c \in \mathcal{L}$.

Dans le papier de [Jaffar et Lassez 86], le langage de contraintes $\mathcal{L}$ est défini comme étant la clôture par conjonction de l'ensemble $\mathcal{A}tom(S,\mathcal{V},\mathcal{F},C)$. La différence essentielle de notre hypothèse est l'exigence d'un langage clos par quantification existentielle d'une part et la possibilité d'intégrer n'importe quelle formule du langage du premier ordre d'autre part. On retrouve cette distinction chez [Maher 87,92] et [Saraswat et Rinard 90].

Les éléments de $\mathcal{L}$ sont des contraintes et sont notées par $c,d,...$ On conserve les notations du chapitre 1 en ce qui concerne la classification des variables qui apparaissent dans une contrainte c : $Var(c)$, $Var_{free}(c)$, $Var_{bound}(c)$, $Var_{\exists}(f)$ et $Var_{\forall}(c)$ représentent respectivement l'ensemble des variables, des variables libres, des variables liées, des variables liées existentiellement et des variables liées universellement, de c.

Pour l'écriture des clauses du CLP-langage $\mathcal{CL}$, on ne considère que des atomes (plats) homogènes, c'est à dire des atomes linéaires (sans double occurrence d'une même variable) et sans symbole de fonction. On note $\mathcal{H}$ l'ensemble des

atomes homogènes construits à partir de $\mathcal{V}$ et $\mathcal{P}$†. Les éléments de $\mathcal{H}$ sont des atomes homogènes et sont notés par $a,b,h,...$ Par la suite, chaque fois qu'une séquence (un n-uplet) d'atomes homogènes (un élément de $\mathcal{H}^n$) est manipulée, on suppose qu'aucune variable n'apparaît deux fois dans cette séquence. Les éléments de $\mathcal{H}^n$ sont des séquences d'atomes et sont notés $as = (a_1,...,a_n)$, ... Les éléments de $\mathcal{L}\times\mathcal{H}^n$ sont des séquences d'atomes contraints et sont notés $cas = (c,a_1,...,a_n)$, ... $\Diamond$ est un symbole spécial séparant dans les notations contraintes et atomes.

Définition 2.1

Une $C\mathcal{L}$-clause définie est une règle de la forme :

$h \leftarrow d \Diamond b_1,...,b_m$

tel que $(d,h,b_1,...,b_m) \in \mathcal{L}\times\mathcal{H}^n$.

Définition 2.2

Un $C\mathcal{L}$-programme défini est un ensemble de $C\mathcal{L}$-clauses définies.

Un $C\mathcal{L}$-but défini est une $C\mathcal{L}$-clause définie sans tête de règle.

Définition 2.3

Un $C\mathcal{L}$-but défini est une règle de la forme :

$\leftarrow c \Diamond a_1,...,a_n$

tel que $(c,a_1,...,a_n) \in \mathcal{L}\times\mathcal{H}^n$.

Un $C\mathcal{L}$-but défini est dit vide lorsque n = 0. Pour la suite, clauses, programmes et buts sont sous-entendus définis.

Le fait de ne considérer que des atomes homogènes constitue une différence avec la présentation de [Jaffar et Lassez 86] où une telle restriction n'est pas faite. Cependant, il est clair que ces deux écritures sont opérationellement équivalentes puisque toute clause :

$$p(\tilde{t}_0) \leftarrow d \Diamond q_1(\tilde{t}_1),...,q_m(\tilde{t}_m),$$

se réécrit en :

$$p(\tilde{X}_0) \leftarrow d' \Diamond q_1(\tilde{X}_1),...,q_m(\tilde{X}_m),$$

---

† $\mathcal{H}$ = {p(X_1,...,X_n) tel que
a) $p \in \mathcal{P}, X_1 \in \mathcal{V}, ..., X_n \in \mathcal{V}$
b) $X_i \neq X_j$ pour $1 \leq i \neq j \leq n$ }

sachant que $\tilde{t}_0, \tilde{t}_1, \ldots, \tilde{t}_m$ représentent des n-uplets de termes, $\tilde{X}_0, \tilde{X}_1, \ldots, \tilde{X}_m$ représentent des n-uplets de variables (toutes distinctes) et que d' est la conjonction de d et de toutes les équations $X = t$ issues du remplacement d'un terme t par une variable X [Filè et Sottero 91].

Pour finir, nous introduisons quelques notations.

Notation 2.4

CL-Clause désigne l'ensemble des CL-clauses.
CL-Prog désigne l'ensemble des CL-programmes.
CL-Goal désigne l'ensemble des CL-buts.

3. Sémantique

On suppose que la syntaxe d'un CLP-langage CL est donnée, i.e., on suppose un couple $(\Sigma, \mathcal{L})$ établi comme ci-dessus. Sémantiquement, CL est définie par une Σ-algèbre $\mathcal{A}$ qui est appelée l'interprétation de CL. On suppose que :

- $\mathcal{A}$ est "solution-compact"
- le symbole = est interprété comme l'égalité syntaxique
- le symbole $\bot$ est interprété à faux
- le symbole $\top$ est interprété à vrai

On donne ainsi aux programmes logiques avec contraintes une sémantique algébrique, les contraintes étant interprétées par $\mathcal{A}$. La propriété de "solution-compactness" permet d'établir de bonnes relations entre les diverses sémantiques des programmes logiques avec contraintes.

Définition 2.5

Une Σ-algèbre de domaine $\mathcal{D}$ est solution-compact ssi

- tout élément de $\mathcal{D}$ est l'unique solution de la conjonction d'un ensemble S (éventuellement infini) de contraintes,
  i.e., $\forall\, d \in \mathcal{D}, X = d \Leftrightarrow \wedge \{c : c \in S\}$

- le complément $c' = \neg c$ d'une contrainte c s'exprime sous la forme d'une disjonction d'un ensemble S (éventuellement infini) de contraintes,
  i.e., $c' \Leftrightarrow \vee \{c : c \in S\}$.

Par rapport aux hypothèses effectuées concernant la syntaxe des contraintes, il est nécessaire que l'interprétation des contraintes de la forme X_S corresponde à l'idée intuitive suivante : $X \in \mathcal{A}(s)$.

Etant donnée une Σ-algèbre $\mathcal{A}$ de domaine $\mathcal{D}$, toute contrainte close (sans variables libres) de l'ensemble $\mathcal{L}$ obtient une valeur de vérité en suivant les

règles indiquées à la section 3 du chapitre 1 (il suffit simplement en plus de prendre en compte la signature des différents symboles). On pose pour la suite une Σ-algèbre $\mathcal{A}$ de domaine $\mathcal{D}$.

Définition 2.6

$\mathcal{V}al$ désigne l'ensemble des applications définies de $\mathcal{V}$ dans $\mathcal{D}$. Tout élément θ de $\mathcal{V}al$ est une $\mathcal{A}$-assignation de variables.

Soit c une contrainte (un élément de $\mathcal{L}$) et θ une $\mathcal{A}$-assignation de variables, $\theta(c)$ désigne la contrainte c où chaque variable libre X de c a été remplacée par $\theta(X)$. Si $\theta(c)$ est évaluée à vraie par rapport à $\mathcal{A}$, alors θ est dit $\mathcal{A}$-solution de c et c est dit $\mathcal{A}$-satisfiable ou $\mathcal{A}$-consistant. Si c n'admet aucune $\mathcal{A}$-solution alors c est dit insatisfiable, sinon c est dit satisfiable. Nous définissons maintenant les notions de $\mathcal{A}$-base et de $\mathcal{A}$-modèle d'un $C\mathcal{L}$-programme P.

Définition 2.7

Soit P un $C\mathcal{L}$-programme, la $\mathcal{A}$-base de P, noté $B_{(P,\mathcal{A})}$, est l'ensemble $B_{(P,\mathcal{A})} = \{\theta(a) \mid \theta \in \mathcal{V}al \text{ et } a \in \mathcal{H}\}$.

Tout sous-ensemble de $B_{(P,\mathcal{A})}$ est appelée une $\mathcal{A}$-interprétation.

Définition 2.8

Soit P un $C\mathcal{L}$-programme, un $\mathcal{A}$-modèle de P est une $\mathcal{A}$-interprétation I telle que pour toute règle C : $h \leftarrow d \Diamond b_1,\ldots,b_m$ de P et pour toute $\mathcal{A}$-assignation θ de variables qui est $\mathcal{A}$-solution de d, on a :
$\{\theta(b_1),\ldots,\theta(b_m)\} \subseteq I \Rightarrow \theta(h) \in I$.

Il existe un plus petit $\mathcal{A}$-modèle de P, noté $M_{(P,\mathcal{A})}$. De plus il est possible de calculer ce modèle à l'aide de l'opérateur conséquence immédiate $T_{(P,\mathcal{A})}$.

Définition 2.9

Soit P un $C\mathcal{L}$-programme, l'application $T_{(P,\mathcal{A})} \in \wp(B_{(P,\mathcal{A})}) \rightarrow \wp(B_{(P,\mathcal{A})})$ est définie par : $T_{(P,\mathcal{A})}(I) = \{\theta(h) \in B_{(P,\mathcal{A})}$ tel que

- $h \leftarrow d, b_1,\ldots,b_m$ est une règle de P
- θ est une $\mathcal{A}$-assignation de variables
- θ est $\mathcal{A}$-solution de d et $\{\theta(b_1),\ldots,\theta(b_m)\} \subseteq I\}$

Propriété 2.10 [Jaffar et Lassez 86]

$T_{(P,\mathcal{A})}$ est un opérateur continu.

Le plus petit point fixe de l'opérateur $T_{(P,\mathcal{A})}$, noté lfp $T_{(P,\mathcal{A})}$, est calculé en au plus ω étapes et est égal à $M_{(P,\mathcal{A})}$. On résume l'ensemble de ces équivalences par le théorème suivant :

Théorème 2.11 [Jaffar et Lassez 86]
Soit P est un CL-programme :
$M_{(P,\mathcal{A})} = \text{lfp}\, T_{(P,\mathcal{A})} = T_{(P,\mathcal{A})}{}^{\omega}(\emptyset)$.

Il existe de nombreux autres résultats, mais nous ne les citerons pas ici. En particulier, il est possible de donner à un CL-programme une sémantique logique équivalente à la sémantique algébrique. Il suffit de considérer une théorie $\mathcal{T}$ du premier ordre construite à partir de $\mathcal{A}tom(S,\mathcal{V},\mathcal{F},C)$ et qui corresponde à $\mathcal{A}$. L'ensemble des conséquences logiques de P en considérant la théorie $\mathcal{T}$ est alors équivalent au plus petit $\mathcal{A}$-modèle de P.

4. Pré-ordre sur un CLP-langage

Soit un CLP-langage $CL = (\Sigma,\mathcal{L},\mathcal{A})$. Dans cette partie est présentée la relation de pré-ordre $\vdash$ définie sur $\mathcal{L}$ et son extension sur les différents objets syntaxiques du langage. On commence par introduire l'opération de projection.

4.1. Projection

La projection (ou restriction) d'une contrainte c par rapport à un ensemble de variables V désigne la contrainte issue de c telle que toutes les variables libres de c autres que V sont quantifiées existentiellement. On utilise une notation opérationnelle pour l'application, notée $\downarrow$, de projection.

Définition 2.12
L'application $\downarrow$ est définie de $\mathcal{L}\times\wp(\mathcal{V})$ vers $\mathcal{L}$ par :
$\forall\, c \in \mathcal{L}, \forall\, V \in \wp(\mathcal{V}),\ c\downarrow V = \exists V'\ c$ où $V' = Var_{free}(c) - V$

Par exemple, pour Prolog†, on sait qu'une contrainte mise sous forme normale est une substitution idempotente. Si $\sigma = \{X=s(Y),Z=b\}$ alors $\sigma\downarrow\{X\} = \exists Y\ \exists Z\ (X=s(Y)\wedge Z=b)$, aussi $\sigma\downarrow\{X\} = \exists Y\ X=s(Y) \wedge \exists Z\ Z=b$ et finalement (si b fait partie du domaine de l'interprétation) $\sigma\downarrow\{X\} = \exists Y\ X=s(Y)$. Or, d'après la définition 1.14, le résultat de $\sigma\downarrow\{X\}$ devrait être $X=s(Y)$. Cette différence s'explique par le fait que pour Prolog, on considère les variables implicitement quantifiées.

Pour la suite, nous utilisons les raccourcis de notation suivants :

---

† Prolog est un CLP-langage qui manipule des équations sur les arbres (ou termes) finis. Pour cette raison, on appele aussi Prolog par CLP($\mathcal{FT}$) où $\mathcal{FT}$ est mis pour "finite trees".

$\forall\ c \in \mathcal{L}, \forall\ a \in \mathcal{H}, c{\downarrow}a = c{\downarrow}\mathrm{Var}_{\mathrm{free}}(a)$
$\forall\ c \in \mathcal{L}, \forall\ as \in \mathcal{H}^n, c{\downarrow}as = c{\downarrow}\mathrm{Var}_{\mathrm{free}}(as)$

ainsi que les projections implicites suivantes (par défaut) :

$\forall\ c \in \mathcal{L}, \forall\ a \in \mathcal{H}, (c,a)$ représente $(c{\downarrow}a, a)$
$\forall\ c \in \mathcal{L}, \forall\ as \in \mathcal{H}^n, (c,as)$ représente $(c{\downarrow}as, as)$

4.2. Relation de pré-ordre $\vdash$ sur $C\mathcal{L}$

L'implication $\rightarrow$ définie sur $\mathcal{L}$ à partir de $\mathcal{A}$ constitue assez naturellement une relation de pré-ordre. Une contrainte c est plus petite qu'une contrainte c' ssi c' est une approximation de c (i.e., c' est moins contraignante, moins précise) :

Définition 2.13
$\forall\ (c,c') \in \mathcal{L}^2,\ c \vdash c'$ ssi toute $\mathcal{A}$-assignation est $\mathcal{A}$-solution de $c \rightarrow c'$[†]

On peut caractériser la relation $\vdash$ comme suit :

Définition 2.14
$\forall\ c \in \mathcal{L}, [c] = \{\theta \in \mathcal{V}al \mid \theta$ est $\mathcal{A}$-solution de $c\}$

Propriété 2.15
$\forall\ (c,c') \in \mathcal{L}^2,\ c \vdash c'$ ssi $[c] \subseteq [c']$

On notera en particulier que $\bot$ (rien n'est possible) et $\top$ (tout est possible) sont tels que $[\bot] = \emptyset$ et $[\top] = \mathcal{V}al$.

Le préordre $\vdash$ engendre naturellement une relation d'équivalence. Par abus de notation, on notera souvent $c = \bot$ (resp. $c \neq \bot$) pour indiquer que c est équivalent (resp. n'est pas équivalent) à $\bot$.

4.3. Propriétés de $\vdash$

Certaines équivalences sont triviales. Par exemple, pour toute contrainte c, on a : $c \wedge \bot$ qui est équivalent à $\bot$, $c \wedge \top$ qui est équivalent à c et $c \wedge c$ qui est équivalent à c. La relation de pré-ordre $\vdash$ est stable (sur $\mathcal{L}$) par renommage, projection et conjonction. Les preuves sont immédiates.

[†] nous avons évité d'utiliser le terme "modèle" ici pour qu'il n'y ait pas de confusion avec la notion de $\mathcal{A}$-modèle définie plus haut.

Propriétés 2.16

$\forall (c,c') \in \mathcal{L}^2, \forall \rho \in \mathcal{R}en, \rho(c) \vdash \rho(c')$ si $c \vdash c'$
$\forall (c,c') \in \mathcal{L}^2, \forall V \subseteq \mathcal{V}, c{\downarrow}V \vdash c'{\downarrow}V$ si $c \vdash c'$
$\forall (c,c') \in \mathcal{L}^2, \forall (d,d') \in \mathcal{L}^2, c \wedge d \vdash c' \wedge d'$ si $c \vdash c'$ et $d \vdash d'$

Voici trois propriétés liant $\vdash$ et $\downarrow$: 1) la projection est une opération qui retourne une contrainte plus grande (moins contraignante), 2) la consistance d'une contrainte est indépendante de $\downarrow$ et 3) $\downarrow$ est un morphisme (pour $\wedge$) lorsque les deux contraintes liées par la conjonction ne partagent aucune variable libre avec $\mathcal{V}$ - V.

Propriétés 2.17

$\forall (c,d) \in \mathcal{L}^2, \forall V \in \wp(\mathcal{V})$,
1) $c \vdash c{\downarrow}V$
2) c est satisfiable ssi $c{\downarrow}V$ est satisfiable
3) $(c \wedge d){\downarrow}V = c{\downarrow}V \wedge d{\downarrow}V$ si $(Var_{free}(c) - V) \cap (Var_{free}(d) - V) = \emptyset$

4.4. Extension de $\vdash$

On étend inductivement la relation $\vdash$ sur différents domaines (ensembles) en considérant le passage 1) aux parties d'un ensemble, 2) au produit cartésien et 3) à l'espace des fonctions. On note $\vdash_A$ la relation $\vdash$ définie sur un ensemble A. Initialement, $\vdash$ est définie uniquement sur $\mathcal{L}$ (et $\mathcal{H}$ comme indiqué plus loin).

Définition 2.18

Si $\vdash$ est définie sur A et B alors :
1) $\vdash$ est définie sur $\wp(A)$ par : $\forall S \in \wp(A), \forall S' \in \wp(A)$,
$S \vdash_{\wp(A)} S'$ ssi $\forall a \in S, \exists a' \in S'$ tel que $a \vdash_A a'$
2) $\vdash$ est définie sur $A \times B$ par : $\forall (a,b) \in A \times B, \forall (a',b') \in A \times B$,
$(a,b) \vdash_{A \times B} (a',b')$ ssi $a \vdash_A a'$ et $b \vdash_B b'$
3) $\vdash$ est définie sur $A \rightarrow B$ par : $\forall f \in A \rightarrow B, \forall g \in A \rightarrow B$,
$f \vdash_{A \rightarrow B} g$ ssi $\forall (a,a') \in A^2, a \vdash_A a' \Rightarrow f(a) \vdash_B g(a')$

De manière à ne pas alourdir les notations, dans la suite nous ne précisons plus en général le domaine sur lequel est défini la relation $\vdash$ (le contexte suffit généralement à déterminer la relation). Par ailleurs, de manière à exploiter la définition 2.18, nous définissons (de manière relativement artificielle) la relation $\vdash$ sur l'ensemble $\mathcal{H}$ comme étant l'identité $I\delta$. Ceci nous permet, entre autre, d'obtenir la définition suivante :

$\forall as \in \mathcal{H}^n, \forall (c,c') \in \mathcal{L}^2, (c,as) \vdash (c',as)$ ssi $c \vdash c'$

Ainsi, en considérant une clause C : $h \leftarrow d \Diamond bs$ comme étant la séquence d'atomes contraints (d,h,bs) et un but G : $\leftarrow c \Diamond as$ comme étant la séquence

d'atomes contraints (c,as), la relation $\vdash$ est directement définie sur CL-Clause et CL-Goal. Puis par extension, $\vdash$ est définie sur CL-Prog en considérant un programme en tant que n-uplet $(C_0,...,C_n)$ de clauses suivant la numérotation établie. Cela signifie donc que pour tout couple (P,P') de CL-Prog^2, $P \vdash P'$ ssi P et P' possèdent le même nombre de clauses et pour toute clause C_i de P : $C_i \vdash C'_i$. Cette définition peut évidemment se généraliser en tenant compte du fait qu'un programme est un ensemble plutôt qu'un n-uplet (nous y renonçons par soucis de simplicité, ceci ne contrariant en rien la généralité des résultats à venir).

4.5. préordre $\vdash_r$ sur CL

A partir de la relation $\vdash$ définie ci-dessus, on introduit une relation de préordre $\vdash_r$ en considérant $\vdash$ modulo le renommage des variables.

Définition 2.19

Soit la relation $\vdash$ définie sur un ensemble D, la relation $\vdash_r$ est définie sur D comme suit $\forall (d,d') \in D^2$, $d \vdash_r d'$ ssi $\exists \rho \in \mathcal{R}en \mid d \vdash \rho(d')$.

Comme la relation $\vdash$ est stable par renommage, il est facile de constater que : $\forall (d,d') \in D^2$, $d \vdash_r d'$ ssi $\exists \rho \in \mathcal{R}en : \rho(d) \vdash d'$. On sait aussi que la relation $\vdash_r$ est stable par renommage mais à la différence de $\vdash$, $\vdash_r$ n'est pas nécessairement stable (sur $\mathcal{L}$) par projection et conjonction. Par exemple, $X=a \vdash_r Y=a$ et $Y=b \vdash_r Y=b$ mais $X=a \wedge Y=b$ n'est pas moins contraignant que $Y=a \wedge Y=b$.

Tandis que, pour un programme donné, le nom des variables est indépendant d'une clause à l'autre, la relation $\vdash_r$ définie sur CL-Prog tient compte du renommage de manière globale. Pour éviter ce problème, nous considérons dorénavant qu'aucune variable n'est partagée par deux clauses distinctes d'un même programme.

5. Résolution SLD

5.1. Description

Au chapitre 1 la résolution SLD a été définie par rapport à Prolog. On la définit maintenant de manière analogue par rapport à tout CLP-langage CL. La différence essentielle est l'élargissement du test d'unification à un test de satisfiabilité à chaque étape de dérivation SLD. Nous commençons par définir les notions de résolvant et d'étape de dérivation SLD.

Définition 2.20

Soient P un CL-programme et G : $\leftarrow c \Diamond a_1,\ldots,a_n$ un CL-but. Etant sélectionnés un atome a_i de G et une clause C_j de P, considérons une variante $h \leftarrow d \Diamond b_s$ de C_j (telle que $a_i = h$ et telle que les autres variables de cette variante soient nouvelles). Si $c \wedge d$ est satisfiable alors le CL-but H : $\leftarrow c \wedge d \Diamond a_1,\ldots,a_{i-1},b_s,a_{i+1},\ldots, a_n$ est appelé le (i,j)-résolvant de (P,G).

Par rapport à la présentation de [Jaffar et Lassez 86], nous nous arrangeons pour choisir une variante de la clause sélectionnée telle que $a_i = h$. Ceci permet de considérer implicitement le passage de paramètres (le lien entre a_i et h) plutôt qu'explicitement dans la nouvelle contrainte. Ceci n'est possible que par la présence d'atomes (plats) homogènes.

Une étape de dérivation SLD consiste à calculer un résolvant. Un arbre SLD est un objet qui est défini à partir d'un couple (P,G) constitué d'un CL-programme P et d'un CL-but G et qui mémorise une séquence d'étapes de dérivation SLD. Chaque noeud de cet arbre est étiqueté par un CL-but (et un entier i dont nous ne tenons pas compte ici) et chaque arc de cet arbre est étiqueté par une contrainte et un entier j. Un noeud étiqueté par la clause vide est appelé un noeud vide.

Un arbre SLD est dépendant d'une fonction de sélection, c'est à dire une fonction qui pour chaque noeud de l'arbre sélectionne un atome de l'étiquette de ce noeud. Etant donnée une fonction de sélection fs, un arbre SLD T est obtenu à partir d'un couple (P,G) comme suit :

- la racine de T est étiquetée par G
- si v est un noeud (non vide) de T étiqueté par H : $\leftarrow c \Diamond a_1,\ldots,a_n$ et si a_i est l'atome sélectionné par fs, alors pour toute clause C_j de P, on considère une variante $h \leftarrow d \Diamond b_s$ de C_j (telle que $a_i = h$ et les autres variables de cette variante soient nouvelles)). Si $c \wedge d$ est satisfiable alors on ajoute un noeud fils w à v tel que w est étiqueté par le (i,j)-résolvant de (P,H) et on ajoute un arc reliant v à w étiqueté par (d,j). Les différents noeuds sont ordonnés suivant l'ordre des clauses utilisées pour les obtenir.

Toutes les autres notions définies par rapport à Prolog : interprétation SLD, stratégie de parcours, résolution OLD, ... restent valables par rapport à CLP. Une seule différence notable concerne l'étiquetage d'un chemin SLD. Dans le cas de Prolog, on considérait la composition des substitutions étiquetant chaque arc du chemin SLD. Dans le cas de CLP, on considère la conjonction des contraintes étiquetant chaque arc du chemin.

5.2. Sémantique opérationnelle SLD

L'interprétation SLD_{sp} d'un couple (P,G) constitué d'un programme P et d'un but G désigne la construction d'un arbre SLD partiel où n'apparaissent que les ω premiers noeuds visités par la stratégie de parcours sp. La sémantique opérationnelle SLD_{sp} d'un programme P est alors caractérisée par l'ensemble des arbres SLD partiels construits à partir de P par une interprétation SLD_{sp}. Notons que par défaut, la stratégie de parcours bf ("breadth-first") est sous-entendue. On retrouve ici les résultats énoncés par rapport à la programmation logique.

Notons que l'on peut faire correspondre à tout élément $a = \theta(p(X_1,...,X_n))$ de $B_{(P,\mathcal{A})}$ un couple $(c,p(X_1,...,X_n))$ où c est égal à $X_1 = \theta(X_1) \wedge ... \wedge X_n = \theta(X_n)$. Par la suite, on confond $\neg a$ avec le but $\leftarrow c \Diamond p(X_1,...,X_n)$. La sémantique de l'ensemble des succès SLD est alors définie comme suit :

Définition 2.21

Soit P un $C\mathcal{L}$-programme, l'ensemble des succès SLD_{sp} de P, noté $SS(SLD_{sp})\llbracket P \rrbracket$, est l'ensemble des atomes a appartenant à $B_{(P,\mathcal{A})}$ tel qu'il existe une réfutation SLD dans l'arbre SLD partiel construit à partir de $(P,\neg a)$ par un interpréteur SLD_{sp}.

L'ensemble $SS(SLD)\llbracket P \rrbracket$ est égal à l'ensemble $[SS(P,\mathcal{A})]$ défini par [Jaffar et Lassez 86]. Ces derniers ont établi l'équivalence de la sémantique algébrique et de la sémantique de l'ensemble des succès SLD.

Théorème 2.22

Soit P un $C\mathcal{L}$-programme, $SS(SLD)\llbracket P \rrbracket = M_{(P,\mathcal{A})}$.

Le théorème reste valable si on restreint l'ensemble des arbres SLD à ceux qui sont construits à partir d'une fonction de sélection particulière. En particulier, $SS(OLD)\llbracket P \rrbracket = M_{(P,\mathcal{A})}$.

Les interpréteurs OLD_{df} ne trouvent pas assez de succès à cause de la stratégie de parcours en profondeur d'abord.

Théorème 2.23

Soit P un $C\mathcal{L}$-programme, $SS(OLD_{df})\llbracket P \rrbracket \subseteq SS(OLD_{bf})\llbracket P \rrbracket = M_{(P,\mathcal{A})}$.

Nous ne nous sommes préoccupés ici que de la sémantique des succès clos d'un programme. La sémantique des succès libres d'un programme est examinée en particulier par [Gabbrielli et Levi 92].

5.3. Arbres SLD comme systèmes de transitions

Tout arbre SLD T peut être caractérisé par un système de transitions $(\mathcal{N}, i, r)$ où :

- $\mathcal{N}$, l'ensemble des états du système, désigne l'ensemble des noeuds de l'arbre.
- $i \in \mathcal{N}$, l'état initial du système, désigne la racine de l'arbre.
- $r \subset \mathcal{N} \times \mathcal{N}$, la relation de transition du système, désigne l'ensemble des arcs de l'arbre.

L'application level indique le niveau (la profondeur) d'un arbre et d'un noeud donné dans cet arbre.

Définition 2.24

Soit T = $(\mathcal{N}, i, r)$ un arbre SLD, on définit l'application level comme suit :

- level(i) = 1 et $\forall (v,w) \in \mathcal{N}^2$, level($w$) = level($v$) +1 ssi $(v,w) \in r$
- level(T) = max {level(v) : $v \in \mathcal{N}$}.

Pour conserver toute l'information inhérente aux arbres SLD, nous avons besoin de trois applications supplémentaires : label associe à chaque noeud d'un arbre l'étiquette de ce noeud, cstr et num associent respectivement à chaque arc d'un arbre la contrainte et le couple d'entiers étiquetant cet arc. L'application cstr est étendue assez naturellement de r à r^+. Elle associe alors à chaque chemin (suite d'arcs) une contrainte qui est la conjonction de toutes les contraintes étiquetant les arcs du chemin. Nous définissons pour finir une application size qui associe à chaque noeud d'un arbre la taille (le nombre d'atomes) constituant l'étiquette de ce noeud.

Dans les chapitres suivants, on considérera la résolution OLD_{bf} par défaut, c'est pourquoi nous donnons les définitions suivantes.

Notation 2.25

$C\mathcal{L}$-Tree désigne l'ensemble des arbres OLD construits à partir de tout couple (P,G) composé d'un $C\mathcal{L}$-programme P et d'un $C\mathcal{L}$-but G.

La relation d'ordre $\vdash$ est définie sur $C\mathcal{L}$-Tree. Etant donnés deux arbres T et T', T $\vdash$ T' signifie que le squelette de T se retrouve dans T' sous une forme moins générale.

Définition 2.26

$\forall$ T = $(\mathcal{N}, i, r) \in C\mathcal{L}$-Tree et $\forall$ T' = $(\mathcal{N}', i', r') \in C\mathcal{L}$-Tree,
T $\vdash$ T' ssi il existe une application map $\in \mathcal{N} \rightarrow \mathcal{N}'$ telle que :

- map(i) = i'
- si map(v) = v' alors
  - label(v) $\vdash$ label(v')

- $\forall (v,w) \in r, \exists (v',w') \in r'$ tel que
  $\text{num}(v,w) = \text{num}(v',w')$
  $\text{map}(w) = w'$.

Notation 2.27
Res désigne l'application qui pour tout couple (P,G) composé d'un CL-programme P et d'un CL-but G associe l'arbre OLD construit à partir de (P,G).

La sémantique opérationnelle (OLD_{bf} par défaut) d'un CLP-langage CL est ainsi illustrée par :

$$CL\text{-Prog} \times CL\text{-Goal} \xrightarrow{\text{Res}} CL\text{-Tree}$$

6. Exemples

Quelques CLP-langages sont maintenant présentés : il s'agit de CLP($\mathcal{FT}$) = Prolog, de CLP(IB) et de CLP(IR).

6.1. CLP($\mathcal{FT}$)

CLP($\mathcal{FT}$) est un CLP-langage $(\Sigma,\mathcal{L},\mathcal{A})$ tel que
- $\Sigma = (S,\mathcal{F},C)$ où
  - S = {term}
  - $\mathcal{F}$ désigne un ensemble fini de symboles de fonction
  - $C = \{=,\bot,\top\}$
- $\mathcal{L}$ désigne l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},C)$ clos par
  - conjonction
  - quantification existentielle
- $\mathcal{A}$ est tel que
  - l'interprétation de S est donnée par $\mathcal{A}$(term) = $\mathcal{T}erm(\mathcal{F})$
  - l'interprétation des symboles de $\mathcal{F}$ est libre.

$\mathcal{L}$ est clairement clos par renommage. Notons qu'il est également possible de choisir l'univers de Herbrand étendu $\mathcal{T}erm(\mathcal{V},\mathcal{F})$, au lieu de l'univers de Herbrand classique $\mathcal{T}erm(\mathcal{F})$. Cet univers correspond alors à l'ensemble $T_{D(V)}$ de [Falaschi et al. 89].

On peut reformuler ici, de façon différente, certains résultats du chapitre 1, section 4. Cette différence provient essentiellement de ce que des systèmes d'équations sont manipulés à la place de termes et de substitutions. Par la suite (en particulier au chapitre 8), nous manipulons plutôt des systèmes d'équations

mais pour les illustrations nous continuons, par simplicité, à raisonner en termes de termes et de substitutions.

Un système d'équations $\mathcal{S}$ est un ensemble d'éléments de $\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{C})$. Dans tout système d'équations $\mathcal{S}$, n'apparaissent que des variables libres ou liées existentiellement. On utilisera donc les ensembles $Var(\mathcal{S})$, $Var_{free}(\mathcal{S})$ et $Var_{\exists}(\mathcal{S})$. Un système $\mathcal{S}$ est implicitement interprété par : $\exists V\ \mathcal{S}$ où V représente $Var_{\exists}(\mathcal{S})$ et $\mathcal{S}$ représente une conjonction d'équations. De ce fait, l'ensemble vide représente $\top$. On considère également comme un système, la contrainte $\bot$. Tout système $\mathcal{S}$ possède une forme résolue qui est notée $res(\mathcal{S})$ et qui désigne le système d'équations obtenue par application de l'algorithme d'unification ou de forme résolue [Martelli et Montanari 82 , Lassez et al. 87]. En cas d'echec, on note $res(\mathcal{S}) = \bot$ et en cas de succès $res(\mathcal{S})$ correspond à une substitution idempotente. On tient compte des variables quantifiées existentiellement pour la forme résolue. Si $\mathcal{S}$ est un système sous forme résolue (ou plus simplement résolu) alors on note respectivement par $Dom(\mathcal{S})$ et par $Ran(\mathcal{S})$ l'ensemble des variables qui apparaissent en partie gauche et en partie droite d'une équation de $\mathcal{S}$. On peut alors supposer que le système résolu $\mathcal{S}$ vérifie les propriétés suivantes :

- $Dom(\mathcal{S}) = Var_{free}(\mathcal{S})$
- $Ran(\mathcal{S}) = Var_{\exists}(\mathcal{S})$

La première propriété s'obtient par projection. On élimine les équations triviales de la forme $X=t$ où $X \in Var_{\exists}(\mathcal{S})$. La seconde propriété s'obtient en introduisant éventuellement de nouvelles variables quantifiées existentiellement. On remplace dans $\mathcal{S}$ toute variable X de $Var_{free}(\mathcal{S})$ qui apparaît dans $Ran(\mathcal{S})$ par une nouvelle variable X' quantifiée existentiellement et on ajoute l'équation X=X'.

La conjonction $\mathcal{S} \wedge \mathcal{S}'$ de deux systèmes $\mathcal{S}$ et $\mathcal{S}'$ désigne le système $\mathcal{S}'' = \mathcal{S} \cup \mathcal{S}'$ tel que $Var_{\exists}(\mathcal{S}'') = Var_{\exists}(\mathcal{S}) \cup Var_{\exists}(\mathcal{S}')$ en supposant $Var_{\exists}(\mathcal{S}) \cap Var_{\exists}(\mathcal{S}') = \emptyset$. La quantification existentielle d'un système $\mathcal{S}$ par un ensemble de variables $V \subseteq Var_{free}(\mathcal{S})$ désigne le système $\mathcal{S}' = \mathcal{S}$ tel que $Var_{\exists}(\mathcal{S}') = Var_{\exists}(\mathcal{S}) \cup V$. L'ensemble des systèmes d'équations est donc clos par conjonction et quantification existentielle.

La relation d'ordre $\vdash$ définie entre les systèmes résolus correspond à la relation $\leq$ définie entre les substitutions idempotentes. On sait que le plus petit majorant (au nom des variables près) de deux systèmes résolus $\mathcal{S}$ et $\mathcal{S}'$ existe. Il est noté $lub(\mathcal{S},\mathcal{S}')$ et correspond à l'anti-unifié de deux substitutions idempotentes. On sait aussi que le plus grand minorant de deux systèmes résolus $\mathcal{S}$ et $\mathcal{S}'$ existe. Il est noté $glb(\mathcal{S},\mathcal{S}')$ et correspond à l'unifié de deux substitutions idempotentes (telles que $Var_{\exists}(\mathcal{S}) \cap Var_{\exists}(\mathcal{S}') = \emptyset$).

On note sol($\mathcal{S}$) l'ensemble des solutions de $\mathcal{S}$, c'est à dire l'ensemble des assignations de variables θ qui satisfont les équations de $\mathcal{S}$. En utilisant cette notation, on sait que pour tout couple ($\mathcal{S},\mathcal{S}'$) de systèmes, on a :

$$sol(lub(\mathcal{S},\mathcal{S}')) \supseteq sol(\mathcal{S}) \cup sol(\mathcal{S}')$$
$$sol(glb(\mathcal{S},\mathcal{S}')) = sol(\mathcal{S}) \cap sol(\mathcal{S}')$$

6.2. CLP(IB)

CLP(IB) est un CLP-langage ($\Sigma,\mathcal{L},\mathcal{A}$) tel que

- $\Sigma = (S,\mathcal{F},C)$ où
  - $S = \{bool\}$
  - $\mathcal{F} = \emptyset$
  - $C = \{=,\bot,\top\}$
- $\mathcal{L}$ désigne l'ensemble des formules du premier ordre construites à partir de l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},C)$.
- $\mathcal{A}$ est tel que l'interprétation de S est donnée par $\mathcal{A}(bool) = \mathbb{B}$.

$\mathcal{L}$ est clairement clos par renommage et conjonction. $\mathcal{L}$ est également clos par quantification existentielle car la formule $\exists$ X A(X) est équivalente à la formule A(faux) $\vee$ A(vrai).

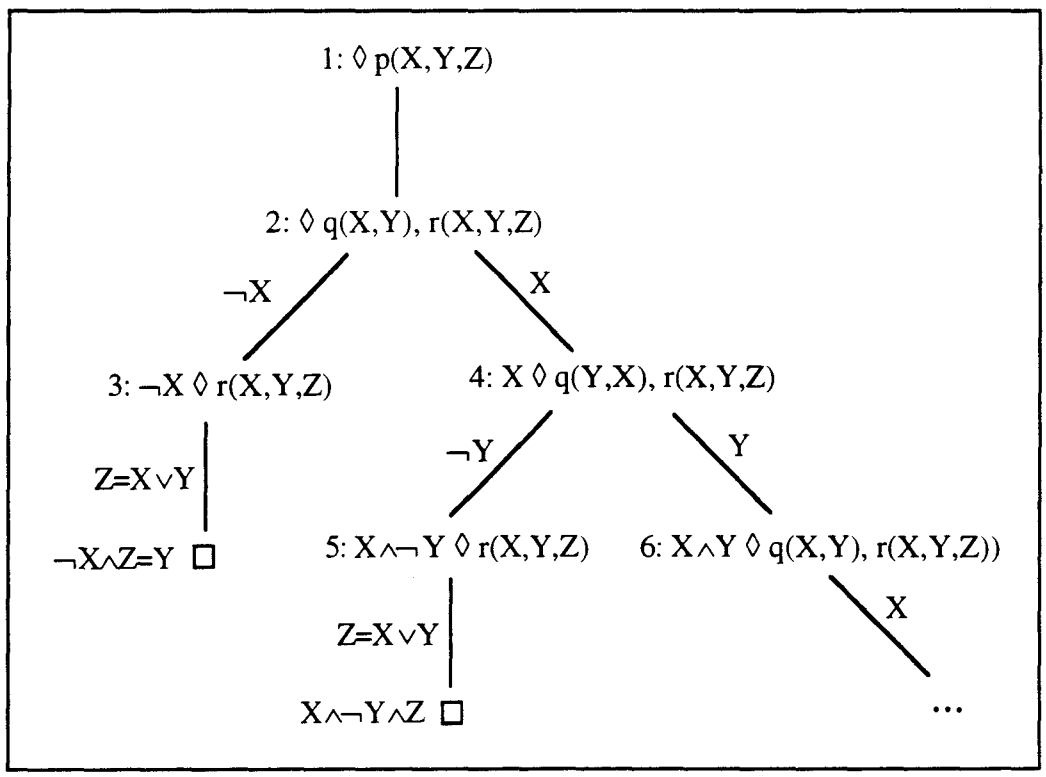

Figure 2.28

Soit le programme P suivant :

p(X,Y,Z) :- ◊ q(X,Y), r(X,Y,Z).
q(X,Y) :- ¬X ◊ .
q(X,Y) :- X ◊ q(Y,X).
r(X,Y,Z) :- Z=X∨Y ◊ .

Soit le but G suivant :

:- ◊ p(X,Y,Z).

On notera que la contrainte ⊤ est sous-entendue par défaut. Ce programme écrit en CLP(IB) code une fonction booléenne à trois arguments. Cet exemple est reconsidéré dans les chapitres suivants. Notons que X et ¬X représentent respectivement la contrainte X=vrai et la contrainte X=faux.

L'interprétation OLD de (P,G) fournit l'arbre OLD de la figure 2.28. Deux solutions sont obtenues : la première est ¬X∧Z=Y et la seconde est X∧¬Y∧Z. La résolution se poursuit indéfiniment sans jamais donner de nouvelles solutions.

6.3. CLP(IR)

CLP(IR) est un CLP-langage $(\Sigma,\mathcal{L},\mathcal{A})$ tel que

- $\Sigma = (S,\mathcal{F},C)$ où
  - $S = \{real\}$,
  - $\mathcal{F} = \{+,-,*,/\}$
  - $C = \{=,\perp,\top\} \cup \{\leq,<,\geq,>\}$
- $\mathcal{L}$ désigne l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},C)$ clos par :
  - conjonction
  - quantification existentielle
- $\mathcal{A}$ est tel que
  - l'interprétation de S est donnée par $\mathcal{A}(real) = \mathbb{R}$
  - les symboles de $\mathcal{F}$ et C sont interprétés de façon naturelle sur $\mathbb{R}$.

CLP(IR) est présenté dans les articles de [Jaffar et Lassez 87] et de [Jaffar et Michaylov 87]. Nous prenons maintenant un exemple de programmation en CLP(IR) tiré de [Jaffar et Lassez 87]. Cet exemple illustre bien la puissance des langages de contraintes car il met en évidence la possibilité de produire des réponses symboliques.

Soit P le programme suivant :

mortgage(P,T,I,R,B) :-

```
        T=1, B+R=P*(1+I/1200) ◊ .

    mortgage(P,T,I,R,B) :-
        T>1, P'=P*(1+I/1200), T'=T-1 ◊ mortgage(P',T',I,R,B)
```

Ce programme écrit en CLP(IR) permet de gérer les modalités d'un prêt. P représente la valeur courante du prêt restant à rembourser, T représente la durée de remboursement du prêt (en mois), I représente le taux d'intérêt mensuel (en %), R représente le remboursement mensuel et B représente l'ajustement final.

Si on propose comme but :

```
:- mortgage(123456,120,12,R,0)
```

alors la réponse fournie par le programme est R=1771.23. Ce type de réponse aurait très bien pu être fournie par un programme écrit dans un langage de programmation impératif (ou procédural). Par contre, le but suivant :

```
:- mortgage(P,120,12,R,B)
```

démontre la puissance d'expression intrinsèque à CLP(IR) puisque la réponse obtenue est une équation liant P, R et B : B=0.302995*P-69.700522*R.

Partie II

Interprétation Abstraite

L'interprétation abstraite [Cousot et Cousot 77] est une technique d'analyse statique qui permet d'étudier le comportement dynamique d'un programme au seul vu de son code source. Une interprétation abstraite est la donnée de deux sémantiques et d'une relation ([Cousot et Cousot 92b], [Marriott 93]). L'une des deux sémantiques est dite concrète (ou standard) tandis que l'autre est dite abstraite (ou non standard). La relation (dite d'approximation) est définie entre le domaine concret et le domaine abstrait. Une interprétation abstraite est consistante ssi le résultat du calcul concret à partir d'une donnée concrète d est approchée par le résultat du calcul abstrait à partir d'une donnée abstraite qui approche d. Dans le chapitre 3, nous rappelons les aspects théoriques de l'interprétation abstraite. Dans le chapitre 4, nous présentons les différentes applications de l'interprétation abstraite en programmation logique.

Chapitre 3

Aspects Théoriques

1. Introduction

D'un point de vue pratique, l'interprétation abstraite est une technique qui permet l'analyse statique du comportement dynamique des programmes. Cette technique était déjà utilisée avant sa formalisation. Par exemple, P. Naur lors de la conception du compilateur GIER ALGOL III en 1963 utilise un processus de vérification de types appelé pseudo-évaluation qu'il décrit en ces termes:

> "A process which combines the operators and operands of the source text in a manner in which an actual evaluation would do it, but which operates on descriptions of the operands, not on their values.".

L'interprétation abstraite est donc une méthode d'approximation finie qui permet, entre autres, d'inférer certaines propriétés sur un programme donné. Ces propriétés se regroupent en différentes classes ou familles en fonction de l'objectif de l'analyse :

- optimiser le code compilé d'un programme,
  e.g., ramasse-miettes plus efficace, élimination du code inutile, ...

- transformer un programme,
  e.g., évaluation partielle, parallélization,...

- établir certaines preuves de programmes,
  e.g., preuve d'arrêt, preuve de correction,...

Si la formalisation de l'interprétation abstraite était initialement prévue pour la classe des langages impératifs (ou procéduraux), son adaptation aux autres classes de programmation n'a pas posé de trop gros problèmes puisque cette

formalisation est exprimée à l'origine en termes de systèmes de transitions modélisant une sémantique opérationnelle. La sémantique concrète désigne généralement une sémantique opérationnelle et la sémantique abstraite est généralement obtenue à partir de celle-ci par observation de certains points d'intérêts. Lorsqu'il est prouvé qu'une sémantique abstraite approche une sémantique concrète via une relation d'approximation, on sait alors que toute information apportée (déduite) par la sémantique abstraite est correcte.

Dans cette partie sont présentées les différentes contributions à l'étude théorique de l'interprétation abstraite, essentiellement issues du travail de [Cousot et Cousot 76,77,79,81,92a,92b,92c] et [Marriott 93]. Le lecteur pourra toutefois se reporter volontiers aux notes et tutoriaux de [Bruynooghe et de Schreye 88], [Marriott et Sondergaard 89a], [Debray 92,93] et [Hermenegildo 92] ainsi qu'aux ouvrages suivants : [Abstract Interpretation 87] et [Journal of LP 92]. La rédaction de cette partie est tantôt assez proche de la source dont nous nous inspirons, tantôt assez personnelle.

[Cousot 92c] et [Marriott 93] montrent les variantes possibles qu'offrent le cadre de l'interprétation abstraite. La sémantique abstraite peut être liée à la sémantique concrète par

1) une relation d'approximation [Mycroft et Jones 86]
2) une fonction d'abstraction [Nielson 82]
3) une fonction de concrétisation [Marriott et Sondergaard 92]
4) une connexion de Galois [Cousot et Cousot 77]

En fait, une fonction d'abstraction (resp. une fonction de concrétisation) est une relation d'approximation particulière, et une connexion de Galois est la combinaison d'une fonction d'abstraction et d'une fonction de concrétisation. Le lien de base est donc la relation d'approximation.

2 Approximation d'une sémantique

2.1 Relation d'approximation ξ

Considérons pour la suite une sémantique concrète caractérisée par une application F^{Δ} (le calcul concret) définie d'un ensemble D^{Δ} (le domaine concret) vers lui-même :

$$D^{\Delta} \xrightarrow{F^{\Delta}} D^{\Delta}$$

ainsi qu'une sémantique abstraite caractérisée par une application F^{∇} (le calcul abstrait) définie d'un ensemble D^{∇} (le domaine abstrait) vers lui-même :

$$D^{\nabla} \xrightarrow{F^{\nabla}} D^{\nabla}$$

Tout objet lié annoté par le symbole $^{\Delta}$ fait partie de l'univers concret et tout objet annoté par le symbole $^{\nabla}$ fait partie de l'univers abstrait. Une relation d'approximation ξ définie entre le domaine concret D^{Δ} et le domaine abstrait D^{∇} est le lien le plus faible pouvant être établi entre la sémantique concrète et la sémantique abstraite. $\xi(d^{\Delta},d^{\nabla})$ (on utilise ici une notation fonctionnelle) indique alors que d^{∇} est une approximation de d^{Δ}. On dira aussi que d^{Δ} est approchée par d^{∇} et que d^{∇} approche d^{Δ}.

Définition 3.1

$\xi \in \wp(D^{\Delta} \times D^{\nabla})$ est une relation d'approximation.

Il est à noter que la relation d'approximation ξ peut être définie de manière équivalente par deux applications. La première, notée ass_{α} et appelée association abstraite, associe à toute donnée concrète d^{Δ} l'ensemble des données abstraites approchant d^{Δ}. La seconde, notée ass_{γ} et appelée association concrète, associe à toute donnée abstraite d^{∇} l'ensemble des données concrètes approchées par d^{∇}. Ces deux applications correspondent respectivement aux 'abstraction functor' et 'concretisation functor' de [Marriott 93].

Définition 3.2

L'application ass_{α} est définie de D^{Δ} vers $\wp(D^{\nabla})$ par :

$\forall\, d^{\Delta} \in D^{\Delta},\ ass_{\alpha}(d^{\Delta}) = \{d^{\nabla} \in D^{\nabla} \mid \xi(d^{\Delta},d^{\nabla})\}$.

Définition 3.3

L'application ass_{γ} est définie de D^{∇} vers $\wp(D^{\Delta})$ par :

$\forall\, d^{\nabla} \in D^{\nabla},\ ass_{\gamma}(d^{\nabla}) = \{d^{\Delta} \in D^{\Delta} \mid \xi(d^{\Delta},d^{\nabla})\}$.

Le rapport entre ξ, ass_{α} et ass_{γ} est clairement établi par la propriété suivante.

Propriété 3.4

$\forall\, d^{\Delta} \in D^{\Delta},\ \forall\, d^{\nabla} \in D^{\nabla},\ \xi(d^{\Delta},d^{\nabla}) \Leftrightarrow d^{\Delta} \in ass_{\gamma}(d^{\nabla}) \Leftrightarrow d^{\nabla} \in ass_{\alpha}(d^{\Delta})$

Suivant [Nielson 82] et [Marriot et Sondergaard 89a], on étend inductivement la relation ξ sur différents domaines (ensembles) en considérant le passage

1) aux parties,
2) au produit cartésien,
3) à l'espace des fonctions.

On note $\xi_{A^{\Delta} \times A^{\nabla}}$ la relation ξ définie entre un ensemble A^{Δ} et un ensemble A^{∇}. Toutefois, de manière à ne pas alourdir les notations, nous ne précisons pas le domaine sur lequel est définie la relation ξ (le contexte suffit généralement à déterminer la relation).

Définition 3.5

Si ξ est définie sur $A^\Delta \times A^\nabla$ et $B^\Delta \times B^\nabla$ alors :

- ξ est définie sur $\wp(A^\Delta) \times \wp(A^\nabla)$ par :
  $\xi(S^\Delta, S^\nabla)$ ssi $\forall\, a^\Delta \in S^\Delta$, $\exists\, a^\nabla \in S^\nabla$ tel que $\xi(a^\Delta, a^\nabla)$
- ξ est définie sur $(A^\Delta \times B^\Delta) \times (A^\nabla \times B^\nabla)$ par :
  $\xi((a^\Delta, b^\Delta), (a^\nabla, b^\nabla))$ ssi $\xi(a^\Delta, a^\nabla)$ et $\xi(b^\Delta, b^\nabla)$
- ξ est définie sur $(A^\Delta \to B^\Delta) \times (A^\nabla \to B^\nabla)$ par :
  $\xi(f^\Delta, f^\nabla)$ ssi $\forall\, (a^\Delta, a^\nabla) \in A^\Delta \times A^\nabla$, $\xi(a^\Delta, a^\nabla) \Rightarrow \xi(f^\Delta(a^\Delta) f^\nabla(a^\nabla))$

On dit qu'une sémantique abstraite approche une sémantique concrète via une relation d'approximation ξ ssi ξ est stable par rapport au calcul de chaque sémantique, i.e., ssi $\xi(F^\Delta, F^\nabla)$ (ssi d'après la définition : $\forall\, d^\Delta \in D^\Delta$, $\forall\, d^\nabla \in D^\nabla$, $\xi(d^\Delta, d^\nabla) \Rightarrow \xi(F^\Delta(d^\Delta), F^\nabla(d^\nabla))$). On dit que F^∇ approche F^Δ via ξ. Ceci est illustré par la figure 3.6.

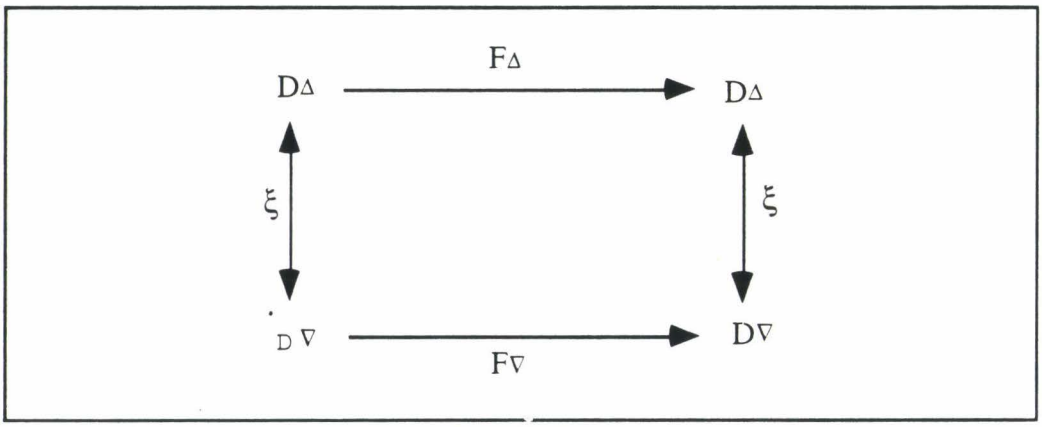

Figure 3.6

Prenons quelques exemples, le premier est incontournable, il s'agit de la "fameuse" règle des signes, illustrée ici par l'addition. Le second traite de l'abstraction de l'intersection sur des parties réelles à l'aide d'intervalles.

Exemple 1

$\oplus$	$-$	0	$+$	$\top$
$-$	$-$	$-$	$\top$	$\top$
0	$-$	0	$+$	$\top$
$+$	$\top$	$+$	$+$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Table 3.7

Considérons comme sémantique concrète l'addition + sur $\mathbb{R}$ et comme sémantique abstraite la règle des signes $\oplus$ sur l'ensemble $\{-,0,+,\top\}$ définie par la table 3.7.

Il est à noter que l'élément $\top$ correspond à l'absence totale d'information sur une donnée et est indispensable à la définition du calcul abstrait $\oplus$, puisque sans cela il serait impossible de déterminer le résultat de $+ \oplus -$.

La relation d'approximation ξ est donnée ici par la fonction d'association concrète ass_γ :

$ass_\gamma(-) =]-\infty,0[$
$ass_\gamma(0) = \{0\}$
$ass_\gamma(+) =]0,+\infty[$
$ass_\gamma(\top) =]-\infty,+\infty[$

$\oplus$ approche $+$ via ξ. Cela signifie que si un couple de réels (x,y) est approchée par un couple de signes (sg,sg') alors le résultat de $x + y$ est approchée par le résultat de $sg \oplus sg'$. Par exemple, (5,3) est approchée par (+,+) et $5 + 3 = 8$ est approchée par $+ \oplus + = +$. Il est à noter que plusieurs abstractions sont possibles pour une même donnée puisque (5,3) est également approchée par (+,$\top$) ; dans ce cas précis le résultat abstrait $+ \oplus \top = \top$ est plus approximatif. ❑

Exemple 2

Considérons comme sémantique concrète l'intersection $\cap$ sur $\wp(\mathbb{R})$ et comme sémantique abstraite l'intersection $\sqcap$ sur l'ensemble noté *Int* des intervalles suivant :

$$Int = \{\bot\} \cup \{[l,u] : l \in \mathbb{R} \cup \{-\infty\}, u \in \mathbb{R} \cup \{+\infty\} \text{ et } l \leq u\}$$

L'intersection $\sqcap$ sur *Int* est définie par :

- $I \sqcap \bot = \bot$
- $\bot \sqcap I = \bot$
- $[l,u] \sqcap [l',u'] = \bot$ si $\max(l,l') > \min(u,u')$
- $[l,u] \sqcap [l',u'] = [\max(l,l'),\min(u,u')]$ si $\max(l,l') \leq \min(u,u')$

La relation d'approximation ξ est définie par :

$\forall X \in \wp(\mathbb{R}), \forall I \in Int, \xi(X,I)$ ssi
- soit $X = \emptyset$
- soit $X \neq \emptyset$, $I = [l,u]$ et $l \leq \min(X) \leq \max(X) \leq u$.

$\sqcap$ approche $\cap$ via ξ. Par exemple, ($\{0,-2,6\},\{0,8\}$) est approchée par ([-2,6],[0,8]) et $\{0,-2,6\} \cap \{0,8\} = \{0\}$ est approchée par $[-2,6] \sqcap [0,8] = [0,6]$. ❑

Définition 3.8

Une interprétation abstraite est un triplet $(F^{\Delta},F^{\nabla},\xi)$ tel que $\xi(F^{\Delta},F^{\nabla})$ sachant que F^{Δ} désigne une sémantique concrète, F^{∇} désigne une sémantique abstraite et ξ désigne une relation d'approximation.

La présentation d'une interprétation abstraite ne s'arrête pas là car la relation d'approximation ξ peut être caractérisée de diverses manières. Cela permet généralement de guider la preuve de la correction de l'approximation considérée. Nous étudions ces propriétés en suivant essentiellement l'approche de [Cousot et Cousot 92c].

2.2 Fonction d'abstraction α

Une propriété naturelle est l'existence pour toute donnée concrète d^{Δ} d'une donnée abstraite approchant d^{Δ}. Cette propriété permet de simuler tout calcul concret par un calcul abstrait.

Propriété 3.9

(1) $\forall\, d^{\Delta} \in D^{\Delta}, \exists\, d^{\nabla} \in D^{\nabla} : \xi(d^{\Delta},d^{\nabla})$

Exemples 1 et 2 (poursuivis)

Cette propriété est vérifiée par les deux exemples considérés ci-dessus. En effet, pour l'exemple 1, quelque soit le réel x, on a $\xi(x,\top)$, et pour l'exemple 2, quelque soit l'ensemble X, on a $\xi(X,[-\infty,+\infty])$. ❑

Il est également assez naturel de considérer une relation $\leq^{\nabla}$ de préordre sur D^{∇} qui soit cohérente avec la relation d'approximation ξ, i.e., telle que si d^{∇} est une approximation de d^{Δ}, alors toute contrainte $d'^{\nabla} \geq d^{\nabla}$ est également une approximation de d^{Δ}.

Propriété 3.10

D^{∇} est munie d'un pré-ordre $\leq^{\nabla}$ cohérent avec ξ, i.e., une relation telle que :

(2) $\forall\, d^{\Delta} \in D^{\Delta}, \forall\, (d^{\nabla},d'^{\nabla}) \in D^{\nabla 2}, \xi(d^{\Delta},d^{\nabla}) \wedge d^{\nabla} \leq^{\nabla} d'^{\nabla} \Rightarrow \xi(d^{\Delta},d'^{\nabla})$

En fait, la relation d'approximation induit de façon naturelle une telle relation de préordre. Il suffit de comparer les images des éléments de D^{∇} par ass_{γ}.

Propriété 3.11

ξ induit sur D^{∇} une relation $\leq^{\nabla}$ de préordre cohérente avec ξ et définie par :

$\forall\, (d^{\nabla},d'^{\nabla}) \in D^{\nabla 2}, d^{\nabla} \leq^{\nabla} d'^{\nabla}$ ssi $ass_{\gamma}(d^{\nabla}) \subseteq ass_{\gamma}(d'^{\nabla})$

Exemple 1 (poursuivi)

On sait qu'en plus des cas réflexifs triviaux, on a :

- $ass_\gamma(-) \subseteq ass_\gamma(\top)$
- $ass_\gamma(0) \subseteq ass_\gamma(\top)$
- $ass_\gamma(+) \subseteq ass_\gamma(\top)$

ξ induit donc la relation $\leq^\nabla$ d'ordre partiel suivante sur $\{-,0,+,\top\}$ (les cas réflexifs sont omis) :

- $- \leq^\nabla \top$
- $0 \leq^\nabla \top$
- $+ \leq^\nabla \top$ ❑

Exemple 2 (poursuivi)

On sait que

- $ass_\gamma(\bot) \subseteq ass_\gamma(I)$, $\forall\ I \in Int$
- $ass_\gamma([l,u]) \subseteq ass_\gamma([l',u'])$, $\forall\ [l,u] \in Int$, $\forall\ [l',u'] \in Int : l' \leq l \leq u \leq u'$

ξ induit donc la relation $\leq^\nabla$ d'ordre partiel suivante sur *Int* :

- $\bot \leq^\nabla I$, $\forall\ I \in Int$
- $[l,u] \leq^\nabla [l',u']$, $\forall\ [l,u] \in Int$, $\forall\ [l',u'] \in Int$ avec $l' \leq l \leq u \leq u'$ ❑

Enfin, lorsque D^∇ est munie d'une relation $\leq^\nabla$ de préordre cohérente avec ξ il est intéressant de constater pour toute donnée concrète d^Δ l'existence d'une plus petite donnée abstraite d^∇ (modulo la relation d'équivalence engendrée par $\leq^\nabla$) approchant d^Δ (lorsqu'il en existe une).

Propriété 3.12

(3) $\forall\ d^\Delta \in D^\Delta$,
$ass_\alpha(d^\Delta) \neq \emptyset \Rightarrow \exists\ d^\nabla \in ass_\alpha(d^\Delta) : \forall\ d^{\nabla\prime} \in ass_\alpha(d^\Delta) : d^\nabla \leq^\nabla d^{\nabla\prime}$

Exemple 1 (poursuivi)

Cette propriété est vérifiée car :

$\forall\ x \in \mathbb{R}$,
- si $x \in\ ]-\infty,0[$, $ass_\alpha(x) = \{-,\top\}$ et $- \leq^\nabla \top$
- si $x = 0$, $ass_\alpha(0) = \{0,\top\}$ et $0 \leq^\nabla \top$
- si $x \in\ ]0,+\infty[$, $ass_\alpha(x) = \{+,\top\}$ et $+ \leq^\nabla \top$ ❑

Exemple 2 (poursuivi)

Cette propriété est vérifiée car :

$\forall X \in \wp(\mathbb{R})$,
- si $X = \emptyset$, $\bot \in ass_\alpha(X)$ et $\forall I \in ass_\alpha(X)$, $\bot \leq^\nabla I$
- si $X \neq \emptyset$, $[\min(X),\max(X)] \in ass_\alpha(X)$ et $\forall I \in ass_\alpha(X)$, $[\min(X),\max(X)] \leq^\nabla I$ ❑

On peut alors caractériser une relation d'approximation ξ vérifiant les propriétés (1), (2) et (3) par une fonction (totale) d'abstraction α.

Propriété 3.13

Si ξ vérifie les propriétés (1), (2) et (3), alors il existe une application α définie de D^Δ sur D^∇ et appelée fonction d'abstraction, telle que :
(4) $\forall d^\Delta \in D^\Delta$, $\alpha(d^\Delta) = d^\nabla : d^\nabla \in ass_\alpha(d^\Delta)$ et $\forall d^{\nabla'} \in ass_\alpha(d^\Delta)$, $d^\nabla \leq^\nabla d^{\nabla'}$.
On dit que ξ est caractérisée par α.

Il existe un rapport d'équivalence entre ξ et α (une relation $\leq^\nabla$ de préordre cohérente avec ξ étant sous-entendue par α).

Propriété 3.14

Si ξ est caractérisée par une fonction d'abstraction α alors :
(5) $\forall d^\Delta \in D^\Delta$, $\forall d^\nabla \in D^\nabla$, $\xi(d^\Delta,d^\nabla)$ ssi $\alpha(d^\Delta) \leq^\nabla d^\nabla$.

Exemple 1 (poursuivi)

α est définie par :

$\forall x \in \mathbb{R}$,
- $\alpha(x) = -$ ssi $x \in]-\infty,0[$
- $\alpha(x) = 0$ ssi $x = 0$
- $\alpha(x) = +$ ssi $x \in]0,+\infty[$ ❑

Exemple 2 (poursuivi)

α est définie par :

$\forall X \in \wp(\mathbb{R})$,
- $\alpha(X) = \bot$ ssi $X = \emptyset$
- $\alpha(X) = [\min(X),\max(X)]$ ssi $X \neq \emptyset$ ❑

Lorsque ξ est caractérisée par une fonction d'abstraction α, il est possible de se limiter à une approximation partielle en ne s'occupant que des éléments mis en relation par α. Cela revient à définir une nouvelle relation (d'approximation).

Définition 3.15

Si ξ est caractérisée par une fonction d'abstraction α, alors la restriction de ξ sur α, notée $\xi_{|\alpha}$, est définie par :

$\forall\ d^{\Delta} \in D^{\Delta}, \forall\ d^{\nabla} \in D^{\nabla}, \xi_{|\alpha}(d^{\Delta},d^{\nabla})$ ssi $d^{\nabla} = \alpha(d^{\Delta})$

2.3 Fonction de concrétisation γ

Les propriétés duales de celles que nous venons d'étudier sont maintenant à prendre en compte. A savoir, tout d'abord l'existence pour toute donnée abstraite d^{∇} d'une donnée concrète approchant d^{∇}.

Propriété 3.16

(1') $\forall\ d^{\nabla} \in D^{\nabla}, \exists\ d^{\Delta} \in D^{\Delta} \mid \xi(d^{\Delta},d^{\nabla})$

Exemples 1 et 2 (poursuivi)

Pour l'exemple 1, cette propriété est vérifiée car quelque soit le signe s, on a $ass_{\gamma}(s) \neq \emptyset$. Pour l'exemple 2, il en est de même puisque quelque soit l'intervalle I, on a $\xi(\emptyset,I)$. ❑

Ensuite, on peut considérer une relation $\leq^{\Delta}$ de préordre sur D^{Δ} qui soit cohérente avec la relation d'approximation, i.e., telle que si d^{Δ} est approchée par d^{∇}, alors toute contrainte $d'^{\Delta} \leq d^{\Delta}$ est également approchée par d^{∇}.

Propriété 3.17

D^{Δ} est munie d'un préordre $\leq^{\Delta}$ cohérent avec ξ, i.e., une relation telle que :

(2') $\forall\ d^{\nabla} \in D^{\nabla}, \forall\ (d^{\Delta},d^{\Delta'}) \in D^{\Delta 2}, \xi(d^{\Delta},d^{\nabla}) \wedge d^{\Delta'} \leq^{\Delta} d^{\Delta} \Rightarrow \xi(d^{\Delta'},d^{\nabla})$

ξ induit de façon naturelle une telle relation de préordre. Il suffit de comparer les images des éléments de D^{Δ} par ass_{α}.

Propriété 3.18

ξ induit sur D^{Δ} une relation $\leq^{\Delta}$ de préordre cohérente avec ξ et définie par :

$\forall\ (d^{\Delta},d^{\Delta'}) \in D^{\Delta 2}$, $d^{\Delta} \leq^{\Delta} d^{\Delta'}$ ssi $ass_{\alpha}(d^{\Delta}) \subseteq ass_{\alpha}(d^{\Delta'})$

Exemple 1 (poursuivi)

La relation d'ordre (usuelle) $\leq$ sur les réels n'est pas cohérente avec ξ puisque $\xi(-3,-)$ et $-3 \leq 5$ n'entraîne pas $\xi(5,-)$. Par contre, ξ induit sur D^{Δ} la relation $\leq^{\Delta}$ de préordre suivante :

$\forall\ (x,y) \in \mathbb{R}^2$,

- si $x \in\ ]-\infty,0[$ et $y \in\ ]-\infty,0[$ alors $x \leq^{\Delta} y$
- si $x \in\ ]0,+\infty[$ et $y \in\ ]0,+\infty[$ alors $x \leq^{\Delta} y$

Cette relation met en relief trois classes d'équivalences (modulo la relation d'équivalence engendrée par $\leq^{\Delta}$) : $]-\infty,0[$, $\{0\}$ et $]0,+\infty[$. ❑

Exemple 2 (poursuivi)

La relation d'ordre $\subseteq$ sur les ensembles de réels est cohérente avec ξ. Il est à noter que $\subseteq$ ne correspond pas à la relation de préordre induite par ξ. ❑

Pour finir, lorsque D^{Δ} est munie d'une relation $\leq^{\Delta}$ de préordre cohérente avec ξ, on peut énoncer la propriété rapportant pour toute donnée abstraite d^{∇} l'existence d'une plus grande donnée concrète d^{Δ} (modulo la relation d'équivalence engendrée par $\leq^{\Delta}$) approchée par d^{∇} (lorsqu'il en existe une).

Propriété 3.19

(3') $\forall\, d^{\nabla} \in D^{\nabla}$,
$ass_{\gamma}(d^{\nabla}) \neq \emptyset \Rightarrow \exists\, d^{\Delta} \in ass_{\gamma}(d^{\nabla}) : \forall\, d^{\Delta'} \in ass_{\gamma}(d^{\nabla}) : d^{\Delta'} \leq^{\Delta} d^{\Delta}$

Exemple 1 (poursuivi)

La propriété n'est pas vérifiée car $ass_{\gamma}(\top) =]-\infty,+\infty[$ et il n'existe pas de plus petit élément dans $ass_{\gamma}(\top)$ (pour $\leq^{\Delta}$). ❑

Exemple 2 (poursuivi)

Cette propriété est vérifiée car :

$\forall\, I \in Int$
- si $I = \bot$, $ass_{\gamma}(I) = \{\emptyset\}$
- si $I = [l,u]$,
  - $\{x \mid l \leq x \leq u\} \in ass_{\gamma}(I)$
  - $\forall\, X \in ass_{\gamma}(I)$, $X \subseteq \{x \mid l \leq x \leq u\}$ ❑

On peut caractériser une relation d'approximation ξ vérifiant les propriétés ci-dessus par une fonction (totale) de concrétisation γ.

Définition 3.20

Si ξ vérifie les propriétés (1'), (2') et (3'), alors il existe une application $\gamma \in D^{\nabla} \rightarrow D^{\Delta}$, appelée fonction de concrétisation, telle que :
(4') $\forall\, d^{\nabla} \in D^{\nabla}$, $\gamma(d^{\nabla}) = d^{\Delta} \mid d^{\Delta} \in ass_{\gamma}(d^{\nabla})$ et $\forall\, d^{\Delta'} \in ass_{\gamma}(d^{\nabla})$, $d^{\Delta'} \leq^{\Delta} d^{\Delta}$
On dit que ξ est caractérisée par γ.

Il existe un rapport d'équivalence entre ξ et γ (une relation de préordre $\leq^{\Delta}$ cohérente avec ξ étant sous-entendue par γ).

Propriété 3.21

Si ξ est caractérisée par une fonction de concrétisation γ alors :

(5') $\forall\, d^{\Delta} \in D^{\Delta}, \forall\, d^{\nabla} \in D^{\nabla}, \xi(d^{\Delta},d^{\nabla})$ ssi $d^{\Delta} \leq^{\Delta} \gamma(d^{\nabla})$.

Exemple 2 (poursuivi)

γ est définie par :

- $\gamma(\perp) = \{\emptyset\}$
- $\gamma([l,u]) = \{x \mid l \leq x \leq u\}$ □

Lorsque ξ est caractérisée par une fonction de concrétisation γ, il est possible de se limiter à une approximation partielle en ne se préoccupant que des éléments mis en relation par γ. Cela revient à définir une nouvelle relation (d'approximation).

Définition 3.22

Si ξ est caractérisée par une fonction de concrétisation γ, alors la restriction de ξ sur γ, notée $\xi_{|\gamma}$, est définie par :

$\forall\, d^{\Delta} \in D^{\Delta}, \forall\, d^{\nabla} \in D^{\nabla}, \xi_{|\gamma}(d^{\Delta},d^{\nabla})$ ssi $d^{\Delta} = \gamma(d^{\nabla})$

2.4 Connexion de Galois (α,γ)

Finalement, en combinant fonction d'abstraction et fonction de concrétisation, on obtient une connexion de Galois.

Propriété 3.23

Si ξ est caractérisée à la fois par une fonction d'abstraction α et par une fonction de concrétisation γ alors (α,γ) forme une connexion de Galois puisque :

(6) $\forall\, d^{\Delta} \in D^{\Delta}, \forall\, d^{\nabla} \in D^{\nabla}, \xi(d^{\Delta},d^{\nabla})$ ssi $\alpha(d^{\Delta}) \leq^{\nabla} d^{\nabla}$ ssi $d^{\Delta} \leq^{\Delta} \gamma(d^{\nabla})$.

On dit que ξ est caractérisée par (α,γ).

A l'annexe A, il est indiqué que les connexions de Galois présentent quelques bonnes propriétés. Nous utilisons celles-ci afin de préciser les conditions suffisantes à l'obtention d'une interprétation abstraite. En utilisant la propriété (6), on établit quelques équivalences.

Propriété 3.24

Si ξ est caractérisée par une connexion de Galois (α,γ) alors les équivalences suivantes sont vérifiées :

- $F^{\Delta} \leq^{\Delta} \gamma \circ F^{\nabla} \circ \alpha$ ssi $\alpha \circ F^{\Delta} \leq^{\nabla} F^{\nabla} \circ \alpha$
- $\alpha \circ F^{\Delta} \circ \gamma \leq^{\nabla} F^{\nabla}$ ssi $F^{\Delta} \circ \gamma \leq^{\Delta} \gamma \circ F^{\nabla}$

Celles-ci coïncident lorsque F^{Δ} et F^{∇} sont deux applications monotones.

Propriété 3.25

Si ξ est caractérisée par une connexion de Galois (α,γ) et si F^{Δ} et F^{∇} sont deux applications monotones alors :

$F^{\Delta} \leq^{\Delta} \gamma o F^{\nabla} o \alpha$ ssi $\alpha o F^{\Delta} o \gamma \leq^{\nabla} F^{\nabla}$.

Preuve

$F^{\Delta} \leq^{\Delta} \gamma o F^{\nabla} o \alpha \Rightarrow \alpha o F^{\Delta} \leq^{\nabla} F^{\nabla} o \alpha \Rightarrow \alpha o F^{\Delta} o \gamma \leq^{\nabla} F^{\nabla} o \alpha o \gamma \Rightarrow \alpha o F^{\Delta} o \gamma \leq^{\nabla} F^{\nabla}$ car $\alpha o \gamma \leq^{\nabla} I\delta$ et F^{∇} est monotone.

$\alpha o F^{\Delta} o \gamma \leq^{\nabla} F^{\nabla} \Rightarrow F^{\Delta} o \gamma \leq^{\Delta} \gamma o F^{\nabla} \Rightarrow F^{\Delta} o \gamma o \alpha \leq^{\Delta} \gamma o F^{\nabla} o \alpha \Rightarrow F^{\Delta} \leq^{\Delta} \gamma o F^{\nabla} o \alpha$ car $I\delta \leq^{\Delta} \gamma o \alpha$ et F^{Δ} est monotone. ❑

La monotonie du calcul concret ou du calcul abstrait permet de reformuler l'approximation de la sémantique concrète par la sémantique abstraite.

Propriété 3.26 (figure 3.27)

Si ξ est caractérisée par une connexion de Galois (α,γ) et si F^{∇} est une application monotone alors : $\xi(F^{\Delta},F^{\nabla})$ ssi $F^{\Delta} \leq^{\Delta} \gamma o F^{\nabla} o \alpha$.

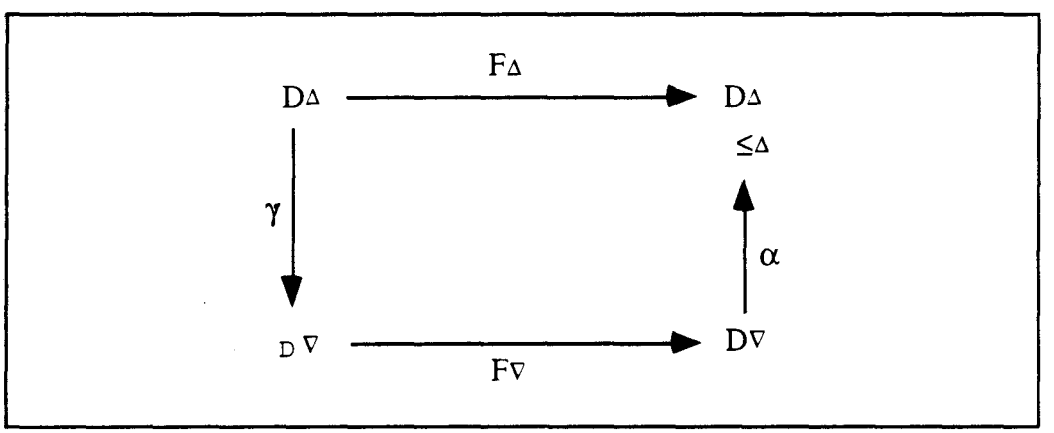

Figure 3.27

Preuve

a) $\Rightarrow$

$\forall\, d^{\Delta} \in D^{\Delta}$, on sait que $\xi(d^{\Delta},\alpha(d^{\Delta}))$ et que $\xi(d^{\Delta},\alpha(d^{\Delta})) \Rightarrow \xi(F^{\Delta}(d^{\Delta}),F^{\nabla}o\alpha(d^{\Delta}))$ puisque $\xi(F^{\Delta},F^{\nabla})$. $\xi(F^{\Delta}(d^{\Delta}),F^{\nabla}o\alpha(d^{\Delta})) \Rightarrow F^{\Delta}(d^{\Delta}) \leq^{\Delta} \gamma o F^{\nabla} o \alpha(d^{\Delta})$.

b) $\Leftarrow$

$\forall\, d^{\Delta} \in D^{\Delta}$, $\forall\, d^{\nabla} \in D^{\nabla} \mid \xi(d^{\Delta},d^{\nabla})$, on sait que $\xi(d^{\Delta},d^{\nabla})$ ssi $\alpha(d^{\Delta}) \leq^{\nabla} d^{\nabla}$.

$\alpha(d^{\Delta}) \leq^{\nabla} d^{\nabla} \Rightarrow F^{\nabla}\alpha(d^{\Delta}) \leq^{\nabla} F^{\nabla}(d^{\nabla})$ puisque F^{∇} est monotone. On sait que $F^{\Delta} \leq^{\Delta} \gamma o F^{\nabla} o \alpha$ ssi $\alpha o F^{\Delta} \leq^{\nabla} F^{\nabla} o \alpha$. De ce fait $\alpha o F^{\Delta}(d^{\Delta}) \leq^{\nabla} F^{\nabla} o \alpha(d^{\Delta})$ et par transitivité $\alpha o F^{\Delta}(d^{\Delta}) \leq^{\nabla} F^{\nabla}(d^{\nabla})$, ce qui est équivalent à $\xi(F^{\Delta}(d^{\Delta}),F^{\nabla}(d^{\nabla}))$. ❑

Propriété 3.28 (figure 3.29)
Si ξ est caractérisée par une connexion de Galois (α,γ) et si F^{Δ} est une application monotone alors : $\xi(F^{\Delta},F^{\nabla})$ ssi $\alpha o F^{\Delta} o \gamma \leq^{\nabla} F^{\nabla}$.

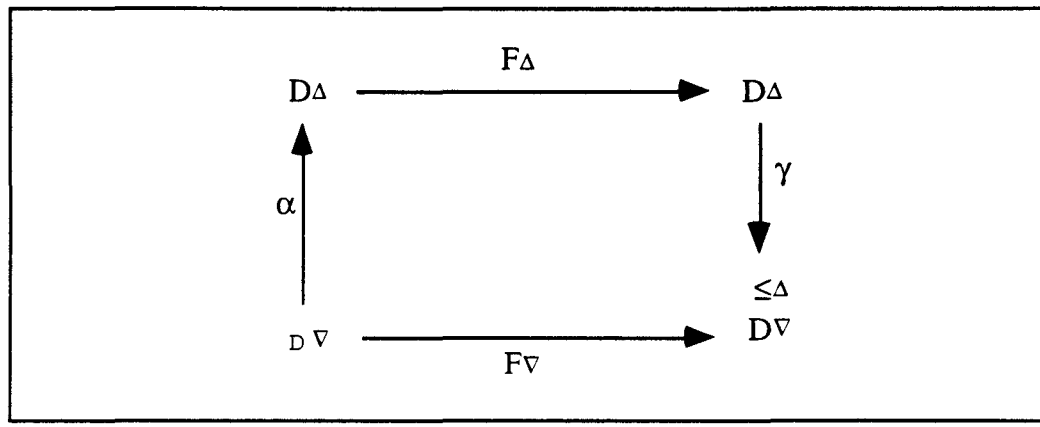

Figure 3.29

La preuve est similaire à la précédente. Finalement, il est possible d'affaiblir les propriétés précédentes en considérant des approximations partielles .

Propriété 3.30
Si ξ est caractérisée par une connexion de Galois (α,γ) alors
$\xi_{|\alpha}(F^{\Delta},F^{\nabla})$ ssi $F^{\Delta} \leq^{\Delta} \gamma o F^{\nabla} o \alpha$.
$\xi_{|\gamma}(F^{\Delta},F^{\nabla})$ ssi $\alpha o F^{\Delta} o \gamma \leq^{\nabla} F^{\nabla}$.

Preuve
$\forall\, d^{\Delta} \in D^{\Delta}$, $\xi(F^{\Delta}(d^{\Delta}),F^{\nabla} o \alpha(d^{\Delta})) \Leftrightarrow F^{\Delta}(d^{\Delta}) \leq^{\Delta} \gamma o F^{\nabla} o \alpha(d^{\Delta})$.
$\forall\, d^{\nabla} \in D^{\nabla}$, $\xi(F^{\Delta} o \gamma(d^{\nabla}),F^{\nabla}(d^{\nabla})) \Leftrightarrow \alpha o F^{\Delta} o \gamma(d^{\Delta}) \leq^{\nabla} F^{\nabla}(d^{\nabla})$. □

3 Approximation d'une sémantique de point fixe

3.1 Relation d'approximation ξ

Lorsque des sémantiques de point fixe sont considérées, il est possible de préciser les conditions d'une approximation. Nous commençons par donner la description d'une sémantique de point fixe, suivant [Cousot et Cousot 92c] (de nombreuses variantes existent, notamment avec des hypothèses plus faibles).

Définition 3.31
Une sémantique de point fixe est une sémantique qui est :

- basée sur le calcul de point fixe (i.e., basée sur le calcul d'une séquence itérative et stationnaire) d'une fonction (totale) $f : D \rightarrow D$

qui utilise un opérateur (fonction totale) de passage à la limite $\cup$: $\wp(D) \rightarrow D$.

- la séquence itérative de f est définie par : $\forall\, d \in D$,

$$f^0(d) = d$$
$$f^\lambda(d) = f(f^{\lambda-1}(d)) \quad \forall\, \lambda \in Ord^{suc}$$
$$f^\lambda(d) = \bigcup_{\beta<\lambda} f^\beta(d) \quad \forall\, \lambda \in Ord^{lim}$$

- cette séquence converge : $\forall\, d \in D$,

$$\exists\, \lambda \in Ord \mid \forall\, \lambda' \in Ord : \lambda' > \lambda \Rightarrow f^{\lambda'}(d) = f^\lambda(d) = lfp_d\, f$$

- définie par l'application $F : D \rightarrow D$ telle que :
  $\forall\, d \in D, F(d) = lfp_d\, f$

Maintenant que la description d'une sémantique de point fixe est effectuée, nous considérons données pour la suite :

- une sémantique concrète de point fixe :

$$D^\Delta \xrightarrow{F^\Delta} D^\Delta$$

- une sémantique abstraite de point fixe :

$$D^\nabla \xrightarrow{F^\nabla} D^\nabla$$

- une relation d'approximation ξ définie entre D^Δ et D^∇.

Dans ce cas particulier où sémantiques concrètes et sémantiques abstraites sont des sémantiques de point fixe, nous reformulons l'approximation de F^Δ par F^∇ via ξ. En effet, pour prouver que $\xi(F^\Delta,F^\nabla)$, il "suffit" de prouver que $\xi(f^\Delta,f^\nabla)$ et $\xi(\cup^\Delta,\cup^\nabla)$.

Propriété 3.32

Si $\xi(f^\Delta,f^\nabla)$ et $\xi(\cup^\Delta,\cup^\nabla)$ alors,

$\forall\, d^\Delta \in D^\Delta, \forall\, d^\nabla \in D^\nabla : \xi(d^\Delta,d^\nabla) \Rightarrow \xi(lfp_{d^\Delta}\, f^\Delta, lfp_{d^\nabla}\, f^\nabla)$

Preuve

On prouve par récurrence sur les ordinaux que $\forall\, \lambda \in Ord$, $\xi(f^{\Delta\lambda}(d^\Delta),f^{\nabla\lambda}(d^\nabla))$. Si $\lambda = 0$, alors $\xi(f^{\Delta 0}(d^\Delta),f^{\nabla 0}(d^\nabla))$ puisque $\xi(d^\Delta,d^\nabla)$. Si λ est un ordinal successeur alors on suppose que $\forall\, \beta < \lambda$, $\xi(f^{\Delta\beta}(d^\Delta),f^{\nabla\beta}(d^\nabla))$. Comme $\xi(f^\Delta,f^\nabla)$, on sait que $\xi(f^{\Delta\lambda-1}(d^\Delta),f^{\nabla\lambda-1}(d^\nabla))$ implique $\xi(f^\Delta(f^{\Delta\lambda-1}(d^\Delta)),f^\nabla(f^{\nabla\lambda-1}(d^\nabla)))$, et donc on en déduit que $\xi(f^{\Delta\lambda}(d^\Delta),f^{\nabla\lambda}(d^\nabla))$. Si λ est un ordinal limite alors on

suppose que $\forall\ \beta < \lambda$, $\xi(f^{\Delta}\beta(d^{\Delta}),f^{\nabla}\beta(d^{\nabla}))$. Comme $\xi(\cup^{\Delta},\cup^{\nabla})$, on en déduit que $\xi(f^{\Delta}\lambda(d^{\Delta}),f^{\nabla}\lambda(d^{\nabla}))$.

En posant $\lambda \in \text{Ord} \mid f^{\Delta}\lambda(d^{\Delta}) = \text{lfp}_{d\Delta}\ f^{\Delta}$ et $f^{\nabla}\lambda(d^{\nabla}) = \text{lfp}_{d\nabla}\ f^{\nabla}$, on conclut que $\xi(\text{lfp}_{d\Delta}\ f^{\Delta},\text{lfp}_{d\nabla}\ f^{\nabla})$. ❑

Corollaire 3.33

Si $\xi(f^{\Delta},f^{\nabla})$ et $\xi(\cup^{\Delta},\cup^{\nabla})$ alors $\xi(F^{\Delta},F^{\nabla})$.

Le corollaire est immédiat puisque $F^{\Delta}(d^{\Delta}) = \text{lfp}_{d\Delta}\ f^{\Delta}$ et $F^{\nabla}(d^{\nabla}) = \text{lfp}_{d\nabla}\ f^{\nabla}$.

3.2 Opérateurs de Widening et Narrowing

Les opérateurs de widening et de narrowing ont été introduit par [Cousot et Cousot 76, 77], et ont trois utilisations reconnues : la première en tant qu'opérateurs d'extrapolation, la seconde en tant qu'opérateurs assurant l'arrêt d'un calcul, la troisième en tant qu'opérateurs garantissant un arrêt rapide.

Dans ce qui précède, nous avons toujours supposé une sémantique concrète et une sémantique abstraite données et étudié à partir de là les relations possibles entre ces deux sémantiques. Considérons maintenant une situation où la sémantique concrète (de point fixe) est donnée et où seul le domaine abstrait est donné. Dans ce sens, il est possible d'induire une sémantique abstraite (de point fixe) à partir d'une relation d'approximation ξ établie entre le domaine concret et le domaine abstrait et à partir d'opérateurs de widening et de narrowing appliqués à l'extrapolation (voir la proposition 4.5 de [Cousot et Cousot 92c]).

Lorsque le calcul d'une sémantique abstraite subit une explosion exponentielle, ou pire encore ne s'arrête pas, nous ne sommes en réalité guère avancés. Pour imposer l'arrêt ou l'accélération du calcul tout en préservant la correction de l'abstraction on peut alors introduire des opérateurs de widening et de narrowing appliqués à la terminaison (ou accélération). L'opérateur de widening est utilisé pour calculer une approximation finie (via un calcul de point fixe) d'une sémantique concrète et l'opérateur de narrowing est utilisée pour calculer une approximation finie (via un calcul de point fixe) plus fine que la précédente (voir les propositions 6.20 et 6.21 de [Cousot et Cousot 92c]).

Nous donnerons la description des opérateurs de widening et de narrowing au paragraphe suivant, et ceci par rapport à un contexte bien particulier.

4 Conception d'une interprétation abstraite

Partant de la donnée d'une sémantique concrète, la conception d'une interprétation abstraite consiste à établir successivement :

- le domaine abstrait
- la relation d'approximation entre le domaine concret et le domaine abstrait
- le calcul abstrait

Il est possible d'être plus précis, en particulier pour l'élaboration du calcul abstrait. Une bonne démarche à suivre est la suivante : d'abord, induire le calcul abstrait à partir du calcul concret et ensuite, forcer (ou accélérer) la terminaison du calcul abstrait. Induire le calcul abstrait à partir du calcul concret signifie ici : définir le calcul abstrait sur la base du comportement opérationnel du calcul concret. Il y a au moins deux bonnes raisons pour envisager une telle simulation opérationnelle :

- la simplicité
- la précision

En effet, si le calcul abstrait imite de façon opérationnelle le calcul concret, d'une part, il est simple à comprendre et à définir, et d'autre part, il conserve une grande part de l'information opérationnelle du calcul concret. Forcer la terminaison du calcul est rendue possible grâce à l'introduction d'opérateurs de widening. Avec la démarche précédente, on peut considérer qu'une interprétation abstraite se compose de deux phases :

- une phase d'abstraction du domaine
- une phase d'abstraction du calcul

La phase d'abstraction du domaine correspond à définir le domaine abstrait, définir la relation d'approximation et induire le calcul abstrait à partir du calcul concret. La phase d'abstraction du calcul correspond à forcer ou accélérer la terminaison du calcul. On sépare ainsi en quelque sorte l'aspect déclaratif (la manière dont on définit le domaine abstrait) de l'aspect calculatoire (la manière dont on obtient un calcul fini). Cette démarche correspond, pour l'essentiel, à l'approche de l'interprétation abstraite basée sur la combinaison d'une connexion de Galois et d'opérateurs de widening et narrowing [Cousot et Cousot 92b]. L'induction du calcul abstrait est guidée par une connexion de Galois (ou plus simplement une relation d'approximation) et la terminaison ou l'accélération du calcul abstrait est guidée par des opérateurs de widening (et narrowing).

[Cousot et Cousot 92b] ont montré que cette approche est plus générale qu'une simple approche basée sur une connexion de Galois car cela sous-entend le choix d'un domaine abstrait fini (ou satisfaisant la condition de chaîne ascendante) afin d'assurer la terminaison du calcul. En effet, une interprétation abstraite basée sur un domaine infini peut donner des résultats équivalents (à l'aide d'opérateurs de widening et de narrowing) à ceux d'une interprétation abstraite basée sur un domaine fini (ou satisfaisant la condition de chaîne

ascendante). Par contre, le contraire n'est pas vrai, et plus précisément, il existe des situations où :

- Pour tout programme, il existe un treillis fini qui peut être utilisé par rapport à ce programme pour concevoir une interprétation abstraite permettant d'obtenir des résultats équivalents à ceux obtenus par une interprétation abstraite basée sur un domaine infini et des opérateurs de widening et de narrowing.

- Un treillis fini ne convient pas à l'ensemble des programmes.

- Pour l'ensemble des programmes, un nombre infini de valeurs abstraites est nécessaire.

- Pour tout programme, il n'est pas possible de déduire de manière statique (par simple inspection du texte du programme) l'ensemble des valeurs abstraites nécessaires.

Cette approche est également plus générale qu'une simple approche basée sur les opérateurs de widening et de narrowing car cela sous-entend que le domaine abstrait et le domaine concret sont identiques.

En résumé, l'approche de l'interprétation abstraite basée sur la combinaison d'une connexion de Galois (relation d'approximation) et d'opérateurs de widening et narrowing est le résultat de deux phases complémentaires : la première permet de caractériser le comportement d'une famille de calculs concrets par un calcul abstrait (ceci, grâce à l'abstraction du domaine), la seconde permet d'assurer la terminaison ou l'accélération du calcul abstrait (ceci, grâce à l'abstraction du calcul).

A propos de l'induction du calcul abstrait, une hypothèse réaliste consiste à supposer que la relation d'approximation ξ considérée est caractérisée par une connexion de Galois. On obtient alors le théorème (d'induction) suivant qui est une adaptation des propositions 23 de [Cousot et Cousot 92a] et 6.9 de [Cousot et Cousot 92c] :

Théorème 3.34

Soient $(D^{\Delta},\leq^{\Delta})$ et $(D^{\nabla},\leq^{\nabla})$ deux cpos, $F^{\Delta} : D^{\Delta} \rightarrow D^{\Delta}$ une sémantique concrète de point fixe telle que f^{Δ} et $\cup^{\Delta}$ sont monotones et $\xi \in \wp(D^{\Delta}\times D^{\nabla})$ une relation d'approximation caractérisée par une connexion de Galois (α,γ). Soient f^{∇} et $\cup^{\nabla}$ deux fonctions définies par :

$f^{\nabla} = \alpha \circ f^{\Delta} \circ \gamma$

$\cup^{\nabla} = \alpha \circ \cup^{\Delta} \circ \gamma^{*}$

Si f^∇ est extensif alors la séquence itérative abstraite de f^∇ (utilisant $\cup^\nabla$ comme opérateur de passage à la limite) converge. La sémantique abstraite $F^\nabla : D^\nabla \rightarrow D^\nabla$ définie par :

$\forall\, d^\nabla \in D^\nabla,\ F^\nabla(d^\nabla) = \mathrm{lfp}_{d^\nabla} f^\nabla$

est une sémantique de point fixe telle que $\xi(F^\Delta, F^\nabla)$.

Preuve

Comme f^∇ est extensif, on sait que la séquence itérative abstraite de f^∇ est croissante. Or, comme $(D^\nabla, \leq^\nabla)$ est un cpo, la séquence itérative abstraite de f^∇ converge.

Posons $d^\nabla \in D^\nabla$ et $d^\Delta \in D^\Delta$ tel que $d^\Delta = \gamma(d^\nabla)$.

On prouve par récurrence sur les ordinaux que $\forall\, \lambda \in \mathrm{Ord}$, $f^{\Delta\lambda}(d^\Delta) \leq^\Delta \gamma(f^{\nabla\lambda}(d^\nabla))$. Si $\lambda = 0$, alors $f^{\Delta 0}(d^\Delta) \leq^\Delta \gamma(f^{\nabla 0}(d^\nabla))$ puisque $d^\Delta = \gamma(d^\nabla)$. Si λ est un ordinal successeur alors on suppose que $\forall\, \beta < \lambda$, $f^{\Delta\beta}(d^\Delta) \leq^\Delta \gamma(f^{\nabla\beta}(d^\nabla))$. $\gamma(f^\nabla(f^{\nabla\lambda-1}(d^\nabla))) = \gamma \circ \alpha \circ f^\Delta \circ \gamma(f^{\nabla\lambda-1}(d^\nabla))$ et $f^\Delta \circ \gamma(f^{\nabla\lambda-1}(d^\nabla)) \leq^\Delta \gamma \circ \alpha \circ f^\Delta \circ \gamma(f^{\nabla\lambda-1}(d^\nabla))$ puisque $I\delta \leq^\Delta \gamma \circ \alpha$. Comme $f^{\Delta\lambda-1}(d^\Delta) \leq^\Delta \gamma(f^{\nabla\lambda-1}(d^\nabla))$ et que f^Δ est monotone, on en déduit que $f^{\Delta\lambda}(d^\Delta) \leq^\Delta f^\Delta \circ \gamma(f^{\nabla\lambda-1}(d^\nabla))$, puis par transitivité que $f^{\Delta\lambda}(d^\Delta) \leq^\Delta \gamma(f^{\nabla\lambda}(d^\nabla))$. Si λ est un ordinal limite alors on suppose que $\forall\, \beta < \lambda$, $f^{\Delta\beta}(d^\Delta) \leq \gamma(f^{\nabla\beta}(d^\nabla))$. On procède comme ci-dessus en prenant en compte la monotonie de $\cup^\nabla$ (nous ne détaillons pas).

On en déduit que $\xi(\mathrm{lfp}_{d^\Delta} f^\Delta, \mathrm{lfp}_{d^\nabla} f^\nabla)$. ❑

A propos de la terminaison (ou de l'accélération) du calcul abstrait, nous donnons maintenant la description des opérateurs de widening et de narrowing :

Définition 3.35

Soit $(D,\leq)$ un ensemble partiellement ordonné. Un opérateur de widening défini sur D est un opérateur ∇ défini de $D \times D$ sur D tel que :

1 $\forall\, (d,d') \in D^2,\ d' \leq d \nabla d'$

2 $\forall\, (d,d') \in D^2,\ d \leq d \nabla d'$

3 Pour toute chaîne $d_0, \ldots, d_i, \ldots$ d'éléments de D, la chaîne $x_0 = d_0, \ldots, x_i = x_{i-1} \nabla d_i, \ldots$ n'est pas strictement croissante.

Définition 3.36

Soit $(D,\leq)$ un ensemble partiellement ordonné. Un opérateur de narrowing défini sur D est un opérateur Δ défini de $D \times D$ sur D tel que :

1 $\forall\, (d,d') \in D^2,\ \forall\, e \in D,\ e \leq d$ et $e \leq d' \Rightarrow e \leq d \Delta d'$

2 $\forall\, (d,d') \in D^2,\ d \Delta d' \leq d$

3 Pour toute chaîne $d_0, \ldots, d_i, \ldots$ d'éléments de D, la chaîne $x_0 = d_0, \ldots, x_i = x_{i-1}\, \Delta d_i, \ldots$ n'est pas strictement décroissante.

On obtient alors les deux théorèmes suivants qui sont des adaptations des propositions 31 et 30 de [Cousot et Cousot 92a] :

Théorème 3.37

Soient $(D,\leq)$ un ensemble partiellement ordonné, $F : D \rightarrow D$ une sémantique de point fixe telle que f est monotone et ∇ un opérateur de widening défini sur D. Si la condition suivante est vérifiée :

$\forall (d,d') \in D^2, \forall \lambda \in Ord^{lim}, (\forall \beta < \lambda : f^{\beta}(d) \leq d') \Rightarrow f^{\lambda}(d) \leq d'$

alors la séquence itérative abstraite avec widening de la fonction g admet un plus petit point fixe après un nombre fini d'itérations sachant que g est définie par :

$\forall d \in D, g(d) = d \nabla f(d)$

De plus, on a :

$\forall d \in D, lfp_d f \leq lfp_d g$

Preuve

La séquence itérative de g est croissante car par définition de ∇ (condition 2) : $d \leq d \nabla f(d)$ et par définition de g : $d \nabla f(d) = g(d)$. De plus, elle admet après un nombre fini d'itérations un plus petit point fixe $lfp_d g$ car par définition de ∇ (condition 3) : il existe un entier n tel que $g^{n+1}(d) = g^n(d)$.

On prouve par récurrence sur les ordinaux que $\forall \lambda \in Ord, f^{\lambda}(d) \leq lfp_d g$. Si $\lambda = 0$, alors $f^0(d) \leq g^0(d)$ puisque $\leq$ est réflexive et $g^0(d) \leq lfp_d g$ puisque la séquence itérative de g est croissante. Aussi $f^0(d) \leq lfp_d g$. Si λ est un ordinal successeur alors on suppose que $\forall \beta < \lambda$, $f^{\beta}(d) \leq lfp_d g$. $f^{\lambda-1}(d) \leq lfp_d g$ implique $f(f^{\lambda-1}(d)) \leq f(lfp_d g)$ puisque f est monotone. Par définition de ∇ (condition 1) : $f(lfp_d g) \leq lfp_d g \nabla f(lfp_d g)$, or $lfp_d g = lfp_d g \nabla f(lfp_d g)$. On en déduit par transitivité que $f^{\lambda}(d) \leq lfp_d g$. Si λ est un ordinal limite alors on suppose que $\forall \beta < \lambda$, $f^{\beta}(d) \leq lfp_d g$. Par hypothèse, on sait que $f^{\lambda}(d) \leq lfp_d g$.

On a donc prouvé en particulier que $lfp_d f \leq lfp_d g$. ❑

Théorème 3.38

Soient $(D,\leq)$ un ensemble partiellement ordonné, $F : D \rightarrow D$ une sémantique de point fixe telle que f est monotone et Δ un opérateur de narrowing défini sur D. La séquence itérative abstraite avec widening de la fonction h admet un plus petit point fixe après un nombre fini d'itérations sachant que h est définie par :

$h(d) = d \Delta f(d)$

De plus, on a :

$\forall d \in D, lfp_d f \leq lfp_e h \leq lfp_d g$ avec $e = lfp_d g$

Preuve

La séquence itérative de h est décroissante car par définition de Δ (condition 2) : $d\Delta f(d) \leq d$ et par définition de h : $h(d) = d\Delta f(d)$. De plus, elle admet après un nombre fini d'itérations un plus petit point fixe $lfp_e\ h$ car par définition de ∇ (condition 3) : il existe un entier n tel que $h^{n+1}(e) = h^n(e)$.

On prouve par récurrence sur les entiers que $\forall\ n \in \mathbb{N}$, $lfp_d\ f \leq h^n(e)$. Si n = 0, alors comme $lfp_d\ f \leq e$ et $h^0(e) = e$, on a $lfp_d\ f \leq h^0(e)$. Si $\forall\ i < n+1$, $lfp_d\ f \leq h^i(e)$ est vrai, montrons que $lfp_d\ f \leq h^{n+1}(e)$. $lfp_d\ f \leq h^n(e)$ implique $f(lfp_d\ f) \leq f(h^n(e))$ puisque f est monotone. Par définition de ∇, $h^{n+1}(e) = h^n(e)\ \nabla\ f(h^n(e))$. Or comme $f(lfp_d\ f) = lfp_d\ f$, on a $lfp_d\ f \leq h^n(e)$ et $lfp_d\ f \leq f(h^n(e))$. On en déduit par définition de ∇ (condition 1) que $lfp_d\ f \leq h^n(e)\ \nabla\ f(h^n(e))$ et donc que $lfp_d\ f \leq h^{n+1}(e)$. ❑

Exemple

La sémantique statique d'un programme Pascal dont les déclarations sont limitées à une seule variable I désigne l'ensemble des valeurs qui peuvent être affectées à cette variable.

Pour le programme suivant :

```
program p1;
   var I : integer;
begin
   I:=1;
   while I <= 100 do I:=I+2;
end.
```

la sémantique statique est donnée par le plus petit point fixe de la fonction monotone f^Δ définie de $\wp(\mathbb{R})$ vers $\wp(\mathbb{R})$ par :

$\forall\ X \in \wp(\mathbb{R})$, $f^\Delta(X) = \{1\} \cup \{x+2 \mid x \in X \text{ et } x \leq 100\}$

On obtient $f^{\Delta 0}(\emptyset) = \emptyset$, $f^\Delta(\emptyset) = \{1\}$, $f^\Delta(\{1\}) = \{1,3\}$, ..., $f^\Delta(\{1,3...99,101\}) = \{1,3...99,101\}$. On a donc $lfp_\emptyset\ f^\Delta$ qui désigne l'ensemble des entiers positifs impairs compris entre 1 et 101.

Pour le programme suivant :

```
program p2;
   var I : integer;
begin
   I:=1;
   while I >= 1 do I:=I+2;
```

end.

la sémantique statique est donnée par le plus petit point fixe de la fonction monotone f^{Δ} définie de $\wp(\mathbb{R})$ vers $\wp(\mathbb{R})$ par :

$\forall X \in \wp(\mathbb{R}), f^{\Delta}(X) = \{1\} \cup \{x+2 \mid x \in X\}$

On obtient $f^{\Delta 0}(\emptyset) = \emptyset$, $f^{\Delta}(\emptyset) = \{1\}$, $f^{\Delta}(\{1\}) = \{1,3\}$, ..., $f^{\Delta}(\{1,3...99,101\}) = \{1,3...99,101,103\}$, ... On a donc $\mathrm{lfp}_{\emptyset} f^{\Delta}$ qui désigne l'ensemble des entiers positifs impairs supérieurs ou égal à 1.

On considère maintenant le domaine abstrait *Int*, la relation d'approximation ξ et la connexion de Galois (α,γ) définis pour l'exemple 2 du paragraphe 2.1.1. On peut utiliser le théorème d'induction pour définir les sémantiques abstraites qui correspondent aux sémantiques statiques concrètes vues ci-dessus. On pose donc : $f^{\nabla} = \alpha \circ f^{\Delta} \circ \gamma$.

Pour le programme p_1, on obtient $f^{\nabla 0}(\bot) = \bot$, $f^{\nabla}(\bot) = [1,1]$, $f^{\nabla}([1,1]) = [1,3]$, ..., $f^{\nabla}([1,,101]) = [1,101]$. On a $\mathrm{lfp}_{\bot} f^{\nabla}$ qui désigne l'intervalle [1,101]. On a bien $\alpha(\emptyset) \leq^{\nabla} \bot$ et $\alpha(\mathrm{lfp}_{\emptyset} f^{\Delta}) \leq^{\nabla} \mathrm{lfp}_{\bot} f^{\nabla}$.

Pour le programme p_2, on obtient $f^{\nabla 0}(\bot) = \bot$, $f^{\nabla}(\bot) = [1,1]$, $f^{\nabla}([1,1]) = [1,3]$, ..., $f^{\nabla}([1,101]) = [1,103]$, ... On a $\mathrm{lfp}_{\bot} f^{\nabla}$ qui désigne l'intervalle $[1,+\infty]$. On a bien $\alpha(\emptyset) \leq^{\nabla} \bot$ et $\alpha(\mathrm{lfp}_{\emptyset} f^{\Delta}) \leq^{\nabla} \mathrm{lfp}_{\bot} f^{\nabla}$..

Si pour le programme p_1 le calcul de la sémantique abstraite termine, ce n'est pas le cas pour le programme p_2. Il est donc nécessaire d'introduire les opérateurs de widening ∇ et de narrowing Δ proposés par [Cousot et Cousot 76] :

- $\bot \nabla I = \bot$
- $I \nabla \bot = \bot$
- $[l,u] \nabla [l',u'] = [l'',u'']$ avec
  $l'' = -\infty$ si $l > l'$, $l'' = l$ sinon
  $u'' = +\infty$ si $u' > u$, $u'' = u$ sinon

- $\bot \Delta I = \bot$
- $I \Delta \bot = \bot$
- $[l,u] \Delta [l',u'] = [l'',u'']$ avec
  $l'' = l'$ si $l = -\infty$, $l'' = l$ sinon
  $u'' = u'$ si $u = +\infty$, $u'' = u$ sinon

L'opérateur de widening permet d'élargir à l'infini les bornes instables. L'opérateur de narrowing permet de rétrécir les bornes infinies. On considère

maintenant les sémantiques abstraites calculées ci-dessus avec widening et narrowing.

Pour le programme p_1, on construit la séquence itérative suivante avec widening : $g^{\nabla 0}(\bot) = \bot$, $g^{\nabla}(\bot) = \bot \nabla [1,1] = [1,1]$, $g^{\nabla}([1,1]) = [1,1] \nabla [1,3] = [1,+\infty]$, $g^{\nabla}([1,+\infty]) = [1,+\infty] \nabla [1,101] = [1,+\infty]$. On a donc $\mathrm{lfp}_{\bot}\, g^{\nabla} = [1,+\infty]$. Puis, on construit la séquence itérative suivante avec narrowing : $h^{\nabla 0}([1,+\infty]) = [1,+\infty]$, $h^{\nabla}([1,+\infty]) = [1,+\infty] \Delta [1,101] = [1,101]$, $h^{\nabla}([1,101]) = [1,101] \Delta [1,101] = [1,101]$. On a donc $\mathrm{lfpt}\, h^{\nabla} = [1,101]$.

Pour le programme p_2, on construit la séquence itérative suivante avec widening : $g^{\nabla 0}(\bot) = \bot$, $g^{\nabla}(\bot) = \bot \nabla [1,1] = [1,1]$, $g^{\nabla}([1,1]) = [1,1] \nabla [1,3] = [1,+\infty]$, $g^{\nabla}([1,+\infty]) = [1,+\infty] \nabla [1,101] = [1,+\infty]$. On a donc $\mathrm{lfp}_{\bot}\, g^{\nabla} = [1,+\infty]$. Puis, on construit la séquence itérative suivante avec narrowing : $h^{\nabla 0}([1,+\infty]) = [1,+\infty]$, $h^{\nabla}([1,+\infty]) = [1,+\infty] \Delta [1,+\infty] = [1,+\infty]$. On a donc $\mathrm{lfpt}\, h^{\nabla} = [1,+\infty]$.

Dans les deux cas, le même résultat est obtenu par le calcul avec et sans les opérateurs. Ce n'est évidemment pas toujours le cas. ❑

Chapitre 4

Application à la Programmation Logique

De nombreuses travaux ont été consacrés à l'application de l'interprétation abstraite en programmation logique. Il n'est pas question ici de traiter l'ensemble de ces travaux, mais plutôt d'en présenter quelques aspects importants. Il est ainsi possible de distinguer les travaux portant sur la description de modèles d'interprétation abstraite (section 1), de domaines abstraits (section 2) ou d'aspects pratiques (section 3).

1 Modèles d'interprétation abstraite

Trois approches sont principalement proposées dans la littérature [Le Charlier et Van Hentenryck 92b] :

- l'approche opérationnelle
- l'approche (par calcul) de point fixe
- l'approche dénotationnelle

L'approche opérationnelle (section 1.1) consiste à imiter le déroulement du calcul concret avec des données et opérations abstraites. L'imitation peut être plus ou moins proche : les sémantiques abstraites de [Mellish 87] et [Bruynooghe 90] sont plus éloignés de la sémantique opérationnelle initiale que celles de [Kanamori et Kawamura 90] ou [Codognet et Filè 90].

L'approche (par calcul) de point fixe (section 1.2) a été introduite d'un point de vue théorique par [Cousot et Cousot 77] et étudiée d'un point de vue (plus) algorithmique par [Le Charlier et al. 91, Le Charlier et Van Hentenryck 92a]. Le principe de cette approche réside dans une séparation claire et précise des aspects

déclaratifs (quelle est la sémantique abstraite ?) et algorithmiques (comment calculer la sémantique abstraite ?). La sémantique abstraite est définie en terme de point fixe puis calculée de manière indépendante.

L'approche dénotationelle (section 3) est une approche où la sémantique abstraite est définie par des équations sémantiques. [Nielson 82] et [Marriot et Sondergaard 90] proposent l'utilisation d'un méta-langage dans lequel tout programme peut être codé et ensuite interprété. L'intérêt de cette approche est que la théorie de l'interprétation abstraite peut être généralisée dans le cadre de ce méta-langage. [Heintze 92] propose une analyse ensembliste qui peut être considérée d'une certaine manière comme une approche dénotationnelle.

1.1 Approche opérationnelle

L'approche opérationnelle est, semble-t-il, la plus naturelle et a été utilisée très tôt en programmation logique. Nous présentons brièvement dans cette section les travaux de [Mellish 87], [Bruynooghe 90] et [Kanamori et Kawamura 90].

Un des premiers articles conçus sur l'interprétation abstraite en programmation logique (Prolog) est celui de [Mellish 87]. Il s'agit, en fait, dans le cadre formel de l'interprétation abstraite d'une reconsidération de travaux antérieurs (en particulier [Mellish 85]), portant notamment sur :

- le mode d'instantiation des variables au cours de l'exécution
- le partage de structures entre variables instanciées

Pour commencer, C. Mellish introduit une sémantique de traces correspondant à la sémantique opérationnelle d'un interpréteur OLD utilisant une stratégie "en profondeur d'abord". En effet, une trace est 4-uplet (IN,OUT,NUM, SUBTRACES) qui correspond au parcours d'une branche *b* d'un arbre OLD [Mallet 92], et où :

IN désigne un appel atomique (l'étiquette du premier noeud de la branche *b*)

OUT désigne une solution atomique (si la branche *b* est finie alors OUT désigne l'application de la substitution étiquetant la branche *b* sur l'appel atomique IN, sinon OUT désigne le symbole spécial Φ)

NUM désigne le numéro de la clause utilisée pour passer au noeud suivant par rapport à la branche *b*

SUBTRACES désigne la séquence des traces pour tous les sous-buts introduits par la clause sélectionnée

A tout couple constitué d'un programme et d'un but, on peut associer l'ensemble des traces possibles. On définit ainsi une sémantique de traces. Une sémantique d'entrées/sorties est par ailleurs considérée. Elle décrit respectivement les ensembles input et output des appels atomiques et des solutions atomiques apparaissant lors de la résolution. La sémantique d'entrées/sorties ainsi définie est complète vis à vis de la sémantique de traces.

Ensuite, [Mellish 87] propose une version abstraite (et générique) de la sémantique d'entrées/sorties. L'algèbre concrète, i.e., le domaine concret (ensemble de termes) et les opérations élémentaires portant sur cet ensemble, est remplacée par une algèbre abstraite. La complétude d'une sémantique abstraite d'entrées/sorties est établie localement, i.e., par rapport aux opérations élémentaires. De plus, si le domaine abstrait est fini, il est possible de calculer en un temps fini (par point fixe) la sémantique d'entrées/sorties abstraite.

Pour finir, deux applications sont données en exemple : la première porte sur l'inférence de modes et la seconde sur le partage de structures. Intégrées à la compilation, elles permettent de réaliser différentes optimisations. Ces deux exemples sont issus de [Mellish 85], leur validité est ainsi démontrée par rapport au cadre formel de l'interprétation abstraite.

[Bruynooghe et al. 87] et [Bruynooghe 90] décrivent un modèle d'interprétation abstraite appliquée à la programmation logique (Prolog pur + quelques prédicats prédéfinis), et pour lequel des applications non triviales sont données [Bruynooghe et Janssens 88]. Ce modèle est construit autour de la notion d'arbre ET/OU abstrait. Un arbre ET/OU abstrait possède la même quantité d'information opérationnelle qu'un arbre SLD. Seulement, son exploitation est plus simple du fait de sa structure. En effet, il est facile de décorer cet arbre avec des substitutions correspondant aux appels et solutions atomiques apparaissant lors de la résolution. Après avoir fixé une algèbre abstraite, i.e., un domaine abstrait et des opérations élémentaires associées, un arbre ET/OU abstrait peut être construit à partir de tout couple composé d'un programme concret et d'un but abstrait. Le rapport entre l'univers concret et l'univers abstrait est basé sur une insertion de Galois (tout du moins pour [Bruynooghe et al. 87]). La sémantique d'un arbre ET/OU abstrait est alors établie par la donnée d'une application définie sur l'ensemble des arbres de preuve (arbres ET). Soit P un programme concret et soit G un but abstrait, la concrétisation d'un arbre ET/OU abstrait construit pour (P,G) doit représenter l'ensemble des arbres de preuve pouvant apparaître lors de l'exécution de tout couple (P,G') où G' est un but concret appartenant à la concrétisation de G. Tout comme pour [Mellish 87], la complétude de la sémantique des arbres ET/OU abstraits est établie localement. Le problème de la terminaison de la construction d'un arbre ET/OU abstrait dépend de la structure du domaine abstrait considéré. Si celui-ci vérifie la condition de chaîne ascendante, alors l'arrêt est assuré.

[Bruynooghe et Janssens 88] présentent une inférence de modes et une inférence de types. Nous donnons ici quelques précisions sur l'inférence de types. La définition d'un type T est donnée par l'équation : T = ex où ex est une expression consistant en une disjonction (non vide) de variants, chaque variant étant une construction de type de la forme $f(T_1,...,T_n)$ avec $n \geq 0$. Par exemple,

$T = f(T_1) \mid g(T_2,T_1)$

$T_1 = k$

$T_2 = f(T_3)$

$T_3 = g(T,T_1)$

est un typage dit normalisé, i.e., vérifiant strictement la syntaxe fixée.

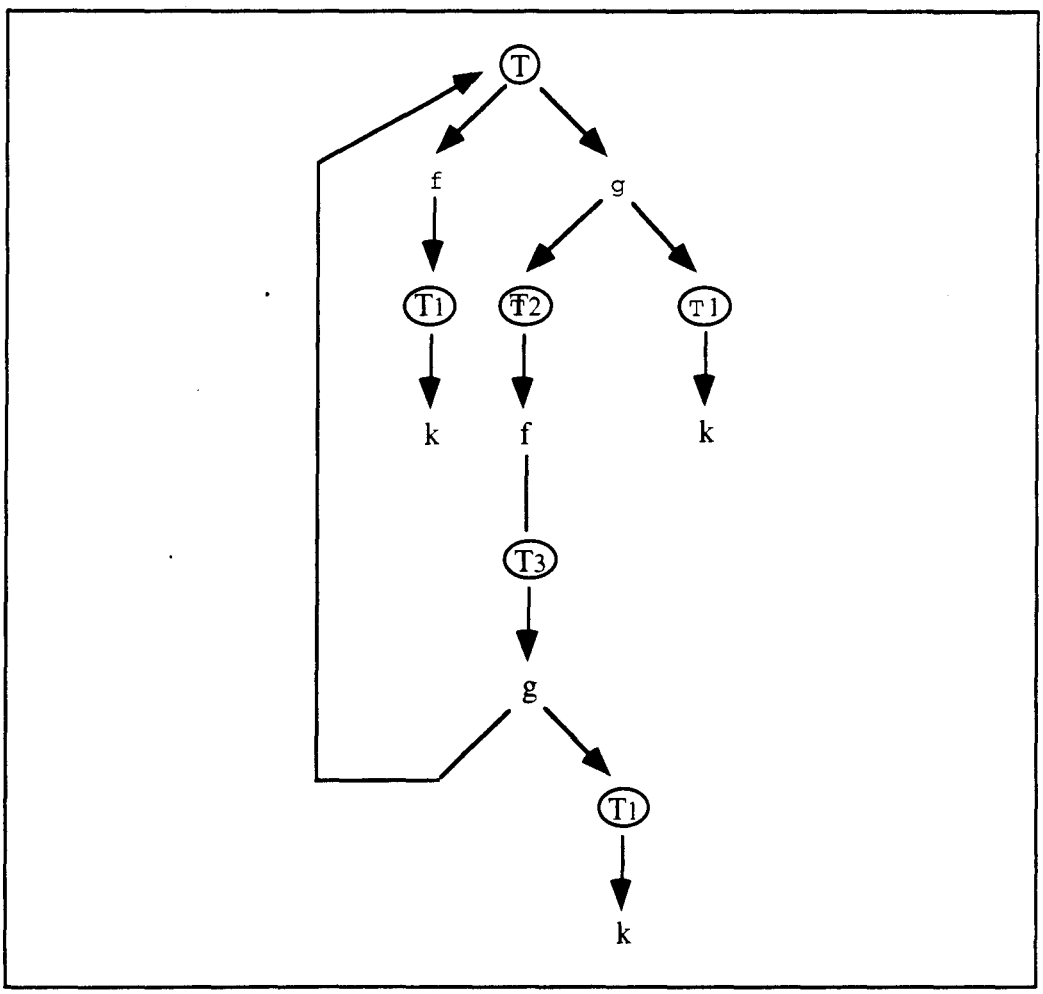

Figure 4.1

A tout type T, on peut associer un graphe de type g(T) dans lequel apparaissent des noeuds de type et des noeuds de foncteurs. La traduction est assez naturelle, avec en particulier l'impossibilité de rencontrer sur le même chemin deux noeuds de type avec la même étiquette (des cycles sont ainsi créés). Deux restrictions sont faites sur la construction des graphes de type. La première stipule que les variants d'un type doivent être de foncteurs différents. De ce fait, la largeur d'un graphe de type est nécessairement fini. La seconde stipule que sur tout chemin acyclique débutant à la racine, le même foncteur ne doit pas figurer deux fois. De ce fait, la profondeur d'un graphe est nécessairement fini. Lorsque l'une de ces restrictions est violée, une simplification est effectuée. Il s'agit en fait d'un opérateur de widening puisque toute chaîne strictement croissante de graphes de type est finie. Un graphe de type représente l'ensemble des termes qui peuvent être pliés sur le graphe. En tenant compte des restrictions indiquées ci-dessus, il apparaît qu'il n'existe au plus qu'une façon de plier un terme sur un graphe de type.

L'impact de l'implémentation dans un compilateur des analyses proposées (inférence de modes, de types, ...) est décrit par [Marrien et al. 89].

De leur côté, [Kanamori et Kawamura 90] propose un modèle d'interprétation abstraite appliquée à la programmation logique (Prolog) basée sur la résolution OLDT. La résolution OLDT est une extension de la résolution OLD proposée par [Tamaki et Sato 86]. Elle incorpore une technique de tabulation qui permet d'éviter certains calculs redondants. Pour tout couple composé d'un programme et d'un but, la résolution OLDT construit une structure OLDT (essentiellement un arbre et deux tables). En fixant un domaine abstrait fini et en redéfinissant l'opération d'unification correspondant à ce nouveau domaine, la terminaison de la résolution OLDT est toujours assurée. Nous ne détaillons pas outre mesure ce modèle car au chapitre 6, nous donnons une description complète de la résolution OLDT.

Pour ce modèle, de nombreuses applications ont été définies. La première est une analyse du flux de données [Kanamori et Kawamura 87]. En utilisant une méthode d'abstraction sur la profondeur des termes, une approximation de la forme des appels et solutions atomiques apparaissant lors de la résolution est obtenue. Par ailleurs, en utilisant une définition préalable de types (sous forme de clauses), une inférence de type est élaborée. Cette analyse est étendue au cas polymorphe par [Horiuchi et Kanamori 87]. Pour finir, une inférence de modes prenant en compte le partage (sharing) est présentée. Les autres applications concernent l'analyse de la terminaison [Kanamori et al. 87a] et de la fonctionnalité [Kanamori et al. 87a] de programmes logiques.

1.2 Approche de point fixe

[Le Charlier et Van Hentenryck 92b], suivant [Cousot et Cousot 77] défendent l'idée d'une approche de l'interprétation abstraite par calcul de point fixe. Ils proposent un algorithme de point fixe universel [Le charlier et Van Hentenryck 92a] qui peut être instancié par rapport à un langage de programmation donné. L'algorithme obtenu ainsi est lui-même générique car il est paramétré par un domaine abstrait. Différentes sémantiques sont définies en fonction du choix du domaine et des opérations élémentaires liées à ce domaine. Toutefois, l'algorithme général (de calcul de la sémantique) reste le même et peut donc être utilisé pour diverses applications.

Un algorithme générique d'interprétation abstraite est proposée pour Prolog dans l'article de [Le Charlier et al. 91]. Concevoir une analyse consiste à instancier cet algorithme, i.e, à définir un ensemble de substitutions abstraites prenant en compte l'information importante pour l'analyse et à définir un ensemble d'opérations abstraites élémentaires qui soient consistantes par rapport à leurs versions concrètes. Les programmes Prolog considérés sont normalisés (les atomes sont homogènes) afin de simplifier la manipulation des substitutions. Une sémantique abstraite de point fixe est élaborée principalement à partir de trois fonctions et une transformation. Elle est définie en termes de n-uplets de la forme $(\beta_{in},p,\beta_{out})$ où p est un symbole de prédicat, et β_{in} et β_{out} représentent des substitutions abstraites. De manière informelle, un n-uplet $(\beta_{in},p,\beta_{out})$ souligne le fait suivant : si σ est une substitution (concrète) satisfaisant l'information exprimée par β_{in} alors toute solution σ' obtenue par l'interprétation de $(P,\sigma(p(X_1,...,X_n)))$ où P est un programme donné, satisfait l'information exprimée par β_{out}. L'algorithme permettant de calculer cette sémantique est construit de manière indépendante. Ainsi, les aspects sémantiques et algorithmiques sont-ils clairement séparés.

1.3 Approche dénotationnelle

[Marriott et Sondergaard 90b] présentent un modèle d'interprétation abstraite appliqué à la programmation logique avec contraintes. Ce travail est d'une certaine manière la suite logique de [Marriott et Sondergaard 89] et [Marriott et Sondergaard 90a] car les aspects top-down et dénotationnel sont introduits dans ces deux articles. L'idée d'analyser le comportement dynamique de CLP-programmes est d'autant plus intéressante que les problèmes de satisfaction de contraintes s'avèrent être couteux pour certains CLP-langages.

[Marriott et Sondergaard 90b], suivant l'approche de [Nielson 88], montrent l'intérêt à définir la sémantique d'un langage de programmation à partir d'un

méta-langage, de sorte que cette définition soit générique, i.e., valable pour n'importe quel langage de programmation. Ils proposent donc un méta-langage adapté à la définition de sémantiques de langages basés sur la logique et le formalisme dénotationnel. Ce méta-langage permet d'élaborer des lambda expressions typées. La sémantique d'une expression du méta-langage varie en fonction de l'interprétation donnée aux éléments de base du méta-langage. On peut ainsi obtenir différentes sémantiques à partir des mêmes expressions. Le rôle de l'interprétation abstraite est d'établir un lien entre une interprétation standard et une interprétation non standard. Ce lien est établi par une fonction de concrétisation.

[Marriott et Sondergaard 90b] formalisent la sémantique opérationnelle d'un CLP langage par rapport au méta-langage introduit. Quelques distinctions sont toutefois précisées. Par exemple, l'échec fini et la non terminaison d'un programme ne peuvent être différenciés dans la sémantique proposée. Ceci permet en contrepartie l'utilisation d'une stratégie de parcours "en largeur d'abord". Par ailleurs, la dénotation d'un programme est une application définie à partir de l'ensemble des atomes plutôt que l'ensemble des séquences d'atomes. Deux analyses sont proposées comme exemples : une analyse de la franchise ("freeness analysis") et une analyse de l'assignation unique ("groundness analysis"). La seconde sera présentée au chapitre suivant.

2 Discussion

Nous allons maintenant analyser plus précisément certains aspects de la définition d'une interprétation abstraite (appliquée à la programmation logique avec contraintes), et clairement énoncer les choix que nous allons effectuer pour la définition du modèle que nous souhaitons élaborer.

Avant toute chose, il est nécessaire de formuler le cahier des charges du modèle d'interprétation abstraite que nous souhaitons définir. Ce modèle doit être :

- générique,
  i.e., utilisable pour l'ensemble des CLP-langages.

- souple,
  i.e., permettre de définir aussi bien des analyses de flux de données (inférence de mode, inférence de type, ...) que des analyses opérationnelles (étude de la terminaison, du déterminisme, ...).

- simple,
  i.e., permettre de concevoir facilement une analyse et de prouver la consistance de celle-ci.

• précis,
i.e., permettre de concevoir des analyses dont le résultat soit précis.

Dans notre cahier des charges, nous avons indiqué notre souhait de pouvoir établir des analyses opérationnelles à partir de notre modèle. Or, plus la sémantique abstraite est proche de la sémantique concrète, plus l'information opérationnelle disponible est grande. De plus, CLP se prête tout à fait naturellement à l'abstraction du calcul concret. En effet, pour obtenir une interprétation abstraite, il suffit de définir un CLP-langage abstrait apte à simuler un CLP-langage concret, et d'introduire une relation d'approximation entre les domaines des deux langages. Ceci constitue l'idée maîtresse de l'approche opérationnelle de [Codognet et Filè 92]. Les prérequis de généricité, souplesse et simplicité sont alors satisfaits. Reste le problème de la précision du résultat des analyses ... mais nous y reviendrons plus loin.

⇒ nous considérons une approche opérationnelle.

Ensuite, il est indispensable de se positionner par rapport à l'interprétation. Suivant [Marriott et Sondergaard 89b], les interprétations abstraites définies pour la programmation logique se rangent essentiellement dans deux classes distinctes : les interprétations top-down et les interprétations bottom-up. Une interprétation top-down tient compte d'un programme et d'un but (ou état initial) et caractérise par chaînage arrière l'ensemble des appels possibles tandis qu'une interprétation bottom-up tient compte d'un programme et de faits (ou états finaux) et caractérise par chaînage avant l'ensemble des solutions possibles. La plupart des travaux consacrés à l'interprétation abstraite en programmation logique correspondent à une interprétation top-down [Mellish 87 , Kanamori et Kawamura 90 , Bruynooghe 91 , Marriott et Sondergaard 90 , Le Charlier et al. 91] mais dans le cadre des interprétations bottom-up, on trouve notamment les travaux de [Marriott et Sondergaard 88,92]. Pour tenir compte des conditions "réelles" de l'exécution d'un programme, il est nécessaire d'utiliser une interprétation top-down.

⇒ nous considérons une interprétation top-down.

Il est à noter qu'un interpréteur Prolog (concret) top-down a parfois un comportement infini (là où un interpréteur bottom-up a un comportement fini), aussi est-il nécessaire d'introduire un "principe bottom-up" lors de l'élaboration d'un interpréteur abstrait top-down, c'est à dire une technique qui prend en compte ce qui a déjà été calculé. Il peut s'agir de la tabulation [Tamaki et Sato 86] ou d'un calcul de point fixe.

⇒ nous considérons l'utilisation de la tabulation.

Pour revenir au problème de la précision du modèle, il est important de noter la manière dont est conçue l'approximation. Les différents travaux en interprétation abstraite se rangent par rapport à :

- l'approximation par combinaison

ou

- l'approximation par abstraction,

mais aucun (à notre connaissance) ne se range explicitement par rapport à :

- l'approximation par extension.

Le modèle d'approximation par combinaison consiste à définir la sémantique abstraite à partir d'un programme concret et d'un but abstrait. A chaque étape élémentaire, il est donc nécessaire de combiner information abstraite et information concrète. Ce modèle est le plus répandu [Mellish 87 , Bruynooghe 91 , Marriott et Sondergaard 90 , Kanamori et Kawamura 90 , Le Charlier et al. 91]. Le modèle d'approximation par abstraction consiste à définir la sémantique abstraite à partir d'un programme abstrait et d'un but abstrait. [Hermenegildo et al. 92] utilisent le terme de compilation abstraite. Du fait de l'abstraction immédiate (avant le début de l'exécution) et dans le cas où cette abstraction est syntaxique, les analyses fondées sur ce modèle peuvent se révéler moins précises (pour un domaine abstrait équivalent). Ce modèle est utilisé par [Codognet et Filè 92 , Warren 92 , Codish et Daemon 93], et nous rappelons que c'est celui sur lequel nous nous basons. Toutefois, nous nous plaçons dans un cadre particulier, à savoir, le domaine abstrait doit être une extension du domaine concret. C'est pourquoi nous appelons ce modèle le modèle d'approximation par extension. L'avantage est qu'aucune abstraction n'est faite à priori, et qu'en conséquence, les analyses peuvent se révéler très précises.

$\Rightarrow$ nous considérons une approximation par extension.

Pour assurer la terminaison de l'analyse, il est nécessaire d'introduire des opérateurs de widening.

$\Rightarrow$ nous considérons l'utilisation d'opérateurs de widening.

Signalons que le choix d'un modèle d'approximation par extension va dans le sens d'une approche de l'interprétation abstraite basée sur une phase d'abstraction (ou plutôt extension) du domaine suivie d'une phase d'abstraction du calcul.

Partie III

Un modèle générique d'interprétation abstraite

Dans cette partie, nous présentons un modèle d'interprétation abstraite générique appliqué à la programmation logique avec contraintes. Ce modèle est constituée d'une phase d'extension du domaine suivie d'une phase d'abstraction du calcul. La phase d'extension du domaine (chapitre 5) consiste à enrichir le domaine concret avec de nouvelles contraintes. La phase d'abstraction du calcul consiste à utiliser la tabulation et à introduire des opérateurs de widening (et narrowing) pour assurer la terminaison du calcul abstrait.

Chapitre 5

Extension du Domaine

Dans ce chapitre, nous présentons la première phase (phase d'extension du domaine) du modèle d'interprétation abstraite que nous développons dans cette thèse. Cette phase consiste à définir une sémantique abstraite qui soit une approximation de la sémantique opérationnelle[†] d'un CLP-langage $\mathcal{CL}^{\alpha}$ (dit concret). Il n'est toutefois pas revendiqué que cette sémantique abstraite soit calculable.

En fait, nous présentons dans ce chapitre trois modèles d'approximation de la sémantique concrète, c'est à dire trois manières de définir la sémantique abstraite. Ceux-ci sont appelés respectivement modèles d'approximation par abstraction, par combinaison et par extension. Dans les trois modèles (section 2), on introduit un CLP-langage abstrait $\mathcal{CL}^{\nabla}$ et une relation d'approximation ξ définie entre $\mathcal{CL}^{\alpha}$ et $\mathcal{CL}^{\nabla}$ (section 1). Pour le premier modèle, i.e. le modèle d'approximation par abstraction, la sémantique abstraite est tout simplement la sémantique opérationnelle de $\mathcal{CL}^{\nabla}$. Ce modèle a été introduit par [Codognet et Filè 92]. Pour le second modèle, i.e. le modèle d'approximation par combinaison, la sémantique abstraite est une sémantique opérationnelle mixte, c'est à dire une sémantique définie à la fois à partir de $\mathcal{CL}^{\alpha}$ et de $\mathcal{CL}^{\nabla}$. Enfin, le troisième modèle, i.e. le modèle d'approximation par extension, est un cas particulier du premier, à savoir : $\mathcal{CL}^{\nabla}$ est une extension de $\mathcal{CL}^{\alpha}$ et la relation d'approximation est (une restriction de) la relation d'ordre $\vdash^{\nabla}$ définie sur le domaine de $\mathcal{CL}^{\nabla}$. Ce dernier modèle a notre préférence car il est à la fois simple, naturel et efficace. Lorsque ce modèle est choisi, la phase d'abstraction du domaine correspond donc à une phase d'extension du domaine.

[†] Nous rappelons que la résolution OLD_{bf} est considérée par défaut

Nous illustrons (section 3) les trois modèles d'approximation présentés ci-dessus avec l'analyse de l'assignation unique. Cette analyse consiste à déterminer quelles sont les variables qui sont contraintes à une assignation unique pendant la résolution. Une telle analyse est envisagée par rapport à CLP($\mathcal{FT}$), puis par rapport à CLP(IR).

1. Relation d'approximation ξ

On suppose fixés un CLP-langage (dit) concret $CL^{\Delta} = (\Sigma^{\Delta}, \mathcal{L}^{\Delta}, \mathcal{A}^{\Delta})$ et un CLP-langage (dit) abstrait $CL^{\nabla} = (\Sigma^{\nabla}, \mathcal{L}^{\nabla}, \mathcal{A}^{\nabla})$. Toute objet lié à CL^{Δ} est annoté par le symbole $^{\Delta}$ et tout objet lié à CL^{∇} est annoté par le symbole $^{\nabla}$. On introduit une relation d'approximation ξ entre les domaines respectifs $\mathcal{L}^{\Delta}$ et $\mathcal{L}^{\nabla}$ de CL^{Δ} et CL^{∇} et on étudie les extensions (section 1.1) et les propriétés (section 1.2) possibles de ξ.

Un lien est établi entre $\mathcal{L}^{\Delta}$ et $\mathcal{L}^{\nabla}$ par le biais d'une relation d'approximation ξ, puis est étendu entre les divers objets syntaxiques de CL^{Δ} et CL^{∇} (clause, programme,...). Cette technique a été proposée par [Codognet et Filé 92] dans un contexte légèrement différent. En effet, les auteurs définissent un cadre générique de systèmes de calcul qui prend en compte les CLP-langages. Le fait de nous "limiter" à l'étude des CLP-langages nous permet de préciser certains points. Il est à noter que nous utilisons une notation fonctionnelle pour ξ.

Dans le contexte de notre étude, une relation d'approximation ξ doit vérifier certaines conditions. Celles-ci seront justifiées plus loin. On dit alors que la relation ξ est valide.

Définition 5.1

Une relation $\xi \in \wp(\mathcal{L}^{\Delta} \times \mathcal{L}^{\nabla})$ est une relation d'approximation valide ssi

- ξ est close par renommage, conjonction et quantification existentielle.
- $\xi(c^{\Delta}, \bot^{\nabla}) \Rightarrow c^{\Delta} = \bot^{\Delta}$.

Il est possible d'effectuer un parallèle avec le travail de [Mesnard 93]. Celui-ci propose une classe d'applications, appelées approximations, entre CLP-langages. Une telle application se définit par la donnée d'une transformation syntaxique et d'un morphisme de structures (algèbres) compatible avec la transformation syntaxique. Les conditions qu'il impose aux approximations sont locales aux contraintes primitives du langage tandis que celles que nous imposons sont globales aux langage.

1.1. Extension de ξ

La relation ξ est étendue de façon inductive comme cela est indiquée par la définition 3.5. De la même manière qu'au chapitre 2 pour la relation $\vdash$, nous définissons (de façon artificielle) la relation ξ sur $\mathcal{H}\times\mathcal{H}$ comme étant l'identité $I\delta$. Ceci nous permet d'obtenir la définition suivante de ξ entre $\mathcal{L}^{\Delta}\times\mathcal{H}^{n}$ et $\mathcal{L}^{\nabla}\times\mathcal{H}^{n}$:

$\forall\ as \in \mathcal{H}^{n},\ \forall\ (c^{\Delta},c^{\nabla}) \in \mathcal{L}^{\Delta}\times\mathcal{L}^{\nabla},\ \xi((c^{\Delta},as),(c^{\nabla},as))$ ssi $\xi(c^{\Delta},c^{\nabla})$

En considérant toute clause et tout but comme une séquence d'atomes contraints (voir chapitre 2, section 2), il en résulte une extension directe de la définition de ξ entre les clauses, les buts et les programmes de $C\mathcal{L}^{\Delta}$ et $C\mathcal{L}^{\nabla}$. On définit également ξ entre $C\mathcal{L}^{\Delta}$-Tree et $C\mathcal{L}^{\nabla}$-Tree.

Définition 5.2

$\forall\ T^{\Delta} = (\mathcal{N}^{\Delta},i^{\Delta},r^{\Delta}) \in C\mathcal{L}^{\Delta}$-Tree, $\forall\ T^{\nabla} = (\mathcal{N}^{\nabla},i^{\nabla},r^{\nabla}) \in C\mathcal{L}^{\nabla}$-Tree,
$\xi(T^{\Delta},T^{\nabla})$ ssi il existe une application map $\in \mathcal{N}^{\Delta} \rightarrow \mathcal{N}^{\nabla}$ telle que :

- $\text{map}(i^{\Delta}) = i^{\nabla}$
- si $\text{map}(v^{\Delta}) = v^{\nabla}$ alors
  - $\xi(\text{label}(v^{\Delta}),\text{label}(v^{\nabla}))$
  - $\forall\ (v^{\Delta},w^{\Delta}) \in r^{\Delta},\ \exists\ (v^{\nabla},w^{\nabla}) \in r^{\nabla}$ tel que :
    $\text{num}(v^{\Delta},w^{\Delta}) = \text{num}(v^{\nabla},w^{\nabla})$
    $\text{map}(w^{\Delta}) = w^{\nabla}$.

A partir de la relation ξ définie ci-dessus, on introduit une relation ξ_r en considérant ξ modulo le renommage des variables. Cette relation nous intéresse car elle permet de nous abstraire du nom des variables.

Définition 5.3

Soit la relation ξ définie sur un ensemble D, la relation ξ_r est définie sur D comme suit : $\forall\ (d,d') \in D^2$, $\xi_r(d,d')$ ssi $\exists\ \rho \in \mathcal{R}en \mid \xi(d,\rho(d'))$.

Comme toute relation ξ est stable par renommage, on sait que : $\forall\ (d,d') \in D^2$, $\xi_r(d,d')$ ssi $\exists\ \rho \in \mathcal{R}en$: $\xi(\rho(d),d')$.

1.2. Propriétés de ξ

Nous discutons maintenant de la caractérisation de ξ comme cela a été indiqué au chapitre 3 (suivant l'approche de [Cousot et Cousot 92c] et de [Marriott 93]). Tout d'abord, nous redéfinissons la relation d'approximation ξ de manière équivalente par l'application ass_{α} et l'application ass_{γ}.

Définition 5.4

L'application $\text{ass}_{\alpha} \in \mathcal{L}^{\Delta} \rightarrow \wp(\mathcal{L}^{\nabla})$ est définie par :
$\forall\ c^{\Delta} \in \mathcal{L}^{\Delta}$, $\text{ass}_{\alpha}(c^{\Delta}) = \{ c^{\nabla} \in \mathcal{L}^{\nabla} \mid \xi(c^{\Delta},c^{\nabla}) \}$.

Définition 5.5

L'application $ass_\gamma \in \mathcal{L}^\nabla \rightarrow \wp(\mathcal{L}^\Delta)$ est définie par :
$\forall\ c^\nabla \in \mathcal{L}^\nabla,\ ass_\gamma(c^\nabla) = \{\, c^\Delta \in \mathcal{L}^\Delta \mid \xi(c^\Delta, c^\nabla) \,\}$.

Si la simulation est définie dans le sens de l'abstraction (du concret vers l'abstrait), alors plusieurs propriétés sont à envisager. On considère la relation d'ordre $\vdash^\nabla$ définie sur $\mathcal{L}^\nabla$ (voir chapitre 2, section 4.2).

Propriété 5.6

(1) $\forall\ c^\Delta \in \mathcal{L}^\Delta, \exists\ c^\nabla \in \mathcal{L}^\nabla \mid \xi(c^\Delta, c^\nabla)$
(2) $\forall\ c^\Delta \in \mathcal{L}^\Delta, \forall\ (c^\nabla, d^\nabla) \in \mathcal{L}^{\nabla 2}, \xi(c^\Delta, c^\nabla) \wedge c^\nabla \vdash^\nabla d^\nabla \Rightarrow \xi(c^\Delta, d^\nabla)$
(3) $\forall\ c^\Delta \in \mathcal{L}^\Delta, ass_\alpha(c^\Delta) \neq \emptyset \Rightarrow \exists\ c^\nabla \in ass_\alpha(c^\Delta) \mid \forall\ d^\nabla \in ass_\alpha(c^\Delta),\ c^\nabla \vdash^\nabla d^\nabla$

La propriété (1) assure la possibilité d'abstraire n'importe quelle contrainte concrète. La propriété (2) assure la cohérence de ξ avec la relation d'ordre $\vdash^\nabla$. La propriété (3) assure l'existence d'une plus petite contrainte abstraite approchant une contrainte concrète (lorsqu'il en existe une).

On peut caractériser une relation d'approximation ξ vérifiant les propriétés ci-dessus par une fonction (application) d'abstraction α.

Propriété 5.7

Si ξ est une relation d'approximation vérifiant les propriétés (1), (2) et (3), et si α est l'application définie de $\mathcal{L}^\Delta$ vers $\mathcal{L}^\nabla$ par :

(4) $\forall\ c^\Delta \in \mathcal{L}^\Delta,\ \alpha(c^\Delta) = c^\nabla \mid c^\nabla \in ass_\alpha(c^\Delta)$ et $\forall\ d^\nabla \in ass_\alpha(c^\Delta),\ c^\nabla \vdash^\nabla d^\nabla$,

alors :

$\forall\ c^\Delta \in \mathcal{L}^\Delta,\ \forall\ c^\nabla \in \mathcal{L}^\nabla,\ \xi(c^\Delta, c^\nabla)$ ssi $\alpha(c^\Delta) \vdash^\nabla c^\nabla$.

Si la simulation est définie dans le sens de la concrétisation (de l'abstrait vers le concret), alors les propriétés duales de celles que nous venons d'introduire sont à envisager. On considère la relation d'ordre $\vdash^\Delta$ définie sur $\mathcal{L}^\Delta$.

Propriété 5.8

(1') $\forall\ c^\nabla \in \mathcal{L}^\nabla, \exists\ c^\Delta \in \mathcal{L}^\Delta \mid \xi(c^\Delta, c^\nabla)$
(2') $\forall\ c^\nabla \in \mathcal{L}^\nabla, \forall\ (c^\Delta, d^\Delta) \in \mathcal{L}^{\Delta 2}, \xi(c^\Delta, c^\nabla) \wedge d^\Delta \vdash^\Delta c^\Delta \Rightarrow \xi(d^\Delta, c^\nabla)$
(3') $\forall\ c^\nabla \in \mathcal{L}^\nabla, ass_\gamma(c^\nabla) \neq \emptyset \Rightarrow \exists\ c^\Delta \in ass_\gamma(c^\nabla) \mid \forall\ d^\Delta \in ass_\gamma(c^\nabla),\ d^\Delta \vdash^\Delta c^\Delta$

La propriété (1') assure la possibilité de concrétiser n'importe quelle contrainte abstraite. La propriété (2') assure la cohérence de ξ avec la relation d'ordre $\vdash^\Delta$. La propriété (3') assure l'existence d'une plus grande contrainte concrète approchée par une contrainte abstraite (lorsqu'il en existe une).

On peut caractériser une relation d'approximation ξ vérifiant les propriétés ci-dessus par une fonction (application) de concrétisation γ.

Propriété 5.9

Si ξ est une relation d'approximation vérifiant les propriétés (1'), (2') et (3'), et si γ est l'application définie de $\mathcal{L}^\nabla$ vers $\mathcal{L}^\Delta$ par :

(4') $\forall\ c^\nabla \in \mathcal{L}^\nabla,\ \gamma(c^\nabla) = c^\Delta \mid c^\Delta \in ass_\gamma(c^\nabla)$ et $\forall\ d^\Delta \in ass_\gamma(c^\nabla),\ d^\Delta \vdash^\Delta c^\Delta$,

alors :

$\forall\ c^\Delta \in \mathcal{L}^\Delta,\ \forall\ c^\nabla \in \mathcal{L}^\nabla,\ \xi(c^\Delta,c^\nabla)$ ssi $c^\Delta \vdash^\Delta \gamma(c^\nabla)$.

Finalement, lorsque toutes ces propriétés sont vérifiées, on obtient une connexion de Galois.

Propriété 5.10

Si ξ est une relation d'approximation vérifiant les propriétés (1), (1'), (2), (2'), (3) et (3'), si α est définie par (4) et si γ est définie par (4') alors (α,γ) est une connexion de Galois définie entre $(\mathcal{L}^\Delta,\vdash^\Delta)$ et $(\mathcal{L}^\nabla,\vdash^\nabla)$.

2. Modèles d'approximation

On suppose fixés un CLP-langage concret $C\mathcal{L}^\Delta = (\Sigma^\Delta,\mathcal{L}^\Delta,\mathcal{A}^\Delta)$, un CLP-langage abstrait $C\mathcal{L}^\nabla = (\Sigma^\nabla,\mathcal{L}^\nabla,\mathcal{A}^\nabla)$ et une relation d'approximation $\xi \in \wp(\mathcal{L}^\Delta \times \mathcal{L}^\nabla)$ valide. Nous présentons dans cette partie plusieurs modèles d'approximation de la sémantique opérationnelle de $C\mathcal{L}^\Delta$: approximation par abstraction (section 2.1), par combinaison (section 2.2) et par extension (section 2.3). Nous discutons (section 2.4) ensuite de l'intérêt et de l'efficacité de ces différents modèles.

2.1. Approximation par abstraction

Nous cherchons à établir (via ξ) l'approximation de la sémantique opérationnelle de $C\mathcal{L}^\Delta$ par la sémantique opérationnelle de $C\mathcal{L}^\nabla$. Cette approximation est dite par abstraction car pour toute analyse le programme concret est substitué (abstrait) par un programme abstrait et le but concret est substitué (abstrait) par un but abstrait. Ce modèle est celui proposé par [Codognet et Filè 92] et un peu plus informellement par [Warren 92]. L'approximation par abstraction est illustrée par la figure 5.11.

En reprenant la définition 3.5 (à une technique de renommage près), on note $\xi(Res^\Delta,Res^\nabla)$ et on dit que Res^∇ approche Res^Δ via ξ ssi

$\forall\ (P^\Delta,G^\Delta) \in C\mathcal{L}^\Delta\text{-}Prog \times C\mathcal{L}^\Delta\text{-}Goal,\ \forall\ (P^\nabla,G^\nabla) \in C\mathcal{L}^\nabla\text{-}Prog \times C\mathcal{L}^\nabla\text{-}Goal,$
$\xi_r((P^\Delta,G^\Delta),(P^\nabla,G^\nabla)) \Rightarrow \xi_r(Res^\Delta(P^\Delta,G^\Delta),Res^\nabla(P^\nabla,G^\nabla))$.

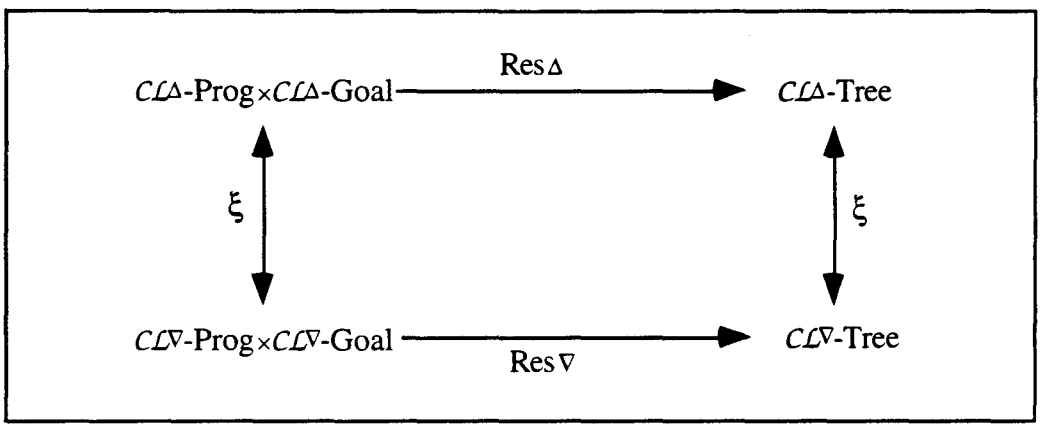

Figure 5.11

On établit d'abord la correction du processus élémentaire de l'approximation.

Propriété 5.12

$\forall$ $(P^{\Delta},G^{\Delta}) \in CL^{\Delta}\text{-Prog}_{\times}CL^{\Delta}\text{-Goal}$, $\forall$ $(P^{\nabla},G^{\nabla}) \in CL^{\nabla}\text{-Prog}_{\times}CL^{\nabla}\text{-Goal}$. Si $\xi_r((P^{\Delta},G^{\Delta}),(P^{\nabla},G^{\nabla}))$ et si il est possible de calculer le (i,j)-résolvant, noté H^{Δ}, de (P^{Δ},G^{Δ}) alors il est possible de calculer le (i,j)-résolvant, noté H^{∇}, de (P^{∇},G^{∇}). De plus $\xi_r(H^{\Delta},H^{\nabla})$.

Preuve

Soient $G^{\Delta} : \leftarrow c^{\Delta},a^{\Delta}_1,\ldots,a^{\Delta}_n$ et $G^{\nabla} : \leftarrow c^{\nabla},a^{\nabla}_1,\ldots,a^{\nabla}_n$

$\xi_r((P^{\Delta},G^{\Delta}),(P^{\nabla},G^{\nabla})) \Rightarrow \xi_r(P^{\Delta},P^{\nabla})$ et $\xi_r(G^{\Delta},G^{\nabla})$

$\xi_r(P^{\Delta},P^{\nabla}) \Rightarrow \xi_r(C^{\Delta}_j,C^{\nabla}_j)$ pour tout j $\Rightarrow \exists\ \rho_1 \in \mathcal{R}en$ tel que $\xi(C^{\Delta}_j,\rho_1(C^{\nabla}_j))$

$\xi_r(G^{\Delta},G^{\nabla}) \Rightarrow \exists\ \rho_2 \in \mathcal{R}en$ tel que $\xi(G^{\Delta},\rho_2(G^{\nabla}))$.

On sait (par hypothèse) qu'une variante $C^{\Delta} : h^{\Delta} \leftarrow d^{\Delta},bs^{\Delta}$ de C^{Δ}_j est considérée telle que $h^{\Delta} = a^{\Delta}_i$, donc $\exists\ \rho_1' \in \mathcal{R}en$ tel que $C^{\Delta} = \rho_1'(C^{\Delta}_j)$.

$\xi_r(C^{\Delta}_j,C^{\nabla}_j)$ et $C^{\Delta} = \rho_1'(C^{\Delta}_j) \Rightarrow \xi_r(C^{\Delta},C^{\nabla}_j)$ car $\rho_1' \circ \rho_1 \in \mathcal{R}en$ et $\xi(C^{\Delta},\rho_1' \circ \rho_1(C^{\nabla}_j))$ (ξ est stable par rapport au renommage).

On sait que :

$h^{\Delta} = a^{\Delta}_i$

$a^{\Delta}_i = \rho_2(a^{\Delta}_i)$

$h^{\Delta} = \rho'_1 \circ \rho_1(h^{\nabla'})$ où $h^{\nabla'}$ est la tête de clause de C^{∇}_j

On en déduit : $\rho_2(a^{\nabla}_i) = \rho_1' \circ \rho_1(h^{\nabla'})$ et donc $a^{\nabla}_i = \rho_2^{-1} \circ \rho_1' \circ \rho_1(h^{\nabla'})$.

Il existe donc une variante $C^{\nabla} : h^{\nabla} \leftarrow d^{\nabla},bs^{\nabla}$ de C^{∇}_j telle que $h^{\nabla} = a^{\nabla}_i$, et ainsi $\exists\ \rho_2' \in \mathcal{R}en \mid C^{\nabla}_j = \rho_2'(C^{\nabla})$.

$\xi_r(C^\Delta, C^\nabla_j)$ et $C^\nabla_j = \rho_2'(C^\nabla) \Rightarrow \xi_r(C^\Delta, C^\nabla)$ car $\rho_2'o\rho_1'o\rho_1 \in \mathcal{R}en$ et $\xi(C^\Delta, \rho_2'o\rho_1'o\rho_1(C^\nabla))$.

Posons $\rho_3 = \rho_2'o\rho_1'o\rho_1$. A partir de ρ_2 et ρ_3 il est possible de construire une substitution de renommage ρ telle que :

$\forall X \in Var(G^\nabla), \rho(X) = \rho_2(X)$.

$\forall X \in Var(C^\nabla), \rho(X) = \rho_3(X)$.

On utilise essentiellement a) $\forall X \in Var(a^\nabla_i), \rho_2(X) = \rho_3(X)$ (en effet, $h^\nabla = a^\nabla_i$, $\rho_2(a^\nabla_i) = a^\Delta_i$, $\rho_3(h^\nabla) = h^\Delta$ et $h^\Delta = a^\Delta_i$), b) $Var(G^\nabla) \cap Var(C^\nabla) = Var(a^\nabla_i)$, c) $Var(G^\Delta) \cap Var(C^\Delta) = Var(a^\Delta_i)$.

Aussi, on déduit de l'existence de ρ que :

$\xi(c^\Delta, \rho_2(c^\nabla)) \Rightarrow \xi(c^\Delta, \rho(c^\nabla))$

$\xi(d^\Delta, \rho_3(d^\nabla)) \Rightarrow \xi(d^\Delta, \rho(d^\nabla))$

puis par stabilité de ξ par rapport à la conjonction que :

$\xi(c^\Delta \wedge d^\Delta, \rho(c^\nabla \wedge d^\nabla))$

On sait que $c^\Delta \wedge d^\Delta$ est satisfiable (par hypothèse), aussi on en déduit que $\rho(c^\nabla \wedge d^\nabla)$ est satisfiable car ξ est valide.

Le (i,j)-résolvant de (P^Δ, G^Δ) est le CL^Δ-but suivant :

$H^\Delta : \leftarrow c^\Delta \wedge d^\Delta, a^\Delta_1, \ldots, a^\Delta_{i-1}, bs^\Delta, a^\Delta_{i+1}, \ldots, a^\Delta_n$

Le (i,j)-résolvant de (P^∇, G^∇) est le CL^∇-but suivant :

$H^\nabla : \leftarrow c^\nabla \wedge d^\nabla, a^\nabla_1, \ldots, a^\nabla_{i-1}, bs^\nabla, a^\nabla_{i+1}, \ldots, a^\nabla_n$

$\xi_r(H^\Delta, H^\nabla)$ car $\rho \in \mathcal{R}en$ et $\xi(H^\Delta, \rho(H^\nabla))$ ❑

Cette preuve met en relief la nécessité de la validité de ξ. De cette propriété, il découle que Res^∇ approche Res^Δ via ξ. Il faut remarquer une certaine analogie de cette déduction avec la propriété 3.32.

Propriété 5.13

Res^∇ approche Res^Δ via ξ.

Preuve

La preuve s'effectue par récurrence sur la profondeur des arbres construits par CL^∇ et CL^Δ. Initialement, la propriété est vraie, puis à chaque étape de dérivation, il suffit de constater que de nouvelles variables sont générées et d'utiliser la propriété précédente (avec i=1 puisque la résolution OLD est considérée par défaut) ❑

2.2. Approximation par combinaison

Pour ce modèle d'approximation, une fonction de combinaison, notée cmb, est introduite. Cette fonction possède comme arguments un couple composé d'une contrainte abstraite c^∇ et d'une contrainte concrète c^Δ et retourne comme résultat une contrainte abstraite prenant en compte toute l'information de c^Δ et c^∇.

Définition 5.14

Une fonction de combinaison est une application cmb définie de $\mathcal{L}^\nabla \times \mathcal{L}^\Delta$ vers $\mathcal{L}^\nabla$ telle que : $\forall\ (c^\Delta, c^\nabla) \in \mathcal{L}^\Delta \times \mathcal{L}^\nabla$,
$\xi(c^\Delta, c^\nabla) \Rightarrow \forall\ c^{\Delta'} \in \mathcal{L}^\Delta : \xi(c^\Delta \wedge c^{\Delta'}, cmb(c^\nabla, c^{\Delta'}))$.

Pour la suite, on considère fixée une fonction de combinaison cmb. A partir de celle-ci, il est possible d'établir les notions de résolvant et d'étape de dérivation SLD mixtes.

Définition 5.15

Soit P^Δ un $C\mathcal{L}^\Delta$-programme et soit $G^\nabla : \leftarrow c^\nabla, a_1, \ldots, a_n$ un $C\mathcal{L}^\nabla$-but. Etant sélectionnés un atome a_i de G^∇ et une clause C^Δ_j de P^Δ, considérons une variante $h \leftarrow d^\Delta, bs$ de C^Δ_j (telle que $a_i = h$ et telle que les autres variables de cette variante soient nouvelles). Si $cmb(c^\nabla, d^\Delta)$ est satisfiable alors le $C\mathcal{L}^\nabla$-but $H^\nabla : \leftarrow cmb(c^\nabla, d^\Delta), a_1, \ldots, a_{i-1}, bs, a_{i+1}, \ldots, a_n$ est appelé le (i,j)-résolvant mixte de (P^Δ, G^∇).

Une étape de dérivation SLD mixte consiste à calculer un résolvant mixte. Pour le reste, les choses (arbre SLD mixte, interprétation SLD mixte, stratégie de parcours, résolution OLD mixte, ...) sont définies de manière usuelle. Il reste simplement à noter que Res^{mix} désigne l'application qui pour tout couple (P^Δ, G^∇) composé d'un $C\mathcal{L}^\Delta$-programme P^Δ et d'un $C\mathcal{L}^\nabla$-but G^∇ associe l'arbre OLD mixte construit à partir de (P^Δ, G^∇). Cette application est appelée la sémantique opérationnelle mixte de $(C\mathcal{L}^\Delta, C\mathcal{L}^\nabla, cmb)$.

Nous cherchons à établir (via ξ) l'approximation de la sémantique opérationnelle de $C\mathcal{L}^\Delta$ par la sémantique opérationnelle mixte de $(C\mathcal{L}^\Delta, C\mathcal{L}^\nabla, cmb)$. Cette approximation est dite par combinaison. Pour toute analyse le but concret est substitué (abstrait) par un but abstrait tandis que le programme concret est conservé tel quel. La différence essentielle de cette méthode par rapport à la précédente tient donc au fait que l'abstraction (du programme) est retardée (puisque prise en compte par la fonction cmb). Elle est à comparer, entre autre, à la méthode de [Marriott et Sondergaard 90b] car même si le contexte est relativement différent, les fonctions c et r que les auteurs utilisent jouent un rôle similaire à la fonction cmb.

En reprenant la définition 3.5 (à une technique de renommage près), on note $\xi(Res^\Delta, Res^{mix})$ et on dit que Res^{mix} approche Res^Δ via ξ ssi

$\forall$ $(P^{\Delta},G^{\Delta}) \in CL^{\Delta}\text{-Prog}_{\times}CL^{\Delta}\text{-Goal}$, $\forall$ $G^{\nabla} \in CL^{\nabla}\text{-Goal}$,
$\xi_r(G^{\Delta},G^{\nabla}) \Rightarrow \xi_r(Res^{\Delta}(P^{\Delta},G^{\Delta}),Res^{mix}(P^{\Delta},G^{\nabla}))$

La propriété suivante établit la correction du processus élémentaire de l'approximation.

Propriété 5.16

$\forall$ $(P^{\Delta},G^{\Delta}) \in CL^{\Delta}\text{-Prog}_{\times}CL^{\Delta}\text{-Goal}$, $\forall$ $G^{\nabla} \in CL^{\nabla}\text{-Goal}$. Si $\xi_r(G^{\Delta},G^{\nabla})$ et si il est possible de calculer le (i,j)-résolvant, noté H^{Δ}, de (P^{Δ},G^{Δ}) alors il est possible de calculer le (i,j)-résolvant mixte, noté H^{∇}, de (P^{Δ},G^{∇}). De plus $\xi_r(H^{\Delta},H^{\nabla})$

Preuve

Soient $G^{\Delta} : \leftarrow c^{\Delta},a^{\Delta}{}_1,\ldots,a^{\Delta}{}_n$ et $G^{\nabla} : \leftarrow c^{\nabla},a^{\nabla}{}_1,\ldots,a^{\nabla}{}_n$
$\xi_r(G^{\Delta},G^{\nabla}) \Rightarrow \exists\ \rho_2 \in \mathcal{R}en$ tel que $\xi(\rho_2(G^{\Delta}),G^{\nabla})$.

On sait (par hypothèse) qu'une variante $C^{\Delta'} : h^{\Delta'} \leftarrow d^{\Delta'},bs^{\Delta'}$ de $C^{\Delta}{}_j$ est considérée telle que $h^{\Delta'} = a^{\Delta}{}_i$, donc $\exists\ \rho_1 \in \mathcal{R}en$ tel que $C^{\Delta'} = \rho_2(C^{\Delta}{}_j)$.

On sait que :
$h^{\Delta'} = a^{\Delta}{}_i$
$\rho_2(a^{\Delta}{}_i) = a^{\nabla}{}_i$
$h^{\Delta'} = \rho_1(h^{\Delta})$ où h^{Δ} est la tête de clause de $C^{\Delta}{}_j$
On en déduit : $\rho_2 o \rho_1(h^{\Delta}) = a^{\nabla}{}_i$.

Il existe donc une variante $C^{\Delta''} : h^{\Delta''} \leftarrow d^{\Delta''},bs^{\Delta''}$ de $C^{\Delta}{}_j$ telle que $h^{\Delta''} = a^{\nabla}{}_i$. $C^{\Delta'}$ et $C^{\Delta''}$ sont deux variantes d'un même clause, aussi il existe une substitution $\rho_3 \in \mathcal{R}en$ tel que $\rho_3(C^{\Delta'}) = C^{\Delta''}$.

A partir de ρ_2 et ρ_3 il est possible de construire une substitution de renommage ρ telle que :
$\forall$ $X \in$ Var(G^{Δ}), $\rho(X) = \rho_2(X)$.
$\forall$ $X \in$ Var($C^{\Delta'}$), $\rho(X) = \rho_3(X)$.

On utilise essentiellement a) $\forall$ $X \in$ Var($a^{\Delta}{}_i$), $\rho_2(X) = \rho_3(X)$ (en effet, $h^{\Delta'} = a^{\Delta}{}_i$, $\rho_2(a^{\Delta}{}_i) = a^{\nabla}{}_i$, $\rho_3(h^{\Delta'}) = h^{\Delta''}$ et $h^{\Delta''} = a^{\nabla}{}_i$), b) Var($G^{\Delta}$) $\cap$ Var($C^{\Delta'}$) = Var($a^{\Delta}{}_i$), c) Var(G^{∇}) $\cap$ Var($C^{\Delta''}$) = Var($a^{\Delta}{}_i$).

On sait que $\xi(\rho_2(c^{\Delta}),c^{\nabla}) \Rightarrow \xi(\rho_2(c^{\Delta}) \wedge d^{\Delta''},cmb(c^{\nabla},d^{\Delta''}))$. Or $\rho_3(d^{\Delta'}) = d^{\Delta''}$, aussi $\xi(\rho_2(c^{\Delta}) \wedge \rho_3(d^{\Delta'}),cmb(c^{\nabla},d^{\Delta''}))$.
On obtient donc $\xi(\rho(c^{\Delta} \wedge d^{\Delta'}),cmb(c^{\nabla},d^{\Delta''}))$.

On sait que $c^{\Delta} \wedge d^{\Delta'}$ est satisfiable (par hypothèse), aussi on en déduit que $cmb(c^{\nabla},d^{\Delta''})$ est satisfiable car ξ est valide et $[c^{\Delta} \wedge d^{\Delta'}] = [\rho(c^{\Delta} \wedge d^{\Delta'})]$.

Le (i,j)-résolvant de (P^Δ,G^Δ) est le CL^Δ-but suivant :
$H^\Delta : \leftarrow c^\Delta \wedge d^{\Delta'}, a^\Delta{}_1, \ldots, a^\Delta{}_{i-1}, bs^{\Delta'}, a^\Delta{}_{i+1}, \ldots, a^\Delta{}_n$

Le (i,j)-résolvant de (P^Δ,G^∇) est le CL^∇-but suivant :
$H^\nabla : \leftarrow cmb(c^\nabla, d^{\Delta''}), a^\nabla{}_1, \ldots, a^\nabla{}_{i-1}, bs^{\Delta''}, a^\nabla{}_{i+1}, \ldots, a^\nabla{}_n$

$\xi_r(H^\Delta, H^\nabla)$ car $\rho \in \mathcal{R}en$ et $\xi(H^\Delta, \rho(H^\nabla))$ ❑

A partir de la propriété précédente, il découle que Res^{mix} approche Res^Δ via ξ. La preuve est similaire à celle de la section précédente.

Propriété 5.17
Res^{mix} approche Res^Δ via ξ.

2.3. Approximation par extension

Nous supposons pour ce dernier modèle d'approximation que CL^∇ est un CLP-langage particulier, à savoir une extension de CL^Δ. Une extension de CL^Δ est définie comme suit :

Définition 5.18
$CL^\nabla = (\Sigma^\nabla, L^\nabla, \mathcal{A}^\nabla)$ est une extension de $CL^\Delta = (\Sigma^\Delta, L^\Delta, \mathcal{A}^\Delta)$ ssi :
- $\Sigma^\Delta = (S, \mathcal{F}^\Delta, C^\Delta)$, $\Sigma^\nabla = (S, \mathcal{F}^\nabla, C^\nabla)$, $\mathcal{F}^\Delta \subseteq \mathcal{F}^\nabla$ et $C^\Delta \subseteq C^\nabla$
- $L^\Delta \subseteq L^\nabla$
- $\mathcal{A}^\Delta$ et $\mathcal{A}^\nabla$ donnent la même interprétation des sortes de S et des symboles de $\mathcal{F}^\Delta$ et de C^Δ.

Nous supposons également que la relation d'approximation ξ est une relation particulière, à savoir la restriction sur $L^\Delta \times L^\nabla$ de la relation d'ordre $\vdash^\nabla$ définie sur CL^∇. Cette relation est encore, par abus de notation, notée $\vdash^\nabla$.

Propriété 5.19
$\vdash^\nabla$ est une relation d'approximation valide.

Il est immédiat que $\vdash^\nabla$ est close par renommage, conjonction et quantification existentielle et $\vdash^\nabla$ est valide. Nous pouvons donc établir (via $\vdash^\nabla$) l'approximation de la sémantique opérationnelle de CL^Δ par la sémantique opérationnelle de CL^∇. Cette approximation est dite par extension. Il s'agit d'un cas particulier d'approximation par abstraction.

Propriété 5.20
Res^∇ approche Res^Δ via $\vdash^\nabla$.

Il est possible à partir de ce modèle d'approximation de prouver la monotonie de la sémantique opérationnelle de tout CLP-langage. En effet, il est clair que pour tout CLP-langage CL : Res approche Res via $\vdash$.

Propriété 5.21
$\forall$ (P,G) $\in$ CL-Prog$\times CL$-Goal, $\forall$ (P',G') $\in$ CL-Prog$\times CL$-Goal,
(P,G) $\vdash_r$ (P',G') $\Rightarrow$ Res(P,G) $\vdash_r$ Res(P',G')

2.4. Quel modèle choisir ?

Etant donné un CLP-langage concret $CL^{\Delta} = (\Sigma^{\Delta}, \mathcal{L}^{\Delta}, \mathcal{A}^{\Delta})$ avec $\Sigma^{\Delta} = (S, \mathcal{F}^{\Delta}, C^{\Delta})$, on cherche à concevoir une sémantique abstraite qui soit une approximation de la sémantique opérationnelle de CL^{Δ}. Par rapport aux trois modèles que nous venons de présenter, cela revient donc à définir principalement un CLP-langage abstrait $CL^{\nabla} = (\Sigma^{\nabla}, \mathcal{L}^{\nabla}, \mathcal{A}^{\nabla})$ et une relation d'approximation $\xi \in \wp(\mathcal{L}^{\Delta} \times \mathcal{L}^{\nabla})$.

Notre idée est qu'il est tout à fait naturel de considérer que les signatures concrètes Σ^{Δ} et abstraites Σ^{∇} soient définies à partir d'un même ensemble de sortes S. En effet, ce que nous cherchons à calculer ce sont des propriétés abstraites qui caractérisent les données concrètes. Ces propriétés abstraites sont alors construites à partir d'un ensemble $\mathcal{F}^{\nabla}$ de symboles de fonction et d'un ensemble C^{∇} de symboles de contraintes. Un domaine de contraintes $\mathcal{L}^{\nabla}$ et une interprétation $\mathcal{A}^{\nabla}$ sont ensuite définis. Il apparaît naturel, à nouveau, que $\mathcal{A}^{\Delta}$ et $\mathcal{A}^{\nabla}$ donnent la même interprétation de S. $CL^{\nabla} = (\Sigma^{\nabla}, \mathcal{L}^{\nabla}, \mathcal{A}^{\nabla})$ avec $\Sigma^{\nabla} = (S, \mathcal{F}^{\nabla}, C^{\nabla})$ désigne un CLP-langage abstrait.

Notons $\Sigma^{\Delta\nabla} = (S, \mathcal{F}^{\Delta\nabla}, C^{\Delta\nabla})$ avec $\mathcal{F}^{\Delta\nabla} = \mathcal{F}^{\Delta} \cup \mathcal{F}^{\nabla}$ et $C^{\Delta\nabla} = C^{\Delta} \cup C^{\nabla}$. Soient $\mathcal{L}^{\Delta\nabla}$ la clôture par renommage, conjonction et quantification existentielle de $\mathcal{L}^{\Delta} \cup \mathcal{L}^{\nabla}$ et $\mathcal{A}^{\Delta\nabla}$ l'algèbre qui donne la même interprétation de S que $\mathcal{A}^{\Delta}$ (et donc $\mathcal{A}^{\nabla}$), la même interprétation des symboles de $\mathcal{F}^{\Delta}$ et C^{Δ} que $\mathcal{A}^{\Delta}$ et la même interprétation des symboles de $\mathcal{F}^{\nabla}$ et C^{∇} que $\mathcal{A}^{\nabla}$. $CL^{\Delta\nabla} = (\Sigma^{\Delta\nabla}, \mathcal{L}^{\Delta\nabla}, \mathcal{A}^{\Delta\nabla})$ désigne une extension de CL^{Δ} et de CL^{∇}. On note $CL^{\Delta\nabla} = CL^{\Delta} + CL^{\nabla}$ lorsque $CL^{\Delta\nabla}$ est obtenu comme indiqué ci-dessus.

Notons $\vdash$ la restriction sur $\mathcal{L}_1 \times \mathcal{L}_2$ de la relation d'ordre $\vdash^{\Delta\nabla}$ définie sur $\mathcal{L}^{\Delta\nabla}$ où $\mathcal{L}_1$ et $\mathcal{L}_2$ représentent indifféremment les langages $\mathcal{L}^{\Delta}$, $\mathcal{L}^{\nabla}$ et $\mathcal{L}^{\Delta\nabla}$. Ceci permet de ne pas alourdir les notations (et le contexte suffit généralement à déterminer la relation). Il est clair que quelque soit la manière dont elle est définie, la relation $\vdash$ est valide. Ceci permet de définir les approximations suivantes :

1 Res$^{\nabla}$ approche Res$^{\Delta}$ via $\vdash$ (définie sur $\mathcal{L}^{\Delta} \times \mathcal{L}^{\nabla}$).
$\Rightarrow$ il s'agit d'une approximation par abstraction.

2 Res^{mix} approche Res^{Δ} via $\vdash$ (définie sur $\mathcal{L}^{\Delta}\times\mathcal{L}^{\nabla}$)[†].
⇒ il s'agit d'une approximation par combinaison.

3 $\text{Res}^{\Delta\nabla}$ approche Res^{Δ} via $\vdash$ (définie sur $\mathcal{L}^{\Delta}\times\mathcal{L}^{\Delta\nabla}$).
⇒ il s'agit d'une approximation par extension.

De toutes ces approximations, l'approximation par extension est la plus précise car elle permet de déduire toutes les propriétés abstraites déduites par les autres approximations. En effet, la relation $\vdash$ (définie sur $\mathcal{L}^{\Delta}\times\mathcal{L}^{\Delta\nabla}$) permet une vision beaucoup plus fine de l'approximation puisqu'il est possible de conjuguer contrainte concrète et contrainte abstraite. Clairement, toute interprétation effectuée par Res^{∇} peut être effectuée par $\text{Res}^{\Delta\nabla}$. Comme, par ailleurs, $\text{Res}^{\Delta\nabla}$ traite à chaque étape une information de la forme $c^{\nabla}\wedge c^{\Delta}$, que Res^{mix} traite à chaque étape une information de la forme $cmb(c^{\nabla},c^{\Delta})$ et que $c^{\nabla}\wedge c^{\Delta} \vdash cmb(^{\nabla},c^{\Delta})$, on en déduit que toute interprétation effectuée par $\text{Res}^{\Delta\nabla}$ est plus précise, dans le même contexte, qu'une interprétation effectuée par Res^{mix}.

Le cas que nous venons d'étudier n'est pas tout à fait général puisque nous avons effectué deux hypothèses, à savoir,

- Σ^{Δ} et Σ^{∇} sont définies à partir d'un même ensemble de sortes S.
- $\mathcal{A}^{\Delta}$ et $\mathcal{A}^{\nabla}$ donnent la même interprétation de S

De plus, il ne semble pas possible de coder directement des propriétés abstraites non monotones (car alors il n'est plus possible d'utiliser $\vdash$ comme relation d'approximation). Toutefois, pour les propriétés monotones, le modèle d'approximation par extension a notre préférence car il est naturel (les propriétés abstraites sont définies directement à partir des données concrètes), simple (la relation d'approximation est l'implication logique) et précis. Avec ce modèle, la phase d'abstraction du domaine devient une phase d'extension du domaine. Evidemment, le problème de la terminaison reste posé. Cependant, ce problème concerne la phase suivante (phase d'abstraction du calcul) du modèle d'interprétation abstraite que nous proposons. Le fait de retarder l'abstraction jusqu'au moment du calcul est un gage de précision.

3. Analyse de l'assignation unique

On suppose fixé un CLP-langage concret $C\mathcal{L}^{\Delta} = (\Sigma^{\Delta},\mathcal{L}^{\Delta},\mathcal{A}^{\Delta})$ où $\Sigma^{\Delta} = (S,\mathcal{F}^{\Delta},C^{\Delta})$, on décide de construire une analyse de l'assignation unique pour $C\mathcal{L}^{\Delta}$. Cette analyse (appelée "definiteness analysis" par [Marriott et Sondergaard 90])

[†] pour toute fonction de combinaison cmb définie de $\mathcal{L}^{\nabla}\times\mathcal{L}^{\Delta}$ sur $\mathcal{L}^{\nabla}$

consiste à déterminer quelles sont les variables qui sont contraintes pendant la résolution à une assignation unique.

3.1. Principe de l'analyse

Nous désirons élaborer cette analyse par rapport aux trois modèles d'approximation vus ci-dessus. Commençons par poser :

$\mathcal{F}^\nabla = \emptyset$ et $C^\nabla = \{=,\bot,\top\} \cup \{\text{ground}_s \mid s \in S\}$

L'ensemble C^∇ contient, mis à part les symboles de l'ensemble $\{=,\bot,\top\}$, un symbole de contrainte ground_s par sorte s de S. Ceci permet de construire des contraintes abstraites de la forme $\text{ground}_s(X)$. On considère $\mathcal{L}^\nabla$ comme étant l'ensemble des formules du premier ordre qu'il est possible de construire à partir de $\mathcal{A}tom(\mathcal{V},\mathcal{F}^\nabla,C^\nabla)$ et on considère la Σ^∇-algèbre $\mathcal{A}^\nabla$ qui donne la même interprétation de S que $\mathcal{A}^\Delta$ et qui donne une interprétation des éléments de C^∇ au moyen de l'équivalence logique suivante : $\forall X \in \mathcal{V}$,

$\text{ground}_s(X)$ ssi $\vee \{c^\Delta \in \mathcal{L}^\Delta \mid X$ est contraint à une assignation unique par $c^\Delta\}$

$C\mathcal{L}^\nabla = (\Sigma^\nabla,\mathcal{L}^\nabla,\mathcal{A}^\nabla)$ avec $\Sigma^\nabla = (S,\mathcal{F}^\nabla,C^\nabla)$ désigne un CLP-langage abstrait. $C\mathcal{L}^{\Delta\nabla} = (\Sigma^{\Delta\nabla},\mathcal{L}^{\Delta\nabla},\mathcal{A}^{\Delta\nabla})$ désigne le CLP-langage $C\mathcal{L}^\Delta + C\mathcal{L}^\nabla$. La relation d'ordre $\vdash$ (définie sur $\mathcal{L}^\Delta \times \mathcal{L}^\nabla$) est une relation qui est caractérisée par une fonction d'abstraction α définie par :

$\forall\, c^\Delta \in \mathcal{L}^\Delta,\ \alpha(c^\Delta) = \wedge\{c^\nabla \in \mathcal{L}^\nabla \mid c^\Delta \vdash c^\nabla\}$

α est une fonction d'abstraction car :

(1) $\forall\, c^\Delta \in \mathcal{L}^\Delta,\ c^\Delta \vdash \top$

(2) $\vdash$ est transitive

(3) $\forall\, c^\nabla \in \mathcal{L}^\nabla,\ c^\Delta \vdash c^\nabla \Rightarrow \alpha(c^\Delta) \vdash c^\nabla$

Il est possible de définir une connexion de Galois (α',γ') entre $(\wp(\mathcal{L}^\Delta),\subseteq)$ et $(\mathcal{L}^\nabla,\vdash)$:

$\forall\, S^\Delta \in \wp(\mathcal{L}^\Delta),\ \alpha'(S^\Delta) = \vee\{\alpha(c^\Delta) \mid c^\Delta \in S^\Delta\}$

$\forall\, c^\nabla \in \mathcal{L}^\nabla,\ \gamma'(c^\nabla) = \{c^\Delta \in \mathcal{L}^\Delta \mid c^\Delta \vdash c^\nabla\}$

(α',γ') est une connexion de Galois car :

(1) α' est monotone.

Si $S^\Delta \subseteq S^{\Delta'}$ alors $\alpha'(S^\Delta) \vdash \alpha'(S^{\Delta'})$ car il existe une contrainte abstraite $c^\nabla \in \mathcal{L}^\nabla$ telle que $\alpha'(S^{\Delta'}) = \alpha'(S^\Delta) \vee c^\nabla$.

(2) γ' est monotone.

Evident.

(3) $\forall\, S^\Delta \in \wp(\mathcal{L}^\Delta),\ S^\Delta \subseteq \gamma'(\alpha'(S^\Delta))$.

Si $c^\Delta \in S^\Delta$ alors $c^\Delta \vdash \alpha(c^\Delta) \vdash \alpha'(S^\Delta)$. De ce fait, $c^\Delta \in \gamma(\alpha'(S^\Delta))$.

(4) $\forall\ c^{\nabla} \in \mathcal{L}^{\nabla}$, $\alpha'(\gamma'(c^{\nabla})) \vdash c^{\nabla}$.
$\forall\ c^{\Delta} \in \gamma'(c^{\nabla})$, $\alpha(c^{\Delta}) \vdash c^{\nabla}$. On en déduit $\alpha'(\gamma'(c^{\nabla})) \vdash c^{\nabla}$.

On peut également définir une fonction de combinaison, notée *cmb*, par :
$\forall\ c^{\Delta} \in \mathcal{L}^{\Delta}$, $\forall\ c^{\nabla} \in \mathcal{L}^{\nabla}$, $cmb(c^{\Delta},c^{\nabla}) = \alpha'(\{c^{\Delta}\} \cup \gamma'(c^{\nabla}))$

cmb est une fonction de combinaison car :
$\forall\ d^{\Delta} \in \gamma'(c^{\nabla})$,
$c^{\Delta} \wedge d^{\Delta} \vdash \alpha(c^{\Delta} \wedge d^{\Delta}) \vdash \alpha(c^{\Delta}) \wedge \alpha(d^{\Delta}) \vdash \alpha(c^{\Delta}) \vee \alpha(d^{\Delta}) \vdash cmb(c^{\Delta},c^{\nabla})$.

Tous les éléments sont maintenant réunis pour définir les trois modèles d'approximations présentés à la section précédente.

3.2. Analyse de l'assignation unique pour CLP($\mathcal{FT}$)

L'analyse de l'assignation unique s'appelle analyse de la clôture quand on considère Prolog. De nombreux travaux ont été consacrés à cette analyse, et plus généralement à l'inférence de modes. Citons entre autre [Mellish 86, Debray et Warren 88, Bruynooghe et Janssens 88, Corsini 89, Cortesi et Filé 91].

Nous illustrons les trois approximations présentées au paragraphe 3.1 par rapport à CLP($\mathcal{FT}$) = Prolog. Cela signifie que $C\mathcal{L}^{\Delta}$ = CLP($\mathcal{FT}$). Pour obtenir une relation d'approximation pertinente, il est important de choisir l'univers de Herbrand étendu $\mathcal{T}erm(\mathcal{F},\mathcal{V})$ comme interprétation de la sorte term.

Soit le $C\mathcal{L}^{\Delta}$-programme P suivant :
q(X,Y) :- Y=f(X,Z), r(Z).
r(X) :- X=a.

Soit le $C\mathcal{L}^{\nabla}$-programme P' suivant :
q(X,Y) :- ground(Y) $\leftrightarrow$ ground(X) $\wedge$ ground(Z), r(Z).
r(X) :- ground(X).

Soit le $C\mathcal{L}^{\nabla}$-but G' suivant :
:- ground(X), q(X,Y).

P' est obtenu à partir de P en utilisant la fonction d'abstraction α. L'interprétation de (P',G') par Res^{∇} fournit comme solution ground(X) $\wedge$ ground(Y). L'interprétation mixte de (P,G') par Res^{mix} fournit également comme solution ground(X) $\wedge$ ground(Y). Enfin, l'interprétation de (P,G') par $Res^{\Delta\nabla}$ fournit comme solution ground(X) $\wedge$ Y=f(X,a). En se ramenant à une information purement abstraite (par la fonction *cmb* par exemple), on obtient ground(X) $\wedge$ ground(Y). Le résultat des différentes approximations est donc pour cet exemple essentiellement le même.

Soit le CL^{Δ}-programme P suivant :

```
q(X) : - X=f(Y,a).
r(X) :- X=f(b,Y).
```

Soit le CL^{∇}-programme P' suivant :

```
q(X) :- ⊤.
r(X) :- ⊤.
```

Soit le CL^{∇}-but G' suivant :

```
:- ⊤, q(X), r(X).
```

P' est obtenu à partir de P en utilisant α. Il faut noter que α(X=f(Y,a)) = ground(Y) $\leftrightarrow$ ground(X) mais comme Y est implicitement quantifié existentiellement, cela revient à α(X=f(Y,a)) = $\top$. L'interprétation de (P',G') par Res^{∇} fournit comme solution $\top$. L'interprétation mixte de (P,G') par Res^{mix} fournit également comme solution $\top$. Enfin, l'interprétation de (P,G') par $Res^{\Delta\nabla}$ fournit comme solution X=f(b,a). En se ramenant à une information purement abstraite (par la fonction *cmb* par exemple), on obtient ground(X). Cet exemple illustre donc la plus grande efficacité de l'approximation par extension.

Notons que le modèle d'approximation par abstraction se code en utilisant CLP(IB) [Codognet et Filé 92]. CLP(IB) est construit sur l'ensemble $\mathcal{P}rop$ des classes d'équivalences de formules propositionnelles. En tant que domaine abstrait, $\mathcal{P}rop$ a été introduit par [Marriot et Sondergaard 89] et reconsidéré par [Cortesi et al. 91].

3.3. Analyse de l'assignation unique pour CLP(IR)

Nous illustrons cette fois-ci les trois approximations présentées au paragraphe 3.1 dans le contexte de CLP(IR). Cela signifie simplement que CL^{Δ} = CLP(IR). Il est nécessaire d'ajouter un élément particulier au domaine des réels IR pour obtenir, comme dans le cas de CLP($\mathcal{FT}$), une relation d'approximation pertinente. Sans cela, toute variable libre X serait approchée par ground(X).

Soit le CL^{Δ}-programme P suivant :

```
mortgage(P,T,I,R,B) :- T=1,
B+R=P*(1+I/1200).

mortgage(P,T,I,R,B) :- T>1,
P'=P*(1+I/1200), T'=T-1, mortgage(P',T',I,R,B)
```

Soit le CL^{∇}-programme P' suivant :

mortgage(P,T,I,R,B) :- T,
$(B \wedge R \wedge P \rightarrow I) \wedge (B \wedge R \wedge I \rightarrow P) \wedge$
$(B \wedge P \wedge I \rightarrow R) \wedge (I \wedge R \wedge P \rightarrow B)$

mortgage(P,T,I,R,B) :-
$(I \wedge P \rightarrow P') \wedge (I \wedge P' \rightarrow P) \wedge (P \wedge P' \rightarrow I)$
$T' \leftrightarrow T$, mortgage(P',T',I,R,B).

Soit le CL^{∇}-but G' suivant :
:- $P \wedge I \wedge R$, mortgage(P,T,I,R,B).

P' est obtenu à partir de P en utilisant α. Pour simplifier (l'écriture), nous avons codé P' et G' en CLP(IB). [Codognet et Filé 92] ont en effet montré, comme pour Prolog, que ce codage était possible pour les équations linéaires. L'interprétation de (P',G') par Res^{∇} fournit comme solution : $P \wedge I \wedge R \wedge T \wedge B$. L'interprétation mixte de (P,G') par Res^{mix} fournit également la même solution. Grâce à une technique de tabulation, il est possible d'assurer la terminaison de la résolution. Si on se limite à une information purement abstraite, l'interprétation de (P,G') par $Res^{\Delta\nabla}$ ne fournit pas de précision supplémentaire sur cet exemple.

Chapitre 6

Abstraction du Calcul

Dans ce chapitre, nous présentons la seconde phase (phase d'abstraction du calcul) du modèle d'interprétation abstraite que nous développons dans cette thèse. Cette phase consiste à assurer la terminaison du calcul de la sémantique (abstraite).

La tabulation est un moyen d'éviter certaines phases de calcul redondantes. Dans un premier temps, nous introduisons la notion de sous-réfutation et démontrons la consistance du principe de tabulation par rapport à tout CLP-langage (section 1). Puis, nous présentons la résolution OLDT (section 2) qui est une extension de la résolution OLD incorporant une technique de tabulation. A l'origine, [Tamaki et Sato 86] ont formalisé cette technique pour Prolog. Nous la généralisons pour tout CLP-langage, et nous montrons que la résolution OLDT est complète vis à vis de la résolution OLD par rapport à la sémantique du flux de données (section 3). Dans certains cas (i.e, pour certains CLP-langages), la tabulation n'est pas un moyen suffisant pour assurer la terminaison du calcul. Il est alors nécessaire d'abstraire le résolution OLDT : la résolution OLDT abstraite est appelée résolution AOLDT (section 4). Celle-ci est composée de deux étapes : une première étape appelée étape de widening qui a pour objectif de fournir une approximation finie de la résolution OLDT et une seconde étape appelée étape de narrowing qui a pour objectif de corriger sensiblement cette approximation.

1 Introduction à la tabulation

Dans cette section, nous introduisons quelques notions essentielles à la technique de tabulation qui est présentée au paragraphe 2. Cette technique est basée sur la notion de sous-réfutation, c'est à dire sur une conception locale

(atomique) de la résolution. Il est nécessaire de montrer la consistance du principe de cette technique par rapport à tout CLP-langage.

La notion de sous-réfutation est spécifique à la résolution OLD (voir chapitre 1) et nécessaire à la compréhension de la tabulation. Voici la définition proposée par [Tamaki et Sato 86].

Définition 6.1

Considérons dans un arbre OLD, un chemin m d'un noeud u à l'un de ses descendants, un noeud w, tel que pour chaque noeud v (différent de w) de ce chemin, on ait size(v) > size(w). Soit n_1 = size(u), n_2 = size(w) et k = n_1 - n_2, le chemin m peut être considéré comme une réfutation des k premiers atomes du noeud u si on ne tient pas compte des n_2 derniers atomes. C'est pourquoi le chemin m est appelé une k-sous-réfutation du noeud u. Celle-ci est appelée sous-réfutation atomique lorsque k = 1.

Les définitions suivantes d'appel et de solution atomiques sont fondamentales pour la suite.

Définition 6.2

Soient T un arbre OLD et v un noeud de T tel que label(v) = $(c,a_1,\ldots,a_n)$. L'appel atomique de v, noté $appel_1(v)$, désigne l'atome contraint (c,a_1). Si m est une sous-réfutation atomique de v alors la solution atomique de v associée à m désigne la contrainte cstr(m). L'ensemble des solutions atomiques de v est notée $solutions_1(v)$, i.e., $solutions_1(v)$ = {cstr(m) tel que m est une sous-réfutation atomique de v}.

Dans la définition 6.2, il convient de rappeler que les quantifications sont implicites. Ainsi, (c,a_1) est mis pour $(c{\downarrow}a_1,a_1)$ et cstr(m) pour cstr(m)$\downarrow a_1$. La notion de sous-réfutation est liée à celle d'extraction. Soit T un arbre OLD, extraire un arbre T' à partir d'un noeud v de T consiste à ne considérer que la partie de T visible depuis $appel_1(v)$. Il est facile de généraliser en prenant en compte les p premiers atomes de l'étiquette de v.

Soient T un arbre OLD et v un noeud (non vide) de T, l'extraction d'un arbre T', noté extract(T,$appel_1(v)$), est obtenu par extraction à partir de T en ne considérant que le premier atome de l'étiquette de v. On construit T' par induction (algorithme 6.3).

Soient T un arbre OLD et v un noeud de T. Un problème important est de déterminer si l'ensemble des solutions atomiques de v, i.e. $solutions_1(v)$, dépend exclusivement de l'appel atomique de v, i.e. $appel_1(v)$? En d'autres termes, est-ce que la résolution du premier atome de l'étiquette d'un noeud est

indépendante du reste de l'étiquette de ce noeud ? La réponse est positive. Ceci assure la consistance du principe de tabulation.

• Initialement

La racine i' de T' est tel que :
label(i') = (c,a_1) où label(v) = (c,a_1,...a_n).
i' est associé à i

• Inductivement

Soit un noeud (non vide) v' de T' associé à un noeud v de T,
Pour tout arc (v,w) de T, il existe un arc (v',w') de T' tel que :
cstr(v,w) = cstr(v',w')
num(v,w) = num(v',w')
label(w') = (c,a_1,...a_{n-d}) où
label(w) = (c,a_1,...a_n)
d = size(v) -size(v')
w' est associé à w.
Fin pour

Algorithme 6.3 : extraction d'un arbre OLD

Propriété 6.4

Soient P un CL-programme et G un CL-but. Soit T un arbre OLD construit à partir de (P,G) et soit v un noeud de T. L'arbre T' construit à partir de (P,appel$_1$(v)) est identique (à un renommage prés) à l'arbre T" = extract(T,appel$_1$(v)).

Preuve

On procède par induction sur la profondeur des arbres T' et T".

Initialement, la racine i' de T' et la racine i'' de T" sont tels que label(i') = label(i'').

Imaginons maintenant que T' et T" soient identiques (à un renommage ρ près) jusqu'à une profondeur n. Soient un couple (v',v'') de noeuds respectifs de T' et T" tels que :
level(v') = level(v'') = n et ρ(label(v') = label(v'').

Posons label(v') = (c',a_1',...,a_m') et label(v'') = (c'',a_1'',...,a_m''). On sait que v'' est issu d'un noeud v de T tel que label(v) = (c,a_1,...,a_n) et m ≤ n.

Soit C une clause de P. Considérons une variante (d',h',bs') de C par rapport à v' et une variante (d,h,bs) de C par rapport à v. Il est clair qu'on peut étendre ρ de telle sorte que ρ(d',h',bs') = (d,h,bs)

D'une part, $c' \wedge d' \neq \bot$ ssi $\rho(c' \wedge d') \neq \bot$ ssi $\rho(c') \wedge \rho(d') \neq \bot$ ssi $c'' \wedge d \neq \bot$
Nous utilisons essentiellement la propriété 2.17

D'autre part, soit $V = Var_{free}(v) - Var_{free}(v'')$, on sait que :
$c = c''\downarrow V$
$d = d\downarrow V$

Aussi, $c'' \wedge d \neq \bot$ ssi $(c'' \wedge d)\downarrow V \neq \bot$ ssi $c''\downarrow V \wedge d\downarrow V \neq \bot$ ssi $c \wedge d \neq \bot$
Nous utilisons essentiellement la propriété 2.17

On en déduit donc que $c' \wedge d' \neq \bot$ ssi $c \wedge d \neq \bot$.

Ainsi, à tout noeud fils w' de v' correspond un noeud fils w de v (et réciproquement) tel que : $cstr(v,w) = cstr(v',w')$ et $num(v,w) = num(v',w')$. On en déduit (par définition de l'extraction d'un arbre) que pour tout noeud fils w' de v' correspond un noeud fils w'' de v'' tel que : $cstr(v'',w'') = cstr(v',w')$ et $num(v'',w'') = num(v',w')$.

On sait que :
$label(w') = (c' \wedge d', bs', a_2', \ldots, a_m')$
$label(w) = (c \wedge d, bs, a_2, \ldots, a_n)$
$label(w'') = (c'' \wedge d, bs, a_2, \ldots, a_m)$

On montre facilement que $\rho(label(w') = label(w'')$.

En généralisant ce raisonnement, on conclut que T' et T'' sont identiques (à un renommage près). ❑

Le corollaire suivant découle directement de cette propriété.

Corollaire 6.5
Soit T un arbre OLD et soient v et v' deux noeuds de T,
si $appel_1(v) = appel_1(v')$ alors $solutions_1(v) = solutions_1(v')$

2 Résolution OLDT

[Tamaki et Sato 86] ont proposé une extension de la résolution OLD : la résolution OLDT, c'est à dire la résolution OLD avec une technique de tabulation. L'intérêt de cette extension est d'éviter les calculs redondants. Via la mémorisation du résultat de certains calculs dans une table, il est possible de réutiliser ceux-ci par simple consultation. Pour le même résultat, [Vieille 89] utilise le terme de SLD-AL résolution où AL signifie "Admissibility test and Lemma resolution". A chaque étape de résolution, un test est effectué pour savoir si le but courant est nouveau (admissibility test). Si ce n'est pas le cas, les

solutions trouvées pour les occurences précédentes (de ce but) sont utilisées (lemma resolution).

La résolution OLDT est définie dans [Tamaki et Sato 86] par rapport à Prolog. Nous étendons cette technique de tabulation par rapport à tout CLP-langage $\mathcal{CL}$. Ceci ne pose pas de problème puisque nous avons montré la correction du principe de tabulation par rapport à tout CLP-langage (propriété 6.4). Une interprétation OLDT d'un couple (P,G) constitué d'un $\mathcal{CL}$-programme P et d'un $\mathcal{CL}$-but G consiste à construire une structure OLDT. Une structure OLDT $\mathcal{S}tr$ = (F,AT,PT) est composée d'une forêt (de dérivation) OLDT F et de deux tables, la table des noeuds actifs (notée AT) et la table des noeuds passifs (notée PT). En fait, lorsque aucune abstraction n'est considérée (ce qui est le cas dans cette partie), la forêt se compose d'un seul arbre. Afin d'introduire une certaine souplesse dans l'utilisation des tables, seuls certains symboles de prédicat (choisis par l'utilisateur) sont concernés par la tabulation. Ces symboles de prédicat sont appelés t-symboles de prédicat (t est mis pour tabulation). Un atome est appelé un t-atome ssi le symbole de prédicat de cet atome est un t-symbole. Un noeud v est appelé un t-noeud ssi $appel_1(v)$ est un t-atome. Un t-noeud v est soit un noeud actif, soit un noeud passif. Si v est un noeud actif alors une entrée de AT correspond à v. Cette entrée est caractérisée par l'appel atomique de v et par une liste de solutions notée list(v). Si v est un noeud passif alors une entrée de PT correspond à v. Cette entrée est caractérisée par l'appel atomique de v et par un pointeur noté pos(v), indiquant une certaine position dans la liste de solutions list(v') associée à un noeud actif v' de AT tel que $appel_1(v) \vdash_r appel_1(v')$. Clairement tout noeud passif v est liée à un noeud actif v'. On dit que v pointe sur v' .

La classification d'un t-noeud comme noeud actif ou comme noeud passif s'opère de la manière suivante. Soit $\mathcal{S}tr$ = (F,AT,PT) une structure OLDT et soit v un t-noeud dont la classification doit être effectuée, deux possibilités se présentent (algorithme 6.6).

```
Si il existe un noeud actif v' tel que appel1(v) ⊢r appel1(v') alors
        -- v est enregistré comme noeud passif
        créer une entrée pour v dans PT tel que pos(v) désigne le début de list(v')
Sinon
        -- v est enregistré comme noeud actif
        créer une entrée pour v dans AT tel que list(v) est initialisée à la liste vide
Fin si
```

Algorithme 6.6 : A) Classification d'un t-noeud

Il faut noter que plusieurs noeuds actifs v' peuvent satisfaire la condition $appel_1(v) \vdash_r appel_1(v')$. Il est alors nécessaire de sélectionner l'un d'entre eux.

Un interpréteur OLDT effectue des interprétations OLDT, et tout comme les interpréteurs OLD, est caractérisé par une stratégie de parcours sp. La résolution $OLDT_{sp}$ désigne le calcul effectué par un interpréteur OLDT utilisant une stratégie de parcours sp. Ce calcul consiste à construire une structure OLDT $\mathcal{S}tr$ à partir d'un $C\mathcal{L}$-programme P et d'un $C\mathcal{L}$-but G. La structure $\mathcal{S}tr$ est construite par étapes successives en considérant la stratégie de parcours sp. Initialement, une structure $\mathcal{S}tr_0 = (F_0, AT_0, PT_0)$ est considérée (algorithme 6.7).

F_0 est composée de l'arbre T réduit au noeud racine i tel que label(i) = G.
AT_0 contient une entrée pour i tel que list(i) est initialisée à la liste vide.
PT_0 est vide.

Algorithme 6.7 : B) Structure OLDT initiale

De façon inductive, la structure $\mathcal{S}tr_{i+1}$ s'obtient à partir de la structure $\mathcal{S}tr_i$ par extension. L'extension d'une structure OLDT consiste à effectuer soit une extension OLD, soit une extension par consultation. L'extension appliquée (et la manière dont elle est appliquée) dépend de la stratégie de parcours sp. Si aucune extension n'est possible, alors le calcul est terminé. Soit $\mathcal{S}tr_i = (F_i, AT_i, PT_i)$ une structure OLDT, soit v tel que label(v) = $(c, a_1, \ldots, a_n)$ et $n \geq 1$ le noeud sélectionné par la stratégie de parcours sp (algorithme 6.8).

Si v n'est pas un noeud passif alors
  -- **extension OLD**
1.1 Pour toute clause C_j de P
    on considère une variante (d, h, bs) de C_j ...
    Si $c \wedge d$ est satisfiable alors
      on ajoute un noeud fils w à v tel que
        label(w) = $(c \wedge d, bs, a_2, \ldots, a_n)$
      on ajoute un arc reliant v à w tel que
        cstr(v,w) = d et num(v,w) = j.
    Fin si
  Fin pour
1.2 On effectue la classification de chaque nouveau t-noeud w.
1.3 Pour toute sous-réfutation atomique m telle que m débute par un noeud actif u et finit par un nouveau t-noeud w
    on pose $appel_1(u) = (c, a)$ et d = cstr(m)$\downarrow a$.
    Si il n'existe pas d'élément d' dans list(u) tel que $c \wedge d \vdash c \wedge d'$
      on ajoute d à la fin de list(u).
    Fin si
  Fin pour
Sinon v est un noeud passif (qui pointe sur un noeud actif v')
  -- **extension par consultation**
2.1 Si pos(v) désigne une (sous-)liste de solutions non vide,

on considére le premier élément d' de cette sous-liste,
on avance le pointeur pos(v) d'une position.
Fin si
2.2 Soient $\rho \in \mathcal{R}en$ tel que $appel_1(v) \vdash \rho(appel_1(v'))$
Si $c \wedge \rho(d')$ est satisfiable alors
on ajoute à F_i un noeud w et un arc (v,w) tel que
label(w) = $(c \wedge \rho(d'), a_2, \ldots, a_n)$
cstr(v,w) = $\rho(d')$.
Fin si
2.3 On fait de même qu'en 1.3.
Fin si

Algorithme 6.8 : C) Extension élémentaire d'une structure OLDT

La numérotation des arcs issus d'un noeud passif n'est pas importante (et donc pas gérée). On considère comme résultat d'une interprétation OLDT la structure OLDT obtenue aprés ω extensions. Par soucis de simplicité on considérera (souvent) plus simplement qu'une forêt OLDT résulte de la résolution OLDT (les tables étant sous-entendues). Pour chaque arbre d'une forêt OLDT, les chemins et réfutations OLDT sont définis de manière analogue aux chemins et réfutations OLD.

[Tamaki et Sato 86] proposent une stratégie de parcours mdf "multi-étape et en profondeur d'abord". On note alors résolution $OLDT_{mdf}$ la résolution OLDT utilisant cette stratégie. Cette stratégie a l'avantage d'être complète si on introduit la td-abstraction non décrite ici. Pour notre part, nous introduisons la résolution $OLDT_{mbf}$, c'est-à-dire la résolution OLDT utilisant une stratégie de parcours mbf "multi-étape et en largeur d'abord". Chaque étape i de la résolution $OLDT_{mbf}$ consiste à considérer successivement (de gauche àdroite) tous les noeuds dont une extension est possible :

- soit une extension OLD pour un noeud actif
- soit une extension par consultation pour un noeud passif.

Pour un noeud passif, il est possible d'effectuer plusieurs extensions successives. On ne s'autorise qu'à utiliser les solutions trouvées avant l'étape i afin d'éviter un comportement consultatif infini (principe "multi-étape" de la stratégie).

Lorsque cela ne sera pas précisé, on considérera la stratégie de parcours mbf par défaut. Res^t désigne le calcul de la résolution OLDT. Cela signifie que si P est un CL-programme et si G est un CL-but alors $Res^t(P,G)$ désigne la forêt OLDT calculée par la résolution OLDT de (P,G). Trois types de noeuds peuvent apparaître dans une forêt OLDT : les noeuds actifs qui sont représentés par un rectangle, les noeuds passifs qui sont représentés par un oval et les noeuds vides

qui sont représentés par un petit carré. Les noeuds actifs et les noeuds passifs sont numérotés dans l'ordre de leur création. Deux types d'arcs peuvent également apparaître : les arcs actifs qui sont attachés aux noeuds actifs et sont représentés par un trait plein et les arcs passifs qui sont attachés aux noeuds passifs et sont représentés par un trait en pointillé. Nous illustrons maintenant la résolution OLDT avec quelques exemples.

Exemple 1

Nous reprenons l'exemple du chapitre 1, section 6. Nous rappelons ci-dessous le programme et le but considérés.

Soit le programme P suivant :

reach(X,Y) ← reach(X,Z), edge(Z,Y).
reach(X,X).
edge(a,b).
edge(a,c).
edge(b,a).
edge(b,d).

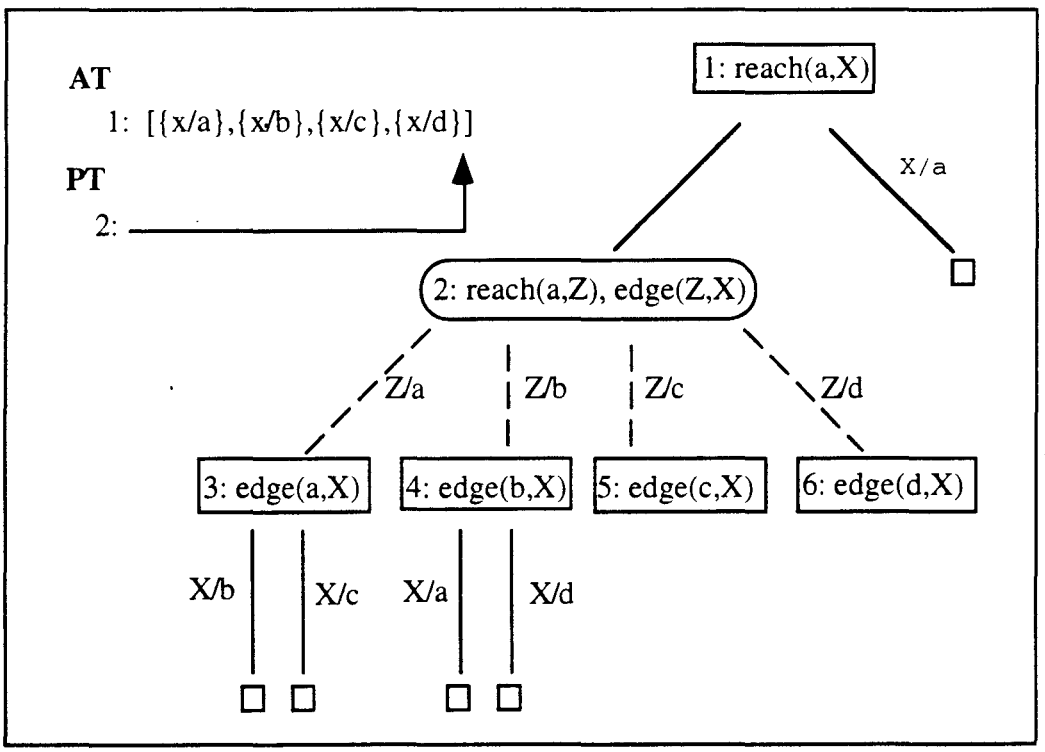

Figure 6.9

Soit le but G suivant :

$\leftarrow$ reach(a,X).

Si reach est un t-symbole de prédicat alors la résolution OLDT de (P,G) construit la forêt OLDT (réduite à un seul arbre) de la figure 6.9. Commentons la résolution OLDT de (P,G). Rappelons que dans le cas de Prolog, par simplicité, nous ne cherchons pas à homogénéiser les atomes. Tout d'abord un premier noeud actif (de numéro 1) est créé, son étiquette est reach(a,X) et une entrée est créée dans AT tel que list(1) = Ø. En utilisant la règle récursive, un second noeud est créé, mais ce noeud est passif car reach(a,Z) est instance de reach(a,X). Une entrée est créé dans PT tel que pos(2) pointe sur le début de list(1). Le noeud numéro 2 pointe donc sur le noeud numéro 1. En utilisant l'un des faits, une première solution {X/a} est trouvée et placée dans list(1). Cette solution peut être consultée par le noeud numéro 2 pour dériver un nouveau noeud dont l'étiquette est edge(a,X). Deux nouvelles solutions sont alors trouvées en résolvant ce noeud. Celles-ci peuvent alors être consultées par le numéro 2 ... A un certain moment, aucun noeud actif ne peut plus être résolu et pos(2) pointe sur une sous-liste de solutions vide. La résolution s'arrête donc. ❑

Exemple 2

Nous reprenons l'exemple du chapitre 2, section 6.2. Nous rappelons ci-dessous le programme et le but considérés.

Soit le programme P suivant :

p(X,Y,Z) :- ◊ q(X,Y), r(X,Y,Z).
q(X,Y) :- ¬X ◊.
q(X,Y) :- X ◊ q(Y,X).
r(X,Y,Z) :- Z=X∨Y ◊.

Soit le but G suivant :

:- ◊ p(X,Y,Z).

Si q est un t-symbole de prédicat alors la résolution OLDT de (P,G) construit la forêt OLDT (réduite à un seul arbre) de la figure 6.10. Le noeud passif 4 pointe sur le noeud actif 2. Ceci permet d'éviter la construction d'une branche infinie. ❑

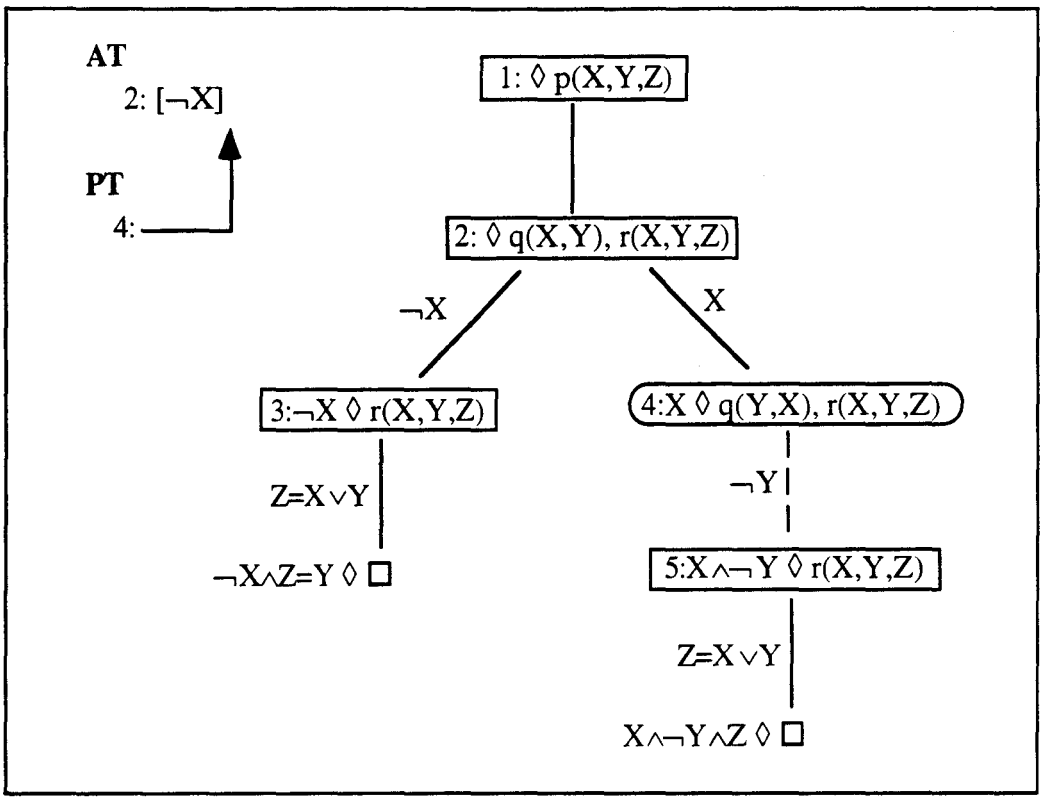

Figure 6.10

Même si elle termine pour les deux exemples précédents, la résolution OLDT, tout comme la résolution OLD, ne termine pas toujours. A cela deux raisons :

1 le nombre de noeuds actifs de la forêt construite peut être infini.

2 le nombre d'éléments d'une liste de solutions peut être infini.

Bien entendu, cela dépend du CLP-langage considéré. Pour CLP(IB), par exemple, on est certain que la résolution OLDT s'arrête puisque le nombre d'appels atomiques distincts (par rapport à un même ensemble de variables libres) est fini. En Prolog, par contre, les deux possibilités d'échec sont présentes.

Nous illustrons ces deux posibilités d'échec avec deux exemples simples.

Exemple 3

Soit le programme P suivant :

lt3(s(s(0))).
lt3(X) ← lt3(s(X)).

Soit le but G suivant :

← lt(s(0)).

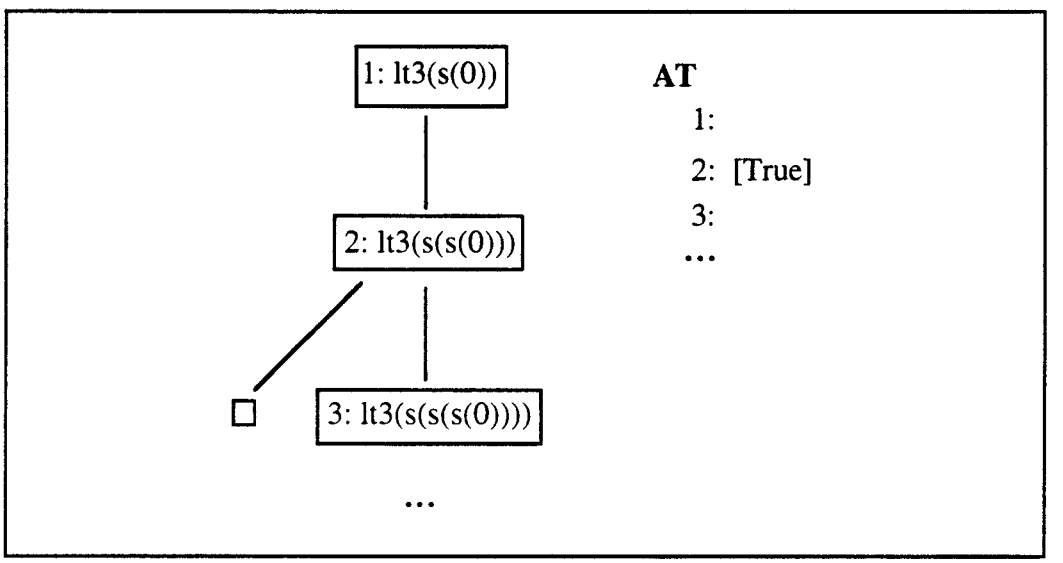

Figure 6.11

Si lt3 (lt3 est mis pour "less than 3") est un t-symbole de prédicat alors la résolution OLDT de (P,G) construit la forêt OLDT (réduite à un seul arbre) décrite par la figure 6.11. La résolution ne s'arrête pas car le nombre de noeuds actifs générés est infini. ❑

Exemple 4

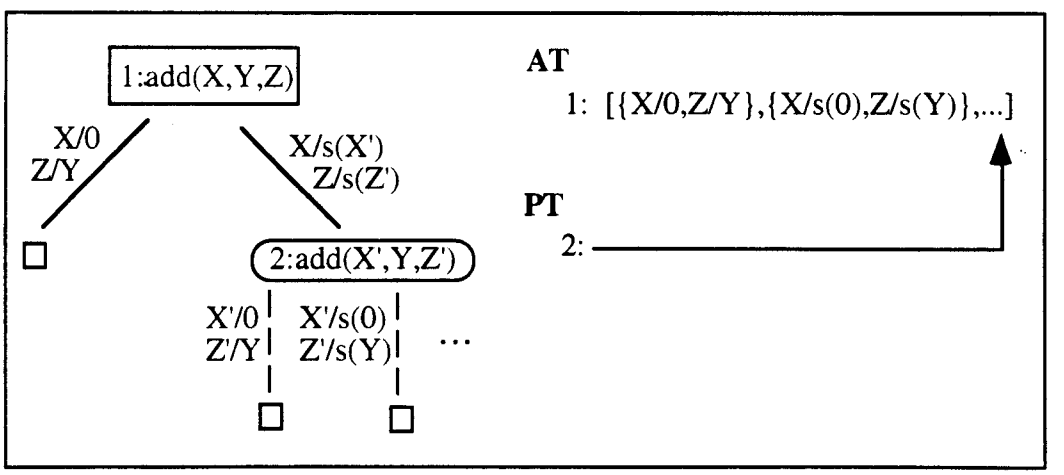

Figure 6.12

Soit le programme P suivant :

add(0,Y,Y).
add(s(X),Y,s(Z))← add(X,Y,Z).

Soit le but G suivant :

← add(X,Y,Z).

Si add est un t-symbole de prédicat alors la résolution OLDT de (P,G) construit la forêt OLDT décrite par la figure 6.12. La résolution ne s'arrête pas car le nombre de solutions est infini. ❑

3 Complétude de la résolution OLDT

Nous allons établir la complétude de la résolution OLDT vis à vis de la résolution OLD par rapport à une sémantique particulière : la sémantique du flux de données.

Définition 6.13

Soit T un arbre OLD, le flux de données associé à T est défini par :
$flux(T) = \{(appel_1(v), solutions_1(v))$ tel que v est un noeud de T$\}$.

Soit F une forêt OLDT, le flux de données associé à F est défini par :
$flux(F) = \bigcup_{T \in F} flux(T)$.

La sémantique du flux de données OLD est l'application qui à tout CL-programme P et tout CL-but G associe le flux de données flux(T) où T = Res(P,G). On définit de manière analogue la sémantique du flux de données OLDT. Pour établir la complétude de la résolution OLDT vis à vis de la résolution OLD par rapport à la sémantique du flux de données, il nous est nécessaire de présenter l'algorithme d'extraction d'une forêt OLDT. Cet algorithme est un peu plus compliqué que l'algorithme 6.3 d'extraction d'un arbre OLD mais le principe reste le même.

Soient F une forêt OLDT et v un noeud (non vide) de F, l'extraction d'une forêt F', notée extract(F,appel$_1(v)$), à partir du noeud v de F est obtenue en ne considérant que le premier atome de l'étiquette de v. On construit F' par induction (algorithme 6.14).

Cet algorithme est fini car le nombre de noeuds actifs pouvant être intégré dans F' diminue strictement à chaque étape. On ne considère pas les tables dans l'algorithme car elles peuvent être reconstituées à partir de la forêt extraite.

• Initialement

La racine i' de l'arbre principal de F' est tel que
label(i') = (c,a_1) où label(v) = $(c,a_1,\ldots a_n)$.
i' est associé à i

• Inductivement

Soit un noeud (non vide) v' de F' associé à un noeud v de F
Si v est un noeud actif alors
v' est un noeud actif
Pour tout arc actif (v,w) de F
il existe un arc actif (v',w') de F' tel que
cstr(v,w) = cstr(v',w') et num(v,w) = num(v',w')
label(w') = $(c,a_1,\ldots a_{n-d})$ où label(w) = $(c,a_1,\ldots a_n)$
et d = size(v) - size(v')
w' est associé à w.
Fin pour
Sinon v est un noeud passif (pointant sur un noeud u)
v' est un noeud passif
Si u a déja été associé à un noeud u' de F' alors
v' pointe sur u'
Sinon on créé un nouvel arbre dont la racine u' est tel que
label(u') = (c,a_1) où label(u) = $(c,a_1,\ldots a_n)$.
u' est associé à u
Finsi
Pour tout arc passif (v,w) de F
il existe un arc passif (v',w') de F' tel que
cstr(v,w) = cstr(v',w') et num(v,w) = num(v',w')
label(w') = $(c,a_1,\ldots a_{n-d})$ où label(w) = $(c,a_1,\ldots a_n)$
et d = size(v) - size(v')
w' est associé à w
Fin pour
Finsi

Algorithme 6.14 : extraction d'une forêt OLDT

Pour tout ce qui suit, on considère un CL-programme P, un CL-but G et on pose T = Res(P,G) et F = Res^t(P,G).

Lemme 6.15

Si v et v' sont deux noeud respectifs de T et F tel que appel$_1(v) \vdash_r$ appel$_1(v')$ alors pour toute sous-réfutation atomique m de v, il existe une sous-réfutation atomique m' de v' tel que cstr(m) $\vdash_r$ cstr(m').

La preuve qui suit est par récurrence. Elle est à rapprocher des preuves du même genre de [Tamaki et Sato 86] et [Kanamori et Kawamura 87,90].

preuve

La preuve s'effectue par récurrence sur le couple (m,v') en utilisant la relation d'ordre strict et bien fondé suivante (|m| désigne la longueur (en nombre d'arcs) du chemin m) :

(m,v) < (m',v') ssi |m| < |m'| ou |m| = |m'| et v est passif et v' est actif.

La propriété 6.4 nous indique qu'il est possible de raisonner non pas avec T mais avec extract(T,appel$_1(v)$) et par extension, qu'il est possible de raisonner non pas avec F mais avec extract(F,appel$_1(v')$). Pour simplifier, et comme cela revient au même, on suppose directement que label(v) = appel$_1(v)$ et label(v') = appel$_1(v')$.

Initialement : |m| = 1 et v' est actif.

m est constitué d'un seul arc (v,w). Comme appel$_1(v) \vdash_r$ appel$_1(v')$, en utilisant la propriété de monotonie de la résolution (propriété 5.23), on sait qu'il est possible de dériver un fils w' au noeud v' tel que num(v,w) = num(v',w'). Clairement, cstr(m) $\vdash_r$ cstr(m') puisque cstr(m) et cstr(m') désignent la même contrainte au nom des variables prés.

Itérativement : on suppose que la propriété est vraie pour tout couple strictement inférieur à (m,v') et on montre qu'elle est vraie pour (m,v').

- Si v' est un noeud passif

  Il existe un noeud actif v'' tel que v' pointe sur v''. On sait que, par transitivité, on a appel$_1(v) \vdash_r$ appel$_1(v') \vdash_r$ appel$_1(v'')$. De plus, (m,v'') < (m,v). Par récurrence, on en déduit qu'il existe une sous-réfutation atomique m'' de v'' tel que cstr(m) $\vdash_r$ cstr(m''). Soient $\rho \in \mathcal{R}en$ et $\rho' \in \mathcal{R}en$ tels que appel$_1(v) \vdash \rho$(appel$_1(v')$) et appel$_1(v') \vdash \rho'$(appel$_1(v'')$). On en déduit, cstr(m) $\vdash \rho(\rho'$(cstr(m'')$))$. Posons appel$_1(v)$ = (c,a), appel$_1(v')$ = (c',a'), d = cstr(m) et d'' = cstr(m'').

  $c \vdash \rho(c')$ et $d \vdash \rho(\rho'(d''))$ implique $c \wedge d \vdash \rho(c' \wedge \rho'(d''))$.

  Comme v' pointe sur v'' et qu'une stratégie "multi-étape et en largeur d'abord" est considérée, à un moment donné la solution atomique d'' est consultée par v'. Il en résulte une contrainte $c' \wedge \rho'(d'')$. Comme $c \wedge d \vdash \rho(c' \wedge \rho'(d''))$ et que $c \wedge d \neq \bot$, on en déduit que $\rho(c' \wedge \rho'(d'')) \neq \bot$. Il existe donc une sous-réfutation atomique m' de v' telle que cstr(m') = $\rho'(d'')$. De plus, cstr(m) $\vdash_r$ cstr(m') puisque cstr(m) $\vdash \rho'$(cstr(m')).

- Si v' est un noeud actif

On construit m' en associant chaque noeud de m' avec certains noeuds de m. Si un noeud v_i de m est associé avec un noeud v'_i de m' alors cela signifie que label(v_i) $\vdash_r$ label(v'_i) et (si i > 1) cstr(v_1,v_i) $\vdash_r$ cstr(v'_1,v'_i). On pose $v_1 = v$ et $v'_1 = v'$ et on associe v_1 avec v'_1.

Si à un certain moment un noeud v_i (non vide) est associé avec un noeud actif v'_i alors soit v_{i+1} le successeur de v_i par m. En utilisant la propriété de monotonie de la résolution (propriété 5.23), on sait qu'il existe un successeur v'_{i+1} de v'_i tel que num(v_i,v_{i+1}) = num(v'_i,v'_{i+1}). Clairement v_{i+1} peut être associé à v'_{i+1} (simple question de renommage).

Si à un certain moment un noeud v_i (non vide) est associé avec un noeud passif v'_i. Dans m, on trouve une sous-réfutation atomique p = (v_i,v_{i+1}) de v_i. Comme (p,v'_i) < (m,v'), par hypothèse de récurrence on sait qu'il existe une sous-réfutation atomique p' = (v'_i,v'_{i+1}) de v'_i tel que cstr(v_i,v_{i+1}) $\vdash_r$ cstr(v'_i,v'_{i+1}). Clairement v_{i+1} peut être associé à v'_{i+1} (simple question de renommage).

Ainsi, par induction on arrive à construire une sous-réfutation atomique m' de v' telle que cstr(m) $\vdash_r$ cstr(m'). ❑

La propriété suivante établit la complétude des solutions atomiques.

Propriété 6.16

$\forall$ (app,$sols$) $\in$ flux(T), $\forall$ (app',$sols'$) $\in$ flux(F),
$app \vdash_r app'$ implique $sols \vdash_r sols'$.

Preuve

Soient v un noeud de T et v' un noeud de F. Si appel$_1$(v) $\vdash_r$ appel$_1$(v') alors on sait (lemme 6.15) qu'à toute sous-réfutation atomique m de v, correspond une sous-réfutation atomique m' de v' tel que cstr(m) $\vdash_r$ cstr(m'). On en déduit directement que solutions$_1$(v) $\vdash_r$ solutions$_1$(v') ❑

La propriété suivante établit la complétude des appels atomiques.

Propriété 6.17

$\forall$ (app,$sols$) $\in$ flux(T), $\exists$ (app',$sols'$) $\in$ flux(F) tel que $app \vdash_r app'$.

Preuve

On commence par associer certains noeuds de T avec l'ensemble des noeuds de l'arbre principal (celui qui contient le premier noeud) de F. Si un noeud v de T est associé avec un noeud v' de F alors cela signifie que label(v) $\vdash_r$ label(v').

Initialement, la racine i de T est associée à la racine i' de F (on sait que label(i) $\vdash_r$ label(i')).

Récursivement, si un noeud v de T est associé à un noeud v' actif de T' alors par monotonie de la résolution, on sait qu'il est possible d'associer à chaque noeud fils w de v un noeud fils w' de v'. Si un noeud v de T est associé à un noeud v' passif de T' (pointant sur un noeud actif v'') alors par utilisation du lemme 6.12, on sait qu'il est possible d'associer à chaque noeud w de T tel que (v,w) constitue une sous-réfutation atomique de v un noeud fils w' de v'. Cependant, qu'en est-il des noeuds éventuels (autres que v et w) apparaissant sur le chemin de chaque sous-réfutation (v,w) de v. En fait, en posant T' = extract(T,appel$_1(v)$) et F' = extract(F,appel$_1(v'')$), il suffit de recommencer le raisonnement que nous venons d'exposer. On sait de plus qu'au moins le fils de v sur le chemin (v,w) pourra être associé à un noeud de F' puisque v'' est un noeud actif. Ainsi, finalement, chaque noeud de T peut être associé avec un noeud issu (directement ou indirectement par extraction) de F.

On en déduit qu'à tout noeud v de T, il est possible d'associer un noeud v' de F tel que appel$_1(v) \vdash_r$ appel$_1(v')$ ❑

Les propriétés 6.16 et 6.17 établissent la complétude de la résolution OLDT vis à vis de la résolution OLD par rapport à la sémantique du flux de données.

4 Résolution AOLDT

Dans certains cas (c'est à dire pour certains CLP-langages), il est nécessaire d'abstraire la résolution OLDT pour obtenir un comportement fini de celle-ci. La résolution OLDT abstraite est alors appelée résolution AOLDT. [Kanamori et Kawamura 87,90] semblent être les premiers à avoir combiné interprétation abstraite et résolution OLDT. [Gallagher et al. 88], [Lecoutre et al. 91], [Warren 92], [Codognet et File 92], [Boulanger et Bruynooghe 93], [Van Hentenryck et al. 93a] ont poursuivi dans cette voie. Nous tenterons plus loin une analyse de ces différents travaux et une comparaison avec le modèle que nous développons maintenant.

La résolution AOLDT que nous proposons est composée de deux étapes : une première étape appelée étape de widening et une seconde étape appelée étape de narrowing.

4.1 Etape de widening

L'étape de widening de la résolution AOLDT a pour objectif de fournir une approximation finie de la résolution OLDT. Nous commençons par donner la définition d'un opérateur de widening, puis nous présentons l'algorithme de l'étape de widening de la résolution AOLDT.

4.1.1 Opérateurs de widening

La définition d'un opérateur de widening est adaptée ici à notre contexte. Pour tout ensemble V de variables, on définit $\mathcal{L}_{/V}$ comme suit :

$$\mathcal{L}_{/V} = \{ c \in \mathcal{L} : Var_{free}(c) = V \}$$

$\mathcal{L}_{/V}$ correspond donc à l'ensemble des contraintes dont les variables libres appartiennent à un ensemble V de variables. Pendant la résolution, nous aurons typiquement à gérer des suites de contraintes dont les variables libres appartiennent au même ensemble V (par exemple, les contraintes d'une liste de solutions associée à un noeud actif).

Définition 6.18

Pour tout ensemble V de variables, un opérateur ∇ défini de $\wp(\mathcal{L}_{/V}) \times \mathcal{L}_{/V}$ sur $\wp(\mathcal{L}_{/V})$ est un opérateur de widening ssi :

(1) $\forall S \in \wp(\mathcal{L}_{/V}), \forall c \in \mathcal{L}_{/V}, S \vdash S \nabla c$ et $\{c\} \vdash S \nabla c$

(2) pour toute chaîne $c_0 \ldots c_i \ldots$ la chaîne croissante définie par

$S_0 = c_0 \ldots S_i = S_{i-1} \nabla c_i \ldots$

est stationnaire (i.e., $\exists n_0 \in \mathbb{N} : \forall n > n_0 : S_{n_0} = S_{n_0} \nabla c_n$).

La description d'un opérateur de widening ∇_c (opérateur lié à un critère c) est alors donnée moyennant l'introduction de deux applications :

- l'application c-pro qui est définie de $\wp(\mathcal{L}_{/V}) \times \mathcal{L}_{/V}$ sur IB
- l'application c-gen qui est définie de $\wp(\mathcal{L}_{/V}) \times \mathcal{L}$ sur $\mathcal{L}_{/V}$

c-pro est une fonction booléenne qui teste si une propriété liée aux arguments de la fonction est vérifiée et c-gen est une fonction de généralisation qui réalise une abstraction. Intuitivement le calcul de $S \nabla_c c$ correspond à l'une des opérations suivantes :

- stagnation
- généralisation
- insertion

Voici la description de ∇_c : $\forall S \in \wp(\mathcal{L}_{/V}), \forall c \in \mathcal{L}_{/V}$,

Si $\exists c' \in S$ tel que $c \vdash c'$ alors $S \nabla_c c = S$	(stagnation)
Sinon	
Si c-pro(S,c) est vrai alors $S \nabla_c c$ = c-gen(S,c)	(généralisation)
Sinon $S \nabla_c c = S \cup \{c\}$	(insertion)
Finsi	
Finsi	

Définition 6.19 : Opérateur de widening ∇_c

Définir un opérateur de widening ∇_c revient donc simplement à définir deux fonctions c-pro et c-gen et à montrer que les conditions (1), et (2) sont vérifiées. En fait, la condition (1) est automatiquement vérifiée dans le cas d'une stagnation ou d'une insertion. D'après la description de ∇_c donnée ci-dessus :

- si $S \nabla_c c = S$ alors $S \vdash S \nabla_c c$ et $\exists c' \in S$ tel que $c \vdash c'$ (stagnation)
- si $S \nabla_c c = S \cup \{c\}$ alors $S \vdash S \nabla_c c$ et $c \vdash c$ (insertion)

Nous donnons maintenant une description générale de trois opérateurs de widening notés respectivement ∇_{depth}, ∇_{prox} et ∇_{card}. La description de ces opérateurs est générale au sens où elle peut être adaptée à différents CLP-langages. En particulier, nous l'adaptons dans cette thèse à CLP($\mathcal{FT}$), CLP($\mathcal{SC}$) et CLP($\mathcal{FT}$+$\mathcal{SC}$). Tous les opérateurs sont paramétrés par un entier k mais il est bien sur possible de généraliser à un nombre arbitraire de paramètres.

Opérateur ∇_{depth}

Nous considérons pour l'opérateur ∇_{depth}, une fonction depth définie de $\mathcal{L}_{/V}$ vers IN et une fonction d'abstraction unaire abs1 définie de $\mathcal{L}_{/V}$ vers $\mathcal{L}_{/V}$. Les fonctions depth-pro et depth-gen sont définies comme suit :

- depth-pro(S,c) ssi depth(c) $\geq$ k.
- depth-gen(S,c) = S $\cup$ abs1(c).

∇_{depth} est bien défini ssi les conditions (1) et (2) sont vérifiées. ∇_{depth} gére l'abstraction sur la "profondeur" des contraintes. Cela signifie que lors du calcul de $S \nabla_{depth} c$, si c atteint une certaine "profondeur" (i.e., si c est tel que depth(c) $\geq$ k) alors une généralisation est effectuée. Il est à noter que les fonctions depth-pro et depth-gen ne tiennent pas réellement compte de l'argument S. Ceci constitue une réelle faiblesse de l'opérateur. En fait, ∇_{depth} est un pseudo opérateur de widening puisque le domaine de calcul est fini et fixé dés le départ (et donc indépendant du contexte).

Opérateur ∇_{prox}

Nous considérons pour l'opérateur ∇_{prox}, une fonction prox définie de $\mathcal{L}_{/V} \times \mathcal{L}_{/V}$ vers IN et une fonction d'abstraction binaire abs2 définie de $\mathcal{L}_{/V} \times \mathcal{L}_{/V}$ vers $\mathcal{L}_{/V}$. Les fonctions prox-pro et prox-gen sont alors définis comme suit :

- prox-pro(S,c) ssi il existe un élément c' de S tel que prox(c,c') $\geq$ k.
- prox-gen(S,c) = S $\cup$ {abs2(c,c')} où c' désigne une contrainte de S telle que prox(c,c') $\geq$ k.

∇_{prox} est bien défini ssi les conditions (1) et (2) sont vérifiées. ∇_{prox} est un opérateur de widening gérant l'abstraction sur la "proximité" des contraintes.

Cela signifie que lors du calcul de S ∇_{prox} c, si S possède un élément c' proche de c (i.e., tel que prox(c,c') $\geq$ k) alors une généralisation est effectuée. Par rapport à ∇_{depth}, ∇_{prox} tient compte du contexte (c'est à dire de S).

Opérateur ∇_{card}

Nous considérons pour l'opérateur ∇_{card}, la fonction, notée card, de cardinalité classique (donc définie en particulier de $\mathcal{P}(\mathcal{L}/\mathcal{V})$ vers $\mathbb{N}$) et une fonction d'abstraction binaire abs2 définie de $\mathcal{L}/\mathcal{V}\times\mathcal{L}/\mathcal{V}$ vers $\mathcal{L}/\mathcal{V}$. Les fonctions card-pro et card-gen sont alors définis comme suit :

- card-pro(S,c) ssi card(S) +1 $\geq$ k.
- card-gen(S,c) = S $\cup$ {abs2(c,c')} - {c'} où c' désigne un élément de S

∇_{card} est bien défini ssi les conditions (1) et (2) sont vérifiées. ∇_{card} est un opérateur de widening gérant l'abstraction sur la cardinalité de l'ensemble S. Cela signifie que lors du calcul de S ∇_{card} c, si S posède déja k éléments alors un élément c' de S est sélectionné et remplacé par abs2(c,c').

Schéma de preuve

Pour prouver qu'un opérateur ∇_c est bien défini, nous utiliserons toujours le même schéma de preuve. Il est basé sur trois affirmations :

(a) le nombre d'insertions possibles est fini,
i.e., il n'est pas possible de trouver une chaîne infinie c_0 ... c_i ... d'éléments de $\mathcal{L}/\mathcal{V}$ et une suite infinie d'indices croissants id_0, ..., id_j, ... telle que toute étape id_j du calcul de la chaîne croissante définie par $S_0 = c_0$ ... $S_i = S_{i-1}$ ∇_c c_i ... soit une étape d'insertion (c-pro($S_{id_{j-1}}$,c_{id_j}) étant donc évalué à faux).

(b) la généralisation est correcte,
i.e., $\forall$ S $\in$ $\wp(\mathcal{L}/\mathcal{V})$, $\forall$ c $\in$ $\mathcal{L}/\mathcal{V}$, S $\vdash$ c-gen(S,c) et {c} $\vdash$ c-gen(S,c).

(c) le nombre de généralisations successives est fini,
i.e., il n'est pas possible de trouver une chaîne infinie c_0 ... c_i ... d'éléments de $\mathcal{L}/\mathcal{V}$ et un entier j positif tel que toute étape k > j du calcul de la chaîne croissante définie par $S_0 = c_0$ ... $S_i = S_{i-1}$ ∇_c c_i ... soit une étape de généralisation (c-pro(S_{k-1},c_k) étant donc évaluée à vrai).

Le (b) signifie que la condition (1) est vérifiée. Pour cela, il suffit de montrer que la fonction d'abstraction utilisée est extensive :

pour ∇_{depth} : $\forall$ c $\in$ $\mathcal{L}$, $c \vdash$ abs1(c)
pour ∇_{prox} et ∇_{card} : $\forall$ (c_1,c_2) $\in$ $\mathcal{L}\times\mathcal{L}$, $c_1 \vee c_2 \vdash$ abs2(c_1,c_2)

Le (a) et le (c) signifient que la condition (2) est vérifiée. En effet, imaginons qu'elle ne le soit pas, cela signifie qu'il existe une chaîne croissante $S_0 = c_0$ ... $S_i = S_{i-1} \nabla_c c_i$ ... non stationnaire. Quelque soit l'entier j, on sait qu'il existe un entier $k > j$ tel que $S_{k-1} \nabla_c c_k = S_{k-1} \cup \{c_k\}$ car la chaîne est non stationnaire et (c). Or, cela est en contradiction avec (a).

Nous proposons ci-dessous une description complète de ∇_{depth}, ∇_{prox} et ∇_{card} par rapport à CLP($\mathcal{FT}$). On suppose fixés pour ce qui suit un entier k et un ensemble fini de variables V.

Opérateur ∇_{depth} pour CLP($\mathcal{FT}$)

On commence par définir les fonctions depth et abs1 par rapport à CLP($\mathcal{FT}$). Ces définitions sont classiques.

Soit t un terme,

$depth(t) = 1$ si $t \in \mathcal{V}$ ou $t \in \mathcal{F}_0$
$depth(t) = 1 + \max \{depth(t_i) : 1 \leq i \leq n\}$ si $t = f(t_1,...,t_n)$.

Soit $\mathcal{S}$ un système résolu,

$depth(\mathcal{S}) = \max \{depth(\mathcal{S}(X)) : X \in \mathcal{V}\}$

Soit t un terme, $abs1(t)$ désigne le terme t dont tout sous-terme à une profondeur égale à k est remplacé par une nouvelle variable. Soit $\mathcal{S}$ un système résolu, $abs1(\mathcal{S})$ représente le système résolu obtenu à partir de $\mathcal{S}$ en remplaçant pour toute variable X de $Var_{free}(\mathcal{S})$, le terme $\mathcal{S}(X)$ par $abs1(\mathcal{S}(X))$. Les nouvelles variables sont quantifiées existentiellement.

Il est à noter que l'opération abs1 a été proposée par [Tamaki et Sato 84] sous le nom de "term-depth abstraction". Par exemple, si k = 3 et si

$\mathcal{S}$ = { X=p(f(X'),g(f(X"))) , Y=b }

alors on a :

$depth(\mathcal{S}) = 4$
$abs1(\mathcal{S})$ = { X=p(f(U),g(V),a) , Y=b }

Pour montrer que ∇_{depth} est un opérateur de widening (bien défini), nous introduisons trois lemmes qui vont nous permettre de vérifier les affirmations (a), (b) et (c) du schéma de preuve.

Lemme 6.20

L'ensemble des systèmes résolus $\mathcal{S}$ tels que $Var_{free}(\mathcal{S}) \subseteq V$ et $depth(\mathcal{S}) \leq k$, est fini.

Preuve

On sait que l'ensemble des termes (au nom des variables près) de profondeur inférieure ou égale à k est fini. A chaque variable X de V, on ne peut donc associer qu'un nombre fini de termes t tel que $depth(t) \leq k$. Comme V est un ensemble fini, on en déduit la propriété 6.20 ❑

Lemme 6.21

Pour tout système résolu $\mathcal{S}$, $\mathcal{S} \leq abs1(\mathcal{S})$

Preuve

On sait que $\mathcal{S}$ et $abs1(\mathcal{S})$ correspondent à des substitutions idempotentes. Clairement il existe une substitution σ telle que $\mathcal{S} = \sigma(abs1(\mathcal{S}))$. ❑

Lemme 6.22

L'ensemble $\{abs1(\mathcal{S}) : \mathcal{S}$ est un système résolu tel que $Var_{free}(\mathcal{S}) \subseteq V\}$ est fini.

Preuve

Par définition, $abs1(\mathcal{S})$ est un système résolu. Par construction, on sait que pour tout système résolu $\mathcal{S}$, $depth(abs1(\mathcal{S})) \leq k$ et $Var_{free}(abs1(\mathcal{S})) \subseteq V$ car les variables introduites lors d'éventuelles transformations sont quantifiées existentiellement. On en déduit que l'ensemble $\{abs1(\mathcal{S}) : \mathcal{S}$ est un système résolu tel que $Var_{free}(\mathcal{S}) \subseteq V\}$ est inclus dans l'ensemble $\{\mathcal{S} : \mathcal{S}$ est un système résolu tel que $depth(\mathcal{S}) \leq k$ et $Var_{free}(\mathcal{S}) \subseteq V\}$. Le lemme 6.22 est donc une conséquence du lemme 6.20. ❑

Propriété 6.23

∇_{depth} est un opérateur de widening.

Preuve

Les lemmes 6.20, 6.21 et 6.22 permettent de vérifier les affirmations (a), (b) et (c) du schéma de preuve. Il suffit de se rappeler pour prouver (a) et (c) qu'il est impossible d'introduire deux fois le même élément puisque sinon l'étape de stagnation est considérée (voir la définition 6.19) ❑

Opérateur ∇_{prox} pour CLP($\mathcal{FT}$)

On commence par définir les fonctions prox et abs2 par rapport à CLP($\mathcal{FT}$).

Soient $\mathcal{S}$ et $\mathcal{S}$ deux systèmes résolus,

$prox(\mathcal{S},\mathcal{S}) = depth(lub(\mathcal{S},\mathcal{S}))$
$abs2(\mathcal{S},\mathcal{S}) = lub(\mathcal{S},\mathcal{S})$

Pour montrer que ∇_{prox} est un opérateur de widening (bien défini), nous introduisons, comme précédemment, trois lemmes qui vont nous permettre de vérifier les affirmations (a), (b) et (c) du schéma de preuve.

Lemme 6.24

Il n'existe pas d'ensemble infini E de systèmes résolustel que :

- si $\mathcal{S}$ appartient à E alors $Var_{free}(\mathcal{S}) \subseteq V$
- si $\mathcal{S}$ et $\mathcal{S}$ sont deux systèmes distincts de E alors $prox(\mathcal{S},\mathcal{S}) < k$

Preuve

Imaginons un tel ensemble E et considérons E' l'ensemble E privé de tous les systèmes $\mathcal{S}$ tels que $depth(\mathcal{S}) < k$. Par la propriété 6.20, on sait que E' est encore un ensemble infini. Soit maintenant l'ensemble $F = \{abs1(\mathcal{S}) : \mathcal{S} \in E'\}$. Comme E' est infini et que F est fini (proposition 6.22), il existe nécessairement deux systèmes $\mathcal{S}$ et $\mathcal{S}$ de E' tels que $abs1(\mathcal{S}) = abs1(\mathcal{S})$. On sait alors que $lub(\mathcal{S},\mathcal{S}) \leq abs1(\mathcal{S})$ et que $depth(lub(\mathcal{S},\mathcal{S})) \geq depth(abs1(\mathcal{S}))$ car on montre facilement que : $\mathcal{S}_1 \leq \mathcal{S}_2 \Rightarrow depth(\mathcal{S}_1) \geq depth(\mathcal{S}_2)$. Comme $prox(\mathcal{S},\mathcal{S}) = depth(lub(\mathcal{S},\mathcal{S}))$ et $depth(abs1(\mathcal{S})) = k$, on en déduit $prox(\mathcal{S},\mathcal{S}) \geq k$, ce qui est une contradiction. ❑

Lemme 6.25

Pour tout couple $(\mathcal{S},\mathcal{S})$ de systèmes résolus,
$\mathcal{S} \vdash abs2(\mathcal{S},\mathcal{S})$ et $\mathcal{S} \vdash abs2(\mathcal{S},\mathcal{S})$

Preuve

Immédiat puisque $abs2(\mathcal{S},\mathcal{S}) = lub(\mathcal{S},\mathcal{S})$ ❑

Lemme 6.26

Pour tout système résolu $\mathcal{S}$ tel que $Var_{free}(\mathcal{S}) \subseteq V$, l'ensemble des majorants $\mathcal{S}$ de $\mathcal{S}$ tels que $Var_{free}(\mathcal{S}') \subseteq V$, est fini.

Preuve

On sait que l'ensemble des majorants (au nom des variables près) pour un terme donné est fini. A chaque variable X de V, on ne peut donc associer qu'un nombre fini de termes t tel que t soit un majorant de $\mathcal{S}(X)$. Comme V est un ensemble fini, on déduit le résultat de la propriété 6.26 ❑

Propriété 6.27

∇_{prox} est un opérateur de widening.

Preuve

Les lemmes 6.24, 6.25 et 6.26 permettent de vérifier les affirmations (a), (b) et (c) du schéma de preuve. ❑

Opérateur ∇_{card} pour CLP($\mathcal{FT}$)

Pour l'opérateur ∇_{card}, on considère la fonction de cardinalité classique, notée card, et la fonction d'abstraction abs2 défini à la section précédente.

Propriété 6.28

∇_{prox} est un opérateur de widening.

Preuve

Les lemmes 6.25 et 6.26 vérifient les affirmations (b) et (c) du schéma de preuve. L'affirmation (a) est trivialement vérifiée par la fonction card car il est possible de faire au plus k insertions. ❑

Même si ils ne sont pas très efficaces dans le cadre restreint de CLP($\mathcal{FT}$), les opérateurs que nous venons de définir vont nous permettent d'illustrer la résolution AOLDT. Dans ce contexte, cela consiste essentiellement en une inférence des appels et solutions atomiques.

4.1.2 Description de l'étape de widening

Deux noeuds v et v' sont dits compatibles ssi $appel_1(v) = (c,a)$, $appl_1(v') = (c',a')$ et $\exists\ \rho \in \mathcal{R}en$ tel que $a = \rho(a')$. Autrement dit, deux noeuds v et v' sont compatibles ssi l'atome le plus à gauche de (l'étiquette de) v et l'atome le plus à gauche de (l'étiquette de) v' partage le même symbole de prédicat. Pour obtenir la terminaison de la résolution OLDT, nous associons à chaque ensemble de noeuds actifs compatibles et à chaque liste de solutions (considérée comme un ensemble) un opérateur de widening ∇_c. Cet opérateur peut être spécifique à chaque ensemble. De plus, comme tout ensemble S généré via un opérateur de widening ∇_c est liée à un t-symbole de prédicat, il est possible de paramétrer ∇_c de façon spécifique à chaque t-symbole de prédicat. Par exemple, si add et mul sont deux t-symboles de prédicat, alors il est possible de paramètrer l'opérateur ∇_{depth} avec $k_{add} = 2$ et $k_{mul} = 3$. Notons pour finir que l'opérateur ∇_{card} ne peut être associé qu'à une liste de solutions en raison de sa définition spécifique (un élément de la liste est remplacé).

Si il existe un noeud actif v' tel que $appel_1(v) \vdash_r appel_1(v')$ alors
    -- v est enregistré comme noeud passif
    créer une entrée pour v dans PT tel que pos(v) désigne le début de list(v')
Sinon Si c-pro(S,$appel_1(v)$) alors
    -- v est enregistré comme noeud passif après abstraction
    créer un nouvel arbre dont la racine v' est telle que :
        label(v') = c-gen(S,$appel_1(v)$)
    créer une entrée pour v' dans AT telle que :
        list(v') est initialisée à la liste vide.
    créer une entrée pour v dans PT tel que
        pos(v) pointe sur le début de list(v').
Sinon-- v est enregistré comme noeud actif
    créer une entrée pour v dans AT tel que list(v) est initialisée à la liste vide
Fin si

Algorithme 6.29 : A) Classification d'un t-noeud

Les algorithmes proposés précédemment sont modifiés en conséquence. Soit $\mathcal{S}tr$ = (F,AT,PT) une structure OLDT, soit v un t-noeud dont la classification doit être effectuée. On pose S = {$\rho(c')$: $appel_1(v) = (c,a)$, $appel_1(u) = (c',a')$, $a = \rho(a')$ et u est un noeud actif de F}. S désigne la liste des (contraintes issues des) noeuds actifs compatibles de v. Trois possibilités se présentent (algorithme 6.29).

L'algorithme présentant la structure OLDT initiale $\mathcal{S}tr_0 = (F_0,AT_0,PT_0)$ reste le même (algorithme 6.7).

L'algorithme d'extension élémentaire d'une structure OLDT est modifié. Le point 1.3 de l'algorithme 6.8 est modifié (algorithme 6.30).

1.3 Pour toute sous-réfutation atomique m telle que m débute par un noeud actif u et finit par un nouveau t-noeud w
    on pose app(u) = (c,a) et d' = cstr(m)$\downarrow a$.
    Si il n'existe pas d'élément d'' dans list(u) tel que $c \wedge d' \vdash c \wedge d''$
        Si c-pro(list(u),$c \wedge d'$) alors
            ajouter c-gen(list(u),$c \wedge d'$) à la fin de list(u)
        Sinon
            ajouter d' à la fin de list(u)
        Finsi
    Fin si
Fin pour

Algorithme 6.30 : C) Extension élémentaire d'une structure OLDT

En résumé, les modifications apportées à la résolution sont les suivantes : pour l'algorithme A, une nouvelle possibilité de classification a été introduite

(classification avec abstraction) et pour l'algorithme C, l'ajout d'un élément dans une liste de solutions (le point 1.3) a été remanié. Ce nouveau calcul est appelé étape de widening de la résolution AOLDT. Tout comme la résolution OLDT, l'étape de widening de la résolution AOLDT calcule une forêt OLDT et est caractérisée par une stratégie de parcours. Chaque fois que cela n'est pas précisé, la stratégie de parcours "multi-étape et en largeur d'abord" est sous-entendue.

Forêt, arbre ou graphe OLDT ?

Soit F une forêt OLDT, considérons les arcs imaginaires (v,v') entre chaque noeud passif v de F et chaque noeud actif v' de F tel que v pointe sur v'. Ces arcs sont dits pointants. Un arc pointant (v,v') est dit remontant ssi le noeud v' est plus ancien que le noeud v (d'après l'ordre de création des noeuds), est dit descendant sinon. On sait (d'aprés l'algorithme) qu'un arc pointant descendant est toujours placé entre un noeud passif v d'un arbre de F et la racine v' (noeud actif) d'un arbre de F telle que size(v') = 1. Si on ajoute à la forêt F l'ensemble des arcs pointant descendant, on obtient un seul arbre (car tous les arbres sont reliés et aucune boucle ne peut apparaitre). Cet arbre est dit OLDT. Si on ajoute maintenant à l'arbre OLDT l'ensemble des arcs pointants remontant (ce qui revient à ajouter l'ensemble des arcs pointants à la forêt OLDT) on obtient un graphe (car certaines boucles peuvent apparaître). Ce graphe est dit OLDT. Par la suite, nous utiliserons indistinctement la forêt, l'arbre ou le graphe OLDT pour notre étude. Nous illustrons notre propos avec un exemple considéré précédemment.

Exemple 1

Soit le programme P suivant :

lt3(s(s(0))).
lt3(X) ← lt3(s(X)).

Soit le but G suivant :

← lt3(s(0)).

Considérons Prolog. Si lt3 est un t-symbole de prédicat et si l'opérateur ∇_{prox} est associé à l'ensemble des noeuds actifs et est paramétré par $k_{lt3} = 3$, alors la forêt OLDT obtenue par l'étape de widening de la résolution AOLDT de (P,G) est décrite par la figure 6.31. Pendant la résolution, lorsque la classification du noeud 3 se présente, prox-pro(S,lt3(s(s(s(0))))) retourne vrai car t = lt3(s(s(X)) est l'anti-unifié de lt3(s(s(0)) et lt3(s(s(s(0))) et depth(t) ≥ 3, aussi un nouvel arbre de racine prox-gen(S,lt3(s(s(s(0))))) =

lt3(s(s(X)) est créé. Notons que les noeuds passifs 3 et 5 pointent sur le noeud actif 4 et qu'ils ne peuvent pas exploiter l'unique solution de list(4).

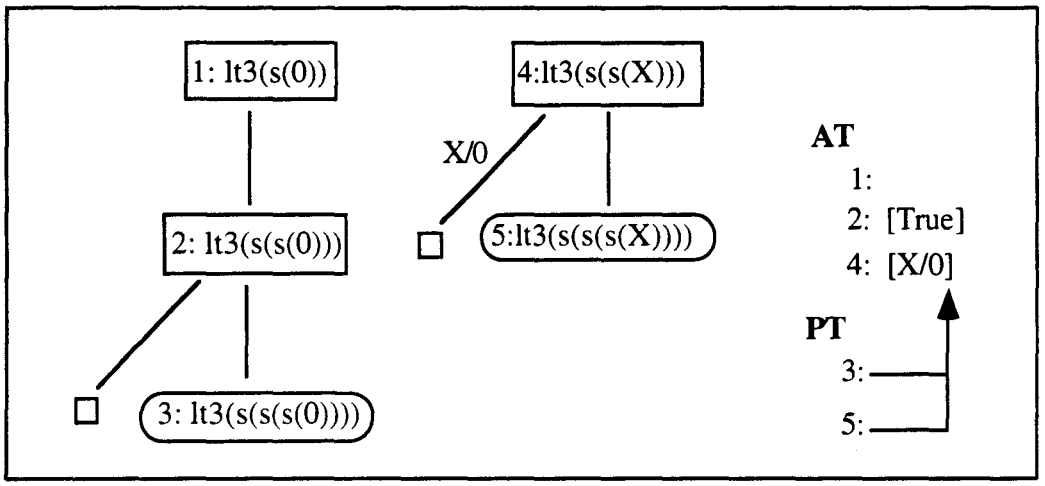

Figure 6.31

L'arbre OLDT est décrit par la figure 6.32 (l'arc (3,4) est un arc pointant descendant). Nous représentons les arcs pointants par une flèche en pointillé.

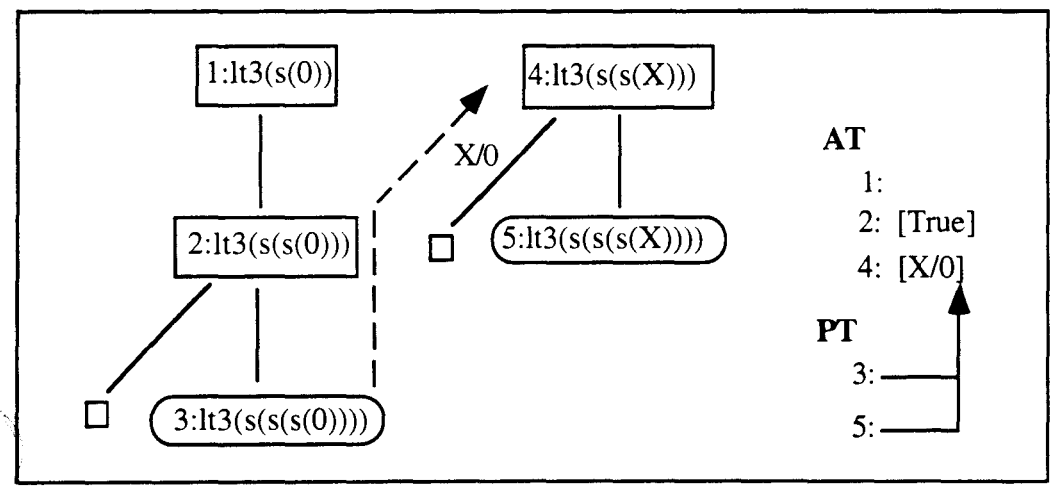

Figure 6.32

Le graphe OLDT est décrit par la figure 6.33 (l'arc (5,4) est un arc pointant remontant).

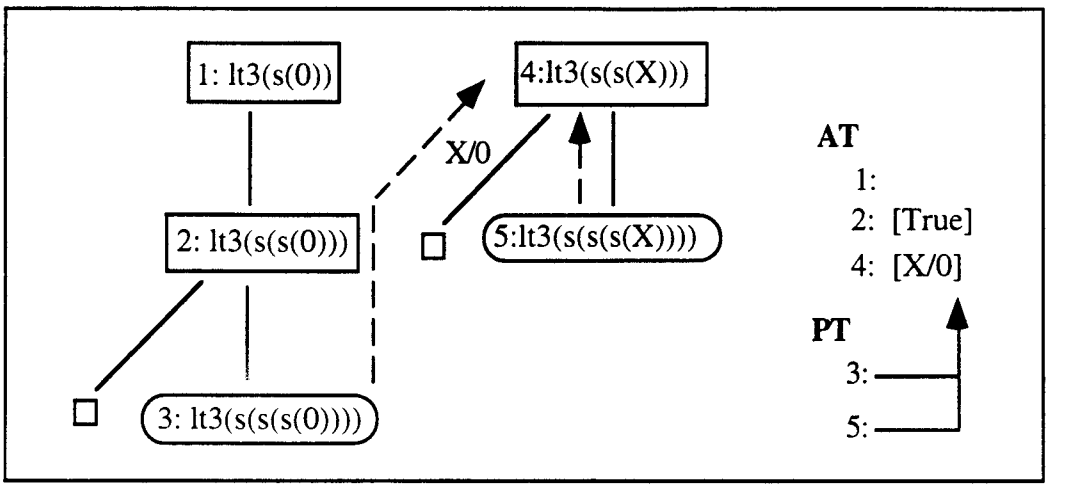

Figure 6.33

Exemple 2

Soit le programme P suivant :

app(nil,L,L).
app(cons(X,L_1),L_2,cons(X,L_3))← app(L_1,L_2,L_3).

Soit le but G suivant :

← app(L_1,L_2,L_3).

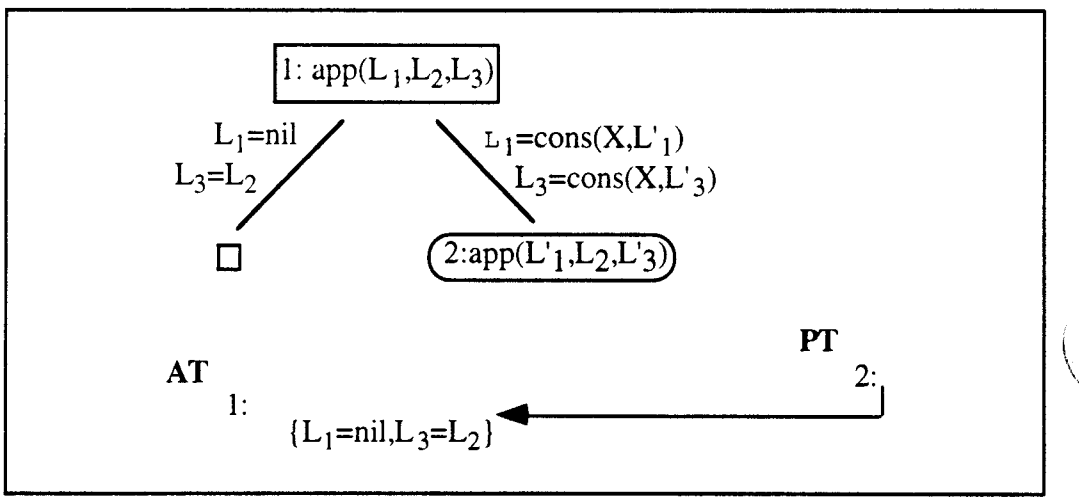

Figure 6.34

Si app est un t-symbole de prédicat et si l'opérateur ∇_{card} est associé à la liste des solutions et est paramétré par $k_{app} = 2$ alors la forêt OLDT obtenue par l'étape de widening de la résolution AOLDT de (P,G) est décrite par la figure 6.37. Dans un premier temps, on obtient la forêt OLDT décrite par la figure

6.34. Dans un second temps, la solution s_1 = {L_1=nil, L_3=L_2} est utilisée afin de générer une nouvelle solution s_2 = {L_1=cons(X,nil), L_3=cons(X,L_2)} à partir du noeud 2 (voir figure 6.21).

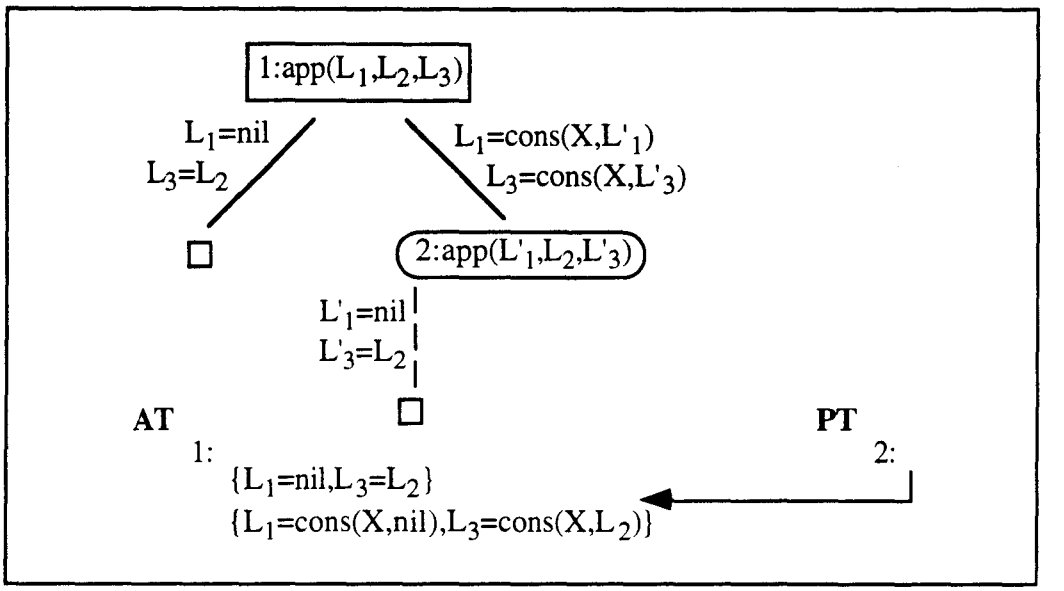

Figure 6.35

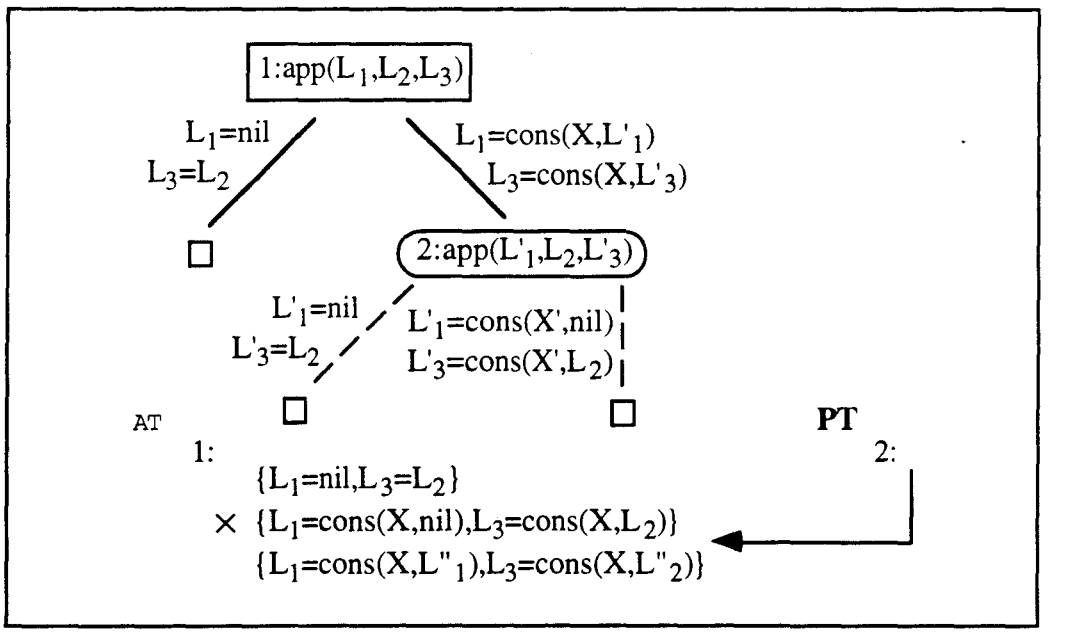

Figure 6.36

Dans un troisième temps, la solution s_2 est utilisée afin de générer une nouvelle solution s_3 = {L_1=cons(X,cons(X',nil)), L_3=cons(X,cons(X',L_2)} à partir du noeud 2. Toutefois, s_3 n'est pas intégrée dans list(1). En effet, comme card(list(1))+1 = 3 > k_{app}, on effectue une anti-unification entre s_2 et

s_3 et le résultat s_3 = {L_1=cons(X,L"$_1$), L_3=cons(X,L"$_2$)} est utilisé pour remplacer s_2 (voir figure 6.21). Sur la figure, nous avons marqué d'une croix la solution s_2 pour indiquer que celle-ci était éliminée. Il est à noter que le choix était possible entre s_1 et s_2 pour l'anti-unification. Ce choix est arbitraire, et peut être guidé par quelques heuristiques (par exemple, le choix peut s'effectuer sur une notion de ressemblance).

Finalement, la solution s_3 est utilisée fin de générer une nouvelle solution s_4 = {L_1=cons(X,cons(X',L"$_1$)), L_3=cons(X,cons(X,L"$_2$)} à partir du noeud 2. Cette dernière solution est instance de s_3, aussi la résolution se termine (voir figure 6.37).

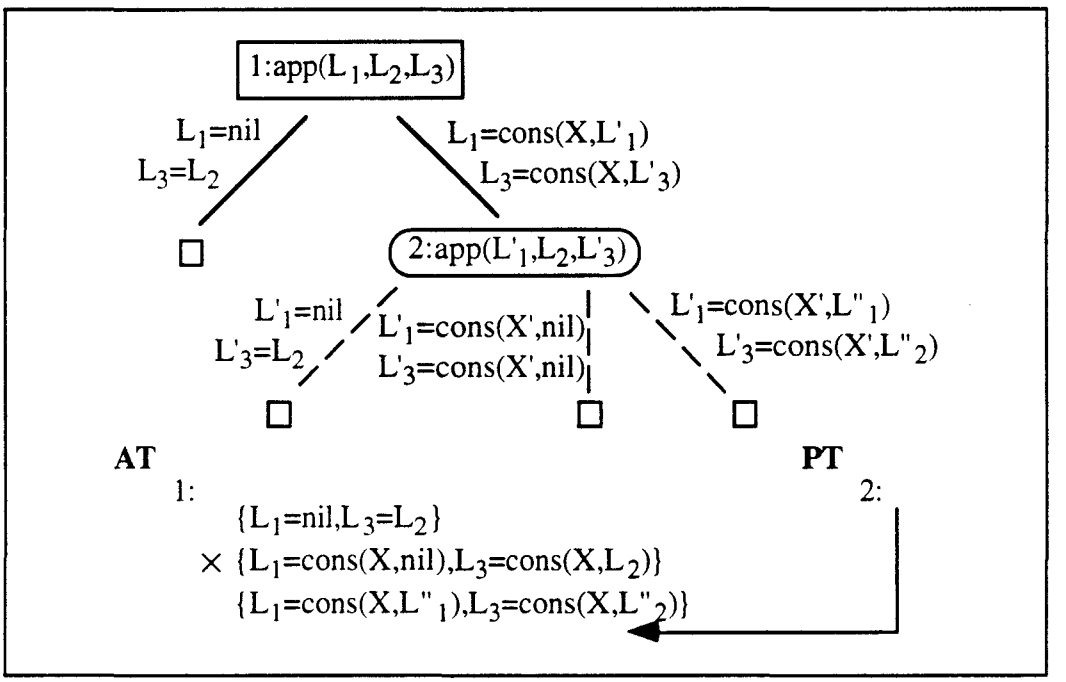

Figure 6.37

Nous obtenons dans la liste des solutions du noeud 1 une caractérisation des solutions pour le but app(L_1,L_2,L_3). Si le paramètre k_{app} avait une valeur plus importante (permettant ainsi la prise en compte de plus de solutions), la précision du résultat serait meilleure. Toutefois, il n'est pas possible d'inferrer la structure récursive des solutions avec cette méthode. Il est donc nécessaire pour améliorer l'analyse d'intégrer des propriétés (contraintes) abstraites permettant de coder une telle information. C'est ce que nous ferons au chapitre 8.

Exemple 3

Soit le programme P suivant :

ack(0,Y,s(Y)).←

ack(s(X),0,Z) ← ack(X,s(0),Z)
ack(s(X),s(Y),Z) ← ack(s(X),Y,Z'), ack(X,Z',Z)

Soit le but G suivant :

← ack(X,Y,Z)

Si ack est un t-symbole de prédicat et si l'opérateur ∇_{card} est associé à la liste des solutions et est paramétré par $k_{ack} = 1$ alors la forêt OLDT obtenue par l'étape de widening de la résolution AOLDT de (P,G) est décrite par la figure 6.38.

Cet exemple illustre la possibilité d'obtenir des graphes plus complexes. Peu d'informations est obtenue sur la forme des solutions. ❑

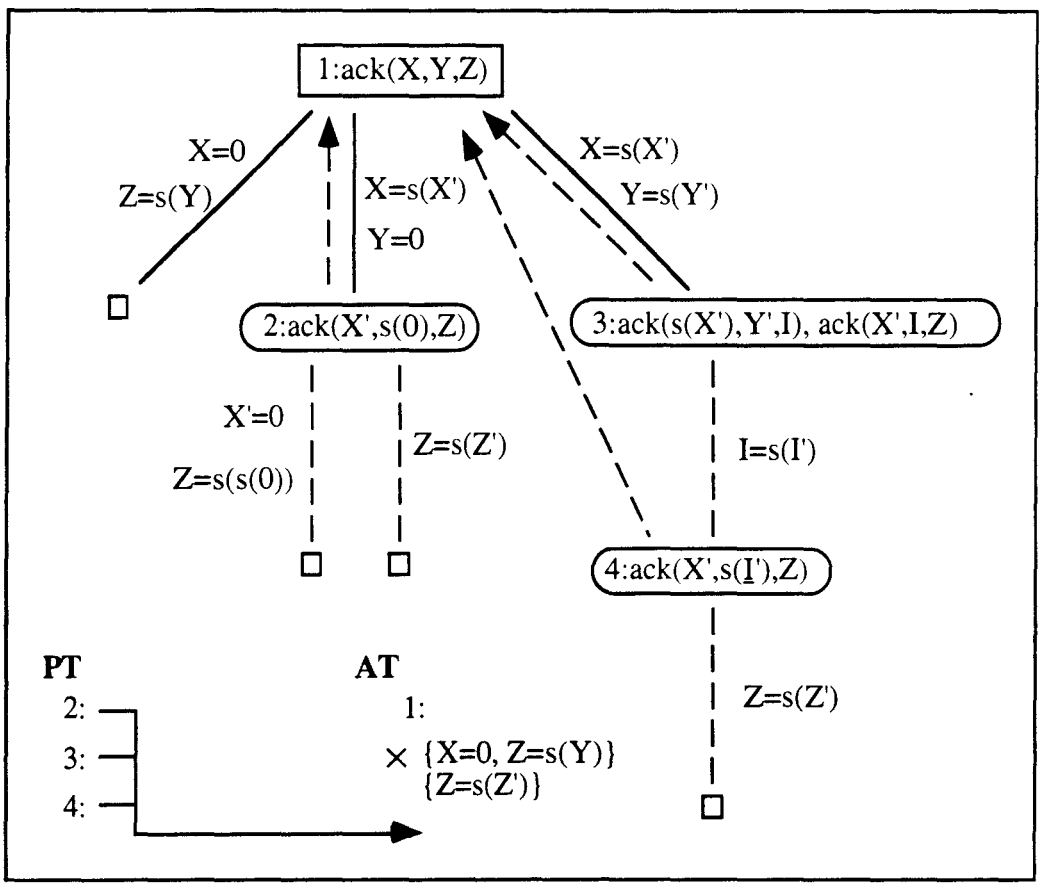

Figure 6.38

Ancêtre réel

Nous présentons maintenant la notion d'ancêtre réel. Soit T un arbre OLDT et soient v et v' deux noeuds de T, v est un ancêtre de v' ssi v est plus ancien que

v' (d'après l'ordre de création) et si il existe un chemin entre v et v'. Soit m le chemin allant de la racine i de T jusqu'à v' et soit v un noeud de m. Si il existe sur le chemin m et à partir de v un arc pointant descendant (v'',w'') alors v est un ancêtre réel de v' ssi size(v) ≤ size(v''), sinon v est un ancêtre réel de v' ssi size(v) ≤ size(v'). En fait, v est un ancêtre réel de v' ssi à partir de appel$_1$(v) il est possible d'atteindre appel$_1$(v'). Prenons un exemple simple (figure 6.38), les noeuds 1 et 3 sont des ancêtres du noeud 4 mais le noeud 3 n'est pas un ancêtre réel du noeud 4 car size(3) > size(4).

Nous modifions l'algorithme de l'étape de widening en tenant compte de cette notion d'ancêtre réel. En fait, ce qui change par rapport à l'algorithme original est simplement le fait de ne considérér pour un noeud v donné que la liste des ancêtres réels de v. On reprend l'algorithme 6.29, mais on pose S = {ρ(c') tel que appel$_1$(v) = (c,a), appel$_1$(u) = (c',a'), a = ρ(a') et u est un noeud actif de la liste des ancêtres réels de v}. Ce changement est justifié à la section 4.2 où il est montré que tout graphe OLDT obtenu par l'étape de widening (modifiée comme ci-dessus) de la résolution AOLDT est adéquat à une analyse par calcul de point fixe. Dorénavant, nous considérons (sauf si le contraire est explicitement indiqué) l'étape de widening de la résolution AOLDT avec la version modifiée de l'algorithme (ci-dessus) de classification d'un t-noeud.

Théorème 6.39

L'étape de widening de la résolution AOLDT est finie.

Preuve

Comme le nombre de symboles de prédicat pouvant apparaître dans un programme est fini, on peut se ramener pour simplifier au cas où ce nombre est égal à 1.

Aux détails près, la considération d'un nouveau noeud v lors de la résolution est équivalente à l'opération suivante : S ∇_c app(v) où S = {ρ(c') | appel$_1$(v) = (c,a), appel$_1$(u) = (c',a'), a = ρ(a') et u est un noeud actif de la liste des ancêtres réels de v}. De même, aux détails près, la considération d'un nouvel élément d pour une liste de solutions list(u) est équivalente à l'opération suivante : list(u) ∇_c d. Ceci nous permet de conclure, d'aprés la définition de ∇_c, que :

- chaque ensemble d'ancêtres réels d'un noeud est finie
- chaque liste de solutions associée à un noeud actif est finie.

En outre, chaque noeud possède un nombre fini de noeuds fils. En effet, le nombre d'étapes de dérivation liées à un noeud actif est fini (car le nombre de clauses d'un programme est fini) et le nombre de solutions pouvant être consultées par un noeud passif est fini (car chaque liste de solutions est finie).

D'une part, un nombre infini de noeuds actifs nécessite d'après le lemme de Koenig (tout arbre infini à branchement fini admet une branche infinie) une branche infinie. Supposons une telle branche infinie notée λ. Soit label(i) = $(c,a_1,\ldots,a_n)$. On sait qu'il existe un atome a_i tel que les i-1 premiers atomes de label(i) sont réfutés sur br et tel que a_i n'est pas réfuté sur λ. Il existe donc un noeud v_1 sur λ tel que level(v_1) est fini et tel que label(v_1) = $(c_1,a_i,\ldots,a_n)$. Ce noeud est actif sinon il existerait une réfutation de a_i sur λ. v_1 possède sur λ un noeud fils w_1 tel que label(w_1) = $(c_1',b_1,\ldots,b_m,a_{i+1},\ldots,a_n)$. On sait qu'il existe un atome b_j tel que les j-1 premiers atomes de label(w_1) sont réfutés sur br et tel que b_j n'est pas réfuté sur λ. Il existe donc un noeud v_2 sur br tel que level(v_2) est fini et tel que label(v_2) = $(c_2,b_j,\ldots,b_m,a_{i+1},\ldots,a_n)$. Ce noeud est actif sinon il existerait une réfutation de b_j sur λ. Clairement v_1 est un ancêtre réel de v_2. Si on continue le raisonnement précédent, on arrive à la conclusion qu'il existe un noeud v_ω tel que la liste des ancêtres réels de v_ω est infinie. Ceci est en contradiction avec a). Le nombre de noeuds actifs est donc fini.

D'autre part, un nombre infini de noeuds passifs nécessite également d'après le lemme de Koenig une branche infinie. Comme le nombre de noeuds actifs est fini, sur cette branche apparaît nécessairement une suite infinie S de noeuds passifs. Pour tout noeud passif v et tout noeud fils w de v, on sait que size(w) < size(v). Or, le nombre d'atomes de l'étiquette du premier noeud de la suite S est nécessairement fini. Ceci est une contradiction. Le nombre de noeuds passifs est donc fini.

Le nombre de noeuds actifs et de noeuds passifs est fini et chaque liste de solutions associée à un noeud actif est finie. Aussi, l'étape de widening de la résolution AOLDT est finie. ❑

Théorème 6.40

L'étape de widening de la résolution AOLDT est complète vis à vis de la résolution OLD par rapport à la sémantique du flux de données.

Preuve

La preuve établie pour la résolution OLDT reste valable. ❑

4.2 Analyse d'un graphe OLDT

Avant d'introduire l'étape de narrowing de la résolution AOLDT, il est nécessaire de fournir quelques éléments d'analyse.

Définition 6.41

Soit T un arbre OLDT et soit G le graphe OLDT associé, une boucle sur G est un chemin m = (v,v') sur T tel que v' pointe sur v et tel que v est plua ancien (d'après l'ordre de création) que v' .

Si λ = (v,v') est une boucle alors le noeud v est appelé le noeud origine de la boucle et le noeud v' est appelé le noeud terminal de la boucle.

Notation 6.42

Soit G un graphe OLDT, loops(G) désigne l'ensemble des boucles de G, et pour tout noeud v de G, loops(v) désigne l'ensemble des boucles de G dont l'origine est le noeud v.

Par exemple, le graphe OLDT de la figure 6.33 possède une seule boucle λ = (4,5). Le graphe OLDT de la figure 6.38 possède trois boucles distinctes λ_1 = (1,2), λ_2 = (1,3) et λ_3 = (1,4), et loops(1) = $\{\lambda_1,\lambda_2,\lambda_3\}$.

Une relation de préordre $\leq$ et une relation d'équivalence $\approx$ sont définies par rapport à tout graphe OLDT. Soit G un graphe OLDT et soient λ_1 et λ_2 deux boucles de G,

$\lambda_1 \leq \lambda_2$ ssi il existe un chemin menant de l'origine de λ_1 à l'origine de λ_2

$\lambda_1 \approx \lambda_2$ ssi $\lambda_1 \leq \lambda_2$ et $\lambda_2 \leq \lambda_1$

Si G est un graphe OLDT alors loops(G)/$_\approx$ désigne l'ensemble des boucles de G modulo $\approx$. Deux boucles appartiennent à la même classe si elles sont étroitement liées (il est possible de passer de l'une à l'autre par dépliage).

Un graphe OLDT possède une structure adéquate à une analyse par calcul de point fixe. En effet, chaque noeud passif pointe sur un ancêtre réel. Aussi, lorsqu'un attribut (une information) est associé à un noeud v, pour vérifier l'invariance locale (i.e., l'invariance ne tenant pas compte des noeuds ancêtres de v) de cet attribut, il suffit de considérer l'ensemble loops(v) des boucles dont l'origine est le noeud v et de montrer que l'attribut est préservé par rapport au dépliage de chacune de ces boucles. Le calcul s'effectue donc simplement et naturellement.

Soit (Å,$\leq$) un ensemble d'attributs Å muni d'une relation d'ordre partielle $\leq$. Etant donné un attribut å et un noeud v, on cherche à calculer un attribut å' qui soit plus petit ou égal à å et qui soit invariant (localement) par rapport au dépliage des boucles de l'ensemble loops(v). Ce calcul s'effectue par point fixe à l'aide d'un opérateur d'invariance noté p. Cet opérateur possède comme arguments un attribut å et un noeud v et retourne comme résultat un attribut å'. Quelque soit l'attribut å et le noeud v, un opérateur d'invariance p doit vérifier :

C_0 p(å,v) $\leq$ å

C_1 $\exists k \in \mathbb{N}$ tel que $\text{lfp } p_{(å,v)} = p^k(å,v) = p^{k+1}(å,v)$

Si $\text{loops}(v) = \emptyset$ alors le résultat de l'opération $p(å,v)$ est l'identité å, sinon il s'agit de calculer l'invariance de l'attribut å par rapport à chaque boucle de l'ensemble $\text{loops}(v)$ et de considérer un minorant global via un opérateur de minoration $\oplus$.

$p(å,v) = å$ ssi $\text{loops}(v) = \emptyset$
$p(å,v) = pp(å,b_1) \oplus ... \oplus pp(å,b_n)$ ssi $\text{loops}(v) = \{\lambda_1,...,\lambda_n\}$.

L'opérateur de minoration $\oplus$ est un opérateur défini de Å×Å sur Å et tel que :

C_2 $\forall å \in Å, \forall å' \in Å, å \oplus å' \leq å$ et $å \oplus å' \leq å'$

Le calcul de l'invariance d'un attribut par rapport à une boucle est donné par l'opérateur pp. L'opération $pp(å,\lambda)$ consiste à propager (descendre) l'attribut å de l'origine de la boucle jusqu'au noeud terminal de la boucle via un opérateur down tout en effectuant un calcul d'invariance pour chaque noeud de la boucle, puis de remonter le résultat å' vers l'origine de la boucle via un opérateur up et de considérer un minorant à å et å'.

si $\lambda=(v_1,v_p)$ alors $pp(å,\lambda) = å \oplus å'$ tel que :

$å_{2.i} = \text{down}(å,v_1,v_2)$ et $å_{2.o} = \text{lfp } p_{(å_{2.i},v_2)}$
...
$å_{p.i} = \text{down}(å_{p-1.o},v_{p-1},v_p)$ et $å_{p.o} = \text{lfp } p_{(å_{p.i},v_p)}$

$å' = \text{up}(å_{p.o},v_p,v_1)$

Les opérations $\text{down}(å,v,v')$ et $\text{up}(å,v,v')$ permettent de propager vers un noeud v' un attribut å associé à un noeud v en considérant une éventuelle transformation (mise à jour) de å. La propagation peut se faire vers le bas (down) ou vers le haut (up). Notons que la condition C_2 implique la condition C_0.

Pour résumer, la spécification d'un opérateur d'invariance p consiste à :
- décrire un ensemble d'attributs Å muni d'une relation d'ordre partielle $\leq$
- décrire un opérateur de minoration $\oplus$
- décrire les opérateurs down et up
- vérifier les conditions C_1, C_2

Deux types d'analyse peuvent être considérés :

- une analyse top-down

- initialement, un attribut å est associé à la racine i de l'arbre OLDT

- inductivement, si un attribut å est associé à un noeud v de l'arbre OLDT, on calcule å' = lfp $p_{(å,v)}$ et on propage å' vers les noeuds fils de v.

• une analyse bottom-up

- initialement, un attribut å est associé à chaque feuille de l'arbre OLDT

- inductivement, si un attribut $å_i$ est associé à chaque noeud fils w_i d'un noeud v de l'arbre OLDT, on propage chacun de ces attributs vers le noeud v, on considère un minorant å à l'ensemble de ces attributs et on calcule lfp $p_{(å,v)}$.

D'un point de vue pratique, il est possible de simplifier toute analyse en ne tenant compte que des noeuds origines des classes de boucle. En effet, si v est l'origine d'une classe de boucles et si å est un attribut associé à v, alors lors du calcul de lfp $p_{(å,v)}$, il est possible de garder la trace du calcul.

4.3 Etape de narrowing

L'étape de narrowing de la résolution AOLDT consiste à améliorer le résultat de l'étape de widening. Cette étape consiste donc à transformer légérement une forêt OLDT obtenue après une étape de widening. Deux phases de transformation sont à prendre en compte : une phase de simplification pendant laquelle les parties redondantes du graphe OLDT sont supprimées et une phase d'instanciation pendant laquelle certaines parties du graphe OLDT sont instanciées en reconsidérant l'information perdue via les généralisations

4.3.1 Phase de simplification

Cette phase consiste à couper certains arcs passifs, et donc à éliminer certains sous-arbres, de l'arbre OLDT obtenu par l'étape de widening. Ceci ne pose aucun problème de complétude car tout noeud actif appartenant à un sous-arbre supprimé ne peut être pointé de l'extérieur de ce sous-arbre puisque tout noeud passif pointe nécessairement sur un noeud ancêtre. On peut rapprocher cette technique avec celle de [Vielle 89] qui consiste à éliminer les parties redondantes d'un arbre SLD.

Définition 6.43

Soit T un arbre OLDT et v un noeud passif de T, un arc passif (v,w_i) connecté à v est dit inutile ssi il existe un autre arc passif (v,w_j) connecté à v et tel que $cstr(v,w_i) \vdash cstr(v,w_j)$.

Soit T un arbre OLDT. L'algorithme de simplification de T est extrémement simple, puisqu'il s'agit d'éliminer un à un les arcs inutiles. simplifier(T) désigne l'arbre OLDT obtenu aprés la transformation suivante :

```
Tant qu'il existe un arc passif inutile dans T
        éliminer de T l'un de ces arcs inutiles
Fin tant que
```

Algorithme 6.44 de simplification

Pour l'arbre OLDT de la figure 6.32, la phase de simplification n'est d'aucune incidence puisqu'il n'existe aucun arc passif dans cet arbre. Par contre, l'arbre OLDT de la figure 6.37 présente deux arcs passifs. L'un est inutile car :

$\{L'_1=cons(X',nil)\ ,\ L'_3=cons(X',nil)\}$
$\leq$
$\{L'_1=cons(X',L''_1)\ ,\ L'_3=\ cons(X',L''_3)\}$.

Aussi, est-il possible de le supprimer. La figure 6.45 décrit la forêt OLDT obtenue après cette phase de simplification.

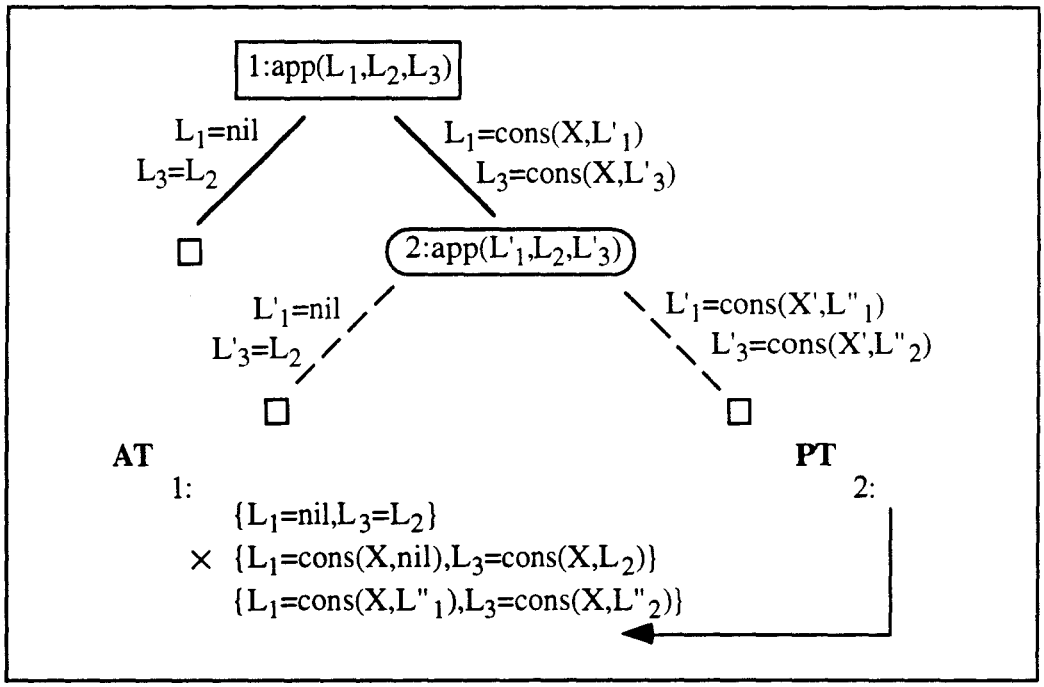

Figure 6.45

4.3.2 Phase d'instanciation

La phase d'instanciation consiste à prendre en compte l'information perdue due aux généralisations (si la forêt OLDT considérée n'est constituée que d'un seul

arbre alors la phase d'instanciation n'apporte rien). L'information retrouvée par rapport à un noeud v est représentée par un attribut (une contrainte) c qui est associé à v. Toutefois, un calcul d'invariance doit être réalisé à partir de c et de v, calcul dont le résultat est un attribut c' qui peut être appliqué à v sans problème de consistance dans le graphe. Nous donnons ci-dessous une spécification partielle d'un opérateur d'invariance p défini par rapport à n'importe quel CLP-langage $C\mathcal{L} = (\Sigma, \mathcal{L}, \mathcal{A})$.

- (Å,≤) désigne $(\mathcal{L},-)$
- down est défini par :

  down$(d,v,v') = d$
  si (v,v') est un arc actif ou passif
  down$(d,v,v') = c \wedge d$
  si (v,v') est un arc pointant descendant et label$(v) = (c,a_1,\ldots,a_n)$

- up est défini par :

  up$(d,v,v') = \rho(c \wedge d)$ tel que
  app$(v) = (c,a)$, app$(v') = (c',a')$
  $\rho(a) = a'$, $\rho(a') = a$ et $\rho(X) = X$ si $X \notin$ Var$(a) \cup$ Var(a') .

L'opérateur down propage l'information courante, une contrainte d, en tenant compte des contraintes étiquettant les arcs actifs et passifs et en ne prétant pas attention aux généralisations via les arcs pointants descendants. L'opérateur up consiste à remonter l'information invariante calculée tout en renommant certaines variables de manière à pouvoir comparer cette information avec l'information originale.

Cette spécification de p est partielle car nous ne donnons pas la description de l'opérateur $\oplus$ (et donc ne pouvons pas montrer les conditions C_1, C_2). Supposons toutefois qu'elle soit donnée (et que les conditions C_1, et C_2 soient vérifiées) par rapport à un CLP-langage fixé.

Soient T un arbre OLDT et i la racine de T. L'algorithme d'instanciation de T est le suivant : instancier(T) désigne l'arbre OLDT obtenu aprés l'appel de inst$(\top,i)$ sachant que inst(d,v) est défini comme suit :

soit e = lfp $p_{(d,v)}$
remplacer label(v) = ($c,a_1,\ldots,a_n$) par label(v) = ($c \wedge e,a_1,\ldots,a_n$)
Pour tout arc (v,w) actif ou passif de T faire
    Si $c \wedge e \wedge$ cstr(v,w) est satisfiable alors inst(e,w)
    Sinon couper l'arc (v,w)
    Fin si
Fin pour
Si v est un noeud passif (pointant sur un noeud actif v') alors
    Si (v,v') est un arc pointant descendant alors inst($c \wedge e,v'$)
fin si

Algorithme 6.46 d'instanciation

Une contrainte d est associée à v. On calcule une contrainte $e \vdash d$ qui soit invariante par rapport au dépliage des boucles dont v est l'origine. On peut alors sans problème prendre en compte la contrainte e au niveau de l'étiquette de v. Ensuite, on propage cette information vers chaque noeud fils w de v. Au cas où le nouveau test de satisfiabilité échoue, l'arc (v,w) est coupé (et donc le sous-arbre rattaché par cet arc est éliminé). L'algorithme d'instanciation est un algorithme top-down.

Pour CLP($\mathcal{FT}$), on définit l'opérateur $\oplus$ par :

$\forall\ (c,c') \in \mathcal{L}^2$, $c \oplus c'$ désigne glb(c,c').

La condition C_1 est vérifiée car pour CLP($\mathcal{FT}$), $-$ est un ordre bien fondé pour $\mathcal{L}$. La condition C_2 est immédiate.

Tentons d'instancier l'arbre OLDT de la figure 6.32. L'appel initial est : inst($\top$,1). Comme lfp $p_{(\top,1)} = \top$, on passe alors (sans modifier l'arbre) à inst($\top$,2) puis inst($\top$,3). Le noeud 3 est un arc passif (pointant sur le noeud actif 4) et l'arc (3,4) est un arc pointant descendant. L'appel suivant est donc : inst({X=s(0)},4). On sait que loops(4) = {(4,5)}.

1 p({X=s(0)},4) = pp({X=s(0)},(4,5))

pp({X=s(0)},(4,5)) = {X=s(0)} $\oplus$ {X=s(s(0))} = {X=s(X')} car

{X=s(0)} = down({X=s(0)},4,5)
{X=s(0)} = lfp $p_{(\{X=s(0)\},5)}$
{X=s(s(0))} = up({X=s(0)},5,4)

2 p({X=s(X')},4) = pp({X=s(X')},(4,5))

p'({X=s(X')},(4,5)) = {X=s(X')} $\oplus$ {X=s(s(X'))} = {X=s(X')} car

$\{X=s(X')\}$ = down($\{X=s(X')\}$,4,5)
$\{X=s(X')\}$ = lfp $p_{(\{X=s(X')\},5)}$
$\{X=s(s(X'))\}$ = up($\{X=s(X')\}$,5,4)

Ainsi, lfp $p_{(\{X=s(0)\},4)}$ = $\{X=s(X')\}$

On transforme l'étiquette lt3(s(s(X))) du noeud 4 en lt3(s(s(s(X')))). Ensuite, comme $X=0 \wedge X=s(X')$ est insatisfiable, on coupe l'arc étiqueté par la contrainte X=0. Finalement, un dernier appel inst($\{X=s(X')\}$,5) est généré. Comme loops(5) = Ø, lfp $p_{(\{X=s(X')\},5)}$ = $\{X=s(X')\}$. On transforme donc l'étiquette lt3(s(s(s(X)))) du noeud 5 en lt3(s(s(s(s(X'))))). La forêt OLDT obtenue après la phase d'instanciation est décrite par la figure 6.47.

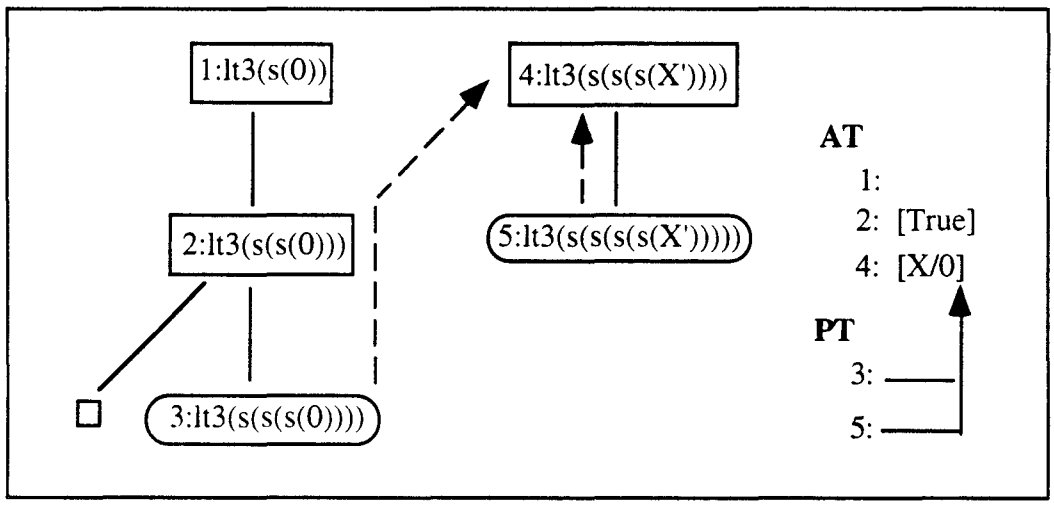

Figure 6.47

4.3.3 Description de l'étape de narrowing

L'étape de narrowing est une séquence de phases de simplification et de phases d'instanciation. Soit T un arbre OLDT, l'étape de narrowing

```
fini := faux
Tant que fini = faux faire
        T' := simplifier(T)
        T'' := instancier(T')
        Si T'' = T alors fini := vrai
        Sinon T := T''
        Fin si
Fin tant que
```

Algorithme 6.48 : étape de narrowing

Théorème 6.49

L'étape de narrowing de la résolution AOLDT est finie.

Preuve

L'algorithme de simplification est fini car le nombre d'arcs passifs dans un arbre OLDT est fini. L'algorithme d'instanciation est fini car par définition le calcul d'un plus petit point fixe est fini et le nombre de noeuds dans un arbre OLDT est fini. Le nombre de phases de simplification consécutives est fini car le nombre d'arcs passifs dans un arbre OLDT est fini. Finalement, comme l'algorithme d'instanciation est idempotent, i.e., instancier(instancier(T)) = instancier(T), on en déduit que l'étape de narrowing de la résolution AOLDT est finie. ❑

Théorème 6.50

L'étape de narrowing de la résolution AOLDT est complète vis à vis de la résolution OLD par rappport à la sémantique du flux de données.

Preuve

Pour l'étape de simplification, il est clair que les transformations préservent la complétude. Pour l'étape d'instanciation, on montre par récurrence que le graphe reste consistant. ❑

5 Discussion

[Kanamori et Kawamura 87,90] ont été les premiers, à notre connaissance, à utiliser la résolution OLDT pour concevoir un interpréteur abstrait. En considérant un domaine abstrait fini, la terminaison du calcul est assurée. [Gallagher et al. 88] reprennent la même idée et l'étendent à FCP ("Flat Concurrent Prolog"). [Filè et Sottero 91 , Codognet et Filè 92] ont également cette démarche mais dans un cadre englobant la programmation logique avec contraintes. Par ailleurs, la résolution OLDT est bien adaptée aux techniques de transformations de programmes. [Gallagher et al. 88], [Warren 92], [Boulanger et Bruynooghe 92] et [Parrain 94] utilisent pour cela l'information qui est diponible dans les tables. A noter que [Boulanger et Bruynooghe 92] introduisent une notion étendue de la résolution OLDT au sens où les entrées des tables ne sont pas des atomes mais des conjonctions d'atomes. Une étude approfondie de l'implémentation de la résolution OLDT a été réalisée par [Van Hentenryck et al. 93]. Les auteurs montrent d'abord qu'un cadre sémantique adapté à la résolution OLDT peut être obtenu à partir d'une sémantique générique de point fixe [Le Charlier et Van Hentenryck 93]. Il est montré ensuite que, d'une part, éliminer les éléments redondants (ou plus spécifiques) des listes de solution lors de la résolution a une incidence significative sur l'efficacité et que, d'autre part, une interprétation abstraite basée sur la résolution OLDT permet

d'obtenir des résultats plus précis qu'une interprétation abstraite travaillant avec une granularité plus faible [Van Hentenryck et al. 93b]. En pratique, toutefois, cette différence est moins flagrante.

Partie IV

Une Inférence de Types

Pour illustrer la puissance du modèle d'interprétation abstraite générique proposé à la partie III, nous proposons une inférence de types basée sur l'utilisation de contraintes ensemblistes. Au chapitre 7, nous étudions les relations entre les types et les contraintes ensemblistes. Au chapitre 8, nous proposons d'intégrer à CLP($\mathcal{FT}$) une forme limitée des contraintes ensemblistes de manière à pouvoir générer les types sous cette forme avec la résolution AOLDT.

Chapitre 7

Types et Contraintes Ensemblistes

Dans ce chapitre, nous étudions les relations entre les types et les contraintes ensemblistes. Dans la première section, nous introduisons d'abord les contraintes ensemblistes en signalant les opérations ensemblistes qui sont utiles à l'analyse de programmes (section 1.1), puis nous présentons les différentes classes de contraintes ensemblistes qui ont été étudiées (section 1.2). Deux catégories de types sont abordées dans la seconde section : les types syntaxiques (section 2.2) et les types sémantiques (section 2.3). Les premiers représentent une restriction sur les appels et les seconds représentent une restriction sur les solutions d'un programme logique. Nous nous intéresserons particulièrement aux systèmes de types sémantiques en présentant (par l'exemple) les différentes approches, à savoir, l'approche opérationnelle, l'approche de point fixe et l'approche dénotationnelle. Pour finir (section 3), nous envisagerons les possibilités d'une adéquation entre contraintes ensemblistes et approche opérationnelle.

1 Contraintes ensemblistes

1.1 Définition

Les contraintes ensemblistes permettent de décrire très naturellement des relations entre des ensembles de termes construits à partir d'une algèbre libre. Les contraintes ensemblistes sont utilisées particulièrement dans le domaine de l'analyse de programmes (et, en particulier, pour l'inférence de types), que ce soit pour les langages fonctionnels ([Reynolds 69], [Jones et Muchnick 79], [Mishra et Reddy 85], [Aiken et Murphy 91], [Heintze 92]), les langages logiques ([Mishra 84], [Zobel 87], [Bruynooghe et al. 87], [Heinze 92]) ou les

langages impératifs ([Heintze 92]). Il est important de remarquer, toutefois, que l'ensemble de ces travaux n'utilise pas toujours explicitement le formalisme des contraintes ensemblistes. C'est ainsi, par exemple, que [Mishra 84] utilise des "regular tree" ou "leaf linear equations" et que [Bruynooghe et al. 87] utilisent des "type graphs". Le seul (mais véritable) intéret d'utiliser le formalisme des contraintes ensemblistes est son caractère clarificateur et unificateur..

Soit $\mathcal{V}sc$ un ensemble de symboles de variable et soit $\mathcal{F}$ un ensemble (fini) de symboles de fonction. Les éléments de $\mathcal{V}sc$ sont appelés variables ensemblistes, et sont notés $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, ...

Définition 7.1

Une expression ensembliste désigne l'une des expressions suivantes :

$\mathcal{X} \mid f(ex_1,...,ex_n) \mid f^{-1}_{(i)}(ex) \mid \bot_{sc} \mid \top_{sc} \mid ex \cup ex' \mid ex \cap ex' \mid \overline{ex}$

où $\mathcal{X} \in \mathcal{V}sc$, $a \in \mathcal{F}_0$, $f \in \mathcal{F}_n$, $1 \leq i \leq n$ et $\bot_{sc}$ et $\top_{sc}$ désignent des symboles spéciaux et ex, ex', ex_i désignent des expressions ensemblistes.

Une expression ensembliste est donc soit une variable ensembliste, un terme (y compris une constante), la projection d'une expression ensembliste, la plus petite expression, la plus grande expression, l'union de deux expressions ensemblistes, l'intersection de deux expressions ensemblistes, le complémentaire d'une expression ensembliste. Notons que si $f \in \mathcal{F}_n$ et que ex est une expression ensembliste, alors il y a n projections possibles applicables à ex par rapport à f, à savoir : $f^{-1}_{(1)}(ex),...,f^{-1}_{(n)}(ex)$.

Définition 7.2

Une contrainte ensembliste désigne l'une des expressions suivantes :

- $ex \subseteq ex'$
- $ex \not\subseteq ex'$

où ex et ex' désignent des expressions ensemblistes.

Une contrainte ensembliste de la première forme est appelée contrainte positive et une contrainte ensembliste de la seconde forme est appelée contrainte négative ([Gilleron et al. 93], [Aiken et al. 93]). Toutefois, à moins que cela ne soit explicitement signalé, nous ne traiterons dans la suite que des contraintes ensemblistes positives.

Définition 7.3

Un système de contraintes ensemblistes désigne un ensemble de contraintes ensemblistes.

Maintenant que tous les objets syntaxiques (expressions, contraintes et systèmes de contraintes ensemblistes) sont définis, il est nécessaire d'étudier leur

interprétation. Dans un premier temps, nous considérons les objets comme étant clos (i.e., sans symboles de variable). Posons la signature $\Sigma = (S, \mathcal{G}, C)$ où

$S = \{term\_set\}$
$\mathcal{G} = \mathcal{F} \cup \{ f_{(i)}^{-1} \mid f \in \mathcal{F}_n \text{ et } 1 \leq i \leq n\} \cup \{0,1,\cup,\cap,^-\}$
$C = \{\subseteq, \not\subseteq, \bot, \top\}$

La Σ-algèbre $\mathcal{A}$ considérée dans le domaine des contraintes ensemblistes est définie comme suit :

- l'interprétation de S est donnée par $\mathcal{A}(term\_set) = \wp(\mathcal{T}erm(\mathcal{F}))$
- l'interprétation de $\mathcal{G}$ est donnée par :
  $\mathcal{A}(a) = \{a\}$
  $\mathcal{A}(f(ex_1,...,ex_n)) = \{f(t_1,...,t_n) \mid t_1 \in \mathcal{A}(ex_1),...,t_n \in \mathcal{A}(ex_n)\}$
  $\mathcal{A}(f_{(i)}^{-1}(ex)) = \{t_i \mid f(t_1,...,t_n) \in \mathcal{A}(ex)\}$
  $\mathcal{A}(\bot_{sc}) = \emptyset$ et $\mathcal{A}(\top_{sc}) = \mathcal{T}erm(\mathcal{F})$
  $\mathcal{A}(ex \cup ex') = \mathcal{A}(ex) \cup \mathcal{A}(ex')$
  $\mathcal{A}(ex \cap ex') = \mathcal{A}(ex) \cap \mathcal{A}(ex')$
  $\mathcal{A}(\overline{ex}) = \mathcal{T}erm(\mathcal{F}) - \mathcal{A}(ex)$
- l'interprétation de C est donnée par :
  $\mathcal{A}(ex \subseteq ex')$ ssi $\mathcal{A}(ex) \subseteq \mathcal{A}(ex')$
  $\mathcal{A}(ex \not\subseteq ex')$ ssi $\mathcal{A}(ex) \not\subseteq \mathcal{A}(ex')$
  $\mathcal{A}(\bot) = false$ et $\mathcal{A}(\top) = true$

Expressions et contraintes ensemblistes closes sont donc interprétées sans ambiguïté par rapport à $\mathcal{A}$. Les systèmes de contraintes ensemblistes sont, quant à eux, interprétés comme une conjonction de contraintes ensemblistes. Il est à souligner qu'une forme courante de contraintes ensemblistes est :

- $ex = ex'$

Cette équation ensembliste correspond naturellement à la conjonction de deux contraintes positives, c'est à dire : $ex = ex'$ ssi $ex \subseteq ex' \wedge ex' \subseteq ex$. Il s'agit donc avant tout d'un raccourci d'écriture.

Dans un second temps, nous prenons en compte les variables ensemblistes. Désignons par $\mathcal{V}al$ l'ensemble des applications définies de $\mathcal{V}ens$ sur $\mathcal{D} = \mathcal{A}(term\_set) = \wp(\mathcal{T}erm(\mathcal{F}))$. Tout élément θ de $\mathcal{V}al$ est appelé une assignation de variables. Soit $\mathcal{S}$ un système de contraintes ensemblistes et soit θ une

assignation de variables, si chaque contrainte ex $\subseteq$ ex' de $\mathcal{S}$ est telle que $\theta(ex) \subseteq \theta(ex')$[†] est évaluée à vrai (par rapport à $\mathcal{A}$), alors on dit que θ est une solution de $\mathcal{S}$ et que $\mathcal{S}$ est satisfiable. Il arrive que dans le domaine des contraintes ensemblistes, une assignation de variables soit appelée une interprétation et qu'une solution soit appelée un modèle (assignation de variables et Σ-algèbre sont alors confondues). Nous avons préféré garder les notations et la démarche que nous avons utilisées jusqu'ici afin de conserver une certaine homogénéité dans notre propos.

Il est intéressant de pouvoir comparer deux solutions et deux systèmes d'équations ensemblistes par rapport à un ensemble de variables.

Définition 7.4

$\forall\ (\theta,\theta') \in \mathcal{V}al^{\mathcal{Q}}, \forall\ V \in \wp(\mathcal{V}_{sc})$,

- $\theta \subseteq_V \theta'$ ssi $\forall\ \mathcal{X} \in V, \theta(\mathcal{X}) \subseteq \theta'(\mathcal{X})$
- $\theta =_V \theta'$ ssi $\forall\ \mathcal{X} \in V, \theta(\mathcal{X}) = \theta'(\mathcal{X})$

Si on note par $\theta_{|V}$ la restriction du domaine de définition d'une assignation θ par rapport à un ensemble de variables V, alors $\theta =_V \theta'$ ssi $\theta_{|V} = \theta'_{|V}$.

Définition 7.5

Soient $\mathcal{S}$ et $\mathcal{S}'$ deux systèmes de contraintes ensemblistes et soit V un ensemble de variables ensemblistes, $\mathcal{S} \Leftrightarrow_V \mathcal{S}'$ ssi

- pour toute solution θ de $\mathcal{S}$, il existe une solution θ' de $\mathcal{S}' \mid \theta =_V \theta'$
- pour toute solution θ' de $\mathcal{S}'$, il existe une solution θ de $\mathcal{S} \mid \theta =_V \theta'$

Suivant les classes de contraintes ensemblistes considérées, les propriétés relatives aux solutions et aux systèmes varient. Les propriétés étudiées sont en général l'existence d'une solution (la satisfiabilité d'un système), l'existence d'une plus petite solution, l'existence d'une plus grande solution, l'équivalence de deux systèmes...

Par exemple, le système

$$\mathcal{S}_1 = \{\ \mathcal{X}=a\cup f(\mathcal{X})\ ,\ \mathcal{Y}=b\}$$

est un système admettant une seule solution θ (par rapport à $\{\mathcal{X},\mathcal{Y}\}$) :

$\theta(\mathcal{X}) = \{a,f(a),f^2(a),...\}$
$\theta(\mathcal{Y}) = \{b\}$

[†] $\theta(ex)$ désigne l'expression ex où toute variable $\mathcal{X}$ a été remplacée par $\theta(\mathcal{X})$

Le système

$$S_2 = \{\ \mathcal{X} \supseteq a \cup f(f(\mathcal{X}))\ \}$$

admet une infinité de solutions (par rapport à $\{\mathcal{Y}\}$) :

$\theta_1(\mathcal{X}) = \{a, f(a), f^2(a), \ldots\}$
$\theta_2(\mathcal{X}) = \{a, f^2(a), f^3(a), \ldots\}$
$\theta_3(\mathcal{X}) = \{a, f^2(a), f^4(a), f^5(a), \ldots.\}$
...

et une plus petite solution θ telle que $\theta(\mathcal{X})$ désigne l'ensemble des termes de la forme $f^n(a)$ où n est un entier pair. Il admet également une plus grande solution θ' telle que $\theta'(\mathcal{X}) = \mathcal{T}erm(\mathcal{F})$.

Le système

$$S_3 = \{\ \mathcal{X} \cap \mathcal{Y} \subseteq 0\ ,\ a \subseteq \mathcal{X} \cup \mathcal{Y}\ \}$$

admet deux solutions (par rapport à $\{\mathcal{X},\mathcal{Y}\}$) qui sont incomparables :

$\theta_1(\mathcal{X}) = \{a\}$, $\theta_1(\mathcal{Y}) = \emptyset$
$\theta_2(\mathcal{X}) = \emptyset$, $\theta_2(\mathcal{Y}) = \{a\}$

Pour finir cette partie, notons que la liste des opérations ensemblistes introduites ci-dessus (union, intersection, projection, complémentaire) n'est pas exhaustive. Il est possible, bien sur, de considérer d'autres opérations. Par exemple, [Heintze 92] utilise un opérateur de complémentation limitée qui ne peut s'appliquer qu'à une expression ensembliste close. [Heintze 92] utilise également des opérateurs quantifiés. Pour simplifier, ces opérateurs permettent d'établir certaines conditions de la forme $X \in ex$ et $X \dagger ex$, conditions exprimant intuitivement une relation d'appartenance et une relation de différence. Un autre opérateur considéré par [Heintze et Jaffar 92a] est l'opérateur d'union ensembliste, noté $\uplus$, qui ignore les dépendances entre les arguments. L'interprétation (définition) de cet opérateur est basée sur celle de l'opérateur, notée *, de clôture distributive ("tuple distributivity") introduit par [Mishra 84]. Intuitivement, la clôture distributive, notée ex*, d'une expression ensembliste (close) ex désigne l'ensemble de tous les termes, obtenus en échangeant simplement les arguments des termes qui appartiennent à $\mathcal{A}(ex)$ et qui sont identiques aux arguments près. Reprenons les définitions adoptées par [Heintze et Jaffar 90] et [Heintze et Jaffar 92a] pour * et $\uplus$:

- $\mathcal{A}(ex^*) = \{c \mid c \in \mathcal{F}_0\} \cup \bigcup_{f \in \mathcal{F}} f((f^{-1}_{(1)}(\mathcal{A}(ex)))^*, \ldots, (f^{-1}_{(n)}(\mathcal{A}(ex)))^*)$
- $\mathcal{A}(ex \uplus ex') = \mathcal{A}((ex \cup ex')^*)$

Par exemple,

$(f(a,b) \cup f(c,d))^* = \{f(a,b),f(a,d),f(c,b),f(c,d)\}$
$f(a,b) \uplus f(b,a) = \{f(a,a),f(a,b),f(b,a),f(b,b)\}$

Une dernière opération ensembliste intéressante est proposée par [Bachmair et al. 92a]. Il s'agit d'une opération de diagonalisation, notée Δ^f, et interprétée comme suit pour tout symbole de fonction f d'arité strictement supérieure à 1 :

- $\mathcal{A}(\Delta^f(ex)) = \{t \mid f(t,...,t) \in \mathcal{A}(ex)\}$

Cette opération est généralisée dans [Bachmair et al. 92b] sous la forme $\Delta^f_{i=j}$. L'opération Δ^f établit une contrainte d'égalité sur l'ensemble des arguments d'un terme de la forme $f(t_1,...,t_n)$ alors que l'opération $\Delta^f_{i=j}$ établit une simple contrainte d'égalité entre les arguments t_i et t_j.

Tous ces opérateurs spécifiques ont été introduits car ils représentent des outils pour l'analyse de programmes. C'est le cas des opérateurs de complémentation limitée et des opérateurs quantifiés par rapport au travail de [Heintze 92]. L'opérateur $\uplus$ (ou l'opérateur *) est utilisé pour effectuer des approximations ([Mishra 84], [Bruynooghe et al. 87], [Yardeni et Shapiro 91] et [Yardeni 92]). L'opérateur de diagonalisation devrait permettre l'élaboration d'analyses plus efficaces car cet opérateur établit une dépendance forte entre les arguments (ce qui n'est pas le cas avec les opérations classiques). Pour le moment, nous n'avons eu connaissance d'aucune utilisation de cet opérateur (si ce n'est par une illustration donnée par [Bachmair et al. 92b]).

1.2 Classes de contraintes ensemblistes

Dans cette partie, nous discutons des différentes classes de contraintes ensemblistes par le biais des algorithmes (essentiellement des algorithmes de transformation) et résultats théoriques (essentiellement des résultats de satisfiabilité) qui ont été conçus, en particulier, pour chacune de ces classes. Il est à remarquer que les résultats de satisfiabilité sont généralement constructifs, i.e., ils mettent en évidence une solution (la plus petite, dans la plupart des cas) sous une forme explicite. De manière générale, une forme explicite est décrit par un système d'équations ensemblistes, une grammaire (régulière) de termes ou encore un automate d'arbres. Cette forme correspond donc, dans la plupart des cas, à un langage régulier, forme qui :

- s'obtient après plusieurs phases de simplification pendant lesquelles certaines opérations (intersection, projection, complémentation, ...) sont éliminées.

- permet de résoudre facilement certains problèmes par utilisation directe des algorithmes connus sur les grammaires de termes ou les automates d'arbres.

Par la suite, nous utiliserons beaucoup les systèmes d'équations ensemblistes. Aussi, donnons-nous la définition suivante :

Définition 7.6

Un système d'équations ensemblistes $\mathcal{S}$ est un ensemble fini $\{\mathcal{X}_1 = ex_1,...,\mathcal{X}_n = ex_n\}$ où $\mathcal{X}_1,...,\mathcal{X}_n$ désignent des variables ensemblistes toutes distinctes et $\mathcal{X}_i \neq ex_i$ pour $1 \leq i \leq n$. L'ensemble de ces variables désigne le domaine de $\mathcal{S}$, noté $Dom(\mathcal{S})$.

Notons la propriété suivante [Heinze 92] : Tout système d'équations n'utilisant que des opérateurs ensemblistes monotones possède une plus petite solution. Une forme explicite relativement utilisée est la forme minimale définie comme suit :

Définition 7.7

Un système d'équations ensemblistes $\mathcal{S}$ est résolu (ou sous forme résolue) ssi toute équation $\mathcal{X} = ex$ de $\mathcal{S}$ est telle que ex désigne une conjonction d'éléments qui sont soit des constantes, soit des termes de la forme $f(\mathcal{Y}_1,...,\mathcal{Y}_n)$.

Un système résolu $\mathcal{S}$ possède un seul modèle (par rapport à $Dom(\mathcal{S})$). Nous abordons maintenant les différentes classes de (systèmes de) contraintes ensemblistes telles qu'elles ont été étudiées.

La première classe est celle des systèmes d'équations ensemblistes où les seules opérations autorisées sont l'union et la projection. Un système de contraintes de cette classe est toujours satisfiable. [Reynolds 69] et [Jones et Muchnick 79] proposent un algorithme de transformation sous forme résolue. [Reynolds 69] s'attache, en particulier, à décrire un algorithme d'inférence de types pour LISP (pur). Les fonctions de projection considérées (fonctions "analytiques pour [Reynolds 69]) sont les fonctions car et cdr qui représentent, de fait, les opérations de projection $cons^{-1}_{(1)}$ et $cons^{-1}_{(2)}$.

La seconde classe abordée dans la littérature est celle des systèmes d'équations ensemblistes où les seules opérations autorisées sont, cette fois-ci, l'union et l'intersection. Un système de contraintes de cette classe est toujours satisfiable. [Heintze 90] propose un algorithme de transformation sous forme résolue. [Uribe 92a,92b] reprend les algorithmes de [Heintze 90] en les modifiant car l'interprétation qu'il a choisie n'est pas classique (elle permet l'expression d'une certaine dépendance entre les variables).

[Heintze et Jaffar 91] ont investi la classe des systèmes de contraintes ensemblistes définies ("definite"), contraintes de la forme ex $\supseteq$ ex' sans complément et sans opérateur ensembliste dans ex. Un système de contraintes de cette classe possède une plus petite solution, lorsqu'il est satisfiable. La plus petite solution (si elle existe) est fournie sous la forme d'une grammaire de termes par un algorithme dont la preuve de correction est assez complexe.

Pour la classe des contraintes ensemblistes n'incluant aucun opérateur de projection, [Aiken et Wimmers 92] propose un algorithme qui, pour tout système de contraintes, retourne celui-ci (lorsqu'il est satisfiable) sous une forme résolue (différente de celle définie plus haut). Le problème de la satisfiabilité d'un tel système est prouvé être EXPTIME-dur et dans NEXPTIME. [Gilleron et al. 92] présentent un algorithme analogue à celui de [Aiken et Wimmers 92] en introduisant une nouvelle classe d'automates : les automates d'ensembles d'arbres (appelés à l'origine automates d'arbres avec variables libres). Les systèmes sous forme résolue de [Aiken et Wimmers 92] sont à comparer aux automates d'ensembles arbres de [Gilleron et al. 92]. L'équivalence n'est pas prouvée.

[Bachmair et al. 92] étendent les résultats de [Aiken et Wimmers 92] et [Gilleron et al. 92] en permettant l'utilisation des opérations de diagonalisation et de projection. Toutefois, l'opération de projection ne peut apparaître dans une expression qu'avec une polarité négative (il s'agit d'une restriction d'occurrence de l'opération de projection liée à la structure de l'expression ensembliste). [Bachmair et al. 93] montrent que les contraintes ensemblistes (sans projections) correspondent à la classe monadique (la classe des formules du premier ordre sans symboles de fonction, avec des symboles de prédicat unaires et avec une quantifications arbitraire) et que les contraintes ensemblistes avec diagonalisations (généralisées) et projections de polarité négative correspondent à la classe monadique avec égalités (où des équations entre variables sont autorisées dans les formules). Les résultats de décidabilité et de complexité de la classe monadique peuvent ainsi s'appliquer : le problème de la satisfiabilité des contraintes ensemblistes est, de cette manière, montrée NEXPTIME-complet.

Pour finir, pour la classe des contraintes ensemblistes positives et négatives n'incluant aucun opérateur de projection, le problème de la satisfiabilité d'un système de contraintes est décidable. [Gilleron et al. 93] et [Aiken et al. 93] ont prouvé ce résultat.

2 Types

2.1 Introduction

Les langages de programmation non typés, tels que Prolog, ont comme principal avantage d'assouplir la tâche du programmeur en ne lui imposant aucune contrainte de typage, mais ont, par conséquent, comme principal inconvénient d'affaiblir la sécurité et l'efficacité des programmes. Il est donc intéressant de chercher à intégrer dans ces langages une notion de typage qui ne soit pénalisante ni pour le programmeur ni pour les programmes. Ceci ne peut s'effectuer que de manière automatique ou semi-automatique, c'est à dire sans l'intervention ou avec une intervention limitée du programmeur. Dans ce dernier cas, le programme peut s'avérer être plus lisible ou plus intelligible. Il en résulte que le typage est considéré en programmation logique sous deux formes d'applications principales :

- la vérification de types ("type checking")
- l'inférence de types ("type inference")

La vérification de type consiste à vérifier la bonne construction des règles d'un programme par rapport à un ensemble de types déclarés au préalable. Son application essentielle est donc la détection, au moment de la compilation, d'erreurs de typage dans un programme. L'inférence de type consiste à générer un ensemble de valeurs possibles pour chaque argument ou chaque variable apparaissant dans un programme. Son application essentielle est alors l'optimisation du code généré par le compilateur. On appellera système de types, tout système permettant d'effectuer de la vérification et/ou de l'inférence de types. On notera au passage qu'inférence et vérification peuvent être associées dans un même système.

La notion de polymorphisme est un autre aspect important de l'étude des types. Les deux sortes de polymorphisme universel [Cardelli et Wegner 85] se révèlent être particulièrement intéressantes à gérer en programmation logique, à savoir :

- le polymorphisme paramétrique (ou généricité)
- le polymorphisme inclusif (ou héritage)

Le polymorphisme paramétrique désigne la possibilité de définir des types génériques, i.e., des types paramétrés. Le polymorphisme inclusif désigne la possibilité de définir des sous-types (héritant). Une définition de type est donc générique ssi l'ensemble des termes qu'il représente n'est pas déterminé de façon unique par sa définition. Par exemple,

type List = nil | cons(All,List)†

désigne l'ensemble des listes dont les éléments sont quelconques (All désigne tous les éléments du domaine considéré). Cette définition n'est pas générique. Par contre,

type List(α) = nil | cons(α,List(α))

désigne l'ensemble des listes dont les éléments sont du type α sachant que α est une variable de type (un paramètre). Cette définition est générique. Une définition de type additive est la donnée de tous les sous-types pour un type donné. Par exemple,

type Parent = Mère | Père

présente les sous-types de Parent, à savoir, Mère et Père. Les sous-types Mère et Père hérite de Parent, i.e., peuvent utiliser les fonctionnalités de Parent.

D'autre part, quand on s'intéresse de plus près aux systèmes de types, on découvre principalement deux modèles d'interprétation pour les types :

- un modèle descriptif pour les appels
- un modèle descriptif pour les solutions.

Le premier modèle d'interprétation est lié aux systèmes de vérification de types. Suivant ce modèle, le type d'un prédicat doit décrire (ou approcher) l'ensemble des atomes pour lesquels le prédicat peut être appelé. Tout atome qui n'est pas pris en compte par le type du prédicat ne peut donc pas être appelé. Aussi, un programme P est bien typé ssi aucun conflit de type ne peut survenir pendant l'exécution de P. Et le fait qu'un programme P soit bien typé est indépendant du comportement opérationnel de P, à savoir, par exemple si un succès ou un échec est obtenu. En d'autres termes, cela revient à considérer les déclarations de types comme des restrictions sur les objets typés. Ce modèle d'interprétation est utilisé, entre autres, par [Milner 78], [Mycroft et O'Keefe 84] et [Frühwirth 90]. Un slogan a été proposé par [Mycroft et O'Keefe 84] pour le qualifier : "Well-typed programs do not go wrong".

Le second modèle d'interprétation est liè aussi bien aux systèmes de vérification de types qu'aux systèmes d'inférence de types. Suivant ce modèle, le type d'un prédicat doit décrire (ou approcher) l'ensemble des succès pour lesquels le

---

† Les définitions de types sont précédées du mot-clef "type" et utilisent le symbole | pour exprimer une disjonction.

prédicat peut réussir. Tout atome qui n'est pas pris en compte par le type du prédicat ne peut donc pas être réfuté. Aussi, pour les systèmes de vérification de types, un programme est bien typé ssi les déclarations de type associées au programme correspondent à une approximation de l'ensemble des succès du programme. Ce modèle d'interprétation est utilisé suivant une approche opérationnelle par [Horiuchi et Kanamori 87], [Bruynooghe et Janssens 88] et [Xu et Warren 88], suivant une approche de point fixe par [Yardeni et Shapiro 91], [Yardeni 92] et [Frühwirth et al. 92] et suivant une approche dénotationnelle par [Mishra 84], [Zobel 87] et [Heintze 92]. Un slogan a été également proposé par [Mishra 84] pour qualifier ce modèle d'interprétation : "Ill-typed programs cannot succeed".

Dans ce qui suit, nous présentons un certain nombre de systèmes de types proposés pour la programmation logique :

- les systèmes de types syntaxiques
- les systèmes de types sémantiques

Les premiers sont basés sur le premier modèle d'interprétation et les seconds sont basés sur le second modèle d'interprétation.

Dans tous les systèmes de types, des restrictions sont imposées à la définition des types. Ceci s'explique par le fait que le problème de la vérification de types et de l'inférence de types est indécidable dans le cas général. Une classe de types a été identifiée comme satisfaisante : la classe des types réguliers, i.e., les types qui peuvent être définis par une grammaire de termes (ou un automate d'arbres). En effet, de nombreuses opérations sont envisageables (comparaison, intersection, ...) pour cette classe [Dart et Zobel 92]. Une classe plus petite que la précédente et très utilisée est celle des types réguliers clos par * (la clôture distributive). Les contraintes ensemblistes sont une autre manière de coder les types. Elles offrent un formalisme relativement clair et puissant.

Nous présentons maintenant un certain nombre de travaux relatifs aux systèmes de types en nous attardant davantage sur les systèmes de types sémantiques. Un certain nombre des derniers papiers du domaine est rassemblé dans l'ouvrage [Types in LP 92].

2.2 Systèmes de types syntaxiques

Les systèmes de types syntaxiques sont par nature des systèmes de vérification de types. La vérification de types est essentiellement utilisée pour détecter de façon précoce les erreurs de typage dans un programme. Cela sous-entend, en général, que l'utilisateur effectue au préalable certaines déclarations, afin de fournir au système d'une part la définition des types élémentaires et d'autre part

la spécification du typage de chaque prédicat. Ces spécifications permettent alors de tester le fait qu'un programme soit bien typé.

Commençons par illustrer de manière informelle le type d'erreur qu'il est possible de rencontrer dans un programme logique. Une erreur courante est la transposition de deux arguments lors de la définition d'un prédicat. Par exemple, si nb_list est un prédicat destiné à caractériser les listes L à N éléments, alors il est évident que la définition suivante est mauvaise du fait de l'inversion des paramètres dans le fait.

```
nb_list(nil,0).
nb_list(s(N),cons(Y,L)) :- nb_list(N,L).
```

Ce type d'erreur peut très bien être rattrapé par une analyse à la compilation si on dispose (par exemple) de la déclaration suivante concernant le typage des arguments du prédicat nb_list.

```
pred nb_list(Int, List(Int))†
```

Cette spécification de type pour le predicat nb_list indique que le premier argument est un entier et que le second argument est une liste d'entiers. Les types Int et List doivent bien entendus être définis par ailleurs et correspondre à la représentation symbolique utilisée par nb_list :

```
type Int = 0 | s(Int)
type List(α) = nil | cons(α,List(α))
```

Une autre erreur relevée par [Mycroft et O'Keefe 84] est l'omission de cas lors de la définition d'un prédicat. Par exemple, si le fait était omis dans l'exemple précédent, alors il serait possible de détecter à la compilation cet oubli en utilisant la définition de type de nb_list. Cette technique nécessite toutefois la prise en compte explicite des cas qui échouent. Par exemple, si sup_1 est un prédicat destiné à caractériser les entiers supérieurs ou égaux à 1 et que la spécification de sup_1 est donnée par :

```
pred sup_1(Int)
```

alors sup_1 doit être défini comme suit :

```
sup_1(0) :- fail.
sup_1(s(0)).
```

---

† Les spécifications de types sont précédées du mot-clef "pred".

```
sup_1(s(N)) :- sup_1(N).
```

Ceci donne une certaine mesure de l'intérêt (et des problèmes) relatif(s) au typage. Nous étudions maintenant quelques contributions apportées à l'étude des systèmes de types syntaxiques. [Bruynooghe 81] est le premier à suggérer l'addition d'information de types et de modes dans un programme Prolog. Utilisant l'information de mode, il propose un algorithme pour vérifier la consistance du typage d'un programme. Toutefois, le cas du polymorphisme n'est pas abordé.

La première étude concernant le polymorphisme (paramétrique) a été formulée, semble-t-il, par [Milner 78]. Celui-ci introduit une "discipline" de type qu'il intègre à un langage applicatif simple. Il est alors possible d'inférer (et par conséquent de vérifier) le typage de toute expression de ce langage à partir d'opérateurs polymorphes primitifs. Une distinction est faite toutefois entre variables de type génériques et non génériques suivant le lien où elles sont attachées dans l'expression. [Mycroft et O'Keefe 84] adapte, en levant certaines restrictions, le travail de [Milner 78] à Prolog. Des déclarations (définitions de types et spécifications du typage de chaque prédicat) sont ajoutées au programme. En un pas de calcul (essentiellement, l'unification), la vérification est effectuée pour tout programme (et tout but). Ce résultat reste valable pour n'importe quelle stratégie de résolution. Mais, comme pour [Milner 78], le polymorphisme inclusif n'est pas envisagé.

Pour sa part, [Frühwirth 90] propose un algorithme de vérification de type basé sur la méta-interprétation. Ce travail correspond, dans une large mesure, à une extension de celui de [Mycroft et O'Keefe 84]. Le système de [Frühwirth 90] supporte le polymorphisme inclusif et les spécifications sont codées directement en Prolog. De ce fait "Prolog is the typed language is the type language is the system implementation language".

2.3 Systèmes de types sémantiques

L'information calculée par un système de types sémantiques permet certaines optimisations telles que la génération de code spécialisée (e.g., sélection d'une meilleure implémentation des structures de données), l'utilisation de modules d'unification (ou de satisfaction de contraintes) spécifiques, la suppression de portions de code inaccessibles et de points de choix. [Marien et al. 89] présentent l'impact d'un tel système sur la génération de code.

Alors qu'au niveau des systèmes de types syntaxiques, le calcul peut s'effectuer en un pas (un peu à la manière d'un raisonnement par récurrence), au niveau des systèmes de types sémantiques une certaine approximation (abstraction) du

calcul d'inférence est nécessaire. De ce fait, on retrouve les trois approches mentionnées au chapitre 4 :

- une approche opérationnelle
- une approche de point fixe
- une approche dénotationnelle

Typiquement, les trois approches procèdent de la manière suivante par rapport à l'inférence de types. L'approche opérationnelle consiste à définir une sémantique opérationnelle abstraite, puis à calculer cette sémantique, en utilisant, si nécessaire, une opération de widening et une technique de tabulation ou une méthode équivalente. L'approche de point fixe consiste à définir un opérateur de conséquence immédiate abstrait, puis à considérer un point fixe de cet opérateur. L'approche dénotationnelle consiste à générer un système de contraintes ensemblistes à partir d'un programme, puis à considérer un modèle de ce système.

2.3.1 Approche opérationnelle

L'approche opérationnelle consiste à adopter une technique d'interprétation abstraite classique (abstraction du domaine + abstraction du calcul). Lorsque le domaine considéré est infini (et ne satisfait pas la condition de chaîne ascendante), une opération de narrowing est introduite et une technique de tabulation (ou une méthode équivalente) utilisée. [Kanamori et Kawamura 87], [Bruynooghe et al. 87] et [Xu et Warren 88] utilisent cette approche.

[Kanamori et Kawamura 87] proposent plusieurs analyses pour illustrer leur modèle d'interprétation abstraite basé sur la résolution OLDT (voir [Kanamori et Kawamura 90]) : une inférence des schémas d'appel et de solution (par abstraction sur la profondeur des termes), une inférence de types et une inférence de modes. L'inférence des schémas d'appel et de solution peut être considérée, dans une certaine mesure, comme une inférence de types puisqu'elle permet de caractériser le "type" des appels et des solutions pour une exécution donnée. Toutefois, cette inférence ne permet pas de coder les structures récursives et se prête donc mal (et même pas du tout) au codage des types récursifs. L'inférence de types, proprement dite, de [Kanamori et Kawamura 87] se base sur des définitions préalables de types (mais pas sur des spécifications de types pour chaque prédicat) codées en prolog. L'ensemble de ces définitions forme un treillis fini (via l'adjonction d'un plus petit élément et d'un plus grand élément) et est considéré comme domaine abstrait. Il est alors possible d'utiliser ce domaine par rapport à la résolution OLDT et d'obtenir un calcul fini. Ce système d'inférence de types monomorphique est étendu en un système d'inférence de types polymorphique par [Horiuchi et Kanamori 87]. Notons cependant que les définitions de types (génériques) doivent être

nécessairement disjointes, i.e., deux types ne peuvent partager des éléments que si ils sont instances d'une même définition.

Pour démontrer l'effectivité du modèle d'interprétation abstraite proposé par [Bruynooghe et al. 87] et [Bruynooghe 90], [Bruynooghe et Janssens 88] ont développé deux instances de ce modèle générique : une inférence de types et une inférence de modes. Nous avons déjà débattu, au chapitre 2, de l'inférence de types mais nous cherchons maintenant à mettre en évidence le lien direct du typage utilisé avec les contraintes ensemblistes. Il est visible, en effet, que les définitions de types utilisées par [Bruynooghe et Janssens 88] correspondent à des systèmes d'équations ensemblistes. De plus, ces systèmes sont sous forme résolue et n'admettent, de ce fait, qu'une seule solution. Notons que les définitions de types proposées par [Bruynooghe et Janssens 88] sont un peu plus générales que celles de [Bruynooghe et al. 87] puisqu'elles supportent le polymorphisme. Notons cependant que les variables de types (paramètres) ne peuvent être associées qu'au but, et instanciées (éventuellement) pendant la résolution. Par ailleurs, les types considérés par [Bruynooghe et Janssens 88] désignent des ensembles de termes clos et non clos (à la différence des approches classiques). Cette possibilité d'intégrer des termes non clos est clairement illustrée par le passage des graphes de type rigides ("rigid type graphs") aux graphes de type intégrés ("integrated type graphs") dans le papier de [Janssens et Bruynooghe 92]. Comme nous l'avons déja précisé au chapitre 4, une opération de widening est utilisée pendant l'analyse d'un programme pour limiter la largeur et la profondeur des graphes de type. La restriction sur la largeur ("principal label restriction") correspond à l'application de la clôture distributive * et la restriction sur la profondeur ("depth restriction") permet de créer des structures récursives. Signalons aussi que pour affiner l'analyse, des informations (essentiellement des informations sur le partage) annexes au typage proprement dit, sont utilisées. En résumé, le travail de [Bruynooghe et al. 87], [Bruynooghe et Janssens 88] et [Janssens et Bruynooghe 92] est assez caractéristique des analyses qui sont conçues pour obtenir le maximum d'information sur le flux de données d'un programme, ceci afin de pouvoir optimiser le code généré lors de la compilation. Cette approche est aussi plus souple que celle de [Kanamori et Kawamura 87] puisqu'aucune définition de types n'est requise.

Le système d'inférence de types proposé par [Xu et Warren 88] cherche à conjuguer l'efficacité des systèmes utilisant des déclarations avec la souplesse des systèmes n'utilisant aucune déclaration. Une base de types (représentant une partition de l'univers de Herbrand) est, pour cela, définie. Elle permet, de manière additive, la construction d'autres types. L'inférence de types proposée consiste à utiliser la sémantique opérationnelle de Prolog à partir du domaine des types. Pour obtenir une interprétation finie, une abstraction des termes sur la profondeur de répétition (la "repetition depth" est une abstraction voisine de la

"depth abstraction") ainsi qu'une technique de tabulation sont introduites. Pour améliorer l'efficacité de l'analyse, des définitions de types (avec certaines limitations, e.g., une constante ne peut être partagée par deux définition) et des spécifications de type peuvent être intégrées au programme. Dans ce cas, le système agit à la fois comme système de vérification de types et comme système d'inférence de types. Ce système supporte le polymorphisme d'inclusion (voir la construction de type) mais ne supporte qu'un polymorphisme paramétrique limité (car les types primitifs ne sont pas paramétrés).

2.3.2 Approche de point fixe

L'approche de point fixe consiste essentiellement à définir un opérateur de conséquence immédiate abstrait, puis à considérer le plus petit point fixe comme un type. [Yardeni et Shapiro 91], [Yardeni et al. 92] et [Frühwirth et al. 91] utilisent cette approche.

[Yardeni et Shapiro 91] utilisent une adaptation de la définition de l'opérateur conséquence immédiate T_P. L'opérateur, noté T_P^*, qu'ils utilisent est en effet défini comme suit :

$$\forall\ I \in \wp(B_P),\ T_P^*(I) = (T_P(I))^*$$

L'abstraction introduite consiste donc à ignorer toutes les dépendances inter-arguments puisque l'opérateur de clôture distributive est appliqué sur T_P. Un programme est bien typé par un type S ssi

- $T_P^*(S) = S$
- $T_{[C]}^*(S) \neq \emptyset$

Le type S doit être un point fixe de l'opérateur T_P^* et aucune clause C de P ne doit être inutile par rapport à S. Pourquoi un point fixe ? En fait, choisir un pré-point fixe (un type S tel que $T_P^*(S) \supseteq S$) ne serait pas consistant et choisir un post-point fixe (un type S tel que $(T_P^*(S) \subseteq S$) serait une approximation peu intéressante alors qu'il existe nécessairement un point fixe plus petit que ce post-point fixe (T_P^* est prouvé continue). Pourquoi pas le plus petit point fixe ? Cela reviendrait à calculer ce point fixe (et cela n'est pas possible en général). On se contente donc d'un point fixe car l'approximation est consistante et il n'est pas possible de vérifier si il s'agit du plus petit point fixe. Tout point fixe de l'opérateur $T_{[C]}^*$ s'avère être un langage régulier, aussi est-il possible de limiter les définitions de types aux ensembles réguliers de termes clos. En résumé, même si le travail de [Yardeni et Shapiro 91] ne supporte pas le polymorphisme, il est clair qu'il constitue une base théorique solide.

[Yardeni et al. 92] est essentiellement une extension de [Yardeni et Shapiro 91] et [Fruhwirth 90]. Dans ce papier, les auteurs considèrent un langage de types qui supporte le polymorphisme (paramétrique et inclusif) non seulement pour les définitions de types mais également pour les définitions de prédicats. Ce gain de généricité permet un style de programmation relativement puissant. Voici un exemple de définition possible :

```
type list(α) = nil | cons(α,list(α))

pred append(α)(list(α),list(α),list(α))

append(α)(nil,L,L).
append(α)(cons(X,L1),L2,cons(X,L3)) :- append(α)(L1,L2,L3).
```

La définition de type est classique, la spécification de type pour le prédicat append l'est moins. En effet, celui-ci est paramétré par une variable de type α. Il est alors possible d'utiliser append en instanciant la variable de type, par exemple,

```
:- append(int)(nil,cons(0,nil),L)
```

et

```
:- append(list(int))(nil,cons(cons(s(0),nil),nil),L)
```

sont deux buts bien typés. La syntaxe d'un programme logique typé est établie dans un langage du second ordre, mais sa sémantique reste du premier ordre. Le principe de l'algorithme de vérification de types est celui présenté par [Yardeni et al. 92].

Le travail de [Frühwirth et al. 91] constitue à la fois une approche dénotationnelle et une approche de point fixe. En effet, les auteurs utilisent la classe des programmes logiques à prédicats unaires (tous les symboles de prédicat sont unaires) pour coder l'approximation d'un programme logique. Un programme de cette classe peut être observé sous l'angle dénotationnel (il est équivalent à un système de contraintes ensemblistes) ou sous l'angle de point fixe (en considérant l'opérateur de conséquence immédiate). Pour un programme P donné, [Frühwirth et al. 91] proposent une première série de transformations qui se termine par un programme dont la sémantique de point fixe est exactement celle obtenue en considérant $\mathcal{T}_P$ (voir [Heintze et Jaffar 90] plus loin). L'approximation effectuée consiste donc à ignorer toutes les dépendances inter-variables. La complexité de la vérification de types et de l'inférence de types par rapport à ce système est montrée exponentielle. [Frühwirth et al. 91] proposent également une seconde série de transformations qui se termine par un programme dont la sémantique de point fixe est exactement celle obtenue en considérant T_P^* (voir [Yardeni et Shapiro 91] plus haut). L'approximation

effectuée consiste donc à ignorer toutes les dépendances inter-arguments. Cette approximation est donc plus forte que la précédente. Aucune indication de complexité n'est donnée pour ce second cas.

2.3.2 Approche dénotationnelle

L'approche dénotationnelle consiste à associer une formule caractérisant un certain nombre de contraintes ensemblistes à partir de tout programme. Cette approche peut être qualifiée de dénotationnelle car la formule obtenue pour le programme est obtenue par composition des sous-formules obtenues pour chaque clause du programme. Une solution particulière de la formule (la plus petite solution, en général) détermine un certain nombre de types. [Mishra 84], [Zobel 87] et [Heintze et Jaffar 90] suivent cette approche.

[Mishra 84] développe un système d'inférence de types (non générique) pour Prolog. Les types sont codés sous formes de "regular trees" et correspondent à des variables ensemblistes. Par exemple, un type T peut être défini par le "regular tree" :

$$T = Z \,!\, 0 + succ(Z)$$

ou la variable ensembliste $\mathcal{T}$ suivante :

$$\mathcal{T} = 0 \cup s(\mathcal{T})$$

Dans le premier cas, + est interprété comme l'union et ! comme l'opérateur de plus petit point fixe. Z ! 0 + succ(Z) se lit donc comme : le plus petit ensemble Z incluant 0 et s(Z). Dans le second cas, l'équation possède une seule solution qui est $\mathcal{T}$ = T. [Mishra 84] utilise clairement le formalisme des contraintes ensemblistes, car il code, par simplicité, les "regular trees" par des "leaf linear equations", et celles-ci sont en fait des contraintes ensemblistes. C'est [Mishra 84] qui introduit l'opération, notée *, de clôture distributive ("tuple distributivity") définie plus haut. Cette opération lui permet d'approcher tout type T par T*. Le système d'inférence de types proposé permet d'engendrer des contraintes ensemblistes (y compris des contraintes ensemblistes positives) directement à partir du programme. Comme il existe des algorithmes bien connus sur les automates d'arbres (pour décider, par exemple, du vide et de l'égalité), il est possible de calculer, si elle existe, la plus grande solution pour l'ensemble des contraintes ensemblistes générées. Cette solution est une approximation de l'ensemble des succès.

[Zobel 87] propose, sur les mêmes bases que [Mishra 84], un système d'inférence de types supportant le polymorphisme. Les types syntaxiques que [Zobel 87] définit correspondent pour nous aux types sémantiques (sic).

L'inférence de types de [Zobel 87] s'accompagne d'une vérification de types, pour le programme à la compilation (comme pour [Mishra 84]), et pour le but à l'exécution. Le coeur de l'algorithme utilisé par [Zobel 87] se compose des deux étapes suivantes :

1) la dérivation de règles de typage en un seul pas de calcul

2) l'unification des règles obtenues.

Si l'unification échoue, alors c'est que le programme est mal typé. Notons qu'il est possible d'insérer des déclarations de types pour améliorer le résultat du système.

N. Heinze et J. Jaffar ont beaucoup travaillé sur l'inférence de types en utilisant l'approche dénotationnelle. [Heintze et Jaffar 90] proposent une approximation qui consiste à ignorer toutes les dépendances inter-variables. Le résultat de cette approximation est non seulement récursif mais peut ,de plus, être représenté sous la forme d'un système d'équations ensemblistes explicite. Il correspond à la plus petite solution ensembliste, notée $\mathcal{M}_P$, d'un programme P ainsi qu'au plus petit point fixe de l'opérateur conséquence immédiate approchée, notée $\mathcal{T}_P$. Pour définir $\mathcal{T}_P$, il est nécessaire d'introduire la notion de substitution ensembliste. Une substitution ensembliste α est simplement une application définie de l'ensemble des variables (ensemblistes) vers l'ensemble des termes clos. Si Θ est un ensemble de substitutions closes alors $\mathcal{A}(\Theta)$ est la substitution ensembliste qui à toute variable (ensembliste) X associe l'ensemble $\{\theta(X) \mid \theta \in \Theta\}$. L'opérateur $\mathcal{T}_P$ est défini comme suit :

$\forall\ I \in \wp(B_P)$, $\mathcal{T}_P(I) = \{a \mid a \in \alpha(h)$ où
- $h \leftarrow b_1, \ldots, b_n$ est une clause de P
- $\alpha = \mathcal{A}(\{\theta \mid \theta$ est une substitution close et $\{\theta(b_1),\ldots,\theta(b_n)\} \subseteq I\})\}$

[Heintze et Jaffar 90] montrent que $\mathcal{M}_P = \text{lfp}\ \mathcal{T}_P = \mathcal{T}_P{}^{\omega}(\emptyset)$ et proposent un algorithme qui à tout programme logique P associe après un certain nombre d'étapes de réduction un système d'equations explicite dont le plus petit (et unique) modèle correspond à lfp $\mathcal{T}_P$. Il s'avère que lfp $\mathcal{T}_P$ = lfp T_P pour une certaine classe de programmes logiques, celle des programmes P tel que pour chaque règle R de P, a) la tête de R soit linéaire et b) le corps de R puisse être partitionné en groupes d'atomes ne partageant aucune variable et ne faisant référence qu'à une seule variable de tête. Cette classe englobe la classe des programmes logiques unaires.

[Heintze 92] propose (suivant [Heintze 91]) une méthode d'analyse de programmes basée sur les contraintes ensemblistes. Cette méthode est appliquée non seulement à la programmation logique mais aussi à la programmation

impérative (et également de manière un peu plus informelle à la programmation fonctionnelle). Le principe de l'analyse est très simple puisque l'ignorance des dépendances inter-variables est la seule approximation considérée. La sémantique abstraite est, en conséquence, définie (de façon très générale), mais le moyen de la calculer reste à être précisé. Le calcul s'effectue en deux temps : le premier temps est consacré à l'obtention d'un système de contraintes ensemblistes dont la plus petite solution désigne la sémantique abstraite ; le second temps est consacré à la résolution (simplification) du système afin d'obtenir une forme explicite de la plus petite solution. [Heintze 92] introduit une formulation de la sémantique collectionneuse ("collecting semantics") sous forme de contraintes de contexte ("environment constraints"). Ces contraintes ont l'avantage de représenter les dépendances locales entre les différents points de programme, points situés entre deux instructions, tout en permettant différentes interprétations (assignations). Les variables apparaissant dans les contraintes de contexte sont appelées des variables de contexte. Un contexte désigne l'état des variables à un moment donné (i.e., une application qui à chaque variable associe une valeur unique) et une assignation permet d'associer un ensemble de contextes (comme valeur) à une variable de contexte. De façon analogue, un contexte ensembliste désigne les différents états possible des variables à un moment donné (i.e., une application qui à chaque variable associe un ensemble de valeurs) et une assignation ensembliste permet d'associer un contexte ensembliste (comme valeur) à une variable de contexte. La plus petite solution d'un système de contraintes de contexte (généré par rapport à un programme P) correspond à la sémantique collectionneuse de P, et la plus petite solution ensembliste correspond à la sémantique abstraite de P. Un système de contraintes de contexte est calculé automatiquement pour tout programme impératif ainsi que pour tout programme logique. De plus, pour la programmation logique, ce système est lié à la sémantique considérée : sémantique bottom-up, sémantique top-down standard (méthode de résolution de Prolog) ou non-standard (méthode de résolution quelconque). Pour finir, les contraintes de contexte sont traduites en contraintes ensemblistes et [Heintze 92] montre que la plus petite solution ensembliste d'un système de contraintes de contexte est égale à la plus petite solution du système de contraintes ensemblistes obtenues aprés traduction des contraintes de contexte. Le second temps du calcul, est consacré à la mise sous forme explicite de la plus petite solution du système. L'algorithme qui est conçu pour cet objectif est une extension des travaux de [Heintze et Jaffar 90,91] car il prend en compte des expressions ensemblistes quantifiées plus générales.

La description précédente aboutit donc à la décidabilité du calcul de la sémantique abstraite. Notons que les objectifs de [Heintze 91] étaient de définir une sémantique abstraite qui soit :

- déclarative
- pertinente
- décidable

Cela signifie que 1) la définition de l'approximation doit être intuitive et indépendante de considérations algorithmiques, 2) la sémantique abstraite doit être utile à l'analyse de programmes et 3) des algorithmes pour calculer la sémantique abstraite doivent exister. Tous ces objectifs semblent atteints. Reste le problème épineux de la complexité. [Frühwirth et al. 91] ont montré que, pour cette approximation, le problème de la vérification et de l'inférence de types était exponentielle. [Heintze 92a,92b] propose donc de tester une implémentation de leur système afin de juger la complexité réelle (pratique) de ses algorithmes. Il considère néanmoins quelques restrictions par rapport à son modèle original. Voici un exemple (figure 7.8) de dérivation de contraintes ensemblistes à partir d'un programme logique lorsque l'on considére la sémantique bottom-up.

```
p(X) :- q(X),r(X).
q(a).
q(f(Y)) :- q(Y).
r(f(Z)).
```

$$Ret_p := p(\mathcal{X}) \wedge \mathcal{X} = q^{-1}(Ret_q) \cap r^{-1}(Ret_r)$$
$$Ret_q := q(a) \cup q(f(\mathcal{Y})) \wedge \mathcal{Y} = q^{-1}(Retq)$$
$$Ret_r := r(f(\mathcal{Z})) \wedge \mathcal{Z} = 1$$

Figure 7.8

Pour tout prédicat p, une variable Ret_p est introduite (par commodité) et représente (une approximation de) l'ensemble des succès associé à p. La plus petite solution du système obtenu (par conjonction de chaque contrainte dérivée) donne (par rapport à $\{Ret_p, Ret_q, Ret_r\}$) :

$Ret_p = \{p(f(a)), p(f^2(a)),...\}$
$Ret_q = \{q(a), q(f(a)),...\}$
$Ret_r = \{r(f(t))$ pour tout terme clos $t\}$

Dans ce cas précis, cette solution correspond exactement à l'ensemble des succès du programme. Un autre exemple (figure 7.9) illustre la possibilité de considérer une sémantique top-down.

:- ^{1}p(W)^2.
p(X) :- ^{3}q(X) ^{4} r(X)^5.
q(a)^6.
r(Y)^7.

$Call_p := p(\mathcal{W}^1) \wedge \mathcal{W}^1 = 1$
$Ret_p := p(\mathcal{X}^5) \wedge \mathcal{X}^5 = p^{-1}(Call_p) \cap q^{-1}(Ret_q) \cap r^{-1}(Ret_r)$
$Call_q := q(\mathcal{X}^3) \wedge \mathcal{X}^3 = p^{-1}(Call_p)$
$Ret_q := q(a)$
$Call_r := r(\mathcal{X}^4) \wedge \mathcal{X}^4 = p^{-1}(Call_p) \cap q^{-1}(Ret_q)$
$Ret_r := r(\mathcal{Y}^7) \wedge \mathcal{Y}^7 = r^{-1}(Call_r)$

Figure 7.9

Des points de programme (représentés par des entiers) sont introduits pour indiquer de façon précise les différentes étapes du calcul. Par exemple, le point 1 correspond à une étape de l'exécution juste avant l'appel de p(W), le point 2 correspond à une étape de l'exécution juste après l'appel de p(W). Par ailleurs, une variable ensembliste est associée à chaque variable et chaque point de programme pour représenter l'évolution du calcul. De plus, pour tout prédicat p, une variable $Call_p$ est introduite et représente (une approximation de) l'ensemble des appels associé à p. L'algorithme de mise sous forme explicite est reconsidéré du point de vue de l'implémentation. Les résultats obtenus soulignent que les contraintes sont plus faciles à résoudre lorsque l'on considère une sémantique bottom-up, et que des techniques et stratégies diverses bien employées permettent d'obtenir des temps de calcul raisonnables.

3 Discussion

Ayant effectué un tour d'horizon sur les types et les contraintes ensemblistes, il nous est possible de tirer maintenant quelques enseignements et conclusions.

Premièrement, [Frühwirth et al. 91] et [Heintze et Jaffar 92a] nous apprennent que l'approche de point fixe est équivalente à l'approche dénotationnelle (pour la sémantique bottom-up). En effet, la formulation des types est équivalente via les contraintes ensemblistes (méthode de [Heintze et Jaffar 90]) et via l'opérateur T_p, cette formulation étant basée sur l'ignorance des dépendances inter-variables. De même, la formulation des types est équivalente via les contraintes ensemblistes où $\uplus$ remplace $\cup$ et via l'opérateur T_p^*, cette formulation étant basée sur l'ignorance des dépendances inter-arguments. Toutefois, l'équivalence de tient pas pour une classe de programmes dégénérés. Par ailleurs, [Heintze 92] affirme que pour l'approximation considérée (l'ignorance des dépendances inter-

variables) l'approche opérationnelle est moins efficace que l'approche dénotationnelle. Par contre, l'approche opérationnelle (basée sur d'autres approximations) peut s'avérer plus efficace que l'approche dénotationnelle. En résumé, suivant [Heintze 92], l'approche dénotationnelle est plus précise que l'approche opérationnelle pour certains programmes grâce à son aptitude à raisonner sur les structures de termes et l'approche opérationnelle est plus précise que l'approche dénotationnelle pour certains programmes grâce à son aptitude à raisonner sur les dépendances inter-variables.

Deuxièmement, les contraintes ensemblistes sont un bon moyen de représenter les types. D'une part, le formalisme des contraintes ensemblistes permet de définir différentes classes de types (tout dépend des opérations ensemblistes utilisées). D'autre part, le polymorphisme est supporté naturellement. Par exemple, pour le polymorphisme paramétrique,

$$\text{List}(\alpha) = \text{nil} \mid \text{cons}(\alpha,\text{List}(\alpha))$$

est une définition de type générique dont le paramètre est la variable de type α. Cette définition est équivalente à l'équation ensembliste suivante :

$$\mathcal{L} = \text{nil} \cup \text{cons}(\mathcal{X},\mathcal{L})$$

Si Int est un type défini par ailleurs, alors List(Int) est une instance de List(α). Pour obtenir la même instance sous forme ensembliste, il suffit de former un système comportant la précédente équation et une équation donnant la définition de $\mathcal{X}$. Pour le polymorphisme inclusif, c'est encore plus immédiat. Les deux exemples suivants illustrent la double possibilité de définir des sous-types :

$$\mathcal{X} \subseteq \mathcal{Y}$$

est la première (par inclusion), et

$$\mathcal{X} = \mathcal{Y} \cup \mathcal{Z}$$

est la seconde (par union).

Pour conclure, désirant développer un système d'inférence de types s'adaptant à notre modèle d'interprétation abstraite (voir chapitre 5 et 6), il nous faut essayer de tirer le meilleur profit des enseignements ci-dessus. Notre modèle est basé sur une approche opérationnelle et les contraintes ensemblistes sont un bon formalisme pour le typage. Y a-t-il possibilité de conjuguer cette double constatations ? La réponse est positive, d'autant plus que les contraintes ensemblistes s'intègrent parfaitement et naturellement dans le cadre de la programmation logique avec contraintes.

Pour développer un système d'inférence de type pour CLP($\mathcal{FT}$) basée sur les contraintes ensemblistes, il suffit dans un premier temps d'étendre le domaine, i.e., introduire les contraintes ensemblistes, puis dans un second temps d'abstraire le calcul, i.e., introduire une opération de widening. Le domaine concret est l'ensemble $\mathcal{FT}$ (l'ensemble des arbres finis), le domaine abstrait est l'ensemble $\mathcal{SC}$ (l'ensemble des contraintes ensemblistes). En donnant dans CLP($\mathcal{FT}$+$\mathcal{SC}$) une interprétation naturelle au domaine de contrainte $\mathcal{SC}$, l'implication (logique) devient une relation d'approximation naturelle. Information concrète et information abstraite sont associées, ce qui permet de raisonner à la fois sur les structures de termes et les dépendances inter-variables. ([Heintze et Jaffar 92] ont sensiblement la même approche). Néanmoins, la définition de l'opération de widening reste essentielle quant à l'efficacité et la précision des résultats. Nous proposons ce modèle d'inférence de types au chapitre suivant.

Chapitre 8

Inférence de Types avec CLP($\mathcal{FT}$+$\mathcal{SC}$)

Nous proposons dans ce chapitre une inférence de types pour CLP($\mathcal{FT}$). Dans ce but, nous commençons par étendre le domaine de CLP($\mathcal{FT}$) en y intégrant des contraintes ensemblistes. Le langage obtenu par cette intégration est noté CLP($\mathcal{FT}$+$\mathcal{SC}$). Les contraintes de ce nouveau langage portent à la fois sur les termes (ou arbres finis) et sur les ensembles. L'intéret de cette combinaison est que les contraintes sur les termes permettent de coder les dépendances entre les variables et que les contraintes ensemblistes permettent de coder les structures récursives et non déterministes. La classe des systèmes de contraintes ensemblistes manipulées par CLP($\mathcal{FT}$+$\mathcal{SC}$) est celle des systèmes d'équations utilisant l'union et l'intersection. Cette classe permet de coder les langages réguliers. Lorsque les systèmes d'équations ensemblistes sont sous forme résolue, il est possible d'appliquer la projection et le complémentaire.

Les contraintes de CLP($\mathcal{FT}$+$\mathcal{SC}$) sont de la forme $c = (\mathcal{ST},\mathcal{SS})$ où $\mathcal{ST}$ représente un système d'équations sur les termes et $\mathcal{SS}$ représente un système d'équations ensemblistes. Nous montrons que lorsque de telles contraintes sont placées sous forme résolue, la satisfiabilité et l'implication sont deux propriétés décidables. Ces décisions sont essentielles à l'utilisation de CLP($\mathcal{FT}$+$\mathcal{SC}$) comme base de calcul abstrait. De fait, en utilisant la tabulation et en introduisant des opérateurs de widening adaptés, nous obtenons un algorithme d'inférence de types qui représente une instance du modèle générique d'interprétation abstraite proposé dans la partie III. La phase d'extension du domaine (voir chapitre 5) correspond à la conception du langage CLP($\mathcal{FT}$+$\mathcal{SC}$)) et la phase d'abstraction du calcul (voir chapitre 6) correspond à l'introduction de la tabulation et d'opérateurs de widening.

Dans un premier temps (section 1), nous discutons des difficultés de gérer l'inférence de types avec CLP($\mathcal{FT}$) ou CLP($\mathcal{SC}$). Puis, nous étudions (section 2) les systèmes d'équations ensemblistes, en nous attardant sur la forme résolue des systèmes et sur la définition d'opérateurs de widening. Ensuite (section 3), nous introduisons le langage CLP($\mathcal{FT}$+$\mathcal{SC}$) et montrons que les problèmes de satisfiabilité et d'implication de contraintes résolues sont décidables pour ce langage. Nous illustrons alors le système d'inférence de types obtenu avec quelques exemples représentatifs. Nous terminons (section 4) avec une discussion.

1 CLP($\mathcal{FT}$) et CLP($\mathcal{SC}$)

1.1 CLP($\mathcal{FT}$)

CLP($\mathcal{FT}$) est un CLP-langage (Σ,$\mathcal{L}$,$\mathcal{A}$) tel que

- Σ = (S,$\mathcal{F}$,C) où
  - S = {term}
  - $\mathcal{F}$ désigne un ensemble fini de symboles de fonction
  - C = {=,⊥,⊤}
- $\mathcal{L}$ désigne l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},C)$ clos par
  - conjonction
  - quantification existentielle
- $\mathcal{A}$ est tel que
  - l'interprétation de S est donnée par $\mathcal{A}$(term) = $\mathcal{T}erm(\mathcal{F})$
  - l'interprétation des symboles de $\mathcal{F}$ est libre.

Après avoir rappelé la définition de CLP($\mathcal{FT}$), nous proposons un petit programme pour illustrer la nécessité d'intégrer de nouvelles contraintes au sein de CLP($\mathcal{FT}$) afin d'obtenir des analyses de types précises.

Soit le programme P :

int(X) ← X=0 ◊ .
int(X) ← X=s(X') ◊ int(X').
int_list(L) ← L=nil ◊ .
int_list(L) ← L=cons(X,L') ◊ int(X), int_list(L').

Soit le but G :

← ◊ int_list(L).

Des trois opérateurs de widening introduits au chapitre 6, aucun ne peut dériver la structure récursive du type de la variable libre L. Ce ne sont pas les opérateurs qui sont mal définis mais c'est le domaine de contraintes qui est mal adapté. Il

est donc nécessaire d'étendre ce domaine afin de pouvoir coder la structure récursive et non déterministe des termes. Les contraintes ensemblistes s'y prêtent très bien.

1.2 CLP($\mathcal{SC}$)

CLP($\mathcal{SC}$) est un CLP-langage (Σ,$\mathcal{L}$,$\mathcal{A}$) tel que

- Σ = (S,$\mathcal{G}$,C) où
  - S = {term_set}
  - $\mathcal{G} = \mathcal{F} \cup \{ f^{-1}_{(i)} \mid f \in \mathcal{F}_n \text{ et } 1 \leq i \leq n\} \cup \{\bot_{sc},\top_{sc},\cup,\cap,\bar{}\ \}$ où $\mathcal{F}$ est un ensemble fini de symboles de fonction
  - $C = \{=,\bot,\top\} \cup \{\subseteq,\not\subseteq\}$
- $\mathcal{L}$ désigne l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},C)$ clos par
  - conjonction
  - quantification existentielle
- $\mathcal{A}$ désigne la Σ-algèbre donnée au chapitre 7, section 1.1.

Il est possible de définir la relation d'approximation suivante entre CLP($\mathcal{FT}$) et CLP($\mathcal{SC}$) : soient c^{Δ} une contrainte définie sur CLP($\mathcal{FT}$) et c^{∇} une contrainte définie sur CLP($\mathcal{SC}$), $\xi(c^{\Delta},c^{\nabla})$ ssi pour toute solution θ^{Δ} de c^{Δ}, pour toute solution θ^{∇} de c^{∇} et pour toute variable X, on a $\theta^{\Delta}(X) \subseteq \theta^{\nabla}(X)$. On peut ainsi, par exemple, approcher le programme de la section précédente par le programme suivant :

```
int(X) ← X⊇0 ◊ .
int(X) ← X⊇s(X') ◊ int(X').
int_list(L) ← L⊇nil ◊ .
int_list(L) ← L⊇cons(X,L') ◊ int(X), int_list(L').
```

Quoiqu'il en soit, pour toute analyse dans CLP($\mathcal{SC}$), il est nécessaire de pouvoir décider du vide[†] (la satisfiabilité d'une contrainte) et de l'implication ou de l'équivalence de deux contraintes. Dans le premier cas, de nombreux résultats ont été obtenus (voir le chapitre 7, section 1.2) mais dans le second cas, le problème est ouvert. Or, pouvoir décider de l'implication ou de l'équivalence de deux contraintes ensemblistes est le fondement de la tabulation. Il n'est donc pas possible d'utiliser toute la puissance des contraintes ensemblistes dans CLP($\mathcal{SC}$) si on désire jouir de la tabulation. Par ailleurs, il n'est pas possible de coder les dépendances entre variables avec les contraintes ensemblistes.

[†] Il n'est pas réellement indispensable de pouvoir décider du vide si CLP($\mathcal{SC}$) désigne une sémantique abstraite car dans ce cas le calcul reste correct.

1.3 $\mathcal{FT}$+$\mathcal{SC}$

L'étude de CLP($\mathcal{FT}$) nous amène à considérer des contraintes ensemblistes pour pouvoir coder la structure récursive et non déterministe des termes. L'étude de CLP($\mathcal{SC}$) nous amène à considérer une classe limitée de contraintes ensemblistes. Par ailleurs, au chapitre 6, nous avons discuté du bien-fondé d'une approche basée sur une extension du domaine concret. C'est pourquoi, dans le but de concevoir un système de types sémantiques pour CLP($\mathcal{FT}$), nous décidons d'intégrer les contraintes ensemblistes au domaine de CLP($\mathcal{FT}$). Le langage obtenu, appelé CLP($\mathcal{FT}$+$\mathcal{SC}$), manipule essentiellement des systèmes d'équations :

- sur les termes
- sur les ensembles

Pour CLP($\mathcal{FT}$+$\mathcal{SC}$), on va alors utiliser des contraintes de la forme ($\mathcal{ST}$,$\mathcal{SS}$) où $\mathcal{ST}$ et $\mathcal{SS}$ représentent respectivement un système d'équations sur les termes et un système d'équations ensemblistes.

2 Systèmes d'équations ensemblistes

2.1 Définition

Dans cette partie, la Σ-algèbre $\mathcal{A}$ définie pour CLP($\mathcal{SC}$) est considérée. Nous présentons la classe de contraintes ensemblistes que nous allons utiliser pour définir CLP($\mathcal{FT}$+$\mathcal{SC}$). Celle-ci est limitée à l'utilisation de l'union et de l'intersection comme opérations ensemblistes. Les définitions de ce chapitre sont donc différentes de celles du chapitre précédent. On peut comparer par exemple la définition 7.1 avec la définition 8.1.

Définition 8.1

Une expression ensembliste désigne l'une des expressions suivantes :

$a \mid f(X_1,\ldots,X_n) \mid \bot_{sc} \mid \top_{sc} \mid ex \cup ex' \mid ex \cap ex'$

où $a \in \mathcal{F}_0$, $f \in \mathcal{F}_n$, $X_i \in \mathcal{V}$ avec $1 \leq i \leq n$, et ex, ex' désignent des expressions ensemblistes.

Définition 8.2

Un système d'équations ensemblistes $\mathcal{S}$ est un ensemble fini d'équations ensemblistes (i.e, un ensemble d'éléments de la forme X = ex où ex désigne une expression ensembliste) tel qu'il n'existe pas deux équations dans $\mathcal{S}$ de même partie gauche.

Dans tout système d'équations $\mathcal{S}$, n'apparaissent que des variables libres ou liées existentiellement. On utilisera donc les ensembles $Var(\mathcal{S})$, $Var_{free}(\mathcal{S})$ et $Var_{\exists}(\mathcal{S})$. Un système $\mathcal{S}$ est implicitement interprété par : $\exists V\ \mathcal{S}$ où V représente $Var_{\exists}(\mathcal{S})$ et $\mathcal{S}$ représente une conjonction d'équations ensemblistes. De ce fait, l'ensemble vide représente $\top$.

Remarque 8.3 (importante)

Pour tout système $\mathcal{S}$, on considère que toute variable $\mathcal{X}$ de $Ran(\mathcal{S})$ n'apparaissant pas dans $Dom(\mathcal{S})$ est interprétée par $\top_{sc}$.

La remarque précédente signifie que toute équation $\mathcal{X} = \top_{sc}$ d'un système peut être sous-entendue. Pour un système donné, $\mathcal{S}(\mathcal{X})$ désigne par analogie avec les substitutions l'expression ex si $\mathcal{X}$ = ex est une équation de $\mathcal{S}$ et l'expression $\top_{sc}$ dans le cas contraire (voir la remarque précédente). $Dom(\mathcal{S})$ et $Ran(\mathcal{S})$ représentent respectivement l'ensemble des variables qui apparaissent en partie gauche et en partie droite d'une équation de $\mathcal{S}$.

Propriété 8.4

Toute système $\mathcal{S}$ possède une unique solution, notée $sol(\mathcal{S})$.

Preuve

La preuve est une conséquence de celle donnée par [Uribe 92b] concernant les systèmes de contraintes sans cycles. Intuitivement, un cycle dans un système signifie qu'une variable peut être définie en fonction d'elle-même et en dehors de la portée d'un symbole de fonction. Par exemple, $\mathcal{X} = f(\mathcal{X})$ ne crée aucun cycle mais $\mathcal{X}=\mathcal{Y}\cup a \wedge \mathcal{Y}=\mathcal{X}\cup b$ en crée un. Les systèmes que nous utilisons sont, par définition, sans cycles. Par ailleurs, les variables apparaissant uniquement en partie droite d'un système (variables appelées paramètres par [Uribe 92b]) ont une assignation fixée. Le résultat de [Uribe 92b] (théorème 2.18) établit qu'il existe une seule solution à un système sans cycles, une fois qu'une assignation a été fixée aux paramètres du système. ❑

Deux systèmes $\mathcal{S}$ et $\mathcal{S}'$ sont équivalents ssi ils admettent la même solution. Pour un système $\mathcal{S}$ donné, on note $sol(\mathcal{S})(ex)$ l'interprétation de l'expression ex par rapport à $sol(\mathcal{S})$.

2.2 Forme résolue

Nous cherchons à calculer une forme résolue pour tout système $\mathcal{S}$. Pour cela, nous utilisons en particulier les résultats de [Heintze et Jaffar 90] et [Uribe 92a,92b].

Définition 8.5

Un système $\mathcal{S}$ est élémentaire ssi $Var_{free}(\mathcal{S}) = \{\mathcal{X}\}$. On note alors $\mathcal{S}$ par $\mathcal{S}_{\mathcal{X}}$.

Définition 8.6

Une expression ensembliste ex est résolue ssi soit ex désigne $\perp_{sc}$ soit ex désigne une expression de la forme $s_1 \cup ... \cup s_m$ telle que chaque sous-expression s_j de ex est une constante ou un terme de la forme $f(\mathcal{X}_1,...,\mathcal{X}_n)$.

Définition 8.7

Un système élémentaire $\mathcal{S}_\mathcal{X}$ est (sous forme) résolu ssi $\mathcal{S}_\mathcal{X}$ est un ensemble fini d'équations de la forme $\mathcal{Y} = ex$ où ex désigne une expression résolue. De plus, si ex est de la forme $s_1 \cup ... \cup s_m$ alors il n'existe pas de couple (i,j) d'entiers distincts compris entre 1 et m tel que $sol(\mathcal{S})(s_i) \subseteq sol(\mathcal{S})(s_j)$.

Définition 8.8

Un système $\mathcal{S}$ est (sous forme) résolu ssi $\mathcal{S} = \mathcal{S}_\mathcal{X} \cup ... \cup \mathcal{S}_\mathcal{Z}$ désigne une union de systèmes élémentaires résolus tel que $Var_{free}(\mathcal{S}) = \{\mathcal{X},...,\mathcal{Z}\}$.

Soit $\mathcal{S}$ un système, on note $res(\mathcal{S})$ le système obtenu après l'algorithme de mise sous forme résolue suivant. La forme résolue est obtenue après cinq séries de transformations. Les trois premières étapes sont essentiellement des adaptations de [Heintze et Jaffar 90].

Algorithme de résolution

Etape 1 : mise sous forme normale disjonctive

(a) : placer chaque expression ensembliste apparaissant en partie droite d'une équation de $\mathcal{S}$ sous forme normale disjonctive.

Etape 2 : éliminer les $\cap$

On considère pour les transformations suivantes l'ensemble des expressions ensemblistes apparaissant en partie droite d'une équation de $\mathcal{S}$. On renomme initialement chaque variable $\mathcal{X}$ par $V_{\{\mathcal{X}\}}$.

(b) : appliquer (tant que possible) les transformations suivantes :

- remplacer $s \cap \perp_{sc}$ par $\perp_{sc}$, $s \cap \top_{sc}$ par s
- remplacer $f(V_{S_1},...,V_{S_n}) \cap g(V_{T_1},...,V_{T_m})$ par $\perp_{sc}$
- remplacer $f(V_{S_1},...,V_{S_n}) \cap f(V_{T_1},...,V_{T_n})$ par $f(V_{S_1 \cup T_1},...,V_{S_n \cup T_n})$.

  Si $V_{S_i \cup T_i}$ n'apparait pas déjà dans le système, alors il s'agit d'une nouvelle variable et dans ce cas on ajoute au système l'équation suivante : $V_{S_i \cup T_i} = \mathcal{S}(V_{S_i}) \cap \mathcal{S}(V_{T_i})$ et on place l'expression ensembliste

apparaissant en partie droite de la nouvelle équation sous forme normale disjonctive.

Après l'application de cet algorithme, on peut renommer chaque variable $V_{\{X\}}$ par X. Par contre, les variables V_S telles que card(S) ≥ 2 n'on pas besoin d'être renommées. Celles-ci sont quantifiées existentiellement. L'étape 2 reprend l'essentiel de l'algorithme "simplify" de [Heintze et Jaffar 90]. Notons simplement que des 5 transformations proposées par [Heintze et Jaffar 90], seules 3 ont été conservées (les transformations 3, 4 et 5) car dans nos systèmes, une "un-nested" occurence d'une variable, i.e., l'occurence d'une variable dans une expression hors de la portée d'un symbole de fonction, n'est pas permise.

Etape 3 : simplifier les $\top_{sc}$ et $\bot_{sc}$

(c) : supprimer toutes les équations $X = \mathcal{S}(X)$ telles que $\top_{sc}$ apparaît dans $\mathcal{S}(X)$.

(d) : pour toute variable X de Dom($\mathcal{S}$) telle que sol($\mathcal{S}$)(X) = $\emptyset$:

- remplacer $X = \mathcal{S}(X)$ par $X = \bot_{sc}$
- supprimer tout terme de $\mathcal{S}$ où apparaît X[†]

L'étape 3 permet d'obtenir un système tel que toute occurence de $\top_{sc}$ a disparu et tel que toute occurence de $\bot_{sc}$ est nécessairement liée à une équation du type $X = \bot_{sc}$. Pour déterminer les variables de $\mathcal{S}$ qui sont telles que sol($\mathcal{S}$)(X) = $\emptyset$, il est possible d'utiliser l'algorithme de [Heintze et Jaffar 90]. Intuitivement, l'algorithme marque successivement toutes les variables X telles que sol($\mathcal{S}$)(X) ≠ $\emptyset$. Initialement, seules sont marquées les variables X de $\mathcal{S}$ n'appartenant pas à Dom($\mathcal{S}$) car elles sont interprétées par $X = \top_{sc}$. Voici l'algorithme :

On applique (tant que cela est possible) un changement d'état (de non marqué à marqué) pour une variable X de Dom($\mathcal{S}$) ssi il existe une constante ou un terme $f(X_1,\ldots,X_n)$ dans $\mathcal{S}(X)$ tel que $X_1,\ldots,X_n$ soient marquées.

Etape 4 : créer les systèmes élémentaires

(e) : coder $\mathcal{S}$ sous la forme $\mathcal{S} = \mathcal{S}_X \cup \ldots \cup \mathcal{S}_Z$ où :

- $Var_{free}(\mathcal{S}) = \{X,\ldots,Z\}$

[†] pour être rigoureux, toute suppression d'un terme correspond en fait au remplacement de ce terme par $\bot_{sc}$ suivi d'éventuelles simplifications de la forme $\bot_{sc} \cup s = s$.

◦ pour toute variable libre $\mathcal{Y}$ de $\mathcal{S}$, $\mathcal{S}_{\mathcal{Y}}$ représente le système élémentaire :

$$\mathcal{S}_{\mathcal{Y}} = \{\mathcal{W}=\text{ex} \in \mathcal{S} : \mathcal{W} \in \text{Desc}^*(\mathcal{S},\mathcal{Y})\}$$

$\text{Desc}(\mathcal{S},\mathcal{X})$ représente l'ensemble des variables $\mathcal{W}$ qui apparaissent dans $\mathcal{S}(\mathcal{X})$. $\text{Desc}^*(\mathcal{S},\mathcal{X})$ représente la clôture réflexive et transitive de $\text{Desc}(\mathcal{S},\mathcal{X})$. On peut supposer, sans perte de généralité, que si $\mathcal{X}$ et $\mathcal{Y}$ sont deux variables libres distinctes de $\mathcal{S}$ alors :

$$\text{Var}_{\exists}(\mathcal{X}) \cap \text{Var}_{\exists}(\mathcal{Y}) = \emptyset$$

La phase (e) consiste à ne faire dépendre toute variable libre $\mathcal{X}$ de $\mathcal{S}$ que de variables existentielles propres à $\mathcal{X}$. Il est ainsi plus facile de manipuler le système $\mathcal{S}$ car tous les systèmes élémentaires sont indépendants.

Etape 5 : réduire les systèmes élémentaires

Pour chaque système élémentaire $\mathcal{S}_{\mathcal{X}}$ de $\mathcal{S}$, on considère les transformations suivantes par rapport à l'ensemble des expressions ensemblistes apparaissant en partie droite d'une équation de $\mathcal{S}_{\mathcal{X}}$. On renomme initialement chaque variable $\mathcal{Y}$ de $\mathcal{S}_{\mathcal{X}}$ par $V_{\{\mathcal{Y}\}}$.

(f) : appliquer (tant que possible) les transformations suivantes sur $\mathcal{S}_{\mathcal{X}}$:

- remplacer $s_1 \cup s_2$ par s_2 si $\text{sol}(\mathcal{S})(s_1) \subseteq \text{sol}(\mathcal{S})(s_2)$

- remplacer V_S par V_T partout dans $\mathcal{S}_{\mathcal{X}}$ et ne garder qu'une seule équation ayant en partie droite V_T ssi
  - $S \neq \{\mathcal{X}\}$
  - $\text{sol}(\mathcal{S}_{\mathcal{X}})(V_S) = \text{sol}(\mathcal{S}_{\mathcal{X}})(V_T)$

- remplacer $f(V_{S_1},\ldots,V_{S_n}) \cup f(V_{T_1},\ldots,V_{T_n})$ par $f(V_{S_1 \cup T_1},\ldots,V_{S_n \cup T_n})$ si n=0 ou si il existe un entier i compris entre 1 et n tel que pour tout entier j distinct de i et compris entre 1 et n, on ait : $V_{S_j} = V_{T_j}$.

  Si $V_{S_i \cup T_i}$ n'apparait pas déjà dans le système, alors il s'agit d'une nouvelle variable et dans ce cas on ajoute au système l'équation suivante : $V_{S_i \cup T_i} = \mathcal{S}(V_{S_i}) \cup \mathcal{S}(V_{T_i})$.

Après l'application de cet algorithme, on renomme la variable $V_{\{\mathcal{X}\}}$ par $\mathcal{X}$. La variable $V_{\{\mathcal{X}\}}$ est encore présente dans $\mathcal{S}_{\mathcal{X}}$ car les différentes transformations ne peuvent l'effacer. La première transformation de la phase (f) est essentielle car elle permet d'éviter de considérer une infinité de systèmes équivalents. Les autres transformations permettent simplement de simplifier le système. Il est possible de décider si $\text{sol}(\mathcal{S})(s_i) = \text{sol}(\mathcal{S})(s_j)$ ou si $\text{sol}(\mathcal{S})(s_i) \subseteq \text{sol}(\mathcal{S})(s_j)$ car

chaque terme s du système $\mathcal{S}$ correspond à une grammaire de termes (ou un automate d'arbres). Nous discutons de cet aspect à la section 2.4.

Exemple

Considérons un exemple de mise sous forme résolue. Soit le système $\mathcal{S}$ suivant :

$\mathcal{S}$ = { $\mathcal{T}$=g($\mathcal{T}$) , $\mathcal{X}$=f($\mathcal{Y}$,$\mathcal{Z}$)∪a , $\mathcal{Y}$=g($\mathcal{T}$)∪a∪b ,
$\mathcal{Z}$=f($\mathcal{Z}_1$,$\mathcal{Z}_1$)∩f($\mathcal{Z}_2$,$\mathcal{Z}_3$) , $\mathcal{Z}_1$=a , $\mathcal{Z}_2$=a∪c , $\mathcal{Z}_3$=a }

tel que $Var_{\exists}(\mathcal{S}) = \{\mathcal{Z}_1,\mathcal{Z}_2,\mathcal{Z}_3\}$

La première étape ne change rien à $\mathcal{S}$ car $\mathcal{S}$ est déja sous forme normale. Après la seconde étape, on obtient :

$\mathcal{S}$ = { $\mathcal{T}$=g($\mathcal{T}$) , $\mathcal{X}$=f($\mathcal{Y}$,$\mathcal{Z}$)∪a , $\mathcal{Y}$=g($\mathcal{T}$)∪a∪b ,
$\mathcal{Z}$=f($\mathcal{Z}_4$,$\mathcal{Z}_5$) , $\mathcal{Z}_4$=a , $\mathcal{Z}_5$=a }

tel que $Var_{\exists}(\mathcal{S}) = \{\mathcal{Z}_4,\mathcal{Z}_5\}$

Aprés la troisième étape, on obtient :

$\mathcal{S}$ = { $\mathcal{T}$=$\perp_{sc}$, $\mathcal{X}$=f($\mathcal{Y}$,$\mathcal{Z}$)∪a , $\mathcal{Y}$=a∪b ,
$\mathcal{Z}$=f($\mathcal{Z}_4$,$\mathcal{Z}_5$) , $\mathcal{Z}_4$=a , $\mathcal{Z}_5$=a }

tel que $Var_{\exists}(\mathcal{S}) = \{\mathcal{Z}_4,\mathcal{Z}_5\}$

A la quatrième étape, on code $\mathcal{S}$ sous la forme d'une union de systèmes élémentaires.

$\mathcal{S} = \mathcal{S}_{\mathcal{T}} \cup \mathcal{S}_{\mathcal{X}} \cup \mathcal{S}_{\mathcal{Y}} \cup \mathcal{S}_{\mathcal{Z}}$ avec
- $\mathcal{S}_{\mathcal{T}}$ = { $\mathcal{V}$=$\perp_{sc}$ }
- $\mathcal{S}_{\mathcal{X}}$ = { $\mathcal{X}$=f($\mathcal{W}_1$,$\mathcal{W}_2$)∪a , $\mathcal{W}_1$=a∪b , $\mathcal{W}_2$=f($\mathcal{W}_3$,$\mathcal{W}_4$) , $\mathcal{W}_3$=a , $\mathcal{W}_4$=a }
- $\mathcal{S}_{\mathcal{Y}}$ = { $\mathcal{Y}$=a∪b}
- $\mathcal{S}_{\mathcal{Z}}$ = { $\mathcal{Z}$=f($\mathcal{W}_5$,$\mathcal{W}_6$) , $\mathcal{W}_5$=a , $\mathcal{W}_6$=a }

Il n'est plus nécessaire de préciser les variables existentielles car elles sont implicites maintenant. On les notera généralement par la lettre $\mathcal{W}$ indicée. La cinquième étape permet de réduire $\mathcal{S}$, on obtient alors :

res($\mathcal{S}$) = $\mathcal{S}_{\mathcal{T}} \cup \mathcal{S}_{\mathcal{X}} \cup \mathcal{S}_{\mathcal{Y}} \cup \mathcal{S}_{\mathcal{Z}}$ avec
- $\mathcal{S}_{\mathcal{T}}$ = { $\mathcal{V}$=$\perp_{sc}$ }
- $\mathcal{S}_{\mathcal{X}}$ = { $\mathcal{X}$=f($\mathcal{W}_1$,$\mathcal{W}_2$)∪a , $\mathcal{W}_1$=a∪b , $\mathcal{W}_2$=f($\mathcal{W}_3$,$\mathcal{W}_3$) , $\mathcal{W}_3$=a }
- $\mathcal{S}_{\mathcal{Y}}$ = { $\mathcal{Y}$=a∪b}
- $\mathcal{S}_{\mathcal{Z}}$ = { $\mathcal{Z}$=f($\mathcal{W}_4$,$\mathcal{W}_4$) , $\mathcal{W}_4$=a } ❑

Notons qu'il est possible de représenter visuellement chaque système élémentaire par un graphe de types ("type graph" [Janssens et Bruynooghe 92]). Nous utiliserons cette forme chaque fois que cela pourra soutenir l'intuition.

Propriété 8.9
res($\mathcal{S}$) est équivalent à $\mathcal{S}$.

Preuve
Toutes les transformations sont correctes. Pour les phases (a) et (b), c'est immédiat. Pour la phase (c), cela résulte de la remarque 8.3. Pour la phase (d), c'est immédiat. Pour la phase (e), il est clair que pour toute variable libre $\mathcal{X}$ de $\mathcal{S}$, on a $sol(\mathcal{S})(\mathcal{X}) = sol(\mathcal{S}_{\mathcal{X}})(\mathcal{X})$ car toute équation $\mathcal{Y}$=ex de $\mathcal{S}$ qui n'apparaît pas dans $Desc^*(\mathcal{X})$ ne concerne pas l'interprétation de $\mathcal{X}$. Pour la phase (f), la correction des deux premières transformations est immédiate. Quant à la troisième, il s'agit d'une simple application de la propriété suivante sur le produit cartésien : $A\times B \cup A\times C = A\times(B\cup C)$ ❑

Propriété 8.10
Le calcul de res($\mathcal{S}$) est fini.

Preuve
Immédiat pour toutes les phases sauf les phases (b) et (f). Pour la phase (b) ([Heintze et Jaffar 90]), il suffit de constater que le nombre de nouvelles variables pouvant être générées est borné par le nombre de parties de l'ensemble Var($\mathcal{S}$). Pour la phase (f), remarquons d'abord que la deuxième transformation peut être remplacée de façon équivalente par :

- remplacer V_S et V_T par $V_{S\cup T}$ partout dans $\mathcal{S}_{\mathcal{X}}$ et ne garder qu'une seule équation ayant en partie droite $V_{S\cup T}$ ssi $sol(\mathcal{S}_{\mathcal{X}})(V_S) = sol(\mathcal{S}_{\mathcal{X}})(V_T)$.

La seule différence est la gestion du nom des variables (en particulier, on ne garde pas la trace de $V_{\{\mathcal{X}\}}$ mais le problème de la terminaison est le même). On peut supposer également que pour toute variable $\mathcal{Y}$ du système $\mathcal{S}_{\mathcal{X}}$ telle que $\mathcal{S}_{\mathcal{X}}(\mathcal{Y}) = T_{sc}$, l'équation $\mathcal{Y}$=T_{sc} apparaît explicitement dans $\mathcal{S}_{\mathcal{X}}$.

Soit $\mathcal{S}_{\mathcal{X}}$ un système élémentaire, on note Syst($\mathcal{S}_{\mathcal{X}}$) l'ensemble des systèmes $\mathcal{T}$ tels que chaque variable de $\mathcal{T}$ est de la forme V_S où S désigne un ensemble de variables de $\mathcal{S}_{\mathcal{X}}$. Pour tout système $\mathcal{T}$ de Syst($\mathcal{S}_{\mathcal{X}}$), on définit les ensembles suivants :

$nb\_var(\mathcal{T}) = card(Var(\mathcal{T}))$
$var\_max(\mathcal{T}) = \{V_S \in Var(\mathcal{T}) : \neg (\exists\, V_T \in Var(\mathcal{T}) : S \subset T)\}$

On définit aussi la relation d'ordre suivante sur Syst($\mathcal{S}_{\mathcal{X}}$) :

$\mathcal{T} < \mathcal{T}$ ssi
var_max($\mathcal{T}$) $\subset$ var_max($\mathcal{T}$) ou
var_max($\mathcal{T}$) = var_max($\mathcal{T}$) et nb_var($\mathcal{T}$) $\supset$ nb_var($\mathcal{T}$)

ainsi que la relation d'équivalence suivante :

$\mathcal{T} \equiv \mathcal{T}$ ssi var_max($\mathcal{T}$) = var_max($\mathcal{T}$) et nb_var($\mathcal{T}$) = nb_var($\mathcal{T}$)

On montre facilement que Syst($\mathcal{S}_X$) vérifie la condition de chaîne ascendante. A la phase (f), on manipule des systèmes de l'ensemble Syst($\mathcal{S}_X$). On montre que chaque transformation de la phase (f) consiste soit à passer d'un système $\mathcal{T}$ à un système $\mathcal{T}$ tel que $\mathcal{T} < \mathcal{T}$ soit à passer d'un système $\mathcal{T}$ à un système $\mathcal{T}$ tel que $\mathcal{T} \equiv \mathcal{T}$'. Le nombre de transformations successives préservant l'équivalence d'un système est fini car initialement le nombre de sous-expressions est fini. On en déduit que la phase (f) termine. ❑

Il est à remarquer qu'une forme résolue n'est pas une forme normale. Par exemple :

$\mathcal{S}_X$ = { X=f($\mathcal{W}_1$,$\mathcal{W}_2$)$\cup$f($\mathcal{W}_3$,$\mathcal{W}_4$) , $\mathcal{W}_1$=a$\cup$b , $\mathcal{W}_2$=c , $\mathcal{W}_3$=b , $\mathcal{W}_4$=c$\cup$d }

$\mathcal{T}_X$ = { X=f($\mathcal{W}_1$,$\mathcal{W}_2$)$\cup$f($\mathcal{W}_3$,$\mathcal{W}_4$) , $\mathcal{W}_1$=a , $\mathcal{W}_2$=c , $\mathcal{W}_3$=b , $\mathcal{W}_4$=c$\cup$d }

$\mathcal{U}_X$ = { X=f($\mathcal{W}_1$,$\mathcal{W}_2$)$\cup$f($\mathcal{W}_3$,$\mathcal{W}_4$) , $\mathcal{W}_1$=a$\cup$b , $\mathcal{W}_2$=c , $\mathcal{W}_3$=b , $\mathcal{W}_4$=d }

sont trois systèmes élémentaires résolus équivalents. Le problème de trouver une forme normale (une forme unique) est ouvert.

2.3 Projection et complémentaire

Les opérations de projection et de complémentation ne sont pas admises dans les expressions ensemblistes que nous manipulons. Toutefois, nous pouvons être amené à les utiliser pour transformer un système résolu.

Un système résolu $\mathcal{S}$ possède une bonne propriété pour la projection, à savoir, que tout terme f(X_1,...,X_n) est tel que sol($\mathcal{S}$)(X_i) $\neq \emptyset$. Ainsi, il n'y a aucune restriction pour appliquer cette opération ensembliste. Pour illustrer le problème, imaginons le système non résolu suivant :

$\mathcal{S}$ = { X=f($\mathcal{Y}$,Z) , $\mathcal{Y}$=$\perp_{sc}$, Z=c }

et la projection suivante : $f^{-1}_{(2)}(X)$. Une application incontrôlée de la projection fournit comme résultat {c} alors qu'une application correcte fournit l'ensemble vide.

Nous définissons maintenant cet opérateur, d'abord par rapport à un terme, puis par rapport à une expression ensembliste et enfin par rapport à une variable. On considère un symbole de fonction f et un entier i compris entre 1 et l'arité de f. Soit $s = f(\mathcal{X}_1,\ldots,\mathcal{X}_n)$ un terme ensembliste, $f_{(i)}^{-1}(s)$ désigne la variable ensembliste $\mathcal{X}_i$. Soit ex une expression ensembliste résolue, $f_{(i)}^{-1}(ex)$ retourne une expression ensembliste résolue :

si $ex = \bot_{sc}$ alors
    $f_{(i)}^{-1}(ex) = \bot_{sc}$
sinon -- $ex = s_1 \cup \ldots \cup s_m$
    $f_{(i)}^{-1}(ex) = \cup \{\mathcal{S}(f_{(i)}^{-1}(s_j)) : s_j$ est de la forme $f(\mathcal{X}_1,\ldots,\mathcal{X}_n)\}$
fin si

Soit $\mathcal{X}$ une variable ensembliste d'un système résolu $\mathcal{S}$, $f_{(i)}^{-1}(\mathcal{X})$ désigne $f_{(i)}^{-1}(\mathcal{S}(\mathcal{X}))$, i.e., une expression résolue.

Par exemple, pour $\mathcal{S} = \{ \mathcal{X}=f(\mathcal{X},\mathcal{Y})\cup 0 , \mathcal{Y}=a\cup b \}$,

- $f_{(1)}^{-1}(\mathcal{X}) = f(\mathcal{X},\mathcal{Y})\cup 0$
- $f_{(2)}^{-1}(\mathcal{X}) = a\cup b$

Nous définissons l'opération de complémentation par rapport à un système élémentaire résolu $\mathcal{S}_\mathcal{X}$. Pour définir l'opération de complémentation par rapport à un système $\mathcal{S}$, il est nécessaire d'utiliser un codage pour se ramener au cas d'un système élémentaire (voir la preuve de la propriété 8.35). Le complémentaire de $\mathcal{S}_\mathcal{X}$ est noté $\overline{\mathcal{S}}_\mathcal{X}$. A chaque variable $\mathcal{Y}$ de $\mathrm{Var}(\mathcal{S}_\mathcal{X})$ correspond une variable $\overline{\mathcal{Y}}$ de $\mathrm{Var}(\overline{\mathcal{S}}_\mathcal{X})$:

$$\overline{\mathcal{S}} = \{ \overline{\mathcal{Y}} = \overline{ex} : \mathcal{Y}=ex \in \mathcal{S}\}$$

Le problème de la complémentation d'un système se ramène donc au problème de la complémentation d'une expression ensembliste résolue ex :

si $ex = \top_{sc}$ alors
    $\overline{ex} = \bot_{sc}$
sinon si $ex = \bot_{sc}$ alors
    $\overline{ex} = \top_{sc}$
sinon -- $ex = s_1 \cup \ldots \cup s_m$
    $\overline{ex} = \overline{s}_1 \cap \ldots \cap \overline{s}_m$
fin si

Si une sous-expression s désigne un terme de la forme $f(\mathcal{X}_1,\ldots,\mathcal{X}_n)$ alors $\overline{s}$ désigne une expression de la forme $s_1 \cup \ldots \cup s_p$ où chaque terme s_j $(1\le j\le p)$ correspond soit à un terme $f(\top_{sc},\ldots,\overline{\mathcal{X}}_i,\ldots,\top_{sc})$ pour tout i compris entre 1 et n,

soit à un terme $g(T_{sc},\ldots,T_{sc})$ pour tout symbole de fonction $g \neq f$ appartenant à $\mathcal{F}$.

Par exemple, si $\mathcal{F}$ = {a,b,f,g} et si S_X = { X=f(W_1,W_1)∪a , W_1=b } alors

res($\overline{S}_X$) = { $\overline{X}$=f($\overline{W}_1$,T_{sc})∪f(T_{sc},$\overline{W}_1$)∪g(T_{sc})∪b , $\overline{W}_1$=f(T_{sc},T_{sc})∪g(T_{sc})∪a }

Par exemple, si $\mathcal{F}$ = {0,s} et si S_X = {X=s(X)∪0} alors

res($\overline{S}_X$) = { $\overline{X}$=$\perp_{sc}$ }

2.4 Grammaires de termes

Les systèmes de contraintes ensemblistes sont liés aux grammaires de termes et aux automates d'arbres. Aussi, nombre de résultats établis par rapport aux grammaires ou aux automates s'appliquent aux contraintes ensemblistes. Nous rappelons dans cette partie quelques résultats sur les grammaires de termes (et automates d'arbres). Pour une approche plus générale, consultez le livre de [Gécseg et Steinby 84]. Une bonne introduction, dont nous nous sommes inspirés, est donnée par ailleurs par [Heintze 92] (pages 166-172) et [Uribe 92b] (pages 25-30).

Etants donnés un ensemble $\mathcal{F}$ de symboles de fonction et un ensemble $\mathcal{N}$ de non terminaux, une grammaire de termes (régulière) est définie par un ensemble fini de productions de la forme nt ⇒ t où nt désigne un non-terminal et t désigne un élément de Term($\mathcal{F}$,$\mathcal{N}$). Etant donné une grammaire de termes G, on écrit $t_1 \Rightarrow_G t_2$ ssi nt ⇒ s est une production de G et t_2 peut être obtenu en remplaçant une occurence de nt dans t_1 par s. La cloture reflexive et transitive de $\Rightarrow_G$ est notée $\Rightarrow^*_G$. Le langage de termes, noté G(nt), correspondant à un non terminal nt de G désigne l'ensemble suivant :

G(nt) = {t : nt $\Rightarrow^*_G$ t et t ne contient aucun non terminal}

Les langages de termes générés par une grammaire de termes sont dits réguliers. La classe des langages réguliers, notée *Reg*, est exactement celle des langages (ou forêts) reconnu(e)s par les automates d'arbres ascendant (déterministe ou non déterministe) ou descendant (déterministe). Il y a de nombreux résultats énoncés et de nombreux algorithmes proposés concernant les grammaires de termes ou les automates d'arbres [Gécseg et Steinby 84].

Reg, est clos par union, intersection et différence (et donc par complémentaire)

Si G et G' sont deux grammaires de termes et nt et nt' deux non terminaux respectifs de G et G', alors les questions suivantes sont décidables :

- le problème de la finitude : est-ce que G(nt) est un langage fini ?
- le problème de l'appartenance : est-ce qu'un terme t appartient à G(nt) ?
- le problème du vide : est-ce que G(nt) représente l'ensemble vide ?
- le problème de l'équivalence : est-ce que G(nt) = G'(nt') ?
- le problème de l'inclusion : est-ce que G(nt) $\subseteq$ G'(nt') ?

En fait, la décidabilité du problème de l'inclusion implique la décidabilité des problèmes de l'appartenance, du vide et de l'équivalence.

Toute grammaire de termes peut être réécrite sous forme normale, c'est à dire sous une forme telle que chaque production soit de la forme $nt \Rightarrow f(nt_1,...,nt_n)$ où f est un symbole de fonction n-aire et nt_1, ..., nt_n des non terminaux. En fait, à toute grammaire de termes sous forme normale, on peut faire correspondre un système d'équations ensemblistes résolu en associant non terminaux et variables ensemblistes.

Soit un système résolu $\mathcal{S}$, si à une variable ensembliste $\mathcal{X}$ on associe un non terminal X alors à chaque équation de la forme $\mathcal{X} = s_1 \cup ... \cup s_m$, on fait correspondre m règles de productions de la forme $X \Rightarrow s_1$, ..., $X \Rightarrow s_m$, à chaque équation de la forme $\mathcal{X} = \bot_{sc}$ on fait correspondre un non terminal X sans règles de productions associées, et à chaque équation de la forme $\mathcal{X} = \top_{sc}$ on fait correspondre une régle de production de la forme X = f(X,...,X) pour chaque symbole de fonction. Soit G la grammaire de termes obtenue, on peut montrer par induction que pour toute variable ensembliste $\mathcal{X}$, on a $sol(\mathcal{S})(\mathcal{X})$ = G(X). De manière réciproque, on montre qu'à toute grammaire G correspond un système d'équations ensemblistes $\mathcal{S}$ tel que pour tout non terminal X, on ait G(X) = $sol(\mathcal{S})(\mathcal{X})$. Il suffit de se ramener à la forme normale de G. Ainsi, tout système d'équations ensemblistes définit un langage (ou une famille de langages) régulier et tout langage régulier peut être reconnu par un système d'équations ensemblistes. On peut donc utiliser, par rapport aux systèmes d'équations ensemblistes, tous les résultats énoncés ci-dessus.

2.5 Borne inférieure et borne supérieure

Par la suite, on considérera la relation d'ordre notée $\subseteq$ et définie ainsi (deux systèmes résolus $\mathcal{S}$ et $\mathcal{S}'$ étant donnés) :

$\mathcal{S} \subseteq \mathcal{S}'$ ssi $sol(\mathcal{S})(\mathcal{X}) \subseteq sol(\mathcal{S})(\mathcal{X})$, $\forall\, \mathcal{X} \in \mathcal{V}$

Notons qu'il est possible de décider si $\mathcal{S} \subseteq \mathcal{S}'$ puisque les systèmes d'équations sont des ensembles finis et que $sol(\mathcal{S})(\mathcal{X})$ et $sol(\mathcal{S})(\mathcal{X})$ représentent des langages réguliers.

Le plus grand minorant, noté glb($\mathcal{S}_1$,$\mathcal{S}_2$), de deux systèmes résolus $\mathcal{S}_1$ et $\mathcal{S}_2$ désigne le système résolu $\mathcal{S}$ tel que pour toute variable $\mathcal{X}$, on a :

$$sol(\mathcal{S})(\mathcal{X}) = sol(\mathcal{S}_1)(\mathcal{X}) \cap sol(\mathcal{S}_2)(\mathcal{X})$$

Comme $sol(\mathcal{S}_1)(\mathcal{X})$ et $sol(\mathcal{S}_2)(\mathcal{X})$ sont des langages réguliers, il est possible de calculer le langage régulier qui correspond à l'intersection de $sol(\mathcal{S}_1)(\mathcal{X})$ et $sol(\mathcal{S}_2)(\mathcal{X})$, puis de coder ce langage sous la forme d'un système élémentaire résolu $\mathcal{S}_{\mathcal{X}}$. Pratiquement, on peut calculer $\mathcal{S}$ (avant mise sous forme résolue) comme suit :

- $\mathcal{S} = \{\mathcal{X} = \mathcal{S}_1(\mathcal{X}) \cap \mathcal{S}_2(\mathcal{X}) : \mathcal{X} \in Var_{free}(\mathcal{S}_1) \cup Var_{free}(\mathcal{S}_2)\} \cup$
  $\{\mathcal{X}$=ex $\in \mathcal{S}_1 : \mathcal{X} \in Var_{\exists}(\mathcal{S}_1)\} \cup \{\mathcal{X}$=ex $\in \mathcal{S}_2 : \mathcal{X} \in Var_{\exists}(\mathcal{S}_2)\}$

- $Var_{\exists}(\mathcal{S}) = Var_{\exists}(\mathcal{S}_1) \cup Var_{\exists}(\mathcal{S}_2)$

Le plus petit majorant, noté lub($\mathcal{S}_1$,$\mathcal{S}_2$), de deux systèmes résolus $\mathcal{S}_1$ et $\mathcal{S}_2$ désigne le système résolu $\mathcal{S}$ tel que pour toute variable $\mathcal{X}$, on a :

$$sol(\mathcal{S})(\mathcal{X}) = sol(\mathcal{S}_1)(\mathcal{X}) \cup sol(\mathcal{S}_2)(\mathcal{X})$$

Comme $sol(\mathcal{S}_1)(\mathcal{X})$ et $sol(\mathcal{S}_2)(\mathcal{X})$ sont des langages réguliers, il est possible de calculer le langage régulier qui correspond à l'union de $sol(\mathcal{S}_1)(\mathcal{X})$ et $sol(\mathcal{S}_2)(\mathcal{X})$, puis de coder ce langage sous la forme d'un système élémentaire résolu $\mathcal{S}_{\mathcal{X}}$. Pratiquement, on peut calculer $\mathcal{S}$ (avant mise sous forme résolue) comme suit :

- $\mathcal{S} = \{\mathcal{X} = \mathcal{S}_1(\mathcal{X}) \cup \mathcal{S}_2(\mathcal{X}) : \mathcal{X} \in Var_{free}(\mathcal{S}_1) \cup Var_{free}(\mathcal{S}_2)\} \cup$
  $\{\mathcal{X}$=ex $\in \mathcal{S}_1 : \mathcal{X} \in Var_{\exists}(\mathcal{S}_1)\} \cup \{\mathcal{X}$=ex $\in \mathcal{S}_2 : \mathcal{X} \in Var_{\exists}(\mathcal{S}_2)\}$

- $Var_{\exists}(\mathcal{S}) = Var_{\exists}(\mathcal{S}_1) \cup Var_{\exists}(\mathcal{S}_2)$

Exemple

$\mathcal{S}_1$ = { $\mathcal{X}$=a , $\mathcal{Y}$=g($\mathcal{Y}$,$\mathcal{W}_1$) , $\mathcal{W}_1$=a$\cup$b }
$\mathcal{S}_2$ = { $\mathcal{X}$=a , $\mathcal{Y}$=b , $\mathcal{Z}$=f($\mathcal{W}_1$) , $\mathcal{W}_1$=a }

glb($\mathcal{S}_1$,$\mathcal{S}_2$) = { $\mathcal{X}$=a , $\mathcal{Y}$=$\perp_{sc}$, $\mathcal{Z}$=f($\mathcal{W}_1$) , $\mathcal{W}_1$=a }
lub($\mathcal{S}_1$,$\mathcal{S}_2$) = { $\mathcal{X}$=a , $\mathcal{Y}$=g($\mathcal{Y}$,$\mathcal{W}_1$)$\cup$b , $\mathcal{W}_1$=a$\cup$b } ❑

La quantification existentielle d'un système $\mathcal{S}$ par un ensemble de variables $V \subseteq Var_{free}(\mathcal{S})$ désigne le système $\mathcal{S}' = \mathcal{S}$ tel que $Var_{\exists}(\mathcal{S}') = Var_{\exists}(\mathcal{S}) \cup V$.

2.6 Opérateurs de widening

Il est à noter que pour un système $\mathcal{S}$ d'équations ensemblistes donné, le nombre de majorants (pour la relation $\subseteq$) n'est pas nécessairement borné, et que de plus,

l'ensemble des majorants de $\mathcal{S}$ ne vérifie pas nécessairement la condition de chaîne ascendante. Par exemple,

$$\{\mathcal{X}=0\} \subseteq \{\mathcal{X}=0 \cup s(0)\} \subseteq ... \subseteq \{\mathcal{X}=0 \cup s(0) \cup ... \cup s^n(0)\} \subseteq ...$$

Nous définissons trois opérateurs ∇_{depth}, ∇_{prox} et ∇_{card} par rapport aux systèmes d'équations ensemblistes. Ces opérateurs sont définis par rapport à la relation $\subseteq$ et non par rapport à la relation $\vdash$ comme au chapitre 6. Ce choix se justifie à la section 3.4.

Nous donnons tout d'abord une caractérisation d'un système en termes de chemins et nous introduisons ensuite une opération de widening (au sens large) appelée étoile. Les opérateurs de widening ∇_{depth}, ∇_{prox} et ∇_{card} sont définis à la suite.

2.6.1 Chemins d'un système

Soit $\mathcal{S}_\mathcal{X}$ un système élémentaire résolu, chemins$^+(\mathcal{S}_\mathcal{X})$ désigne l'ensemble des chemins complétement spécifiés de $\mathcal{S}_\mathcal{X}$. Il correspond à l'ensemble $CCS_\emptyset(\mathcal{S}_\mathcal{X},\mathcal{X})$ qui est calculé comme indiqué ci-dessous. Si S représente un ensemble de variables et $\mathcal{Y}$ une variable quelconque :

- $CCS_S(\mathcal{S},\mathcal{Y}) = \{(\mathcal{Y},\mathcal{Y})\}$ si $\mathcal{Y} \in S$,
- $CCS_S(\mathcal{S},\mathcal{Y}) = \{(\mathcal{Y},\top_{sc})\}$ si $\mathcal{S}(\mathcal{Y}) = \top_{sc}$
- $CCS_S(\mathcal{S},\mathcal{Y}) = \bigcup_{1\le i\le n} \{CCS_S(\mathcal{S},\mathcal{Y},s_i)\}$ si $\mathcal{S}(\mathcal{Y}) = s_1 \cup ... \cup s_n$
- $CCS_S(\mathcal{S},\mathcal{Y},a) = (\mathcal{Y},a)$
- $CCS_S(\mathcal{S},\mathcal{Y},f(\mathcal{Y}_1,...,\mathcal{Y}_n)) = \bigcup_{1\le i\le n} \{(\mathcal{Y},f_{(i)}).m : m \in CS_S(\mathcal{S},\mathcal{Y}_i)\}$

Un chemin complétement spécifié, généralement noté ms, est une suite non vide de lettres. Chaque lettre représente un couple $(\mathcal{X},\lambda)$ où $\mathcal{X}$ est une variable et λ est une variable ou un symbole de fonction indicé (sauf si le symbole est d'arité nulle). Intuitivement, le premier élément du couple spécifie quelle est l'équation (variable) utilisée pour prendre en compte le second élément. La première règle évite de tomber dans une récursion incontrolée (l'ensemble S mémorise l'ensemble des variables déja rencontrées sur le chemin). Les autres règles permettent de continuer la construction de l'ensemble des chemins spécifiés en tenant compte des symboles de fonction rencontrés.

Soit ms un chemin complétement spécifié, l'image de ms désigne un chemin (non complétement spécifié) m tel que si ms = ps.$(\mathcal{X},\lambda)$ alors m = p.λ où p désigne l'image de ps. Soient $\mathcal{S}$ un système résolu et $\mathcal{X}$ une variable,

chemins($\mathcal{S}_X$) désigne l'ensemble des chemins de $\mathcal{S}_X$, c'est à dire l'ensemble des images de chemins$^+$($\mathcal{S}_X$).

Par exemple, soit

$$\mathcal{S}_X = \{\ X=f(\mathcal{W}_1,\mathcal{W}_2)\cup f(\mathcal{W}_3,\mathcal{W}_4)\ ,\ \mathcal{W}_1=g(\mathcal{W}_1)\cup b\ ,\ \mathcal{W}_3=b\ ,\ \mathcal{W}_4=a\ \},$$

chemins$^+$($\mathcal{S}_X$) =
{ $(X,f_{(1)}).(\mathcal{W}_1,b)$
$(X,f_{(1)}).(\mathcal{W}_1,g_{(1)}).(\mathcal{W}_1,\mathcal{W}_1)$
$(X,f_{(2)}).(\mathcal{W}_2,\top_{sc})$
$(X,f_{(1)}).(\mathcal{W}_3,b)$
$(X,f_{(2)}).(\mathcal{W}_4,a)$,
}

chemins($\mathcal{S}_X$) =
{ $f_{(1)}.b$,
$f_{(1)}.g_{(1)}.\mathcal{W}_1$
$f_{(2)}.\top_{sc}$

$f_{(2)}.a$,
}

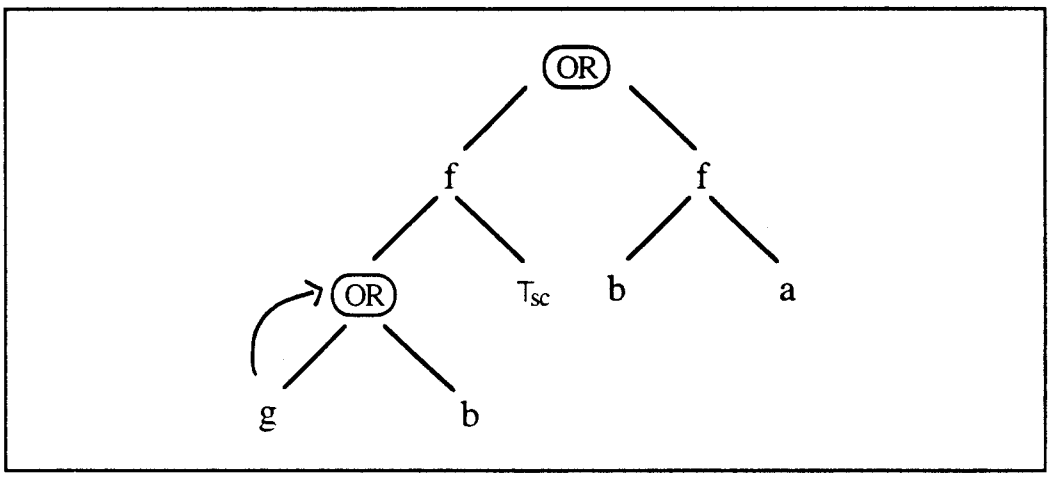

Figure 8.11

Le graphe de type associé à $\mathcal{S}_X$ (figure 8.11) donne une illustration visuelle sur la manière dont sont construits les chemins. Remarquons que l'image de $(X,f_{(1)}).(\mathcal{W}_1,b)$ et de $(X,f_{(1)}).(\mathcal{W}_3,b)$ est le chemin $f_{(1)}.b$. Il existe une surjection de chemins$^+$($\mathcal{S}_X$) sur chemins($\mathcal{S}_X$).

Définition 8.12

Soient m et m' deux chemins, on dit que m admet une récursivité potentielle par rapport à m' ssi il existe m_1, m_2 ($\neq \varepsilon$) et m_3 tel que $m = m_1.m_2.m_3$ et $m' = m_1.m_3$.

Par exemple, $m = s_{(1)}.0$ admet une récursivité potentielle par rapport à $m' = 0$. Il suffit de choisir $m_1 = \varepsilon$, $m_2 = s_{(1)}$ et $m_3 = 0$. De même, $m = f_{(1)}.g_{(2)}.f_{(1)}.b$ admet une récursivité potentielle par rapport à $m' = f_{(1)}.b$. Il suffit de choisir $m_1 = \varepsilon$, $m_2 = f_{(1)}.g_{(2)}$ et $m_3 = f_{(1)}.b$ ou encore $m_1 = f_{(1)}$, $m_2 = g_{(2)}.f_{(1)}$ et $m_3 = b$. Implicitement, $m_3 \neq \varepsilon$ car il n'est pas possible de trouver deux chemins m et m'

tels que $m = m_1.m_2$ et $m' = m_1$ avec $m_2 \neq \varepsilon$. En effet, la dernière lettre de m_1 désigne soit une constante, soit T_{sc} soit une variable. Dans tous les cas, il n'est pas possible de trouver dans tout système un chemin obtenu à partir de m_1 par concaténation d'une lettre.

L'idée derrière la définition 8.12 est de définir une opération de widening (au sens large) appelée étoile qui tienne compte des récursivités potentielles dans un système résolu. L'opération consiste alors à remplacer dans le système m et m' par $m_1.m_2^*.m_3$.

2.6.2 Opération étoile

La fonction étoile permet de généraliser un système en "passant à l'étoile" les récursivités potentielles détectées pour ce système. Pour tout système résolu S, étoile(S) désigne le système S où tout système élémentaire S_X de S a été remplacé par étoile(S_X). Pour tout système élémentaire S_X, étoile(S_X) désigne le système élémentaire (mis sous forme résolue) obtenu à partir de S_X après les deux étapes suivantes.

Algorithme de "passage à l'étoile"

Etape 1 : calcul des ensembles de variables à fusionner

On considère initialement l'ensemble $\mathcal{F}us = \{\{\mathcal{Y}\} : \mathcal{Y} \in \text{Var}(S_X)\}$. $\mathcal{F}us$ désigne les ensembles de variables à fusionner. $\mathcal{F}us$ est modifié en considérant les récursivités potentielles.

Pour tout couple (m,m') d'éléments distincts de chemins(S_X) faire
  Si m admet une récursivité potentielle par rapport à m' alors
    -- soient $m = m_1.m_2.m_3$ et $m' = m_1.m_3$ sa description
    -- soient i,j les positions respectives dans m des 1ères lettres de m_2 et m_3
    -- soit k la position respective dans m' de la 1ère lettre de m_3.
    E := Ø
    Pour tout élément ms de chemins+(S_X) tel que m est l'image de ms faire
      -- soient ($\mathcal{Y}$,λ) et ($\mathcal{Y}$',λ') la ième et la jème lettres de ms
      ajouter $\mathcal{Y}$ et $\mathcal{Y}$' dans E
    Fin pour
    Pour tout élément ms' de chemins+(S_X) tel que m' est l'image de ms' faire
      -- soient ($\mathcal{Y}$,λ) la kème lettre de ms'
      ajouter $\mathcal{Y}$ dans E
    Fin pour
    on fusionne tous les ensembles S de $\mathcal{F}us$ tels que $S \cap E \neq \emptyset$
  Fin si
Fin pour

Il est à noter que $\mathcal{F}us$ constitue une partition de Var($\mathcal{S}_X$).

Etape 2 : modification de $\mathcal{S}_X$

Pour tout ensemble S de $\mathcal{F}us$ faire
-- soit S = $\{\mathcal{W}_1,...,\mathcal{W}_m\}$
on choisit une variable $\mathcal{W}$ de S ($\mathcal{W}$ désigne X si X appartient à S)
on remplace $\mathcal{W}$=$\mathcal{S}_X(\mathcal{W})$ dans $\mathcal{S}_X$ par $\mathcal{W}$=$\mathcal{S}_X(\mathcal{W}_1) \cup ... \cup \mathcal{S}_X(\mathcal{W}_m)$
on supprime de $\mathcal{S}_X$ toutes les équations $\mathcal{W}_j$=$\mathcal{S}_X(\mathcal{W}_j)$ si $\mathcal{W}_j \in$ S et si $\mathcal{W}_j \neq \mathcal{W}$
on remplace dans $\mathcal{S}_X$ toute occurence d'une variable de S par $\mathcal{W}$
Fin pour
résoudre $\mathcal{S}_X$

L'étape 1 du calcul consiste à coder les récursivités potentielles sous forme de fusions d'ensembles de variables. L'idée étant qu'en liant les variables de ces différents ensembles, un "passage à l'étoile" est effectué. A l'étape 2, la variable X est gérée de manière différente des autres variables car elle ne peut être renommée.

Propriété 8.13
Pour tout système résolu $\mathcal{S}$, $\mathcal{S} \subseteq$ étoile($\mathcal{S}$)

Preuve
La preuve se ramène à prouver que pour tout système élémentaire $\mathcal{S}_X$ de $\mathcal{S}$,

$$sol(\mathcal{S}_X)(X) \subseteq sol(\text{étoile}(\mathcal{S}_X))(X)$$

Pour toute variable $\mathcal{W}_j$ de Var($\mathcal{S}_X$), on sait que $\mathcal{W}_j$ appartient à un ensemble S de $\mathcal{F}us$ après l'étape 1. Si $\mathcal{W}$ désigne la variable de S qui a été choisie pour la transformation de $\mathcal{S}_X$, on sait que l'ensemble sol($\mathcal{S}_X$)($\mathcal{W}_j$) considérée avant la transformation est inclus dans l'ensemble sol($\mathcal{S}_X$)($\mathcal{W}$) considérée après la transformation. Comme la variable $\mathcal{W}_j$ est remplacée par la variable $\mathcal{W}$ partout dans $\mathcal{S}_X$ et que les opérateurs sont monotones, il s'agit d'une généralisation du système. ❑

Exemple 1
Pour illustrer l'opération étoile, prenons un premier exemple. Soit le système élémentaire suivant :

$$\mathcal{S}_X = \{\ X=cons(\mathcal{W}_1,\mathcal{W}_2) \cup nil\ ,\ \mathcal{W}_1=a\ ,\ \mathcal{W}_2=nil\ \}$$

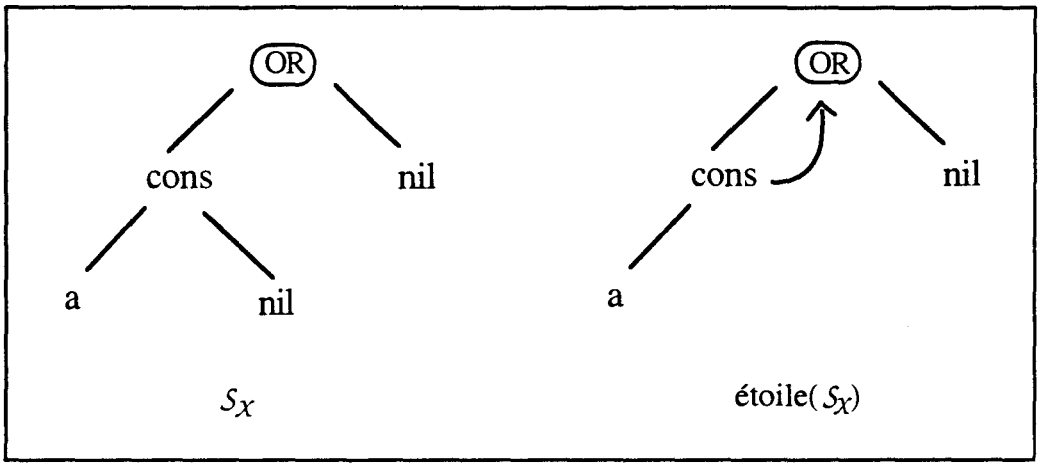

Figure 8.14

La figure 8.14 représente S_X et étoile(S_X) sous la forme de graphes de types. Voici comment est obtenu ce résultat. On a :

chemins$^+$(S_X) = { (X,nil) , (X,cons$_{(1)}$).(W_1,a) , (X,cons$_{(2)}$).(W_2,nil) }
chemins(S_X) = { nil , cons$_{(1)}$.a , cons$_{(2)}$.nil }

Etape 1 : on s'aperçoit que cons$_{(2)}$.nil admet une récursivité potentielle par rapport à nil. Aussi obtient-t-on :

$\mathcal{F}us$ = { {X,W_2} , {W_1} }

Etape 2 :

- on remplace $X = S_X(X)$ par $X = S_X(X) \cup S_X(W_2)$, on obtient :
  { X=cons(W_1,W_2)∪nil∪nil , W_1=a , W_2=nil }
- on supprime $W_2=S_X(W_2)$, on obtient :
  { X=cons(W_1,W_2)∪nil∪nil , W_1=a }
- on remplace partout W_2 par X, on obtient :
  { X=cons(W_1,X)∪nil∪nil , W_1=a }
- on résout, on obtient :
  étoile(S_X) = { X = cons(W_1,X)∪nil , W_1=a } ❑

Exemple 2

Soit le système élémentaire suivant :

S_X = { X=f(W_1)∪f(W_2)∪g(W_1) , W_1=0 , W_2=g(W_1) }

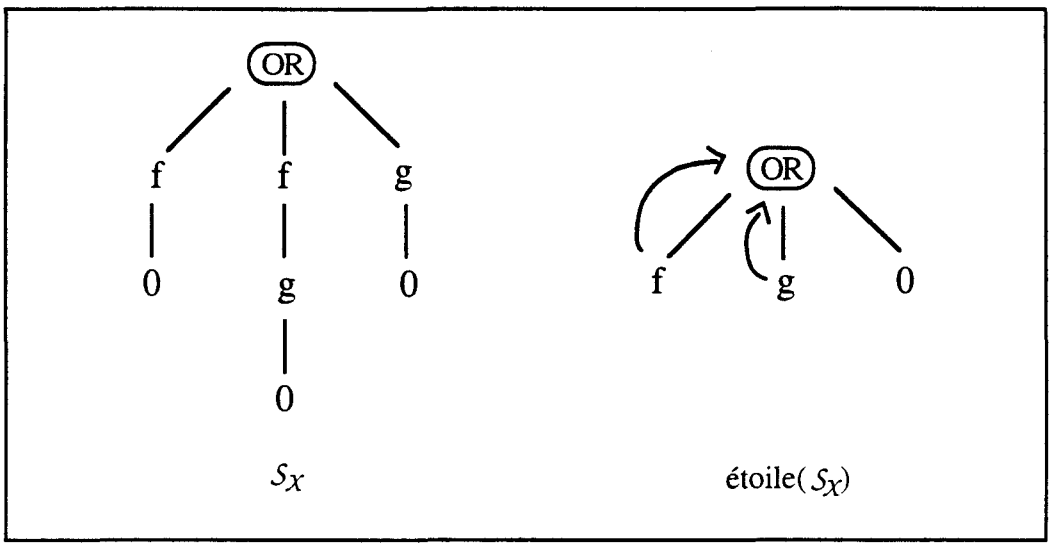

Figure 8.15

La figure 8.15 représente $\mathcal{S}_X$ et étoile($\mathcal{S}_X$) sous la forme de graphes de types. Voici comment est obtenu ce résultat. On a :

chemins$^+$($\mathcal{S}_X$) =
{ $(\mathcal{X},f_{(1)}).(\mathcal{W}_1,0)$, $(\mathcal{X},f_{(1)}).(\mathcal{W}_2,g_{(1)}).(\mathcal{W}_1,0)$, $(\mathcal{X},g_{(1)}).(\mathcal{W}_1,0)$ }

chemins($\mathcal{S}_X$) =
{ $f_{(1)}.0$, $f_{(1)}.g_{(1)}.0$, $g_{(1)}.0$ }

Etape 1 : on s'aperçoit que $f_{(1)}.g_{(1)}.0$ admet une récursivité potentielle par rapport à $f_{(1)}.0$. Aussi obtient-t-on d'abord :

$\mathcal{F}us$ = { $\{\mathcal{X}\}$, $\{\mathcal{W}_1,\mathcal{W}_2\}$ }

puis on s'aperçoit également que $f_{(1)}.g_{(1)}.0$ admet une récursivité potentielle par rapport à $g_{(1)}.0$. Aussi obtient-t-on finalement :

$\mathcal{F}us$ = { $\{\mathcal{X},\mathcal{W}_1,\mathcal{W}_2\}$ }

Etape 2 :

- on remplace $\mathcal{X} = \mathcal{S}_X(\mathcal{X})$ par $\mathcal{X} = \mathcal{S}_X(\mathcal{X}) \cup \mathcal{S}_X(\mathcal{W}_1) \cup \mathcal{S}_X(\mathcal{W}_2)$, on obtient :
  { $\mathcal{X}$=f($\mathcal{W}_1$)∪f($\mathcal{W}_2$)∪g($\mathcal{W}_1$)∪0∪g($\mathcal{W}_1$) , $\mathcal{W}_1$=0 , $\mathcal{W}_2$=g($\mathcal{W}_1$) }

- on supprime $\mathcal{W}_1=\mathcal{S}_X(\mathcal{W}_1)$ et $\mathcal{W}_2=\mathcal{S}_X(\mathcal{W}_2)$, on obtient :
  { $\mathcal{X}$=f($\mathcal{W}_1$)∪f($\mathcal{W}_2$)∪g($\mathcal{W}_1$)∪0∪g($\mathcal{W}_1$) }

- on remplace partout $\mathcal{W}_1$ et $\mathcal{W}_2$ par $\mathcal{X}$, on obtient :

$\{ X=f(X)\cup f(X)\cup g(X)\cup 0\cup g(X) \}$

- on résout, on obtient :
  étoile(S_X) = $\{ X=f(X)\cup g(X)\cup 0 \}$ ❑

Il est à noter que dans certains cas, il est préférable d'effectuer un renommage préalable des variables du système afin d'éviter le partage de structures. Ceci permet d'obtenir une généralisation moins forte (et plus naturelle). Par exemple, si :

$$S_X = \{ X=f(W_1,W_1)\cup f(W_2,W_1)\ ,\ W_1=0\ ,\ W_2=s(W_1) \}$$

alors

$$\text{étoile}(S_X) = \{ X=f(W_1,W_1)\ ,\ W_1=s(W_1)\cup 0 \}$$

Par contre, après renommage, on obtient :

$$S_X = \{ X=f(W_1,W_2)\cup f(W_3,W_4)\ ,\ W_1=0\ ,\ W_2=0\ ,W_3=s(W_5)\ ,\ W_4=0\ ,\ W_5=0 \}$$

et donc

$$\text{étoile}(S_X) = \{ X=f(W_1,W_2)\ ,\ W_1=s(W_1)\cup 0\ ,\ W_2=0 \}$$

On considère fixés pour les sections suivantes un entier k (≥1) et un ensemble fini de variables V.

2.6.3 Opérateur ∇_{depth}

On commence par définir les fonctions depth et abs1. La profondeur, notée depth(m), d'un chemin m (resp. d'un chemin complétement spécifié) désigne le nombre de lettres qui composent le chemin. La profondeur d'un système S est défini en fonction de la profondeur des systèmes élémentaires de S qui eux-mêmes sont définis en fonction de la profondeur de leur chemins associés :

$$\text{depth}(S) = \max \{\text{depth}(S_X) : X \in \text{Var}_{\text{free}}(S)\}$$
$$\text{depth}(S_X) = \max \{\text{depth}(m) : m \in \text{chemins}(S_X)\}$$

En reprenant l'exemple précédent (figure 8.15), on obtient depth(S_X) = 3. La profondeur d'un système élémentaire S_X correspond à la plus longue branche du graphe de type associé à S_X (on ne considère donc pas les cycles).

Soit S un système résolu, l'opération abs1(S) consiste à couper à une profondeur k donnée toutes les branches du graphe de type associé à chaque système élémentaire de S. abs1(S) désigne donc le système résolu obtenu après

avoir considéré pour chaque système élémentaire S_X de S et chaque élément ms de chemins$^+(S_X)$ la transformation suivante :

Si depth(ms) > k, alors
-- soit $(\mathcal{Y},f_{(i)}).(\mathcal{W},\mu)$ les kième et (k+1)ième lettres de ms
remplacer par une nouvelle variable (existentielle) toute occurence de $\mathcal{W}$ dans $S_X(\mathcal{Y})$ située comme ième argument d'un terme de foncteur f.
Fin si

Pour montrer que ∇_{depth} est un opérateur de widening (bien défini), nous introduisons trois lemmes qui vont nous permettre de vérifier les affirmations (a), (b) et (c) du schéma de preuve (voir la section 4.1.1 du chapitre 6).

Lemme 8.16
L'ensemble des systèmes résolus S tels que $Var_{free}(S) \subseteq V$ et $depth(S) \leq k$, est fini.

Preuve
Comme V est fini, il suffit de prouver la propriété pour un système élémentaire S_X. Supposons que la propriété soit vraie pour tout système élémentaire S_X tel que $depth(S_X) \leq k$ et montrons qu'elle reste vraie pour un système élémentaire S_X tel que $depth(S_X) = k+1$. Si $depth(S_X) = k+1$ alors on sait qu'il existe une équation dans S_X de la forme $X = s_1 \cup ... \cup s_m$. Si on supprime $X=S_X(X)$ de S, alors à chaque variable $\mathcal{Y}$ de $S_X(X)$ on peut faire correspondre un système élémentaire $S_\mathcal{Y}$ tel que $depth(S_\mathcal{Y}) \leq k$. De plus, comme S est résolu, on sait que l'ensemble des équations de ces systèmes élémentaires désigne l'ensemble des équations de S privé de $X=S_X(X)$. En utilisant l'hypothèse de récurrence, on déduit qu'à tout terme s_j de $S_X(X)$ on ne peut associer qu'un nombre fini de systèmes résolus. Il reste alors à déterminer que le nombre de termes s_j pouvant apparaître dans $S_X(X)$ est borné. Si ce n'était pas le cas, comme le nombre de symboles de fonctions est fini et que le nombre de systèmes résolus pouvant être associé à chaque terme s_j est fini, cela signifirait que deux termes s_i et s_j sont identiques (représentent le même ensemble). Or d'après la définition d'un système sous forme résolue, cela est impossible. ❑

Lemme 8.17
Pour tout système résolu S, $S \subseteq abs1(S)$.

Preuve
La transformation précédente consiste à remplacer une (occurence de) variable du système par une nouvelle variable dont l'interprétation implicite est T_{SC}. Comme dans un système résolu ne se trouvent que des opérateurs

monotones, chaque transformation généralise le système (par rapport à la relation $\subseteq$). ❑

Lemme 8.18

L'ensemble $\{abs1(\mathcal{S}) : \mathcal{S}$ est un système résolu tel que $Var_{free}(\mathcal{S}) \subseteq V\}$ est fini.

Preuve

Par définition, $abs1(\mathcal{S})$ est un système résolu. Par construction, on sait que pour tout système résolu $\mathcal{S}$, $depth(abs1(\mathcal{S})) \leq k$ et $Var_{free}(abs1(\mathcal{S})) \subseteq V$ puisque les variables introduites lors d'éventuelles transformations sont quantifiées existentiellement. On en déduit que l'ensemble $\{abs1(\mathcal{S}) : \mathcal{S}$ est un système résolu tel que $Var_{free}(\mathcal{S}) \subseteq V\}$ est inclus dans l'ensemble $\{\mathcal{S} : \mathcal{S}$ est un système résolu tel que $depth(\mathcal{S}) \leq k$ et $Var_{free}(\mathcal{S}) \subseteq V\}$. Le lemme 8.18 est donc une conséquence du lemme 8.16. ❑

Propriété 8.19

∇_{depth} est un opérateur de widening

Preuve

Les lemmes 8.16, 8.17 et 8.18 permettent de vérifier les affirmations (a), (b) et (c) du schéma de preuve (section 4.1.1 du chapitre 6). ❑

2.6.4 Opérateur ∇_{prox}

On commence par définir les fonctions prox et abs2.

Soient $\mathcal{S}$ et $\mathcal{S}'$ deux systèmes résolus,

- $prox(\mathcal{S},\mathcal{S}')$ ssi $\exists\, \mathcal{X} \in V, \exists\, m \in chemins(\mathcal{S}_{\mathcal{X}}), \exists\, m' \in chemins(\mathcal{S}'_{\mathcal{X}}) : m_{|k} = m'_{|k}$
- $abs2(\mathcal{S},\mathcal{S}') = abs1(étoile(lub(\mathcal{S},\mathcal{S}')))$

On note $m_{|k}$ le chemin m restreint aux k premières lettres. Il est à noter que abs2 n'est pas uniquement défini en fonction de la fonction étoile. En effet, cette fonction ne permet pas d'assurer le fait que le nombre de généralisations successives soit fini. Il est, par exemple, possible d'obtenir la chaîne ascendante suivante :

$\{X=0\} \subseteq \{X=0 \cup f(g(f(0)))\} \subseteq ... \subseteq \{X=0 \cup ... \cup f^n(g(f^n(0)))\} \subseteq ...$

La fonction étoile est sans effet sur chaque élément de cette suite. On peut ainsi dire, par analogie avec les langages, que la fonction étoile est régulière et non pas algèbrique.

Pour montrer que ∇_{prox} est un opérateur de widening (bien défini), nous introduisons deux lemmes qui vont nous permettre de vérifier les affirmations (a) et (b) du schéma de preuve (voir la section 4.1.1 du chapitre 6).

Lemme 8.20

Il n'existe pas d'ensemble infini E de systèmes résolus tel que :

- si $\mathcal{S}$ appartient à E alors $Var_{free}(\mathcal{S}) \subseteq V$
- si $\mathcal{S}$ et $\mathcal{S}'$ sont deux systèmes distincts de E alors $prox(\mathcal{S},\mathcal{S}') < k$

Preuve

Résulte directement du lemme 8.16. ❑

Lemme 8.21

Pour tout couple $(\mathcal{S},\mathcal{S}')$ de systèmes résolus,
$\mathcal{S} \subseteq abs2(\mathcal{S},\mathcal{S}')$ et $\mathcal{S}' \subseteq abs2(\mathcal{S},\mathcal{S}')$.

Preuve

On a $\mathcal{S} \subseteq lub(\mathcal{S},\mathcal{S}') \subseteq étoile(lub(\mathcal{S},\mathcal{S}')) \subseteq abs1(étoile(lub(\mathcal{S},\mathcal{S}')))$. ❑

Propriété 8.22

∇_{prox} est un opérateur de widening.

Preuve

Les lemmes 8.20, 8.21 et 8.18 vérifient les affirmations (a), (b) et (c) du schéma de preuve (section 4.1.1 du chapitre 6). ❑

2.6.5 Opérateur ∇_{card}

En considérant la fonction de cardinalité et la fonction abs2 définie ci-dessus, on obtient un nouvel opérateur de widening.

Propriété 8.23

∇_{card} est un opérateur de widening.

Preuve

Les propriétés 8.21 et 8.15 vérifient les affirmations (b) et (c) du schéma de preuve (section 4.1.1 du chapitre 6). L'affirmation (a) est trivialement vérifiée par la fonction card. ❑

3 CLP($\mathcal{FT}$+$\mathcal{SC}$)

Maintenant que nous avons introduit les systèmes d'équations ensemblistes, nous les intégrons au domaine de CLP($\mathcal{FT}$). Les contraintes du langage

CLP($\mathcal{FT}$+$\mathcal{SC}$) obtenu par cette extension sont alors de la forme ($\mathcal{ST}$,$\mathcal{SS}$) où $\mathcal{ST}$ et $\mathcal{SS}$ représentent respectivement un système d'équations sur les termes et un système d'équations ensemblistes.

3.1 Définition

CLP($\mathcal{FT}$+$\mathcal{SC}$) est un CLP-langage (Σ,$\mathcal{L}$,$\mathcal{A}$) tel que

- Σ = (S,$\mathcal{F}$,C) où
  - S = {term}
  - $\mathcal{F}$ est un ensemble fini de symboles de fonction.
  - C = {=,⊥,⊤} ∪ {$\in_L$: L est un langage régulier}

- $\mathcal{L}$ désigne l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},C)$ clos par :
  - conjonction
  - quantification existentielle

- $\mathcal{A}$ est une Σ-algèbre telle que
  - l'interprétation de S est donnée par $\mathcal{A}$(term) = $\mathcal{T}erm(\mathcal{F})$
  - l'interprétation des symboles de $\mathcal{F}$ est libre
  - l'interprétation de chaque symbole $\in_L$ est définie comme suit :
    $\forall\ t \in \mathcal{T}erm(\mathcal{F})$, $\mathcal{A}(\in_L(t))$ ssi $t \in L$

Voici un exemple de contrainte c (un élément de $\mathcal{L}$) :

$$X=f(Y) \wedge Z=a \wedge \in_{L_1}(f(Y)) \wedge \in_{L_2}(Z)$$

les langages réguliers L_1 et L_2 étant définis respectivement par

$L_1 := \{f^n(a) : n \geq 0\}$
$L_2 := a \cup b$

Définition 8.24
Soit c une contrainte (un élément de $\mathcal{L}$), sol(c) représente l'ensemble des solutions de c, i.e., l'ensemble des assignations de variables qui satisfont c.

Pour l'exemple précédent, sol(c) désigne l'ensemble des assignations de variables θ tel que $\theta(X) = f^{n+1}(a)$, $\theta(Y) = f^n(a)$ et $\theta(Z) = a$.

Pour simplifier, nous considérons dans la suite, uniquement des contraintes de la forme $\in_L(X)$ où X représente une variable. Il s'agit donc d'une restriction au niveau de l'ensemble $\mathcal{L}$. Par exemple, la contrainte précédente peut s'écrire (ici de façon équivalente) par :

$$X=f(Y) \wedge Z=a \wedge \in_{L_1}(Y) \wedge \in_{L_2}(Z)$$

car $f_{(i)}^{-1}(L_1) = L_1$. Nous préférons, par commodité, changer l'écriture de telles contraintes en introduisant un ensemble de variables ensemblistes $\mathcal{V}_{sc}$ mis en bijection avec l'ensemble $\mathcal{V}$. On considère qu'à chaque variable X de V correspond une variable ensembliste $\mathcal{X}$ de $\mathcal{V}_{sc}$. Tout langage régulier peut être codé sous la forme d'un système d'équations ensemblistes.

Notation 8.25

Pour toute variable X, on décide de coder de manière équivalente l'information $\in_L(X)$ par $X \in \mathcal{X} \wedge \mathcal{S}$ où $\mathcal{S}$ est un système d'équations ensemblistes tel que $Var_\exists(\mathcal{S}) = \{\mathcal{X}\}$ et $sol(\mathcal{S})(\mathcal{X}) = L$. De manière plus générale, on codera $\in_{L_1}(X) \wedge ... \wedge \in_{L_n}(X)$ par $X \in \mathcal{X} \wedge \mathcal{S}$ où $\mathcal{S}$ est un système tel que $Var_\exists(\mathcal{S}) = \{\mathcal{X}\}$ et $sol(\mathcal{S})(\mathcal{X}) = L_1 \cap ... \cap L_n$.

A l'absence d'information de la forme $\in_L(X)$, on fait correspondre $X \in \mathcal{X} \wedge \mathcal{X} = \top_{sc}$. Par simplicité, on peut sous-entendre toute information de la forme $X \in \mathcal{X}$ et toute information de la forme $\mathcal{X} = \top_{sc}$. Pour l'exemple précédent, on obtient d'abord :

$$X=f(Y) \wedge Z=a \wedge X \in \mathcal{X} \wedge \mathcal{X}=\top_{sc} \wedge Y \in \mathcal{Y} \wedge \mathcal{Y}=f(\mathcal{Y}) \cup a \wedge Z \in \mathcal{Z} \wedge \mathcal{Z}=a \cup b$$

puis :

$$X=f(Y) \wedge Z=a \wedge \mathcal{Y}=f(\mathcal{Y}) \cup a \wedge \mathcal{Z}=a \cup b$$

Finalement, l'information peut être codée sous forme de deux ensembles, un ensemble, noté $\mathcal{ST}$, d'équations sur les termes et un ensemble, noté $\mathcal{SS}$, d'équations ensemblistes.

$$(\{ X=f(Y) , Z=a \} , \{ \mathcal{Y}=f(\mathcal{Y}) \cup a , \mathcal{Z}=b \cup c \})$$

Par la suite, une contrainte c sera représentée sous la forme d'un couple $(\mathcal{ST},\mathcal{SS})$. Dans toute contrainte c, n'apparaissent que des variables libres ou liées existentiellement. On utilisera donc les ensembles $Var(c)$, $Var_{free}(c)$ et $Var_\exists(c)$. Une contrainte $c = (\mathcal{ST},\mathcal{SS})$ est implicitement interprétée par : $\exists V\ (\mathcal{ST} \wedge \mathcal{T}riv_\in \wedge \mathcal{SS})$ où V représente $Var_\exists(c)$, $\mathcal{ST}$ et $\mathcal{SS}$ représentent respectivement un système d'équations sur les termes et un système d'équations ensemblistes, et $\mathcal{T}riv_\in$ désigne la formule d'appartenance triviale suivante :

$$\mathcal{T}riv_\in = (\bigwedge_{X \in \mathcal{V}} X \in \mathcal{X})$$

De ce fait, $(\emptyset,\emptyset)$ représente $\top$. Au niveau de toute contrainte $c = (\mathcal{ST},\mathcal{SS})$, on peut ignorer les variables "locales" de l'ensemble $Var_\exists(\mathcal{SS})$. Par ailleurs, on peut supposer que $Var_\exists(\mathcal{ST}) = \emptyset$ car toute variable X de $Var_\exists(\mathcal{ST})$ peut être

considérée comme appartenant à $Var_{\exists}(c)$ puisque, en évitant les conflits de noms, cela revient à introduire la contrainte triviale suivante $X \in \mathcal{X} \wedge \mathcal{X} = \top_{sc}$.

Pour effectuer le lien entre l'ensemble $sol(\mathcal{ST})$ et l'ensemble $sol(\mathcal{SS})$, nous introduisons l'application Φ.

Définition 8.26

Soit $\mathcal{SS}$ un système d'équations ensemblistes, $\Phi(sol(\mathcal{SS}))$ désigne l'ensemble des assignations de variables θ telles que pour toute variable X :

$$\theta(X) \in sol(\mathcal{SS})(\mathcal{X}).$$

Propriété 8.27

Soit $c = (\mathcal{ST},\mathcal{SS})$ une contrainte, $sol(c) = sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS}))$

Preuve

Résulte de la définition de l'application Φ. □

Il est à noter que les deux composantes d'une contrainte c sont de même importance. La première, un système d'équations sur les termes, permet de coder les dépendances entre variables et la seconde, un système d'équations ensemblistes, permet de représenter la structure des termes de manière récursive et non déterministe. L'information déterministe peut être codée par les deux composantes. Par exemple, pour la contrainte suivante :

$$c = (\ \{\ X=f(X',X')\ \}\ ,\ \{\ \mathcal{X}=f(\mathcal{W}_1,\mathcal{W}_2)\cup a\ ,\ \mathcal{W}_1=a\cup b\cup c\ ,\ \mathcal{W}_2=b\cup c\ ,\ \mathcal{Z}=c\ \}\ )$$

$\mathcal{ST}$ établit une dépendance via la variable X' et $\mathcal{SS}$ établit la structure non déterministe de $\mathcal{X}$. Il ne serait pas possible de coder la même information avec un seul ensemble $\mathcal{ST}$ ou un seul ensemble $\mathcal{SS}$. Notons que le système précédent peut être transformé en un système équivalent par un transfert d'informations entre les ensembles $\mathcal{ST}$ et $\mathcal{SS}$:

$$c = (\ \{\ X=f(X',X')\ ,\ Z=c\ \}\ ,\ \{\ \mathcal{X}=f(\mathcal{W}_1,\mathcal{W}_1))\ ,\ \mathcal{W}_1=b\cup c\ ,\ \mathcal{Z}=c\ \}\ )$$

Pour finir, nous introduisons la notation suivante.

Notation 8.28

Soit $c = (\mathcal{ST},\mathcal{SS})$ une contrainte telle que $\mathcal{ST}$ et $\mathcal{SS}$ sont résolus,

$\mathcal{Left}(c) = (Var(c) - Ran(\mathcal{ST}))$

$\mathcal{Right}(c) = (Var(c) - Dom(\mathcal{ST}))$

Nous représentons respectivement les variables de $\mathcal{Left}(c)$ et de $\mathcal{Right}(c)$ par $\{X,...,Z\}$ et $\{X',...,Z'\}$.

3.2 Forme résolue

Nous cherchons à calculer une forme résolue pour toute contrainte c. L'objectif recherché est d'obtenir une forme qui nous permette aisément de décider du vide (si une contrainte donnée est satisfiable ou non) et de l'implication (si une contrainte donnée est plus petite, i.e., moins générale, qu'une autre). Cette décision est nécessaire, dans le premier cas, à l'utilisation de la résolution OLD et, dans le second cas, à l'utilisation de la tabulation. On souhaite également pouvoir effectuer un calcul de borne inférieure (traiter la conjonction) et un calcul de borne supérieure. Ce calcul est nécessaire, dans le premier cas, à l'utilisation de la résolution OLD et, dans le second cas, à la définition d'opérateurs de widening. La forme résolue d'une contrainte c est définie comme suit .

Définition 8.29

Une contrainte c est (sous forme) résolue ssi

- si c est insatisfiable alors $c = \bot$
- si c est satisfiable alors $c = (\mathcal{ST}, \mathcal{SS})$ et
  - $\mathcal{ST}$ et $\mathcal{SS}$ sont résolus
  - $\mathcal{ST}$ est normalisé, i.e., $\mathcal{ST} \approx \text{lub}(\text{sol}(c))$
  - $\mathcal{SS}$ est normalisé, i.e, $\forall\ Y \in \text{Ran}(\mathcal{ST})$, $\mathcal{SS}(\mathcal{Y}) = \top_{sc}$.

Une forme résolue met en évidence l'insatisfiabilité d'une contrainte. Par ailleurs, le fait que $\mathcal{ST}$ soit normalisé implique que toute l'information déterministe et toutes les dépendances entre variables sont codées par $\mathcal{ST}$. On considère ici implicitement une assignation de variables appartenant à sol(c) sous la forme (équivalente) d'un système d'équations sur les termes. D'autre part, le fait que $\mathcal{SS}$ soit normalisé implique que toute l'information de typage initiale concernant les variables de Ran($\mathcal{ST}$) a été reportée sur les variables de $\mathcal{Left}(c)$. La normalisation de $\mathcal{ST}$ et de $\mathcal{SS}$ permet de concevoir une procédure de décision pour décider de l'implication de contraintes (voir la propriété 8.34).

Soit c une contrainte, on note res(c) la contrainte obtenue après l'algorithme de mise sous forme résolue proposée ci-dessous. Il est à noter que cet algorithme possède quelques idendités avec l'algorithme de transformation de formules équationnelles avec contraintes d'appartenance de [Comon et Delor 90] et l'algorithme d'unification de [Uribe 92b]. Mais [Comon et Delor 90] et [Uribe 92b] s'intéressent essentiellement au problème de la satisfiabilité. La forme résolue de c est obtenue après six étapes. Le lecteur peut suivre les différentes étapes sur l'exemple qui est donné après la description de l'algorithme de résolution.

Algorithme de résolution

Etape 1 : initialisation

(a) : résoudre $\mathcal{ST}$

Remarque : c est insatisfiable si $\mathcal{ST} = \bot$.

(b) : résoudre $\mathcal{SS}$

On peut considérer qu'aprés cette étape, aucune variable libre de c n'apparaît dans Ran($\mathcal{ST}$). Pour cela, il suffit d'introduire éventuellement de nouvelles variables quantifiées existentiellement.

Etape 2 : translation de $\mathcal{SS}$ vers la droite.

Cette étape consiste à coder toutes les contraintes de $\mathcal{SS}$ par rapport aux variables de $\mathcal{R}ight(c)$. Cela signifie qu'après cette étape, toute variable X de c qui appartient à Dom($\mathcal{ST}$), ou de manière équivalente qui n'appartient pas à $\mathcal{R}ight(c)$, est telle qu'aucune contrainte n'est imposée par $\mathcal{SS}$ sur cette variable. Cette translation de $\mathcal{SS}$ vers la droite est effectuée via l'information codée par $\mathcal{ST}$. Il y a alors un **report des contraintes structurelles et de non linéarité de $\mathcal{ST}$ vers $\mathcal{SS}$**. Les contraintes structurelles désignent la structure déterministe des termes et les contraintes de non-linéarité désignent les dépendances entre les variables.

Pour cette étape, au lieu de considérer une contrainte c sous la forme d'un couple ($\mathcal{ST}$,$\mathcal{SS}$) sachant que la formule d'appartenance triviale $\mathcal{T}riv_{\in}$ est sous-entendue, on considérera plutôt c sous la forme d'une conjonction $\mathcal{ST} \wedge f_{\in}$ où $f_{\in}$ désigne une formule d'appartenance non triviale. Le système $\mathcal{SS}$ est alors sous-entendu (car sa gestion ne pose aucun problème particulier).

(c) : remplacer $\mathcal{T}riv_{\in}$ par $f_{\in}$.

$$\mathcal{T}riv_{\in} = (\bigwedge_{X \in \mathcal{V}} X \in \mathcal{X})$$

et

$$f_{\in} = (\bigwedge_{X \in \mathcal{V}} \mathcal{ST}(X) \in \mathcal{X}),$$

(d) : remplacer (tant que possible) toute contrainte $f(t_1,...,t_n) \in \mathcal{X}$ de $f_{\in}$ par :

- $\bigvee_{\substack{s_j = f(\mathcal{X}_{j.1},...,\mathcal{X}_{j.n}) \\ 1 \le j \le m}} (t_1 \in \mathcal{X}_{j.1} \wedge ... \wedge t_n \in \mathcal{X}_{j.n})$ ssi $\mathcal{SS}(\mathcal{X}) = s_1 \cup ... \cup s_m$

- $\top$ ssi $\mathcal{SS}(\mathcal{X}) = \top_{sc}$
- $\bot$ ssi $\mathcal{SS}(\mathcal{X}) = \bot_{sc}$

(e) : mettre $f_\in$ sous forme normale disjonctive et appliquer (tant que possible) sur $f_\in$ les transformations suivantes :

- remplacer $X' \in \mathcal{X}'_i$ par $\bot$ ssi $\mathcal{SS}(\mathcal{X}'_i) = \bot_{sc}$
- remplacer $\bot \wedge ex$ par $\bot$ et $\bot \vee ex$ par ex
- remplacer $\top \wedge ex$ par ex et $\top \vee ex$ par $\top$
- remplacer $X' \in \mathcal{X}'_i \wedge X' \in \mathcal{X}'_j$ par $X' \in \mathcal{X}'_k$ et coder dans $\mathcal{SS}$ le système élémentaire $\mathcal{SS}_{\mathcal{X}'_k}$: $sol(\mathcal{SS})(\mathcal{X}'_k) = sol(\mathcal{SS})(\mathcal{X}'_i) \cap sol(\mathcal{SS})(\mathcal{X}'_j)$.

Remarque : c est insatisfiable si $f_\in = \bot$.

En pratique, au lieu de considérer l'ensemble $\mathcal{V}$ (voir la phase (c)), on ne tient compte que des variables de c. De manière classique, pour la première transformation de la phase (d), on interprète une disjonction vide par la contrainte $\bot$ et une conjonction vide par la contrainte $\top$ (cas éventuel si $f(t_1,...,t_n)$ désigne une constante). La phase (d) permet de coder toutes les contraintes d'appartenance par rapport aux variables de $\mathcal{R}ight(c)$. Après cette phase, notre convention de codage (voir la notation 8.25) n'est plus valable puisque nous n'avons plus nécessairement dans $f_\in$ des contraintes atomiques de la forme $X' \in \mathcal{X}'$. De plus, dans $f_\in$, plusieurs contraintes atomiques peuvent porter sur une même variable X'. Par souci de simplicité, et sans perte de généralité, nous écrivons chaque occurrence d'une contrainte atomique portant sur une variable X' (disons la ième) par $X' \in \mathcal{X}'_i$ et on considére codée dans $\mathcal{SS}$ un système élémentaire $\mathcal{SS}_{\mathcal{X}'_i}$. La phase (e) permet d'obtenir une bonne forme pour $f_\in$. On considère sans perte de généralité que :

soit $f_\in = \bot$,
soit $f_\in = (f_1 \vee ... \vee f_m)$ où :
- f_j est de la forme $(X' \in \mathcal{X}'_j \wedge ... \wedge Z' \in \mathcal{Z}'_i)$ avec $1 \le j \le m$
- $Dom(f_j) = \mathcal{R}ight(c)$.

Nous notons $Dom(f_j)$ l'ensemble des variables qui apparaissent en partie gauche d'une contrainte atomique de f_j. Pour obtenir $Dom(f_j) = \mathcal{R}ight(c)$, il peut être nécessaire d'introduire de nouvelles contraintes atomiques triviales, i.e., des contraintes atomiques de la forme $Y' \in \mathcal{Y}'_i$ et telles que $\mathcal{SS}(\mathcal{Y}'_i) = \top_{sc}$.

Etape 3 : report des contraintes de non-linéarité de $\mathcal{SS}$ vers $\mathcal{ST}$

Cette étape consiste à reporter sur $\mathcal{ST}$ les contraintes de non-linéarité codées par $\mathcal{SS}$. Celles-ci apparaissent sous la forme de dépendances inter-arguments et ne peuvent être prises en compte que pour des ensembles dont le cardinal est égal à 1.

(f) : fusionner (tant que possible) deux variables X' et Y' de $\mathcal{Right}(c)$ ssi pour tout j compris entre 1 et m :

$$sol(\mathcal{SS})(\mathcal{X}_j) = sol(\mathcal{SS})(\mathcal{Y}_j) \text{ et } card(sol(\mathcal{SS})(\mathcal{X}_j)) = card(sol(\mathcal{SS})(\mathcal{Y}_j)) = 1$$

Fusionner signifie ici renommer X' par Y' (ou l'inverse) dans $\mathcal{ST}$ et éliminer toute information concernant X' dans $\mathcal{SS}$. Tester si le cardinal d'une expression ensembliste est égal à l'unité est clairement décidable (voir par exemple l'algorithme de [Heintze 92] page 171-172).

Etape 4 : translation de $\mathcal{SS}$ vers la gauche

Cette étape permet d'effectuer une translation de $\mathcal{SS}$ vers la gauche. Cela signifie que toute variable X de c qui appartient à Ran($\mathcal{ST}$), ou de manière équivalente qui n'appartient pas à $\mathcal{Left}(c)$, est telle qu'aucune contrainte n'est imposée par $\mathcal{SS}$ sur cette variable.

(g) : remplacer $f_\in = (f_1 \vee ... \vee f_m)$ par $g_\in = (g_1 \vee ... \vee g_m)$ où

- $g_j = (X \in \mathcal{X}_j \wedge ... \wedge Z \in \mathcal{Z}_j)$ avec $1 \leq j \leq m$
- $Dom(g_j) = \mathcal{Left}(c)$
- pour tout entier j compris entre 1 et m et toute variable Y de $Dom(g_j)$, $\mathcal{SS}_{\mathcal{Y}_j}$ désigne le système élémentaire résolu tel que :

  $$sol(\mathcal{SS})(\mathcal{Y}_j) = sol(SS)(\mathcal{ST}(\mathcal{Y}))$$

  où $\mathcal{ST}(\mathcal{Y})$ représente le terme $\mathcal{ST}(Y)$ où toute variable W a été remplacée par la variable ensembliste $\mathcal{W}$.

(h) : remplacer $g_\in = (g_1 \vee ... \vee g_m)$ par $\mathcal{Triv}_\in$ et coder dans $\mathcal{SS}$ pour toute variable Y de $\mathcal{Left}(c)$ le système élémentaire résolu $\mathcal{SS}_\mathcal{Y}$ tel que :

$$sol(\mathcal{SS})(\mathcal{Y}) = sol(\mathcal{SS})(\mathcal{Y}_1) \cup ... \cup sol(\mathcal{SS})(\mathcal{Y}_m)$$

La phase (g) consiste à reporter toutes les contraintes d'appartenance par rapport aux variables de $\mathcal{Left}(c)$. La phase (h) consiste à obtenir de nouveau la formule d'appartenance triviale $\mathcal{Triv}_\in$ par fusion des éléments de $g_\in$. On peut ainsi

considérer à nouveau c sous la forme d'un couple ($\mathcal{ST}$,$\mathcal{SS}$) en sous-entendant la formule d'appartenance.

Etape 5 : report des contraintes structurelles de $\mathcal{SS}$ vers $\mathcal{ST}$

Cette étape consiste à reporter sur $\mathcal{ST}$ toute l'information structurelle, i.e., l'information déterministe sur la structure des termes, codée par $\mathcal{SS}$.

(i) : pour toute variable X de (Var(c) - Ran($\mathcal{ST}$)) :

- ajouter l'équation X = det({$\mathcal{X}$}) dans $\mathcal{ST}$
- résoudre $\mathcal{ST}$.

Si S = {$\mathcal{X}_1$,...,$\mathcal{X}_n$} est un ensemble de variables ensemblistes alors :

- det(S) = f(det({$\mathcal{X}_{1.1}$,...,$\mathcal{X}_{m.1}$}),...,det({$\mathcal{X}_{1.p}$,...,$\mathcal{X}_{m.p}$}) si (avec n ≤ m)

  $\mathcal{SS}(\mathcal{X}_1) \cup \ ... \cup \mathcal{SS}(\mathcal{X}_n) = f(\mathcal{X}_{1.1},...,\mathcal{X}_{1.p}) \cup ... \cup f(\mathcal{X}_{m.1},...,\mathcal{X}_{m.p})$

- det(S) = X' sinon et X' désigne une nouvelle variable.

Les nouvelles variables introduites dans $\mathcal{ST}$ appartiennent à Var$_\exists$(c).

Etape 6 (facultative) : simplifier

(j) : pour toute variable existentielle X de $\mathcal{L}eft(c)$:

- éliminer X=$\mathcal{ST}$(X) de $\mathcal{ST}$
- éliminer $\mathcal{SS}_\mathcal{X}$ de $\mathcal{SS}$.

Exemple

Nous illustrons maintenant les différentes phases de la mise sous forme résolue d'une contrainte c.

Contrainte initiale : Soit c = ($\mathcal{ST}$,$\mathcal{SS}$) où :

$\mathcal{ST}$ = { X=f(X',X') , Y=g(Z,c) }
$\mathcal{SS}$ = $\mathcal{SS}_\mathcal{X} \cup \mathcal{SS}_\mathcal{Y} \cup \mathcal{SS}_\mathcal{Z}$ et
$\mathcal{SS}_\mathcal{X}$ = { $\mathcal{X}$=f($\mathcal{W}_1$,$\mathcal{W}_2$)∪f($\mathcal{W}_1$,$\mathcal{W}_1$)∪f($\mathcal{W}_2$,$\mathcal{W}_2$) , $\mathcal{W}_1$=a , $\mathcal{W}_2$ =b }
$\mathcal{SS}_\mathcal{Y}$ = { $\mathcal{Y}$=g($\mathcal{W}_3$,$\mathcal{W}_4$)∪g($\mathcal{W}_5$,$\mathcal{W}_6$) , $\mathcal{W}_3$=a , $\mathcal{W}_4$ =d , $\mathcal{W}_5$=b , $\mathcal{W}_6$ =c }
$\mathcal{SS}_\mathcal{Z}$ = { $\mathcal{Z}$=a∪b }
Var$_\exists$(c) = {X'}

(a) : Aucun changement.

(b) : Introduction d'une nouvelle variable. Seuls changent $\mathcal{ST}$ et $\mathrm{Var}_{\exists}(c)$:

$\mathcal{ST}$ = { X=f(X',X') , Y=g(Y',c) , Z=Y' }
$\mathrm{Var}_{\exists}(c)$ = {X',Y'}

(c) : Passage de $\mathcal{T}riv_{\in}$ à $f_{\in}$. on a $f_{\in}$ =

(f(X',X') $\in \mathcal{X} \wedge$ g(Y',c) $\in \mathcal{Y} \wedge$ Y' $\in \mathcal{Z} \wedge$ X' $\in \mathcal{X}' \wedge$ Y' $\in \mathcal{Y}'$)

(d) : Développement de $f_{\in}$. On obtient $f_{\in}$ =

$(((X' \in \mathcal{W}_1 \wedge X' \in \mathcal{W}_2) \vee (X' \in \mathcal{W}_1 \wedge X' \in \mathcal{W}_1) \vee (X' \in \mathcal{W}_2 \wedge X' \in \mathcal{X}_2))$
$\wedge ((Y' \in \mathcal{W}_3 \wedge \bot) \vee (Y' \in \mathcal{W}_5 \wedge \top)) \wedge Y' \in \mathcal{Z} \wedge X' \in \mathcal{X}' \wedge Y' \in \mathcal{Y}'$)

(e) : Simplification de $f_{\in}$. On obtient $f_{\in} = (f_1 \vee f_2)$ avec

- $f_1 = (X' \in \mathcal{X}'_1 \wedge Y' \in \mathcal{Y}'_1)$
- $f_2 = (X' \in \mathcal{X}'_2 \wedge Y' \in \mathcal{Y}'_2)$
- $\mathcal{SS} = \mathcal{SS}_{\mathcal{X}'_1} \cup \mathcal{SS}_{\mathcal{X}'_2} \cup \mathcal{SS}_{\mathcal{Y}'_1} \cup \mathcal{SS}_{\mathcal{Y}'_2}$
  $\mathcal{SS}_{\mathcal{X}'_1}$ = { $\mathcal{X}'_1$=a }
  $\mathcal{SS}_{\mathcal{Y}'_1}$ = { $\mathcal{Y}'_1$=b }
  $\mathcal{SS}_{\mathcal{X}'_2}$ = { $\mathcal{X}'_2$=b }
  $\mathcal{SS}_{\mathcal{Y}'_2}$ = { $\mathcal{Y}'_2$=b }

(f) : Aucun changement (cela n'eut pas été le cas si $\mathcal{Y}'_1$=a remplaçait $\mathcal{Y}'_1$=b)

(g) : Passage de $f_{\in}$ à $g_{\in}$. On obtient $g_{\in} = (g_1 \vee g_2)$ avec

- $g_1 = (X \in \mathcal{X}_1 \wedge Y \in \mathcal{Y}_1 \wedge Z \in \mathcal{Z}_1)$
- $g_2 = (X \in \mathcal{X}_2 \wedge Y \in \mathcal{Y}_2 \wedge Z \in \mathcal{Z}_2)$
- $\mathcal{SS} = \mathcal{SS}_{\mathcal{X}_1} \cup \mathcal{SS}_{\mathcal{X}_2} \cup \mathcal{SS}_{\mathcal{Y}_1} \cup \mathcal{SS}_{\mathcal{Y}_2} \cup \mathcal{SS}_{\mathcal{Z}_1} \cup \mathcal{SS}_{\mathcal{Z}_2}$
  $\mathcal{SS}_{\mathcal{X}_1}$ = { $\mathcal{X}_1$=f($\mathcal{W}_1$,$\mathcal{W}_1$) , $\mathcal{W}_1$=a }
  $\mathcal{SS}_{\mathcal{Y}_1}$ = { $\mathcal{Y}_1$=g($\mathcal{W}_2$,$\mathcal{W}_3$) , $\mathcal{W}_2$=b , $\mathcal{W}_3$=c } ,
  $\mathcal{SS}_{\mathcal{Z}_1}$ = { $\mathcal{Z}_1$=b }
  $\mathcal{SS}_{\mathcal{X}_2}$ = { $\mathcal{X}_2$=f($\mathcal{W}_4$,$\mathcal{W}_4$) , $\mathcal{W}_4$=a }
  $\mathcal{SS}_{\mathcal{Y}_2}$ = { $\mathcal{Y}_2$=g($\mathcal{W}_5$,$\mathcal{W}_6$) , $\mathcal{W}_5$=b , $\mathcal{W}_6$=c }
  $\mathcal{SS}_{\mathcal{Z}_2}$ = { $\mathcal{Z}_2$=b }

(h) : Passage de $g_{\in}$ à $\mathcal{T}riv_{\in}$. On obtient $c = (\mathcal{ST},\mathcal{SS})$ avec

$\mathcal{ST}$ = { X=f(X',X') , Y=g(Y',c) , Z=Y' }
$\mathcal{SS} = \mathcal{SS}_{\mathcal{X}} \cup \mathcal{SS}_{\mathcal{Y}} \cup \mathcal{SS}_{\mathcal{Z}}$
  $\mathcal{SS}_{\mathcal{X}}$ = { $\mathcal{X}$=f($\mathcal{W}_1$,$\mathcal{W}_1$)$\cup$f($\mathcal{W}_2$,$\mathcal{W}_2$) , $\mathcal{W}_1$=a , $\mathcal{W}_2$=b}

$\mathcal{SS}_{\mathcal{Y}}$ = { $\mathcal{Y}$=g($\mathcal{W}_3$,$\mathcal{W}_4$) , $\mathcal{W}_3$=b , $\mathcal{W}_4$=c } ,
$\mathcal{SS}_{\mathcal{Z}}$ = { $\mathcal{Z}$=b }
Var$_\exists$(c) = {X',Y'}

(i) : Calcul de l'information déterministe de $\mathcal{SS}$

det({X}) retourne f(X1,X2), det({Y}) retourne g(b,c) et det({Z}) retourne b. En ajoutant X=f(X1,X2), Y=g(b,c) et Z=b à $\mathcal{ST}$, on obtient :

$\mathcal{ST}$ = { X=f(X',X') , Y=g(b,c) , Z=b }

(j) : Aucun changement. ❑

Propriété 8.30
res(c) est équivalent à c.

Preuve

Nous montrons ici que toutes les phases de transformations sont correctes, i.e., préservent l'équivalence.

(a) : Immédiat.

(b) : Immédiat.

(c) : Initialement, au début de l'étape 2, nous avons une contrainte c de la forme :

$$\mathcal{ST} \wedge (\bigwedge_{X \in \mathcal{V}} X \in \mathcal{X})$$

Cette contrainte est équivalente à :

$$\mathcal{ST} \wedge (\bigwedge_{X \in \mathcal{V}} \mathcal{ST}(X) \in \mathcal{X})$$

puisque X=$\mathcal{ST}$(X).

(d) : D'une part, si $\mathcal{SS}(\mathcal{Y}) = \top_{sc}$ alors $f(t_1,...,t_n) \in \mathcal{Y}$ est équivalent à $\top$ car pour toute asignation de variables θ : $\theta(f(t_1,...,t_n)) \in \mathcal{Term}(\mathcal{F})$. D'autre part, si $\mathcal{SS}(\mathcal{Y}) = \bot_{sc}$ alors $f(t_1,...,t_n) \in \mathcal{Y}$ est équivalent à $\bot$ car il n'existe aucune assignation de variables θ telle que $\theta(f(t_1,...,t_n)) \in \emptyset$. La correction de la première transformation est immédiate.

(e) : Si $\mathcal{SS}(\mathcal{X}_i) = \perp_{sc}$ alors $X \in \mathcal{X}_i$ est équivalent à $\perp$ car il n'existe aucune assignation de variables θ telle que $\theta(X) \in \emptyset$. Les autres transformations de (e) sont immédiates.

Caractérisation de $f_\in$: Pour prouver la correction de la phase (h), nous donnons après l'étape (e) une caractérisation de $f_\in$.

La contrainte

$$\mathcal{ST} \wedge (\bigwedge_{X \in \mathcal{V}} \mathcal{ST}(X) \in \mathcal{X})$$

est équivalente à

$$\mathcal{ST} \wedge (\bigwedge_{X \in \mathcal{V}} f_X)$$

f_X représente la formule obtenue à partir de $\mathcal{ST}(X) \in \mathcal{X}$ en appliquant les transformations de l'étape 2. On peut considérer sans perte de généralité que :

soit $f_X = \perp$,
soit $f_X = ((f_X)_1 \vee ... \vee (f_X)_k)$ où :
- $(f_X)_{k_x}$ est de la forme $(X' \in \mathcal{X}'_{k_x} \wedge ... \wedge Z' \in \mathcal{Z}'_{k_x})$ avec $1 \leq k_x \leq k$
- $\mathrm{Dom}((f_X)_{k_x}) = \mathrm{Var}(\mathcal{ST}(X))$.

On peut noter ensuite qu'en appliquant les transformations de la phase (e) sur :

$$\bigwedge_{X \in \mathcal{V}} f_X$$

on obtient la formule $f_\in$. Cela revient à dire que si $f_\in = (f_1 \vee ... \vee f_m)$ alors pour tout entier j compris entre 1 et m, f_j est équivalent à une conjonction, notée conj(f_j), de la forme :

$$\bigwedge_{X \in \mathcal{V}} (f_X)_{k_x}$$

(f) : Immédiat.

(g) : Cette phase consiste à

remplacer $f_\in = (f_1 \vee ... \vee f_m)$ par $g_\in = (g_1 \vee ... \vee g_m)$.

Pour prouver la correction de cette transformation, il suffit de montrer que :

$\mathcal{ST} \wedge f_j$ est équivalent à $\mathcal{ST} \wedge g_j$

puisque

$\mathcal{ST} \wedge f$ est équivalent à $(\mathcal{ST} \wedge f_1) \vee ... \vee (\mathcal{ST} \wedge f_m)$
$\mathcal{ST} \wedge g$ est équivalent à $(\mathcal{ST} \wedge g_1) \vee ... \vee (\mathcal{ST} \wedge g_m)$

On prouve que $\mathcal{ST} \wedge f_j$ est équivalent à $\mathcal{ST} \wedge g_j$ en remarquant simplement que (par construction de g_j) :

pour toute assignation de variables θ et pour toute variable Y de $\mathcal{L}eft(c)$, on a $\theta(Y) \in sol(\mathcal{SS})(\mathcal{Y}_j)$ ssi pour toute variable Y' de $\mathcal{ST}(Y)$: $\theta(Y') \in sol(\mathcal{SS})(\mathcal{Y}_j)$.

(h) : Cette phase consiste à

remplacer $g_\in = (f_1 \vee ... \vee f_m)$ par $\mathcal{T}riv_\in$.

Pour prouver la correction de cette transformation, il faut montrer que :

$\mathcal{ST} \wedge g_\in$ est équivalent à $\mathcal{ST} \wedge \mathcal{T}riv_\in$

Nous commençons d'abord par montrer que toute solution θ de $\mathcal{ST} \wedge g_\in$ est une solution de $\mathcal{ST} \wedge \mathcal{T}riv_\in$. Si θ est solution de $\mathcal{ST} \wedge g_\in$ alors il existe un entier j tel que θ est solution de $\mathcal{ST} \wedge g_j$. Aussi, pour toute variable Y de $\mathcal{L}eft(c)$, on a $\theta(Y) \in sol(\mathcal{SS})(\mathcal{Y}_j)$. Comme $sol(\mathcal{SS})(\mathcal{Y}) = sol(\mathcal{SS})(\mathcal{Y}_1) \cup ... \cup sol(\mathcal{SS})(\mathcal{Y}_m)$, on en déduit que $\theta(Y) \in sol(\mathcal{SS})(\mathcal{Y})$ et que θ est solution de $\mathcal{ST} \wedge \mathcal{T}riv_\in$.

Considérons maintenant une solution θ de $\mathcal{ST} \wedge \mathcal{T}riv_\in$ et montrons que θ est une solution de $\mathcal{ST} \wedge g_\in$. Il faut donc montrer qu'il existe un entier p tel que θ est solution de $\mathcal{ST} \wedge g_p$.

Si θ est solution de $\mathcal{ST} \wedge \mathcal{T}riv_\in$ alors pour toute variable Y de $\mathcal{L}eft(c)$, on sait que $\theta(Y) \in sol(\mathcal{SS})(\mathcal{Y})$. Comme $sol(\mathcal{SS})(\mathcal{Y}) = sol(\mathcal{SS})(\mathcal{Y}_1) \cup ... \cup sol(\mathcal{SS})(\mathcal{Y}_m)$, on en déduit qu'il existe un entier j tel que $\theta(Y) \in sol(\mathcal{SS})(\mathcal{Y}_j)$. En outre, $\mathcal{ST} \wedge f_j$ est équivalent à $\mathcal{ST} \wedge g_j$, aussi, pour toute variable Y' de $\mathcal{ST}(Y)$, on a $\theta(Y') \in sol(\mathcal{SS})(\mathcal{Y}_j)$. Ensuite, on sait que f_j est est équivalent à une conjonction, notée $conj(f_j)$, de la forme :

$$\bigwedge_{X \in \mathcal{V}} (f_X)_{k_x}$$

Il en résulte que θ est solution de $(f_Y)_{k_y}$ où $(f_Y)_{k_y}$ est la contrainte associée à Y dans $conj(f_j)$. θ est solution de $(f_Y)_{k_y}$ car :

◦ pour toute variable Y' de $\mathcal{ST}(Y)$,

$\theta(Y') \in sol(\mathcal{SS})(\mathcal{Y}_j)$
$sol(\mathcal{SS})(\mathcal{Y}_j) \subseteq sol(\mathcal{SS})(\mathcal{Y}_{k_y})$ par construction

- $Dom((f_Y)_{k_y}) = Var(\mathcal{ST}(Y))$

En considérant le raisonnement précédent pour toute variable de $\mathcal{L}eft(c)$, on peut construire une conjonction de la forme :

$$(f_X)_{k_x} \wedge ... \wedge (f_Z)_{k_z}$$

On sait que si θ est solution de $\mathcal{ST} \wedge \mathcal{T}riv_{\in}$ alors θ est solution de $(f_X)_{k_x} \wedge ... \wedge (f_Z)_{k_z}$. Il reste à considérer les variables Y' de $Ran(\mathcal{ST})$. On sait que $f_{Y'}$ désigne simplement la contrainte atomique $Y' \in \mathcal{Y}$ car le terme $\mathcal{ST}(Y')$ désigne Y'. Est-il possible que θ ne soit pas solution de :

$$(f_X)_{k_x} \wedge ... \wedge (f_Z)_{k_z} \wedge f_{Y'} ?$$

Si c'est le cas, cela signifie que θ n'est pas solution de $f_{Y'}$. Mais pour tout f_j, $f_{Y'}$ apparaît nécessairement dans $conj(f_j)$. On en déduit alors que $\theta(Y') \notin sol(\mathcal{SS})(\mathcal{Y}_j)$, puis par construction que $\theta(Y') \notin sol(\mathcal{SS})(\mathcal{Y})$. Ceci constitue une contradiction car dans ce cas θ n'est pas solution de $\mathcal{ST} \wedge \mathcal{T}riv_{\in}$.

Nous déduisons de ceci qu'il existe un entier p tel que f_p correspond à la conjonction $(f_X)_{k_x} \wedge ... \wedge (f_Z)_{k_z} \wedge f_{X'} \wedge ... \wedge f_{Z'}$ et tel que θ est solution de f_p.

(i) : Immédiat puisque pour toute variable X et toute solution θ de c, on a $\theta(X) \leq det(\{\mathcal{X}\})$. La preuve se fait par récurrence.

(j) : Sous réserve que c soit satisfiable, on sait que toute information concernant une variable X de $Var_{\exists}(c)$ est triviale et peut de ce fait être éliminée. ❑

Propriété 8.31
L'insatisfiabilité d'une contrainte est décidable.

Preuve
Pour le montrer, nous utilisons l'algorithme de mise sous forme résolue. Tout d'abord, il est important de noter que la possibilité d'obtenir la contrainte $\perp$ apparaît à deux niveaux différents : au niveau de $\mathcal{ST}$ et au niveau de $f_{\in}$ (car $X \in \mathcal{X}_i \wedge \mathcal{SS}(\mathcal{X}_i) = \perp_{sc}$ donne nécessairement $\perp$). Une contrainte c est insatisfiable ssi $\mathcal{ST} = \perp$ après la phase (a) ou $f_{\in} = \perp$ après la phase (e). D'une part, puisque $c = \mathcal{ST} \wedge f_{\in}$, il est évident que si $\mathcal{ST} = \perp$ ou si $f_{\in} = \perp$ alors c est insatisfiable. D'autre part, supposons que $\mathcal{ST} \neq \perp$ et $f_{\in} \neq \perp$. Le fait que $f_{\in} \neq \perp$ implique que $f_{\in}$ soit de la forme $(f_1 \vee ... \vee f_m)$ et que chaque

f_j soit de la forme ($X' \in \mathcal{X}'_j \wedge ... \wedge Z' \in \mathcal{Z}'_j$). Après la phase (e), on sait que pour toute variable Y' de Dom(f_j), on a $\mathcal{SS}(\mathcal{Y}'_j) \neq \bot_{sc}$. Il existe donc au moins une assignation de variables θ telle que $\theta(Y') \in sol(\mathcal{SS}_{\mathcal{Y}'_j})$ pour toute variable de $\mathcal{L}eft(c)$. Cette assignation s'étend alors sans problèmes aux variables de Dom($\mathcal{ST}$) en considérant les équations de $\mathcal{ST}$. On en déduit donc que la contrainte c est satisfiable. ❑

Propriété 8.32
res(c) désigne une contrainte résolue.

Preuve

Si c est insatisfiable alors on considère que res(c) = $\bot$ car l'insatisfiabilité de c est décidable (voir la propriété précédente).

Si c est satisfiable alors res(c) = ($\mathcal{ST}$,$\mathcal{SS}$). $\mathcal{ST}$ est sous forme résolue après la phase (i) et $\mathcal{SS}$ est sous forme résolue après la phase (h). $\mathcal{ST}$ est normalisé car toutes les contraintes structurelles (étape 5) et de non-linéarité (étape 3) de $\mathcal{SS}$ ont été reportées sur $\mathcal{ST}$. $\mathcal{SS}$ est normalisé car $\mathcal{SS}$ a subi une translation vers la gauche (étape 4). ❑

Propriété 8.33
Le calcul de res(c) est fini.

Preuve

Immédiat pour toutes les phases sauf la phase (i). Comme $Var_{free}(\mathcal{SS})$ est un ensemble fini, il suffit de prouver que le calcul de det(S) termine. Si celui-ci ne termine pas, cela revient à dire qu'il existe un ensemble S' tel que le calcul de det(S') se répète de manière récursive car le nombre de variables dans $\mathcal{SS}$ est fini. Pour toute variable ensembliste $\mathcal{X}$ de S', cela implique que $sol(\mathcal{SS})(\mathcal{X}) = \emptyset$ car tout terme t appartenant à $sol(\mathcal{SS})(\mathcal{X})$ possède nécessairement une branche infinie. Comme le système $\mathcal{SS}$ est sous forme résolue, on devrait donc avoir $\mathcal{X}=\bot_{sc}$. On déduit de cette contradiction que le calcul de det(S) est fini. ❑

La relation $\vdash$ désigne la relation d'ordre définie sur $\mathcal{L}$ par $\mathcal{A}$. La propriété suivante établit qu'il est décidable de tester si deux contraintes résolues c et c' sont telles que $c \vdash c'$. Nous excluons le cas trivial où au moins l'une des deux contraintes est insatisfiable.

Propriété 8.34
Soient c = ($\mathcal{ST}$,$\mathcal{SS}$) et c' = ($\mathcal{ST}'$,$\mathcal{SS}'$) deux contraintes résolues,
$c \vdash c'$ ssi $\mathcal{ST} \leq \mathcal{ST}'$ et ($\mathcal{ST}$,glb($\overline{\mathcal{SS}}'$,$\mathcal{SS}$)) = $\bot$.

Preuve

En fait, comme l'opération de complémentation n'est définie que pour les systèmes élémentaires, il est nécessaire de coder c et c' sous une forme équivalente. On pose $V = Var_{free}(c) \cup Var_{free}(c')$ et on note $V = \{X,...,Z\}$. On considère une nouvelle variable notée P et un symbole de fonction particulier noté p. On ajoute alors à $\mathcal{ST}$ et $\mathcal{ST}'$ l'équation P=p(X,...,Z) et à $\mathcal{SS}$ et $\mathcal{SS}'$ l'équation $\mathcal{P}$=p($\mathcal{X}$,...,$\mathcal{Z}$). On obtient deux nouvelles contraintes notées d et d' et on impose $Var_{free}(d) = Var_{free}(d') = \{P\}$. Il est facile de constater que $c \vdash c'$ ssi $d \vdash d'$.

On considère ce codage implicite et on suppose donc, pour simplifier, que $Var_{free}(c)$ et $Var_{free}(c')$ désignent une seule variable commune.

Prouvons d'abord que : $\mathcal{ST} \leq \mathcal{ST}$ et $(\mathcal{ST}, glb(\overline{\mathcal{SS}'}, \mathcal{SS})) = \perp \Rightarrow c \vdash c'$

- $\mathcal{ST} \leq \mathcal{ST}$

$\Leftrightarrow sol(\mathcal{ST}) \subseteq sol(\mathcal{ST})$ (1)

- $(\mathcal{ST}, glb(\overline{\mathcal{SS}'}, \mathcal{SS})) = \perp$

$\Leftrightarrow sol(\mathcal{ST}) \cap \Phi(sol(glb(\overline{\mathcal{SS}'}, \mathcal{SS}))) = \emptyset$

$\Leftrightarrow sol(\mathcal{ST}) \cap \Phi(sol(\overline{\mathcal{SS}'}) \cap sol(\mathcal{SS})) = \emptyset$

$\Leftrightarrow sol(\mathcal{ST}) \cap \Phi(sol(\overline{\mathcal{SS}'})) \cap \Phi(sol(\mathcal{SS})) = \emptyset$ (2)

Soit θ une solution de c : $\theta \in sol(c)$ ssi $\theta \in sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS}))$

$\theta \in sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS})) \Rightarrow \theta \in sol(\mathcal{ST})$ car (1)

$\theta \in sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS})) \Rightarrow \theta \notin \Phi(sol(\overline{\mathcal{SS}'}))$ car (2) $\Rightarrow \theta \in \Phi(sol(\mathcal{SS}'))$

Finalement, $\theta \in sol(\mathcal{ST})$ et $\theta \in \Phi(sol(\mathcal{SS}')) \Rightarrow \theta \in sol(c')$. On a donc démontré que : $\mathcal{ST} \leq \mathcal{ST}$ et $(\mathcal{ST}, glb(\overline{\mathcal{SS}'}, \mathcal{SS})) = \perp \Rightarrow c \vdash c'$.

Prouvons ensuite que : $c \vdash c' \Rightarrow \mathcal{ST} \leq \mathcal{ST}$ et $(\mathcal{ST}, glb(\overline{\mathcal{SS}'}, \mathcal{SS})) = \perp$

- $c \vdash c'$

$\Rightarrow sol(c) \subseteq sol(c')$

$\Rightarrow lub(sol(c)) \leq lub(sol(c'))$

$\Rightarrow \mathcal{ST} \leq \mathcal{ST}$ car $\mathcal{ST} \approx lub(sol(c))$ et $\mathcal{ST} \approx lub(sol(c'))$

- $c \vdash c'$

$\Rightarrow sol(c) \subseteq sol(c')$

$\Rightarrow sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS})) \subseteq sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS}'))$

$\Rightarrow sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS})) \subseteq \Phi(sol(\mathcal{SS}'))$

$\Rightarrow sol(\mathcal{ST}) \cap \Phi(sol(\mathcal{SS})) \cap \Phi(sol(\overline{\mathcal{SS}'})) = \emptyset$

$\Rightarrow (\mathcal{ST}, glb(\overline{\mathcal{SS}'}, \mathcal{SS})) = \perp$ ❑

3.3 Borne inférieure et borne supérieure

On montre ci-dessous que par rapport à l'ensemble $\mathcal{A}tom(\mathcal{V},\mathcal{F},C)$ des contraintes atomiques, le calcul d'une borne inférieure (d'un plus grand minorant) correspond à la conjonction, mais que le calcul d'une borne inférieure ne correspond pas à la disjonction.

Propriété 8.35

Soient $c = (\mathcal{ST},\mathcal{SS})$ et $c' = (\mathcal{ST}',\mathcal{SS}')$ deux contraintes résolues, le plus grand minorant de c et c', noté glb(c,c'), désigne la contrainte $c'' = (\mathcal{ST}'',\mathcal{SS}'')$ où :

- $\mathcal{ST}''$ = glb($\mathcal{ST}$,$\mathcal{ST}'$)
- $\mathcal{SS}''$ = glb($\mathcal{SS}$,$\mathcal{SS}'$)

Preuve

sol(glb(c,c'))
= sol(glb($\mathcal{ST}$,$\mathcal{ST}'$)) $\cap$ Φ(sol(glb($\mathcal{SS}$,$\mathcal{SS}'$)))
= (sol($\mathcal{ST}$) $\cap$ sol($\mathcal{ST}'$)) $\cap$ (Φ(sol($\mathcal{SS}$)) $\cap$ Φ(sol($\mathcal{SS}'$)))
= sol($\mathcal{ST}$) $\cap$ Φ(sol($\mathcal{SS}$)) $\cap$ sol($\mathcal{ST}'$) $\cap$ Φ(sol($\mathcal{SS}'$))
= sol(c) $\cap$ sol(c') ❑

La preuve précédente met en évidence que le plus grand minorant de deux contraintes c et c' est bien équivalent à la conjonction $c \wedge c'$.

Propriété 8.36

Soient $c = (\mathcal{ST},\mathcal{SS})$ et $c' = (\mathcal{ST}',\mathcal{SS}')$ deux contraintes résolues, le plus petit majorant de c et c', noté lub(c,c'), désigne la contrainte $c'' = (\mathcal{ST}'',\mathcal{SS}'')$ où :

- $\mathcal{ST}''$ = lub($\mathcal{ST}$,$\mathcal{ST}'$)
- $\mathcal{SS}''$ = lub($\mathcal{SS}$,$\mathcal{SS}'$)

Preuve

sol(lub(c,c'))
= sol(lub($\mathcal{ST}$,$\mathcal{ST}'$)) $\cap$ Φ(sol(lub($\mathcal{SS}$,$\mathcal{SS}'$)))
= (sol($\mathcal{ST}$) $\cup$ sol($\mathcal{ST}'$)) $\cap$ (Φ(sol($\mathcal{SS}$)) $\cup$ Φ(sol($\mathcal{SS}'$)))
= (sol($\mathcal{ST}$) $\cap$ Φ(sol($\mathcal{SS}$))) $\cup$ (sol($\mathcal{ST}'$) $\cap$ Φ(sol($\mathcal{SS}'$)))
$\cup$ (sol($\mathcal{ST}$) $\cap$ Φ(sol($\mathcal{SS}'$))) $\cup$ (sol($\mathcal{ST}'$) $\cap$ Φ(sol($\mathcal{SS}$)))
$\subseteq$ (sol($\mathcal{ST}$) $\cap$ Φ(sol($\mathcal{SS}$))) $\cup$ (sol($\mathcal{ST}'$) $\cap$ Φ(sol($\mathcal{SS}'$)))
= sol(c) $\cup$ sol(c') ❑

La preuve précédente met en évidence que le plus grand minorant de deux contraintes c et c' n'est pas nécessairement équivalent à la disjonction $c \vee c'$. Par exemple, pour

c = ({ X=f(X',X')} , { $\mathcal{X}$=f($\mathcal{W}_1$,$\mathcal{W}_1$) , $\mathcal{W}_1$=a$\cup$b })

c' = ({ X=f(X',X") } , { $\mathcal{X}$=f($\mathcal{W}_1$,$\mathcal{W}_2$) , $\mathcal{W}_1$=a , $\mathcal{W}_2$=a$\cup$b })

on a

lub(c,c') = ({ X=f(X',X")} , { $\mathcal{X}$=f($\mathcal{W}_1$,$\mathcal{W}_1$) , $\mathcal{W}_1$=a$\cup$b })

Par rapport à X, on obtient quatre solutions distinctes pour lub(c,c') :

θ_1(X) = f(a,a)
θ_2(X) = f(a,b)
θ_3(X) = f(b,a)
θ_4(X) = f(b,b)

et seulement trois pour $c \vee c'$:

θ_1(X) = f(a,a)
θ_2(X) = f(a,b)
θ_4(X) = f(b,b)

La quantification existentielle d'une contrainte c par un ensemble de variables V $\subseteq$ $Var_{free}(c)$ désigne la contrainte $c' = c$ telle que $Var_{\exists}(c') = Var_{\exists}(c) \cup V$. L'ensemble des contraintes est donc clos par conjonction et quantification existentielle.

3.4 Opérateurs de widening

Il est clair que, pour CLP($\mathcal{FT}$+$\mathcal{SC}$), lors de la résolution, il sera nécessaire d'introduire des opérateurs de widening. Nous introduisons les opérateurs ∇_{depth}, ∇_{prox} et ∇_{card}.

On considère fixés pour les sections suivantes un entier k et un ensemble fini de variables V.

3.4.1 Opérateur ∇_{depth}

On définit les fonctions depth et abs1 à partir de celles qui ont été définies pour CLP($\mathcal{FT}$) et CLP($\mathcal{SC}$).

Soit $c = (\mathcal{S}_{ft},\mathcal{S}_{sc})$ une contrainte résolue :

depth(c) = (depth($\mathcal{ST}$),depth($\mathcal{SS}$))
abs1(c) = (abs1($\mathcal{ST}$),abs1($\mathcal{SS}$))

Lemme 8.37
L'ensemble des contraintes résolues c tels que $Var_{free}(c) \subseteq V$ et $depth(c) \leq (k,k)$, est fini.

Preuve
découle directement des lemmes 6.20 et 8.16. ❑

Lemme 8.38
Pour toute contrainte résolue c, $c \subseteq abs1(c)$.

Preuve
découle directement des lemmes 6.21 et 8.17. ❑

Lemme 8.39
L'ensemble $\{abs1(c) : c$ est une contrainte résolue et $Var_{free}(c) \subseteq V\}$ est fini.

Preuve
découle directement des lemmes 6.22 et 8.18. ❑

Propriété 8.40
∇_{depth} est un opérateur de widening

Preuve
Les lemmes 8.37, 8.38 et 8.39 vérifient les affirmations (a), (b) et (c) du schéma de preuve (section 4.1.1 du chapitre 6). ❑

3.4.2 Opérateur ∇_{prox}

Comme pour ∇_{depth}, on définit les fonctions prox et abs2 à partir de celles qui ont été définies pour CLP($\mathcal{FT}$) et CLP($\mathcal{SC}$).

Soient deux contraintes résolues $c = (\mathcal{ST},\mathcal{SS})$ et $c' = (\mathcal{ST}',\mathcal{SS}')$,

$prox(c,c') = (prox(\mathcal{ST},\mathcal{ST}'),prox(\mathcal{SS},\mathcal{SS}'))$
$abs2(c,c') = (abs2(\mathcal{ST},\mathcal{ST}'),abs2(\mathcal{SS},\mathcal{SS}'))$

Lemme 8.41
Il n'existe pas d'ensemble infini E de contraintes tel que :
- si c appartient à E alors $Var_{free}(c) \subseteq V$
- si c et c' sont deux contraintes distinctes de E alors $prox(c,c') < (k,k)$

Preuve
découle directement des lemmes 6.24 et 8.20. ❑

Lemme 8.42
Pour tout couple (c,c') de contraintes résolues,
$c \vdash \text{abs2}(c,c')$ et $c' \vdash \text{abs2}(c,c')$.

Preuve
découle directement des lemmes 6.25 et 8.21. ❑

Propriété 8.43
∇_{prox} est un opérateur de widening.

Preuve
Les lemmes 8.41, 8.42 vérifient les affirmations (a) et (b) du schéma de preuve. Pour l'affirmation (c), on utilise les lemmes 6.26 et 8.16 ❑

3.4.3 Opérateur ∇_{card}

En considérant la fonction de cardinalité et la fonction abs2 définie ci-dessus, on obtient un nouvel opérateur de widening.

Propriété 8.44
∇_{card} est un opérateur de widening.

Preuve
Le lemme 8.42 vérifie l'affirmation (b). L'affirmation (a) est trivialement vérifiée par la fonction card. Pour l'affirmation (c), on utilise les lemmes 6.26 et 8.16. ❑

3.5 Exemples d'inférence de types

Nous présentons dans cette section quelques exemples d'inférence de types qui utilisent les différents résultats obtenus précédemment. Pour les illustrations qui accompagnent ces exemples, nous nous permettons quelques abus de notations afin de ne pas surcharger les figures. En particulier, nous ne manipulons pas toujours les contraintes sous forme résolue et nous projetons au niveau de chaque noeud v les contraintes par rapport aux variables qui apparaissent dans l'étiquette de v. Les exemples proposés restent assez simples et ne mettent en oeuvre que l'opérateur ∇_{card} et, par conséquent, l'opération étoile. L'utilisation des opérateurs ∇_{depth} et ∇_{prox} n'est pas foncièrement différente ou plus difficile. Seulement, les exemples à considérer sont généralement plus complexes (voir l'annexe B).

3.5.1 Les listes d'entiers

Soit le programme P suivant :

int(X) ← X=0 ◊ .
int(X) ← X=s(Y) ◊ int(Y).
int_list(L) ← L=nil ◊ .
int_list(L) ← L=cons(X,L') ◊ int(X), int_list(L').

Soit le but G suivant :

← int_list(L)

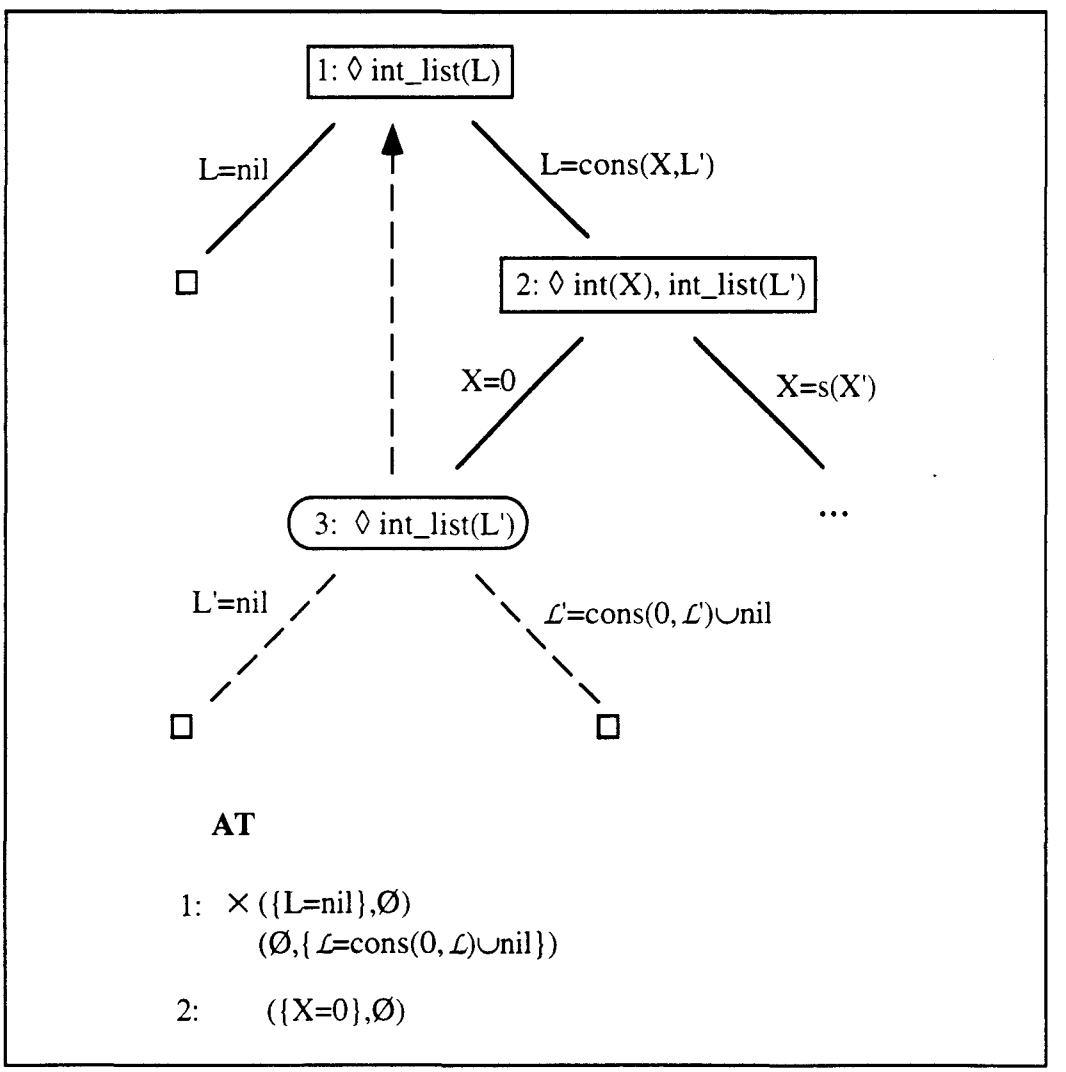

Figure 8.45

On considère que int et int_list sont des t-symboles de prédicat, que l'opérateur ∇_{card} est associé à toute liste de solutions et que ∇_{card} est paramétré par $k_{int} = 1$ et $k_{int\_list} = 1$. La résolution AOLDT de (P,G) est décrite successivement par les figures 8.45, 8.46, 8.47 et 8.48. Nous commentons cette résolution. Au niveau de la gestion de la table active, une première solution :

$c_1 = (\{L=nil\},\emptyset)$

est générée pour le noeud actif 1 et une première solution $(\{X=0\},\emptyset)$ est générée pour le noeud actif 2. Ensuite, à partir du noeud passif 3 et en utilisant la solution de list(1), on obtient une nouvelle solution

$c_2 = (\{L=cons(0,nil)\},\emptyset)$.

Comme la liste des solutions est limitée à un seul élément, on remplace c_1 dans list(1) par

$abs2(c_1,c_2) = (\emptyset,\{\mathcal{L}=cons(0,\mathcal{L})\cup nil\})$.

Cette nouvelle solution est utilisée afin de générer à partir du noeud passif 3, une nouvelle réfutation. Cependant, la solution associée à cette réfutation est moins générale que la précédente, aussi n'est-elle pas pris en compte par list(1).

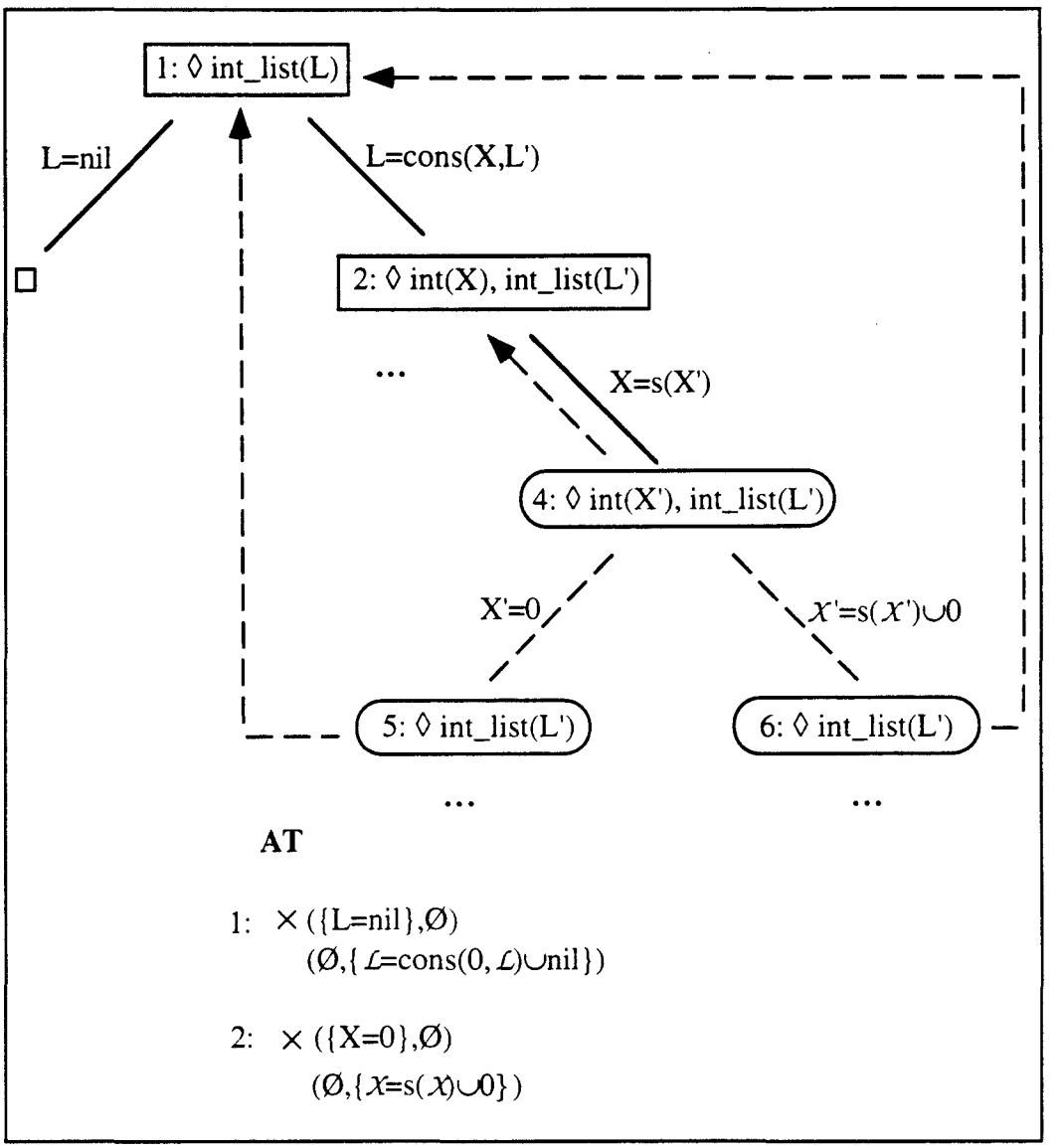

Figure 8.46

De manière similaire, à partir du noeud passif 4 et en utilisant la solution :

c_1 = ({X=0},Ø)

de list(2), on obtient une nouvelle solution

c_2 = ({X=s(0)},Ø).

Comme la liste des solutions est limitée à un seul élément, on remplace c_1 dans list(2) par

abs2(c_1,c_2) = (Ø,{X=s(X)∪0}.

Cette nouvelle solution est utilisée afin de générer à partir du noeud passif 4, une nouvelle réfutation mais la solution associée à cette réfutation est moins générale que la précédente, aussi n'est-elle pas pris en compte par list(2).

Ensuite, à partir du noeud passif 5 et en utilisant la solution

$c_1 = (\emptyset, \{\mathcal{L}=\text{cons}(0,\mathcal{L})\cup\text{nil}\})$

de list(1), on obtient une nouvelle solution :

$c_2 = (\{\text{L}=\text{cons}(\text{s}(0),\text{L}')\},\{\mathcal{L}'=\text{cons}(0,\mathcal{L}')\cup\text{nil}\})$.

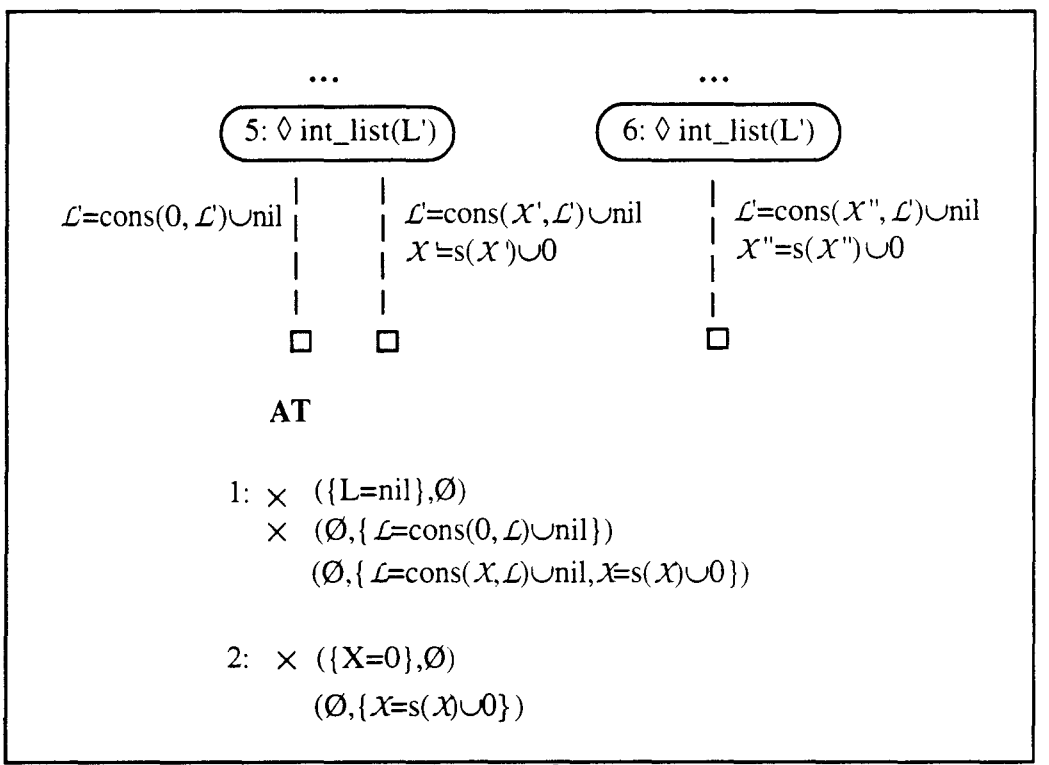

Figure 8.47

On remplace alors c_1 dans list(2) par

$\text{abs2}(c_1,c_2) = (\emptyset,\{\ \mathcal{L}=\text{cons}(\mathcal{X},\mathcal{L})\cup\text{nil}\ ,\ \mathcal{X}=\text{s}(\mathcal{X})\cup 0\ \})$.

Ce résultat est obtenu car lors du calcul de $\text{abs2}(c_1,c_2)$, on considère étoile($S_{\mathcal{L}}$) où :

$S_{\mathcal{L}}$ = { $\mathcal{L}$=cons($\mathcal{W}_1$,$\mathcal{L}$)∪nil∪cons($\mathcal{W}_2$,$\mathcal{L}$') , $\mathcal{W}_1$=0 ,
$\mathcal{L}$'=cons($\mathcal{W}_1$,$\mathcal{L}$')∪nil , $\mathcal{W}_2$=s($\mathcal{W}_1$) }†

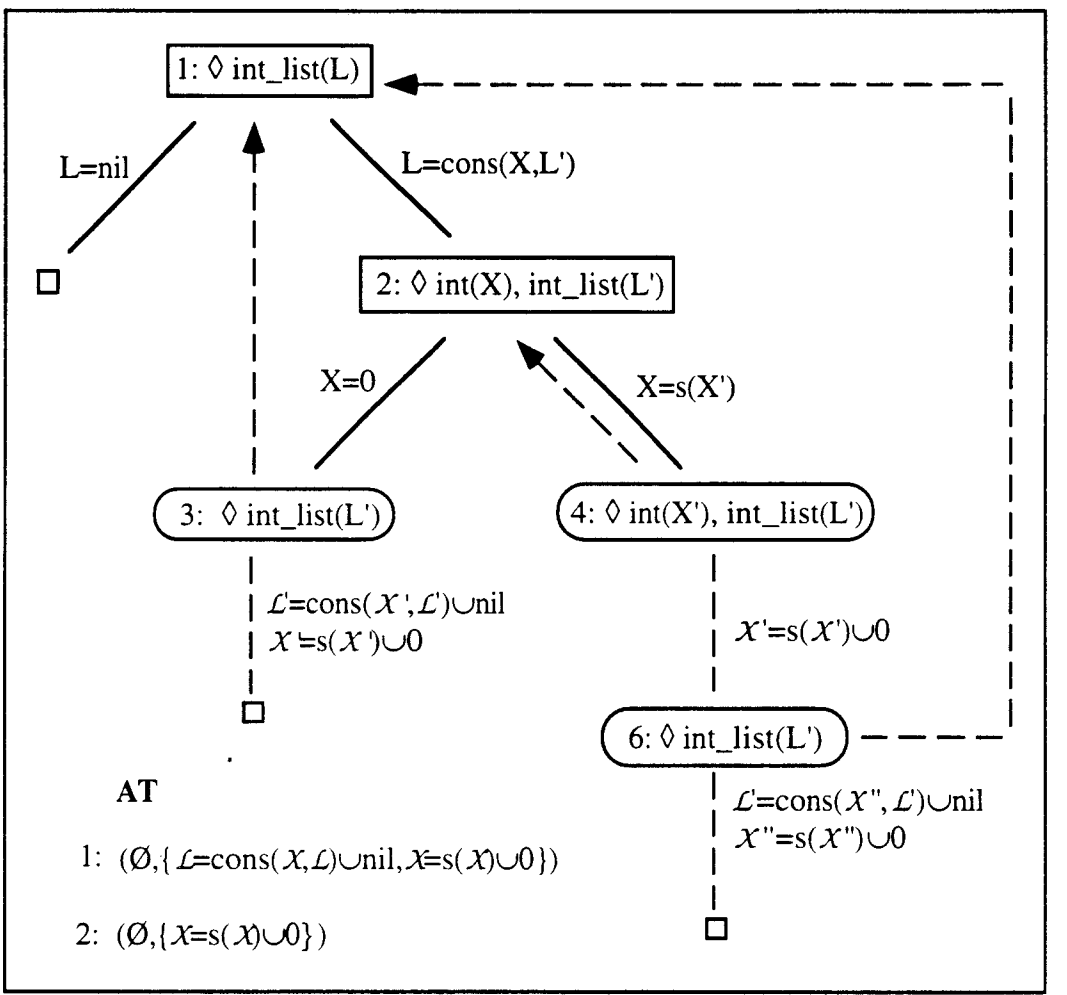

Figure 8.48

Trois récursivités potentielles sont alors détectées :

m = $cons_{(2)}$.nil et m'=nil
m=$cons_{(1)}$.0 et m'=$cons_{(1)}.s_{(1)}$.0
m=cons(1).0 et m'=cons(2).cons(1).0

aussi, obtient-on la fusion de $\mathcal{L}$' et $\mathcal{L}$ et la fusion de $\mathcal{W}_1$ et $\mathcal{W}_2$.

---

† notons que, par simplicité, nous n'avons pas effectué de renommages de variables afin déviter les partages de structure car le résultat serait le même.

La dernière solution ajoutée à list(1) est utilisée afin de générer à partir des noeuds passifs 3, 5 et 6 de nouvelles réfutations. Cependant, dans les trois cas, les solutions associées sont moins générales, aussi ne sont-elles pas prises en compte par list(1). L'étape de widening de la résolution AOLDT est alors terminée.

L'étape de narrowing consiste à supprimer les arcs inutiles du graphe OLDT obtenu. Le résultat final est décrit par la figure 8.48.

Le résultat de cette inférence de types indique deux appels atomiques et deux solutions atomiques associées :

int_list(L) $(\emptyset,\{\mathcal{L}=\text{cons}(\mathcal{X},\mathcal{L})\cup\text{nil}\ ,\ \mathcal{X}=\text{s}(\mathcal{X})\cup 0\})$
int(X) $(\emptyset,\{\mathcal{X}=\text{s}(\mathcal{X})\cup 0\})$

3.5.2 Les entiers égaux

Voici un petit exemple qui met en évidence l'intérêt des deux composantes d'une contrainte.

Soit le programme P suivant :

int_equal(X,Y) ← X=0, Y=0 ◊ .
int_equal(X,Y) ← X=s(X'), Y=s(Y') ◊ int_equal(X',Y').

Soit le but G suivant :

← int_equal(X,Y)

On considère que int_equal est un t-symbole de prédicat, que l'opérateur ∇_{card} est associé à toute liste de solutions et que ∇_{card} est paramétré par $k_{int\_equal} = 1$. Le résultat de la résolution AOLDT de (P,G) sans étape de narrowing est décrit par la figure 8.49.

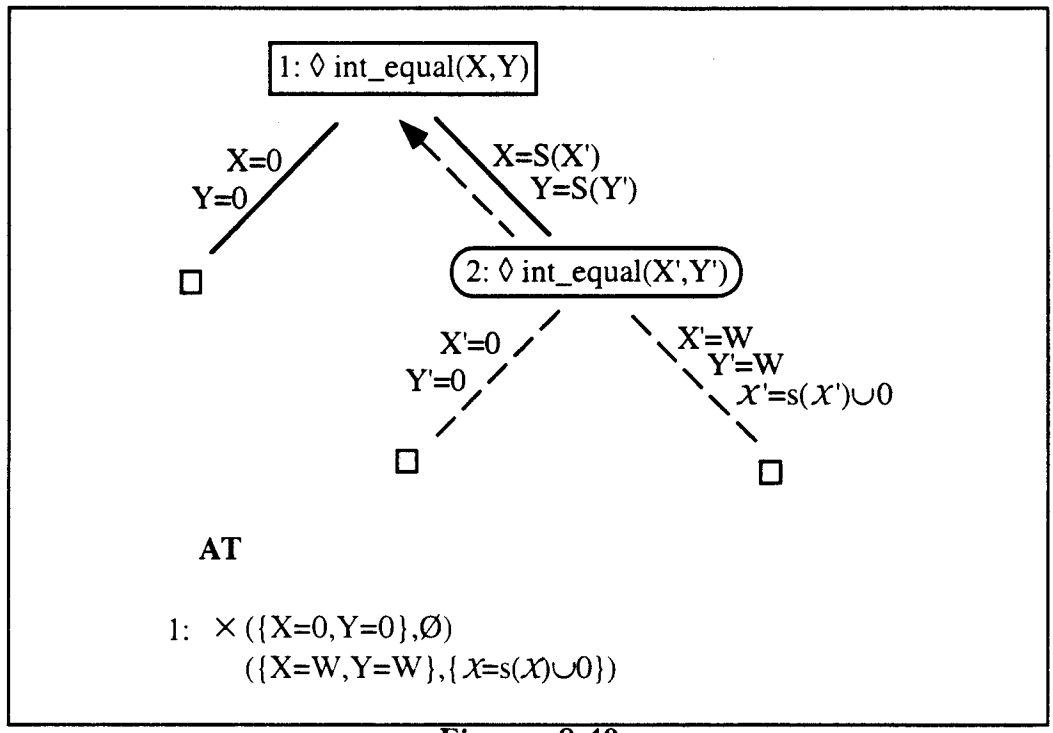

Figure 8.49

Au niveau de la gestion de la table active, une première solution :

c_1 = ({X=0,Y=0},Ø)

est générée pour le noeud actif 1. Ensuite, à partir du noeud passif 2 et en utilisant la solution de list(1), on obtient une nouvelle solution

c_2 = ({X=s(0),Y=s(0)},Ø).

Comme la liste des solutions est limitée à un seul élément, on remplace c_1 dans list(1) par

abs2(c_1,c_2) = ({X=W,Y=W},{X=s(X)∪0}).

Cette nouvelle solution est utilisée afin de générer à partir du noeud passif 2, une nouvelle réfutation. Cependant, la solution associée à cette réfutation est moins générale que la précédente, aussi n'est-elle pas pris en compte par list(1).

Le résultat obtenu met en évidence la complémentarité des deux composantes d'une contrainte. Ce résultat ne pourrait être obtenu avec une solution purement ensembliste.

3.5.3 La fonction d'Ackermann

Soit le programme P suivant :

ack(X,Y,Z).← X=0, Z=s(Y) ◊ .
ack(X,Y,Z) ← X=s(X'), Y=0, Y'=s(0) ◊ ack(X',Y',Z)
ack(X,Y,Z) ← X=s(X'), Y=s(Y') ◊ ack(X,Y',Z'), ack(X',Z',Z)

Soit le but G suivant :

← $\mathcal{X}$=s($\mathcal{X}$)∪0, $\mathcal{Y}$=s($\mathcal{Y}$)∪0 ◊ ack(X,Y,Z)

On considère que ack est un t-symbole de prédicat, que l'opérateur ∇_{card} est associé à toute liste de solutions et que ∇_{card} est paramétré par k_{ack} = 1. Le résultat de la résolution AOLDT de (P,G) est décrit par la figure 8.50.

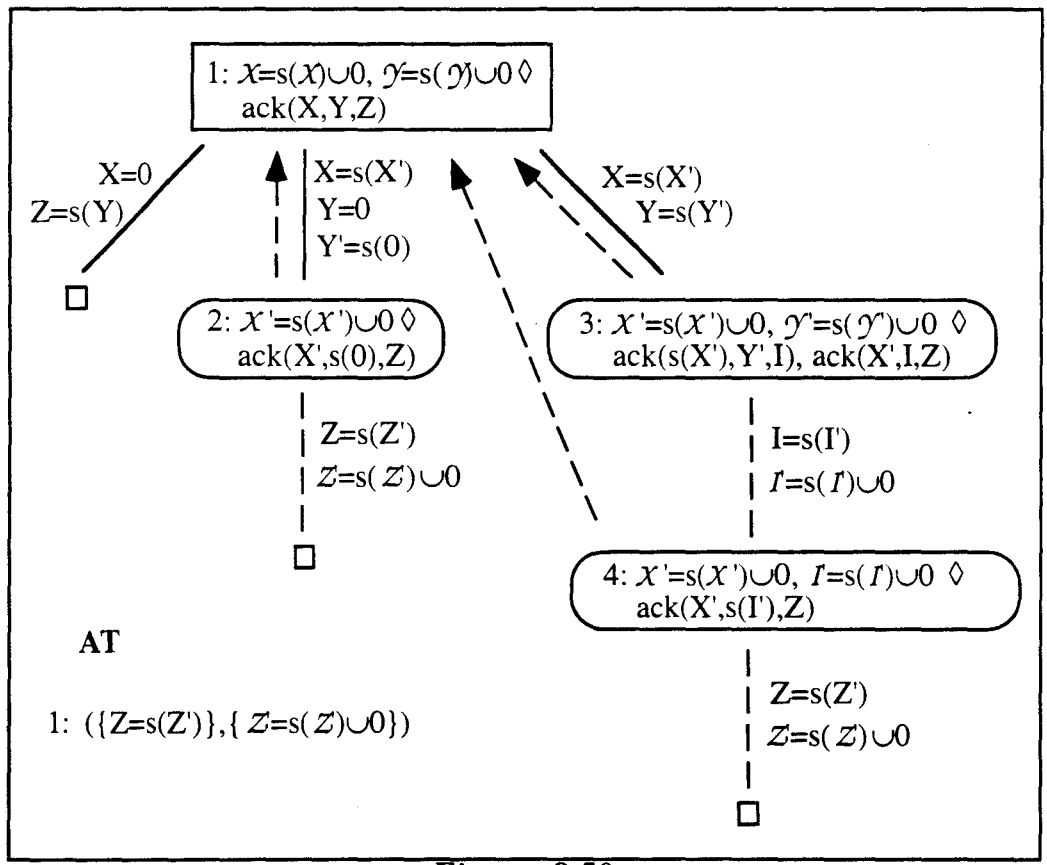

Figure 8.50

A partir d'un but où il est précisé le type des deux premiers arguments (le type des entiers naturels), on arrive à inférer le type du troisième argument, à savoir le type des entiers naturels strictement positifs. Nous reconsidérerons cet exemple dans la conclusion par rapport à l'étude de le terminaison et du déterminisme.

3.5.4 Remarques

Un opérateur de widening est un opérateur qui effectue des généralisations. Il arrive que ces généralisations soient trop importantes. Ceci n'est pas le cas pour les exemples considérés précédemments. Par contre, pour l'exemple suivant :

Soit le programme P :

p(X).← X=0 ◊ .
p(X) ← X=s(0) ◊ .

Soit le but G :

← ◊ p(X)

le résultat obtenu est $(\emptyset,\{X=s(X)\cup 0\})$ si la liste des solutions associée à p(X) est limitée à un seul élément et si l'opération étoile est utilisée. Cette généralisation est excessive ici. Cependant, il est à remarquer que si la liste des solutions associée à p(X) accepte au moins deux éléments, aucune généralisation n'est effectuée. Le problème des généralisations excessives se ramène donc en partie au problème du paramétrage des opérateurs de widening.

Il est important de noter par ailleurs que les opérateurs de widening que nous avons définis n'utilisent pas l'opération de cloture distributive. Pour l'exemple suivant :

Soit le programme P :

p(X).← X=f(a,b) ◊ .
p(X) ← X=f(b,a) ◊ .

Soit le but G :

← ◊ p(X)

le résultat obtenu est $(\emptyset,\{X=f(W_1,W_2)\cup f(W_2,W_1)\ ,\ W_1=a\ ,\ W_2=b\})$ si la liste des solutions associée à p(X) est limitée à un seul élément. Aucune perte d'information n'a lieu.

4 Discussion

Nous avons proposé un système d'inférence de types pour CLP($\mathcal{FT}$) basée sur une approche opérationnelle. Pour obtenir des analyses de programmes précises, nous avons conçu un nouveau CLP-langage noté CLP($\mathcal{FT}$+$\mathcal{SC}$) conciliant contraintes sur les termes et contraintes ensemblistes. Il est intéressant de comparer l'utilisation des contraintes ensemblistes effectuée dans notre modèle avec celle qui est effectuée dans les autres modèles.

Nombre de travaux ([Mishra 84], [Bruynooghe et al. 87], [Yardeni et Shapiro 91] et [Yardeni 92]) utilisent l'opération de cloture distributive pour limiter la complexité des contraintes ensemblistes. Pour notre part, nous n'avons pas considéré cette restriction. Il en résulte que la difficulté de mettre en place certains algorithmes est nettement accrue (voir par exemple l'algorithme de résolution de la section 3.2). Par contre, par rapport à [Heintze 92], les contraintes ensemblistes que nous manipulons sont moins générales. Nous n'autorisons pas, par exemple, l'occurence d'opérateurs de projection dans les expressions ensemblistes (mais nous nous permettons l'utilisation de ces opérateurs pour transformer une expression sous forme résolue). Nous donnons une assignation implicite aux variables qui apparaissent uniquement en partie droite d'un système d'équations ensemblistes. Ces différentes restrictions s'expliquent par le fait que nous devons gérer à la fois les contraintes sur les termes et les contraintes ensemblistes. Il n'est pas possible de comparer directement en termes de solutions ensemblistes les résultats obtenus par notre méthode et celle de [Heintze 92]. En effet, nous utilisons une approche opérationnelle tandis que [Heintze 92] utilise une approche dénotationnelle. A grossièrement parler, notre méthode est ascendante (nous contruisons une solution ensembliste à partir de rien) tandis que celle de [Heintze 92] est descendante (il construit une solution ensembliste à partir de tout).

Les résultats de [Heintze 92] sont parfaits si on ne considère comme approximation que l'ignorance des dépendances entre les variables. Mais, en pratique, ces dépendances sont importantes. C'est pourquoi, [Heintze et Jaffar 92b, Heintze 92] proposent un compromis entre l'approche dénotationnelle (analyse basée uniquement sur les contraintes ensemblistes) et l'approche de point fixe permettant la prise en compte de dépendances entre les variables. Nous pouvons donc relier ce travail avec le notre puisque dans les deux cas, il s'agit de concilier contraintes ensemblistes et information sur la dépendance entre les variables. L'idée principal de [Heintze et Jaffar 92b] est de construire des équations sémantiques à partir d'un programme puis de déplier ces équations de manière à obtenir l'ensemble des solutions du programme. Une approximation est nécessaire toutefois pour asurer la terminaison du processus de dépliage. Nous estimons que notre méthode est conceptuellement plus simple que celle de [Heintze et Jaffar 92b] car à la différence de [Heintze et Jaffar 92b] qui introduisent différentes notions telles que les groupes, clusters, ... nous conservons le même cadre formel. Quant à la précision des résultats respectifs, il est difficile d'apporter une réponse du fait de la différence des approches et de la diversité des opérateurs de widening que nous pouvons introduire.

D'autres travaux concilient directement contraintes ensemblistes et dépendances entre variables. [Uribe 92a, 92b] proposent une interprétation ensembliste permettant la prise en compte de dépendances entre différents arguments (frères) d'un même terme. Les systèmes d'équations ensemblistes interprétés de cette

façon correspondent alors aux automates d'arbres avec tests d'égalités de [Bogaert 90] et [Bogaert et Tison 92].

L'algorithme de mise sous forme résolue que nous présentons est à comparer avec l'algorithme de transformation de formules équationnelles avec contraintes d'appartenance de [Comon et Delor 90] et l'algorithme d'unification de [Uribe 92b]. La différence essentielle est que notre algorithme permet à la fois de décider de la satisfiabilité et de l'implication alors que [Comon et Delor 90] et [Uribe 92b] s'intéressent exclusivement à la satisfiabilité.

Conclusion

Dans cette thèse, nous avons proposé un modèle générique d'interprétation abstraite appliqué à la programmation logique avec contraintes. Nous avons illustré ce modèle avec un système d'inférence de types pour CLP($\mathcal{FT}$). En conclusion, nous tenons tout d'abord à reprendre les points essentiels de notre démarche en mettant en avant les similarités et les différences des divers travaux qui ont influencé celle-ci. Nous discutons ensuite de l'aptitude de notre modèle à permettre la conception d'analyses opérationnelles portant sur la terminaison, le déterminisme, ... Nous terminons en indiquant une connection possible avec l'évaluation partielle.

Travaux de référence

La base de ce travail est issue des articles de [Cousot et Cousot 77] sur l'interprétation abstraite et de [Jaffar et Lassez 86] sur la programmation logique avec contraintes. En ce qui concerne la définition de l'interprétation abstraite, nous avons choisi une caractérisation très faible (en terme de relation d'approximation) suivant en cela [Cousot et Cousot 92c] et [Marriott 93].

Le premier choix important que nous avons effectué lors de la conception de notre modèle est celui d'une approche opérationnelle. De ce point de vue, les travaux de [Tamaki et Sato 86], [Codognet et Filè 92] et [Kanamori et Kawamura 90] représentent différentes influences. En premier lieu, nous empruntons à [Tamaki et Sato 86] la technique de tabulation. En second lieu, nous utilisons le même principe opérationnel (résolution SLD + tabulation) que [Codognet et Filè 92] et [Kanamori et Kawamura 90]. Toutefois, par rapport à [Kanamori et Kawamura 90], notre modèle s'applique à la classe des langages de programmation logique avec contraintes et non simplement à Prolog. Et par rapport à [Codognet et Filè 92], nous avons choisi une notion plus faible de l'approximation (une relation d'approximation versus une fonction de concrétisation). Mais la distinction la plus importante se situe au niveau du modèle d'approximation. [Kanamori et Kawamura 90] utilisent un modèle d'approximation par combinaison, [Codognet et Filè 92] utilisent un modèle

d'approximation par abstraction, et pour notre part, nous utilisons un modèle d'approximation par extension. Cela signifie que nous choisissons, dans un premier temps, d'étendre le domaine concret en y insérant des données abstraites puis d'introduire, dans un second temps, des opérateurs de widening pour assurer la terminaison du calcul abstrait. Ceci correspond à la volonté de retarder le plus longtemps possible l'abstraction des données, ce qui constitue un gage de précision des résultats. Dans ces conditions, une interprétation abstraite est définie par une phase d'extension du domaine suivie par une phase d'abstraction du calcul. Il s'agit alors, pour l'essentiel, d'une approche de l'interprétation abstraite basée sur la combinaison d'une connexion de Galois et d'opérateurs de widening et narrowing [Cousot et Cousot 92a].

Le système d'inférence de types que nous avons présenté pour illustrer notre modèle tient à la fois de [Janssens et Bruynooghe 92] et de [Heintze 92]. Comme [Janssens et Bruynooghe 92], nous procédons de façon opérationnelle. Cependant, les contraintes ensemblistes que nous manipulons sont plus générales que les graphes de types de [Janssens et Bruynooghe 92] car nous n'appliquons pas systématiquement l'opération de clôture distributive. En outre, notre formalisme est plus simple puisque nous nous basons sur la sémantique opérationnelle d'un CLP-langage pour définir la sémantique abstraite. De son côté, [Heintze 92] procède de façon dénotationnelle pour définir l'analyse ensembliste d'un programme. Toutefois, aucune information sur la dépendance entre variables n'est permise par cette analyse. Aussi, notre système d'inférence de types est plutôt à comparer avec celui de [Heintze et Jaffar 92b].

Analyses opérationnelles

Dans le cahier des charges, nous exprimions le désir de pouvoir concevoir à la fois des analyses de flux de données et des analyses opérationnelles. Avec notre modèle, les analyses opérationnelles peuvent être définies à partir des graphes OLDT obtenus par les analyses de flux de données [Lecoutre et al. 92a,92b]. Ceci constitue une différence avec [Kanamori et al. 87a,87b] où analyses de flux de données et analyses opérationnelles sont gérées simultanément.

L'analyse de flux de données est exprimée dans [Lecoutre et al. 92a,92b] en termes de chemins au dépliage limité (ou lul pour "limited unfolding loop"). Cette notion de lul correspond en fait à une restriction des contraintes ensemblistes que nous considérons dans cette thèse. De façon très générale, il est possible de concevoir des analyses de comportement opérationnel (top-down ou bottom-up) à partir d'un graphe OLDT en suivant le principe décrit à la section 4.2 du chapitre 6. Les analyses top-down définissent des procédures de vérification (c'est à dire des procédures qui permettent de vérifier que telle ou telle propriété est vérifiée) et les analyses bottom-up définissent des procédures d'extraction (c'est à dire des procédures qui permettent d'extraire des conditions

suffisantes pour que telle ou telle propriété soit vérifiée). On sait qu'une analyse de flux de données est définie à partir d'un CLP-langage concret et d'un CLP-langage abstrait. On dit alors qu'une contrainte c est une concrétisation d'une contrainte c' ssi $c \vdash c'$ et si c est une contrainte concrète, i.e., une contrainte qui appartient au domaine du CLP-langage concret. Les analyses opérationnelles se rapportent aux concrétisations des contraintes qui apparaissent dans un graphe OLDT.

Considérons comme exemple l'analyse top-down de la terminaison. Pour prouver la terminaison d'un programme, il suffit de montrer que toute boucle du graphe OLDT correspondant à ce programme ne peut être déplié qu'un nombre fini de fois. Pour cela, on met en évidence certaines décroissances via certaines applications appelées "level mappings". Prenons l'exemple du graphe OLDT décrit à la figure 8.50. On définit comme level mapping (associé à X) l'application qui à toute concrétisation de X=s(X)$\cup$0 associe l'entier n qui est représenté de façon symbolique par cette concrétisation. On prouve facilement qu'il y a une décroissance stricte de la valeur du level mapping associé à X par rapport au dépliage de la boucle (1,2) et par rapport au dépliage de la boucle (1,4). Comme par ailleurs, la valeur du level mapping associé à X reste identique par rapport au dépliage de la boucle (1,3), on en déduit que la boucle (1,2) et la boucle (1,4) ne peuvent être dépliées qu'un nombre fini de fois. On peut alors éliminer virtuellement ces boucles du graphe OLDT avant de poursuivre l'analyse. On montre de la même façon que la valeur du level mapping associé à Y décroît strictement par rapport au dépliage de la boucle (1,3) et on en déduit que la boucle (1,3) ne peut être déplié qu'un nombre fini de fois. La terminaison du programme est par conséquent prouvé pour toute concrétisation du but. Le raisonnement précédent peut être effectué de manière automatique. Il peut sans doute aussi être généralisé en considérant des relations inter-arguments comme celles proposées par [Plumer 90] et [Verschaetse et De Schreye 91].

Considérons également comme exemple l'analyse top-down du déterminisme. Un programme est déterministe ssi il n'existe pas deux façons différentes d'obtenir un succès. Prenons de nouveau l'exemple du graphe OLDT décrit à la figure 8.50. Pour toute concrétisation du but, on montre facilement qu'il y a exclusion mutuelle entre les trois arcs attachés au noeud 1 du graphe. Comme le déterminisme est une propriété monotone et que tous les noeuds passifs du graphe sont plus spécifiques que le noeud 1, on prouve ainsi que le programme est déterministe pour toute concrétisation du but.

Beaucoup de travail reste à faire concernant les différentes analyses opérationnelles. L'idée est d'étendre les résultats de [Lecoutre et al. 92a,92b] par rapport à l'inférence de type présentée dans cette thèse, d'intégrer les relations

inter-arguments pour l'étude de la terminaison et de définir des analyses bottom-up.

Interprétation abstraite et évaluation partielle

Nous proposons avec [Parrain 94] un système qui allie transformations de programmes (évaluation partielle proprement dite) et analyse statique. Le système consiste en trois étapes : l'élimination des prédicats prédéfinis avant l'analyse statique ; l'analyse statique ; l'incorporation des informations provenant de l'analyseur dans le programme Prolog complet original et les transformations de ce programme. Les informations fournies par l'interprétation abstraite sont utilisées pour guider les transformations à appliquer sur le programme pour l'optimiser. Leur incorporation au programme permet ainsi de manipuler des règles typées pendant la phase d'évaluation partielle.

En résumé...

Nous pensons que le modèle que nous avons introduit satisfait le cahier des charges. En particulier, ce modèle nous semble :

- simple
  - conceptuellement, il ne présente aucune difficulté.
  - pratiquement, l'intégration au modèle des algorithmes restant à définir pour une application donnée est clairement précisée : algorithmes pour tester la satisfiabilité d'une contrainte, pour tester l'équivalence de deux contraintes ; algorithmes implémentant des opérateurs de widening et de narrowing.
- précis : ce modèle est basé sur un modèle d'approximation par extension.

Reste le problème de l'efficacité. Pour le moment, nous avons implémenté une version limitée du système d'inférence de types du chapitre 8 (voir l'annexe B). La complexité de toute analyse est en fait fonction des opérateurs de widening introduits et de la manière de paramétrer ceux-ci. La précision du résultat de l'analyse est alors à mettre en balance avec l'efficacité (en temps et en espace) de l'analyse.

Références

[Abstract Interpretation 87]
Abstract interpretation of declarative languages. Ellis Horwood series in computers and their applications, Abramsky S. and Hankin C. editors, Ellis Horwood, 1987.

[Aiba et al. 88] Aiba A., Sakai K., Sato Y. and Hawley D.J.
Constraint Logic Programming Language CAL. International Conference on Fifth Generation Computer System (FGCS), pages 263-276, Japan, 1988.

[Aiken et Murphy 91] Aiken A. and Murphy B.
Implementing regular trees. 5th ACM Conference on Functional Programming and Computer Architecture, LNCS 523, pages 427-447, Cambridge, August 1991.

[Aiken et Wimmers 92] Aiken A. and Wimmers E.
Solving systems of set constraints. 7th IEEE Symposium on Logic in Computer Science (LICS), pages 329-340, Santa-Cruz, June 1992.

[Aiken et al. 93] Aiken A. Kozen D. and Wimmers E.
Decidability of systems of set constraints with negative constraints. Technical Report RJ 9421, IBM, Almaden Research Center, June 1993.

[Apt 90] Apt K. R.
Logic Programming. Handbook of Theoretical Computer Science, Vol. B, Chapter 10, pages 493-574, 1990.

[Apt 92] Apt K. R.
Notes on Termination of Positive Programs. Notes of the fourth International School for Computer Science Researchers, Acireale, 1992.

[Apt et Pedreschi 90] Apt K.R. and Pedreschi D.
Studies in pure Prolog: termination. Symposium on Computational Logic, LNAI 1, pages 150-176, Brussels, November 1990.

[Bachmair et al. 92] Bachmair L., Ganzinger H. and Waldmann U.
Solving set constraints by ordered resolution with simplification. Technical Report MPI-I-92-240, Max-Planck-Institute, September 1992.

[Bachmair et al. 93] Bachmair L., Ganzinger H. and Waldmann U.
Set constraints are the monadic class. 8th IEEE Symposium on Logic in Computer Science (LICS), pages 75-83, 1993.

[Bernot et Bidoit 91] Bernot G. and Bidoit M.
Introduction aux spécifications algèbriques. Notes of the "Ecole des Jeunes Chercheurs du GRECO de Programmation", Sophia-Antipolis, April 1991.

[Bogaert 90] Bogaert B.
Automates d'arbres avec tests d'égalités. Ph.D. thesis, Université des Sciences et Technologies de Lille, Lille, December 1990.

[Bogaert et Tison 92] Bogaert B. and Tison S.
Equality and disequality constraints on direct subterms in tree automata. 9th Symposium on Theoretical Aspects of Computer Science, LNCS 577, pages 161-171, Cachan, France, February 1992.

[Boulanger et Bruynooghe 92] Boulanger D. and Bruynooghe M.
Deriving transformations of logic programs using abstract interpretation. Logic Program Synthesis and Transformation (LOPSTR), pages 99-117, Manchester, July 1992.

[Bruynooghe 82] Bruynooghe M.
Adding redundancy to obtain more reliable and more readable Prolog programs. 1st International Conference on Logic Programming (ICLP), pages 129-133, Marseille, 1982.

[Bruynooghe 91] Bruynooghe M.
A practical framework for the abstract interpretation of logic programs. Journal of Logic Programming, Vol 10, pages 91-124, February 1991.

[Bruynooghe et al. 87] Bruynooghe M., Janssens G., Callebaut A. and Demoen B.
Abstract Interpretation : towards the global optimization of Prolog programs. International Symposium on Logic Programming, pages 192-204, San Fransisco, 1987.

[Bruynooghe et De Shreye 88] Bruynooghe M. and De Schreye D.
Abstract interpretation in logic programming. Tutorial notes of the 5th International Conference and Symposium on Logic Programming, Seatle, August 1988.

[Bruynooghe et Janssens 88] Bruynooghe M. and Janssens G.
An instance of Abstract Interpretation integrating type and mode inferencing. 5th International Conference and Symposium on Logic Programming, pages 669-683, Seatle, August 1988.

[Cardelli et Wegner 85] Cardelli L. and Wegner P.
On understanding types, data abstraction and polymorphism. Computing Surveys, 17(4), pages 471-522, December 1985.

[Codish et Daemon 93] Codish M. and Daemon B.
Analysing logic programs using "Prop"-ositional logic programs and a magic wand. 1993 International Logic Programming Symposium, pages 114-129, Vancouver, 1993.

[Codognet et Filè 92] Codognet P. and Filè G.
Computations, abstractions and constraints in logic programs. International Conference on Computer Language (ICCL), San Fransisco, 1992.

[Cohen 88] Cohen J.
A view of the origins and development of Prolog. Communication of the ACM, Vol. 31-1, pages 26-36, January 1988.

[Cohen 90] Cohen J.
Constraint Logic Programming Languages. Communication of the ACM, Vol. 33-7, pages 54-68, July 1990.

[Colmerauer et al. 73] Colmerauer A., Kanoui H., Roussel P. and Pasero R.
Un système de communication Homme-Machine en Français. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[Colmerauer 82] Colmerauer A.
Prolog II : Manuel de référence et modèle théorique. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1982.

[Colmerauer 90] Colmerauer A.
An introduction to Prolog III. Communication of the ACM, Vol. 33-7, pages 70-90, July 1990.

[Corsini 89] Corsini MM.
Interprétation abstraite en programmation logique : théorie et applications. Ph.D. thesis, Université de Bordeaux I, January 1989.

[Corsini et Musumbu 93] Corsini MM. and Musumbu K.
Type inference in Prolog: a new approach . Theoritical Computer Science (TCS), Vol 119, pages 23-38, October 1993.

[Cortesi et al. 91] Cortesi A., Filè G. and Winsborough W.
Prop revisited: propositionnal formula as abstract domain for groundness analysis. IEEE symposium on logic in computer science (LICS), pages 322-327, Amsterdam, 1991.

[Cortesi et al. 94] Cortesi A., Le Charlier B. and Van Hentenryck P.
Combinations of abstract domains for logic programming. 21th ACM Symposium on Principles of Programming Languages (POPL), Portland, January 1994.

[Cortesi et Filè 91a] Cortesi A. and Filè G.
Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. ACM Symposium on partial evaluation and semantics-based program manipulation, 1991.

[Cortesi et Filè 91b] Cortesi A. and Filè G.
Abstract interpretation of Prolog: the treatment of built-ins. Internal report 11, dipartimento di mathematica pura ed applicata, universita degli studi di padova, Italy, 1991.

[Cousot 81] Cousot P.
Semantics foundations of program analysis. In Program Flow Analysis : Theory and applications, Muchnick S.S. and Jones N.D. (eds), Prentice-Hall, pages 303-342, 1981.

[Cousot et Cousot 76] Cousot P. and Cousot R.
Static determination of of dynamic properties of programs. 2nd International Symposium on Programming, Dunod, pages 106-130, 1976.

[Cousot et Cousot 77] Cousot P. and Cousot R.
Abstract Interpretation: a unified lattice for static analysis of programs by construction of approximation of fixpoints. 4th annual ACM Symposium on Principles of Programming Languages (POPL), pages 238-252, Los Angeles, 1977.

[Cousot et Cousot 79] Cousot P. and Cousot R.
Systematic design of program analysis frameworks. 6th ACM Symposium on principles of programming languages (POPL), pages 269-282, San Antonio, 1979.

[Cousot et Cousot 92a] Cousot P. and Cousot R.
Comparing the Galois Connexion and widening/narrowing approaches to abstract interpretation. Technical Report LIX/RR/92/09, LIX, Ecole Polytechnique, Palaiseau, 1992.

[Cousot et Cousot 92b] Cousot P. and Cousot R.
Abstract interpretation and applications. Journal of logic programming, pages 103-179, July 1992.

[Cousot et Cousot 92c] Cousot P. and Cousot R.
Abstract interpretation frameworks. Logic and Computation, Vol 2(4), pages 511-547, August 1992.

[Dart et Zobel 92] Dart P.W. and Zobel J.
A regular type language for logic programs. In [Types in LP 92], pages 157-187, 1992.

[Debray 92] Debray S.K.
Formal Bases for analysis and optimization of logic programs. Notes of the fourth International School for Computer Science Researchers, Acireale, Italy, 1992.

[Debray 93] Debray S.K.
Static analysis of logic programs. An advanced tutorial. International Logic Programming Symposium (ILPS), Vancouver, October 1993.

[Debray et Warren 86] Debray S.K. and Warren D.S.
Detection and Optimization of functional computation in Prolog. 3rd International Confrence on Logic Programming (ICLP), LNCS 225, Springer-Verlag, pages 490-504, London, 1986.

[Delahaye 86] Delahaye J.P.
Outils logiques pour l'intelligence artificielle. Editions Eyrolles, 1986.

[Delahaye 88] Delahaye J.P.
Sémantique logique et dénotationnelle des interpréteurs Prolog. Informatique théorique et applications. Vol 22-1, pages 3-42, 1998.

[De Schreye et al. 90] De Schreye D., Verschaetse K. and Bruynooghe M.
A practical technique for detecting non-terminating queries for a restricted class of Horn clauses using directed, weighted graphs. Report CW-109, March 1990.

[Devienne 88] Devienne P.
Weighted Graphs: a tool for expressing the behaviour of recursive rules in Logic Programming. FGCS conferences, November 1988.

[Devienne et al. 90a] Devienne P., Lebègue P. and Dauchet M.
Weighted Systems of Equations. LIFL Technical report IT-188, May 1990.

[Devienne et al. 90b] Devienne P., Lebègue P. and Dauchet M.
Weighted Graphs, a tool for studying the halting problem and time complexity in term rewriting systems and Logic Programming. Theoretical Computer Science (TCS), December 1990.

[Dincbas et al. 88] Dincbas M., Van Hentenryck P., Simonis H., Aggoun A. Graf T...
... and Berthier F. *The constraint logic programming language CHIP.* International Conference on fifth Generation Computer System (FGCS), pages 693-702, Japan, 1988.

[Eder 85] Eder E.
Properties of substitutions and unifications. Journal of Symbolic Computation 1, pages 31-46, 1985.

[Falaschi et al. 89] Falaschi M., Levi G., Palamidessi C. and Martelli M.
Declarative modelling of the operational behaviour of logic programs. Theoritical Computer Science (TCS), Vol. 69, pages 289-318, 1989.

[Filè et Sottero 91] Filè G. and Sottero P.
Abstract interpretation for type checking. 3rd Conference on Programming Languages Implementation and Logic Programming (PLILP), pages 311-322, Passau, 1991.

[Fruhwirth 90] Fruhwirth T.
Using meta-interpreters for polymorphic type checking. 2nd Workshop on meta-programming in logic (META), pages 339-351, Leuven, 1990.

[Fruhwirth et al. 92] Fruhwirth T., Herold A., Kuchenhoff V., Le Provost T...
...Lim P., Monfroy E. and Wallace M. *Constraint logic programming - an informal introduction.* 2nd International logic programming summer school, LNAI 636, pages 3-35, 1992.

[Gabbrielli et Levi 92] Gabbrielli M. and Levi G.
Modeling answer constraints in constraint logic programs. Joint International Conference and Symposium on Logic Programming, pages 238-252, Washington, 1992.

[Gallagher et al. 88] Gallagher J., Codish M. and Shapiro E.
Specialisation of Prolog and FCP Programs using Abstract Interpretation. New Generation Computing 6, pages 159-186, 1988.

[Gécseg et Steinby 84] Gécseg F. and Steinby M.
Tree Automata. Akadémiai Kiado, Budapest, Hungary, 1984.

[Gilleron et al. 92] Gilleron R., Tison S. and Tommasi M.
Solving systems of set constraints using tree automata. Technical Report IT-235, Lille, July 1992 (also in STACS'93, LNCS 665).

[Gilleron et al. 93] Gilleron R., Tison S. and Tommasi M.
Solving systems of set constraints with negated subset relationships. Technical Report IT-247, Lille, March 1993 (also in 34th FOCS).

[Gloess 90] Gloess P.Y.
Contribution a l'optimisation de mecanismes de raisonnement dans les structures specialisees de representation des connaissances. Thèse d'état. Compiegne. 1990.

[Heintze 91] Heintze N.
Set-based program analysis. Thesis proposal, Carnegie Mellon University, March 1991.

[Heintze 92a] Heintze N.
Set-based program analysis. Ph.D. thesis, Carnegie Mellon University, October 1992.

[Heintze 92b] Heintze N.
Practical aspects of set-based analysis. Joint International Conference and Symposium on Logic Programming. pages 765-779, Washington, 1992.

[Heintze et Jaffar 90] Heintze N. and Jaffar J.
A finite presentation theorem for approximating logic programs. IBM technical report RC 16089 (# 71415), August 1990 (also in 17th ACM-POPL).

[Heintze et Jaffar 91] Heintze N. and Jaffar J.
A decision procedure for a class of Herbrand set constraint. Carnegie Mellon University technical report CMU-CS-91-110, February 1991 (also in 5th IEEE LICS).

[Heintze et Jaffar 92a] Heintze N. and Jaffar J.
Semantics types for logic programs. In "Types in logic programming", Pfenning F. (Ed), MIT Press, pages 141-155, 1992.

[Heintze et Jaffar 92b] Heintze N. and Jaffar J.
An engine for logic program analysis. 7th IEEE Symposium on Logic in Computer Science (LICS), pages 318-328, Santa Cruz, June 1992.

[Hermenegildo 92] Hermenegildo M.
Abstract Interpretation and its applications. Advanced Tutorial on abstract interpretation of the Joint International Conference and Symposium on Logic Programming, Washington, 1992.

[Hermenegildo et al. 92] Hermenegildo M., Warren R. and Debray S.K.
Global flow analysis as a practical compilation tool. Journal of logic programming, vol. 13(4), pages 349-366, August 1992.

[Höfeld et Smolka 88] Höfeld M. and Smolka G.
Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, October 1988.

[Huet 76] Huet G.
Resolution d'équations dans les langages d'ordre 1, 2, ...,ω. Thèse d'état, Université Paris VI, 1976.

[Horiuchi et Kanamori 87] Horiuchi K. and Kanamori T.
Polymorphic type inference in Prolog by abstract interpretation. Logic Programming Conference, pages 107-116, Tokyo, 1987.

[Jaffar et Lassez 86] Jaffar J. and Lassez J.L.
Constraint Logic Programming. Technical Report 86/73, Monash University, Victoria, Australia, June 1986 (also in 14th ACM POPL, pages 111-119, Munich, January 1987).

[Jaffar et Lassez 87] Jaffar J. and Lassez J.L.
From unification to constraints. 6th Japanese Logic Programming Conference, LNCS 315, pages 1-18, Tokyo, June 1987.

[Jaffar et Michaylov 87] Jaffar J. and Michaylov S.
Methodology and implementation of a CLP system. 4th International Conference on Logic Programming, pages 196-218, 1987.

[Janssens et Bruynooghe 92] Janssens G. and Bruynooghe M.
Deriving descriptions of possible values of program variables by means of abstract interpretation. Journal of logic programming, pages 205-258, July 1992.

[Jones et Sondergaard 87] Jones N. and Sondergaard H.
A semantics-based framework for the abstract interpretation of Prolog. In Abstract interpretation of declarative languages, S. Abramsky and C. Hankin (Eds), Ellis Horwood, 1987.

[Journal of LP 92] Journal of logic programming.
Special issue: abstract interpretation. July 1992.

[Kanamori et al. 87a] Kanamori T., Kawamura T. and Horiuchi K.
Detecting termination of logic programs based on abstract hybrid interpretation. ICOT Technical Report TR-398, Tokyo, 1987.

[Kanamori et al. 87b] Kanamori T., Kawamura T. and Horiuchi K.
Detecting functionality of logic programs based on abstract hybrid interpretation. ICOT Technical Report TR-331, Tokyo, 1987.

[Kanamori et Kawamura 87] Kanamori T. and Kawamura T.
Analysing success patterns of logic programs by abstract hybrid interpretation. ICOT Technical Report, 1987.

[Kanamori et Kawamura 90] Kanamori T. and Kawamura T.
Abstract interpretation based on OLDT resolution. ICOT Technical report,Tokyo, 1990.

[Kirchner 85] Kirchner H.
Preuves par completion dans les varietes d'algebres. Thèse de doctorat d'état en informatique, Nancy, France, 1985.

[Kowalski 74] Kowalski R.
Predicate logic as a programming language. 6th IFIP congress, pages 569-574, North Holland, 1974.

[Kowalski 88] Kowalski R.
The early years of logic programming. Communication of the ACM, Vol. 31-1, pages 38-43, 1988.

[Jones et Muchnick 79] Jones N.D. and Muchnick S.S.
Flow analysis and optimisation of Lisp-like structures. 6th ACM Symposium on Principles of Programming Languages, pages 244-246, 1979.

[Lassez et al. 87] Lassez J.L., Maher M. and Marriot K.G.
Unification revisited. In Foundations of Deductive Databases and Logic Programing, Minker J. (eds), Morgan-Kauffman, pages 587-625, 1988.

[Lebegue 88] Lebègue P.
Contribution à l'étude de la Programmation Logique par les Graphes Orientés Pondérés. Ph.D. thesis, Université de LILLE I, Nov 1988.

[Le Charlier et al. 91] Le Charlier B. Musumbu K. and Van Hentenryck P.
A generic abstract interpretation algorithm and its complexity analysis. 8th International Conference on Logic Programming (ICLP), pages 64-78, Paris, June 1991.

[Le Charlier et Van Hentenryck 92a] Le Charlier B. and Van Hentenryck P.
A universal top-down fixpoint algorithm. Technical report 20/92, Institute of computer science, university of Namur, Belgium, April 1992.

[Le Charlier et Van Hentenryck 92b] Le Charlier B. and Van Hentenryck P.
On the design of generic abstract interpretation frameworks. Workshop on Static Analysis (WSA), pages 229-246, Bordeaux, September 1992.

[Le Charlier et Van Hentenryck 92c] Le Charlier B. and Van Hentenryck P.
Reexecution in abstract interpretation of Prolog. International Joint Conference and Symposium on Logic Programming (IJCSLP), pages 750-764, Washington, November 1992.

[Le Charlier et Van Hentenryck 93] Le Charlier B. and Van Hentenryck P.
A generic fixpoint semantics for logic programming and its application for abstract interpretation. Technical report, CS department, Brown university, 1993.

[Lecoutre et al. 91] Lecoutre C., Devienne P. and Lebegue P.
Abstract Interpretation and recursive behaviour of logic programs. In Logic program synthesis and transformation, T.P. Clement and K.-K Lau (Eds), Springer-Verlag, Manchester, July 1991.

[Lecoutre et al. 92a] Lecoutre C., Devienne P. and Lebegue P.
Termination induction by means of an abstract OLDT resolution. 1ère journées francophones sur la programmation logique, pages 353-373, Lille, May 1992.

[Lecoutre et al. 92b] Lecoutre C., Devienne P. and Lebegue P.
Determinacy induction by means of an abstract OLDT resolution. Worshop on Static Analysis (WSA), Bordeaux, September 1992.

[Lloyd 87] Lloyd J.
Foundations of Logic Programming. Series in symbolic of computation, Springer Verlag, 1987.

[Maher 87] Maher M.
Logic semantics for a class of commited-choice programs. 4th International Conference on Logic Programming (ICLP), pages 858-876, Melbourne, May 1987.

[Maher 92] Maher M.
Lecture Notes on Constraint Logic Programming: Analysis, Transformation and Semantics. Notes of the 4th International School for Computer Science Researchers, Acireale, 1992.

[Mallet 92] Mallet O.
Interprétation abstraite appliquée à la compilation et la parallélisation en programmation logique. Ph.D. thesis, Paris, 1992.

[Marrien et al. 89] Marien A., Janssens G., Mulkers A. and Bruynooghe M.
The impact of abstract interpretation : an experiment in code generation. 6th International Conference on Logic Programming (ICLP), pages 33-47, Lisbon, Portugal, 1989.

[Marriott 93] Marriott K.
Frameworks for abstract interpretation. Acta Informatica 30, pages 103-129, 1993.

[Marriott et Sondergaard 88] Marriott K. and Sondergaard H.
Bottom-up abstract interpretation of logic programs. 5th International Conference on Logic Programming (ICLP), pages 733-748, Seattle, August 1988.

[Marriott et Sondergaard 89a] Marriott K. and Sondergaard H.
A tutorial on abstract interpretation of logic programs. Tutorial notes of the 1989 North American Conference on logic programming (NACLP). 1989.

[Marriott et Sondergaard 89b] Marriott K. and Sondergaard H.
Semantics-based dataflow analysis of logic programs. Information Processing 89, pages 601-606, North-Holland, 1989.

[Marriott et Sondergaard 90a] Marriott K. and Sondergaard H.
Abstract interpretation of logic programs: the denotational approach. GULP, pages 399-424, Italy., 1990.

[Marriott et Sondergaard 90b] Marriott K. and Sondergaard H.
Analysis of constraint logic programs. North American Conference on logic programming (NACLP), pages 531-547, Austin, 1990.

[Marriott et Sondergaard 92] Marriott K. and Sondergaard H.
Bottom-up data-flow analysis of normal logic programs. Journal of logic programming, pages 181-204, July 1992.

[Martelli et Montanari 82] Martelli A. and Montanari U.
An efficient unification algorithm. ACM TOPLAS, vol.4(1), 1982.

[Mellish 85] Mellish CS.
Some global optimizations for a Prolog compiler. Journal of logic programming 1, pages 43-66, 1985.

[Mellish 87] Mellish CS.
Abstract Interpretation of Prolog programs. In Abstract interpretation of declarative languages, Abramsky S. and Hankin C. (Eds), Ellis Horwood, 1987.

[Melton et al. 86] Melton A., Scmidt D.A. and Strecker G.E.
Galois Connections and computer science applications. Lecture notes in computer science 240, Springer-Verlag, pages 299-312, 1986.

[Mesnard 93] Mesnard F.
Etude de la terminaison des programmes logiques avec contraintes, au moyen d'approximation. Ph.D. thesis, Paris, 1993.

[Mesnard et Ganascia 92] Mesnard F. and Ganascia J.G.
CLP(X) for proving program properties. 1ère journées francophones sur la programmation logique, pages 328-338, Lille, May 1992.

[Miller et Nadathur 86] Miller D.A. et Nadathur G.
Higher-order logic programming. 3rd International Conference on Logic Programming (ICLP), pages 448-462, London, 1986.

[Milner 78] Milner R.
A theory of type polymorphism in programming. Journal of computer and system sciences, Vol 17(3), pages 348-375, 1978.

[Mishra 84] Mishra P.
Towards a theory of types in Prolog. 1st IEEE Symposium on Logic Programming, pages 456-461, Atlantic City, 1984.

[Mishra et Reddy 85] Mishra P. and Reddy U.
Declaration-free type checking. 12th ACM Symposium on Principles of Programming Languages (POPL), pages 7-21, New Orleans, January 1985.

[Musumbu 90] Musumbu K.
Interprétation abstraite des programmes Prolog. Ph.D. thesis, Institut d'Informatique, F.U.N.D.P, Namur, September 1990.

[Mycroft et O'Keefe 84] Mycroft A. and O'Keefe R.
A polymorphic type system for Prolog. Artificial Intelligence, Vol. 23, pages 295-307, 1984.

[Nielson 82] Nielson F.
A denotationnal framework for data-flow analysis. In Acta Informatica 18, pages 265-287, 1982.

[Parrain 94] Parrain A.
Transformations de programmes logiques et sémantique opérationnelle. Ph.D. thesis, Lille, February 1994.

[Plümer 90] Plümer L.
Termination proofs for logic programs. LNCS 446, 1990.

[Raynolds 69] Raynolds J.C.
Automatic computation of data set definition. Information Processing, Vol. 68, pages 456-461, 1969.

[Robinson 65] Robinson J.A.
A machine-oriented logic based on the resolution principle. Journal of the ACM, Vol 12-1, pages 23-41, 1965.

[Robinson 92] Robinson J.A.
Logic and logic programming. Communication of the ACM, Vol. 35-3, pages 40-65, March 1992.

[Saraswat et Rinhard 90] Saraswat V.A. and Rinard M.
Concurrent constraint programming. 7th ACM Symposium on Principles of Programming Languages (POPL), pages 232-245, New-York, 1990.

[Smolka 89] Smolka G.
Logic programming over polymorphically order-sorted types. Ph.D. thesis, Universität Kaiserslautern, W. Germany, May 1989.

[Solnon 93] Solnon C.
Un système d'inférence de relations intertypes pour Prolog. Ph.D. thesis, Université de Nice - Sophia Antipolis, February 1993.

[Tamaki et Sato 84] Tamaki H. and Sato T.
Enumeration of success patterns in logic programs. Theoretical Computer Science 34, pages 227-240, 1984.

[Tamaki et Sato 86] Tamaki H. and Sato T.
OLD resolution with Tabulation. 3rd International Conference on Logic Programming (ICLP), LNCS 225, Springer-Verlag, pages 84-98, London, 1986.

[Types in LP 92] Types in Logic Programming
Logic Programming series, Frank Pfenning (Editor), MIT Press, 1992.

[Uribe 92a] Uribe T.E.
Sorted unification using set constraints. 11th International Conference on Automated Deduction, New-York, 1992.

[Uribe 92b] Uribe T.E.
Sorted unification and the solution of semi-linear membership constraints. Ph.D. thesis, University of Illinois, Urbana, Illinois, 11th International Conference on Automated Deduction, New-York, 1992.

[van Emdem et Kowalski 76] van Emden M. and Kowalski R.
The semantics of logic as a programming language. Journal of the ACM, Vol 23(4), pages 733-742, October 1976.

[Van Hentenryck 89] Van Hentenryck P.
Constraint satisfaction in logic programming. Logic programming series, MIT Press, Cambridge, 1989.

[Van Hentenryck et al. 93a] Van Hentenryck P., Dagimbe O., Le Charlier B. ...
... and Michel L. *Abstrcat Interpretation of Prolog based on OLDT-resolution* Technical Report CS-93-05, Brown University, February 1993.

[Van Hentenryck et al. 93b] Van Hentenryck P., Dagimbe O., Le Charlier B. ...
... and Michel L. *The impact of granularity in abstract interpretation of Prolog.* Worshop of Static Analysis (WSA), pages 1-14, Padova, September 1993.

[Verschaetse et De Schreye 91] Verschaetse K. and De Schreye D.
Deriving termination proofs for logic programs using abstract procedures. 8th International Conference on Logic Programming (ICLP), pages 301-315, Paris, June 1991.

[Warren 92] Warren D.S.
Memoing for logic programs. Communication of the ACM, Vol. 35(3), pages 93-111, March 1992.

[Yardeni et al. 92] Yardeni E., Frühwirth T. and Shapiro E.
Polymorphically typed logic programs. In "Types in logic programming", Pfenning F. (Ed), MIT Press, pages 63-90, 1992.

[Yardeni et Shapiro 91] Yardeni E. and Shapiro E.
A type system for logic programs. Journal of logic programming, Vol 10, pages 125-153, February 1991.

Annexe A

Préliminaires

Dans cette annexe, nous rappelons quelques notions, définitions et propriétés essentielles concernant les algèbres (section 1), les ensembles ordonnés (section 2) et les connexions de Galois (section 3). Nous en profitons également pour fixer quelques notations.

1 Algèbres

La notion d'algèbre est utilisée en programmation logique. Pour cette courte présentation, nous nous sommes inspirés essentiellement de [Bernot et Bidoit 91] et du chapitre introductif de [Kirchner 85].

Soit $\mathcal{F}$ un ensemble dénombrable de symboles de fonction. A chaque symbole f de $\mathcal{F}$, on peut associer un entier appelé arité de f. On partitionne alors $\mathcal{F}$ en une réunion de sous-ensembles $\mathcal{F}_i$ où $\mathcal{F}_i$ est l'ensemble des symboles de fonction d'arité i. Par convention, les symboles de fonction d'arité 0 sont appelés constantes et notés a,b,c... Ceux d'arité non nulle sont désignés par les lettres f,g,h...

Soit S un ensemble de noms appelés sortes, $S = \{s_1,...,s_n\}$. La signature sur S d'un symbole de fonction d'arité n est une séquence de n+1 éléments de S. On note généralement un symbole de fonction f muni d'une signature $(s_1,...,s_n,s)$ par $f : s_1 \times ... \times s_n \rightarrow s$ ou encore $f_{s_1 \times ... \times s_n \rightarrow s}$. Intuitivement, les sortes à gauche de la flèche sont les types des arguments de la fonction et l'unique sorte à droite de la flèche est le type du résultat retourné par la fonction.

Définition A.1

Une signature $\Sigma = (S, \mathcal{F})$ est définie par un ensemble S de sortes et par un ensemble $\mathcal{F}$ de symboles de fonction munis d'une signature sur S.

Une signature est une donnée purement syntaxique. La sémantique associée à une signature $\Sigma = (S, \mathcal{F})$ est définie via la notion de Σ-algèbre (hétérogène) et consiste à interpréter chaque sorte et chaque symbole de fonction.

Définition A.2

Soit $\Sigma = (S, \mathcal{F})$ une signature, une Σ-algèbre $\mathcal{A}$ est définie par :

- l'interprétation de chaque sorte s de S par un ensemble (non vide) noté $\mathcal{A}(s)$.
- l'interprétation de chaque symbole de fonction $f : s_1 \times ... \times s_n \rightarrow s$ de $\mathcal{F}_n$ par une application notée $\mathcal{A}(f)$ et définie de $\mathcal{A}(s_1) \times ... \times \mathcal{A}(s_n)$ vers $\mathcal{A}(s)$.

L'union de l'interprétation de chaque sorte constitue le domaine de la Σ-algèbre. On considère maintenant l'algèbre des termes clos et l'algèbre des termes libres.

Définition A.3

L'algèbre des termes clos est la Σ-algèbre $\mathcal{A}$ telle que :

- les sortes sont interprétées de manière inductive par :

  $\forall\, a : \rightarrow s \in \mathcal{F}_0$,
  $a \in \mathcal{A}(s)$

  $\forall\, f : s_1 \times ... \times s_n \rightarrow s \in \mathcal{F}_n$, $\forall\, (t_1,...,t_n) \in \mathcal{A}(s_1) \times ... \times \mathcal{A}(s_n)$,
  $f(t_1,...,t_n) \in \mathcal{A}(s)$
- l'interprétation de chaque symbole de fonction $f : s_1 \times ... \times s_n \rightarrow s$ de $\mathcal{F}_n$ est donnée par l'application $\mathcal{A}(f)$ qui à tout n-uplet $(t_1,...,t_n)$ de $\mathcal{A}(s_1) \times ... \times \mathcal{A}(s_n)$ associe le terme $f(t_1,...,t_n)$.

Soit $\mathcal{V}$ un ensemble infini dénombrable de symboles de variables. L'arité d'un symbole de variable est nulle. Les symboles de variable sont notés par X,Y,Z... La signature sur S d'un symbole de variable de $\mathcal{V}$ est un élément de S. On note généralement un symbole de variable X muni d'une signature s par $X : \rightarrow s$ ou encore X_s.

Définition A.4

L'algèbre des termes libres est la Σ-algèbre $\mathcal{A}$ telle que :

- les sortes sont interprétées de manière inductive par :

  $\forall\, X : \rightarrow s \in \mathcal{V}$,
  $X \in \mathcal{A}(s)$

  $\forall\, a : \rightarrow s \in \mathcal{F}_0$,
  $a \in \mathcal{DA}(s)$

  $\forall\, f : s_1 \times ... \times s_n \rightarrow s \in \mathcal{F}_n$, $\forall\, (t_1,...,t_n) \in \mathcal{A}(s_1) \times ... \times \mathcal{A}(s_n)$,
  $f(t_1,...,t_n) \in \mathcal{A}(s)$
- l'interprétation de chaque symbole de fonction $f : s_1 \times ... \times s_n \rightarrow s$ de $\mathcal{F}_n$ est donnée par l'application $\mathcal{A}(f)$ qui à tout n-uplet $(t_1,...,t_n)$ de $\mathcal{A}(s_1) \times ... \times \mathcal{A}(s_n)$ associe le terme $f(t_1,...,t_n)$.

L'algèbre des termes libres est notée $Term(S,\mathcal{V},\mathcal{F})$ et l'algèbre des termes clos est notée $Term(S,\mathcal{F})$. Les éléments (termes) de $Term(S,\mathcal{V},\mathcal{F})$ et $Term(S,\mathcal{F})$ sont notés $t,s,...$On confond souvent par la suite les algèbres $Term(S,\mathcal{F})$ et $Term(S,\mathcal{V},\mathcal{F})$ avec le domaine sur lequel elles sont définies.

Une Σ-algèbre homogène (ou mono-sortée) est une Σ-algèbre hétérogène (ou multi-sortée) basée sur un ensemble de sortes S ne contenant qu'un seul élément. La signature des symboles est de ce fait triviale. Dans ce cas, on note les algèbres de termes clos et libres par $Term(\mathcal{F})$ et $Term(\mathcal{V},\mathcal{F})$.

Soit $\mathcal{P}$ un ensemble dénombrable de symboles de prédicat. Les symboles de prédicat sont en fait des symboles de fonction particuliers. On partitionne $\mathcal{P}$ en une réunion de sous-ensembles $\mathcal{P}_i$ où $\mathcal{P}_i$ est l'ensemble des symboles de fonction d'arité i. Par convention, les symboles de prédicat sont désignés par les lettres p,q,r... La signature sur S d'un symbole de prédicat d'arité n est une séquence de n éléments de S. On note généralement un symbole de prédicat p muni d'une signature $(s_1,...,s_n)$ par $p : s_1\times...\times s_n$ ou encore $p_{s_1\times...\times s_n}$. Intuitivement, la signature d'un symbole de prédicat indique le type des arguments du prédicat et le résultat retourné par le prédicat est implicitement du type booléen $\mathbb{B} = \{faux,vrai\}$.

On étend la définition d'une signature, puis celle d'une algèbre, en considérant un ensemble dénombrable de symboles de prédicat.

Définition A.5

Une signature $\Sigma = (S,\mathcal{F},\mathcal{P})$ est définie par un ensemble de sortes S, un ensemble dénombrable $\mathcal{F}$ de symboles de fonction munis d'une signature sur S et par un ensemble dénombrable $\mathcal{P}$ de symboles de prédicat munis d'une signature sur S.

Définition A.6

Soit $\Sigma = (S,\mathcal{F},\mathcal{P})$ une signature, une Σ-algèbre $\mathcal{A}$ est définie comme suit :

- les sortes sont interprétées de manière inductive par :
  $\forall\, a : \rightarrow s \in \mathcal{F}_0$,
  $a \in \mathcal{A}(s)$
  $\forall\, f : s_1\times...\times s_n \rightarrow s \in \mathcal{F}_n$, $\forall\, (t_1,...,t_n) \in \mathcal{A}(s_1)\times...\times\mathcal{A}(s_n)$,
  $f(t_1,...,t_n) \in \mathcal{A}(s)$
- l'interprétation de chaque symbole de fonction $f : s_1\times...\times s_n \rightarrow s$ de $\mathcal{F}_n$ est donnée par une application $\mathcal{A}(f)$ qui à tout n-uplet $(t_1,...,t_n)$ de $\mathcal{A}(s_1)\times...\times\mathcal{A}(s_n)$ associe $f(t_1,...,t_n)$.
- l'interprétation de chaque symbole de prédicat $p : s_1\times...\times s_n$ de $\mathcal{P}_n$ par une application $\mathcal{A}(p)$ définie de $\mathcal{A}(s_1)\times...\times\mathcal{A}(s_n)$ vers $\mathbb{B}$.

Pour finir, on définit les deux ensembles suivants :

- $\mathcal{A}tom(S,\mathcal{F},\mathcal{P}) = \{p(t_1,...,t_n) \mid p \in \mathcal{P}_n \text{ et } (t_1,...,t_n) \in \mathcal{T}erm(S,\mathcal{F})\}$
- $\mathcal{A}tom(S,\mathcal{V},\mathcal{F},\mathcal{P}) = \{p(t_1,...,t_n) \mid p \in \mathcal{P}_n \text{ et } (t_1,...,t_n) \in \mathcal{T}erm(S,\mathcal{V},\mathcal{F})\}$

Tout élément de $\mathcal{A}tom(S,\mathcal{F},\mathcal{P})$ est appelé un atome clos et tout élément de $\mathcal{A}tom(S,\mathcal{V},\mathcal{F},\mathcal{P})$ est appelé un atome libre. Les éléments (atomes) de $\mathcal{A}tom(S,\mathcal{F},\mathcal{P})$ et $\mathcal{A}tom(S,\mathcal{V},\mathcal{F},\mathcal{P})$ sont notés $a,b,h,...$ Lorsque l'ensemble S est réduit à un seul élément, on note ces ensembles par : $\mathcal{A}tom(\mathcal{F},\mathcal{P})$ et $\mathcal{A}tom(\mathcal{V},\mathcal{F},\mathcal{P})$.

2 Ensembles ordonnés

Nous présentons maintenant quelques concepts élémentaires empruntés à la théorie des ensembles. On pourra, par exemple, se reporter à l'état de l'art de [Gloess 90].

Soit D un ensemble, $\wp(D)$ représente l'ensemble des parties de D. Une séquence, notée $(d_1,...,d_n)$ de n éléments de D est appelé un n-uplet. D^n représente l'ensemble des n-uplets d'éléments de D.

Une relation (binaire) r définie sur D est un sous-ensemble de D^2 (également notée D×D). d r d' signifie que d est en relation avec d' par r. Une fonction définie de D vers D est une relation définie sur D telle que tout élément d de D est en relation avec au plus un élément d' de D. Une fonction est dite partielle lorsque certains éléments de D ne sont pas en relation avec d'autres éléments de D, est dite totale sinon. Une fonction totale est usuellement appelée une application.

Soit D un ensemble et soit r une relation définie sur D. A partir de r, on définit de façon classique les relations suivantes :

- r^0 désigne la relation $\{(d,d) : d \in D\}$.
- r^n désigne la relation $r^{n-1} \circ r = r \circ r^{n-1}$ avec $r \circ r' = \{(d,d'') \text{ tel que } \exists\, d' \in D \mid (d,d') \in r \text{ et } (d',d'') \in r'\}$.
- r^+ désigne la cloture transitive de r, i.e., $r^+ = \bigcup_{i \geq 0} r^i$.
- r^* désigne la cloture réflexive et transitive de r, i.e., $r^* = \bigcup_{i \geq 0} r^i$.

Un préordre est une relation qui est réflexive et transitive. Une relation d'ordre est un préordre qui est antisymétrique. Une relation d'ordre définie sur un ensemble D est dite totale lorsque deux éléments quelconques de D sont comparables, est dite partielle sinon. Une relation d'équivalence est une relation qui est réflexive, symétrique et transitive. Tout préordre $\leq$ défini sur un ensemble D engendre une relation d'équivalence $\approx$ défini sur D comme suit : d $\approx$

d' ssi d ≤ d' et d' ≤ d. La classe d'équivalence d'un élément d de D désigne l'ensemble des éléments de D équivalents à d.

Définition A.7

Un ensemble partiellement ordonné (D,≤) est un ensemble D muni d'une relation d'ordre partielle ≤.

Soit (D,≤) un ensemble partiellement ordonné et soit S un sous-ensemble de D. S est une chaîne de D ssi pour tout couple d'éléments (d,d') de S, on a d ≤ d' ou d' ≤ d. D vérifie la condition de chaîne ascendante (resp. descendante) ssi toute chaîne croissante (resp. décroissante) de D est finie. On dit également que D est une relation d'ordre bien fondée sur D lorsque D vérifie la condition de chaîne descendante. Un élément d de D est un majorant (resp. minorant) de S ssi pour tout élément d' de S, on a : d' ≤ d (resp. d ≤ d'). Un élément d de D est le plus petit majorant (ou la borne supérieure) de S ssi d est un majorant de S et pour tout majorant d' de S, on a : d ≤ d'. Un élément d de D est le plus grand minorant (ou la borne inférieure) de S ssi d est un minorant de S et pour tout minorant d' de S, on a : d' ≤ d.

Définition A.8

Un cpo est un ensemble partiellement ordonné (D,≤) qui est complet, i.e., tel que toute chaîne S de D admet un plus petit majorant.

Un cpo (complete partial ordering) est un ensemble partiellement ordonné complet. L'opérateur qui pour tout sous-ensemble S de D associe (lorsqu'il existe) le plus petit majorant de S est appelé l'opérateur de borne supérieure et est généralement notée ∨. L'opérateur qui pour tout sous-ensemble S de D associe (lorsqu'il existe) le plus grand minorant de S est appelé l'opérateur de borne inférieure et est généralement notée ∧. Un élément de D, généralement noté ⊥, est appelé le plus petit élément de D ssi pour tout élément d de D, on a : ⊥ ≤ d. Un élément de D, généralement noté ⊤, est appelé le plus grand élément de D ssi pour tout élément d de D, on a : d ≤ ⊤. Tout cpo possède un plus petit élément.

Définition A.9

Un treillis est un ensemble partiellement ordonné (D,≤) tel que tout couple d'éléments (d,d') de D admet un plus petit majorant et un plus grand minorant.

Un treillis (D,≤) est complet ssi toute partie (éventuellement vide et éventuellement infinie) S de D admet un plus petit majorant et un plus grand minorant. Tout treillis complet est un cpo.

Soient D^{Δ} et D^{∇} deux ensembles, une application f définie de D^{Δ} vers D^{∇} est étendue en une application f* définie de $\wp(D^{\Delta})$ vers $\wp(D^{\nabla})$ par : $\forall S \in \wp(D^{\Delta})$, $f^*(S) = \{f(d) \mid d \in S\}$. Soient $(D^{\Delta},\leq^{\Delta})$ et $(D^{\nabla},\leq^{\nabla})$ deux ensembles partiellement ordonnés. Si f et g sont deux applications définies de $(D^{\Delta},\leq^{\Delta})$ sur $(D^{\nabla},\leq^{\nabla})$, on note $f \leq^{\nabla} g$ ssi pour tout d de D^{Δ}, on a : $f(d) \leq^{\nabla} g(d)$. Une application f définie de $(D^{\Delta},\leq^{\Delta})$ sur $(D^{\nabla},\leq^{\nabla})$ est monotone ssi : $\forall d,d' \in D^{\Delta}$, $d \leq^{\Delta} d' \Rightarrow f(d) \leq^{\nabla} f(d')$. Une application f définie de $(D^{\Delta},\leq^{\Delta})$ sur $(D^{\nabla},\leq^{\nabla})$ est continue ssi pour toute chaîne S (non vide) de D^{Δ} telle que S possède une borne supérieure notée $\vee S$, l'ensemble $f^*(S)$ possède une borne supérieure notée $\vee f^*(S)$ et $f(\vee S) = \vee f^*(S)$. Une application monotone est un morphisme d'ordre et une application continue est un morphisme de l'opérateur de borne supérieure. La continuité implique la monotonie.

Soit $(D,\leq)$ un ensemble partiellement ordonné. On note $I\delta$ l'application (identité) qui pour tout élément d de D associe d, i.e., $d = I\delta(d)$. Un point fixe pour une application f définie de $(D,\leq)$ sur $(D,\leq)$ est un élément d de D tel que $d = f(d)$. Lorsqu'il existe, le plus petit point fixe pour f est un élément d de D tel que pour tout point fixe d' de f, on a : $d \leq d'$.

Propriété A.10

Si $(D,\leq)$ est un cpo et si f est une application monotone définie de $(D,\leq)$ sur $(D,\leq)$, alors f possède un plus petit point fixe, noté lfp f.

Ce plus petit point fixe de f peut être calculé par itération à partir du plus petit élément, noté $\perp$, de D. On note respectivement Ord, Ord^{suc} et Ord^{lim} la classe des ordinaux, la classe des ordinaux successeurs et la classe des ordinaux limites. Le calcul de point fixe s'effectue comme suit (sachant que $\vee$ désigne l'opérateur de borne supérieure) :

$$f^0(\perp) = \perp$$

$$f^{n+1}(\perp) = f(f^n(\perp)),\ \forall\ n+1 \in Ord^{suc}$$

$$f^{\lambda}(\perp) = \bigvee_{\beta<\lambda} f^{\beta}(\perp),\ \forall\ \lambda \in Ord^{lim}$$

On note ω le premier ordinal limite. Lorsque f est continue, on sait que :

$$lfp\ f = f^{\omega}(\perp).$$

3 Connexions de Galois

Les connexions de Galois sont très utilisées dans le cadre de l'interprétation abstraite. Nous présentons ici quelques unes des propriétés rappelées par [Melton et al. 86] et [Cousot et Cousot 92a].

Définition A.11

Soient $(D^{\Delta},\leq^{\Delta})$ et $(D^{\nabla},\leq^{\nabla})$ deux ensembles partiellement pré-ordonnés, une connexion de Galois est la donnée de deux applications :

$\alpha : D^{\Delta} \rightarrow D^{\nabla}$

$\gamma : D^{\nabla} \rightarrow D^{\Delta}$

telles que :

$\forall\, d^{\Delta} \in D^{\Delta}, \forall\, d^{\nabla} \in D^{\nabla}, \alpha(d^{\Delta}) \leq^{\nabla} d^{\nabla}$ ssi $d^{\Delta} \leq^{\Delta} \gamma(d^{\nabla})$

En fait, les connexions de Galois sont généralement étudiées par rapport à des ensembles partiellement ordonnés. Les propriétés suivantes s'inscrivent dans ce sens.

Propriété A.12

Si (α,γ) est une connexion de Galois définie entre deux ensembles partiellement ordonnés $(D^{\Delta},\leq^{\Delta})$ et $(D^{\nabla},\leq^{\nabla})$ alors

$\forall\, d^{\Delta} \in D^{\Delta}, \forall\, d^{\nabla} \in D^{\nabla}, \alpha(d^{\Delta}) \leq^{\nabla} d^{\nabla}$ ssi $d^{\Delta} \leq^{\Delta} \gamma(d^{\nabla})$

est équivalent à :

α est monotone

γ est monotone

$I\delta \leq^{\Delta} \gamma o \alpha$ ($\gamma o \alpha$ est extensif)

$\alpha o \gamma \leq^{\nabla} I\delta$ ($\alpha o \gamma$ est réductif)

Propriété A.13

Si (α,γ) est une connexion de Galois définie entre deux ensembles partiellement ordonnés $(D^{\Delta},\leq^{\Delta})$ et $(D^{\nabla},\leq^{\nabla})$ alors

$\alpha o \gamma o \alpha = \alpha$

$\gamma o \alpha o \gamma = \gamma$

Propriété A.14

Si (α,γ) est une connexion de Galois définie entre deux ensembles partiellement ordonnés $(D^{\Delta},\leq^{\Delta})$ et $(D^{\nabla},\leq^{\nabla})$ alors

$\forall\, d^{\Delta} \in D^{\Delta}, \alpha(d^{\Delta}) = \wedge^{\nabla} \{ d^{\nabla} \mid d^{\Delta} \leq^{\Delta} \gamma(d^{\nabla}) \}$

$\forall\, d^{\nabla} \in D^{\nabla}, \gamma(d^{\nabla}) = \vee^{\Delta} \{ d^{\Delta} \mid \alpha(d^{\Delta}) \leq^{\nabla} d^{\nabla} \}$

où $\wedge^{\nabla}$ et $\vee^{\Delta}$ représentent respectivement l'opérateur de borne inférieure sur $(D^{\nabla},\leq^{\nabla})$ et l'opérateur de borne supérieure sur $(D^{\Delta},\leq^{\Delta})$.

Une insertion de Galois est une connexion de Galois particulière. On retrouve très souvent cette caractérisation dans la littérature concernant l'interprétation abstraite.

Définition A.15

Si (α,γ) est une connexion de Galois alors (α,γ) est appelée une insertion de Galois ssi α est surjective.

Propriété A.16

Si (α,γ) est une connexion de Galois définie entre deux ensembles partiellement ordonnés $(D^{\Delta},\leq^{\Delta})$ et $(D^{\nabla},\leq^{\nabla})$ alors α est surjective ssi $\alpha \circ \gamma = I\delta$.

Annexe B

Implémentation

Nous avons implémenté une version limitée, appelée la_oldt, du système d'inférence de types que nous proposons au chapitre 8. Les contraintes ensemblistes sont codées sous la forme de chemins au dépliage limité ou lul pour "limited unfolding loop" [Lecoutre et al. 92a,92b] Intuitivement, un chemin au dépliage limité définit une structure récursive finie. Par exemple, le but

```
:- add(X,Y,Z) with X:s(1)
```

correspond à

```
:- X=s(X)∪ex ◊ add(X,Y,Z).
```

où ex correspond à l'union des termes de la forme f(⊤,...,⊤) pour tout symbole de fonction f différent de s. La contrainte X=s(X)∪ex est donc moins contraignante que la contrainte X=s(X)∪0.

Nous donnons ci-dessous un exemple de programme écrit pour la_oldt :

```
paths={s(1)}.

add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mul(0,X,0).
mul(s(X),Y,Z) :- mul(X,Y,I), add(Y,I,Z).

exp(s(N),0,0).
exp(0,s(X),s(0)).
exp(s(N),s(X),Z) :- exp(N,s(X),I), mul(I,s(X),Z).
```

La première ligne est une directive qui indique quels sont les luls à prendre en compte pour l'analyse. La figure B.1 présente le graphe OLDT obtenu par la résolution AOLDT à partir du programme précédent et du but indiqué sur la figure. Pour avoir de plus amples informations sur les noeuds ou les substitutions, il suffit de cliquer sur les rectangles correspondants. La figure B.2 propose la table des solutions. Il n'y a en fait qu'une seule entrée puisqu'un seul noeud actif. Des variables anonymes (de la forme _n) sont utilisées. Elles permettent d'indiquer ici le type du second et du troisième argument. La figure B.3 représente un autre graphe OLDT obtenu à partir d'un but différent. Si on lance la procédure de vérification pour la terminaison et la procédure de vérification pour le déterminisme sur cet exemple, on obtient une réponse positive. Il est à noter que des graphes beaucoup plus complexes peuvent être obtenus en faisant varier divers paramètres.

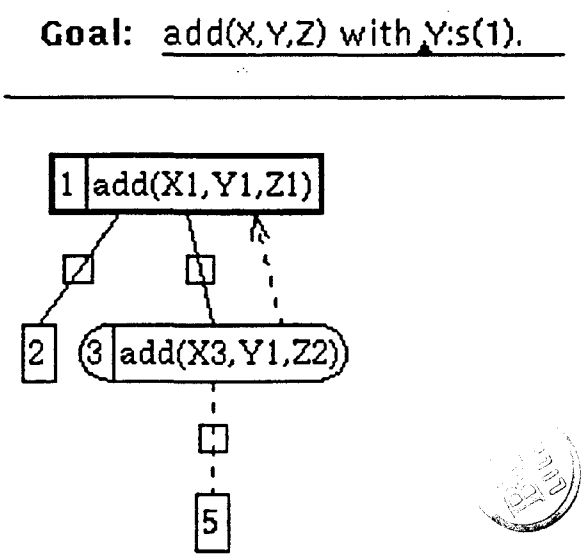

Figure B.1

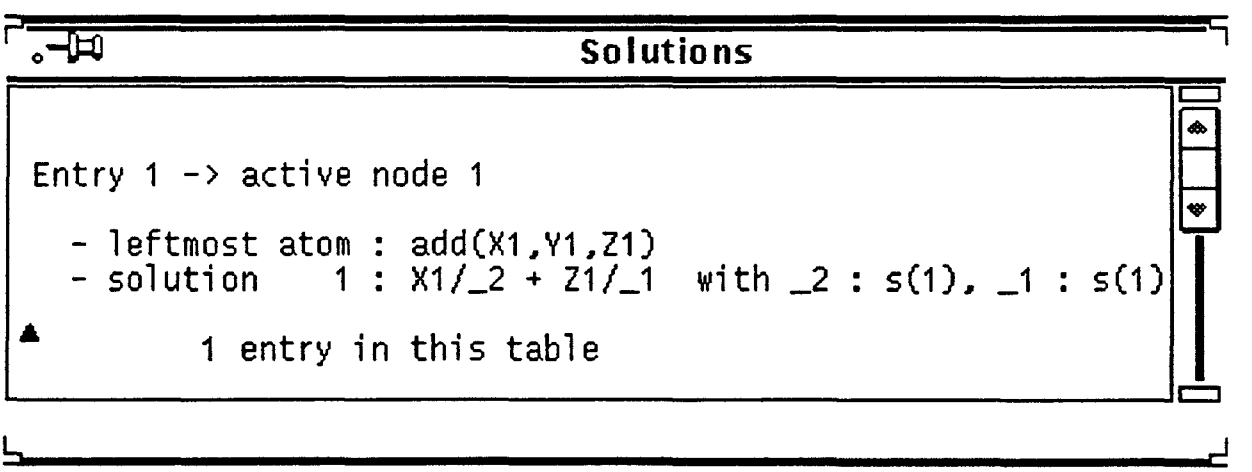

Figure B.2

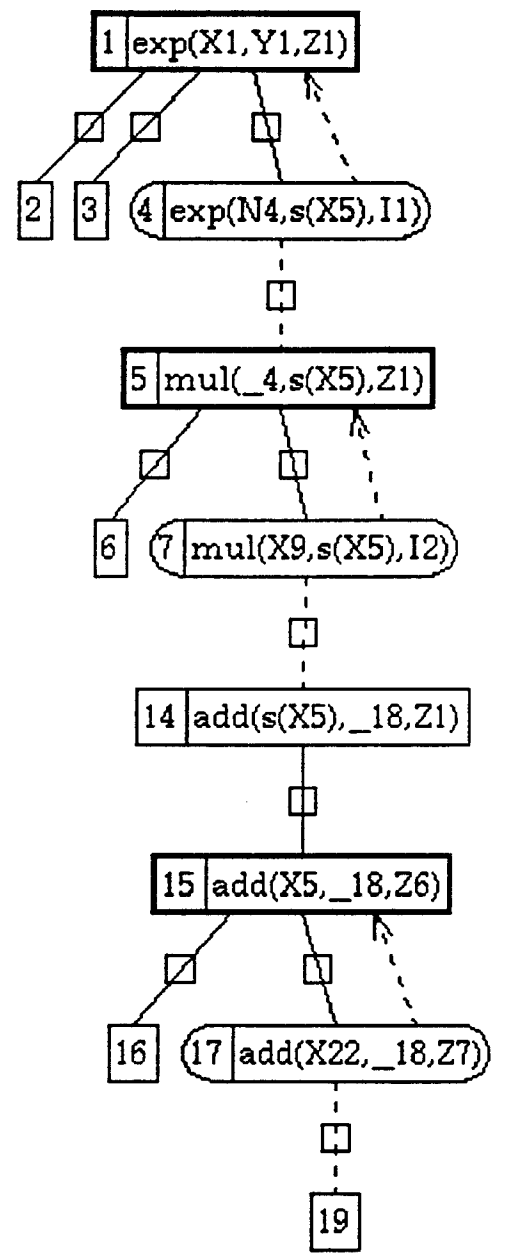
Goal: exp(X,Y,Z) with X:s(1), Y:s(1),
1 exp(X1,Y1,Z1)
2
3
4 exp(N4,s(X5),I1)
5 mul(_4,s(X5),Z1)
6
7 mul(X9,s(X5),I2)
14 add(s(X5),_18,Z1)
15 add(X5,_18,Z6)
16
17 add(X22,_18,Z7)
19

Figure B.3

Notations

$\mathbb{B}$: ensemble des booléens, $\mathbb{B}$={faux,vrai}
$\mathbb{N}$: ensemble des entiers naturels
$\mathbb{R}$: ensemble des réels

card E : cardinal de l'ensemble E max E : plus grand élément de l'ensemble E
$\wp$(E) : ensemble des parties de E
$\bot$ et $\top$: plus petit et plus grand élément d'un ensemble E
glb(x,y) : plus grand minorant (borne inférieure) des éléments x et y
lub(x,y) : plus petit majorant (borne supérieure) des éléments x et y

$I\delta$: application identité
lfp f : plus petit point fixe de l'application f
ω : premier ordinal limite

$\mathcal{V}$: ensemble de symboles de variables ; X,Y,... élements de $\mathcal{V}$
$\mathcal{V}_{sc}$: ensemble de symboles de variables ensemblistes ; $\mathcal{X},\mathcal{Y}$,... élements de $\mathcal{V}_{sc}$
$\mathcal{F}$: ensemble de symboles de fonctions ; f,g,...a,b,... élements de $\mathcal{F}$
$\mathcal{P}$: ensemble de symboles de prédicats ; p,q,... élements de $\mathcal{P}$
$\mathcal{C}$: ensemble de symboles de contraintes
S : ensemble de sortes ; s, s_1,s_2,... éléments de S

Σ : signature, $\Sigma = (S,\mathcal{F})$ ou $\Sigma = (S,\mathcal{F},\mathcal{P})$
$\mathcal{L}$: langage de contraintes (ensemble de formules) ; c,d,... éléments de $\mathcal{L}$
$\mathcal{A}$: Σ-algèbre
$\mathcal{CL}$: CLP-langage, $\mathcal{CL} = (\Sigma,\mathcal{L},\mathcal{A})$
$\vdash$ relation d'ordre définie sur $\mathcal{L}$ par $\mathcal{A}$, pour un CLP-langage donné $\mathcal{CL} = (\Sigma,\mathcal{L},\mathcal{A})$
$\vdash_T$ relation $\vdash$ modulo le nom des variables

$\mathcal{Term}(S,\mathcal{V},\mathcal{F})$: algèbre des termes libres ; si card S = 1 alors $\mathcal{Term}(\mathcal{V},\mathcal{F})$
$\mathcal{Term}(S,\mathcal{F})$: algèbre des termes clos (sans variables) ; si card S = 1 alors $\mathcal{Term}(\mathcal{F})$
t,s,... éléments de $\mathcal{Term}(\mathcal{V},\mathcal{F})$ et $\mathcal{Term}(S,\mathcal{V},\mathcal{F})$

$\mathcal{Atom}(S,\mathcal{V},\mathcal{F},\mathcal{P})$: ensemble des atomes libres ; si card S = 1 alors $\mathcal{Atom}(\mathcal{V},\mathcal{F},\mathcal{P})$
$\mathcal{Atom}(S,\mathcal{F},\mathcal{P})$: ensemble des atomes clos ; si card S = 1 alors $\mathcal{Atom}(\mathcal{F},\mathcal{P})$
a,b,h,... éléments de $\mathcal{Atom}(S,\mathcal{F},\mathcal{P})$ et $\mathcal{Atom}(S,\mathcal{V},\mathcal{F},\mathcal{P})$

$\mathcal{H}$: ensemble des atomes homogènes (i.e., construits à partir de $\mathcal{V}$ et $\mathcal{P}$)
$\Diamond$: séparateur entre éléments de $\mathcal{L}$ et éléments de $\mathcal{H}$
$\downarrow$: opérateur de projection définie sur $\mathcal{L} \times \wp(\mathcal{V})$

$\sigma,\beta,...$ substitutions
$\mathcal{V}al$: ensemble des assignations de variables ; $\theta,\theta',...$ éléments de $\mathcal{V}al$
$\mathcal{R}en$: ensemble des substitutions de renommage ; $\rho,\rho',...$ éléments de $\mathcal{R}en$

fs : fonction de sélection sp : stratégie de parcours
bf ("breadth-first"), df ("depth-first"),... stratégies de parcours

Var(o) : ensemble des variables de l'objet o
$Var_{bound}(o)$ et $Var_{free}(o)$: ensemble des variables liées et libres de l'objet o
$Var_{\exists}(o)$ et $Var_{\forall}(o)$: ensemble des variables liées par $\exists$ et par $\forall$ de l'objet o

$^{\vartriangle}$ et $^{\triangledown}$: symboles utilisés pour faire référence à un objet concret et abstrait

ξ : relation d'approximation
α : fonction d'abstraction γ : fonction de concrétisation
(α,γ) : connection de Galois

∇ : opérateur de widening Δ : opérateur de narrowing

T : arbre OLD ou arbre OLDT F et G : forêt et graphe OLDT
AT et PT : tables des noeuds actifs et des noeuds passifs
$\mathcal{S}tr$: structure OLDT ; $\mathcal{S}tr$ = (AT,PT,F)

i : racine ou entrée d'un arbre, d'une forêt ou d'un graphe OLDT
$v,w,...$ noeuds d'un arbre, d'une forêt ou d'un graphe OLDT
pos(v) : position du pointeur associé à un noeud passif v
list(v) : liste des solutions associées à un noeud actif v

ex : expression ensembliste
$\bot_{sc}$ et $\top_{sc}$: plus petit et plus grand élément ensembliste
$f^{-1}_{(i)}$: opérateur ensembliste de projection
* : opérateur de cloture distributive

$\mathcal{S}$: système d'équations (sur les arbres finis ou les ensembles)
res($\mathcal{S}$) : forme résolue d'un système d'équations
sol($\mathcal{S}$) : solution du système $\mathcal{S}$
$\mathcal{ST}$: systèmes d'équations sur les termes
$\mathcal{SS}$: systèmes d'équations ensemblistes
$\mathcal{S}_X$ ou $\mathcal{SS}_X$: système élémentaire associé à la variable ensembliste X

Index